On I/O Performance and Cost Efficiency of Cloud Storage: A Client's Perspective

Binbing Hou
Louisiana State University and Agricultural and Mechanical College

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_dissertations

Part of the Computer and Systems Architecture Commons, and the Data Storage Systems Commons

Recommended Citation
https://digitalcommons.lsu.edu/gradschool_dissertations/5090
ON I/O PERFORMANCE AND COST EFFICIENCY OF CLOUD STORAGE: A client's perspective

A Dissertation

Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in
The Department of Computer Science and Engineering

by
Binbing Hou
B.S., Wuhan University of Technology, 2011
M.S., Huazhong University of Science and Technology, 2014
December 2019
Acknowledgments

At the end of the doctorate program, I would like to express my deepest gratitude to many people. It is their great help and support that make this dissertation possible.

First of all, I would like to thank my advisor, Dr. Feng Chen, for his continuous support. He gave me numerous valuable guidances throughout the process of my research studies, from selecting topics to designing experiments, from writing papers to practicing presentations. In addition to the technical guidance, his enthusiastic and rigorous attitude towards research has also deeply impressed me and inspired me. It is my fortune to have Dr. Feng Chen in my journey of pursuing the Ph.D.

I would also like to thank the other committee members, Dr. Bijaya B. Karki, Dr. Qingyang Wang, and Dr. Tao Jin, for their services and insightful comments and feedback, and many other professors, such as Dr. Jianhua Chen, Dr. Jian Zhang, Dr. Konstantin Busch, Dr. Rahul Shah, Dr. Evangelos Triantaphyllou, Dr. Gerald Baumgartner, and Dr. Steve Brandt, for their wonderful courses.

I would also like to thank my labmates, Jace Courville, Yichen Jia, Jian Liu, and Kefei Wang. They gave me many constructive comments on my research work and helped me a lot in my daily life. I will remember the time we enjoyed together.

Finally, I would like to express my heartfelt thanks to my family for their endless support and selfless love. Their happiness is the meaning of all my work and my whole life.

This work was supported in part by the Louisiana Board of Regents under grants LEQSF(2014-17)-RD-A01 and LEQSF-EPS(2015)-PFUND-391, the National Science Foundation under grants CCF-1453705 and CCF-1629291, and a grant from Intel Corporation.
# Table of Contents

ACKNOWLEDGMENTS ........................................... ii

LIST OF TABLES ............................................ v

LIST OF FIGURES ........................................... vi

ABSTRACT .................................................. 1

CHAPTER
  1 INTRODUCTION .......................................... 2
    1.1 Dissertation Statement and Contributions ............. 4
    1.2 Dissertation Organization .......................... 5
  2 BACKGROUND ........................................... 7
    2.1 Cloud Storage Model ................................ 7
    2.2 Cloud Storage Services ............................. 7
    2.3 Cloud Storage Applications ......................... 8
    2.4 Cloud Storage Pricing Policies ..................... 9
  3 UNDERSTANDING I/O PERFORMANCE BEHAVIORS 
     OF CLOUD STORAGE FROM A CLIENT’S PERSPECTIVE .... 11
    3.1 Introduction ....................................... 11
    3.2 Measurement Methodology ............................ 14
    3.3 Basic Observations .................................. 19
    3.4 Effects of Client Capabilities ...................... 27
    3.5 Effects of Geographical Distance .................... 32
    3.6 Interferences of Mixed Requests .................... 35
    3.7 System Implications ................................ 39
    3.8 A Pilot Solution: Sampling and Inference .......... 42
    3.9 Case Studies on Cloud-based Applications .......... 47
    3.10 Related Work ...................................... 59
    3.11 Conclusion ........................................ 61
  4 PACACA: A PARALLELISM- AND CORRELATION-AWARE 
     CLIENT CACHE FRAMEWORK .............................. 62
    4.1 Introduction ....................................... 62
    4.2 Frequent Cluster Mining ............................ 65
    4.3 Parallelized Prefetching ............................ 70
    4.4 Parallelization-aware Caching ...................... 72
    4.5 Putting It All Together ............................ 76
    4.6 Evaluation ......................................... 78
    4.7 Discussion ......................................... 90
    4.8 Related Work ....................................... 91
    4.9 Conclusion ......................................... 93
5 GDS-LC: A LATENCY- AND COST-AWARE CLIENT CACHING
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Introduction</td>
<td>94</td>
</tr>
<tr>
<td>5.2 Caching Issues</td>
<td>97</td>
</tr>
<tr>
<td>5.3 Scheme Design</td>
<td>99</td>
</tr>
<tr>
<td>5.4 Evaluation</td>
<td>106</td>
</tr>
<tr>
<td>5.5 Related Work</td>
<td>132</td>
</tr>
<tr>
<td>5.6 Conclusion</td>
<td>135</td>
</tr>
</tbody>
</table>

6 LIMITATIONS AND FUTURE WORK      | 136  |

7 CONCLUSIONS                      | 139  |

REFERENCES                          | 140  |

APPENDIX
A THE ACM PUBLICATION AGREEMENT     | 153  |
B THE IEEE PUBLICATION AGREEMENT    | 154  |

VITA                                | 155  |
## List of Tables

3.1 Configurations of Amazon EC2-based Clients ............................ 15
3.2 Magnetic vs. SSD on Amazon EC2 ........................................... 16
3.3 Object-based Workloads ...................................................... 17
3.4 Peak Upload Bandwidth Comparison (*Baseline* vs. *MEM-minus*) ........ 31
3.5 Peak Download Bandwidth Comparison (*Baseline* vs. *MEM-minus*) ..... 31
3.6 Mixed vs. Non-mixed Upload/Download Bandwidths (4MB) .................. 36
3.7 Average Latency Reduction by Prefetching .................................. 57
4.1 An Illustrative Example of Pacaca’s Caching Scheme ....................... 73
4.2 Access Costs of the Objects/Clusters ....................................... 73
4.3 Details of the Training Traces and Mining Results ....................... 80
4.4 Details of the Testing Traces ................................................ 80
5.1 Trace Characteristics ...................................................... 107
5.2 The Pricing Model of Amazon S3 Services .................................. 109
# List of Figures

2.1 Client-side Caching for Cloud Storage ................................................. 9
3.1 Upload and Download Bandwidths on *Baseline* .................................. 20
3.2 Upload Latencies on *Baseline* ............................................................ 21
3.3 Download Latencies on *Baseline* ....................................................... 21
3.4 CDFs of Upload Latencies with Different Parallelism Degrees on *Baseline* . 22
3.5 CDFs of Download Latencies with Different Parallelism Degrees on *Baseline* . 22
3.6 CDFs of Download Latencies with Different Parallelism Degrees on *Baseline* with *ramfs* ................................................................. 23
3.7 Upload Bandwidths of Different Combinations on *Baseline* .................. 26
3.8 Peak Bandwidth Comparison (*Baseline* vs. *CPU-plus*) ......................... 28
3.9 Peak Bandwidth Comparison (*Baseline* vs. *STOR-ssd*) ......................... 29
3.10 Average 16MB Download Latency Comparison (*Baseline* vs. *STOR-ssd*) ................................................................. 30
3.11 Peak Bandwidth Comparison (*Baseline* vs. *GEO-Sydney*) ..................... 33
3.12 Average 16MB Upload Latency Comparison (*Baseline* vs. *GEO-Sydney*) ................................................................. 34
3.13 Bandwidths of Mixed Upload/Download (4KB) ....................................... 36
3.14 Bandwidths of Mixed 4KB/4MB Upload ............................................... 37
3.15 Latencies of Mixed 4KB/4MB Upload ................................................. 38
3.16 Sampling for Uploading a 4MB Object .............................................. 43
3.17 Combinations for Uploading a 256KB Object ......................................... 44
3.18 An Example of Selecting a Proper Parallelism Degree ............................ 45
3.19 Evaluation of an Inferred Combination 256KB×21 ................................ 46
3.20 Execution Time of *grep* with Different Parallelism Degrees .................. 48
3.21 Execution Time of `tar` (Trace 1) ........................................ 50
3.22 Execution Time of `tar` (Trace 2) ........................................ 50
3.23 CDF of File Sizes ........................................................... 52
3.24 Download Bandwidths of Different Chunk Sizes with Single Thread .......... 54
3.25 Average Latencies with Different Chunking ................................ 55
3.26 Hit Ratios with Different Chunking ....................................... 55
3.27 Average Download Latencies with Different Parallelism Degrees .......... 56
3.28 Average Read Latency Comparison ....................................... 57
3.29 Average Write Latency Comparison ....................................... 58
4.1 An illustration of the Look-around Circle .................................. 67
4.2 Cache Replacement Algorithm ............................................. 75
4.3 An Illustration of the Architecture of Pacaca .............................. 76
4.4 Integration of Caching and Prefetching ................................... 77
4.5 Performance for Calgary .................................................... 82
4.6 Performance for NASA ..................................................... 83
4.7 Performance for Deasna2 ................................................... 83
4.8 Performance for Home02 ................................................... 83
4.9 Pacaca Performance for Calgary with Different Settings of `search_limit` .... 85
4.10 Pacaca Performance for Calgary with Different Settings of `min_confidence` .... 86
4.11 Pacaca Performance for Calgary with Different Settings of `min_support` .... 87
4.12 Pacaca Performance for Calgary w/o Mis-prefetching Detection ............ 88
4.13 Average Latency vs. Radius ............................................... 88
4.14 Miss Ratio and Download Time vs. Radius ................................ 89
5.1 The Two-region Structure for Caching ................................... 99
5.2 Algorithm of Migrating Objects between Regions .......................... 101
5.3 CDFs of Object Sizes ................................................................. 107
5.4 Local Cloud: Experimental Results with Different Traces. ............... 113
5.5 Local Cloud: An Example to Investigate the Caching Behaviors of Different Caching Schemes. ................................................. 114
5.6 Local Cloud: The Number of Uploadings and Detailed Monetary Cost of Netfs with Different Caching Schemes. ............................. 116
5.7 Internet Cloud: Experimental Results with Different Traces. ............... 117
5.8 Internet Cloud: Detailed Monetary Cost of Different Traces ................. 118
5.9 Heterogeneous Cloud: Experimental Results with Different Traces .... 121
5.10 Internet Cloud: Effects of Different Size Ratios ............................... 123
5.11 Internet Cloud: Comparisons of GDS-LC and GDS-L ..................... 124
5.12 Internet Cloud: Comparisons of GDS-LCF and GDS-LF. ................. 124
5.13 Local Cloud: Observed Variances of Average Latencies ................... 125
5.14 Internet Cloud: Observed Variances of Average Latencies ................. 126
5.15 Heterogeneous Cloud: Observed Variances of Average Latencies .......... 127
5.16 Internet Cloud: Comparisons of GDS-LCF, GDSF(latency), and GDSF(price) 128
5.17 Local Cloud: Effects of Different Normalization Factors ..................... 129
5.18 Internet Cloud: Effects of Different Normalization Factors .................. 130
5.19 Heterogeneous Cloud: Effects of Different Normalization Factors ....... 130
Abstract

Cloud storage has gained increasing popularity in the past few years. In cloud storage, data are stored in the service provider’s data centers; users access data via the network and pay the fees based on the service usage. For such a new storage model, our prior wisdom and optimization schemes on conventional storage may not remain valid nor applicable to the emerging cloud storage.

In this dissertation, we focus on understanding and optimizing the I/O performance and cost efficiency of cloud storage from a client’s perspective. We first conduct a comprehensive study to gain insight into the I/O performance behaviors of cloud storage from the client side. Through extensive experiments, we have obtained several critical findings and useful implications for system optimization. We then design a client cache framework, called Pacaca, to further improve end-to-end performance of cloud storage. Pacaca seamlessly integrates parallelized prefetching and cost-aware caching by utilizing the parallelism potential and object correlations of cloud storage. In addition to improving system performance, we have also made efforts to reduce the monetary cost of using cloud storage services by proposing a latency- and cost-aware client caching scheme, called GDS-LC, which can achieve two optimization goals for using cloud storage services: low access latency and low monetary cost. Our experimental results show that our proposed client-side solutions significantly outperform traditional methods. Our study contributes to inspiring the community to reconsider system optimization methods in the cloud environment, especially for the purpose of integrating cloud storage into the current storage stack as a primary storage layer.
Chapter 1
Introduction

Cloud storage has been increasingly popular in recent years, and its market is quickly growing. According to a report from Statista, personal cloud storage users will increase from 1.1 billion in 2015 to 2.3 billion by 2020 [132]. The global market is predicted to grow from $18.87 billion in 2015 to $65.41 billion by 2020 [106]. In addition to archiving personal data, cloud storage also plays an indispensable role in various core commercial services, such as serving videos on demand and storing unstructured scientific data.

Cloud storage presents a compelling new storage model to end users. In this model, data are stored in the service provider’s data centers, and users access data through an HTTP-based REST protocol and pay the fees based on actual service usage. By separating data storage from data consumers, this model provides a high degree of flexibility, elasticity, and cross-platform capability. From the perspective of system organization and data access, such a model is dramatically different from that of 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 “I/O protocol” is an HTTP-based REST protocol, in which a variable-sized object is a basic unit for data access; the “host” is not a single computing entity any more but could be any kind of computing devices (e.g., PCs or smartphones). All these properties make cloud storage services fundamentally different from its conventional counterpart.

Besides these technical differences, the pricing policies of cloud storage services are also unique. For using conventional storage, the expense includes one-time purchase and setup fees, and additional fees used for system maintenance and management, such as machine room rental fees, energy consumption and network connection fees, and labor fees. By contrast, the pricing policies of cloud storage services are generally usage-based, which means that the consumers pay the fee based on the actually used storage space, the
number and the type of I/O requests, and the network traffic caused by the I/O requests. In other words, users’ I/O activities, which play a marginal role in determining the cost for using conventional storage, can directly and dramatically affect the fee for using cloud storage.

A direct impact of such differences is that much of our prior understanding and optimization schemes for conventional storage may not continue to be valid or applicable to the emerging cloud-based storage.

First, the massively parallelized storage cluster, where data are 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 [53, 90]. 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. Third, the clients, which consume the data and drive the I/O activities, are highly diverse in all aspects, from CPU, memory, storage, to communication. Finally, the I/O traffic generated by the clients is not a significant factor affecting the fee for using conventional storage, but can directly determine the fee for using cloud storage services.

Unfortunately, our current understanding of storage behaviors and the optimization schemes are mostly confined to the conventional storage, which are 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 [98, 49, 35] and cloud storage clients [49, 110, 56, 55, 57, 69, 118]. These studies mostly focused on either benchmarking cloud storage on the server side or testing specific cloud storage client applications; however, they failed to consider client-related
factors (e.g., client capabilities). Some prior work has made an attempt to improve the performance of cloud storage with classic optimization schemes, such as LRU-based caching schemes [140, 54, 125] and sequentiality-directed prefetching schemes [33]; however, these schemes are designed for conventional storage (e.g., HDDs) without considering the unique characteristics of cloud storage (e.g., parallelism), thus are sub-optimal for cloud storage. More importantly, conventional storage optimization schemes (e.g., caching and prefetching) mostly do not consider the fee caused by I/O activities of the clients, making these schemes not suitable in the scenario of cloud storage, especially for I/O intensive workloads.

1.1 Dissertation Statement and Contributions

To understand the I/O performance behaviors of cloud storage and design optimization schemes for cloud storage based on its unique characteristics, we study cloud storage services from a client’s perspective, and our work includes system measurement and scheme design and implementation. The contributions of this dissertation can be summarized as follows:

- We first consider the cloud storage service as a “black box” and aim to observe and analyze the end-to-end performance behaviors from the perspective of data consumers. In specific, we take Amazon Simple Storage Services (S3) as the cloud storage provider and conduct a series of experiments with different client settings to study the effect of a client’s capabilities and locations on end-to-end performance. Through extensive experiments and quantitative analysis, we have obtained several important and interesting findings related to the effects of I/O parallelization, client’s capabilities, geographical distance, and request interference. Based on our findings, we present a pilot solution and a set of case studies to showcase how to exploit the characteristics of cloud storage in terms of parallelism and request size. Our studies show that the end-to-end performance of cloud storage services can be significantly improved by sufficiently exploiting the capabilities of clients and the great performance potential of cloud storage services. These findings and results are also reported in our prior papers [79, 80].
• For improving system performance, we further revisit conventional optimization schemes, especially caching and prefetching schemes, in the scenario of cloud storage. Aiming to sufficiently utilize the unique characteristics of cloud storage for optimizing access latency, we propose a client cache management framework, called Pacaca, which integrates a parallelized prefetching scheme and a cost-aware caching scheme, based on the object correlations obtained from access history. With such a client cache framework, we can sufficiently exploit the great parallelism potential of cloud storage services and the correlations among objects in cloud storage systems. This work is also reported in our prior paper [78].

• In addition to improving system performance, we have also made efforts to reduce the monetary cost of using cloud storage services by proposing a latency- and cost-aware client caching scheme called GDS-LC. GDS-LC considers both access latency and monetary cost when deciding victim objects, aims to achieve two optimization goals: low access latency and low monetary cost. Our experimental results show that our proposed client-side solutions significantly outperform traditional schemes. This work is also reported in our prior paper [77].

All our efforts, together, aim to improve the I/O performance and cost efficiency from a client’s perspective, especially for the purpose of integrating cloud storage into the current storage stack as a primary storage layer to serve I/O intensive workloads.

1.2 Dissertation Organization

The rest of the dissertation is organized as follows. Chapter 2 introduces the background. Chapter 3 presents our measurement work on understanding cloud storage services from a client’s perspective, including the findings and observations, system implications, pilot solutions, and case studies. Chapter 4 presents our proposed client cache management framework that exploits the I/O parallelism and object correlations of cloud storage to enhance caching and prefetching schemes for reducing cloud access latencies. Chapter 5 presents our proposed client caching scheme to reduce both access latency and monetary
cost for using cloud storage services. Chapter 6 discusses the limitations of our work presented in this dissertation and our future work, and Chapter 7 concludes the dissertation.
Chapter 2
Background

In this chapter, we introduce the background of cloud storage, including the cloud storage model, services, applications, and pricing policies.

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. Objects are further organized into logical groups, which are generally called buckets or containers. 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).

Almost all cloud storage service providers offer an HTTP-based Representational State Transfer (REST) interface to users for accessing cloud storage objects. 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 [17] is one of the most typical and popular cloud storage services. Other cloud storage services, such as OpenStack Swift [117], share a similar structure. Typically, the cloud storage service is running on a large-scale storage cluster consisting of many servers for different purposes: proxy servers, which handle incoming HTTP requests; account servers, which manage the user accounts and bucket listing; bucket and object servers, which manage the listing and storage of ob-

\footnote{Parts of this chapter have been previously published as: Binbing Hou and Feng Chen, “GDS-LC: A Latency- and Cost-Aware Client Caching Scheme for Cloud Storage”, ACM Transactions on Storage, 13(4):40:1–40:33, 2017. DOI: 10.1145/3149374. © 2017 ACM. Reprinted with permission.}
jects. 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 filesystem-like interface to allow end users to access cloud storage. From the perspective of data exchange, these clients often use syncing or caching to improve client performance. 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. The syncing mode is adopted by almost all personal cloud storage applications, such as Dropbox [58], Google Drive [67], and OneDrive [113], to provide data archiving and sharing services for consumers.

With the caching approach, the client only maintains the most frequently used data in local, and any cache miss leads to on-demand data fetching from the cloud. Figure 2.1 illustrates the client-side caching for cloud storage. In practice, a client cache can be not only a local storage device but also a client-side gateway or proxy. 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 [54], S3FS [125], S3backer [124], BlueSky [140], and SCFS [32]. In general, all the above-mentioned applications essentially convert the POSIX-like file
operations into HTTP requests (e.g., a `read` function call is converted to a `GET` HTTP request).

In this dissertation, we first conduct experiments 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, prefetching, compression, and deduplication) but directly uses the HTTP protocol, which is the underlying communication protocol in cloud storage. The findings obtained from the system measurement provide design guidance for client-side solutions, such as caching and prefetching schemes.

### 2.4 Cloud Storage Pricing Policies

Different from conventional storage, the fee for using cloud storage is usage-based. This means that users only have to pay the fee for the actual usage. Personal and enterprise cloud storage services generally have different pricing policies.

For personal cloud storage services (e.g., Dropbox), users generally need to use the storage space for file sharing and backup, thus the fee is charged based on the requested storage capacity. For example, the standard pricing policy for Dropbox is $12.5 for 3 TB storage space per user per month (cited on September 13, 2019 from Dropbox’s website [59]). The fee may vary based on the quality levels of the services (e.g., standard, professional, and enterprise) [59].

For enterprise cloud storage services (e.g., Amazon S3), which provide storage services
for enterprise users to serve I/O intensive workloads. The specific pricing policies of the same cloud storage service provider may vary under different scenarios, and different cloud storage service providers generally have different pricing policies. These pricing policies are generally based on the storage capacity and I/O activities, including the volume of data transfer, the number of I/O requests, and request types.

We can take Amazon S3 standard storage in Ohio (US EAST) as an example (cited on September 13, 2019 from Amazon S3’s website [16]). For the storage fee, the price is $0.023, $0.022, and $0.021 per GB, for the actual usage of the first 50 TB, the next 450 TB, and the storage space over 500 TB per month, respectively. For the request fee, the fee for PUT requests is $0.005 per 1,000 requests, and for GET requests is $0.0004 per 1,000 requests. The data transfer fee is charged based on the data transfer direction (i.e., transfer out from the cloud or transfer to the cloud) and the accumulated data transfer per month. For example, transferring data from Amazon S3 to the Internet is free when the accumulated data transfer is up to 1 GB and as high as $0.09 per GB for the next 9.999 TB.

In this dissertation, we aim to improve the I/O performance and cost efficiency of cloud storage for serving I/O intensive workloads by providing client-side solutions (e.g., caching and prefetching). Therefore, our solutions focus more on enterprise cloud storage services used with the caching approach and aim to reduce the monetary cost caused by the I/O activities of the clients.
Chapter 3
Understanding I/O Performance Behaviors of Cloud Storage from a Client’s Perspective

In this chapter, we present our measurement work on investigating the I/O performance behaviors of cloud storage from a client’s perspective. Our work focuses on studying the effect of various client-related factors, including I/O parallelization, request size, client capability, and geographical distance. Our observations and findings bring new optimization opportunities for improving the end-to-end I/O performance of cloud storage.

3.1 Introduction

To understand the performance behaviors of cloud storage and propose client-side solutions, our work starts from conducting a comprehensive study based on system measurement. Different from prior measurement work that mostly focused on the server side of cloud storage services and specific applications, we consider the cloud storage service as a “black box” and aim to observe and analyze the end-to-end performance behaviors from the perspective of data consumers. From this perspective, the end-to-end cloud storage performance is a result of the interactions between the cloud and the client, and a unique challenge for cloud storage is the highly diverse working scenarios—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, depending on the network speeds and geographical locations.

This strongly motivates us to obtain key insight on the unique I/O behaviors of cloud

---

1 Parts of this chapter have been previously published as: Binbing Hou, Feng Chen, Zhonghong Ou, Ren Wang, and Michael Mesnier, “Understanding I/O Performance Behaviors of Cloud Storage from a Client’s Perspective”, in Proceedings of the 32nd International Conference on Massive Storage Systems and Technology (MSST’16), 2016. DOI: 10.1109/MSST.2016.7897089. © 2016 IEEE. Reprinted with permission.

storage, a storage solution for cloud, especially from the client side. Specifically, we attempt to investigate 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.

The main purpose of our experimental studies is to characterize the performance behaviors of cloud storage from the client’s perspective. Specifically, we attempt to answer a set of important questions listed below. Successfully answering these questions can not only help us understand 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 [42]. 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?

- 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 observed 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 [44].

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 system performance, and 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 end-to-end performance but does not always result in lower bandwidth, and appropriately parallelizing cloud I/Os can effectively hide the impact of long geo-distances. (4) The interference among mixed cloud I/Os may cause significant performance degradation.

Based on our findings, we present a series of pilot solutions and case studies to showcase how to exploit the characteristics of cloud storage in terms of parallelism and request size. We first showcase a sampling- and inference-based approach to determine a proper combination of parallelism degree and request size to achieve different optimization goals, and also present a set of case studies on client-side chunking and parallelization for typical cloud-based applications. Our studies show that the end-to-end performance of cloud storage services can be significantly improved by sufficiently exploiting the capabilities of clients and the great performance potential of cloud storage services.

The rest of the chapter is organized as follows. Section 3.2 presents the measurement methodology. Section 3.3, Section 3.4, Section 3.5, and Section 3.6 describe our observations and findings. Section 3.7 presents related system implications. Section 3.8 and Section 3.9 describe our proposed pilot solution and case studies, respectively. Section 3.10 gives the related work, and Section 3.11 concludes this chapter.
3.2 Measurement Methodology

For our experiments, we adopt a “blackbox” approach and generate direct I/O requests to the cloud to observe the performance behaviors without the interference of client-side optimizations (e.g., caching). We select a representative cloud storage service (Amazon S3) as the target cloud and various machines with different configurations (including Amazon Elastic Compute Cloud (EC2) instances and local workstations) as the clients; we also develop a homemade testing tool running on the clients to issue I/O requests to the cloud.

By using latencies and bandwidths, which are the two key metrics used in storage studies, we perform a series of experiments with different client settings to study the effect of a client’s capabilities and locations on end-to-end performance. It is 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 cloud and its client and gain insight on how to make proper optimizations on the client side. To achieve this, we investigate each factor by controlled comparison. Namely, we change one configuration of the baseline client each time and observe its impact on performance.

To comprehensively reveal the effects of different factors, our measurement work is composed of two parts. We first conduct a set of general experiments on a baseline client to evaluate the properties of cloud storage, including parallelism degree and request size (see Section 3.3). We then focus on studying the effects of client capabilities, including CPU, memory, storage, and geographical locations of the clients (see Section 3.4 and Section 3.5). 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 (see Section 3.6).

3.2.1 Experimental Platform

Cloud storage services. Our experiments are conducted on Amazon 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
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 of 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 which feature different capabilities in terms of CPU, memory, storage, and geographical location. Table 3.1 shows these configurations. The Baseline client is located in Oregon and equipped with 2 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 3.2. 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 client CPU by comparing the performance observed on the Baseline client with that on the CPU-plus.
Table 3.2. Magnetic vs. SSD on Amazon EC2

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

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 with the other comparison clients.

**Additional Experimental Platform.** The above-mentioned 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 4-core 3.2 GHz Intel Xeon CPU, 8 GB memory, a 910 GB disk drive, 1000 Mbps network connection, and installed with Ubuntu 12.04.5 LTS and Ext4 file system. The read speed of the disk is tested as 167 MB/s, and the write speed is 137 MB/s for sequential access. With these additional test platforms, we can verify our findings in a more general way.
3.2.2 Testing Tool

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 [38], 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 issued to the cloud; object number refers to the number of the objects to be requested in test. Limited by the current implementation of Amazon S3 APIs [37], our testing tool generates requests for one object per connection.

Table 3.3. Object-based Workloads

<table>
<thead>
<tr>
<th>Object Size</th>
<th>Object Number</th>
<th>Workload Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1KB</td>
<td>81,920</td>
<td>80MB</td>
</tr>
<tr>
<td>4KB</td>
<td>40,960</td>
<td>160MB</td>
</tr>
<tr>
<td>16KB</td>
<td>40,960</td>
<td>640MB</td>
</tr>
<tr>
<td>64KB</td>
<td>40,960</td>
<td>2,560MB</td>
</tr>
<tr>
<td>256KB</td>
<td>40,960</td>
<td>10,240MB</td>
</tr>
<tr>
<td>1MB</td>
<td>16,384</td>
<td>16,384MB</td>
</tr>
<tr>
<td>4MB</td>
<td>4,096</td>
<td>16,384MB</td>
</tr>
<tr>
<td>16MB</td>
<td>2,048</td>
<td>32,768MB</td>
</tr>
</tbody>
</table>

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 3.3 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. In specific,
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 \( \frac{\text{object size} \times \text{object num}}{\text{duration}} \), in which \( \text{object size} \) denotes the size
of a single object, \( \text{object num} \) denotes the number of objects, and \( \text{duration} \) denotes the time
taken to upload/download all the objects. In this dissertation, we also use peak bandwidth
to refer to the maximum bandwidth observed on the client for a given workload.

### 3.2.3 Accuracy

Considering the possible variance of network services and multi-thread scheduling, we
take the following measures to ensure the accuracy and repeatability of the experiments:
(1) As stated above, 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 3.3) 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. (4) Each experiment is repeated for five times, and we report the average value.

3.3 Basic Observations

The first step for our measurement work is to observe the I/O performance behaviors of the cloud on the baseline client, primarily aiming to reveal the effects of two critical factors that significantly affect the storage performance: parallelism and request size. 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 [34], typically smaller than 10MB [57]. 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 [34]. Therefore, we set the request size up to 16MB to study the size effect.

3.3.1 Effects 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 observed at the client side.

Effects of parallelism on bandwidth. Proper parallelization can dramatically improve the bandwidth, while overparallelization may lead to bandwidth degradation. As shown in Figure 3.1, for example, the bandwidth of 1KB upload requests can be improved up to 27-fold (from 0.025 MB/s to 0.666 MB/s), and the bandwidth of 1KB download requests can be improved up to 21-fold (from 0.03 MB/s to 0.63 MB/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 over-

Figure 3.1. Upload and Download Bandwidths on Baseline
parallelization. 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 1 job to 8 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.

Effects 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 3.2 and Figure 3.3, 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 36 ms). When the parallelism degree further increases from 16 to 64, the average latency increases by 43% (from 36.1 ms to 51.5 ms). 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.

![CDFs of 4KB Upload Latencies](image1.png)

(a) CDFs of 4KB Upload Latencies

![CDFs of 4MB Upload Latencies](image2.png)

(b) CDFs of 4MB Upload Latencies

Figure 3.4. CDFs of Upload Latencies with Different Parallelism Degrees on Baseline

![CDFs of 4KB Download Latencies](image3.png)

(a) CDFs of 4KB Download Latencies

![CDFs of 4MB Download Latencies](image4.png)

(b) CDFs of 4MB Download Latencies

Figure 3.5. CDFs of Download 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 3.4 and Figure 3.5, 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 200 ms, 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.

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 effect. In this case, since the disk speed is limited (only 23 MB/sec) and lower than the downloading speed (78 MB/sec), the client memory will be quickly used up, and the disk flushing time would be reflected in the critical path and affect the end-to-end 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 3.6 that such “steps” disappear, which confirms our speculation.
3.3.2 Effects 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.

**Effects 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 request size exceeds a threshold. As shown in Figure 3.1(a) and Figure 3.1(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 (58.9 MB/s vs. 2.5 MB/s); the peak bandwidth of 4MB download requests is 10.7 times that of 4KB download requests (28.9 MB/s vs. 2.7 MB/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 3.2). Another reason is that larger requests have higher efficiency of data transmission via network due to the packet-level parallelism [83]. 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 3.2, 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 [18], [87]. 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, etc., can also limit the achievable bandwidth. All these observations demonstrate that the benefit obtained by increasing request size is significant but is not unlimited.
Effects of request size on latency. Both increasing request size and parallelizing small requests can lead to increased latency. For example, as shown in Figure 3.2 and Figure 3.3, when the parallelism degree is 1, the average latency of 4MB download requests is 192 ms—5.8 times that of 1MB download requests (33 ms). However, when taking parallelism degree into consideration, things become different. For example, the average latency of 4MB download requests at parallelism degree 1 is 192 ms, which is 13.8 times lower than the average latency of 1MB download requests at parallelism degree 64 (2.9 s). Therefore, without considering the latency increase caused by over parallelization, 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 3.2(a) shows that the average latencies of 1KB and 4KB upload requests are nearly the same. Similarly, in Figure 3.3(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 [83], the transmission time of the data that are smaller than 64KB is supposed to be similar.

3.3.3 Combining Parallelism and 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 3.7 shows the performance under different combinations of parallelism degree and request size. Obviously, 256KB×16 has the highest bandwidth (44.2 MB/s), which is about 3 times of the lowest (14.5 MB/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.

We also find that, in some cases, either increasing parallelism degree or increasing 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 (0.4 MB/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 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 less CPU resources. Through `vmstat` in Linux, we find that the CPU utilization of the above three cases are 65%, 15% and 5%, respectively. This indicates that for the combinations that can achieve comparable bandwidth, a larger request size consumes less 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.

### 3.3.4 Remarks and Summary

**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 3.4 and Section 3.5, 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., [118]), we find that parallelism and request size are important to the end-to-end performance of cloud storage and also several 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 3.8.

### 3.4 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 clients, including **CPU-plus**, **STOR-ssd** and **MEM-minus**, with the performance of the **Baseline** to reveal the effects of each factor.
3.4.1 Effects of 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 (2 CPUs) and CPU-plus (4 CPUs).

Effects of client CPU on bandwidth. The client CPU has a strong impact on cloud I/O bandwidth, especially for small requests. Figure 3.8 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 3.8(b), the peak download bandwidth of 1KB, 4KB and 16KB requests doubles, as the computation capability doubles (2 CPUs vs. 4 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 3.8, 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.

Effects 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.

3.4.2 Effects of Client Storage

Client storage plays an important role in data uploading and downloading: For uploading, the data are first read from the client storage; for downloading, the data are written to the client storage. To evaluate the effect of client storage, we set up a comparison client \textit{STOR-ssd}. The only difference between \textit{Baseline} and \textit{STOR-ssd} is storage (Magnetic vs. SSD). Table 3.2 shows more details about testing results of I/O performance.

![Figure 3.9. Peak Bandwidth Comparison (Baseline vs. STOR-ssd)](image)

**Effects of the client storage on bandwidth.** We find that \textit{client storage is a critical factor affecting the achievable peak bandwidth}. As shown in Figure 3.9, on \textit{STOR-ssd}, the peak download bandwidth increases significantly. For example, the peak download bandwidth of 4MB requests increases by 165\% (76.6 MB/s vs. 28.9 MB/s). On the other hand, we also notice that the upload bandwidth increases slightly. Different from the significant improvement of download bandwidth, for example, the peak upload bandwidth of 4MB requests increases only by 2\% (60.3 MB/s vs. 59.2 MB/s). The reason why \textit{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 (77.2 MB/s for uploading and 80.3 MB/s for downloading). In this case, the bandwidth is close to the limit of the network bandwidth (82 MB/s), indicating that the network becomes a bottleneck when the client storage is highly capable.

![Figure 3.10. Average 16MB Download Latency Comparison (Baseline vs. STOR-ssd)](image-url)

**Effects of the client storage on latency.** Similar to bandwidth, we did not observe significant effects of client storage to 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 speed, 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 3.10, when the parallelism degree is 1, *STOR-ssd* can reduce the latency by 24% (0.49 s vs. 0.64 s); when parallelism degree is 32, the latency can be reduced by 65% (7.7 s vs. 21.8 s).
3.4.3 Effects of Client Memory

Memory in the clients is used for two aspects. 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 7.5 GB memory while MEM-minus has only 3.5 GB.

Table 3.4. Peak Upload Bandwidth Comparison (Baseline vs. MEM-minus)

<table>
<thead>
<tr>
<th></th>
<th>1MB</th>
<th>4MB</th>
<th>16MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>59.2 MB/s</td>
<td>59.1 MB/s</td>
<td>58.9 MB/s</td>
</tr>
<tr>
<td>MEM-minus</td>
<td>58.9 MB/s</td>
<td>58.7 MB/s</td>
<td>58.7 MB/s</td>
</tr>
</tbody>
</table>

Table 3.5. Peak Download Bandwidth Comparison (Baseline vs. MEM-minus)

<table>
<thead>
<tr>
<th></th>
<th>1MB</th>
<th>4MB</th>
<th>16MB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline</td>
<td>26.7 MB/s</td>
<td>28.9 MB/s</td>
<td>28.9 MB/s</td>
</tr>
<tr>
<td>MEM-minus</td>
<td>23.7 MB/s</td>
<td>23.8 MB/s</td>
<td>20.8 MB/s</td>
</tr>
</tbody>
</table>

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 3.4) while the download bandwidth dropped heavily (see Table 3.5). For example, on MEM-minus, the bandwidth of 16MB download is 20.8 MB/s, which is 28% lower than that on Baseline (28.9 MB/s). That is because the write speed of the Magnetic is much lower than 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.

3.4.4 Remarks and Summary

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, 60 GB
memory and 10 Gbps 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 470 MB/s, which demonstrates the great performance potential of cloud storage services and also clearly shows the important role of client’s capabilities in determining the end-to-end I/O performance.

Summary. Client’s capabilities have a strong impact to the end-to-end I/O performance of cloud storage. Based on our observations, CPU is important to small and highly parallelized requests, while storage and memory have significant effects to large requests. Understanding the effects of client’s capabilities can provide us guidance to properly reshape workloads (e.g., selecting a proper parallelism degree and request size combination) to sufficiently exploit client’s 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 be not desirable.

3.5 Effects of Geographical Distance

For cloud storage, the geographical distance between the client and the cloud determines the Round-Trip Time (RTT), which accounts for a significant part of the observed I/O latency. The RTT between the Baseline client and the cloud is 0.28 ms, 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 (176 ms). This section discusses the effects of geographical distance.

Effects of geo-distance on bandwidth. The effect of geographical distance on the achievable peak bandwidth is weaker than expected. As shown in Figure 3.11, 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 (53.3 MB/s vs. 59.2 MB/s) while the peak download bandwidths of 4MB download requests are basically the same (29.3 MB/s vs. 28.9 MB/s). This means that RTT is not a critical factor affecting the peak bandwidth, which is mostly due to the Bandwidth-Delay Product.
Figure 3.11. Peak Bandwidth Comparison ($Baseline$ vs. $GEO-Sydney$)

(BDP) of the network and is also consistent with the conclusion obtained by Burgen et al. [29] that the end-to-end bandwidth observed 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 3.4.1, 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.

Effects of geo-distance on latency. As expected, we also find that the geo-distance would significantly increase the latency, and its impact to latency makes the client less sensitive to the negative effects caused by overparallelization to latency. As shown in Figure 3.12, when the parallelism degree is 1, the average latency of 16MB upload requests on $GEO-Sydney$ is 2.1 s, which is about 2.6 times of the counterpart on $Baseline$ (0.8 s); as the parallelism degree increases, the average latencies gradually get closer; when the parallelism degree reaches 16, the average latencies are comparable (4.3 s vs. 4.2 s). Comparatively, $GEO-Sydney$ 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
Figure 3.12. Average 16MB Upload Latency Comparison (Baseline vs. GEO-Sydney) appears later.

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 128 ms, which is much higher than the RTT between the Baseline client and the cloud in Oregon (0.28 ms). We repeated the same experiments on GEO-Ireland, and the experimental results have confirmed our findings shown above.

We further conducted a test on the Local-campus client, which has a stronger capabilities (4-core 3.2 GHz CPU, 8 GB memory, 910 GB disk, 1000 Mbps network) but is remote to the Oregon data center. The observed peak bandwidths for uploading and downloading can reach close to 100 MB/s, which is much higher than that achieved on the Amazon EC2 clients. This also shows that the end-to-end performance on cloud storage is significantly affected by client’s 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 [29] 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.

3.6 Interferences of Mixed Requests

In prior sections, we have 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 for
ten times.

3.6.1 Interferences 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.

Interferences of uploading and downloading to bandwidth. For small requests,
as shown in Figure 3.13, the bandwidth of both upload and download requests can still in-
crease, but the increasing rate is much slower than that is without interference. Since both
of their parallelism degrees increase, their bandwidth can be improve before the pipeline
is sufficiently saturate. Due to the competition, the increase rate is relatively slower. In-
terestingly, small download requests can obtain more bandwidth when competing with small
upload requests. The bandwidth of 4KB upload requests stops increasing at parallelism degree 8 when being interfered by 4KB download requests of parallelism degree 8, as shown in Figure 3.13(a), while the bandwidth of 4KB download requests can continuously increase, as shown in Figure 3.13(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.

Table 3.6. Mixed vs. Non-mixed Upload/Download Bandwidths (4MB)

<table>
<thead>
<tr>
<th></th>
<th>mixed</th>
<th>non-mixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upload</td>
<td>29.8 MB/s</td>
<td>58.9 MB/s</td>
</tr>
<tr>
<td>Download</td>
<td>18.2 MB/s</td>
<td>28.9 MB/s</td>
</tr>
</tbody>
</table>

For large requests, the bandwidth of both upload requests and download requests decreases dramatically. The peak bandwidth of both upload and download requests drop 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 3.6, the bandwidth of 4MB uploading requests is 58.9 MB/s without interference, while the total peak bandwidth of the mixed requests is 48 MB/s (the bandwidth of upload requests is 29.8 MB/s, and the bandwidth of download requests is 18.2 MB/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 almost 50% (from 20 MB/s to 10.7 MB/s), while the upload bandwidth drops slightly (from 15.5 MB/s to 13.5 MB/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 read speed, the downloads, especially those involving intensive write operations on the client, is more sensitive to the interference.

Interferences 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 are more intensive for client resources, which is consistent with our prior observations (see Section 3.3.2).

3.6.2 Interferences 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 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.

![Figure 3.14. Bandwidths of Mixed 4KB/4MB Upload](image-url)
Interferences 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 3.14(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 3.14(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.

![Graphs showing latencies of mixed 4KB/4MB upload requests](image)

Figure 3.15. Latencies of Mixed 4KB/4MB Upload

Interferences of small requests and large requests to latency. As shown in Figure 3.15, 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.
3.6.3 Remarks and 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; (2) download requests are more sensitive to the competition of the co-running upload requests. The former is mostly caused by the competition of CPU resources, while the latter is mainly caused by the fact 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 are also client-dependent.

3.7 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 re-transmitted. 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 client’s capabilities and geographical distance, but the sampling- and inference-based method helps make a proper decision (see Section 3.8).

The client’s capability has a strong impact to the end-to-end 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 to 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 two-fold. 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 co-runners, 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 batch, 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 performance potential of cloud storage demands careful consideration of various factors.
3.8 A Pilot Solution: Sampling and Inference

We have observed the tradeoff between parallelism and request size and their effects in Section 3.3.3; 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 to achieve different optimization goals.

### 3.8.1 Basic Idea

The basic idea of the *sampling* - and *inference*-based approach 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 1 job to 64 jobs). The performance data with different combinations can be obtained by purposefully running a simple test (as described in Section 3.2.2). We can also leverage some hints from the workload characteristics to 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 the 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.

**Inference.** 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 combinations, $4\text{MB}\times4$ and $1\text{MB}\times8$, can achieve the maximum uploading bandwidth on the client, we can speculate that the proper parallelism degree for $2\text{MB}$ 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.

### 3.8.2 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 are 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 256 MB object, as shown in Figure 3.1(a), 4MB×4, 1MB×8, and 256KB×32 are regarded as “good” combinations, because they can achieve the peak bandwidth on the client.

![Figure 3.16. Sampling for Uploading a 4MB Object](image)

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 3.16, 256KB×16 can achieve the highest bandwidth. Similarly, for uploading a 256KB object, 16KB×16 is the proper combination.

Figure 3.17 shows the bandwidth comparison of different combinations. Among these combinations, 256KB×1, 64KB×4, and 16KB×16 are sampling combinations shown in Figure 3.16; 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.

3.8.3 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 par-
Figure 3.18. An Example of Selecting a Proper Parallelism Degree

Parallelism degrees ranging from 1 to 32) are shown in Figure 3.18. In Figure 3.18(a), the pair of numbers in parenthesis 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 3.18(b), the first number in the parenthesis 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). With a single thread, the average latency of uploading 256KB objects is 0.078 s, and the bandwidth is 3.2 MB/s. If the application prefers to minimize the latency increase caused by parallelization, the parallelism degree 8 may be a proper choice (see Figure 3.18(a)), leading to a possibly maximum bandwidth of 25 MB/s (see Figure 3.18(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 3.18(a), the parallelism degree of 21 jobs can be selected, and the corresponding bandwidth is 48.5 MB/s (see Figure 3.18(b)).

To evaluate the accuracy of our inference on the combination of 256KB×21, using our testing 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 for five times. The comparison of the inferred performance and
the measured performance is shown in Figure 3.19: the Inferred bar presents the inferred value of the combination; the Measured bar presents the measured results with a range. As measured, the average latency is 0.1002 s with the standard error is 0.00606 s, and the bandwidth is 50.11 MB/s with the standard error of 2.92 MB/s. Comparatively, our inference (0.1 s for average latency and 48.5 MB/s for bandwidth) is quite accurate.

3.8.4 Discussion

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

3.9 Case Studies on Cloud-based Applications

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. 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.

The applications we select are grep, tar, and filesystem. From the perspective of
performance, the first two applications are more bandwidth oriented for achieving short
overall execution time, while the last requires low average request latencies. By investigat-
ing 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.

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 PC equipped with
a 2-core 3.3 GHz Intel Core i5 CPU, 8 GB memory, a 450 GB disk drive, and installed with
Ubuntu 12.04.5 LTS and Ext4 file system.

3.9.1 Parallelizing Informed Prefetching

Informed prefetching refers to the prefetching method that is based on hints given
by the applications. Since the to-be-accessed data set 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 the prefetching I/O requests. The function of *grep* is to search for a certain string in a data set (a file or a directory) which 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 data set from the command parameters and thus download the data set in parallel if necessary.

![Figure 3.20. Execution Time of grep with Different Parallelism Degrees](image)

In our experiments, we use *grep* to search for the string “prefetch” in the directory “linux-4.3/fs”, which is a sub-directory of the source code tree of Linux kernel (version 4.3) stored in the cloud. The execution time of *grep* with different parallelism degrees (see Figure 3.20) shows that via sufficiently parallelizing the downloading processes, the overall execution time can be reduced up to 94.65% (277.51 s vs. 14.86 s). This is in accordance with our expectation. Since cloud storage is parallelization-friendly and the average size of files in the data set is small (18.44 KB), 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 data set is loaded to the client is trivial (about 0.05 s).

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 (17.56 s vs. 14.86 s). 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 (17.56 s vs. 23.59 s) and 93.7% lower than that with a single thread (17.56 s vs. 277.51 s). 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 data set before deciding the parallelism degree. For example, on our client, for small file size (e.g., 4KB), we may use a relatively high parallelism (e.g., 64); for large file size (e.g., 2MB), the parallelism degree can be relatively low (e.g., 8). In the meantime, the overhead of analyzing the data set 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.

### 3.9.2 Parallelizing Synchronization

For many cloud-based applications, the data are maintained in a local directory called “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 system performance. 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 needs 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 3.21) 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% (92.71 s vs. 1002.85 s). That is because the average size of the documents of the latex project in our experiments is rather small (60.5 KB), 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 3.22. When the parallelism degree is low (\(\leq 8\)), the overall execution time decreases significantly. For example, the execution time with parallelism degree 8 decreases by 39.5% (70.29 s vs 116.13 s). 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 3.22). That is mostly because the average file size in Trace 2 is relatively large (2.64 MB). 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 attached sufficient attention to. In addition to the performance degradation, the waste of the client resources should not be ignored. 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 (116.13 s vs. 98.58 s). This means overparallelization consumes more client resources but results in worse performance.

Similar to parallelizing informed prefetching, analyzing the average file size of the data set before deciding the parallelism degree is a feasible way to avoid overparallelization. However, comparatively, the cost of analyzing the data set 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 data set (Trace 1 and Trace 2), which is negligible to the overall execution time.

### 3.9.3 Caching and Prefetching for Filesystems

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

![Figure 3.23. CDF of File Sizes](image)

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. The local cache is arranged by a standard LRU caching replacement algorithm. Similar to Linux write-back policy, we upload the dirty files that have reside on local cache for more than 30 seconds periodically (every five seconds). 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; `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; (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
local cache for more than 30 seconds periodically (every five 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 Univer-
sity [63]. We extract the object sizes and the sequence of read and write requests from the
trace. The total volume of referenced data is 4.8 GB, the average file size is 12.9 MB, and
the distribution of file sizes is shown in Figure 3.23.

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 are formatted
with a fixed block size that can be defined by the user [124]. The choice on 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 to the performance of cloud storage can guide us to 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 3.24 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 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 3.25. 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 95.2 ms to 50.2 ms), and the average write latency decreases by 40.4% (from 109.9 ms to 65.5 ms). This benefit is due to the increase of cache
hit ratio: The read hit ratio increases from 77.8% to 98.4%, and the write hit ratio increases from 88.9% to 99.4% (see Figure 3.26). This is mostly because using a relatively large chunk size allows to pre-load 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 3.26 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. In specific, 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 analysis above 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.

**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 mis-prefetching and resource waste.

![Figure 3.27. Average Download Latencies with Different Parallelism Degrees](image)

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 3.27 gives such an example: 4 parallel jobs for 4MB, 8 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 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 $n-i+1$, $\ldots$, chunk $n-1$ ($1 \leq i \leq n$) are detected to be sequentially accessed, 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.

![Figure 3.28. Average Read Latency Comparison](image)

**Table 3.7. Average Latency Reduction by Prefetching**

<table>
<thead>
<tr>
<th>Chunk Size</th>
<th>Read Latency Reduction</th>
<th>Write Latency Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>64KB</td>
<td>70.4%</td>
<td>31.1%</td>
</tr>
<tr>
<td>1MB</td>
<td>56.9%</td>
<td>24.6%</td>
</tr>
<tr>
<td>4MB</td>
<td>22.6%</td>
<td>-1.2%</td>
</tr>
</tbody>
</table>
The performance comparison of no-prefetching and prefetching are shown in Figure 3.28 and Figure 3.29. We can see that, with prefetching, the optimal chunk size is 1MB. Obviously, 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 3.7).

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 overweight the benefits. The negative effects may be resulted from two factors. First, a lot of unnecessary data are 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).

3.10 Related Work

Many prior studies have focused on addressing various issues of cloud storage, including performance, reliability, and security (e.g., [12, 36, 39, 44, 51, 64, 72, 76, 82, 100, 145, 146]). Some other work studies the design of cloud-based file systems to better integrate cloud storage into current storage systems (e.g., [32, 54, 140]). Our measurement work is orthogonal to these studies and focuses on understanding the behaviors of cloud storage from the client’s perspective and providing client-side optimization schemes.

Our work is related to several prior measurement works on cloud storage. Li et al. compared the performance of major cloud providers: Amazon AWS, Google AppEngine and Rackspace CloudServers [98]. Bermudez et al. presented a characterization of Amazon’s Web Services (AWS) [30]. Ou et al. compared a CloudFuse-based filesystem for OpenStack Swift, an open-source cloud storage, with two other IP-based storage, NFS and iSCSI [118]. Copper et al. benchmarked cloud storage systems with YCSB [49]. Meng et al. presented a benchmarking work on cloud-based data management systems to evaluate the effects of different implementation on cloud storage [110]. Bocchi et al. 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 [35]. He et al. 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 [75].

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 to achievable bandwidth [118], we have obtained findings specifically from a client’s perspective, such as the combina-
tion between parallelism degree and request size, the interference among co-running cloud storage I/Os, the effect of geographical distance, and the effect of the client’s capabilities. 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 work studied the cloud storage client applications (e.g., [56, 55, 57, 81, 104, 141, 69, 34]). 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, etc. 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, reported by Bocchi et al. [34], 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.
3.11 Conclusion

Our work on understanding the end-to-end performance of cloud storage services includes both performance measurement and case studies. Through a set of comprehensive measurement and quantitative analysis on Amazon S3, we have observed several important and interesting findings in terms of I/O parallelization, data chunking, client capability, geographical distance, and I/O interferences. Based on these findings, we have proposed a sampling- and inference-based approach to determine a proper combination of parallelism and request size to achieve different optimization goals, and illustrated how to optimize real-world applications, including informed prefetching, synchronization, and filesystems, with proper parallelization and chunking. Our findings have also inspired us to design client-side optimization schemes to improve the I/O performance and cost efficiency of cloud storage, which we will present in later chapters.
Chapter 4
Pacaca: A Parallelism- and Correlation-aware Client Cache Framework

In this chapter, we propose a client cache framework called Pacaca, which leverages the I/O parallelization and object correlations of cloud storage with particularly designed client-side caching and prefetching schemes. The experimental results demonstrate the efficiency and effectiveness of Pacaca, indicating the importance of sufficiently utilizing the unique characteristics of cloud storage for performance optimization.

4.1 Introduction

In the previous chapter, we have studied the I/O performance behaviors of cloud storage, and obtained several critical findings related to the unique characteristics of cloud storage (e.g., the effect of I/O parallelism and request size). Such unique characteristics make our long-held common-sense understandings on storage optimization not applicable or less effective in the cloud scenario. In this chapter, we focus on several critical issues that we need to consider for optimizing cloud storage.

First, I/O parallelization, rather than sequentiality, is key to optimizing the end-to-end performance of cloud storage. In cloud storage, data are stored in a large-scale storage cluster, which is designed to simultaneously process a huge number of independent parallel requests. For example, Amazon S3 and Microsoft Azure Blob Storage are able to handle millions of requests per second [1, 5]. Such an inherent capability of processing parallel I/Os has a strong implication—creating parallel I/Os should be given a top priority for improving the performance of cloud storage; on the contrary, organizing sequential I/Os, which is a classic approach for optimizing traditional storage, becomes less rewarding with cloud storage.

---

1 Parts of this chapter have been previously published as: Binbing Hou and Feng Chen, “Pacaca: Mining Object Correlations and Parallelism for Enhancing User Experience with Cloud Storage”, in Proceedings of the 26th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS’18), 2018. DOI: 10.1109/MASCOTS.2018.00036. © 2018 IEEE. Reprinted with permission.
We have conducted experiments on Amazon S3 and compared the time of downloading one thousand 4KB objects in different orders from an EC2 instance. The observed performance difference was almost non-existent. The same observations were obtained in the experiments with the object size varying from 16 KB to 4 MB. By contrast, properly parallelizing I/O jobs (e.g., downloading the objects of 4 KB with four parallel threads) can significantly improve the bandwidth and shorten the overall I/O completion time. This indicates that the existing schemes designed for optimizing rotating media, such as organizing sequential I/Os through caching [90] and prefetching [53], become less effective for cloud storage; instead, I/O parallelization is more important to end-to-end performance.

Second, the object abstraction of cloud storage enables rich opportunities to explore the semantic relationships among objects. Unlike conventional block-based storage, which provides a simple Logical Block Address (LBA) interface, cloud storage presents an object-based abstraction. In object-based cloud storage [17, 4, 117, 11], the basic entity of user data is an object, which is associated with certain metadata. Objects are further organized into logical groups, called buckets or containers, forming a flat namespace.

Such an object-based storage model can carry much richer semantic information. A particularly useful knowledge is the relationship among objects. For example, when a Netflix movie trailer is accessed, the full movie is likely to be downloaded soon. Another example is compiling a programming project. The source code files have inherent logical dependencies: the related header files need to be read together with the main source code files. Therefore, compared to obtaining the relationships from the block layer [102], mining the relationships among objects is more effective and much simpler. This opportunity will enable numerous unprecedented optimization opportunities for caching, prefetching, compression, scheduling, and many others.

Third, accessing cloud storage objects may result in drastically different access costs. For direct-attached storage, such as HDDs and SSDs, access costs (I/O latencies) vary in a relatively small range (at the level of milliseconds, or smaller) and can be accurately
modeled [111, 112]. Comparatively, access costs for cloud storage are more variable. First, end-to-end cloud I/O latencies largely depend on object sizes and network conditions, such as the network speed and the geographic distance, and thus may vary significantly (from milliseconds to seconds). Second, parallelized accesses can significantly change the access cost. For example, our experiments show that downloading four small objects (e.g., 4 KB) in parallel demands almost identical time as downloading an individual one.

A strong implication to us is that we cannot continue to assume that the data access cost is a constant value. For many system schemes, such as object-based caching, we need to differentiate the miss penalties of different objects and design a cost-aware caching scheme. Unfortunately, many traditional caching schemes (e.g., [89, 91, 109, 123, 147]) are cost-unaware, and thus unsuitable for cloud storage; the classic cost-aware caching schemes, such as GreedyDual-Size (GDS) [41], are not designed for recognizing the change of access costs caused by cloud I/O parallelization.

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

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

- To accurately identify the most appropriate candidates for prefetching, we have designed an efficient mining scheme, called Frequent Cluster Mining (FCM), to discover
object correlations, and group the correlated objects into clusters. FCM adopts a “black-box” approach, without relying on application-specific knowledge or data semantics, to fit the cloud environment.

- To properly exploit the great parallelism potential of cloud storage, we have developed a parallelized prefetching scheme, which is closely coordinated with the mining scheme FCM for achieving high prefetching accuracy, proper control on parallelism degree, and effective mis-prefetching detection and handling.

- To improve the caching efficiency, we have studied the impact of parallelized prefetching on the access costs of objects, and further developed a cost-aware caching scheme to leverage the awareness of object correlations and parallelized prefetching.

- To evaluate the efficiency of client-side caching and prefetching, we have designed a cache management framework. Besides the caching scheme of Pacaca, this framework supports three traditional cache replacement policies, including LRU, ARC [109], and GreedyDual-Size (GDS) [41], which are integrated with our prefetching scheme. The experiments on Amazon S3 show that our optimization schemes can effectively reduce cloud I/O latencies, outperforming traditional schemes by up to 58%.

The rest of the chapter is organized as follows. Section 4.2 and 4.3 present the mining and prefetching schemes. Section 4.4 presents our caching scheme. Section 4.5 describes the cache management framework, which integrates all the schemes together. Section 4.6 gives the evaluation results. Section 4.7 discusses some related issues for using Pacaca in practice. Section 4.8 presents the related work, and Section 4.9 concludes the chapter.

4.2 Frequent Cluster Mining

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

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

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

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

These two goals, unfortunately, cannot be properly satisfied by using conventional approaches. For example, the graph-based schemes [61, 70, 71, 96, 97], though effective for small data sets, are difficult to efficiently present the correlations involving many objects and could suffer scalability issues [102]. Some correlation mining schemes are application-specific. For example, Dependency Graphs used in Web mining [115] assumes and relies on link dependencies among web pages. SEER [96] partially relies on file attributes to determine the importance of different files. Finally, the methods that are designed for other purposes, such as analyzing user behaviors [62] and making hoarding decisions [96], are not optimized for prefetching.

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

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

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

Search scope: radius. To identify the correlated objects to a given object, we first define a search scope (i.e., the neighbor accesses in the access stream). We use a “look-around circle”, whose size is determined by radius (see Figure 4.1). When setting radius to two, for example, FCM searches the correlated objects for a given object in the scope of the past two neighbor accesses and the following two neighbor accesses. The rationale is that the correlated objects for a given object may appear in the access sequence before or after that object. Some objects are correlated but do not necessarily have strict semantic dependencies, and thus they can be accessed in different orders. By restricting radius, FCM can discover the correlated objects that are closely accessed within a small access distance.

In general, we can set the search scope to a reasonably small value, such as 64 in C-Miner [102] or 20 in SEER [96]. In our design, since the search scope also determines the potential cluster size, we set radius to half of the upper bound of a proper cluster size, which is estimated by the parallelism control for prefetching (see Section 4.3.1).

Search depth: search_limit. Another important factor is the recency of the identified
correlations—a recent object correlation is more useful than an outdated one for prefetching. Thus, different from prior methods [96, 102], FCM particularly considers the recency of object correlations. For each object, when identifying its correlated objects, we use a threshold, search_limit, to restrict the maximum backward search distance (how far we look back in the access sequence). For example, if the search_limit is 100 for a given object, FCM only examines the past 100 accesses to the object and searches the potentially correlated objects in its look-around circles. This brings two benefits. First, it ensures the identified correlation clusters to be strong and up-to-date. Second, it limits the search depth and reduces the mining time. We will further discuss the effects of the search_limit threshold later.

**Metrics:** support and confidence. To evaluate the accuracy of object correlations, we introduce the association rules and metrics, which are widely used in correlation mining [102, 131]. We use the association rule \( x \rightarrow y \) to present the correlation that if object \( x \) is accessed, object \( y \) is likely to be accessed before or after the access to object \( x \). We use two metrics, confidence and support, to estimate the accuracy and repeatability of an identified correlation. Specifically, if object \( x \) appears \( N \) times in the access sequence, and object \( y \) appears \( M \) times within the look-around circles of object \( x \), then we have the association rule \( x \rightarrow y \), and its confidence = \( \frac{M}{N} \) and support = \( M \). A high confidence means that two objects have a high possibility to be correlated, and a high support means that such a correlation is highly repeatable. So, we set the thresholds for both metrics, min_support and min_confidence, to filter out weak and rare correlations. The effects of these two thresholds will be further discussed later.

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

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

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

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

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

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

In addition to discarding infrequent objects (with the $\text{min\_support}$ threshold) and limiting the search depth (with the $\text{search\_limit}$ threshold), another technique for FCM to improve the mining efficiency is to early prune the “unpromising” objects and the association rules that are predicted to be impossible to satisfy the rules in the remaining process of searching.

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

\textit{Phase 3}: FCM generates clusters based on the cluster definition (see Section 4.2). The association rules that do not satisfy the definition are removed first. Then FCM scans each object to find potential clusters based on its association rules. For example, for an object \( a \), which has the association rules \( a \to b \) and \( a \to c \), FCM checks the association rules in the descending order of their \textit{confidence} values. For the association rule \( a \to b \), if \( b \to a \) also exists, the two objects are grouped as a cluster \{\textit{ab}\}. Then, FCM continues to check \( a \to c \). Object \( c \) can be added to cluster \{\textit{ab}\} if and only if the association rules \( c \to a \), \( c \to b \), and \( b \to c \) also exist, which is based on the definition of a cluster that any two objects in a cluster should be correlated to each other. Once an object is added into a cluster, it is not further considered in the remaining process of clustering. This procedure is iterated over all the remaining objects until completion.

\textbf{Discussion}. Similar to prior work [102], FCM has time complexity \( O(n) \). In practice, the efficiency of FCM is further optimized with several important measures. First, FCM only focuses on the correlations of frequent objects, rather than the semantic relationships of all the objects. Second, FCM particularly considers the \textit{recency} by using the \textit{search\_limit} threshold to limit the search depth, which together with other thresholds constrains the number of candidate association rules for an object. Third, FCM prunes the “unpromising” objects and the association rules as early as possible.

\section*{4.3 Parallelized Prefetching}

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

4.3.1 Parallelism Control

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

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

To properly restrict the cluster size, we set the search scope $\text{radius}$ in our mining scheme (see Section 4.2) to half of the upper bound of a proper cluster size, which ensures that the size of the obtained clusters would not be excessively large. With such a setting, fetching objects in parallel would not cause significant over-parallelization.

4.3.2 Handling Mis-prefetching

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

4.4 Parallelization-aware Caching

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

4.4.1 Impact of Parallelized Prefetching

Parallelized prefetching can change the relative costs of accessing objects from the cloud. Specifically, for the correlated objects that can be prefetched in parallel, the access cost is amortized, and thus the relative cost is lower than fetching each object individually.

A direct implication to caching is that the relative cost of fetching an object in a cluster upon a cache miss would be significantly smaller (i.e., a lower miss penalty). This would change the equation for making a caching decision—evicting a low-cost object is a wise choice. Without such awareness, simply combining parallelized prefetching with traditional caching algorithms, such as LRU, ARC [109], and GreedyDual-Size (GDS) [41], would be sub-optimal.

4.4.2 An Illustrative Example

To illustrate the impact of parallelized prefetching on caching, we give a simple example in Table 4.1 to show the difference between the caching scheme of Pacaca and the traditional LRU caching scheme, which is widely adopted in current cloud-based storage systems [3, 114, 7, 8, 9, 125, 32, 140]. In the example, both schemes handle the same access stream in the scenario of parallelized prefetching (downloading all the objects of a cluster in parallel upon a related cache miss). Table 4.2 describes the sizes, latencies, and the access costs of the objects and clusters.

As shown in Table 4.1, Pacaca has resulted in a lower aggregate latency than LRU
Table 4.1. An Illustrative Example of Pacaca’s Caching Scheme

<table>
<thead>
<tr>
<th>Step</th>
<th>Access</th>
<th>LRU</th>
<th>Lat.</th>
<th>Pacaca</th>
<th>Lat.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>[A]</td>
<td>7</td>
<td>[A, {B1, B2, B3, B4}]</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>B2</td>
<td>[B2, B1, B3, B4, A]</td>
<td>0</td>
<td>[A, {B1, B2, B3, B4}]</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>B3</td>
<td>[B3, B2, B1, B4, A]</td>
<td>0</td>
<td>[A, {B1, B2, B3, B4}]</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>B4</td>
<td>[B4, B3, B2, B1, A]</td>
<td>0</td>
<td>[A, {B1, B2, B3, B4}]</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>C1</td>
<td>[C1, C2, C3, C4, B4, B3, B2, B1]</td>
<td>2</td>
<td>[A, {C1, C2, C3, C4}]</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>C2</td>
<td>[C2, C1, C3, C4, B4, B3, B2, B1]</td>
<td>0</td>
<td>[A, {C1, C2, C3, C4}]</td>
<td>0</td>
</tr>
<tr>
<td>8</td>
<td>C3</td>
<td>[C3, C2, C1, C4, B4, B3, B2, B1]</td>
<td>0</td>
<td>[A, {C1, C2, C3, C4}]</td>
<td>0</td>
</tr>
<tr>
<td>9</td>
<td>C4</td>
<td>[C4, C3, C2, C1, B4, B3, B2, B1]</td>
<td>0</td>
<td>[A, {C1, C2, C3, C4}]</td>
<td>0</td>
</tr>
<tr>
<td>10</td>
<td>B1</td>
<td>[B1, C4, C3, C2, C1, B4, B3, B2]</td>
<td>0</td>
<td>[A, {B1, B2, B3, B4}]</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>B2</td>
<td>[B2, B1, C4, C3, C2, C1, B4, B3]</td>
<td>0</td>
<td>[A, {B1, B2, B3, B4}]</td>
<td>0</td>
</tr>
<tr>
<td>12</td>
<td>B3</td>
<td>[B3, B2, B1, C4, C3, C2, C1, B4]</td>
<td>0</td>
<td>[A, {B1, B2, B3, B4}]</td>
<td>0</td>
</tr>
<tr>
<td>13</td>
<td>B4</td>
<td>[B4, B3, B2, B1, C4, C3, C2, C1]</td>
<td>0</td>
<td>[A, {B1, B2, B3, B4}]</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Total Time</td>
<td>18</td>
<td></td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>

Note: This is an example illustrating the advantages of the caching scheme of Pacaca over the traditional LRU caching scheme in the scenario of parallelized prefetching, in which all the objects of a cluster are downloaded in parallel upon related cache misses. In this example, the cache space is set to 16, and the cache is empty before Step 1. The objects shown in the cache from left to right have caching priorities from high to low. The objects of the lowest caching priority have the least “value” to be held in cache. The objects downloaded from the cloud are boldfaced. The sizes, downloading latencies, and costs of the objects and clusters are shown in Table 4.2.

Table 4.2. Access Costs of the Objects/Clusters

<table>
<thead>
<tr>
<th>Object/Cluster</th>
<th>Size</th>
<th>Latency</th>
<th>Latency/Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>8</td>
<td>7</td>
<td>0.875</td>
</tr>
<tr>
<td>{B1, B2, B3, B4}</td>
<td>8</td>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>B1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>B2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>B3</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>B4</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>{C1, C2, C3, C4}</td>
<td>8</td>
<td>2</td>
<td>0.25</td>
</tr>
<tr>
<td>C1</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>C2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>C3</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>C4</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: \{B1, B2, B3, B4\} denotes the cluster containing objects B1, B2, B3, and B4; \{C1, C2, C3, C4\} denotes the cluster containing objects C1, C2, C3, and C4. The latency of a cluster is the time units of downloading the objects of the cluster in parallel. The cost of each object or cluster is calculated by \(\text{latency/size}\).
(13 time units vs. 18 time units). Initially, the two caching algorithms have the same content. At Step 6, Pacaca and LRU begin to make distinct caching decisions. Since LRU makes the caching decisions only based on the recency of each object and finds that object A has a lower recency than other objects; consequently, LRU decides to evict object A. This decision leads to a cache miss of object A at a later time (Step 14), causing a high miss penalty (7 time units). By contrast, knowing that objects B1, B2, B3, and B4 are correlated and could be fetched in a cluster \{B1, B2, B3, B4\} in a parallelized manner, Pacaca estimates that the miss penalty of the cluster is lower than that of object A (0.25 cost unit vs. 0.875 cost unit). Thus, Pacaca decides to evict the cluster \{B1, B2, B3, B4\}, which leads to a relatively lower penalty (2 time units) at Step 10.

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

4.4.3 Cache Replacement Policy

Pacaca adopts a cost-aware cache replacement algorithm based on GDS [41]. Our augmented algorithm is capable of recognizing clusters of objects. The objects in a cluster are fetched together in parallel, when a related cache miss happens. We use a cluster as the basic unit for cost estimation. An object that does not have any correlated objects is considered as a special cluster containing a single object.

Figure 4.2 shows the algorithm of the caching scheme. Each cluster is associated with a value \(H\) to determine the caching priority (lines 5 and 8). The cluster with the lowest \(H\) value is selected as the victim and will be evicted first (lines 7-9). The \(H\) value is calculated as \(H(c) = L + \frac{Lat(c)}{Size(c)}\), which includes two components:

- \(L\) is a global inflation value, tracking the \(H\) value of the most recently evicted cluster. Since the cluster having the lowest \(H\) value is always selected as the victim cluster (lines 7-9), \(L\) keeps growing and indicates the access recency of the clusters. Thus, a low \(L\) value means that the cluster has not been accessed recently.
initialize \(L = 0\)
upon the request of object \(x\)
let \(c\) be the cluster containing \(x\)

if cache hit
\[
H(c) = L + \frac{\text{Lat}(c)}{\text{Size}(c)}
\]

if cache miss
while not enough cache space
update \(L = \min(H)\)
evict cluster \(d\) such that \(H(d) = L\)
parallelized prefetching for cluster \(c\)
\[
H(c) = L + \frac{\text{Lat}(c)}{\text{Size}(c)}
\]

Figure 4.2. Cache Replacement Algorithm

- \(\frac{\text{Lat}(c)}{\text{Size}(c)}\) evaluates the cost of the cluster, considering the miss penalty of the cluster per size unit. It incorporates the time of fetching the cluster in a parallelized way, \(\text{Lat}(c)\), and the size of the cluster, \(\text{Size}(c)\).

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

It is worth noting that the latency function, \(\text{Lat}(c)\), and the size function, \(\text{Size}(c)\), here should only involve the objects that have been accessed on demand rather than the entire originally identified cluster. This is because some prefetched objects could be evicted earlier due to mis-prediction, or have not reached its expiration time and are waiting to be accessed (see Section 4.3.2). Therefore, when calculating the cost of a cluster, we only consider the objects that have been accessed on demand. Similarly, when evicting a victim cluster, only the objects that have been accessed on demand will be evicted. The prefetched objects that are detected to expire will be evicted by the mis-prefetching handler (see Section 4.3.2 and Section 4.5).
4.5 Putting It All Together

After describing each of the schemes above, we are in the position to present the architecture of *Pacaca*, which is a cache management framework that incorporates these schemes to provide client-side (i.e., clients or client-side gateways/proxies) caching and prefetching for cloud storage. Figure 4.3 shows the architecture of *Pacaca*, which integrates our proposed schemes. Since the details of the three schemes have been presented in prior sections, in this section we particularly focus on the integration of these components.

**Figure 4.3. An Illustration of the Architecture of Pacaca**

**Integration of cluster mining (FCM) and prefetching.** In the framework, FCM is responsible for discovering object correlations, and the prefetching scheme exploits the correlations to make prefetching decisions. These two schemes are closely coordinated for achieving high prefetching accuracy, proper control on parallelism degree, and effective mis-prefetching detection and handling (see Section 4.2 and Section 4.3).

**Integration of caching and prefetching.** To properly integrate the caching and prefetching schemes, an important issue is to manage the cache space for caching (to store the on-demand objects) and prefetching (to store the prefetched objects). In our design, we *logically* divide the local cache into two parts: a *demand cache* for caching on-demand data and a *prefetch cache* for holding prefetched data. These two areas use different management schemes: the prefetch cache manages objects in an LRU list, and the demand cache manages objects with our caching scheme (see Section 4.4.3).

Unlike prior methods that split the cache space into two fixed-size partitions (e.g., [102]),
in our design, caching and prefetching share the cache space. This is for two practical considerations. First, without static partitioning, the cache space can be sufficiently utilized, even when prefetching does not happen frequently. For example, a cache miss to an independent object that does not have correlated objects would not trigger prefetching at all. Second, our prefetching scheme is optimized with high prefetching accuracy and correlation-aware detection for handling mis-prefetching (see Section 4.3.2). Thus we do not have to isolate the prefetched objects in a fixed-size area to reduce the cache pollution.

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

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

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

Figure 4.4. Integration of Caching and Prefetching

**Cache space management.** Figure 4.4 shows the core space management flow. Upon a request of object x, which is associated with cluster c (lines 1-2), for a cache hit in the *prefetch cache*, the object is promoted to the *demand cache* (lines 4-5); for a cache hit in the *demand cache*, the caching priority of the object is refreshed by the caching scheme (lines 6-7). If a cache miss happens (line 8), the prefetching scheme will be triggered to fetch the correlated objects in parallel (line 15), after which the on-demand object is added to the *demand cache* (line 16) and the other prefetched objects are added to the *prefetch*
cache (line 17). If the cache space is not enough, the reclaiming priority of the objects from high to low is: (1) the mis-prefetched objects (lines 9-10); (2) the on-demand objects selected by the caching scheme (lines 11-12); and (3) the prefetched objects in the LRU order (lines 13-14). With such a policy, we first evict the mis-prefetched objects identified by the mis-prefetching handler and give a higher priority to protect the on-demand objects and the recently prefetched objects, which are likely to be accessed soon.

4.6 Evaluation

4.6.1 Methodology

Emulation. We have implemented an emulator to evaluate the performance of our client-side cache management framework Pacaca. The prototype simulates a client for cloud storage similar to S3FS [125]. It provides POSIX-like APIs for users to access data stored on Amazon S3 buckets and has the support of a client cache to enable sophisticated caching and prefetching schemes. Particularly for dirty data, since a write-through policy would cause significant performance degradation [26], a write-back policy is often adopted by the cache solutions to optimize the end-to-end performance with cloud storage [79, 140, 143]. In our prototype, we adopt a write-back policy similar to that of Linux memory management mechanism: we use a background daemon to synchronize the dirty objects to the cloud storage repository when they are aged (older than 30 seconds) or evicted.

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

Scheme comparisons. Besides the schemes of Pacaca, in our prototype, we have also implemented three classic caching algorithms: (1) LRU, a popular caching algorithm used in current cloud-based storage systems in both academia [125, 32, 140] and industry [3, 114,
(2) ARC [109], one of the advanced caching algorithms that is recently adopted by gateway caching for cloud storage [143]; and (3) GreedyDual-Size (GDS) [41], a classic cost-aware caching algorithm.

We have integrated all the three caching algorithms with our parallelized prefetching schemes. With such comparisons, we can not only investigate the capability of our parallelized prefetching scheme to improve different caching algorithms but also evaluate the advantages of our caching scheme, which is aware of the parallelized prefetching. We note that sequential prefetching is a technique used by some cloud storage systems [124, 143]. However, it is only applicable to block-based cloud storage, in which the block sequentiality is visible to the block-level caching layer. For a cache serving object-based cloud storage, in which an object is the basic caching entity, sequential prefetching is not applicable due to the lack of object correlations. This has motivated us to develop the FCM mining scheme to discover object correlations. Therefore, we do not further use sequential prefetching as a comparison scheme.

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

- Calgary contains the logs of HTTP requests to the servers of Department of Computer Science of University of Calgary at Calgary, Canada [23, 25].
- NASA has 2-month HTTP requests to the web servers of NASA Kennedy Space Center in Florida [24, 25].
- Deasna2 is an NFS trace of a general workload from the Department of Engineering and Applied Sciences at Harvard University. This trace is a mix of research, web, and email workloads [105, 129].
- Home02 is another NFS trace collected in the main network of Harvard University,
which serves 10,000 active user accounts from the colleges, the graduate school, and the administration [105, 130].

Table 4.3. Details of the Training Traces and Mining Results

<table>
<thead>
<tr>
<th>Training Trace</th>
<th>Length</th>
<th># of Objects</th>
<th># of Clusters</th>
<th>Avg. Cluster Size</th>
<th>Time (secs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calgary (3 months)</td>
<td>218,519</td>
<td>5,133</td>
<td>549</td>
<td>4.7</td>
<td>4</td>
</tr>
<tr>
<td>NASA (1 month)</td>
<td>1,556,258</td>
<td>11,068</td>
<td>1,187</td>
<td>3.3</td>
<td>146</td>
</tr>
<tr>
<td>Deasna2 (1 day)</td>
<td>547,295</td>
<td>40,961</td>
<td>2,592</td>
<td>3.4</td>
<td>23</td>
</tr>
<tr>
<td>Home02 (1 day)</td>
<td>143,180</td>
<td>13,067</td>
<td>1,593</td>
<td>3.9</td>
<td>5</td>
</tr>
</tbody>
</table>

Note: The cluster size in the context of this table refers to the number of objects included in a cluster.

Table 4.4. Details of the Testing Traces

<table>
<thead>
<tr>
<th>Testing Trace</th>
<th>Length</th>
<th>Num. of Objects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Calgary (3 months)</td>
<td>238,519</td>
<td>7,913</td>
</tr>
<tr>
<td>NASA (1 month)</td>
<td>1,305,596</td>
<td>10,093</td>
</tr>
<tr>
<td>Deasna2 (1 day)</td>
<td>488,145</td>
<td>36,532</td>
</tr>
<tr>
<td>Home02 (1 day)</td>
<td>135,363</td>
<td>12,559</td>
</tr>
</tbody>
</table>

Trace pre-processing. For the filesystem traces, we convert the NFS requests by extracting file_id, offset, and length from the read and write requests. For the web traces, we focus on GET and PUT requests (corresponding to read and write requests) and use the original link as file_id. Another pre-processing on the filesystem workloads is to split large files into smaller segments (1 MB). It simulates chunking, which is widely adopted in cloud storage clients (e.g., Dropbox) for various purposes such as deduplication, compression, delta encoding, and partial updating [34, 57]. Our framework leverages local storage as the client-side cache, and all the traces are first filtered with a memory cache. The memory cache has the size of 0.1% of the data set and uses the LRU cache replacement algorithm.

Trace splitting: training and testing. To fairly test the effectiveness of identifying object correlations, we split each trace into two parts: one for training (denoted as training trace) and the other for testing (denoted as testing trace). For Calgary, we use the first three-month trace as the training trace and the next three-month trace as the testing trace;
for NASA, the first-month and the second-month data are used for training and testing, respectively. For the filesystem traces, Deasna2 and Home02, we use one-day trace for training and the next-day trace for testing. The details of the testing traces are shown in Table 4.4. The details of the training traces and mining results are shown in Table 4.3 (see Section 4.6.2).

4.6.2 Mining Correlations in Real Traces

Parameter Settings. The main purpose of mining object correlations is for parallelized prefetching. When using FCM to cluster correlated objects, we set the search_limit as 10,000, min_confidence as 0.5, and min_support as 3. The setting of these thresholds can affect the performance of Pacaca, which will be further discussed in Section 4.6.3.

As for the look-around circle, we need to set a relatively small radius to restrict the search scope for each object so that we can find correlated objects that are accessed within a small time frame; such object correlations are useful to direct prefetching. Specifically, in our experiments, we set the radius to 16 for mining web traces and 8 for mining filesystem traces. This setting considers the possible cluster size. Considering the 90th percentile of the object sizes in the web traces is smaller than 32 KB, based on the method of restricting the cluster sizes (see Section 4.3.1), the proper number of objects for parallel accesses is tested to be 32 in our systems, so setting radius to no larger than 16 is a sound choice. For the same reason, we set the radius for the filesystem traces to 8.

Mining Results. The efficiency of mining object correlations determines its practicality. Table 4.3 shows the object clusters obtained from the training traces on our platform and the related overhead. Generally, the time overhead of FCM is reasonably low. For example, it takes only 4 seconds to complete searching for object correlations in Calgary. The overhead for mining filesystem traces is also small. For example, it takes only 5 seconds to mine object correlations in Home02. Even for the most costly one, NASA, it takes only 146 seconds to cluster correlated objects from the training trace which contains the log of one-month accesses.
From Table 4.3, we can also find that the average sizes of the object clusters are larger than 3, which means that the correlations involving multiple objects are abundant in real-system traces. This demonstrates the capability of FCM to find the correlations of multiple objects.

4.6.3 Performance Evaluation

Performance Comparison. In this experiment, we set the entire cache size as 5% of the working set (i.e., the total size of the unique objects). The prefetching scheme is directed by the object correlations obtained from the traces (see Section 4.6.2). Figures 4.5-4.8 show the performance for different optimization methods working with the four workloads. For a complete comparison, we also enhanced three traditional caching algorithms (LRU, ARC, and GDS) with the same prefetching scheme as Pacaca.

Overall performance improvement. Compared to the traditional caching algorithms without prefetching (LRU, ARC, and GDS), Pacaca can significantly improve the performance. Compared to LRU and ARC, Pacaca can reduce the average latencies by up to 58%. For example, for the Calgary workload, the average latency achieved by Pacaca is 10 milliseconds while the time used by LRU is 23.8 milliseconds, and by ARC is 20.3 milliseconds (see Figure 4.5). Even compared to GDS, Pacaca can also reduce the average latency by up to 35.5% with different workloads. This demonstrates the effectiveness and

![Figure 4.5. Performance for Calgary](image)

(a) Average Latency

(b) Miss Ratio
Figure 4.6. Performance for NASA

Figure 4.7. Performance for Deasna2

Figure 4.8. Performance for Home02
efficiency of Pacaca.

*Efficiency of parallelized prefetching.* In the figures, it is clear that the prefetching scheme can substantially improve all the traditional caching algorithms. For example, for the NASA workload, our prefetching scheme can reduce the average latency of LRU by 26.5%, ARC by 32.7%, and GDS by 22% (see Figure 4.6). This means that the obtained object clusters are effective for improving performance through parallelized prefetching. We also note that the mis-prefetching ratio (i.e., the percentage of the mis-prefetched objects of all the prefetched objects) is about 4%-15% with current settings. The effects of mis-prefetching with different settings will be discussed later.

*Efficiency of cost-aware caching.* Compared to the traditional caching algorithms that are enhanced with the same prefetching scheme, Pacaca can further improve system performance. Impressively, Pacaca can significantly outperform LRU with parallelized prefetching, reducing the average latencies by up to 45.4% (see Figure 4.5). Compared to ARC with the same prefetching scheme, Pacaca can reduce the average latencies by up to 38.3% with different workloads. Specifically, for example, for the Calgary workload, Pacaca outperforms ARC with parallelized prefetching by 38.3% (see Figure 4.5). The performance improvement of Pacaca is due to its cost-aware caching scheme. Both LRU and ARC are cost-unaware, thus they select the victim objects without considering the different miss penalties of the objects. By contrast, Pacaca makes caching decisions based on the access costs of the objects and prefers to evict the low-cost objects first, especially the correlated objects that can be prefetched in parallel, which leads to a lower average latency.

Compared to GDS with parallelized prefetching, Pacaca can reduce the average latencies by up to 17.2% with different workloads. Note that in this comparison, GDS is enhanced with our proposed prefetching scheme. If compared to GDS without prefetching, Pacaca can reduce the average latencies by up to 35.5% (see Figure 4.7). As stated in Section 4.4, the difference between the caching policies of Pacaca and GDS is that GDS only considers the access cost of each individual object, while Pacaca can further recognize the
cost changes caused by parallelized prefetching. Therefore, the advantage of Pacaca over GDS with parallelized prefetching demonstrates the benefits of making caching decisions with the awareness of parallelism and object correlations.

This is consistent with our observation that Pacaca has similar miss ratios as GDS with parallelized prefetching but it achieves lower average latencies. For example, for the Calgary workload (see Figure 4.5), the miss ratios of Pacaca and GDS with parallelized prefetching are comparable (about 10%), but Pacaca can reduce the average latency by 16%. It is because evicting correlated objects, which have relatively lower access costs than individual objects, does not necessarily reduce miss ratios but can achieve lower overall miss penalties.

**Sensitivity Study of Parameters.** The performance of Pacaca can be affected by several parameters. In this section, we discuss the effects of three critical parameters: search limit, min_confidence, and min_support (see Section 4.2.2). Figures 4.9-4.11 show the performance achieved by Pacaca for the testing trace of Calgary with object correlations obtained from the training trace using different thresholds.

![Figure 4.9. Pacaca Performance for Calgary with Different Settings of search limit](image)

*Figure 4.9. Pacaca Performance for Calgary with Different Settings of search limit*

*Effects of the search_limit thresholds.* Figure 4.9 shows the performance of Pacaca for Calgary using different search_limit thresholds from 10 to 10,000. When search_limit increases from 10 to 100, the performance is significantly improved, and after that, the performance gains diminish and the average latency even slightly increases when search_limit
exceeds 500. This is because with a reasonable search scope, the object correlations become relatively stable, and further increasing the search depth cannot find more useful object correlations and could lead to performance loss. As for the mining overhead, reducing the \textit{search limit} from 10,000 to 100 results in 29\% less time consumption. For the Calgary trace, using a \textit{search limit} of 100 can uncover 90\% of the object occurrences. In our experiments, setting the \textit{search limit} close to the 90th percentile of the object occurrences also works well for other traces.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure.png}
\caption{Pacaca Performance for Calgary with Different Settings of \textit{min\_confidence}}
\end{figure}

\textit{Effects of the min\_confidence thresholds.} Figure 4.10 shows the average latencies and mis-prefetching ratios achieved by Pacaca for Calgary using the confidence thresholds (\textit{min\_confidence}) ranging from 0.1 to 0.9. When the confidence threshold increases from 0.1 to 0.2, the average latency decreases by 8.8\%; after that, the performance remains stable; when the confidence threshold exceeds 0.5, the average latency increases slightly. This can be explained from two aspects. First, a lower confidence threshold is helpful to find more object correlations but may suffer a higher mis-prefetching ratio. When the confidence threshold increases from 0.1 to 0.9, shown as Figure 4.10(b), the mis-prefetching ratio decreases from 53\% to 4\%, demonstrating that prefetching accuracy is determined by the confidence threshold. Since Pacaca can evict the mis-prefetched objects as early as possible and fetch the objects with proper parallelization (see Section 4.3), the negative effect of mis-prefetching on performance is largely mitigated. However, considering intensive
mis-prefetching would waste the system resources (e.g., network bandwidth), we find that a relatively higher confidence threshold (e.g., between 0.5 and 0.8) is more desirable.

![Figure 4.11](image)

Figure 4.11. Pacaca Performance for Calgary with Different Settings of \( \text{min\_support} \)

**Effects of the \( \text{min\_support} \) thresholds.** Figure 4.11 shows the average latencies and mis-prefetching ratios achieved by Pacaca for Calgary using the support thresholds (\( \text{min\_support} \)) ranging from 1 to 12. When the support thresholds are smaller than 3, the average latencies are comparable. After that, the performance degradation is significant as the support threshold increases. Although a low support threshold may introduce some object correlations with weak repeatability, a high support threshold would filter out a lot of useful object correlations and thus lose many opportunities for prefetching. As shown in Figure 4.11(b), the support thresholds do not have obvious effects on mis-prefetching ratios. That is because the prefetching accuracy is mainly determined by the confidence threshold. Therefore, setting the support threshold to a reasonably small value (e.g., 3 in our experiments) is generally appropriate for performance.

**Impact of optimization methods.** We further evaluate the impact of optimization methods, including the correlation-aware mis-prefetching detection and restricting cluster sizes for parallelism control.

**Impact of correlation-aware mis-prefetching detection.** To reduce the possible cache pollution, we adopt a correlation-aware approach to detect the mis-prefetched objects, and quickly reclaim their space for efficiently utilizing the cache space (see Section 4.3.2
Figure 4.12. Pacaca Performance for Calgary w/o Mis-prefetching Detection and Section 4.5). Figure 4.12 shows the performance for the Calgary workload with and without the correlation-aware detection for handling mis-prefetching, setting the confidence thresholds ranging from 0.1 to 0.9. From Figure 4.12(a), we can see that the correlation-aware approach can reduce the average latency by up to 13.1%. This is because the approach can quickly remove the mis-prefetched objects to reclaim their space, resulting in the reduced miss ratios, as shown in Figure 4.12(b). In addition, since a higher confidence threshold leads to a lower mis-prefetching ratio, the impact of the mis-prefetching detection diminishes as the confidence threshold increases.

Impact of restricting cluster sizes for parallelism control. To avoid over-parallelization, we set the search scope (radius) to restrict the cluster sizes for prefetching (see Sec-
Based on the performance capabilities of the client, we set the \textit{radius} to search for object clusters from the filesystem traces and the web traces to 8 and 16, respectively. Figure 4.13 shows the performance for the Home02 workload with different search scopes. Our setting (\textit{radius} set to 8) achieves the lowest average latency.

For setting a smaller \textit{radius} (2 or 4), the performance loss is caused by a higher miss ratio (see Figure 4.14). That is because a smaller search scope would limit the obtained object clusters in a smaller size, resulting in a smaller prefetching granularity and lower prefetching efficiency. For setting a higher \textit{radius} (16, 32, or 64), the miss ratio decreases due to a larger prefetching granularity; however, the latency increase caused by the interference among over-parallelized requests can offset the benefits of a lower miss ratio, resulting in performance degradation. In particular, when setting \textit{radius} to 64, we find that 8.4\% of the obtained clusters contain more than 16 objects; 22.3\% of these large clusters have even more than 32 objects. Downloading such large clusters in parallel would cause significant over-parallelization, impairing system performance, as shown in Figure 4.13 and Figure 4.14.

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

Figure 4.14. Miss Ratio and Download Time vs. Radius
4.7 Discussion

In this section, we discuss other related issues of using the proposed client cache framework Pacaca in practice.

**Monetary cost caused by mis-prefetching.** Cloud storage services generally adopt a usage-based pricing policy; thus, mis-prefetching may cause additional monetary cost. For example, the data center of Amazon S3 in Oregon charges $0.0004 per 1,000 GET requests, $0.005 per 1,000 PUT requests, and $0.0007 per GB for the data transfer out from the cloud [16]. In our experiments, compared to the performance improvement, the additional monetary cost caused by mis-prefetching with our prefetching scheme is insignificant. For example, with the NASA workload, integrated with ARC, our prefetching scheme can reduce the access latency by 37%, while the additional monetary cost is about 5% of the overall monetary cost. The additional monetary cost can be further reduced by lowering the mis-prefetching ratio via setting a high confidence threshold, with the penalty of losing some prefetching opportunities.

**Training interval.** FCM can be launched between a long interval to discover object clusters. We find that as the length of access sequence increases, the number of clusters discovered by FCM can increase accordingly, but the increase rate gradually decreases; after a certain length, the number of the discovered clusters will become stable. For example, the number of clusters obtained from the access sequence of the NASA trace is almost unchanged after 1.4 million accesses. This indicates that we do not have to frequently launch FCM to discover object correlations, especially after a certain period when the number of the discovered clusters becomes stable.

**Adaptive parameter tuning.** In Section 4.6.3, we have studied the effects of the critical parameters for system performance. In practice, we can first set these parameters to empirical values as suggested in Section 4.3.1; and then gradually tune them adaptively to real-time system performance. For example, if the mis-prefetching ratio is higher than expected (e.g., 10%), we can increase the min\_confidence threshold to discover the strong
and tight correlations; if only a small portion (e.g., 20%) of the obtained clusters can be used for prefetching, we can increase the \textit{min\_support} to filter out infrequent object correlations and reduce the mining overhead. We can also gradually increase or decrease the \textit{search\_limit} threshold to approach a workload-specific search scope for discovering the up-to-date and stable correlations.

\textbf{Practical working scenarios.} In addition to individual clients, and client-side proxies and gateways, Pacaca can also be used in extensive working scenarios. For example, in the environment of private cloud storage, Pacaca can coordinate the clients with the cloud servers. Specifically, the cloud servers can run our mining tool FCM to discover object correlations and share them with the clients to serve our proposed caching and prefetching schemes.

\textbf{Other common issues for client-side cache.} The usage of Pacaca also involves some common issues for client-side cache, such as cache consistency, cache partitioning, and tuning prefetching granularity when system resources (e.g., CPU or network bandwidth) are dynamic. These issues have been sufficiently studied by prior work [39]. In our work, we particularly focus on developing caching and prefetching schemes for cloud storage by exploiting its unique characteristics—object correlations and parallelism potential.

\section*{4.8 Related Work}

In this section, we present other related work that has not been discussed in this chapter. Several prior measurement work on cloud storage focuses on investigating the performance and behaviors of cloud storage services [30, 49, 98, 110] and client applications [29, 56, 55, 57, 79, 81, 104, 141]. These prior studies lay a foundation for us to understand cloud storage services. Pacaca particularly focuses on exploiting I/O parallelism, a unique characteristic of cloud storage.

Caching is widely adopted in cloud environments [28, 44, 84]. This work particularly focuses on the client-side caching and prefetching for cloud storage. LRU is a popular caching algorithm adopted in cloud-based storage systems [125, 32, 140] and also being used
in many commercial products [3, 114, 7, 8, 9]. Prefetching is another important technique to optimizing storage performance. Tombolo [143] implements a sequential prefetching scheme integrated with the SARC [33] cache replacement algorithm in a gateway simulator. This scheme provides block-based optimizations and is thus not applicable in object-based cloud storage. Our efforts include clustering semantically correlated objects, prefetching objects in a properly parallelized manner, and making cost-aware caching decisions, which also make our work different from other approaches of integrating caching and prefetching [15, 52, 60, 137, 73, 95].

Our caching scheme is a type of GDS-based cost-aware caching optimized for parallelized prefetching in cloud storage. GreedyDual-Size (GDS) [41] is originally designed for web caching, which considers locality with cost and size in cache replacement. Other prior GDS-based caching algorithms [43, 90, 92, 99] also introduce additional factors, such as power and spatial locality. Our algorithm particularly recognizes the change of access costs caused by I/O parallelization in cloud storage, and it makes caching decisions by leveraging the awareness of parallelism and object correlations.

Prior studies have also studied mining data correlations for system optimizations; however, these methods cannot satisfy the requirement of obtaining useful object correlations to guide prefetching on cloud storage. For example, the graph-based schemes (e.g., [96, 97, 61]), though effective for small data sets, cannot flexibly present the correlations involving multiple objects and suffer poor scalability due to high overheads [102]. In addition, some correlation mining schemes are application-specific. For example, Dependency Graphs used in Web mining [115] assumes and relies on link dependencies among web pages, and SEER [96] partly relies on file attributes to determine the importance of different files. Some other mining schemes designed for block storage are not suitable for cloud storage. For example, BPP [148] aims to improve the efficiency for mining block I/O patterns by differentiating sequential access patterns and other more complicated patterns. Finally, the methods that are designed for other purposes (e.g., analyzing user behav-
iors [62] and making hoarding decisions [96]) are not optimized for prefetching on cloud storage. Unlike these conventional schemes, our object correlation mining scheme adopts a “black-box” approach without relying on application-level semantics, and particularly designed for parallelized prefetching on cloud storage.

4.9 Conclusion

In this chapter, we present a client-side cache management framework, called Pacaca, to optimize end-to-end cloud I/O latencies. In this framework, we first design a cluster-based mining scheme FCM to obtain object correlations, based on which an optimized prefetching scheme preloads correlated objects in parallel. A cost-aware caching scheme further leverages the awareness of parallelism and object correlations to optimize the cache management. The experimental results show the effectiveness and efficiency of our proposed schemes, which demonstrates that it is important to consider the unique characteristics of cloud storage, such as its parallelism potential and object correlations, to achieve the desired optimization goals.
Chapter 5
GDS-LC: A Latency- and Cost-aware Client Caching Scheme

In this chapter, we propose a client caching scheme called GDS-LC to optimize both the access latency and monetary cost of using cloud storage. To evaluate the performance of GDS-LC, we have conducted experiments. The experimental results show that GDS-LC works well in different scenarios of using cloud storage and significantly outperforms other popular schemes.

5.1 Introduction

As a cloud-based service, cloud storage provides a platform-independent storage abstraction with a high degree of efficiency, elasticity, and flexibility. However, integrating cloud storage as a primary storage layer for serving I/O intensive workloads is still highly challenging. This is mostly caused by the following two reasons.

First, cloud I/O latencies are high and sometimes variable. For using cloud storage, the client is generally connected with the cloud via the Internet. A cloud I/O may travel an excessively long distance (e.g., thousands of miles) through dozens of network components (e.g., NICs, routers, and switches). Thus, a cloud storage I/O latency could be hundreds of milliseconds or even higher, which is about 100x longer than a typical local I/O latency. Even worse, the cloud I/O latency could be highly variant depending on geographical locations of the client and the cloud. In heterogeneous cloud systems where data are stored in multiple distant data centers, such a situation is even more complex. Therefore, directly using cloud storage as a primary storage without proper optimization could incur high I/O latencies.

Second, cloud storage adopts an unconventional pricing model, which is based on the

\footnote{Parts of this chapter have been previously published as: Binbing Hou and Feng Chen, “GDS-LC: A Latency- and Cost-Aware Client Caching Scheme for Cloud Storage”, ACM Transactions on Storage, 13(4):40:1–40:33, 2017. DOI: 10.1145/3149374. © 2017 ACM. Reprinted with permission.}
actual use of the cloud services. The pricing of cloud storage is generally composed of
three components: *storage cost*, which is based on the amount of data stored in the cloud;
*request cost*, which is based on the number of I/O requests (e.g., GET and PUT) issued to
the cloud; *data transfer cost*, which is based on the volume of actual data transfer out from
the cloud. With such a pricing model, each cloud storage I/O causes a certain amount of
monetary cost. Thus, users’ I/O activities would directly impact the operation cost. This is
completely different from conventional storage, which is typically priced based on capacity
and only involves a one-time expense for the initial installation. As so, the monetary
cost of I/Os during runtime is not an issue with conventional storage but a must-have
consideration with cloud storage. Without appropriate optimization, simply using cloud
storage as a *primary* storage may incur undesirable economic loss.

Caching is a classic technique to address the above-said two issues. By using local stor-
age to temporarily reserve a copy of the most “valuable” data, most I/O requests can be
served locally, so that we can effectively reduce the I/O requests issued to the cloud and con-
sequently lower both the access latency and monetary cost for using cloud storage services.
Unfortunately, current caching schemes are sub-optimal in the cloud environment. Despite
being widely adopted in cloud-based storage systems (e.g., BlueSky [140] and S3FS [125]),
conventional caching schemes, such as Least Recently Used (LRU), can only exploit the
access pattern (e.g., temporal locality) of the workloads and do not have the capability
of differentiating the miss penalties associated with different objects. Cost-aware caching
schemes, such as GreedyDual-Size (GDS) [41], are able to make caching decisions based on
both temporal locality and other factors, including object size and access cost. In our pre-
viously proposed client cache framework Pacaca (see Chapter 4), the parallelization-aware
caching scheme extends GreedyDual-Size (GDS) [41] and further considers I/O paralleliza-
tion. However, these schemes can only focus on minimizing one target (generally access
latency) and thus cannot satisfy the requirements of minimizing both access latency and
monetary cost, requiring further optimization in the cloud environment.
In this chapter, we present a client caching scheme for cloud storage, called GDS-LC, aiming to optimize cloud storage from two aspects: access latency and monetary cost. The key idea of GDS-LC is to label each cloud storage object by their value, in terms of the access locality, object size, retrieving latency and monetary cost from the cloud, and offer high priority to protect the high-value objects in the client cache while aggressively evicting the low-value objects (i.e., the objects accessing which incurs relatively low latency and monetary cost). To achieve this, GDS-LC virtually partitions the cache space into two regions: a high-priority latency-aware region, and a low-priority price-aware region. Each region is managed by a cost-aware caching algorithm, which is based on GreedyDual-Size (GDS) [41] and designed for cloud storage scenario by adopting clean-dirty differentiation and latency normalization. The objects of high locality and high value in terms of latency and price are identified for being kept in the cache, which allows us to reshape the cloud I/O streams to the desired low-latency and low-cost pattern. With such a two-region design, GDS-LC well balances several key factors for cloud storage caching, namely locality, size, latency, and price, which helps improve overall system performance and cost. Our solution can also be flexibly extended by considering other factors for caching. For example, by incorporating frequency in the caching decision, we further present a scheme called GDS-LCF, which gives a relatively high caching priority to frequently accessed objects.

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

The rest of the chapter is organized as follows. Section 5.2 analyzes the challenges of making an effective caching design in the cloud environment. Section 5.3 describes the design of our caching schemes. Section 5.4 gives the experimental evaluation. Section 5.5 presents the related work, and Section 5.6 concludes this chapter.

5.2 Caching Issues

5.2.1 Challenges

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

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

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

First, the original GreedyDual-Size (GDS) can only optimize for one cost target. Cloud storage users are highly sensitive to both performance and monetary cost, especially for running I/O intensive workloads. Unfortunately, these two optimization goals are orthogonal, thus directly combining these two optimization dimensions together as a single numeric value lacks a concrete semantic basis. A straightforward method, for example, is to set the weighted average value of the two optimization targets as a combined cost. However, this method is based on the assumption that access latency and monetary cost are exchangeable and can be directly compared (i.e., 1 second = 1 dollar), which is not semantically meaningful. Also, the diversity of working scenarios and pricing models further complicates the selection of the weights. Therefore, the method of using a single value as the combined cost is sub-optimal. To achieve two optimization goals, we adopt a two-region design, in which each region focuses on minimizing one cost target (see Section 5.3.1).

Second, unlike storage I/Os in traditional web systems, in which the web pages are frequently read and rarely modified, storage I/Os in the cloud environment are bi-directional, meaning that the data can be not only frequently read (downloaded) but also frequently written (uploaded). The problem of directly using the cost function of the original GreedyDual-Size (GDS) algorithm designed for web caching is that it simply defines the cost as the penalties of fetching objects and ignores the cost differences of clean objects and dirty objects. Using cloud storage as a primary storage, an object can be both read (downloaded) or written (uploaded), and consequently, the cost of evicting clean objects and dirty objects can be different in cache management—evicting a dirty object incurs a high on-demand uploading latency, in addition to the downloading latency of fetching the object upon a related cache miss. Considering this, we define the cost as the penalties of evicting an
object, including the cost of downloading the object and the cost of uploading for dirty objects (see Section 5.3.2).

Third, assessing the access cost of cloud storage should also consider several cloud environment issues. For example, the access time could fluctuate due to many real-time factors (e.g., network conditions). The variance of access latency may degrade the efficiency of cost-aware caching and thus has to be well considered. To address this issue, we adopt an adaptive normalization approach (see Section 5.3.2).

5.3 Scheme Design

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

5.3.1 Cache Space Management

![Figure 5.1. The Two-region Structure for Caching](image)

To achieve both optimization goals in terms of access latency and monetary cost, in the design of GDS-LC, we adopt a two-region design: each region is managed with a dedicated cost-aware caching scheme to achieve a specific optimization target (either low access latency or low monetary cost), respectively; objects are migrated between the two regions, and the low-locality, low-latency, and low-cost ones will be finally evicted from the local cache.
**Cache partitioning.** As shown in Figure 5.1, GDS-LC logically splits the cache space into two regions: a *performance region* and a *price region*. The performance region is a high-priority region, which is reserved to contain performance-critical objects, i.e., hot objects that are to be reaccessed shortly and have long access latencies. The cost region is a low-priority region, which contains relatively cold objects with a weaker locality but a higher monetary cost. The two regions adopt two different replacement algorithms: *GDS-Latency* and *GDS-Cost*, which focus on the latency and price goals respectively. Particularly, when considering the cost, we focus on cost per size unit by using \( \text{latency/size} \) and \( \text{price/size} \), which is based on GreedyDual-Size (GDS) (see Section 5.3.2).

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

It is worth noting that GDS-LC adopts “logical” partitioning, which means that it is unnecessary to physically partition the cache space, and we simply keep track of the actual space occupied by the objects in each region. For the partition sizes, we adopt a scheme similar to the memory management in the Linux kernel by reserving one third of the total client cache space for the performance region, and the rest is reserved for the price region. Theoretically, such a cache partitioning can be dynamically tuned. In our experiments, we find the ratio 1:2 works very well across all the workloads in our test. We will further discuss the impact of different caching partitioning ratios in Section 5.4.3.

**Object migration.** Figure 5.2 shows the algorithm of object migration between the two regions. Initially, an object is admitted into the performance region (line 23-31). If
/* Procedure is invoked upon a reference to object b */
reference_object (b)
{
    if b is in cache { /* hit in cache */
        if b is in performance region
            hit_object_in_region(b, performance)
        else {
            remove_object_from_region(b, price)
        }
    /* if no space in performance region,
    demote some objects to price region*/
    while (b.size > performance.free_space)i{
        a = evict_object_from_region(performance)
        /* if no enough space in price region,
        evict some objects*/
        while (a.size > price.free_space)
            evict_object_from_region(price)
        add_object_to_region(a, price)
    }
    add_object_to_region(b, performance)
} else { /* miss in cache*/
    download b from the cloud storage
    while (b.size > performance.free_space){
        a = evict_object_from_region(performance)
        while (a.size > price.free_space)
            evict_object_from_region(price)
        add_object_to_region(a, price)
    }
    add_object_to_region(b, performance)
}

Figure 5.2. Algorithm of Migrating Objects between Regions

the performance region has available space, the object will be added into the performance region (line 31). If the performance region has insufficient space, we need to run the GDS-Latency algorithm (see Section 5.3.2) to move one or multiple low-latency objects to the price region to accommodate the new object (line 25-30). In this process, the object
with the weakest locality and the smallest latency will be demoted (line 26). If there is insufficient space in the price region, by running GDS-Cost (see Section 5.3.2), we further evict low-cost objects from the price region and reclaim enough partition space (line 27-28).

A second access to an object in the price region will promote it into the high-priority performance region (line 6-21), since the object has proven itself to have high temporal locality. If the performance region has available space, the object is added into the region (line 20); otherwise, we have to evict one or multiple objects from the performance region and demote them into the lower-priority price region (line 12-19).

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

5.3.2 Cost-aware Caching Replacement

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

**Latency function.** Compared to the latency function used in the original GreedyDual-Size (GDS) [41], our latency function has two particular considerations: clean-dirty differentiation and adaptive normalization.

*Clean-dirty differentiation.* For evicting a clean object, we set the cost to be the latency of downloading the object from the cloud, which is similar to the original GreedyDual-Size (GDS) [41]. The difference is that for evicting a dirty object, we set the cost to be the
sum of the latency of downloading the object and the latency of uploading it to the cloud. This is motivated by the fact that dirty objects have to be synchronized to the cloud before being discarded. Therefore, in addition to the miss penalty (i.e., the downloading latency caused by a cache miss), the cost of evicting a dirty object should also include the uploading latency. Such a clean-dirty differentiation gives relatively higher values to dirty objects and is thus helpful to reduce the on-demand uploading latencies.

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

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

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

It is also worth noting that a normalization factor with a fixed absolute value cannot fit all scenarios in the real world. With an excessively small normalization factor, the negative effects of the variance of access latencies cannot be effectively reduced. On the other hand, an excessively large normalization factor may weaken the capability of differentiating distinct costs of objects. In our experiments, we set an adaptive normalization factor based on the Round Trip Time (RTT) between the client and the cloud. Specifically, we set the normalization factor to be multiple times of RTT. If the client simultaneously connects with multiple clouds located in different geographic locations, we set the normalization factor to be multiple times of $RTT_{\text{min}}$ (i.e., the minimum RTT). The normalization factor can be tuned under different working scenarios. We will further study the impact of normalization in Section 5.4.6.

**Price function.** Due to its service nature, each cloud I/O takes certain amount of monetary cost. Based on the service provider’s current pricing model, it includes three components: storage cost, request cost, and data transfer cost. Since the storage cost is based on the total size of all the objects stored on the cloud, it is not related to the real-time accesses. The price of a cloud I/O includes two latter components: the request cost and data transfer cost.

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

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

### 5.3.3 Frequency-aware Enhancement

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

In GDS-LC, the two regions adopt two caching schemes: GDS-Latency and GDS-Cost, where the $H$ value (used to determine the caching priority) is updated when the object is admitted into the cache (see Section 5.3.2). Both approaches do not effectively reflect how frequently an object is referenced while it is resident in cache. As a further enhancement, we propose the second method to incorporate the frequency information into the calculation of the object values. We call the frequency-version of GDS-LC as GDS-LCF. Correspondingly, we call the frequency-version of GDS-Latency as GDS-LF and the frequency-version of GDS-Cost as GDS-CF.

By incorporating frequency into the equation used in each region (see Section 5.3.2), we get a new equation for GDS-LF and GDS-CF to calculate the value of an object (e.g., $obj$): $H(obj) = L_{region} + Cost(obj) \times Freq(obj)/Size(obj)$, in which $Freq(obj)$ refers to the approximation function of frequency. Determining a proper approximation function of frequency, i.e., $Freq(obj)$, is an important issue, since it may affect the caching efficiency substantially [108]. Frequently accessed objects are important even if not recently accessed. Some caching algorithms count at most two most recent references to each cache page (e.g., ARC [108], LRU-2 [116]). Similarly, we set $Freq(obj)$ to not become greater than two and
Specifically, we associate each object with a \textit{counter}, which is incremented by one upon an access to the object and stays unchanged when the object is demoted. $Freq(obj)$ is updated as follow: (1) For an object in the top region, $Freq(obj)$ is set to two if the \textit{counter} is larger than two; otherwise, $Freq(obj)$ is set to the value of the \textit{counter}. (2) For an object in the bottom region, $Freq(obj)$ is set to four if the \textit{counter} is larger than four; otherwise, $Freq(obj)$ is set to the value of the \textit{counter}.

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

5.4 Evaluation

5.4.1 Experimental Methodology and Environment

\textbf{Trace-driven emulation.} In order to evaluate our proposed caching schemes, we have developed a prototype to emulate a cloud storage cache manager. Our emulation simulates a typical cloud storage client cache, which leverages a specified amount of local storage space as cache for cloud storage. I/O accesses that cannot be satisfied in the local cache will be converted into \texttt{PUT} or \texttt{GET} requests to the cloud storage. For each request, we recorded the execution information including the request type, the end-to-end completion time, and whether it is a cache hit. This information can be used to calculate the hit ratio, average latency, and monetary cost of each run of the experiments.

Considering web services, file system services (e.g., S3FS [125], BlueSky [140], Tombolo [143], and SCFS [32]), and multimedia services (e.g., Netflix is deployed on Amazon S3 [19], and Spotify has moved to Google Cloud [86]) are popular and typical services using cloud stor-
Table 5.1. Trace Characteristics

<table>
<thead>
<tr>
<th>Trace</th>
<th>Service Type</th>
<th>Total Unique Object Size</th>
<th>PUT Requests</th>
<th>GET requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clark</td>
<td>Web</td>
<td>164 MB</td>
<td>645</td>
<td>229,233</td>
</tr>
<tr>
<td>Netfs</td>
<td>Filesystem</td>
<td>707 MB</td>
<td>594,433</td>
<td>135,949</td>
</tr>
<tr>
<td>Media</td>
<td>Multi-media</td>
<td>1,294 MB</td>
<td>0</td>
<td>166,366</td>
</tr>
</tbody>
</table>

Figure 5.3. CDFs of Object Sizes

...as a primary storage, we selected three representative workloads in the experiments: Clark, Netfs, and Media. The details of the traces are shown in Table 5.1. Figure 5.3 shows the distributions of object sizes.

- **Clark** is a web trace [48], which accesses 164 MB of unique objects (i.e., web pages), and consists of 229,233 GET requests and only 645 PUT requests. The object size distribution is shown in Figure 5.3(a). This workload is highly read-intensive. Thus we use a write-through policy to synchronize the data to the cloud for this workload.

- **Netfs** is a file system workload converted from the networkfs workload in the FileBench 1.4.9 [107] running on S3FS [125]. We collect the PUT and GET requests in a trace file. This workload is more write-intensive. It accesses 707 MB of unique objects (i.e., files) in total, including 135,949 GET requests and 594,433 PUT requests. The object size distribution is shown in Figure 5.3(b). A write-back policy is adopted to sync back the dirty data that reside in the cache for more than 30 seconds periodically (every 5 seconds), similar to the Linux write-back policy. All updated objects are filled in with randomly generated content.
• **Media** is a multimedia workload synthesized using the open-source generator MediSyn, in which the access pattern of multimedia objects (e.g., small video and audio files) follows Zipfian distribution [136]. We synthesized this workload by collecting the size of each unique object and traced the access sequence of object ID. By replaying this trace, we aim to simulate the object-based client caching for multimedia objects, which has attracted attention from academia (e.g., [13, 74, 126]) and is widely adopted in industrial products (e.g., VideoCache [139], Blue Coat ProxySG Appliances [21]). This workload accesses 1,294 MB of unique objects. As a typical multimedia workload, it is read-only and contains 166,366 GET requests. All objects are filled in with randomly generated content. The object size distribution is shown in Figure 5.3(c).

**Experimental platform.** Our experiments were conducted with Amazon Simple Storage Services (S3). As a representative cloud storage service, Amazon S3 is widely adopted as a storage layer in various consumer and commercial services such as Netflix [19]. Some consumer cloud storage services, such as Dropbox, directly use S3 as the low-level storage system for data hosting [2]. In our experiments, we used the S3 storage hosted in Amazon’s data centers in Oregon (s3-us-west-2.amazonaws.com) and Tokyo (s3-ap-northeast-1.amazonaws.com) as the cloud storage service providers. We also used three clients: two Amazon EC2 instances and a workstation on our campus. All the three clients use Linux 3.2.1 kernel and Ext4 file system.

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

- **Local cloud** simulates a typical cloud system where the client and the storage servers are in the same data center. In our experiments, the client is an Amazon EC2 instance and located in the Oregon data center with the Amazon S3 cloud storage.

- **Internet cloud** simulates a public cloud system in consumer environment where the
client connects to the storage service through the Internet. The client is a workstation on our campus in Louisiana and the S3 cloud storage is in the Oregon data center.

- **Hetero cloud** simulates a special scenario where a client connects simultaneously to two different clouds. The client is an EC2 instance in Singapore and connects to two S3 cloud storage systems, one in Tokyo and the other in Oregon.

Table 5.2. The Pricing Model of Amazon S3 Services

<table>
<thead>
<tr>
<th>Client</th>
<th>Data Center</th>
<th>Request Cost</th>
<th>Transfer Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>PUT</td>
<td>GET</td>
</tr>
<tr>
<td>Local cloud</td>
<td>Oregon</td>
<td>$0.005/1,000</td>
<td>$0.004/10,000</td>
</tr>
<tr>
<td>Internet cloud</td>
<td>Louisiana</td>
<td>$0.005/1,000</td>
<td>$0.004/10,000</td>
</tr>
<tr>
<td>Hetero cloud</td>
<td>Singapore</td>
<td>$0.0047/1,000</td>
<td>$0.0037/10,000</td>
</tr>
<tr>
<td></td>
<td>Tokyo</td>
<td>$0.0047/1,000</td>
<td>$0.0037/10,000</td>
</tr>
</tbody>
</table>

Note: This table shows a simplified version of Amazon S3 pricing policy. The referenced price data was collected on December 6th, 2016. Actual price fluctuates. Interested readers may refer to Amazon website for more details [16].

Table 5.2 shows the details of the pricing model corresponding to each system setup used in our experiments. It is also worth noting that we do not intend to cover all the possible use cases in the experiments; instead, using these system setups with different features in terms of access latencies and pricing policies, we attempt to evaluate various latency/cost implications in our solution.

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

**Basic experiments.** We compare our caching schemes GDS-LC and GDS-LCF with two traditional caching algorithms that focus on improving hit ratios, i.e., LRU and ARC, and two different settings of the original GreedyDual-Size (GDS) algorithm, i.e., $GDS(latency)$ and $GDS(price)$. We present the basic experimental results in Section 5.4.2. The configurations of these algorithms are as follows:
• **LRU.** The traditional LRU policy, which applies the least recently used replacement algorithm. As far as we can see in practical systems, LRU is currently the most widely adopted caching algorithm in cloud-based storage services in academia (e.g., BlueSky [140] and SCFS [32]) and industry (e.g., Nasuni [7] and SteelStore [114]).

• **ARC.** ARC is an advanced caching algorithm, which improves LRU by making use of history access references with ghost buffers to efficiently filter one-time access [108]. ARC splits the cache space into two LRU lists, i.e., T1 and T2, to manage the cache entries that are recently referenced and the cache entries that are frequently referenced (at least twice), respectively. The cache entries in T1 are promoted to T2 when they are referenced again. ARC also maintains two ghost LRU lists, i.e., B1 and B2, to track the cache entries evicted from T1 and T2, respectively. The sizes of the four LRU lists can be tuned adaptive to the access pattern of workloads (see the literature [108] for details). Since the original ARC replacement algorithm is designed for page cache and each caching unit is a fixed-sized page or block (generally 4KB), the basic adaptation granularity of ARC is the page size. In our experiments, we replace the original adaptation granularity (page size) with the object size. With such a customization, ARC can work for variable-sized objects but does not rely on object sizes and the associated costs to make caching decisions, since its methodology and working principles are not changed. Comparing our caching schemes with ARC, we aim to reveal that it is not enough to only consider recency and frequency for cloud storage caching.

• **GDS(latency).** The original GreedyDual-Size (GDS) caching scheme that directly uses the downloading latency of each object as the cost function. With this configuration, GreedyDual-Size (GDS) aims at minimizing the overall latency.

• **GDS(price).** The original GreedyDual-Size (GDS) caching scheme that uses the monetary cost of downloading each object as the cost function. With this configuration,
GreedyDual-Size (GDS) aims at minimizing the overall monetary cost.

- **GDS-LC.** The cache is divided into two regions (a performance region and a cost region). We use a size ratio of 1:2, similar to page cache management in Linux. The performance region is managed with GDS-Latency scheme, and the cost region uses GDS-Cost scheme. The difference between GDS-Latency and GDS(latency) is that the former scheme differentiates the cost of clean and dirty object and uses the normalized latency as the cost function. Similarly, compared with GDS(price), GDS-Cost has a different monetary cost function for dirty objects. For the normalization factor, we set it to ten times of RTT (Round Trip Time) between the client the cloud. In particular, in the scenario of heterogeneous cloud, we set it to ten times of the minimum RTT from the client to the clouds.

- **GDS-LCF.** The cache partitioning is the same as GDS-LC, but we further introduce the frequency factor into the cost function, thus a more frequently read or written object will have a larger weight to be protected in the local cache. For the frequency approximation, we count the access frequency at most two when it is in the performance region and at most four when it is in the price region (see Section 5.3.3). GDS-LCF sets the same normalization factor as GDS-LC.

**Extensive experiments.** In addition to the basic experiments, we investigate the effect of partition sizes in Section 5.4.3 and study the impact of latency normalization in Section 5.4.6. We also compare our proposed caching schemes with the frequency-version of GDS and the enhanced GDS-based caching schemes that can recognize clean and dirty objects in Section 5.4.5 and Section 5.4.4, respectively.

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

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

### 5.4.2 Basic Experimental Results

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

Figure 5.4 shows the experimental results with all the three workloads in the local cloud scenario. From the results, we can not only see the advantages of our caching design but also observe some interesting behaviors of different caching schemes. In this section, we first present the observations on the experimental results of the read-intensive workloads (Clark and Media) and then present some different observations on the experimental results of the write-intensive workload (Netfs).

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

*The GDS-based policies are observed to be better than LRU and ARC.* This is because
Figure 5.4. Local Cloud: Experimental Results with Different Traces.

the GDS-based policies take recency, object size, and the cost (in terms of both latency and price) into account, while LRU and ARC are cost-unaware. Since the GDS-based policies prefer to evict the objects of larger size and smaller cost, these replacement policies have higher caching efficiency. Particularly, in this scenario, the price for evicting each object is equal, since all the objects to be evicted are clean and the price only includes the cost of GET requests for internal data transfer in the data center (see Table 5.2). In this case, the GDS-based caching schemes cause less monetary cost than LRU and ARC, since they generally have higher hit ratios (see Figure 5.4(a) and 5.4(g)). As shown in Figure 5.5, we also take the experimental results with Clark, of which the cache size is set to 10% of the total
Figure 5.5. Local Cloud: An Example to Investigate the Caching Behaviors of Different Caching Schemes.

size of unique objects, to investigate the caching behaviors of different caching schemes. Figure 5.5(a) shows the distributions of the end-to-end completion time of all the requests (including both the requests served by the local cache and those served by the cloud). Figure 5.5(b) shows the differences among the size distributions with different caching schemes. For example, the object size larger than 20 KB is 37% of all the downloaded objects upon related cache misses with GDS-LC, but the corresponding number with LRU is 18%. The reason is that the GDS-based caching schemes prefer evicting larger objects. In contrast, the differences of the latency distributions are not so significant (see Figure 5.5(c)). This is because the access latency does not increase as the request size increases for small requests (e.g., smaller than 64 KB).

Compared with GDS(latency) and GDS(price), GDS-LC can minimize both average
latency and monetary cost. Specifically, the average latency of GDS-LC is close to that of GDS(latency) (see Figure 5.4(b) and 5.4(h)), and the monetary cost of GDS-LC is close to that of GDS(price) (see Figure 5.4(c) and 5.4(i)). This demonstrates the effect of the two-region design of GDS-LC: via adopting GDS-Latency in the performance region and GDS-Cost in the price region, GDS-LC keeps the most “expensive” objects in terms of both latency and monetary cost in the cache so that it can optimize both metrics at the same time.

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

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

GDS-LC can achieve lower average latency than both GDS(latency) and GDS(price), especially when the cache size is relatively small. As shown in Figure 5.4(e), for example, when the cache size is 5% of the working set, GDS-LC reduces the average latency by 21% (from 33 ms to 26 ms). That is because GDS-LC particularly considers the cost of data synchronization for evicting dirty objects, so that less dirty objects are discarded when the cache space is not enough; consequently, GDS-LC makes the requests suffer less
from waiting for on-demand synchronization (i.e., uploading). This is consistent with our observation on the number of uploadings with different caching schemes:

Figure 5.6 shows the number of uploadings and detailed monetary cost of Netfs achieved by different caching schemes with the cache size set to 10% of the working set (i.e., the total size of unique objects). The number of uploadings refers to the number of uploading requests caused by synchronizing dirty objects to the cloud. In specific, on-demand uploading refers to synchronizing dirty objects to the target cloud when being evicted from the local cache; background uploading refers to synchronizing dirty objects to the target cloud with the background write-back daemon. The monetary cost for accessing cloud objects includes data transfer fee and request fee (see Table 5.2 for the pricing model used in our experiments).

As shown in Figure 5.6(a), compared with GDS(latency), GDS-LC decreases the num-
ber of on-demand uploadings by 46% (from 2,800 to 1,500). As for the price, we find that GDS-LC and GDS-LCF do not have obvious advantages over GDS(price). This is because the total uploadings of these three caching schemes (i.e., GDS-LC, GDS-LCF, and GDS(price)) are comparable (see Figure 5.6(b)). At the same time, since data transfer is not charged in this scenario, and the fee of PUT request is 12.5 times as that of GET request (see Table 5.2), the charge of the PUT requests dominates the overall monetary cost; thus, the monetary costs of these three caching schemes (i.e., GDS-LC, GDS-LCF, and GDS(price)) are comparable (see Figure 5.6(c)).

![Figure 5.7. Internet Cloud: Experimental Results with Different Traces.](image)

ARC has more on-demand uploadings than other caching schemes (see Figure 5.6(a)).

117
That is because LRU always evicts the least recently accessed objects, which means that the most recently written objects (i.e., dirty objects) will be protected in the cache. In contrast, ARC also attempts to recognize one-time accesses and select such objects as victim objects, resulting in more on-demand uploadings than LRU. On the other hand, ARC has a higher cache hit ratio than LRU, which means that ARC can absorb more write traffic than LRU, therefore ARC has less background uploadings than LRU and finally creates less total uploadings and monetary cost than LRU. However, compared with GDS-based policies, both LRU and ARC have more uploadings and monetary cost (see Figure 5.6), since both LRU and ARC have lower hit ratios than the GDS-based policies and do not actively differentiate clean and dirty objects (compared with GDS-LC and GDS-LCF).

GDS(latency) does not work as well as expected. From Figure 5.4(e), we find that the performance of GDS(latency) is worse than that of GDS(price), and even worse than that of LRU when the cache size is 20% of the working set. It is understandable. Working with Netfs, the objects may be frequently updated with an object size change; in this case,
the access latencies could be different from the previously observed values. Without a reasonable estimation, the cost used in the caching replacement scheme may be different from the real value. Comparatively, the performance of our solution GDS-LC is more stable. This is because we only adopt the latency-aware caching scheme in the first region, and optimization including clean-dirty cost differentiation and latency normalization contribute to improving the caching efficiency.

**Internet cloud.** The Internet cloud system setup simulates a typical consumer cloud storage environment. In this case, the client locates on our campus in Louisiana and accesses cloud storage data stored in Amazon’s Oregon data center. Different with the local cloud scenario, the RTT between the client and the cloud is high (113 ms), and the price of data transfer is also more expensive. Figure 5.7 shows the results of different caching schemes in the Internet cloud scenario. Particularly, for *Clark* and *Media*, in which the requests are read-intensive (dominated by GET requests), the data transfer fee is much higher than request fee (see Figure 5.8(a) and 5.8(c)).

Similar to the results achieved in the local cloud scenario, GDS-LC and GDS-LCF perform the best, and LRU performs the worst. However, we can also find some differences caused by the distinct characteristics of the system setup in terms of latency and pricing policies in this scenario. The most obvious difference is about the results of the monetary cost. For the *Clark* workload, for example, the monetary cost of LRU is close to that of other caching schemes except GDS-LCF (see Figure 5.7(c)). Comparatively, in the local cloud scenario, significant gaps can be observed between the result of LRU and other caching schemes (see Figure 5.4(c)). This difference is caused by the charging of data transfer out from the cloud. As shown in Figure 5.8(a), compared with LRU, GDS-LC has a lower request fee but a much higher data transfer fee, so that the gap between the overall monetary cost is narrowed down. The reason why GDS-LC has a higher data transfer fee is that GDS-LC prefers to evict larger objects, leading to a larger data transfer traffic on cache misses.
In addition, for the monetary cost charged with the Netflix workload, GDS-LC and GDS-LCF significantly outperform GDS(price) in the Internet cloud scenario (see Figure 5.7(f)); comparatively, GDS-LC and GDS-LCF do not have obvious advantages over GDS(price) in the local cloud scenario (see Figure 5.4(f)). From Figure 5.8(b), we can see that the data transfer fee and GET request fee of GDS-LC and GDS(price) are close; the main difference comes from the PUT request fee. This is understandable. Since the RTT in this scenario is quite high, the dirty objects that are not synchronized by cache replacement cannot be quickly synchronized to the cloud; consequently, GDS-LC, which adds weight to dirty objects, has better caching efficiency for dirty objects. This explains why GDS-LC leads to lower PUT request fee.

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

**Heterogeneous cloud.** Heterogeneous cloud storage systems are generally adopted to exploit the advantages of multiple clouds. For example, RACS [12] adopts an RAID-like structure, which provides high-level data availability and reliability and prevents vendor lock-in problem. Several other cloud-based storage systems, such as NCCloud [82] and DepSky [31], are also based on distributing data to multiple clouds. Another use case is to integrate different cloud storage services to uniformly access the storage space, especially for the purpose of utilizing the free tiers (e.g., AWS free tier [20] and Google cloud platform free tier [68]). In this case, the data may also be distributed to heterogeneous clouds.

In our experiments, to emulate the heterogeneous cloud storage system environment, we set up an EC2 instance in Amazon’s Singapore data center as the client, which simultaneously connects to two cloud storage locating in Amazon’s Oregon and Tokyo data centers. For each data set, we evenly distribute the objects to these two data centers, organizing the data similar to RAID-0. An interesting fact is the difference of the pricing
Figure 5.9. Heterogeneous Cloud: Experimental Results with Different Traces

and network delay – Due to a shorter geographic distance to the client, the Tokyo data center can provide a shorter latency (a 74 ms RTT) for cloud storage I/Os than the Oregon data center (a 161 ms RTT). However, its pricing on data transfers is significantly higher than the Oregon data center (see Table 5.2). The client caching scheme has to intelligently tradeoff and balance the two cloud storage sources for data accesses – for each eviction decision, it needs to choose either the closer but more expensive Tokyo data center or the farther but cheaper Oregon data center. This is particularly difficult for caching schemes.

As shown in Figure 5.9, our proposed GDS-LC and GDS-LCF caching schemes perform very well in this complicated scenario. LRU performs the worst, and GDS-LC can optimize both latency and monetary cost when being compared with the original GDS
algorithms (i.e., GDS(latency) and GDS(price)); particularly in some cases, GDS-LC and GDS-LCF can perform much better than the GDS algorithms. These results well demonstrate the effect of our approach – our caching solution can well optimize both the latency and monetary cost in complicated environment.

5.4.3 Impact of Partition Size

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

As shown in Figure 5.10, we can see that the effect of cache partitioning is workload dependent. In Clark and Media, the three partition ratios have a relatively weak impact on the observed latencies, hit ratio, and cost. In contrast, the Netfs workload exhibits certain distinctions. Generally, the ratio 1:2 is a reasonably sound choice to effectively reduce both access latency and monetary cost (see Figure 5.10(e) and 5.10(f)). Particularly, compared to the ratio 1:3, the ratio 1:2 can achieve comparable monetary cost but significantly lower average latency. Thus, the 1:2 ratio is a proper choice.

Interestingly, we also note that a larger performance region does not necessarily result in a lower average latency. As shown in Figure 5.10(e), for example, when the cache size is 10% of the working set, increasing the performance region from one fourth of the cache size (with the ratio 1:3) to one third of the cache size (with the ratio 1:2), the average latency decreases from 74 ms to 59 ms; however, further increasing the performance partition to two thirds of the cache size (with the ratio 2:1), the average latency increases from 59 ms to 80 ms. The effect is caused by the object migration between the two regions (see Section 5.3.2). On the one hand, a larger performance region means that the objects that have high values in terms of latency are more likely to be kept in local cache. On the other hand, a larger performance region leads to a smaller price region, which means that the
objects denoted to the price region may be quickly evicted from the local cache and thus have less opportunities to be promoted to the performance region again upon a second access. Consequently, the relationship between the latency and the size of the performance region is not a simple linear function.

5.4.4 Comparisons to GDS with Clean-dirty Differentiation

As stated in Section 5.3.2, a significant difference between the latency functions used in our caching schemes (i.e., GDS-LC and GDS-LCF) and the original GDS-based policies (i.e., GDS(latency) and GDSF) is that our latency functions have the capability of distinguishing clean and dirty objects. Since in prior sections we have compared our caching
schemes with the original GDS-based policies, in this section, we further compare our caching schemes with the improved GDS-based policies, which have the same latency functions as our caching schemes, called GDS-L and GDS-LF. Compared to GDS-L and GDS-LC, GDS-L and GDS-LF do not have a price region and take all the cache space as the performance region. Particularly, GDS-LF counts the access frequency at most four in its cost functions, which is the same as that of GDSF (see Section 5.4.5).

Since no victim objects are dirty when working with the Clark trace and the Media trace (note that Clark is highly read-intensive and a write-through policy is adopted and Media is read-only), we use the Netfs trace in the experiments. Figure 5.11 shows the experimental results of GDS-L and GDS-LC in the scenario of Internet cloud. Since GDS-LC has a price region to keep high-price objects, it can significantly reduce the monetary cost. With respect to access latency, compared to GDS-L, GDS-LC achieves comparable (even slightly better) performance. Although GDS-LC reserves two-thirds of the cache
space as the price region, the objects that have the highest cost in terms of access latency are kept in the performance region, and the objects demoted to the price region still have opportunities to be fetched back to the performance region; thus, GDS-LC can achieve comparable average latency even with a smaller performance region. For similar reasons, compared to GDS-LF, GDS-LCF achieves lower monetary cost and comparable average latency (see Figure 5.12). The experimental results further demonstrate the advantages of our two-region design.

**Variance of experimental results.** In addition to system performance, we have also examined the variance, which can be caused by the unexpected dynamics of network performance and cloud services. In prior sections, we use the standard error of the values of each metric (i.e., hit ratio, average latency, and monetary cost) measured from five runs of the experiments to describe the variance. Since the observed variances of hit ratios and monetary cost are insignificant, for brevity, we only show the observed variances of the

![Average Latency Error Bars with Clark](a) Clark

![Average Latency Error Bars with Netfs](b) Netfs

![Average Latency Error Bars with Media](c) Media

Figure 5.13. Local Cloud: Observed Variances of Average Latencies
average latencies in Figure 5.13, 5.14, and 5.15.

From the figures, we can see that the absolute values of the variances observed in the local cloud scenario are lower than those observed in the other two scenarios; however, we do not observe obvious differences with respect to the relative variances (i.e., the ratio of the variance and the average latency), which are about 5% - 10% in all the working scenarios. As for the latency variances observed on the experimental results of different caching schemes, we find that when the hit ratios are relatively low, the variances of average latencies are relatively higher. That is because a lower hit ratio means that the client has to more frequently access the cloud and thus is more likely to subject to a larger variation of average access latencies. In particular, for LRU and ARC, the miss ratios of these caching schemes are higher than others, especially when the cache sizes are small, we can observe relatively larger variances on the average access latencies achieved by these two caching schemes.
Figure 5.15. Heterogeneous Cloud: Observed Variances of Average Latencies

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

5.4.5 Further Evaluation on the Frequency Enhancement

GDS-LCF is an enhanced version of GDS-LC. By introducing frequency into the cost functions of GDS-LC, GDS-LCF gives higher caching priority to frequently accessed objects. As shown in Section 5.4.2, GDS-LCF outperforms traditional caching schemes (i.e., LRU and ARC) and GreedyDual-Size (GDS) with different settings (i.e., GDS(latency) and GDS(price)) and can successfully improve the caching efficiency of GDS-LC in most cases. In this section, we further compare GDS-LCF with the frequency-enhanced version of GDS called GDSF and discuss the enhancement.

Both GDS-LCF and GDSF are enhanced by introducing frequency. GDS-LCF is an
Figure 5.16. Internet Cloud: Comparisons of $GDS-LCF$, $GDSF(latency)$, and $GDSF(price)$ enhanced version of GDS-LC, and GDSF is an enhanced version of GDS. We expect that the advantage of GDS-LCF over GDSF is similar to the advantage of GDS-LC over GDS: For read-intensive workloads, GDS-LCF can optimize both performance and monetary cost instead of only one optimization goal; for write-intensive workloads, GDS-LCF can significantly outperform GDSF, since the former has a two-region design and can also differentiate the cost of evicting clean objects and dirty objects.

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

Figure 5.16 shows the performance comparison of GDS-LCF, GDSF(latency), and GDSF(price) with different traces in the Internet cloud scenario. The experimental results have confirmed our speculation: for the read-intensive traces Clark and Media, GDS-LCF can achieve comparable average latency to GDSF(latency) and comparable monetary cost to GDSF(price), successfully optimizing both goals; for the write-intensive trace Netfs, GDS-LCF have much better performance than GDSF(latency) and GDSF(price).

5.4.6 Impact of Latency Normalization

As analyzed in prior sections, the latency variance for accessing cloud storage may affect

Figure 5.17. Local Cloud: Effects of Different Normalization Factors.
the efficiency of latency-aware caching schemes, and we adopt an adaptive normalization approach based on the observed Round Trip Time (RTT) between the client and the cloud to alleviate this problem. In this section, we further discuss the impact of normalization.

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

Figure 5.17 shows the experiments in the local cloud scenario, and the cache size is set to be 10% of the working set. From the figure, we can see that the hit ratios increase as the values of the normalization factors increase; however, the increase of hit ratios does not always lead to lower average latencies. As shown in the figures, we find that setting
the normalization factor to be 10x RTT achieves the best performance among the settings; meanwhile, setting the normalization factor as 1x RTT brings trivial benefits and 100x RTT may reduce the benefits. That is because local cloud has low RTT (i.e., 0.28 ms). With setting the normalization factor to be 1x RTT, the interference of the latency variance cannot be effectively reduced; while setting the normalization factor to 100x RTT (i.e., 28 ms) normalizes the latencies of many objects to 1, which may decrease the overall system performance. As for the impact of normalization on different caching schemes, we note that the impact of normalization on GDS(latency) is more significant than that on GDS-LC and GDS-LCF. That is because only the top region in the design of GDS-LC and GDS-LCF adopts the latency-aware caching scheme and the object evicted from the top region will be migrated to the second region and still has the opportunity to be fetched back instead of being immediately evicted from the local cache, which makes them less sensitive to normalization than GDS(latency).

We also note that in the scenarios of Internet cloud and heterogeneous cloud, the normalization factor 10x RTT can still achieve better performance than no normalization, but 1x RTT performs better. The performance achieved by different caching schemes with the Media trace is shown in Figure 5.18: when the normalization factor is larger than one RTT, the benefit brought by normalization is diminishing; when the normalization factor is 100x RTT, the aggressive normalization approach leads to performance loss. In the scenario of Internet cloud, the RTT is 113 ms, thus setting the normalization factor to be 100x RTT (i.e., 11.3 seconds) means almost all the latencies of the objects are normalized to 1. In this case, although the hit ratio is improved, the overall system performance is decreased. Similarly, in the scenario of heterogeneous cloud, the minimum RTT between the two (74 ms and 161 ms) is 74 ms, and setting the normalization factor to be higher than 10x RTT may cause negative effects (i.e., setting the normalization factor to be 100x RTT). Shown in Figure 5.19 is the performance achieved by different caching schemes with the Clark trace, which indicates setting the normalization factor to be 1x RTT performs the best among
Therefore, based on our observations, a proper normalization factor varies with different working scenarios. In our platform, we find 10x RTT is a good choice for the scenario in which the client and the cloud are in the same data center. A smaller normalization factor (e.g., 1x RTT) is good for the scenarios in which the clients access the cloud across data centers, where the RTT between the client and the cloud is a relatively larger value. Setting the normalization factor to an excessively large one (e.g., 100x RTT) is generally undesirable, since it removes the capability of differentiating access costs. It is also worth noting that the negative effects of latency variance to cost-aware caching schemes cannot be completely eliminated due to the difficulty of accurately predicting latency variance. In our proposed caching schemes, we attempt to reduce the interference of latency variance by using an adaptive normalization approach. In practice, we can further improve the accuracy of cost evaluations with the knowledge of the performance behaviors of cloud storage services and the variance of network services, which can be gained by long-term observations. For example, if we know that an object will be reloaded during the “busy hours” of the target cloud storage (at the time when the cloud is busy with handling intensive requests, leading to longer response time), the cost of evicting the object should be estimated higher than the download latency measured beyond the “busy hours”.

5.5 Related Work

Both cloud storage and cache replacement algorithms have received extensive studies. In this section we present other prior work most related to our work of GDS-LC.

Cloud storage recently has attracted a lot of research attention. A variety of issues of cloud storage systems have been studied, such as performance, reliability, availability, confidentiality, and service lock-in concerns [12, 27, 36, 64, 72, 82, 145]. Much research has been first conducted to characterize the performance and I/O behaviors of cloud storage (e.g., [30, 56, 57, 81, 104, 141, 79, 80, 118]). Our work of GDS-LC is orthogonal to these studies.
For easy use of cloud storage, prior research has also attempted to unify the I/O interfaces of cloud storage and file systems. For example, a cloud-backed network file system for the enterprise use, called BlueSky [140], stores data in cloud storage and accesses storage through an on-site proxy, which caches data and supports multiple protocols including NFS and CIFS. Another similar network file system design, called RFS [54], is proposed for mobile devices. SCFS [32] provides a POSIX-like interface on top of cloud storage. Similarly, S3FS [125] also provides simple filesystem-like interfaces for Amazon Simple Storage Services (S3). These solutions typically adopt an LRU-based caching scheme on local clients or proxies. Our work focuses particularly on caching schemes and can potentially enhance these systems.

Our work is also related to the caching algorithms adopted by the commercial cloud storage products, including cloud proxies and gateways (e.g., Nasuni [6], Twinstrata [138], CTERA [50], Panzura [119], StorSimple [133]). These products mainly provide storage accelerating services, acting as a cloud-based cache between user applications and remote clouds. Although the implementation details of these products are not publicly available, according to open documents, LRU is the most popular caching algorithm adopted by the majority of these products [8, 7, 114, 9]. Our work aims at optimizing the cloud-based storage systems with cost-aware caching by considering various factors, including access latency, price, object size, and access recency and frequency, and can be flexibly applied in these working scenarios to improve cloud storage services in terms of not only performance but also monetary cost.

Recent studies have studied cost-aware caching in different working scenarios for different proposes. Jiang et al. presented an OS kernel buffer cache management scheme, called DULO [90]. DULO leverages the speed distinction of random and sequential I/Os on hard disk drives and gives higher caching priority to the blocks that are randomly accessed, since random accesses are slower than sequential accesses on hard disk drives. Similarly, Li and Cox customized also proposed a caching scheme based on GreedyDual-Size (GDS) [41],
called GD-Wheel, in the scenario of key-value stores by considering recomputing latency as cost [99]. Kim and Anh presented a caching scheme, called BPLRU, for improving random writes in flash storage [93]. PS-BC [45] leverages the filtering effect of OS buffer cache to create bursty disk I/Os for disk power saving. Forney et al. introduced a set of storage-aware caching algorithms that partition the file buffer for heterogeneous storage and dynamically tune the partition sizes to balance the workloads across the storage devices [65]. Liang et al. studied caching replacement policies for distributed storage systems and proposed two off-line heuristics and an on-line algorithm by considering access latencies as the major cost when deciding the victim data [103]. Araldo et al. proposed two optimization models that either minimize the overall costs or maximize the hit-ratio, jointly considering cache sizing, object placement, and path selection, and taking the retrieval latency as the cost in the scenario of Information Centric Networks (ICNs) [22]. Jeong and Dubois made several extensions of LRU, taking into account non-uniform miss costs (e.g., the latency, penalty, power consumption, bandwidth consumption, or any other ad-hoc numerical property attached to a miss) in different practical cases, such as multiprocessor memory systems and single super-scalar processor systems [88].

Though sharing a similar design principle with these solutions by leveraging cost-awareness in caching decisions, our solution particularly aims to enhance caching for cloud storage, which shows distinct properties compared to other systems. In particular, its special performance behaviors and pricing models demand us to focus on improving cloud storage services with regard to both access latency and monetary cost. Second, our caching scheme is designed for using cloud storage as primary storage. In this scenario, I/O accesses are both read and write intensive, requiring us to fully consider the access time of handling both clean and dirty data, rather than one-direction cloud I/Os. Third, different from prior schemes that only consider the cost from only one aspect (e.g., latency, bandwidth, or energy), we aim at minimizing the cost from two orthogonal dimensions (latency and price) at the same time.
In addition to cost-aware caching algorithms, some other advanced caching optimization has been designed to improve caching efficiency. These caching schemes include exploiting application-informed data semantics [40, 120, 44], exploiting access pattern and data correlations [89, 108, 91, 147, 46, 47, 66, 94, 142, 115, 61, 101], and partitioning the cache space [65, 122, 134]. Our solution is largely orthogonal to these classic caching approaches.

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

5.6 Conclusion

Client caching is crucial to truly integrating cloud storage as a primary storage layer in computer systems. By keeping the most valuable objects in local cache and evicting the least important ones, client caching policies can influence future accesses. Leveraging such a filtering effect, we design two unique caching schemes, called GDS-LC and GDS-LCF, with an attempt to minimize future access latencies and monetary costs. Compared with traditional caching schemes, our experimental results show that our solution can effectively improve the system performance and reduce the system cost.
Chapter 6
Limitations and Future Work

Our work focuses on understanding and optimizing the I/O performance and cost efficiency of cloud storage services from a client’s perspective. We have investigated the I/O performance behaviors of cloud storage based on Amazon S3, and proposed a client cache framework called Pacaca and a latency- and cost-aware caching scheme called GDS-LC. The experimental results have demonstrated the efficiency and effectiveness of our findings and the proposed solutions.

Our work has several limitations. First, our measurement work 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, client’s properties are important and could lead to different performance observed at the client side. Thus, we may have different observations on these mobile clients, which is worth a further study in the future. Second, our study on the effect of client’s capabilities covers 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 [118]. 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.

In addition to conventional clients, the emerging Internet-of-Things (IoT) devices, such as Raspberry Pi [121], are also used as clients for data collection and processing. These devices have limited resources and are generally used as a data collector or deployed to offload the tasks of the remote cloud-based data centers. Extending our work to these resource-limited clients may bring new optimization opportunities. Another trend is that high-performance computing units (e.g., GPU [135, 127, 128]) and new storage devices (e.g., persistent memory [10, 144]) are equipped on the clients, which can dramatically improve
the client’s capabilities and change the tradeoff for system optimization. For example, GPU is highly capable of handling parallel requests and sufficiently optimized for providing high throughput; thus, for GPU-enhanced clients, the client’s compute capability may not continue to be a bottleneck, which may change the tradeoff between a high parallelism degree and a large request size. For the clients that are equipped with persistent memory, the performance bottleneck may be shifted from client-side storage to other components of the system. We will extend our work to study the interaction between the new clients and the cloud in the future work, which is helpful for us to verify our current findings and obtain new findings.

For optimizing cloud storage services, we further propose a client cache framework called Pacaca, which integrates a parallelized prefetching scheme and a cost-aware caching scheme with the awareness of object correlations and access cost. We also propose a client caching scheme called GDS-LC, which is a cost-aware caching scheme with the awareness of latency and associated monetary cost of cloud I/Os. These two solutions can be further optimized. For Pacaca, a frequent cluster mining scheme, called FCM, is designed to obtain object correlations. FCM adopts a “black-box” methodology and does not rely on application-informed information. As a future enhancement, we can add an interface for users to provide useful information to Pacaca. Such an option allows Pacaca to utilize more data semantics to make proper caching and prefetching decisions. For GDS-LC, we have observed that the cache partitioning ratio 1:2 is an empirically good choice. In the future work, we will seek to provide the mathematical proof for the optimal cache partitioning ratio or design a dynamic cache partitioning scheme, which will make our work more solid.

In addition to optimizing the current scheme design, another import future work is to integrate Pacaca and GDS-LC. Both of these two solutions are related to the management of the client cache, but aim to improve clouds storage services from different perspectives. Pacaca defines a client cache framework and focuses on utilizing data correlations and I/O parallelization, while GDS-LC aims to incorporate the awareness of monetary cost
into making caching decisions. In the future work, we will integrate GDS-LC into Pacaca to seamlessly fuse the two cost-aware caching schemes for sufficiently utilizing the unique characteristics of cloud storage. For example, we can define a new function to include all the three factors that we have particularly considered in Pacaca and GDS-LC, including parallelization, access latency, and monetary cost. This will provide a more sophisticated solution for client-side cache management for cloud storage.
Chapter 7
Conclusions

Our work on understanding and optimizing the end-to-end performance and cost efficiency of cloud storage services includes both performance measurement and scheme design.

In our measurement work, through a set of comprehensive measurement and quantitative analysis on Amazon S3, a typical cloud storage provider, we have observed several important and interesting findings. Based on these findings, we have proposed a sampling-and inference-based approach to determine a proper combination of parallelism and request size to achieve different optimization goals, and illustrated how to optimize real-world applications, including informed prefetching, synchronization, and filesystems, with proper parallelization and chunking.

Based on our measurement work, we have designed a client cache framework called Pacaca to reduce the access latency with cloud storage. Pacaca integrates client-side caching and prefetching schemes that are optimized for cloud storage by sufficiently considering its unique properties, including I/O parallelism and object correlations. Besides performance optimization, we have also made efforts to reduce the monetary cost of using cloud storage services by designing a latency- and cost-aware caching scheme called GDS-LC, which can optimize cloud storage services for both low access latency and low monetary cost. The experimental results show the advantages of our proposed framework and schemes over conventional solutions, demonstrating the importance of considering the unique characteristics for optimizing I/O performance and cost efficiency of cloud storage.

In the future work, we plan to extend our measurement work to other clients (e.g., mobile clients and IoT) and further optimize and integrate our proposed client cache framework Pacaca and our proposed client caching scheme GDS-LC. We hope that our work can inspire system practitioners and application designers to optimize cloud storage services on the client side by sufficiently exploiting its unique characteristics.
References


Appendix A
The ACM Publication Agreement

This dissertation reuses the materials of two of the author’s previous papers [80, 77] that were published by the Association for Computing Machinery (ACM). Based on the publication agreement of the ACM, authors are granted to reuse any portion of their own work in their dissertation as long as they cite the work in a proper manner. The permission policy of the ACM for reusing the published materials [14] is as below:

**REUSE**

Authors can reuse any portion of their own work in a new work of *their own* (and no fee is expected) as long as a citation and DOI pointer to the Version of Record in the ACM Digital Library are included.

- Contributing complete papers to any edited collection of reprints for which the author is *not* the editor, requires permission and usually a republication fee.

Authors can include partial or complete papers of their own (and no fee is expected) in a dissertation as long as citations and DOI pointers to the Versions of Record in the ACM Digital Library are included. Authors can use any portion of their own work in presentations and in the classroom (and no fee is expected).

- Commercially produced course-packs that are *sold* to students require permission and possibly a fee.
Appendix B

The IEEE Publication Agreement

This dissertation reuses the materials of two of the author’s previous papers [79, 78] that were published by the Institute of Electrical and Electronics Engineers (IEEE). Based on the publication agreement of the IEEE, authors are granted to reuse any portion of their own work in their dissertation as long as they cite the work in a proper manner. The permission policy of the IEEE for reusing the published materials [85] is as below:

Can I reuse my published article in my thesis?

You may reuse your published article in your thesis or dissertation without requesting permission provided that you fulfill the following requirements, depending on which aspects of the article you wish to reuse.

- **Text excerpts:** Provide the full citation of the original published article followed by the IEEE copyright line: © 20XX IEEE. If you are reusing a substantial portion of your article and you are not the senior author, obtain the senior author’s approval before reusing the text.

- **Graphics and tables:** The IEEE copyright line (© 20XX IEEE) should appear with each reprinted graphic and table.

- **Full text article:** Include the following copyright notice in the references: “© 20XX IEEE. Reprinted, with permission, from [full citation of original published article].”

When posting your thesis on your university website, include the following message:

“In reference to IEEE copyrighted material which is used with permission in this thesis, the IEEE does not endorse any of [name of university or educational entity]’s products or services. Internal or personal use of this material is permitted. If interested in reprinting/republishing IEEE copyrighted material for advertising or promotional purposes or for creating new collective works for resale or redistribution, please go to http://www.ieee.org/publications_standards/publications/rights/rights_link.html to learn how to obtain a License from RightsLink. If applicable, University Microfilms and/or ProQuest Library, or the Archives of Canada may supply single copies of the dissertation.”

Only the accepted version of your article, not the final published version, may be posted online in your thesis.
Vita

Binbing Hou was born on February 11, 1989, in Daye City, Hubei Province, China. He finished his undergraduate studies at Wuhan University of Technology in June 2011. He earned a master of science degree in Computer System and Architecture from Huazhong University of Science and Technology in April 2014. In August 2014 he came to Louisiana State University to pursue graduate studies in Computer Science. He is currently a candidate for the degree of Doctor of Philosophy in Computer Science, which will be awarded in December 2019.