View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by The University of Utah: J. Willard Marriott Digital Library

USING SIMILARITY IN CONTENT AND ACCESS PATTERNS TO IMPROVE SPACE EFFICIENCY AND PERFORMANCE IN STORAGE SYSTEMS

by

Xing Lin

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

School of Computing

The University of Utah

December 2015

Copyright © Xing Lin 2015

All Rights Reserved

The University of Utah Graduate School

STATEMENT OF DISSERTATION APPROVAL

The dissertation of **Xing Lin**

has been approved by the following supervisory committee members:

Name	Role	Date Approved
Robert Ricci	, Chair	**8/17/2015**
Rajeev Balasubramonian	, Member	**7/22/2015**
Fred Douglis	, Member	**7/22/2015**
Feifei Li	, Member	**8/27/2015**
Jacobus Van der Merwe	, Member	**7/22/2015**

and by **Ross Whitaker**, Chair/Dean of

the Department/College/School of **Computing**

and by David B. Kieda, Dean of The Graduate School.

ABSTRACT

In the past few years, we have seen a tremendous increase in digital data being generated. By 2011, storage vendors had shipped 905 PB of purpose-built backup appliances. By 2013, the number of objects stored in Amazon S3 had reached 2 trillion. Facebook had stored 20 PB of photos by 2010. All of these require an efficient storage solution. To improve space efficiency, compression and deduplication are being widely used. Compression works by identifying repeated strings and replacing them with more compact encodings while deduplication partitions data into fixed-size or variable-size chunks and removes duplicate blocks. While we have seen great improvements in space efficiency from these two approaches, there are still some limitations. First, traditional compressors are limited in their ability to detect redundancy across a large range since they search for redundant data in a fine-grain level (string level). For deduplication, metadata embedded in an input file changes more frequently, and this introduces more unnecessary unique chunks, leading to poor deduplication. Cloud storage systems suffer from unpredictable and inefficient performance because of interference among different types of workloads.

This dissertation proposes techniques to improve the effectiveness of traditional compressors and deduplication in improving space efficiency, and a new IO scheduling algorithm to improve performance predictability and efficiency for cloud storage systems. The common idea is to utilize *similarity*. To improve the effectiveness of compression and deduplication, similarity in content is used to transform an input file into a compression- or deduplication-friendly format. We propose *Migratory Compression*, a generic data transformation that identifies similar data in a coarse-grain level (block level) and then groups similar blocks together. It can be used as a preprocessing stage for any traditional compressor. We find metadata have a huge impact in reducing the benefit of deduplication. To isolate the impact from metadata, we propose to separate metadata from data. Three approaches are presented for use cases with different constrains. For the commonly used tar format, we propose Migratory Tar: a data transformation and also a new tar format that deduplicates better. We also present a case study where we use deduplication to reduce

storage consumption for storing disk images, while at the same time achieving high performance in image deployment. Finally, we apply the same principle of utilizing similarity in IO scheduling to prevent interference between random and sequential workloads, leading to efficient, consistent, and predictable performance for sequential workloads and a high disk utilization.

CONTENTS

ABSTRACT . . . . **iii**

LIST OF TABLES . . . . **viii**

ACKNOWLEDGMENTS . . . . **ix**

CHAPTERS

1. INTRODUCTION **1**

1.1 Compression 2
1.2 Deduplication 4
1.2.1 Improve Deduplication by Separating Metadata From Data 5
1.2.2 Using Deduplication for Efficient Disk Image Deployment 5
1.3 Performance Efficiency and Predictability for Cloud Storage 7
1.4 Summary 8
1.5 Thesis Statement 8
1.6 Contributions 8

2. MIGRATORY COMPRESSION **10**

2.1 Overview 10
2.2 Introduction 10
2.3 Alternatives 14
2.3.1 Migratory versus Traditional Compression 14
2.3.2 Migratory versus Delta Compression 14
2.4 Approach 16
2.4.1 Single-file Migratory Compression 16
2.4.2 Intra-file Delta Compression 20
2.4.3 Migratory Compression in an Archival Storage System 21
2.5 Methodology 23
2.5.1 Metrics 23
2.5.2 Parameters Explored 24
2.5.3 Datasets 25
2.6 `mzip` Evaluation 25
2.6.1 Compression Effectiveness and Performance Tradeoff 25
2.6.2 Data Reorganization Throughput 27
2.6.3 Delta Compression 29
2.6.4 Sensitivity to Environment 29
2.7 Additional Use Cases of MC 34
2.7.1 Archival Migration in DDFS 34
2.7.2 File Transfer 37
2.8 Related Work 39

2.9 Summary 40

3. IMPROVE DEDUPLICATION BY SEPARATING METADATA FROM DATA 41

3.1 Overview 41
3.2 Introduction 41
3.3 Deduplication-friendly Formats 43
3.4 Application-level Postprocessing 44
3.4.1 Tar 44
3.4.2 Migratory Tar 46
3.4.3 Evaluation 47
3.5 Format-aware Deduplication 50
3.6 Related Work 51
3.7 Summary 52

4. USING DEDUPLICATING STORAGE FOR EFFICIENT DISK IMAGE DEPLOYMENT 53

4.1 Overview 53
4.2 Introduction 54
4.3 Foundation: Frisbee and Venti 56
4.3.1 Frisbee 57
4.3.2 Venti 60
4.4 Design and Implementation 61
4.4.1 Compression 61
4.4.2 Chunking 62
4.4.3 Block Size 65
4.4.4 Frisbee Chunk Construction 66
4.4.5 Design Summary 66
4.5 Evaluation 66
4.5.1 Frisbee Pipeline Measurements 67
4.5.2 The Impact of Compression 68
4.5.3 The Space/Time Trade-off 69
4.5.4 Delivering a Large Catalog 71
4.6 Discussion 76
4.7 Related Work 77
4.8 Summary 80

5. DIFFERENTIAL IO SCHEDULING FOR REPLICATING CLOUD STORAGE 81

5.1 Overview 81
5.2 Introduction 82
5.3 Interference Analysis 83
5.3.1 Benchmark with FIO 84
5.3.2 Higher-level experiments 86
5.3.3 Discussion 87
5.4 Differential IO Scheduling 88
5.4.1 Ceph Background 89
5.4.2 Deterministic Replica Placement 91

5.4.3 Workload Type Detection 92
5.5 Evaluation 92
5.6 Related Work 94
5.7 Summary 96

6. CONCLUSION 97

6.1 Summary of the Dissertation 97
6.2 Future Research Directions 99

REFERENCES 103

LIST OF TABLES

2.1 Dataset summary: size, deduplication factor of 8 KB variable chunking and compression ratios of standalone compressors. 26

2.2 Datasets used for archival migration evaluation. 35

3.1 Datasets for evaluating *mtar* 48

4.1 Average throughput rate and execution time of each stage in the baseline Frisbee pipeline. Throughput values are measured relative to the uncompressed data, except for network transfer, which reflects compressed data (with uncompressed rate in parentheses). 68

4.2 The effect of different Venti block sizes for deduplicating disk data. These storage figures are for disk data only, and do not account for image metadata. 70

4.3 Total storage space required for storing images in different repository formats, including metadata. `ndz` is for baseline Frisbee images stored in a filesystem. 70

5.1 The configuration of a d820 machine 93

ACKNOWLEDGMENTS

I would like to first thank my parents. Many parents prefer their children to go to a university that is close by, but my parents supported my decision to pursue my bachelor's degree in NanKai University, which is 872 miles away from home, and now, to pursue a PhD across the Pacific Ocean. Without their support, it would have been impossible for me to get such great opportunities to receive good education and to grow so far.

I also want to thank my wife, Liwei Fu. It is a great pleasure to meet you during this long journey. It is a big accomplishment to get married to you. Thank you for your encouragement whenever I feel down and thank you for your support behind me.

Many thanks to my advisor, Robert Ricci. The greatest advantage working with you is that you allow students to work on topics that they are interested in. Thanks to that, I found I really enjoyed the work I have done. Also thanks a lot for your efforts in teaching me how to do research, write papers and prepare talks. Thanks for leading me through this journey.

Special thanks to Fred Douglis. Thank you for offering the opportunities to work with you for two summers. I am missing the days I worked with you in the Princeton lab. Thank you for helping me get a paper at the USENIX Conference on File and Storage Technologies. Thanks to your collaboration, we have another paper at the 7th USENIX Workshop on Hot Topics in Storage and File Systems. You have greatly influenced my graduate study and it is great to have you as my mentor. Also thanks to my office colleagues at this lab. They included Philip Shilane, Hyong Shim, Stephen Smaldone, Grant Wallace, Ujwala Tulshigiri, and Cheng Li. Each of you made my summer days there memorable.

I want to thank Weibin Sun, my friend and office-mate. Thanks for discussing projects, offering advice and sharing your experience with me.

I want to thank Feifei Li, who has provided great advice and support for me.

I also want to thank Jacobus Van der Merwe, for providing feedback for the Migratory Compression presentation, and inputs for the cloud storage project. Also thanks for providing the opportunities to work on the TCloud project and the funding for my final year.

Thanks to Anton Burtsev for sharing your experience and providing advice.

Thanks to Mike Hilber for helping me with experiments and writing for the TridentCom paper and for solving all sorts of problems with Emulab.

Thanks to Eric Eide for interviewing and recruiting me into the Flux group. Thank you for writing up the Venti Frisbee paper while I was busy with implementation and evaluation. Thank you for organizing the Christmas party every year.

Thanks to Rajeev Balasubramonian for leading me to publish my first paper during my graduate study!

Thanks to Xudong Li for being my undergraduate advisor. Thanks for getting me interested in system research!

Thanks to Zhichao Li, a friend whom I met at USENIX FAST. Thanks for the great times we have had during that conference ever since 2011.

Thanks to Professor Christopher (Krzysztof) Sikorski for offering the Information Based Complexity seminar, even when I was the only student who registered. Thanks for making the Foundation of Computer Science course really interesting. I learned some interesting history of Poland during World War II, geography of Europe, and some great mathematicians.

Also thanks to many other friends with whom I have collaborated. They include Dallin Abendroth, Hyun-wook Baek, Jingxin Feng, Yuankai Guo, Jim Li, Richard Li, Guanlin Lu, Yun Mao, Raghuveer Pullakandam, and Gokul Soundararajan.

I also want to thank all the FLUX members at the University of Utah. They include Jonathon Duerig, David Johnson, Leigh Stoller, Kirk Webb, Gary Wong, and Dan Reading. Thanks for providing feedback on my projects, papers, and presentations. Thanks for offering helps whenever I need you.

Thanks for my other office-mates. They include Binh Nguyen, Mohamed Mehdi Jamshidy, Jithu Joseph, and Ren Quinn.

CHAPTER 1

INTRODUCTION

Digital data are growing exponentially, with the International Data Corporation (IDC) projecting that they will reach 44 ZB by 2020 [1]. By 2011, 905 PB of capacity had been shipped by purpose-built backup appliance vendors, and this number was projected to increase to be more than 8 EB by 2016 [2]. By February 2015, there were more than 37,000 public virtual machine images, provided and managed by the Amazon EC2 Cloud [3]. This trend is expected only to continue. How to store this rapidly growing amount of digital data efficiently becomes a big challenge for storage systems. In this dissertation, we propose two methods to improve compression and deduplication [4], a case study where deduplication is used to improve space efficiency in storing disk image and a new IO scheduling algorithm which improves performance predictability and efficiency.

Space efficiency is an important metric for storage systems. It measures how efficient a storage system is in *storing* information. A system is more space efficient if 1) more information can be stored for a given amount of storage space or 2) less storage space is required to store a certain amount of information. Two data reduction techniques are commonly used to improve space efficiency: *compression* and *deduplication*. Compression identifies repeated strings within a certain window size and represents them with more compact encodings. Deduplication works by partitioning the data into chunks, identifying duplicate chunks, and storing only one copy of the duplicates. By improving space efficiency, the effective size of a storage system can be improved and thus storage space requirements and cost can be reduced. Tuduce and Gross [5] showed that with compression, one can increase the effective main memory size from $2\times$ to $10\times$. It allows us to run applications with a smaller memory size that would otherwise be impossible. With the improvement of space efficiency, we can use fewer storage resources to store the same amount of data, enabling new possibilities in using storage techniques. Tapes were the primary storage media for storing backups and archives because they are cheaper than hard

drives. However, with deduplication, the storage space requirement for storing backups and archives can be reduced dramatically. Now it becomes economically feasible to replace tapes with hard drives for this use case [4]. Improved space efficiency also enables storage consolidation that reduces cost [6,7] and overhead in management, maintenance and power consumption [8–10]. With Nitro [11], a compressed and deduplicating Solid-State Drive (SSD) cache, it is possible to build a single storage system that can be used for both backup/archive workloads and for primary workloads (such as email servers or file servers). Space efficiency is thus an important property for storage systems, and there is a continuous demand for more efficient storage solutions.

While we have seen wide adoption of compression and deduplication, there are some limitations. Traditional compressors detect redundancy in a fine granularity: they try to detect redundancy in the string level. This approach fundamentally does not scale to find redundancy across a large range. Many file formats co-locate metadata with data. Metadata change more frequently, introducing many unnecessary unique chunks and leading to poor deduplication. To improve the effectiveness for compression and deduplication, we propose two data transformations. The key idea is to use *similarity* in data content. To improve the effectiveness of compression, we propose Migratory Compression: it identifies similarity in a coarse granularity, the chunk level, and then groups similar chunks together so that traditional compressors can detect more redundant strings with a small window (Chapter 2). To improve the effectiveness of deduplication, we propose to separate metadata blocks from data blocks (Chapter 3). For the *tar* format, we present Migratory Tar, a data transformation and a new tar format, which stores the same type of blocks together.

We further present a case study where we use deduplication, together with compression, to improve space efficiency in storing disk images (Chapter 4). Finally, we apply the same idea of using similarity in the area of disk IO scheduling. We propose Differential IO Scheduling, which schedules the same type of IO requests to the same disk, improving performance for cloud storage systems (Chapter 5).

1.1 Compression

Compression is a classic method which represents information with fewer bits than its original form, by exploiting redundancy in the information it represents. Among the most popular compression algorithms are the Lempel-Ziv (LZ) [12] compression methods. They

search for redundant content in string-level and work as follows: they scan a certain amount (called the window size) of the input data, identify repeated strings within that window, and encode repeated strings with the position of the first appearance of that string. By avoiding storing duplicate strings multiple times, the compressed form can represent the same amount of information as its original form but with less storage space. Compression has been widely used in the storage hierarchy, such as compressed main memory [5], compressed SSD caches [13], compressed file systems [14], and backup and archive storage systems [4].

For these compression algorithms, there is a trade-off between compressibility and computation overhead. The larger the window size, the greater the opportunity to find repeated strings, leading to better compression. However, a larger window size requires more computation and memory, leading to a longer runtime. In practice, most real-world implementations use small window sizes. For example, DEFLATE, used by `gzip`, has a 64 KB sliding window [15], and the maximum window for `bzip2` is 900 KB [16]. The only compression algorithm we are aware of that uses larger window sizes is LZMA in `7z` [17], which supports a window size up to 1 GB.[1] It usually compresses better than `gzip` and `bzip2` but takes significantly longer. Overall, since these traditional compressors search for redundant information *only* at the level of individual strings, they are limited in their capability in exploiting redundancy across a larger window size, leading to suboptimal compression.

To improve the compressibility for traditional compressors, we introduce a generic data transformation called *Migratory Compression* (Chapter 2). In contrast to traditional compression algorithms, it works at chunk [2] level: an input file is partitioned into chunks and then similar chunks are detected and grouped together. By grouping similar chunks together, it becomes easier for standard compressors to detect redundant strings, even with small window sizes. We evaluate Migratory Compression and find that it improves compressibility significantly, and sometimes runtime for traditional compressors.

---

[1]The specification of LZMA supports windows up to 4 GB, but we have not found a practical implementation for Linux that supports more than 1 GB, and use that number henceforth. One alternative compressor, xz [18], supports a window of 1.5 GB, but we found its decrease in throughput highly disproportionate to its increase in compression.

[2]We use 'chunk' or 'block' interchangeably in this dissertation

1.2 Deduplication

Compared to compression, deduplication is a more recent technique to remove redundant data. One early work was Venti [19], proposed by Quinlan and Dorward in 2002. Rather than searching for repeated information in string level as compression does, deduplication works at a coarse-grained chunk level, with a typical chunk size of a few kilobytes. To store input data, it partitions the input into chunks and calculates a fingerprint for each chunk, typically a cryptographic hash value, such as SHA1, based on chunk content. If two chunks match on their fingerprint, they are considered to be identical and only a single copy will be stored in the storage system. A fingerprint index is maintained to track all unique chunks stored in the system, and when storing new chunks, the fingerprint index is checked for duplication. Compression can be applied after the input data is deduplicated [4].

Much work has been done to make a deduplicating storage system efficient. Among them, many focus on improving read/write throughput in storing or retrieving data. For a large storage system, the fingerprint index can become too large to fit in main memory, and several approaches have been proposed to address this problem. They include using a Bloom filter structure [20] to quickly detect new unique chunks, and utilizing data locality across different generations of the same backup [4, 21]. Furthermore, some systems, such as SparseIndex [22], trade-off a small loss in space efficiency for better performance by deduplicating new data against only a subset of stored chunks (a small number of duplicate chunks are allowed in such systems). Another important problem arising with deduplication is fragmentation: because deduplication only stores a single copy of duplicate blocks, when a new file containing duplicate blocks is stored into the system, these duplicate blocks are not stored but refer to identical blocks which have been stored earlier. This leads to degraded sequential read performance for newly stored files. Several attempts have tried to address this problem, with the use of a rewrite algorithm [23, 24] to re-introduce duplicate blocks when writing a new file and garbage-collect duplicate blocks at a later time, or by building a better caching algorithm [25] using future access information that is available when reading a file.

With the previous work, one can build an efficient deduplicating storage system. However, there are still two questions that are unanswered. First, it is unclear how well current file formats work for deduplication. We examine a few popular backup and archive file

formats, including *tar*, and found these file formats are not optimal for deduplication. The second question we answer is whether it is possible to use a deduplicating storage system to store disk images without affecting the performance of disk image deployment. We provide answers to these two questions in this dissertation.

1.2.1 Improve Deduplication by Separating Metadata From Data

While much work has been done to improve read and write throughput for deduplicating storage systems, little has been done to study the impact of input data formats in deduplication. We found that many file formats in wide use suffer from an inherent design property that is incompatible with deduplication: they **intersperse metadata with data**. Metadata is changed more frequently and this introduces many unnecessary unique chunks, resulting in poor deduplication.

To isolate the interference between metadata and data blocks, we propose to separate metadata from data blocks (Chapter 3). Three approaches exist to achieve that goal. The first is to re-design file formats so that metadata is stored separately from data. This is the ideal case when the file format can be changed. When it is challenging to change a file format for compatibility reasons, a second approach is to postprocess an input file and transform it into a deduplication-friendly format that separates metadata from data. As a last resort, we can make the deduplicating storage systems aware of file formats, to identify metadata from data and treat it differently. In this dissertation, we will focus on the second approach with the *tar* format as a case study. We design a data transformation method called Migratory Tar, to transform a tar file into an *mtar* file. In the *mtar* format, data and metadata are stored at two separate places and the deduplication can be improved by more than 5×.

1.2.2 Using Deduplication for Efficient Disk Image Deployment

Compression and deduplication can improve space efficiency. However, they come with a cost: they may affect the runtime performance of an application. So, while it is important to improve space efficiency, it is also important that the use of compression and deduplication does not affect runtime performance. In Chapter 4, we present a case study where we use compression and deduplication to improve space efficiency in storing disk images for a high-performance disk image deployment system without negatively affecting

its runtime performance in image deployment.

Efficient disk image deployment is an important component for cloud platforms and network testbeds. In these environments, a *disk image* is used to instantiate compute instances. Each disk image ranges from a few to hundreds of GBs of space, and the complete category of disk images requires a large storage space. At the same time, to provide good user experience, it is important to instantiate compute instances quickly. This requires an efficient and scalable disk image deployment system that can distribute and install disk images quickly.

Frisbee [26] is the highly optimized disk image deployment system used in Emulab, a network testbed running at the University of Utah. It has a combinations of optimizations that make it efficient, such as the use of compression for efficient storage and image distribution, and using filesystem information to skip unallocated disk sectors when storing and installing a disk image. A Frisbee disk image file is composed of independently installable units. Each unit can be installed in parallel in a pipelined fashion. To get the highest possible performance, a key design in Frisbee is to install a disk image at the maximal disk write throughput and optimize all other stages in the pipeline. When using a deduplicating storage system for Frisbee, it could become a new bottleneck in the pipeline and affect its performance.

To answer the question whether it is feasible to use a deduplicating storage system for a high-performance disk image deployment system such as Frisbee, we have built a prototype system, called Venti-Frisbee (VF for short). The new system is designed in such a way that it does not break the key design in the original Frisbee and thus achieves comparable performance in image deployment while achieving storage space saving with the use of a deduplicating storage system. When using the Venti deduplicating storage system for Frisbee, we strike a balance between space saving from using Venti and its read performance in providing data for image distribution. We also explored where to do compression so that it does not affect image deployment performance and deduplication. Finally, we evaluate the end-to-end image deployment performance of the new system and show that it achieves a comparable performance to the original Frisbee system.

1.3 Performance Efficiency and Predictability for Cloud Storage

The previous sections focus on improving space efficiency with compression and deduplication by utilizing similarity in content. Now, we take a look at how we can use similarity in IO access patterns to improve performance predictability and efficiency for cloud storage systems.

With attracting benefits such as elasticity, agility, and massive economies of scale, many applications are migrating to cloud [27–29]. A key driver behind this is the development of virtualization techniques that enable efficient sharing of hardware resources. For storage, there are several types of service abstractions provided by cloud providers, including object stores (e.g., Amazon S3), block stores (e.g., Amazon EBS), and databases (e.g., Amazon RDS, Google Cloud SQL, and Microsoft SQL Azure). Among these cloud storage systems, replication is commonly used to improve reliability. The reason behind this is that in the large clusters that powers today's cloud computing, hardware component failures such as hard drive failures become common [30]. For example, both Google Filesystem (GFS) [31] and Ceph [32] use replication in their systems.

Because of the multitenant nature, the performance experienced by end-users in cloud storage environments varies unpredictably, sometimes more than an order of magnitude, compared with a dedicated cluster [33, 34]. For disk-based storage systems, when two or more tenants share the same physical disk, they compete for the disk head position for I/O accesses. For instance, random workloads from one tenant can destructively interfere with sequential workloads of another tenant [35], and reads may conflict with writes [36]. In order to provide predictable and efficient performance, some mechanism are required to isolate the interference between different types of workloads.

In this work, we focus on improving predictability and efficiency for read-intensive workloads, by minimizing the interference between random and sequential read requests. Random read requests have a destructive effect in reducing disk utilization and this leads to unpredictable bandwidths for sequential workloads. We propose Differential IO Scheduling (DIOS for short). The key idea is to utilize replication: in a replicating storage system, one could dedicate each replica for serving a particular type of request, either random or sequential. Since each replica now serves only one type of request, it separates random read requests from sequential ones and prevents interference between them. Replicas that serve

sequential requests can continue to provide high disk utilization. We implement DIOS in Ceph and evaluate its effectiveness. We find the new scheduling algorithm provides more predictable performance and higher disk utilization.

1.4 Summary

In this dissertation, we propose several techniques to improve space efficiency and performance for storage systems. The common idea across them is to utilize ***similarity***. For Migratory Compression and Migratory Tar, we use similarity in data content. Migratory Compression improves compression by grouping *similar* data blocks, while to improve deduplication for many data formats, we propose to separate metadata from data and store the *same* type of blocks together. We further explore the use of compression and deduplication to improve space efficiency when storing disk images for a high-performance disk image deployment system. Finally, we extend the same idea and apply it in IO scheduling. We propose Differential IO Scheduling, which schedules the *same* type of IO requests to the same disk. This isolates interference between different types of workloads, improving a system's throughput and performance predictability.

1.5 Thesis Statement

Similarity in content and access patterns can be utilized to improve space efficiency by storing similar data together, and performance predictability and efficiency by scheduling similar IO requests to the same hard drive.

1.6 Contributions

1. This dissertation proposes a new and generic data transformation technique called Migratory Compression. It can improve compressibility and sometimes runtime for traditional compressors (Chapter 2).

2. This dissertation reveals many file formats are not deduplication-friendly and proposes to separate data from metadata. It proposes three approaches to deal with this problem for use cases with different constraints. For the *tar* format, this dissertation proposes a data transformation method and a new tar format, called Migratory Tar. It improves deduplication for tar files (Chapter 3).

3. This dissertation exploits the similarity across disk images and uses data deduplication to improve storage efficiency for a high-performance disk image deployment system (Chapter 4).

4. This dissertation proposes a new chunk algorithm - Aligned Fixed-size Chunking (*AFC*), to do fixed-size chunking for deduplicating allocated data blocks for disk images (Chapter 4).

5. This dissertation proposes Differential IO Scheduling to improve performance predictability and efficiency for cloud storage systems (Chapter 5).

CHAPTER 2

MIGRATORY COMPRESSION

2.1 Overview

This chapter presents *Migratory Compression* (MC), a coarse-grained data transformation, to improve the effectiveness of traditional compressors in modern storage systems[1]. In MC, similar data chunks are re-located together to improve compression. After decompression, migrated chunks return to their previous locations. We evaluate the compression effectiveness and overhead of MC, explore reorganization approaches on a variety of datasets, and evaluate MC with different configurations, including four compressors, different compression levels, fixed- or variable-size chunking, and various chunk sizes. We present a prototype implementation of MC in a commercial deduplicating file system and evaluate effectiveness of MC in improving file transfer performance across slow networks. We also compare MC to the more established technique of delta compression, which is significantly more complex to implement within file systems.

We find that Migratory Compression improves compression effectiveness compared to traditional compressors by 11–105%, with relatively low impact on runtime performance. Frequently, adding MC to a relatively fast compressor like gzip results in compression that is more effective in both space and runtime than slower alternatives. When applying MC in archive storage, it improves gzip compression by 44–157%. MC can also reduce the network transfer time by up to 76% for distributing RPM packages. Most importantly, MC can be implemented in broadly used, modern file systems.

2.2 Introduction

Compression is a class of data transformation techniques to represent information with fewer bits than its original form by exploiting statistical redundancy. It is widely used in the storage hierarchy, such as compressed memory [5], compressed SSD caches [13],

[1]This work was published in USENIX FAST in 2014 [37].

file systems [14], and backup storage systems [4]. Generally, there is a tradeoff between computation and compressibility: often much of the available compression in a dataset can be achieved with a small amount of computation, but more extensive computation (and memory) can result in better data reduction [38].

There are various methods to improve compressibility, which largely can be categorized as increasing the *lookback* window and *reordering data*. Most compression techniques find redundant strings within a window of data; the larger the window size, the greater the opportunity to find redundant strings, leading to better compression. However, to limit the overhead in finding redundancy, most real-world implementations use small window sizes. For example, DEFLATE, used by `gzip`, has a 64 KB sliding window [15] and the maximum window for `bzip2` is 900 KB [16]. The only compression algorithm we are aware of that uses larger window sizes is LZMA in `7z` [17], which supports history up to 1 GB.[2] It usually compresses better than `gzip` and `bzip2` but takes significantly longer. Some other compression tools, such as `rzip` [39], find identical sequences over a long distance by computing hashes over fixed-sized blocks and then rolling hashes over blocks of that size throughout the file; this effectively does intrafile deduplication but cannot take advantage of small interspersed changes. Delta compression (DC) [40] can find small differences in similar locations between two highly similar files. While this enables highly efficient compression between similar files, it cannot delta-encode widely dispersed regions in a large file or set of files without targeted pair-wise matching of similar content [41].

Data reordering is another way to improve compression: since compression algorithms work by identifying repeated strings, one can improve compression by grouping similar characters together. The Burrows-Wheeler Transform (BWT) [42] is one such example that works on relatively small blocks of data: it permutes the order of the characters in a block, and if there are substrings that appear often, the transformed string will have single characters repeat in a row. BWT is interesting because the operation to invert the transformed block to obtain the original data requires only that an index be stored with the transformed data and that the transformed data be sorted in the lexicographical order to

[2]The specification of LZMA supports windows up to 4 GB, but we have not found a practical implementation for Linux that supports more than 1 GB and use that number henceforth. One alternative compressor, xz [18], supports a window of 1.5 GB, but we found its decrease in throughput highly disproportionate to its increase in compression.

use the index to identify the original contents. bzip2 uses BWT as the second layer in its compression stack.

What we propose is, in a sense, a coarse-grained BWT over a large range (typically tens of GBs or more). We call it *Migratory Compression* (MC) because it tries to rearrange data to make it more compressible, while providing a mechanism to reverse the transformation after decompression. Unlike BWT, however, the unit of movement is kilobytes rather than characters, and the scope of movement is an entire file or group of files. Also, the recipe to reconstruct the original data is a nontrivial size, though still only ~0.2% of the original file.

With MC, data is first partitioned into chunks. Then we 'sort' chunks so that similar chunks are grouped and located together. Duplicate chunks are removed and only the first appearance of that copy is stored. Standard compressors are then able to find repeated strings across adjacent chunks.[3] Thus MC is a *preprocessor* that can be combined with arbitrary adaptive lossless compressors such as gzip, bzip2, or 7z; if someone invented a better compressor, MC could be integrated with it via simple scripting. We find that MC improves gzip by up to a factor of two on datasets with high rates of similarity (including duplicate content), usually with better performance. Frequently gzip with MC compresses both better and faster than other off-the-shelf compressors like bzip2 and 7z at their default levels.

We consider three principal use cases of Migratory Compression:

- **mzip** is a term for using MC to compress a single file. With mzip, we extract the resemblance information, cluster similar data, reorder data in the file, and compress the reordered file using an off-the-shelf compressor. The compressed file contains the recipe needed to restore the original contents after decompression. The bulk of our evaluation is in the context of stand-alone file compression; henceforth mzip refers to integrating MC with traditional compressors (gzip by default unless stated otherwise).

[3]It is possible for an adaptive compressor's history to be smaller than size of two chunks, in which case it will not be able to take advantage of these adjacent chunks. For instance, if the chunks were 64 KB, gzip would not match the start of one chunk against the start of the next chunk. By making the chunk size small relative to the compressor's window size, we avoid such issues.

- **Archival** involves data migration from backup storage systems to archive tiers, or data stored directly in an archive system such as Amazon Glacier [43]. Such data are cold and rarely read, so the penalty resulting from distributing a file across a storage system may be acceptable. We have prototyped MC in the context of the archival tier of the Data Domain File System (DDFS) [4].

- **File transfer** over low-bandwidth networks can benefit from better compression, even if that compression is relatively slow, and `mzip` goes beyond strict deduplication of systems like LBFS [44]. Our experiments show that `mzip` can lead to a shorter end-to-end transfer time than traditional compression, even including the overhead of reorganizing data.

There are two runtime overheads for MC. One is to detect similar chunks: this requires a preprocessing stage to compute *similarity features* for each chunk, followed by clustering chunks that share these features. The other overhead comes from the large number of I/Os necessary to reorganize the original data, first when performing compression and later to transform the uncompressed output back to its original contents. We quantify the effectiveness of using fixed-size or variable-size chunks, three chunk sizes (2 KB, 8 KB and 32 KB), and different numbers of features, which trade compression against runtime overhead. For the data movement overhead, we evaluate several approaches as well as the relative performance of hard disks and solid state storage.

In summary, our work makes the following contributions. First, we propose *Migratory Compression*, a new data transformation algorithm. Second, we evaluate its effectiveness with real-world datasets, quantify the overheads introduced, and evaluate three data reorganization approaches with both HDDs and SSDs. Third, we compare `mzip` with delta compression and show that these two techniques are comparable, though with different implementation characteristics. Last, we demonstrate its effectiveness with two additional use cases. The first is to use MC to improve space efficiency for the archive tier within a deduplicating storage system, DDFS; in the second use case, we look into improving network transfer for distributing RPM packages with MC.

2.3 Alternatives

One goal of any compressor is to distill data into a minimal representation. Another is to perform this transformation with minimal resources (computation, memory, and I/O). These two goals are largely conflicting, in that additional resources typically result in better compression, though frequently with diminishing returns [38]. Here we consider two alternatives to spending extra resources for better compression: moving similar data together and delta-compressing similar data in place.

2.3.1 Migratory versus Traditional Compression

Figure 2.1 compares traditional compression and Migratory Compression. The blue chunks at the end of the file (*A'* and *A"*) are similar to the blue chunk at the start (*A*), but they have small changes that keep them from being entirely identical. With (a) traditional compression, there is a limited window over which the compressor will look for similar content, so *A'* and *A"* later in the file do not get compressed relative to *A*. With (b) MC, we move these chunks to be together, followed by two more similar chunks, *B* and *B'*. Note that the two green chunks labeled *D* are identical rather than merely similar, so the second is replaced by a reference to the first.

One question is whether we could simply obtain extra compression by increasing the window size of a standard compressor. We see later (Section 2.6.4.4) that the "maximal" setting for 7z, which uses a 1 GB lookback window (and a memory footprint over 10 GB) and substantial computation, often results in worse compression with poorer throughput than the default 7z setting integrated with MC.

2.3.2 Migratory versus Delta Compression

Another obvious question is how MC compares to a similar technology, *delta compression* (DC) [40]. The premise of DC is to encode an object A' relative to a similar object A, and it is effectively the same as compressing A, discarding the output of that compression, and using the compressor state to continue compressing A'. Anything in A' that repeats content in A is replaced by a reference to its location in A, and content within A' that repeats previous content in A' can also be replaced with a reference.

When comparing MC and DC, there are striking similarities because both can use features to identify similar chunks. These features are compact (64 bytes per 8 KB chunk by

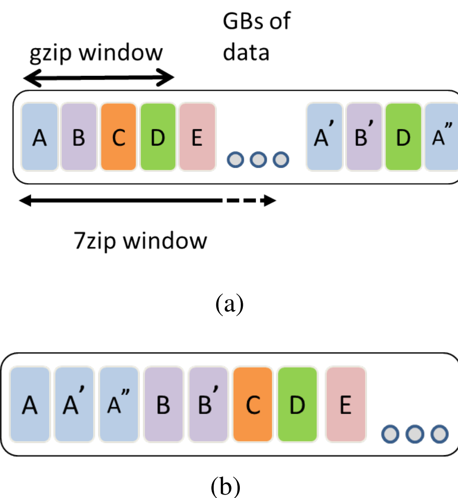


Figure 2.1. Compression alternatives. a) traditional compression. b) migratory compression. With MC similar data moves close enough together to be identified as redundant, using the same compression window.

default), allowing GBs or even TBs of data to be efficiently searched for similar chunks. Both techniques improve compression by taking advantage of redundancy between similar chunks: MC reads the chunks and writes them consecutively to aid standard compressors, while DC reads two similar chunks and encodes one relative to the other. We see in Section 2.6.3 that MC generally improves compression and has faster performance than intrafile DC, but these differences are rather small and could be related to internal implementation details.

One area where MC is clearly superior to DC is in its simplicity, which makes it compatible with numerous compressors and eases integration with storage systems. Within a storage system, MC is a nearly seamless addition, since all of the content still exists after migration—it is simply at a different offset than before migration. For storage systems that support indirection, such as deduplicated storage [4], MC causes few architectural changes, though it likely increases fragmentation. On the other hand, DC introduces dependencies between data chunks that span the storage system: storage functionality has to be modified to handle indirections between delta compressed chunks and the *base* chunks against which they have been encoded [45]. Such modifications affect such system features as garbage collection, replication, and integrity checks.

2.4 Approach

Much of the focus of our work on Migratory Compression is in the context of reorganizing and compressing a single file (`mzip`), described in Section 2.4.1. We also use `mzip` to optimize network transfer for distributing RPM packages. In addition, we compare `mzip` to in-place delta-encoding of similar data (Section 2.4.2) and discuss how we implement data reorganization during migration to an archival tier within DDFS[4] (Section 2.4.3).

2.4.1 Single-file Migratory Compression

The general idea of MC is to partition data into chunks and reorder them to store similar chunks sequentially, increasing compressors' opportunity to detect redundant strings and leading to better compression. For standalone file compression, this can be added as a preprocessing stage, which we term `mzip`. A reconstruction process is needed as a postprocessing stage in order to restore the original file after decompression.

2.4.1.1 Similarity Detection with Super-features

The first step in MC is to partition the data into chunks. These chunks can be fixed-size or variable-size "content-defined chunks." Prior work suggests that, in general, variable-size chunking provides a better opportunity to identify duplicate and similar data [41]; however, virtual machines use fixed-sized blocks, and deduplicating VM images potentially benefits from fixed-sized blocks [46]. We default to variable-sized chunks based on the comparison of fixed-sized and variable-sized units below.

One big challenge to doing MC is to identify similar chunks efficiently and scalably. A common practice is to generate *similarity features* for each chunk. To generate a feature for a chunk, we use a hash function with a rolling window: for every possible overlapping window, we calculate a value. Among these values, we pick a special one (for example, the maximal value) as the feature from that hash. To reduce false positive in detecting similar blocks, we use a few hash functions to generate many features. Two chunks are likely to be similar if they share multiple features. While it is possible to enumerate the closest matches by comparing all features, a useful approximation is to group sets of features into *super-features* (SFs): two data objects that have a single SF in common are likely to be

---

[4] Guanlin implemented and evaluated data reorganization for archive migration in DDFS.

fairly similar [47]. This approach has been used numerous times to successfully identify similar web pages, files, and/or chunks within files [41, 45, 48]. In this work, we use an existing tool from Shilane's previous work [45] to generate super-features.

Four superfeatures are generated for each chunk. We adopt the "FirstFit" approach of Kulkarni et al. [41], which we will term the *greedy* matching algorithm. Each time a chunk is processed, its *N* SFs are looked up in *N* hash tables, one per SF. If any SF matches, the chunk is associated with the other chunks sharing that SF (i.e., it is added to a list and the search for matches terminates). If no SF matches, the chunk is inserted into each of the *N* hash tables so that future matches can be identified.

We explored other options, such as sorting all chunks on each of the SFs to look for chunks that match several SFs rather than just one. Across the datasets we analyzed, this sort marginally improved compression, but the computational overhead was disproportionate. Note, however, that applying MC to a file that is so large that its metadata (fingerprints and SFs) is too large to process in memory would require some out-of-core method, such as sorting.

2.4.1.2 Data Migration and Reconstruction

Given information about which chunks in a file are similar, our `mzip` preprocessor outputs two recipes: *migrate* and *restore*. The migrate recipe contains the chunk order of the reorganized file: chunks identified to be similar are located together, ordered by their offset within the original file. (That is, a later chunk is moved to be adjacent to the first chunk it is similar to.) The restore recipe contains the order of chunks in the reorganized file and is used to reconstruct the original file. Generally, the overhead of generating these recipes is orders of magnitude less than the the overhead of physically migrating the data stored in disk.

Figure 2.2 presents a simplified example of these two procedures, assuming fixed chunk sizes. We show a file with a sequence of chunks A through D, including A' and B' to indicate chunks that are similar to A and B, respectively. The reorganized file places A' after A and B' after B, so the *migrate* recipe specifies that the reorganized file consists of chunk 0 (A), chunk 3 (A'), chunk 1 (B), chunk 5 (B'), and so on. The *restore* recipe shows that to obtain the original order, we output chunk 0 (A), chunk 2 (B), chunk 4 (C), chunk 1 (A'), etc. from the reorganized file. (For variable-length chunks, the recipes contain byte

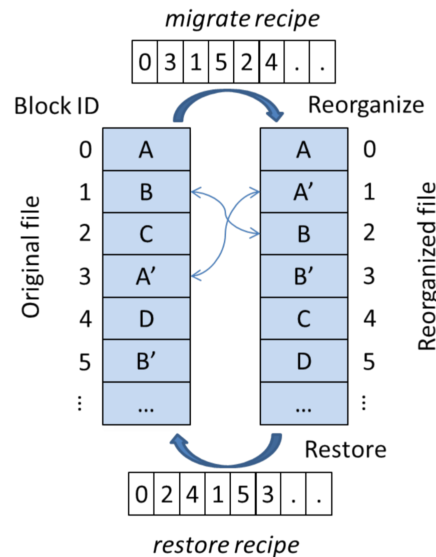


Figure 2.2. An example of the reorganization and restore procedures.

offsets and lengths rather than block offsets.)

Once we have the migrate recipe and the restore recipe, we can create the reorganized file. Reorganization (migration) and reconstruction are complements of each other, each moving data from a specific location in an input file to a desired location in the output file. (There is a slight asymmetry resulting from deduplication, as completely identical chunks can be omitted completely in the reorganized file, then copied 1-to-N when reconstructing the original file.)

There are several methods for moving chunks.

- **In-Memory** When the original file can fit in memory, we can read in the whole file into memory and output chunks in the reorganized order sequentially. We call this the 'in-memory' approach.

- **Chunk-level** When we cannot fit the original file in memory, the simplest way to reorganize a file is to scan the chunk order in the migrate recipe: for every chunk needed, seek to the offset of that chunk in the original file, read it, and output it to the reorganized file. When using HDDs, this could become very inefficient because of the number of random I/Os involved.

- **Multi-pass** We also designed a 'multipass' algorithm, which scans the original file repeatedly from start to finish; during each pass, chunks in a particular *reorg range* of the reorganized file are saved in a memory buffer while others are discarded. At the end of each pass, chunks in the memory buffer are output to the reorganized file and the reorg range is moved forward. This approach replaces random I/Os with multiple scans of the original file. (Note that if the file fits in memory, the in-memory approach is the multipass approach with a single pass.)

Our experiments in Section 2.6.2 show that the in-memory approach is best, but when memory is insufficient, the multipass approach is more efficient than chunk-level. We can model the relative costs of the two approaches as follows. Let T be elapsed time, where T_{mp} is the time for multipass and T_c is the time when using individual chunks. Focusing only on I/O costs, T_{mp} is the time to read the entire file sequentially N times, where N is the number of passes over the data. If disk throughput is D and the file size is S, $T_{mp} = S*N/D$. For a size of 15GB, 3 passes, and 100MB/s throughput, this works out to 7.7 minutes for I/O. If CS represents the chunk size, the number of chunk-level I/O operations is S/CS and the elapsed time is $T_c = \frac{S/CS}{IOPS}$. For a disk with 100 IOPS and an 8KB chunk size, this equals 5.4 hours. Of course there is some locality, so what fraction of I/Os must be sequential or cached for the chunk approach to break even with the multipass one? If we assume that the total cost for chunk-level is the cost of reading the file once sequentially[5] plus the cost of random I/Os, then we solve for the cost of the random fraction (RF) of I/Os equaling the cost of $N-1$ sequential reads of the file:

$$S*(N-1)/D = \frac{S*RF/CS}{IOPS}$$

Solving this equation, we have the following.

$$RF = \frac{(N-1)*IOPS*CS}{D}.$$

In the example above, this works out to $\frac{16}{1024} = 1.6\%$: if more than 1.6% of the data has dispersed similarity matches, then the multipass method should be preferred.

---

[5]Note that if there are so many random I/Os that we do not read large sequential blocks, T_{mp} is reduced by a factor of $1-RF$.

Solid-state disks, however, offer a good compromise. Using SSDs to avoid the penalty of random I/Os on HDDs causes the chunk approach to come closer (up to 90%) to the in-memory performance.

2.4.1.3 `mzip` Workflow

Figure 2.3 presents the compression and decompression workflows in `mzip`. Compression/decompression and segmentation are adopted from existing tools, while similarity detection and reorganization/restoration are specially developed and highlighted in red. The original file is read once by the segmenter, computing cryptographically secure fingerprints (for deduplication) and resemblance features, then it is read again by the reorganizer to produce a file for compression. (This file may exist only as a pipeline between the reorganizer and the compressor, not separately stored, something we did for all compressors but `rzip`, as it requires the ability to *seek*.) To restore the file, the compressed file is decompressed and its restore recipe is extracted from the beginning of the resulting file. Then the rest of that file is processed by the restorer, in conjunction with the recipe, to produce the original content.

2.4.2 Intra-file Delta Compression

When applied in the context of a single file, we hypothesized that `mzip` would be slightly better than delta compression (DC) because its compression state at the time the similar chunk is compressed includes content from many KBs-MBs of data, depending on the compressor. To evaluate how `mzip` compares with DC within a file, we implemented a version of DC that uses the same workflows as `mzip`, except the ‘reorganizer’ and the ‘restorer’ in `mzip` are replaced with a ‘delta-encoder’ and a ‘delta-decoder.’ The delta-encoder encodes each similar chunk as a delta against a base chunk, while the delta-decoder reconstructs a chunk by patching the delta to its base chunk. (In our implementation, the chunk which appears first in the file is selected as the base for each group of similar chunks. We use `xdelta` [49] for encoding, relying on the later compression pass to compress anything that has not been removed as redundant.)

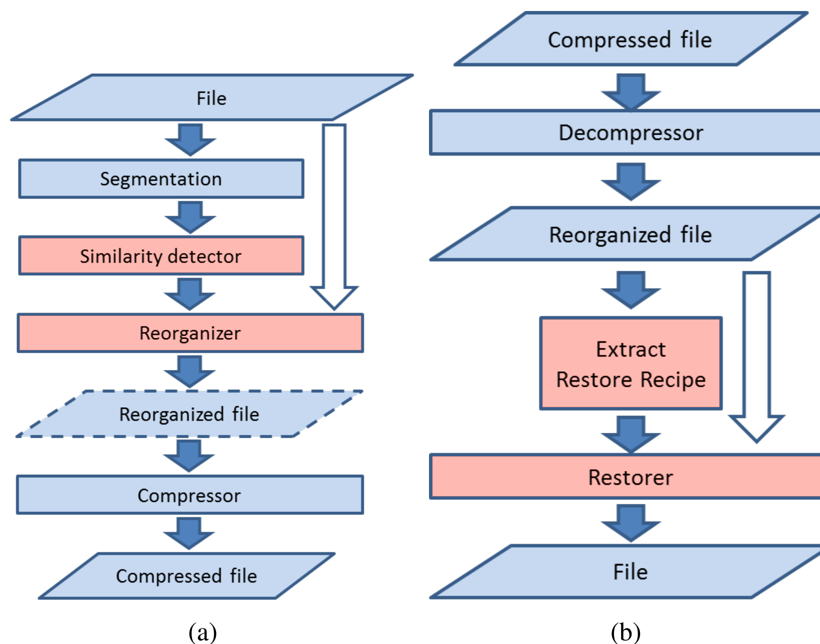


Figure 2.3. Workflow in Migratory Compression. a) Compression workflow. b) Decompression workflow.

2.4.3 Migratory Compression in an Archival Storage System

In addition to reorganizing the content of individual files, MC is well suited for reducing data requirements within an entire file system. However, this impacts read locality, which is already an issue for deduplicating storage systems [25]. This performance penalty therefore makes it a good fit for systems with minimal requirements for read performance. An archival system, such as Amazon Glacier [43], is a prime use case, as much of its data will not be reread; when it is, significant delays can be expected. When the archival system is a tier within a backup environment, such that data moves in bulk at regular intervals, the data migration is an opportune time to migrate similar chunks together.

To validate the MC approach in a real storage system, we implemented a prototype using the existing deduplicating Data Domain Filesystem (DDFS) [4]. After deduplication, chunks in DDFS are aggregated into *compression regions* (CRs), which in turn are aggregated into *containers*. DDFS can support two storage tiers: an *active* tier for backups and a long-term retention tier for *archival*; while the former stores the most recent data within a time period (e.g., 90 days), the latter stores the relatively 'cold' data that needs to

be retained for an extended time period (e.g., 5 years) before being deleted. Data migration is important for customers who weigh the dollar-per-GB cost over the migrate/retrieval performance for long-term data.

A daemon called *data migration* is used to migrate selected data periodically from the active tier to the archive tier. For performance reasons, data in the active tier is compressed with a simple LZ algorithm while we use `gzip` in the archive tier for better compression. Thus, for each file to be migrated in the namespace, DDFS reads out the corresponding compression regions from the active tier, decompresses each, and recompresses with `gzip`.

The MC technique would offer customers a further tradeoff between the compression ratio and migrate/retrieval throughput. It works as follows:

- **Similarity Range.** Similarity detection is limited to files migrated in one iteration, for instance all files written in a span of two weeks or 90 days.

- **Super-features.** We use 12 similarity features, combined as 3 SFs. For each container to be migrated, we read out its metadata region, extract the SFs associated with each chunk, and write these to a file along with the chunk's fingerprint.

- **Clustering.** Chunks are grouped in a similar fashion to the greedy single SF matching algorithm described in Section 2.4.1.1, but via sorting rather than a hash table.

- **Data reorganization.** Similar chunks are written together by collecting them from the container set in multiple passes, similar to the single-file multipass approach described in Section 2.4.1.2 but without a strict ordering. Instead, the passes are selected by choosing the largest clusters of similar chunks in the first one-third, then smaller clusters, and finally dissimilar chunks. Since chunks are grouped by any of 3 SFs, we use 3 Bloom filters, respectively, to identify which chunks are desired in a pass. We then copy the chunks needed for a given pass into the CR designated for a given chunk's SF; the CR is flushed to disk if it reaches its maximum capacity.

Note that DDFS already has the notion of a mapping of a file identifier to a tree of chunk identifiers, and relocating a chunk does not affect the chunk tree associated with a file. Only the low-level index mapping a chunk fingerprint to a location in the storage system need be updated when a chunk is moved. Thus, there is no notion of a *restore recipe* in the DDFS case, only a recipe specifying which chunks to co-locate.

In theory, MC could be used in the backup tier as well as for archival: the same mechanism for grouping similar data could be used as a background task. However, the impact on data locality would not only impact read performance [25], it could degrade ingest performance during backups by breaking the assumptions underlying data locality: DDFS expects an access to the fingerprint index on disk to bring nearby entries into memory [4].

2.5 Methodology

We discuss evaluation metrics in Section 2.5.1, tunable parameters in Section 2.5.2, and datasets in Section 2.5.3.

2.5.1 Metrics

The high-level metrics by which to evaluate a compressor are the **compression factor** (CF) and the **resource usage** of the compressor. CF is the ratio of an original size to its compressed size, i.e., higher CFs correspond to more data eliminated through compression; deduplication ratios are analogous.

In general, resource usage equates to processing time per unit of data, which can be thought of as the **throughput** of the compressor. There are other resources to consider, such as the required memory: in some systems memory is plentiful and even the roughly 10 GB of DRAM used by `7z` with its maximum 1 GB dictionary is fine; in some cases the amount of memory available or the amount of compression being done in parallel results in a smaller limit.

Evaluating the performance of a compressor is further complicated by the question of parallelization. Some compressors are inherently single-threaded while others support parallel threads. Generally, however, the fastest compression is also single-threaded (e.g., `gzip`), while a slower but more effective compressor such as `7z` is slower despite its multiple threads. We consider end-to-end time, not CPU time.

Most of our experiments were run inside a virtual machine, hosted by an ESX server with 2×6 Intel 2.67GHz Xeon X5650 cores, 96 GB memory, and 1-TB 3G SATA 7.2k 2.5in drives. The VM is allocated 90 GB memory except in cases when memory is explicitly limited, as well as 8 cores and a virtual disk with a 100 GB ext4 partition on a two-disk RAID-1 array. For 8 KB random accesses, we have measured 134 IOPS for reads (as well

as 385 IOPS for writes, but we are not evaluating random writes), using a 70 GB file and an I/O queue depth of 1. For 128 KB sequential accesses, we measured 108 MB/s for reads and 80 MB/s for writes. The SSD used is a Samsung Pro 840, with 22K IOPS for random 8 KB reads and 264 MB/s for 128 KB sequential reads (write throughputs become very low because there is no TRIM support in the hypervisor: 20 MB/s for 128 KB sequential writes). To minimize performance variation, all other virtual machines were shut down except those providing system services. Each experiment was repeated three times; we report averages. We do not plot error bars because the vast majority of experiments have a relative standard error under 5%; in a couple of cases, decompression timings vary with 10–15% relative error.

To compare the complexity of MC with other compression algorithms, we ran most experiments in-memory. In order to evaluate the extra I/O necessary when files do not fit in memory, some experiments limit memory size to 8 GB and use either an SSD or hard drive for I/O.

The tool that computes chunk fingerprints and features is written in C, while the tools that analyze that data to cluster similar chunks and reorganize the files are written in Perl. The various compressors are off-the-shelf Linux tools installed from repositories.

2.5.2 Parameters Explored

In addition to varying the workload by evaluating different datasets, we consider the effect of a number of parameters. Defaults are shown in **bold**.

- **Compressor.** We consider **gzip**, `bzip2`, `7z`, and `rzip`, with or without MC.
- **Compression tuning.** Each compressor can be run with a parameter that trades off performance against compressibility. We use the **default parameters** unless specified otherwise.
- **MC chunking.** Are chunks fixed or **variable** sized?
- **MC chunk size.** How large are chunks? We consider 2, **8**, and 32 KB; for variable-sized chunks, these represent target averages.
- **MC resemblance computation.** How are super-features matched? (Default: **Four SFs**, matched greedily, one SF at a time.)

- MC **data source.** When reorganizing an input file or reconstructing the original file after decompression, where is the input stored? We consider an **in-memory file system**, SSD, and hard disk.

2.5.3 Datasets

Table 2.1 summarizes salient characteristics of the input datasets used to test `mzip`, two types of backups and a pair of virtual machine images. Each entry shows the total size of the file processed, its deduplication ratio (half of them can significantly boost their CF using MC simply through deduplication), and the CF of the four off-the-shelf compressors. We find that `7z` and `rzip` both compress significantly better than the others and are similar to each other.

- We use four single backup image files taken from production deduplication backup appliances. Two are backups of workstations while the other two are backups of Exchange email servers.

- We use two virtual machine disk images consisting of VMware VMDK files. One has Ubuntu 12.04.01 LTS installed while the other uses Fedora Core release 4 (a dated but stable build environment).

2.6 `mzip` Evaluation

The most important consideration in evaluating MC is whether the added effort to find and relocate similar content is justified by the improvement in compression. Section 2.6.1 compares CF and throughput across the six datasets. Section 2.6.2 looks specifically at the throughput when memory limitations force repeated accesses to disk and finds that SSDs would compensate for random I/O penalties. Section 2.6.3 compares `mzip` to a similar intra-file DC tool. Finally, Section 2.6.4 considers additional sensitivity to various parameters and configurations.

2.6.1 Compression Effectiveness and Performance Tradeoff

Figure 2.4 plots compression throughput versus compression factor, using the six datasets. All I/O was done using an in-memory file system. Each plot shows eight points, four for the off-the-shelf compressors (`gzip`, `bzip2`, `7z`, and `rzip`) using default settings and four for these compressors using MC.

Table 2.1. Dataset summary: size, deduplication factor of 8 KB variable chunking and compression ratios of standalone compressors.

Dataset		Size (GB)	Dedupe (X)	Compression Factor of Standalone Compressors (X)			
Type	Name			gzip	bzip2	7z	rzip
Workstation Backup	WORKSTATION1	17.36	1.69	2.70	3.22	4.44	4.46
	WORKSTATION2	15.73	1.77	2.32	2.61	3.16	3.12
Email Server Backup	EXCHANGE1	13.93	1.06	1.83	1.92	3.35	3.99
	EXCHANGE2	17.32	1.02	2.78	3.13	4.75	4.79
VM Image	Ubuntu-VM	6.98	1.51	3.90	4.26	6.71	6.69
	Fedora-VM	27.95	1.19	3.21	3.49	4.22	3.97

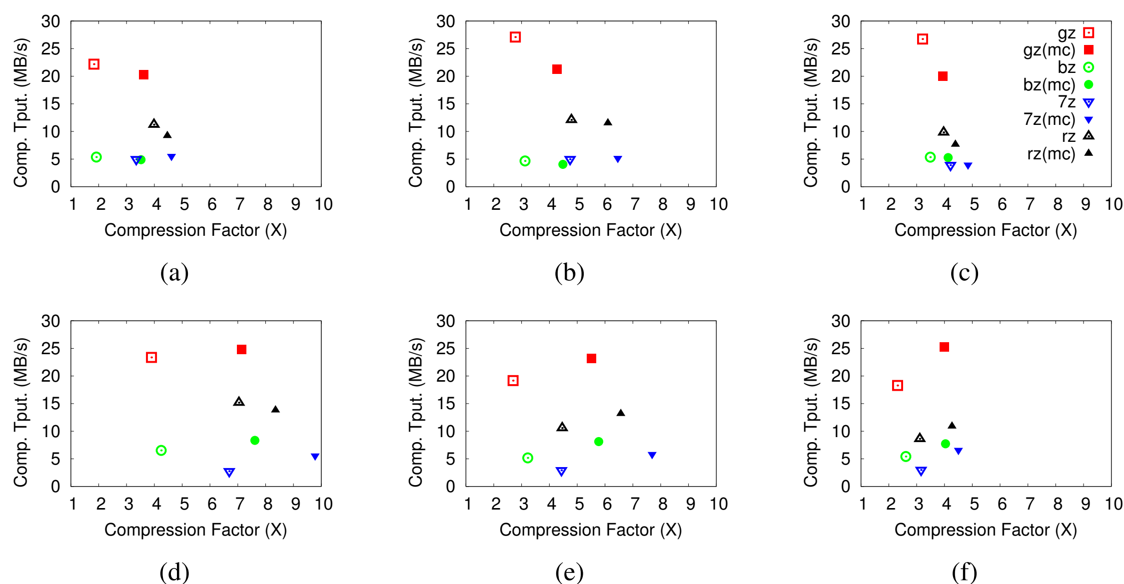


Figure 2.4. Compression throughput vs. compression factor for all datasets, using unmodified compression or MC, for four compressors, one figure per dataset. (a) EXCHANGE1. (b) EXCHANGE2. (c) Fedora-VM. (d) Ubuntu-VM. (e) WORKSTATION1. (f) WORKSTATION2. The legend for all plots appears in (c)

Generally, adding MC to a compressor significantly improves the CF (23–105% for gzip, 18–84% for bzip2, 15–74% for 7z and 11–47% for rzip). It is unsurprising that rzip has the least improvement, since it already finds duplicate chunks across a range of a file, but MC further increases that range. Depending on the compressor and dataset, throughput may decrease moderately or it may actually improve as a result of the compressor getting (a) deduplicated and (b) more compressible input. We find that 7z with MC always gets the highest CF, but often another compressor gets nearly the same compression with better throughput. We also note that in general, for these datasets, off-the-shelf rzip compresses just about as well as off-the-shelf 7z but with much higher throughput. Better, though, the combination of gzip and MC has a comparable CF to any of the other compressors without MC, and with still higher throughput, making it a good choice for general use.

Figure 2.5 shows decompression throughput versus CF for all datasets. Decompression performance may be more important than compression performance for use cases where something is compressed once but uncompressed many times. From these figures, we can see decompression throughputs are improved for most compressors. It is likely because deduplication leads to less data to decompress. For EXCHANGE1 and EXCHANGE2, CF improves substantially as well, with throughput not greatly affected. Only for Fedora-VM does gzip decompression throughput decrease in any significant fashion (from about 140 MB/s to 120).

2.6.2 Data Reorganization Throughput

To evaluate how mzip may work when a file does not fit in memory, we experimented with a limit of 8 GB RAM when the input data is stored in either a solid state disk (SSD) or hard disk drive (HDD). The output file is stored in the HDD. When reading from HDD, we evaluated two approaches: chunk-level and multipass. Since SSD has no random-access penalty, we use only chunk-level and compare SSD to in-mem.

Figure 2.6 shows the compression throughputs for SSD-based and HDD-based mzip (Henceforth mzip refers to gzip + MC.).

We can see that SSD approaches in-memory performance, but as expected, there is a significant reduction in throughput using the HDD. This reduction can be mitigated by the multipass approach. For instance, using a reorg range of 60% of memory, 4.81 GB, if the

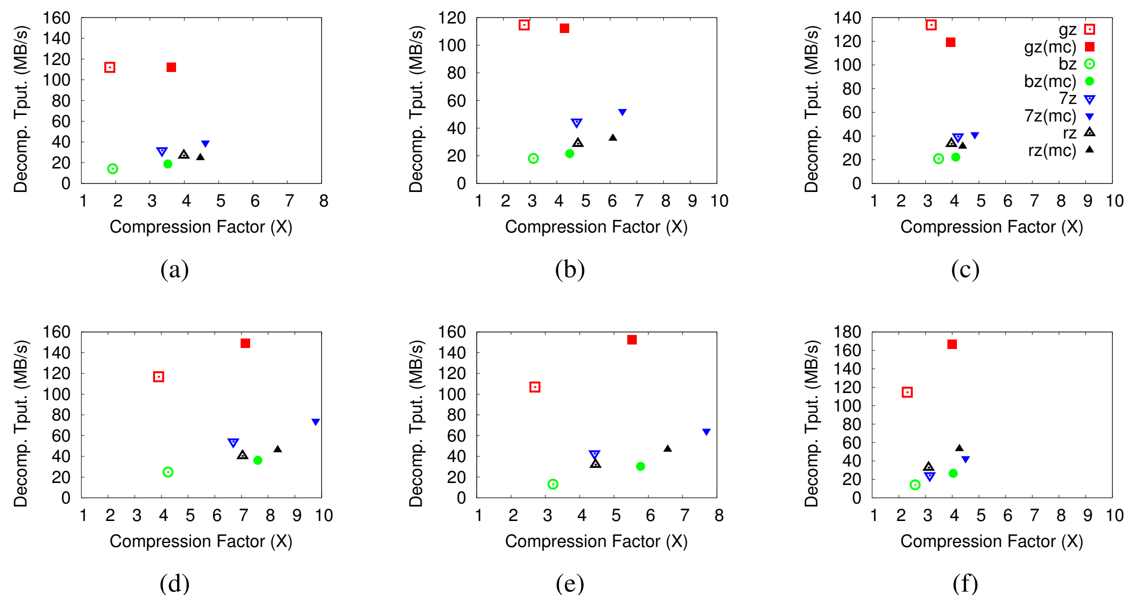


Figure 2.5. Decompression throughput vs. compression factor for all datasets, using unmodified compression or MC, one figure per dataset. (a) EXCHANGE1. (b) EXCHANGE2. (c) Fedora-VM. (d) Ubuntu-VM. (e) WORKSTATION1. (f) WORKSTATION2. The legend for all plots appears in (d)

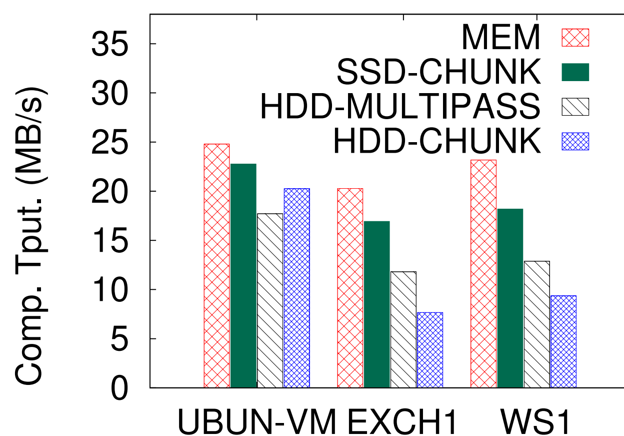


Figure 2.6. Compression throughput comparison for HDD-based or SSD-based `gzip` (MC).

file does not fit in memory, the throughput can be improved significantly for HDD-based mzip by comparison to accessing each chunk in the order it appears in the reorganized file (and paying the corresponding costs of random I/Os).

Note that Ubuntu-VM can approximately fit in available memory, so the chunk-level approach performs better than multipass: multipass reads the file sequentially twice, while chunk-level can use OS-level caching.

2.6.3 Delta Compression

Figure 2.7 compares the compression and performance achieved by mzip to compression using in-place delta-encoding,[6] as described in Section 2.4.2. Both use gzip as the final compressor. Figure 2.7(a) shows the CF for each dataset, broken down by the contribution of each technique. The bottom of each stacked bar shows the impact of deduplication (usually quite small but up to a factor of 1.8). The next part of each bar shows the *additional* contribution of gzip after deduplication has been applied, but with no reordering or delta-encoding. Note that these two components will be the same for each pair of bars. The top component is the additional benefit of either mzip or delta-encoding. mzip is always slightly better (from 0.81% to 4.89%) than deltas, but with either technique we can get additional compression beyond the gain from deduplication and traditional compression: $>$ 80% more for EXCHANGE1, $>$ 40% more for EXCHANGE2 and $>$ 25% more for WORKSTATION1.

Figure 2.7(b) plots the compression throughput for mzip and DC, using an in-memory file system. mzip is consistently faster than DC. For compression, mzip averages 7.21% higher throughput for these datasets, while for decompression mzip averages 29.35% higher throughput.

2.6.4 Sensitivity to Environment

The effectiveness and performance of MC depend on how it is used. We looked into various chunk sizes, compared fixed-size with variable-size chunking, evaluated the number of SFs to use in clustering, and studied different compression levels and window sizes.

[6]Delta-encoding plus compression is delta compression. Some tools, such as vcdiff [50], do both simultaneously, while our tool delta-encodes chunks and then compresses the entire file.

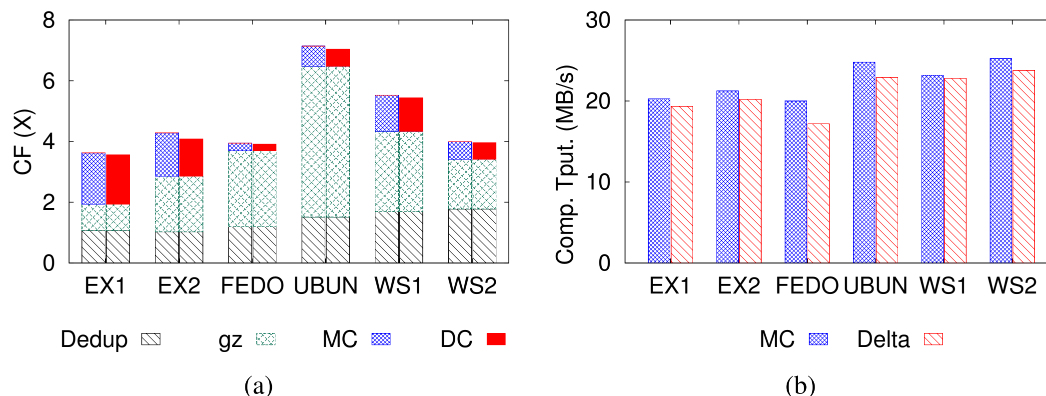


Figure 2.7. Comparison between mzip and gzip (delta compression) in terms of compression factor and compression throughput. (a) compression factor, by contributing technique. (b) compression throughput. CFs are broken down by dedup and gzip (same for both), plus the additional benefit of either MC or DC.

2.6.4.1 Chunk Size

Figure 2.8 plots gzip-MC (a) CF and (b) runtime as a function of chunk size (we show runtime to break down individual components by their contribution to the overall delay). We shrink and increase the default 8 KB chunk size by a factor of 4. Compression increases slightly in shrinking from 8 KB to 2 KB but decreases dramatically moving up to 32 KB. The improvement from the smaller chunksize is much less than seen when only deduplication is performed [51], because MC eliminates redundancy among similar chunks as well as identical ones. The reduction when increasing to 32 KB is due to a combination of fewer chunks being detected as identical and similar, and the small gzip lookback window: similar content in one chunk may not match content from the preceding chunk.

Figure 2.8(b) shows the runtime overhead, broken down by processing phase. The right bar for each dataset corresponds to standalone gzip without MC, and the remaining bars show the additive costs of segmentation, clustering, and the pipelined reorganization and compression. Generally performance is decreased by moving to a smaller chunk size, but interestingly, in two of the three cases it is also worse when moving to a larger chunk size. We attribute the lower throughput to the poorer deduplication and compression achieved, which pushes more data through the system.

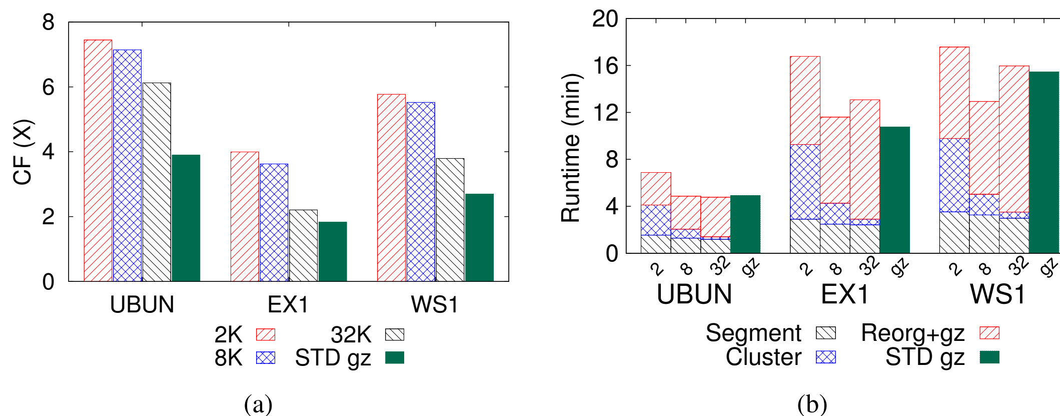


Figure 2.8. Compression factor and runtime for `mzip`, varying chunk size. (a) compression factor. (b) runtime, by component cost.

2.6.4.2 Chunking Algorithm

Data can be divided into fixed-sized or variable-sized blocks. For MC, supporting variable-sized chunks requires tracking individual byte offsets and sizes rather than simply block offsets. This increases the recipe sizes by about a factor of two, but because the recipes are small relative to the original file, the effect of this increase is limited. In addition, variable chunks result in better deduplication and matching than fixed, so CFs from using variable chunks are 14.5% higher than those using fixed chunks.

Figure 2.9 plots `mzip` compression for three datasets, when fixed-size or variable-size chunking is used. From Figure 2.9(a), we can see that variable-size chunking gives consistently better compression. Figure 2.9(b) shows that the overall performance of both approaches is comparable, and sometimes variable-size chunking has better performance. Though variable-size chunking spends more time in the segmentation stage, the time to do compression can be reduced considerably when more chunks are duplicated or grouped together.

2.6.4.3 Resemblance Computation

By default we use sixteen features, combined into four SFs, and a match on any SF is sufficient to indicate a match between two chunks. In fact, most similar chunks are detected by using a single SF. However, when using all four SFs, compression factors are improved greatly in some cases (e.g., a 13.6% improvement for EXCHANGE1) while compression

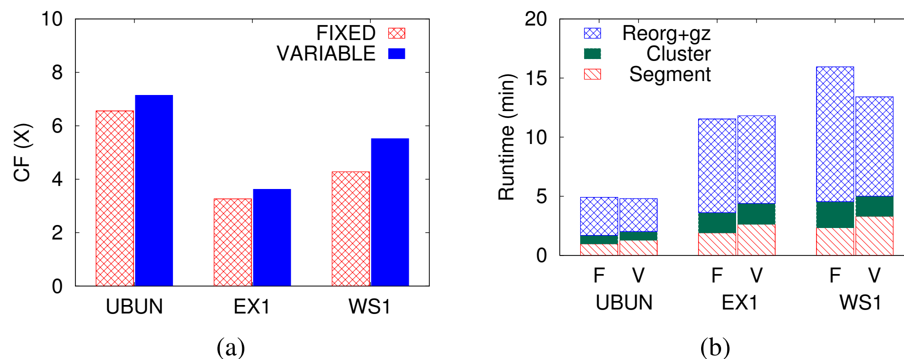


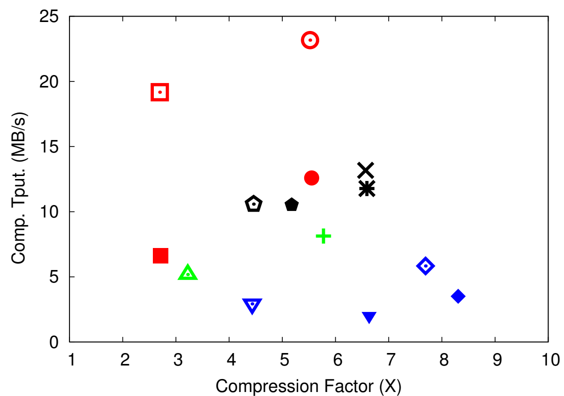
Figure 2.9. Compression factor and runtime for `mzip`, when either fixed-size or variable–size chunking is used. (a) compression factor. (b) runtime.

throughputs are not affected much. We therefore default to using 4 SFs.

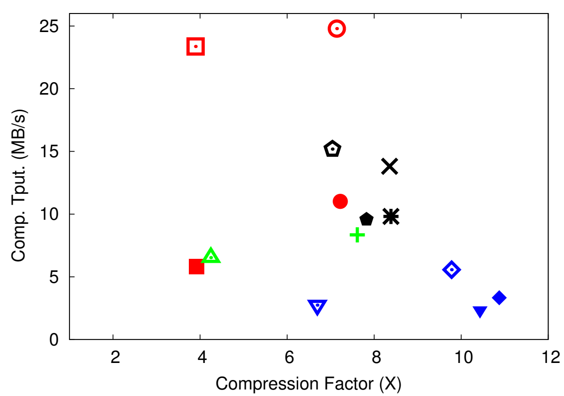
2.6.4.4 Compression Window

So far, we have focused on the default behavior of the three compressors we have been considering. Now, we turn to examine the "maximal" level. For `gzip`, the maximal level makes only a small improvement in CF but with a significant drop in throughput, compared to the default. In the case of `bzip2`, the default is equivalent to the level that does the best compression, but overall execution time is still manageable, and lower levels do not change the results significantly. In the case of `7z`, there is an enormous difference between its default level and its maximal level: the maximal level generally gives a much higher CF with only a moderate drop in throughput. For `rzip`, we use an undocumented parameter "-L20"to increase the window to 2 GB; increasing the window beyond that had diminishing returns because of the increasingly coarse granularity of duplicate matching.

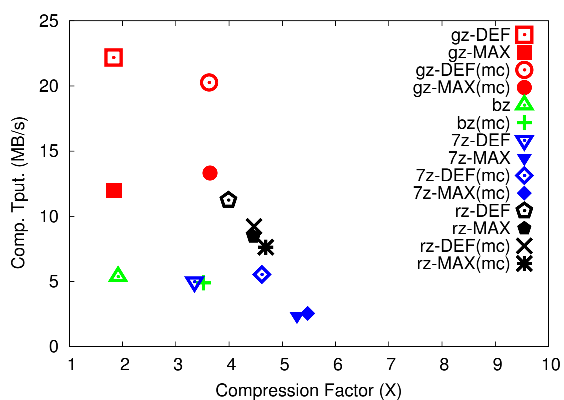
Figure 2.10 shows the compression throughput and CF for WORKSTATION1, Ubuntu-VM, and EXCHANGE1 when the default or maximum level is used, for different compressors with and without MC. From these figures, we can tell that maximal `gzip` reduces throughput without discernible effect on CF, and maximal `rzip` and `7z` improves CF and reduces performance. More importantly, for `gzip` and `rzip` with MC, we can achieve comparable or higher CFs with much higher compression throughput than compressors' standard maximal level. For example, the black cross marking `rzip`-DEF(MC) is above and to the right of the filled black pentagram marking `rzip`-MAX. This is also true for `7z` for WORKSTATION1. For this dataset, we also find that `rzip`-DEF with MC achieves



(a)



(b)



(c)

Figure 2.10. Comparison between the default and the maximum compression level, for standard compressors with and without MC, for three datasets. (a) WORKSTATION1. (b) Ubuntu-VM. and (c) EXCHANGE1.

comparable compressibility as 7z-MAX. The best compression comes from 7z-MAX with MC, which also has better throughput than 7z-MAX without MC.

2.7 Additional Use Cases of MC

In addition to using MC in the context of a single file, we consider two other uses of MC. First, we evaluate a prototype of MC in an existing deduplicating storage system. We then evaluate briefly the use of mzip for improving network transfer for distributing RPM packages.

2.7.1 Archival Migration in DDFS

We implemented MC during data migration from the active (backup) tier to the archive tier within DDFS. We evaluated the benefit of MC using a DDFS, running on a Linux-based backup appliance equipped with 8x2 Intel 2.53GHz Xeon E5540 cores and 72 GB memory. In our experiment, either the active tier or archive tier is backed by a disk array of 14 1-TB SATA disks. To minimize performance variation, no other workloads ran during the experiment.

Table 2.2 shows the characteristics of a few backup datasets using either form of compression. (Note that the WORKSTATIONS dataset is the union of several workstation backups, including WORKSTATION1 and WORKSTATION2, and all datasets are many backups rather than a single file as before.) The logical size refers to prededuplication data, and most datasets deduplicate substantially. DDFS compresses each compression region using either LZ or gzip.

The table shows that gzip compression is 25–44% better than LZ on these datasets, hence DDFS uses gzip by default for archival. We therefore compare base gzip with gzip after MC preprocessing. For these datasets, we reorganize all backups together, which is comparable to an archive migration policy that migrates a few months at a time; if archival happened more frequently, the benefits would be reduced.

Figure 2.11(a) depicts the compressibility of each dataset, including separate phases of data reorganization. As described in Section 2.4.3, we migrate data in thirds. The top third contains the biggest clusters and achieves the greatest compression. The middle third contains smaller clusters and may not compress quite as well, and the bottom third contains the smallest clusters, including clusters of a single chunk (nothing similar to combine it

Table 2.2. Datasets used for archival migration evaluation.

Type	Name	Logical (GB)	Dedup. (GB)	Dedup. +LZ (GB)	LZ (CF)	Dedup. +gzip (GB)	gzip (CF)
Workstation	WORKSTATIONS	2471	454	230	1.97	160	2.84
Email Server	EXCHANGE1	570	51	27	1.89	22	2.37
	EXCHANGE2	718	630	305	2.07	241	2.61
	EXCHANGE3	596	216	103	2.10	81	2.67

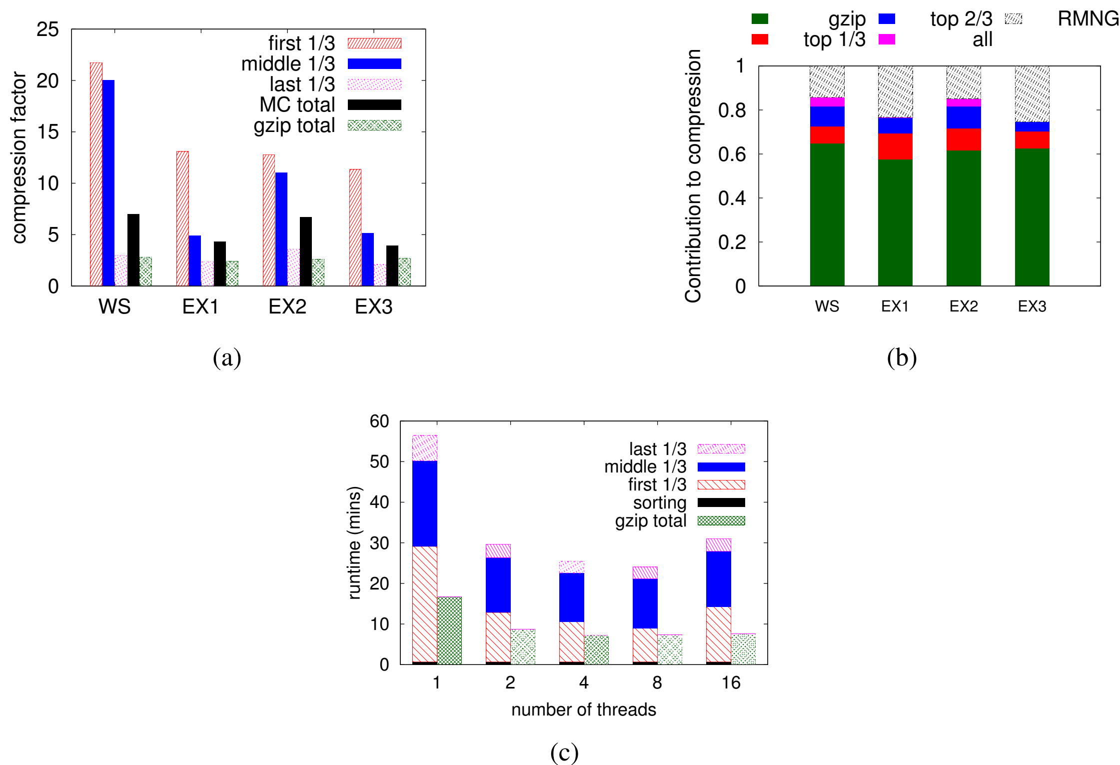


Figure 2.11. Breakdown of the effect of migrating data, using just gzip or using MC in 3 phases. (a) CFs as a function of migration phase. (b) fraction of data saved in each migration phase. (c) durations, as a function of threads, for EXCHANGE1.

with). The next bar for each dataset shows the aggregate CF using MC, while the right-most bar shows the compression achieved with gzip and no reorganization. Collectively, MC achieves 1.44–2.57× better compression than the gzip baseline. Specifically, MC outperforms gzip most (by 2.57×) on the workstations dataset, while it improves the least (by 1.44×) on EXCHANGE3.

Figure 2.11(b) provides a different view into the same data. Here, the cumulative fraction of data saved for each dataset is depicted, from bottom to top, normalized by the post-deduplicated dataset size. The greatest savings (about 60% of each dataset) come from simply doing gzip, shown in green. If we reorganize the top third of the clusters, we additionally save the fraction shown in red. By reorganizing the top two-thirds we include the fraction in blue; interestingly, in the case of WORKSTATIONS, the reduction achieved by MC in the middle third relative to gzip is higher than that of the top third, because gzip alone does not compress the middle third as well as it compresses the top. If we reorganize everything that matches other data, we may further improve compression, but only two datasets have a noticeable impact from the bottom third. Finally, the portion in gray at the top of each bar represents the data that remains after MC.

There are some costs to the increased compression. First, MC has a considerably higher memory footprint than the baseline: compared to gzip, the extra memory usage for reorganization buffers is 6 GB (128 KB compression regions * 48 K regions filled simultaneously). Second, there is run-time overhead to identify clusters of similar chunks and to copy and group the similar data. To understand what factors dominate the run-time overhead of MC, Figure 2.11(c) reports the elapsed time to copy the post-deduplication 51 GB EXCHANGE1 dataset to the archive tier, with and without MC, as a function of the number of threads (using a log scale). We see that multithreading significantly improves the processing time of each pass. We divide the container range into multiple subranges and copy the data chunks from each subrange into in-memory data reorganization buffers with multiple worker threads. As the threads increase from 1 to 16, the baseline (gzip) duration drops monotonically and is uniformly less than the MC execution time. On the other hand, MC achieves the minimum execution time with 8 worker threads; further increasing the thread count does not reduce execution time, an issue we attribute to intra-bucket serialization within hash table operations and increased I/O burstiness.

Reading the entire EXCHANGE1 dataset, there is a 30% performance degradation after MC compared to simply copying in the original containers. Such a read penalty would be unacceptable for primary storage, problematic for backup [25], but reasonable for archival data, given lower performance expectations. But reading back just the final backup within the dataset is 7× slower than without reorganization, if all chunks are relocated whenever possible. Fortunately, there are potentially significant benefits to *partial* reorganization. The greatest compression gains are obtained by grouping the biggest clusters, so migrating only the top third of clusters can provide high benefits at moderate cost. Interestingly, if just the top third of clusters are reorganized, there is only a 24% degradation reading the final backup.

2.7.2 File Transfer

One use case of `mzip` is to compress files prior to network transfer, either statically (done once and saved) or dynamically (when the cost of compression must be included in addition to network transfer and decompression). We performed analyses of `RPM` files to see how `mzip` might work on smaller files that are commonly transmitted over WANs. In a Fedora 11 machine, we sorted installed `RPMs` according to their installed size as reported by `dpkg-query`, selecting the top 40 largest packages. We then used `yumdownloader` to download binary packages for each, followed by `rpmtocpio` to convert the rpm packages into `cpio` format, an archival format similar to `tar`. The 40 packages resulted in 54 `cpio` files, since multiple binary files could be available for a given package, compiled for different CPU architectures. By default, `gzip` is used to compress a `cpio` file to be the payload of an RPM package.

Figure 2.12 compares the compression effectiveness between `mzip` and `gzip`, for the `RPM` dataset. As the baseline, we compressed the `cpio` files with `gzip`. For `mzip`, we first reorganized these `cpio` files in 1 KB chunks (scaling down the unit of reorganization because the files are much smaller than the backups and VMDKs) and then ran `gzip`. The aggregate size of these `cpio` files is 1710 MB. `gzip` compressed them to 576 MB while `mzip` compressed them to 488 MB, 15% smaller than `gzip`. The average CF for `gzip` is 3.17×, compared to 4.94× for `mzip`, and the average unweighted CF improvements (`mzip` over `gzip`) across this dataset is 32%. When we take account of the file size as the weight, we get an average weighted CF improvement of 15%. The improvements are not as huge as

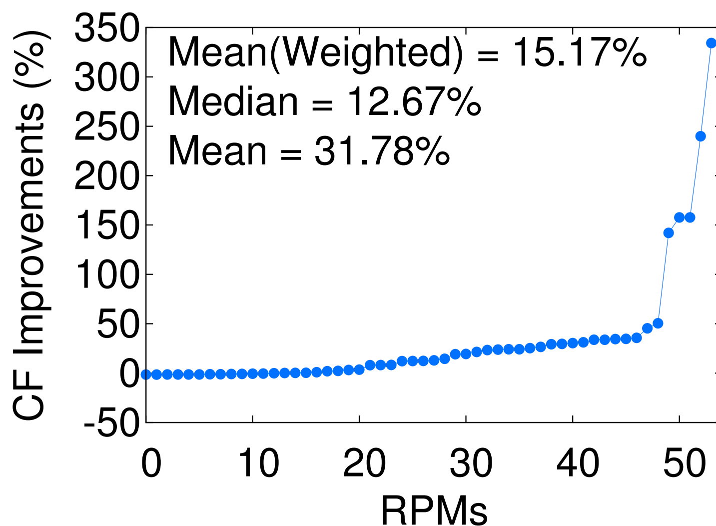


Figure 2.12. `mzip` improvement over `gzip` for `RPM`s.

we have seen in the evaluation section for `mzip`. We believe two reasons contribute to this result. First, files in this dataset are much smaller than those used in the previous dataset, which includes large backups and VM images. Second, this dataset contains binary data, which may have fewer duplicate and similar data blocks.

Figure 2.13 presents the results where we transfer extracted `RPM` packages over a dual-T1 network (a maximum of 3 Mbps in aggregate), in uncompressed (cpio), `gzip`-compressed (essential `RPM`) and `mzip`-compressed formats. We found a 13% overall reduction (from 2040 seconds to 1780 seconds) in transfer time through `mzip` (transfer time includes decompression at the destination). If compression at the source is also included, the reduction is 5%, since `mzip` takes a longer time to do compression. However, for the five RPMs with the greatest improvement from `mzip`, the transfer time decreased by 76%, when compression is done before-hand, and 69% when the cost of compression is also included. The relatively small improvement across the full set of `RPM`s contrasts with the significant improvement in some cases, suggesting that `mzip` may be useful for applications such as software distribution, where MC can sometimes greatly reduce the compressed size.

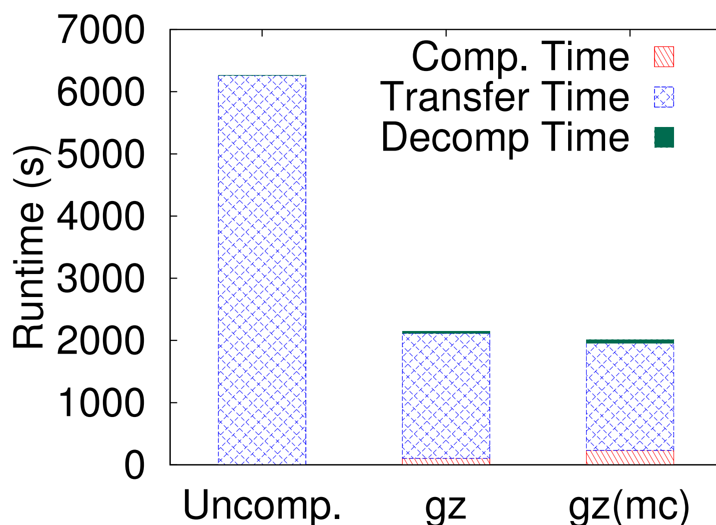


Figure 2.13. RPM end-to-end transfer times for uncompressed data and for data compressed with `gzip` and `mzip`.

2.8 Related Work

Compression is a well-trodden area of research. Adaptive compression, in which strings are matched against patterns found earlier in a data stream, dates back to the variants of Lempel-Ziv encoding [12, 52]. Much of the early work in compression was done in a resource-poor environment, with limited memory and computation, so the size of the adaptive dictionary was severely limited. Since then, there have been advances in both encoding algorithms and dictionary sizes, so for instance Pavlov's `7z` uses a "Lempel-Ziv-Markov-Chain" (LZMA) algorithm with a dictionary up to 1 GB [17]. With `rzip`, standard compression is combined with rolling block hashes to find large duplicate content, and larger lookahead windows decrease the granularity of duplicate detection [39].

The Burrows-Wheeler Transform (BWT), incorporated into `bzip2`, rearranges data within a relatively small window to make it more compressible [42]. This transform is reasonably efficient and easily reversed, but it is limited in what improvements it can effect.

Delta compression, described in Section 2.3.2, refers to compressing a data stream relative to some other known data [40]. With this technique, large files must normally be compared piecemeal, using subfiles that are identified on the fly using a heuristic to match data from the old and new files [50]. MC is similar to that sort of heuristic, except it permits deltas to be computed at the granularity of small chunks (such as 8 KB) rather than a sizable

fraction of a file. It has been used for network transfers, such as updating changing Web pages over HTTP [53]. One can also deduplicate identical chunks in network transfers at various granularities [44, 48].

DC has also been used in the context of deduplicating systems. Deltas can be done at the level of individual chunks [45] or large units of MBs or more [54]. Fine-grained comparisons have a greater chance to identify similar chunks but require more state.

These techniques have limitations in the range of data over which compression will identify repeated sequences; even the 1 GB dictionary used by `7-zip` is small compared to many of today's files. There are other ways to find redundancy spread across large corpora. As one example, REBL performed fixed-sized or content-defined chunking and then used resemblance detection to decide which blocks or chunks should be delta-encoded [41]. Of the approaches described here, MC is logically the most similar to REBL, in that it breaks content into variable sized chunks and identifies similar chunks to compress together. The work on REBL only reported the savings of pair-wise DC on any chunks found to be similar, not the end-to-end algorithm and overhead to perform standalone compression and later reconstruct the original data. From the standpoint of *rearranging* data to make it more compressible, MC is most similar to BWT.

2.9 Summary

Storage systems must optimize space consumption while remaining simple enough to implement. Migratory Compression reorders content, improving traditional compression by up to 2× with little impact on throughput and limited complexity. When compressing individual files, MC paired with a typical compressor (e.g., `gzip` or `7z`) provides a clear improvement. More importantly, MC delivers slightly better compression than delta-encoding without the added complexities of tracking dependencies. Migratory Compression can be applied broadly in other file systems.

CHAPTER 3

IMPROVE DEDUPLICATION BY SEPARATING METADATA FROM DATA

In the previous chapter, we talked about Migratory Compression, which improves compression by grouping similar blocks together. Now, we discuss a technique to improve deduplication by separating metadata from data[1].

3.1 Overview

Deduplication is widely used to improve space efficiency in storage systems. While much attention has been paid to making the process of deduplication fast and scalable, the effectiveness of deduplication can vary dramatically, depending on the data stored. We show that many file formats suffer from a fundamental design property that is incompatible with deduplication: they intersperse metadata with data in ways that result in otherwise identical data being different. We examine three models for improving deduplication in the presence of embedded metadata: deduplication-friendly data formats, application-level postprocessing, and format-aware deduplication. Working with real-world file formats and datasets, we find that by separating metadata from data, deduplication ratios are improved significantly—in some cases as dramatically as 5.6×.

3.2 Introduction

The amount of digital data continues to grow rapidly. Data deduplication has been shown to be effective in improving space efficiency for backup/archive storage systems [4, 22, 56], and there is an increasing interest in applying deduplication to general-purpose file systems [57, 58]. The effectiveness of deduplication is therefore crucial to the efficiency of such storage systems.

---

[1] This work was published at the USENIX HotStorage 2015 [55].

Generally, there are three types of deduplication: whole-file (also known as single-instance store [59]), fixed-size blocks [19], and variable-size content-defined chunks [4,44]. Whole-file and fixed-block deduplication work well in some environments [46, 60, 61], but using the content itself to determine deduplication unit boundaries is popular for two reasons. First, within a file, a small edit that shifts the remaining content would cause fixed-size blocks to align differently such that they would not deduplicate. Second, even small unmodified files may be written to the backup system by applications such as EMC *NetWorker* or Symantec *NetBackup* as part of a larger *aggregate* file to amortize overheads [51]; these aggregate files resemble UNIX *tar* files.

In this work, we show that many file formats suffer from a fundamental design property that is incompatible with deduplication: they **intersperse metadata with data** in ways that result in otherwise identical data being different. Metadata is changed more frequently, sometimes in trivial ways, leading to poor deduplication.

We observe that there are at least three ways to adapt ill-behaved data to deduplicating storage:

- **Deduplication-friendly formats** The best solution is to design file formats that separate metadata and data in a way that preserves potential deduplication. We provide a case study of EMC *NetWorker*, which has migrated to a new deduplication-friendly data format for backup data.

- **Application-level postprocessing** When it is hard to change the file format for an established application, it is often possible to postprocess files to produce a new format that is better suited to deduplication. We describe *mtar*, which transforms *tar* files into a more deduplication-friendly format.

- **Format-aware deduplication** Sometimes neither of the previous approaches is feasible, and special logic is required within the deduplicating system. We describe how EMC *Data Domain* appliances handle two cases of file formats that use special *markers* interspersed with data: tape markers for virtual tape libraries and block headers within the Oracle RMAN backup format [62].

This work makes three contributions. 1) This is the first detailed study that identifies the impact of metadata on deduplication. 2) It proposes two new formats (Common Data For-

mat and *mtar*) that improve deduplication. 3) It evaluates the new formats with real-world datasets and shows quantitative improvements to deduplication ratios. We hope that this study contributes to an increased understanding of the role of metadata in deduplication, and thus improved storage efficiency in future deduplicating systems and file formats.

3.3 Deduplication-friendly Formats

In this section, we[2] discuss our experiences with EMC *NetWorker*, a commercial backup software system. *NetWorker* was first developed in the age of tape-based backups and has since evolved. It uses application-specific data formats that describe data in different formats for different types of backup devices: disk backups and tape backups use different formats. The format is proprietary, but we discuss it here in general terms.

For disk-based file systems, the *NetWorker* save format includes these fields, among others: internal file identifier, file name, offset and size, and file attributes. These metadata fields precede the data of each saved file, and some of these fields are unfriendly to deduplication. In particular, the internal file identifier is a monotonically increasing sequence number, so adding a file to a directory shifts the sequence number of every file that follows, and the blocks are no longer identical. Attributes such as timestamps can thwart deduplication; we discuss this further in the context of *tar* in section 3.4.

We have designed a new Common Data Format (CDF), which separates data from metadata. The metadata of all files is grouped together and stored in one section, where it references file data stored in another section. This separation has a substantial impact on deduplication, and as a result, EMC's backup software products are migrating to this new format.

We evaluated CDF by estimating the deduplication across the backups of 15 hosts using content-defined chunks (8 KB average, 4 KB min, 12 KB max). Fingerprints are checked using Bloom filters [20]. This dataset is a subset of the *workstations* dataset used by Douglis et al. in an earlier study [63]; here we have fewer workstations and evaluate deduplication ratios (defined as $\frac{original_size}{deduplicated_size}$) only for the full backups. There were 15–25 backups per workstation, totalling up to about 420 GB. The more backups there are for a

[2]This section was mainly contributed by EMC collaborators. Thus, 'we' refers to EMC collaborators in this section.

host, the higher its deduplication ratio is likely to be, since the same data may appear more times. In addition, some hosts have remarkably high *internal* deduplication: the fraction of data within even a single backup that is eliminated by deduplication with other data in the same backup.

Figure 3.1 shows the deduplication within each of the datasets using the original *NetWorker* data format and the deduplication-friendly CDF format. Since both the number of backups and the internal deduplication affect the overall deduplication ratio, these are shown (colon-separated) as the x-axis labels for each host. (We ignore traditional LZ compression, which is applied after deduplication.) The datasets are sorted in decreasing order of deduplication.

The backups in the original format deduplicated rather poorly, with deduplication ratios of 3.6–6.1× even with over 20 backups stored. Moving to the CDF format produces deduplication ratios from 17.0–33.4×, with an average improvement (shown in red) of 4.9×. Many systems had aggregate deduplication better than simply finding the same data once in each backup. We attribute this to these workstations being engineering workstations containing multiple copies of certain data such as source code, leading to internal deduplication as high as 1.9×.

In addition we found that interhost duplication reported in the earlier study [63], using the original *NetWorker* format, understated the available deduplication. For instance, the most content in common across two hosts reported in that study was 74% of one host's chunks, but when considering only the data without the impact of metadata, it rose to 93%.

3.4 Application-level Postprocessing

Next, we look at *tar* as an example of an application that has a well-defined data format that is 1) unfriendly to deduplication; and 2) in wide use for decades, and is thus hard to change for compatibility reasons. For this class of applications, we propose *postprocessing* as a way to de-interleave data and metadata, improving deduplication.

3.4.1 Tar

tar [64] was initially designed for archival storage on magnetic tapes, and the format was optimized for sequential IO. A *tar* file is a sequence of entries, one per file, each containing a file header and data blocks. The file header includes metadata for that file,

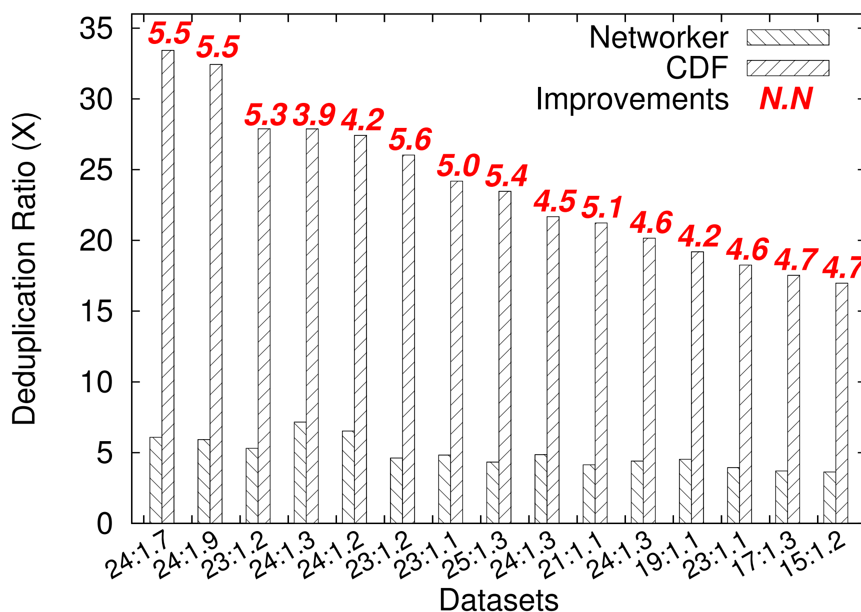


Figure 3.1. Deduplication ratios for workstation backups, using the original *NetWorker* and deduplication-friendly CDF formats. X-axis labels are X:Y, where X is the number of full backup generations analyzed and Y is the average internal deduplication ratio. Hosts were sorted by the CDF deduplication ratio. The number above each bar shows the improvement due to CDF.

including its path, ownership and modification time. The *tar* program works in 512-byte blocks. Thus, each file entry consists of one header block[3] and many data blocks. File headers are placed immediately before the corresponding data blocks to avoid multiple "seeks" when extracting a single file. An illustration of the *tar* format is presented in Figure 3.2.

While the *tar* format works well for tape backups, it is now also commonly used to store and distribute source code, binaries, and disk-based backups. *tar*'s decision to place metadata and data together in its file format, however, interacts poorly with deduplicating storage. Specifically, we found that when storing *tar* files of multiple releases of source code, we were only able to achieve deduplication ratios of about 2×; if we simply concatenated the files (thus throwing out the metadata), we were able to achieve deduplication ratios of up to 18× (with an average chunk size of 8 KB). The *tar* format clearly interferes with deduplication.

The underlying reason is that metadata changes more frequently than data blocks. By

[3]Multiple header blocks are used when the file path is too long to fit in a single header block.

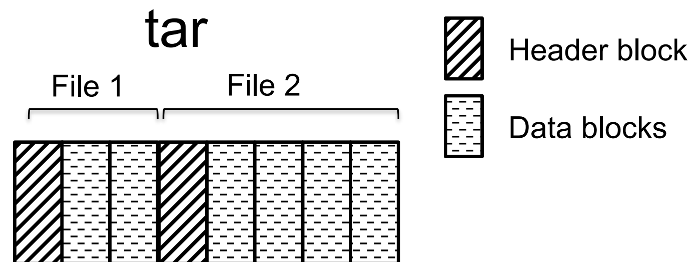


Figure 3.2. The *tar* format.

mixing more frequently changing metadata with data blocks, the *tar* format unnecessarily introduces many more unique chunks. An illustration of this problem is presented in Figure 3.3.

3.4.2 Migratory Tar

We propose a new Migratory Tar format (*mtar* for short), in which we separate metadata from data blocks by co-locating metadata blocks at the end of the *mtar* file. Changes in metadata are localized and isolated from data blocks, enabling better deduplication of the data.

An *mtar* file can be created by *migrating* a *tar* file. Specifically, we scan a *tar* file, output all data blocks to the *mtar* file and all header blocks to a temporary file, and then concatenate the two.[4] We store the offset of the metadata block in the first block of a *mtar* file for efficient access. To get back the original *tar* file, a *restore* operation reads the first block, finds the first header block, reads all data blocks for that file starting from the second block, and outputs it. This process is repeated for every file, resulting in re-creation of the original *tar* file. This dynamic reorganization of the *tar* file is similar to Migratory Compression (MC), discussed in Chapter 2. An illustration of the *mtar* format and the migrate and restore operations are shown in Figure 3.4.

mtar works best when a *tar* file includes many small files, because metadata interleaves with data more frequently. This is generally true for source code distributions, which we evaluate. For *tar* files that include mostly large files, we expect less benefit from *mtar*.

[4]Putting metadata at the end of the *mtar* file allows us to make a single pass over the input *tar* file: the amount of metadata and data cannot be known without reading the entire file, and appending the smaller metadata to the larger data is more efficient than the reverse.

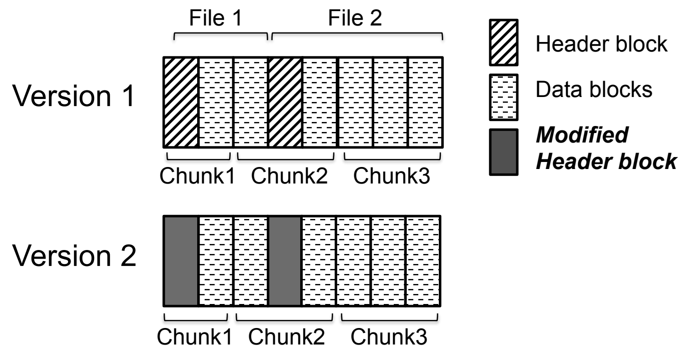


Figure 3.3. In *tar*, changes in header blocks lead to many new unique chunks. Chunk1 and chunk2 in version 2 are different from those in version 1 because of changes in header blocks.

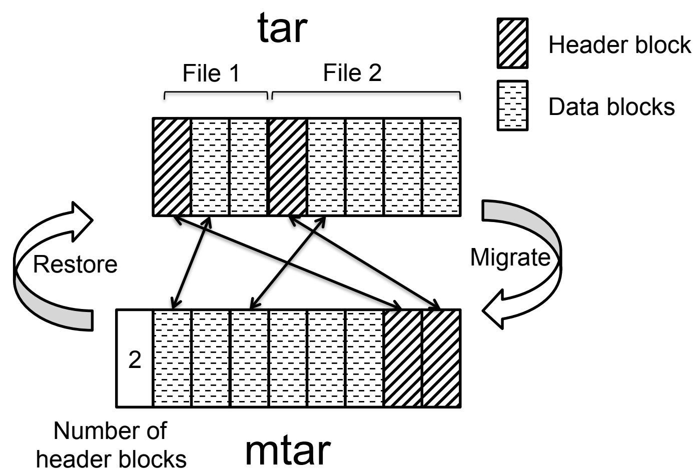


Figure 3.4. The *tar* and *mtar* formats and transformations between them.

We implemented *mtar* by extending GNU *tar* version 1.27.1 (the extension is available at `https://github.com/xinglin/mtar`).

3.4.3 Evaluation

To evaluate *mtar*, we use source code distributions of the Linux kernel and 9 GNU software packages from `ftp://ftp.gnu.org/gnu/`. For each package, we examine deduplication for multiple released versions. The software packages are shown in Table 3.1.

For each dataset, we download compressed *tar* files and decompress them. We remove

Table 3.1. Datasets for evaluating *mtar*

Software	Versions	Size (MiB)
automake	64	304.72
bash	23	276.69
coreutils	37	1284.49
fdisk	13	21.61
gcc	68	20315.45
gdb	32	4004.77
glibc	43	3811.48
smalltalk	33	685.39
tar	21	219.86
linux	308	98444.58

padded blocks at the end of each *tar* file,[5] then use our modified *tar* program to convert each *tar* file into an *mtar* file. We compared deduplication for *tar* and *mtar* using the *fs-hasher* tool, released by the File systems and Storage Lab at Stony Brook University [65]. We use variable-size chunking, with 8 KB as the average chunk size (4 KB min, 16 KB max). We also examined an average chunk size of 2 KB and 32 KB, with the same ratio for the minimal and maximal chunk sizes. An MD5 hash value is generated for each chunk; note that this is not collision-resistant, but for our analysis an occasional error is unimportant. We compare bytes in unique chunks to bytes in all chunks.

Figure 3.5 shows that *mtar* improves deduplication ratios over *tar* by 1.1–5.3×, at the 8 KB average chunk size. Using a 2 KB average chunk size, *mtar* achieves 1.1–3.3× improvements. For a 32 KB average chunk size, the improvements range from 1.1–6.2×. These results show *mtar* improves deduplication significantly.

Next, we examined deduplication ratios for metadata blocks and data blocks separately. For comparison, we also included deduplication ratios where we simply concatenated the content of each file. The results are presented in Figure 3.6, for the average chunk size of 8 KB. It is clear from the figure that metadata blocks have no duplication (with a deduplication ratio of 1 for all datasets), while data blocks show high deduplication ratios. More importantly, deduplication ratios for data blocks are close to the case where we concatenate source files, showing that *mtar*'s improvement over *tar* comes from increased

[5] *tar* does IO in fixed multiples of blocks, called records. It pads the last few blocks to be a full record.

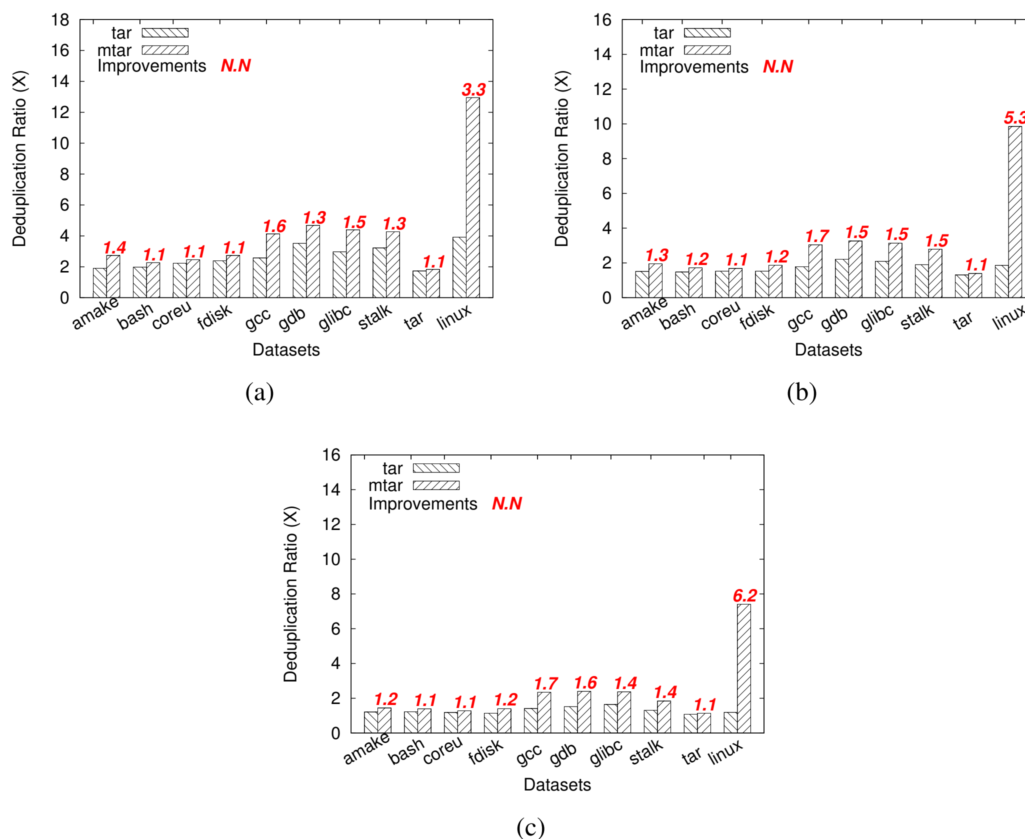


Figure 3.5. Deduplication ratios for *tar* and *mtar*, with different average chunk sizes. (a) 2 KB. (b) 8 KB. (c) 32 KB. The number in red above each bar shows the improvement due to *mtar*.

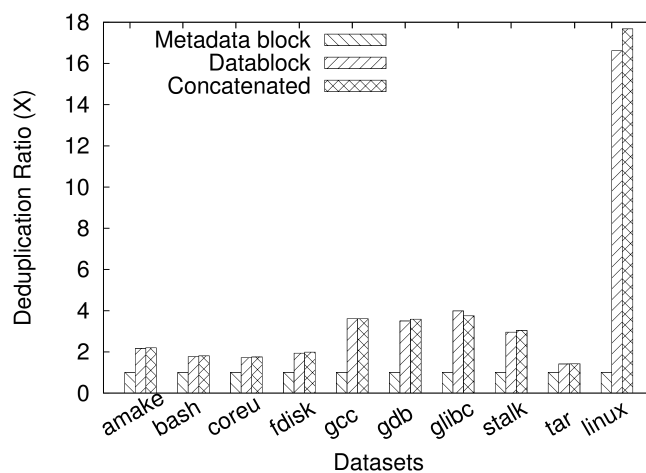


Figure 3.6. Deduplication ratios for metadata blocks, data blocks and source files that are concatenated together.

deduplication among the data blocks.

It is interesting to note the effect on stored (post-deduplication) blocks: as *mtar* improves the deduplication ratios for data blocks, the fraction of stored blocks that consist of undeduplicatable metadata blocks increases significantly. For the Linux kernel, before deduplication, 95.61% are data blocks and only 4.39% are metadata blocks. After deduplication, the unique data blocks become 5.75% (16.62× deduplication ratio), while we still have the same metadata blocks. While the percentage of metadata blocks is small in the original data, the weight becomes much more significant after deduplication: here, 43.3% of post-deduplication storage comes from metadata ($\frac{4.39}{(4.39+5.75)}$). Future work should also study how to store metadata efficiently.

3.5 Format-aware Deduplication

Formats that are not designed with deduplication in mind may needlessly degrade deduplication effectiveness. If it is not possible to change the data format or postprocess prior to writing it to storage, then the storage system needs to understand and address the effects on the fly. Here we [6] describe two examples of format-aware deduplication in EMC *Data Domain* appliances [4].

One of the earliest data types requiring special handling was the existence of "block markers" intended for magnetic tapes. When using a disk-based system to emulate tape, the incoming stream for a "virtual tape library" (VTL) device continues to periodically include block markers with a special bit pattern. Since these block markers appear at fixed intervals, a shift in content results in the marker appearing at a different point within a chunk, and the chunk does not deduplicate. Worse, the block markers can also contain variable metadata, even preventing deduplication of unmodified data. Data Domain addressed this by allowing the system to scan for block markers while performing chunking and fingerprinting. If one is identified, then it is removed from the content and stored in a separate location, thus the fingerprints of the remaining data are unaffected. Upon a read, the marker is inserted at the specified offset to restore the original data. Marker handling can have significant impact: for example, one customer saw deduplication improve from 9.9× to 16.8× with proper treatment of the interspersed metadata (a 70% improvement).

---

[6] Work presented in this section is also mainly contributed by EMC collaborators.

Another data type requiring special handling arises in Oracle RMAN backups [62]. RMAN writes fixed-sized blocks, configurable between 2KB and 64KB, each containing a block header and footer. Portions of the header and footer can change from backup to backup even when the block data remains unchanged, due to internal Oracle data formats. Additionally, RMAN may multiplex multiple data files together into a backup and the ordering of multiplexing can change; this affects content-defined chunk creation. Without special handling, therefore, deduplication would be degraded due to the modified metadata in the header and footer and the possibility of a change to the order in which files were multiplexed.

To solve the above problem, Data Domain modified the system to use block headers to delineate Oracle blocks as the (fixed-size) unit of deduplication and to remove portions of the block header and footer similar to tape marker handling: the variable portion is stripped out and stored as a small inlined data unit within the file system metadata, and the remaining content is fingerprinted. By doing this, deduplication becomes impervious to multiplexing order or changes isolated to the header.

In experiments doing 6 backups of an Oracle database with a 5% change rate between backups, we observed a 1.2–2× improvement in deduplication (4.5→5.28× without multiplexing and 2.48→5.07× with multiplexing). Thus, removing block headers restores the deduplication to that expected from the underlying data.

While we see promising results by making deduplication systems aware of data formats, this approach has a significant drawback, requiring deduplicating appliance manufacturers to track a moving target. Changes to the format cause deduplication to drop, requiring engineers to address the change. This results in a cycle of analysis, implementation, testing, and finally a patch release—a process that can take substantial time and effort.

3.6 Related Work

Most work in the deduplication space has focused on improving write throughput (e.g., Data Domain [4], Guo, et al. [66], Sparse Indexing [22] and Silo [21]), with a significant recent effort on improving restore performance [23, 25, 67]. Min et al. [24] proposed a general-purpose framework DeFrame, which allowed us to explore a variety of parameters (indexing structure, rewriting algorithm, etc.) in the design space within a single system. Li et al. [9] also studied power consumption for backup storage systems. However, little

work has been done to examine the impact of input data in deduplication. The closest work to both RMAN block special handling and *mtar* is a poster proposing a *tar*-format aware chunking algorithm [68]. To prevent the interference from metadata blocks, Sung et al. partition every header block as a deduplication chunk, while in our *mtar* approach, we group header blocks together and separate header blocks from data blocks. Their approach requires changes in the chunking algorithm for existing deduplication systems to make them aware of the tar format. *mtar* does not require any changes. Their chunking algorithm produces small chunks in the tar block size (512 bytes). This could break minimal chunk size requirements; in addition, small chunks dramatically increase deduplication system metadata overhead [51].

mtar has similarity to MC (Chapter2): both reorganize data to improve space efficiency, but *mtar* uses knowledge of the *tar* format to improve deduplication while MC rewrites generic data to improve traditional compression.

3.7 Summary

We have examined the effect of metadata on deduplication effectiveness. When metadata changes frequently over time, it is essential to separate it from data that stays more stable and would otherwise deduplicate. This separation can occur within the deduplication process, but that leads to complexity as well as dependencies on both data formats and deduplication environments. It can be done as a postprocessing step, which makes the benefits more generic: any deduplication back-end can benefit from the conversion process, but the postprocessor still must closely track the input format. Designing a data format to be deduplication-friendly has the best benefits of all, as the application improves deduplication in a platform-independent manner while isolating the storage system from the data format.

CHAPTER 4

USING DEDUPLICATING STORAGE FOR EFFICIENT DISK IMAGE DEPLOYMENT

In Chapter 2 and Chapter 3, we presented two techniques, Migratory Compression and separating metadata from data, to improve compression and deduplication. In this chapter, we present a case study where we use deduplication to improve space efficiency in storing disk images while at the same time maintaining high performance in image deployment[1].

4.1 Overview

Many clouds and network testbeds use *disk images* to initialize local storage on their compute devices. Large facilities must manage thousands or more images, requiring significant amounts of storage. At the same time, to provide a good user experience, they must be able to deploy those images quickly. Driven by our experience in operating the Emulab site at the University of Utah—a long-lived and heavily-used testbed—we have created a new service for efficiently storing and deploying disk images. This service exploits the redundant data found in similar images, using deduplication to greatly reduce the amount of physical storage required. In addition to space savings, our system also integrates with an existing highly-optimized disk image deployment system, Frisbee, without significantly increasing the time required to distribute and install images. In this chapter, we explain the design of our system and discuss the trade-offs we made to strike a balance between efficient storage and fast disk image deployment. We also propose a new chunking algorithm, called AFC, which enables fixed-size chunking for deduplicating allocated disk sectors. Experimental results show that our system reduces storage requirements by up to 3$\times$ while imposing only a negligible runtime overhead on the end-to-end disk-deployment process.

[1]This work was published at TridentCom 2015 [69].

4.2 Introduction

Disk images are widely used by modern, large-scale facilities to initialize the contents of a local disk when bringing up compute instances. A disk image captures, at a block level, the contents of a disk; this typically consists of an operating system and other software or data. Each disk image ranges from several GBs to hundreds of GBs, and maintaining a large catalog of images requires a large amount of storage space. On the other hand, for physical and virtual machines that will be booted from a local disk, this image must be transferred over the network from an image server and installed on the local disk before booting can begin. Because it is on the critical path for provisioning and booting nodes, the performance of image distribution and installation is critical. In this work, we consider two interrelated needs of a large-scale disk image deployment system: keeping the storage needs modest by using deduplication, and retaining high performance in image deployment through careful integration into an existing high-performance image deployment system.

IaaS facilities generally make a large collection of disk images available to their users; these images may contain a variety of operating systems and sets of standard software. In addition, most allow users to create disk images of their own. The Amazon EC2 Web site [3], for example, lists more than 37,000 public Amazon Machine Images. The Utah Emulab testbed (which we operate) manages more than 1,000 images—public and private—for its users [70], and the DETER testbed manages more than 400 [71]. These catalogs represent large amounts of data (21 TB for Emulab), and moreover, they grow steadily over time [70]. A facility's operators and users continually create new images, while old images need to be retained to support existing users or the reproducibility of previous results. It becomes important to store these large numbers of disk images efficiently.

Data deduplication has been shown to be an efficient way to save disk space for storing disk images. In a deduplicating storage system, large pieces of data—e.g., disk images—are divided into blocks, and every unique block is stored exactly once. If two images have a block in common, they share the single copy of that block. Because disk images are typically derived from other images by making small changes, there is significant duplication between an image and its "children." Previous work has shown that deduplication can greatly reduce the storage requirements of disk-image catalogs across virtual machines [46, 60, 72] and across machines in a commercial environment [61].

To support efficient and scalable image deployment, systems like Emulab have designed sophisticated mechanisms. Frisbee [26], used in Emulab, includes the following features. First, image data is compressed before it is stored, and it is transferred in compressed format during image deployment. Second, Frisbee utilizes filesystem information to skip unallocated disk sectors. This reduces the amount of data to store during image creation. More importantly, less data needs to be transferred across the network and fewer disk writes are needed during image installation. Third, the image file created by Frisbee is composed of independently installable chunks. Each Frisbee chunk [2] can be requested and installed independently and in any order. Last, Frisbee uses pipelining so that Frisbee chunks at different stages in the pipeline can be processed in parallel. To get the highest possible performance, the pipeline is designed so that the last stage (writing image data to disk) is the bottleneck. This ensures that Frisbee can install the image at the full speed of the disk. To be scalable, it implements its own application-level multicast protocol.

We present Venti-Frisbee (*VF*), our new image-deployment system that utilizes a deduplicating storage system to reduce the amount of physical storage while maintaining Frisbee's high performance in image deployment. We use Venti [19] as our deduplicating storage system, but any similar system should work.

Several challenges need to be addressed in order to use Venti to store disk images for Frisbee. Specifically, the integration should not break any of the features of Frisbee that make it efficient for image deployment. We deal with the following challenges:

- Compression plays an important role in Frisbee, so we have to decide when to do compression for the new system. Compressing images before storing them into Venti leads to poor deduplication, while storing raw image data into Venti requires Frisbee to compress it before distribution. To resolve this tension, we compress deduplication blocks before storing them into Venti. In this way, we get good deduplication and avoid compression during image deployment.

- Frisbee skips unallocated sectors and concatenates allocated sectors. This implies that, in the face of sector allocation and deallocation, the positions of sectors in the

[2]We use "Frisbee chunks" to denote chunks for Frisbee images. 'chunk' and 'block' refers to the deduplication unit and they are used interchangeably.

output image data will not remain the same. Fixed-size chunking may thus become less effective. We propose a new chunking algorithm, called Aligned Fixed-size Chunking (AFC). It utilizes disk offsets to pad the start and the end of each contiguous allocated sector range to ensure full blocks from each allocated range.

- To ensure that block retrieval from Venti does not become the new bottleneck in the pipeline, we select the block size for deduplication carefully. We use a larger block size (32 KB) in VF than those commonly used for backup and archival storage (a few KBs).

- The new system also needs to support Frisbee's ability to deploy an image in independently installable chunks. To support this feature, we precompute the Frisbee chunk header metadata.

With all these design elements working together, VF gets similar image deployment performance to unmodified Frisbee, while achieving significant space savings.

To summarize, this work makes three contributions. First, it presents the design of VF, which uses a deduplicating storage system for an efficient image deployment system, with goals to achieve efficient storage and image deployment simultaneously. Although VF builds upon Frisbee, we believe that the principles of its design are broadly applicable to IaaS image-deployment systems that need to combine efficient catalog storage with fast and scalable image deployment. Second, it presents a new chunking algorithm, called AFC. AFC enables us to retain the performance of fixed-size chunking for allocated disk sectors while achieving much better deduplication. Third, it evaluates VF using data from the Utah Emulab testbed. Experimental results show that VF achieves significant storage savings while also achieving run-time performance nearly identical to that of Frisbee. For a fixed space budget, a site of any size could store 3× more images for its users.

4.3 Foundation: Frisbee and Venti

VF is built on top of two existing systems: Frisbee [26], a scalable, high-performance disk-deployment system, and Venti [19], a content-addressable deduplicating archival storage system. In its original design, Frisbee stores disk images as files in a regular filesystem on the Frisbee server; VF replaces this back-end storage with Venti. While this change is

a conceptually simple one, Frisbee's design principles for efficient image distribution have four implications for VF. We discuss each implication in turn.

4.3.1 Frisbee

Frisbee is a disk-deployment system that was designed for clusters, datacenters, clouds, and other environments in which identical disk images must be deployed to a large number of servers in a short amount of time. The disk images can be distributed on demand to target machines, where they fully replace the contents of the target disks.

Frisbee's design principles are directly relevant to our new design with Venti, and so we describe them here. While our discussion focuses on Frisbee, similar principles can be found in other scalable high-performance disk-imaging systems. The overriding goal of these design decisions is *to install the disk image at full disk speed*: the disk's write speed represents a bound on how quickly the image deployment can complete. As long as the disk-deployment system can supply data fast enough to keep the target disk busy, disk deployment proceeds at the maximum speed possible. VF aims to preserve this property.

- **Utilize filesystem information.** For maximum generality and robustness, Frisbee works at the block level rather than the filesystem level. Utilizing information from the filesystem, however, helps Frisbee to distinguish allocated disk sectors from unallocated ones. Since filesystems typically have a large amount of unallocated space (only about 10% is allocated for images in Emulab), this brings several benefits to Frisbee. First, by storing only allocated sectors when creating a disk image, the storage requirements for each disk image are reduced. Second, it reduces the network bandwidth required to distribute the image to clients. Third, it reduces disk writes during image installation, as unallocated sectors can be skipped. However, it does mean that the sequence of disk sectors that goes into a Frisbee image is different from that of another image that has only a single additional sector allocated.

  Implication 1: VF must take block layout into account when deciding block boundaries for deduplication. If blocks are not aligned consistently between different images, this could result in poor deduplication.

- **Compress image data.** The data read from allocated disk sectors is compressed as it is added to the image file. As with filesystem-awareness, data compression

reduces storage requirements and network bandwidth during image distribution. The additional decompression step added during image installation does not introduce a significant overhead: decompressing image data can be done twice as fast as writing decompressed image data to disk, and the two tasks can be pipelined. On the other hand, doing compression for image data is significantly slower than any stage in the image deployment pipeline.

Implication 2: Image data should be compressed, but that compression must not be done at image-deployment time.

- **Independently installable Frisbee chunks.** As illustrated in Figure 4.1, Frisbee identifies ranges of contiguous allocated sectors, then compresses and concatenates them to form fixed-size (1 MB) "Frisbee chunks." Frisbee chunks are stored in the "on the wire" format so that the Frisbee server can send them without any processing overhead. They are also self-describing: all information needed to install a Frisbee chunk (such as where the data goes on the target disk) is kept in its header. This allows Frisbee chunks to be installed independently and in any order. When a new client joins an image-deployment session, it can begin processing the Frisbee chunks it receives immediately; it does not have to process the image sequentially starting from the beginning. To scale to a large number of clients, Frisbee uses IP multicast. Clients can join an in-progress distribution session at any time and the network protocol is client-driven. Each client asks for Frisbee chunks it does not yet have, and the Frisbee server multicasts these Frisbee chunks to all clients. This also enables clients with different processing power and disk throughputs to participate at different speeds. Retransmission for lost packets is handled at 1 KB granularity.

  Implication 3: To retain Frisbee's existing optimizations, VF must be able to construct Frisbee chunks independently and in any order.

- **Pipelining.** The design of independently installable Frisbee chunks also enables pipelining: the installation of one Frisbee chunk can be pipelined with the transmission and decompression of other Frisbee chunks. In Frisbee, there are two levels of pipelining, shown in Figure 4.2. The image-deployment pipeline has three stages: image data is *read* from the Frisbee server's disks, *transmitted* on the network,

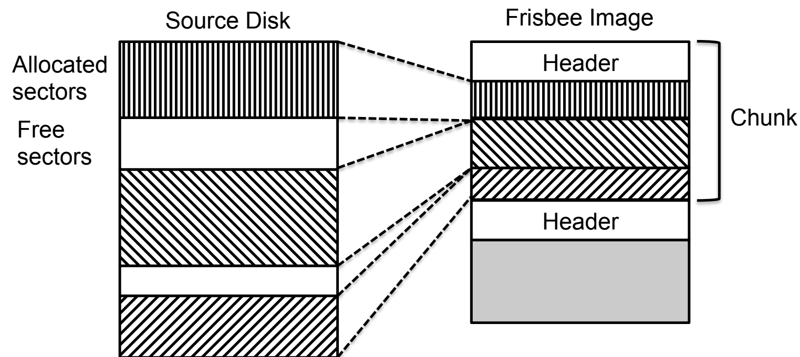


Figure 4.1. Frisbee identifies allocated disk sectors, compresses them, and concatenates them into 1 MB Frisbee chunks.

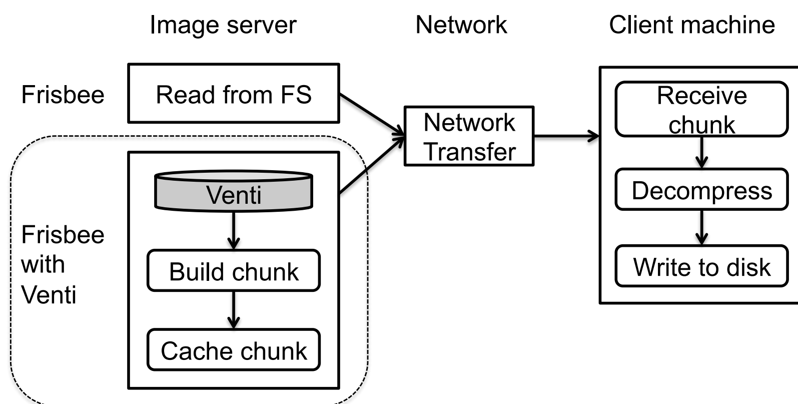


Figure 4.2. Frisbee's two-level pipelining design and the design of VF. In the new design of VF, the first stage (Read from FS) is replaced with Frisbee chunk construction from Venti.

> and *installed* on the target disk. Image installation at the client machine is further decomposed into three pipeline stages: *receiving* Frisbee chunks from the network, *decompressing* the data in those Frisbee chunks, and *writing* the decompressed data to the target disk. These stages are handled by separate threads so that they can proceed in parallel. The pipeline is designed such that the last stage (writing to disk) is the bottleneck of the pipeline overall. This results in a highly efficient disk-deployment system that succeeds in writing at the full speed at the target disk during image installation.

VF replaces the *image read* stage in this pipeline with a process that constructs Frisbee chunks from data stored in Venti. To meet its performance goals, VF must not allow this construction process to become longer than the other stages in the pipeline and thus become the new bottleneck.

Implication 4: To get a high level of performance comparable to the original Frisbee, the Frisbee chunk-construction stage from Venti in VF must be faster than the slowest stage (writing to disk) in the image-deployment pipeline.

4.3.2 Venti

Our second building block is Venti, a deduplicating storage system by Quinlan and Dorward [19], with enhancements from the Foundation [73] system. It has been used for daily archival snapshots of filesystems in the Plan 9 operating system. We use the Venti archival storage server. It provides a large data repository and exposes a simple object interface for clients to read and write variable-size blocks. A block can be any size from 512 B–56 KB. When a block is written to Venti, it returns a handle to retrieve that block. The handle includes the fingerprint (the SHA–1 hash of its content) for that block and it uniquely identifies a data block within the storage system. Venti skips writes of duplicate blocks and stores only unique ones. When compression is turned on, each unique block is compressed and then written to disk.

Venti is publicly available and it served our purposes in developing the VF prototype. As long as it is “fast enough” to not be a new bottleneck, VF should get similar high performance as the original Frisbee system. The design space explored in this work and its conclusions are independent of the particular deduplicating system used. Other dedupli-

cating systems, including commercial ones such as the EMC Data Domain Deduplication Storage System [4], could be used in place of Venti.

4.4 Design and Implementation

In this section, we lay out the design and implementation of VF, describing our design decisions relating to compression, chunking and selection of block sizes, and image reconstruction.

4.4.1 Compression

Traditional data compression plays an important role in Frisbee, reducing the data transfer across the network during image deployment; without it, network transfer would become the bottleneck in the image deployment. In terms of disk savings, compression results in a 3× reduction for the 430 Linux images we used in this study: it compresses 651 GB of allocated data to just 216 GB. In comparison, we found that deduplication gives us a 3–5× reduction in space. In order to get further space reduction, we must use both techniques together to get further improvement in storage—a deduplication scheme that is designed without compression will not likely lead to a significant decrease in storage requirements. However, the role of compression in the overall system must be carefully designed so that it will not affect deduplication or image deployment performance significantly. There are three possible alternative designs, shown in Figure 4.3.

Figure 4.3(a) presents the most straightforward approach to integrating Frisbee and Venti: it depicts storing compressed Frisbee images directly into Venti, by first partitioning them into blocks and then storing those blocks in Venti. This approach has the advantage that, at image-deployment time, Frisbee chunk construction requires only concatenating the data retrieved from Venti to reform chunks. However, it requires that deduplication be done on compressed data. Compressed data does not deduplicate well. There are two reasons. First, compressors have already tried to identify and replace repeated strings with more compact encodings. Moreover, even small changes to a disk (e.g., the allocation of a single sector) can produce a dramatically different compressed image compared to the original, leading to poor deduplication across multiple disk images. The overall effect is that this approach results in high image-deployment performance but little savings from deduplication.

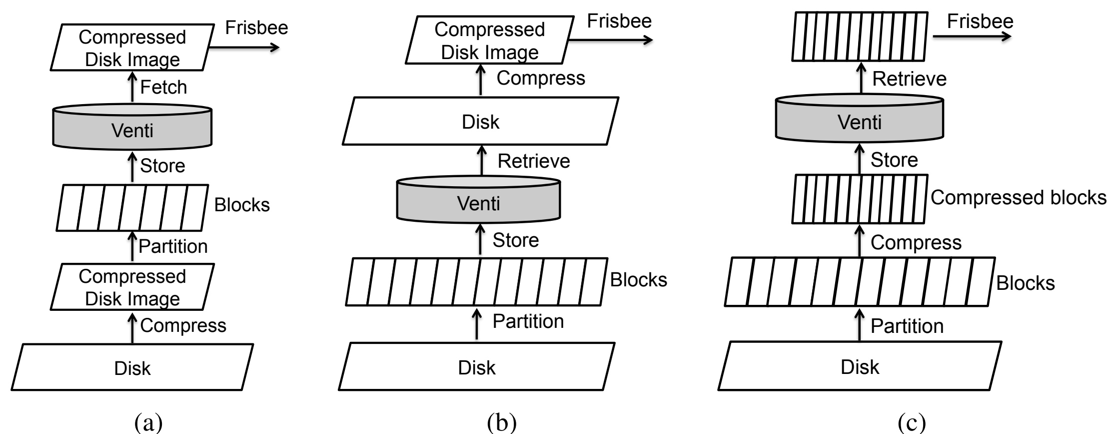


Figure 4.3. Three possible compression schemes. (a) store compressed images. (b) store uncompressed images. (c) store compressed blocks.

Figure 4.3(b) shows another option, which improves deduplication. It uses the Frisbee image-creation tool to identify allocated ranges on the disk, breaks uncompressed data from these ranges into blocks, and stores them in Venti. Venti fingerprints the blocks and stores only unique ones. This approach achieves efficient deduplication, but it incurs major overhead on the image-deployment path: data must now be compressed after retrieval and before sending it on the wire. This scheme meets our storage-saving goals, but falls short on high-performance image deployment.

A third approach, shown in Figure 4.3(c), performs compression immediately after partitioning image data into blocks and stores compressed blocks in Venti. During image deployment, compressed blocks are retrieved from Venti and concatenated to build Frisbee chunks, ready for Frisbee to deploy. No compression is needed during image deployment. This retains the full benefit of deduplication based on uncompressed data, since compressing two identical blocks results in identical compressed blocks. This slightly decreases the effectiveness of the compression itself (as compressors tend to operate better on larger blocks), but we found that this effect is very small. Since this approach gives us both good deduplication and high image-deployment performance, we adopted this approach in VF.

4.4.2 Chunking

When deciding how to deduplicate image data, we found two major design decisions to consider.

The first is the type of chunking algorithm to use. There are two types of chunking algorithms: fixed-size [19] or variable-size [4,44,74]. Fixed-size chunking determines block boundaries based on data offsets, while variable-size chunking is based on data content and is more resistant to boundary shifts from data insertions and deletions. Fixed-size chunking is straightforward and requires low computational overhead, while variable-size chunking has considerably higher computational overhead (almost one order of magnitude slower, as we will show later on in this section).

Second, we had to consider whether the source of data being deduplicated preserves the position of existing data when it is modified by an allocation or deallocation, or whether the data shifts due to these changes. "Stream-style" data, such as a file or a collection of concatenated files (e.g., Linux tar files), does not preserve positions: adding or removing data in a file shifts all data that follows the change. Variable-size chunking was designed specifically for stream-style data, as it is driven by content and not position. "Disk-style" data does preserve position: allocating or deallocating a sector does not cause other sectors to shift positions. However, disk-style data is usually larger because it does not distinguish between allocated and unallocated sectors. Thus, a larger amount of data needs to be processed and this increases processing time.

As described earlier, Frisbee uses filesystem information to identify allocated sectors and generates a data stream by concatenating only these sectors when creating a disk image. Because of this, a Frisbee image itself resembles "stream-style" data, and the most obvious choice would seem to be variable-size chunking. However, we found that we can get good performance and deduplication using a new approach we have designed, called Aligned Fixed-size Chunking (AFC), that allows us to do fixed-size chunking for allocated sectors. The key idea is to combine disk offsets with padding: breaking up contiguous ranges of allocated sectors in aligned units of the target blocksize using disk offsets, padding the first and last as necessary to ensure full blocks. The result is identical to performing fixed-size chunking on the disk itself after first zeroing unused sectors.

Figure 4.4 shows a comparison between using conventional ("baseline") fixed-size chunking and AFC. Image A is the base image. Image A' is different from A, by freeing the first two sectors (1 and 2) and allocating the next three ("a", "b", and "c"). Assume we are partitioning this image into fixed-size blocks of four sectors each. Frisbee will concatenate

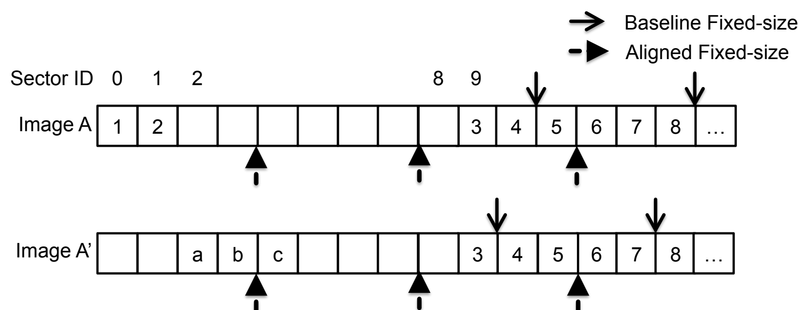


Figure 4.4. Comparison between baseline fixed-size chunking and aligned fixed-size chunking (AFC).

the second allocated range starting from the ninth sector with the first allocated range. Thus, in the baseline fixed-size chunking, for Image A, the first block will contain [1,2,3,4] and the second block will contain [5,6,7,8]. However, for image A', the first block will contain [a,b,c,3] and the second block will contain [4,5,6,7]. Sector allocations and deallocations cause block boundary shifts resulting in no deduplication between the images.

In AFC, we zero-pad ("z") at the start and the end of each range to ensure full blocks for each range. Thus, for Image A, we will generate the following blocks: [1,2,z,z], [z,3,4,5], and [6,7,8,z]. When applying this technique to the second image, we will get exactly the same block boundaries for deduplication, yielding [z,z,a,b], [c,z,z,z], [z,3,4,5], and [6,7,8,z]. Here the allocations and deallocation only affect the first two blocks, leaving the last two and other following blocks identical.

We performed an experiment measuring the deduplication ratio (defined as $\frac{original_size}{deduplicated_size}$) and runtime for these five chunking options, over the 430 Linux images used in our study. We used fs-hasher (the same tool we used in *mtar* in autorefchap3), a chunking and hash tool developed by Stony Brook University, with hash calculations disabled when measuring the chunking time.

Figure 4.5 presents the results. Here we can see that the variable-size chunking options (Variable-disk and Variable-stream) yield the best deduplication but perform poorly, with chunking times about 10× longer than their corresponding fixed-size alternatives (Fixed-disk and Fixed-stream). We also observe that chunking at the disk level (Variable-disk and Fixed-disk) increases chunking time by more than 2× compared with chunking at the stream level (Variable-stream and Fixed-stream). Finally, we note that AFC has perfor-

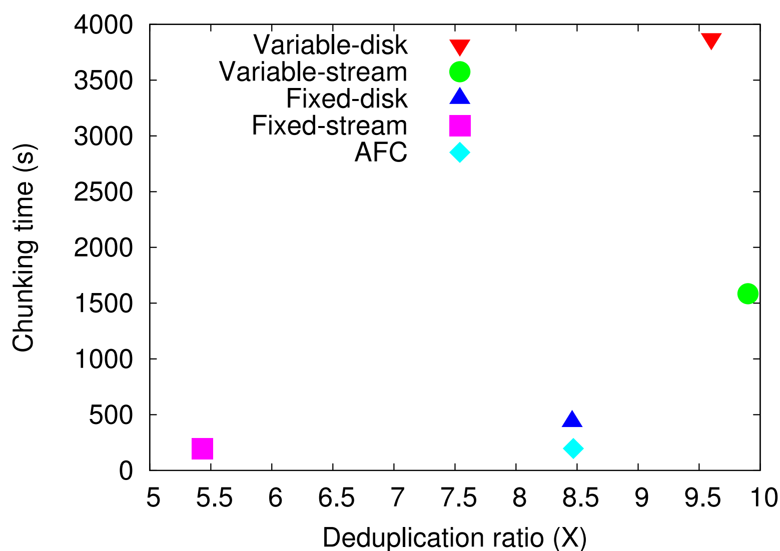


Figure 4.5. Comparison of different chunking options in terms of deduplication and runtime.

mance similar to fixed-size chunking at the stream level, while achieving the same level of deduplication as fixed-size chunking at the disk level.

4.4.3 Block Size

Having decided the chunking algorithm to use, we must now decide on a particular block size. Block size directly affects the efficiency of deduplication. Smaller sizes present more opportunities to find duplication, but at the cost of a higher ratio of metadata to data. (Smaller blocks mean that Venti must store more fingerprints.) Block size also affects the image-construction performance: more accesses to the Venti store are needed with smaller block sizes to fetch the same amount of data. This in turn can affect the performance of image deployment. Thus we must strike a balance between the deduplication ratio and the image-construction performance when choosing the block size.

The possible block sizes for Venti (and therefore VF) range from 512 B to 56 KB. Because we are dealing primarily with filesystem data in our images, we consider the lower bound to be 4 KB, which is the minimum block size used in many OS filesystem implementations. In subsection 4.5.3, we compare five candidate block sizes from the range 4 KB–48 KB, and find 32 KB to work best in practice.

4.4.4 Frisbee Chunk Construction

The previous sections have discussed the design space with respect to storing images in Venti. We now turn to retrieving them, or "constructing" Frisbee images from the deduplicated data. We focus on the construction of Frisbee chunks.

When a block is stored in Venti, Venti returns a "fingerprint" (a hash of the block's content) that can be used to retrieve it. Together, the list of fingerprints resulting from storing the entire image constitutes a "recipe file" for the image. At the time the image is created and stored, we precompute the mapping of Frisbee chunks to fingerprints. This is done by running the same process that Frisbee runs to create a disk image. The difference is, instead of storing compressed data blocks after the header of this Frisbee chunk, we store fingerprint indexes for data blocks in the recipe file for this Frisbee chunk.

Chunkmaker is responsible for constructing a Frisbee chunk. When it receives a request for a Frisbee chunk, it looks into the corresponding chunk header to get fingerprint indexes and then uses indexes to get fingerprints from the recipe file. After that, it retrieves the corresponding blocks from Venti and concatenates them with the precomputed chunk header to produce a complete Frisbee chunk. No complicated processing is required at image construction time; it simply reads from the deduplicating store. Repeated requests for the same Frisbee chunk are optimized by caching recently constructed Frisbee chunks.

4.4.5 Design Summary

The new image-deployment pipeline of the VF system is shown in the bottom half of Figure 4.2. Though similar to the original pipeline, the whole system incorporates four main considerations to make the new system as efficient as the original one for image deployment while improving storage efficiency significantly. The considerations include the informed choice of the deduplication block size, careful alignment of block boundaries, precompression of data blocks, and precomputation of header metadata for Frisbee chunks.

4.5 Evaluation

We start our evaluation by providing data about the performance of the unmodified Frisbee pipeline. These results support our claim that disk writes are the bottleneck and provide a lower bound for the performance of image construction.

Following that, our evaluation of VF is presented in three parts. The first presents an

empirical analysis of the three alternatives for performing compression and validates our choice for VF. The second describes the experiments we performed to measure storage savings and the image-construction time as a function of deduplication block size (mentioned in subsection 4.4.3). The third compares our VF system against the standard Emulab Frisbee implementation (hereafter referred to as "baseline Frisbee"), measuring both their storage demands and their image-deployment performance.

All experiments were performed on the Utah Emulab testbed [75]. The infrastructure for our evaluation consists of one server machine, acting as both a Venti archival storage system and a Frisbee image server, and 20 client machines all connected via dedicated 1 Gbps switched Ethernet. All machines are Dell PowerEdge R710s: each machine has a single quad-core 2.4 GHz 64-bit Xeon processor, 12 GB RAM, and two 250 GB, 7200 RPM Seagate SATA disks, each capable of sustained sequential read and write throughput of up to 110 MB/s. The server has one additional 1.5 TB, 7200 RPM Western Digital Caviar Black SATA disk hosting the Venti repository. This disk can perform sequential reads and writes at rates up to 150 MB/s. All machines run a 64-bit version of the Ubuntu 10.04 operating system.

For disk images, we used a collection of 430 Linux images from the Utah Emulab facility. Of these, 76 are "standard" images provided by the Emulab facility and 354 are custom images created by users. The chosen images were created between 2002 and 2011 and include images based on RedHat, Fedora, CentOS, and Ubuntu distributions. Individual images range in size from 146 MB to 1,836 MB, with a total disk size of 217 GB.

For all end-to-end image deployments, both the baseline Frisbee server and VF are configured to distribute data at a bandwidth of 500 Mbps. Factoring out network and Frisbee protocol overheads, this translates to a maximum image data rate of 57.5 MB/s.

4.5.1 Frisbee Pipeline Measurements

One thesis of our work is that extracting an image from Venti and constructing a Frisbee image needs to be fast, but ultimately just "fast enough" to not be the bottleneck for image deployment. To support this, we empirically measured the stages of the Frisbee pipeline. The results are shown in Table 4.1.[3] The last three lines show the stages of the image-

[3] We did not include image read time in our measurements as the disk where Frisbee images are stored can provide up to 150 MB/s read bandwidth and is unlikely to be a bottleneck.

Table 4.1. Average throughput rate and execution time of each stage in the baseline Frisbee pipeline. Throughput values are measured relative to the uncompressed data, except for network transfer, which reflects compressed data (with uncompressed rate in parentheses).

Stage	Throughput (MB/s)	Time (sec)
image compress	30.29	53.97
network transfer	54.27 (165.27)	9.54
image decompress	160.87	9.96
disk write	71.07	22.03

deployment pipeline. (The first, compression, is performed at image-creation time.) The network transfer rate of 54 MB/s appears to be the bottleneck, but this is a compressed data rate. The effective (uncompressed) data rate delivered to subsequent stages is actually 165 MB/s.

These results confirm that the client disk (71 MB/s) is in fact the bottleneck during image deployment. This is true even when the client is zeroing, rather than skipping (seeking over), unused disk space—measured at 89 MB/s. Finally, the results provide a lower bound for image-construction time (22 seconds). The result for image compression further shows the expense of compression relative even to disk writes, highlighting the necessity of keeping compression off of the image-deployment path.

4.5.2 The Impact of Compression

In Section 4.4.1 we presented three alternatives for where to do compression in VF (see Figure 4.3) and argued that the third alternative (c) was best. Here we present the empirical data to support our conclusion.

The expected drawback to the first alternative, storing compressed Frisbee chunks in Venti (a), is poor deduplication. To measure this, we loaded the compressed Frisbee chunks of 430 Frisbee images into a Venti store in 32 KB blocks. We observed a deduplication ratio of only 1.11×, compared to 3.26× for VF, confirming our expectation.

The second alternative of storing uncompressed image data and letting Venti compress it (b) introduces image compression in the deployment path. As we see in Table 4.1, image-data compression is much slower than disk write and would make image construction the new bottleneck. This would seriously impact the end-to-end performance of image deployment. We want to emphasize that this conclusion holds, independently of what

deduplication storage systems are used.

One concern with the approach we ultimately took for VF (c) is that we are compressing data in smaller units (individual deduplication blocks), which results in a lower compression ratio and hence larger Frisbee images. Larger images in turn mean that more data must be sent across the network. To investigate this issue, we measured the total size (number of chunks) for 430 Linux images compressed both in baseline Frisbee and in VF. The result was that images were indeed larger, but only by 6% (547.6 chunks per image versus 515.5).

4.5.3 The Space/Time Trade-off

As mentioned in Section 4.4.3, the choice of a block size for Venti storage can impact not only the storage savings but also the time required to retrieve and construct an image from Venti. To explore this trade-off, we populated five Venti repositories with all Linux images using 4 KB, 8 KB, 16 KB, 32 KB, and 48 KB block sizes, respectively, and measured the effect on deduplication and image-construction performance.

Table 4.2 summarizes the data deduplication achieved at the various block sizes. Overall, we achieve a 3–5× deduplication ratio. The results also show that the deduplication ratio increases as we decrease the block size. This is not a surprising result, because intuitively, smaller block sizes tend to increase opportunities for deduplication. Finally, we note that a larger deduplication block size improves compression and thus leads to a smaller image size (237.3 GB for the 48 KB block size versus 263.8 GB for 4 KB).

Whereas Table 4.2 shows just the image data stored in Venti, Table 4.3 shows the total amount of storage required, including the image metadata. The table includes baseline Frisbee `ndz` image files as the basis for computing storage savings. For baseline Frisbee, metadata consists of the per-chunk headers that record the ranges present in each Frisbee chunk; the “Total” column shows the total size of the `ndz` files for our 430-image collection. For VF images, metadata includes the fingerprints (SHA–1 hashes) for retrieving data from Venti and headers for Frisbee chunk construction, while “Total” is the sum of metadata size and the size of the Venti repository. This table shows that we can reduce the total storage space by more than 3×. That also means that we can store more disk images, given a fixed storage space.

Based on Table 4.2 and Table 4.3, it is tempting to choose the smallest Venti block size for VF in order to maximize storage savings. This is the decision one would make if only

Table 4.2. The effect of different Venti block sizes for deduplicating disk data. These storage figures are for disk data only, and do not account for image metadata.

Venti block size (KB)	Image data (GB)	Image data in Venti (GB)	Dedup. ratio (×)
4	263.795	50.310	5.24
8	253.305	60.025	4.22
16	245.665	67.943	3.62
32	239.892	73.617	3.26
48	237.313	76.173	3.12

Table 4.3. Total storage space required for storing images in different repository formats, including metadata. `ndz` is for baseline Frisbee images stored in a filesystem.

Repository format	Total (GB)	Metadata (GB)	Data reduction vs. `ndz` (×)
`ndz`	233.391	0.912	–
Venti 4 KB	55.456	5.146	4.21
Venti 8 KB	63.085	3.060	3.70
Venti 16 KB	69.534	2.010	3.36
Venti 32 KB	75.103	1.485	3.11
Venti 48 KB	77.486	1.314	3.01

considering storage savings. However, it is also important to consider the effect on image construction.

To produce an overall image-construction time, we measured the times required to construct individual Frisbee chunks. These times are summed, for every Frisbee chunk in an image, to get the construction time for a single image. We then averaged the per-image times to get an average image construction time for each block size. Because Venti maintains a cache of recently accessed blocks, we further consider two cases: one in which that cache is completely empty ("cold") and one in which it is not ("hot").

Figure 4.6 presents the results of this experiment. The stacked bars represent the average time required to construct an image in both the cold and hot cases. The horizontal line shows the average disk write time when installing an image at the target disk (from Table 4.1). This time is the "goal" that we must beat in order to avoid becoming the bottleneck in the image-deployment pipeline.

From this figure it is clear that increasing the block size significantly decreases the image-construction overhead, especially for Venti with a cold cache. It is also clear that we cannot use 4 KB and 8 KB as the block size because even with a hot cache, image-construction time exceeds the time required to write the image to disk. Of the remaining block sizes, it is tempting to use 48K since the image-construction time, even with a cold cache, can match the disk-write time. However, for the purposes of this work, we chose to use 32K, given that the majority of the time, the Venti cache will not be empty and we do gain slightly better deduplication. We note that the optimal block size is, in large part, an artifact of the specific performance characteristics of the deduplicating store. If we were to use a storage system other than Venti, we would need to re-evaluate the exact optimal block size, though the fundamental tradeoffs would remain the same.

4.5.4 Delivering a Large Catalog

The experiments described below compare VF to the baseline Frisbee system for deploying images.

4.5.4.1 Storage Savings

To explore storage savings at scale, we loaded our corpus of 430 disk images into the Venti repository in 32 KB blocks and measured the storage size of the repository after

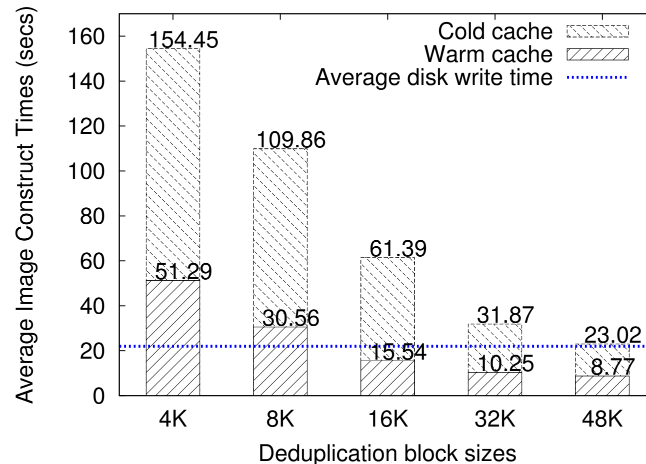


Figure 4.6. Average image-construction time for different Venti block sizes.

adding each image. We loaded the images into Venti in order of their creation times, oldest to newest, to obtain a realistic sense of the growth rate for an image repository over time. As we did this, we also tracked the storage that would be required by a conventional Emulab image store—a directory of `ndz` image files—in which the images are added in the same order.

The results show the growth of these two repositories is approximately linear in the number of images, but the Venti repository grows much more slowly than the `ndz` file repository. The absolute difference between the Venti storage and the file-based `ndz` storage is ∼75 GB vs. ∼233 GB. With a fixed storage budget, VF can store ∼3× more disk images. In the long run, VF may give more substantial savings. This is especially true in environments where new images are most often derived from existing images.

4.5.4.2 End-to-End Image Deployment

To determine how well VF works in a production environment, we performed end-to-end tests, deploying an image from the Frisbee server to one or more client machines using both baseline Frisbee and VF with a 32 KB block size. Frisbee scales quite well with an increasing number of clients [26]. We therefore ran tests to measure image deployment with 1, 8, and 16 simultaneous clients. Running with simultaneous clients increases the request load on the server and also tends to randomize and duplicate requests—situations which the VF server must be able to handle efficiently to compete with baseline Frisbee.

Our tests involved deploying an Ubuntu 10 image to one or more clients, then measuring the time until the last client completes. For both baseline Frisbee and VF, the Frisbee server was configured to distribute image data at a maximum throughput of 500 Mbps. The Ubuntu image contained 1.4 GB of uncompressed filesystem data. The compressed `ndz` file for baseline Frisbee is 394 chunks, while the image encoded by VF is 422 chunks. Each test configuration used either baseline Frisbee or VF to send the image to a given number of clients. We ran each configuration ten times, measuring the time required for all clients to download and install the image.

Figure 4.7 summarizes our results. The bars in the figure show the average time to deploy over the ten runs of each configuration. (The figure plots the standard deviation of the time in each configuration, but these are so small that they are hardly visible.) The results show that for 1, 8, or 16 clients, VF has just over a 2% increase in run time compared to baseline Frisbee.

Another scenario that Frisbee was designed to handle well is efficient deployment of images in the face of clients joining and leaving a session at different times. To ensure that VF handled this case efficiently as well, we ran another test with 20 clients evenly divided into five groups. Groups joined the Frisbee session at five-second intervals: the first group joined at time zero, and the last after 20 seconds. We designed this experiment so that the final group joins just before the first group is expected to finish, based on the run times from the previous experiment. For baseline Frisbee and VF, we ran this experiment ten times.

Figure 4.8 shows the results. The times shown are the average elapsed time for each group over the ten trials. Clients in all groups took a similar amount of time to finish. The difference between baseline Frisbee and VF is always less than 3%. These results show that VF performance is very close to that of baseline Frisbee.

4.5.4.3 Pipelined Distribution

The experiments in Section 4.5.4.2 show that VF suffered only a small performance impact compared to baseline Frisbee. Yet the most significant source of additional overhead in VF is the chunk-construction process (described in Section 4.5.3), which could take significant time. To understand how this overhead is masked, we instrumented the end-to-end distribution process and analyzed the steps involved.

Figure 4.9 presents the timeline for deploying the first 10 chunks of an image using

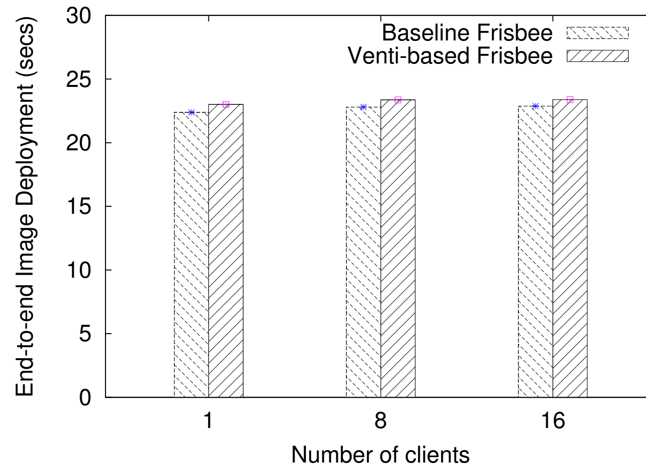


Figure 4.7. Average time to deploy an image to 1, 8, or 16 clients with synchronized start times.

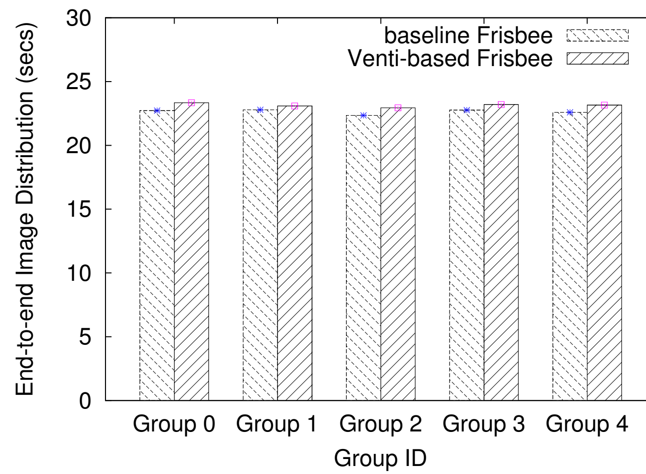


Figure 4.8. Average time to deploy an image to 20 clients with staggered start times. Clients start in five groups of four; for each group, we measure the time from client start to client finish.

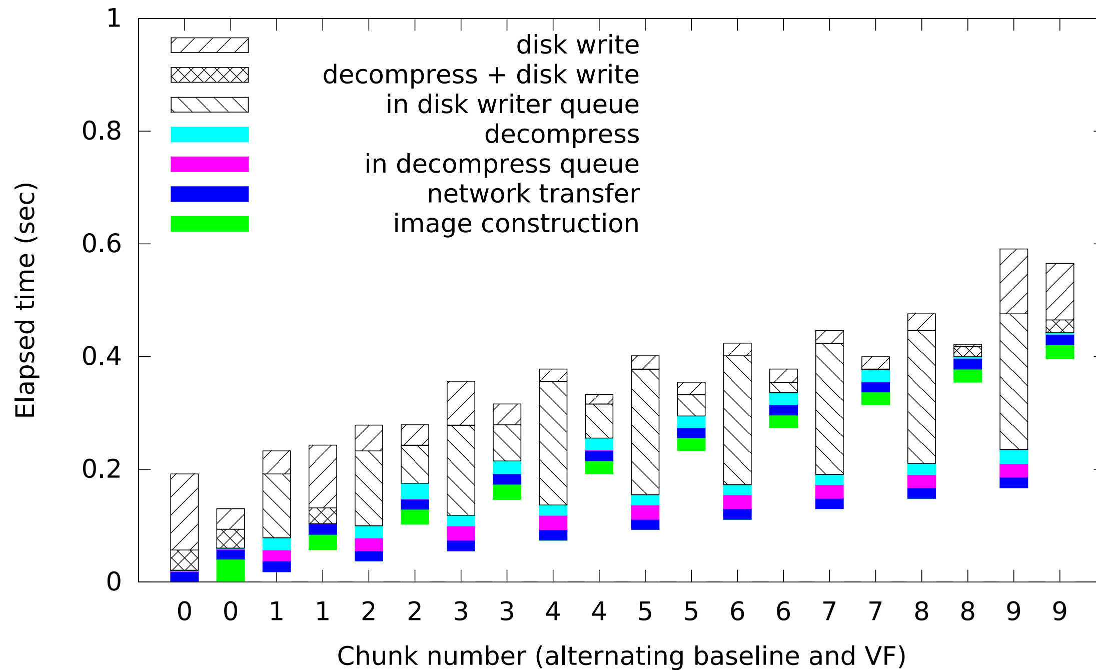


Figure 4.9. Timeline of chunk deployment for baseline Frisbee and VF. The x-axis interleaves per-chunk measurements from runs of baseline Frisbee and VF for the first 10 chunks of the same image. The y-axis shows elapsed time (relative to the first read of the first chunk) for each of the chunks broken down by categories.

both the baseline Frisbee and VF. Each vertical bar represents the time taken to deploy a single chunk, with each pair of bars showing the time for the same chunk using both Frisbee implementations. Within each bar, the time is divided into seven categories:

- **Image construction**: the time to read (baseline) or read and construct (VF) the Frisbee chunk.

- **Network transfer**: the time to transfer the entire Frisbee chunk over the network, measured from the server sending the first packet of the chunk to the client receiving the final packet of the chunk.

- **Decompress**, **decompress + disk write**, and **disk write**: because Frisbee chunk decompression is overlapped with writing decompressed data to the disk, we break down time to show when only the decompressor is active, both are active, or only the disk writer is active.

- **In decompress queue**: the time between when the last packet of a Frisbee chunk is received over the network and that chunk starts to be decompressed.
- **In disk writer queue**: the time between when the last piece of a Frisbee chunk is decompressed and the first piece of that chunk starts to be written to disk. Where this time is nonzero, it represents idle time in the processing of the chunk and indicates that the disk is the bottleneck in the system.

Note that the clocks for the server and client machines were synchronized within one millisecond.

For most Frisbee chunks sent by the baseline Frisbee (the long bars), one can see a rapid read, transmission, and decompression followed by a long queue time before the chunk is written to disk. The long queue time is due to earlier chunks still being written and clearly illustrates that the disk is the bottleneck in chunk processing.

For chunks sent by VF, there is a construction cost (the "image construction" bar), requiring more server processing per Frisbee chunk. This is reflected by the increasing gap between the start of processing of each chunk relative to baseline Frisbee. However, this cost is not reflected in the overall time for Frisbee chunk processing, as it is largely masked by the client-side disk bottleneck. This is shown by the decreased queue time for each chunk in VF. Thus, Frisbee chunks arrive at the client later in VF, but join a shorter write queue once they arrive.

4.6 Discussion

The design of Frisbee and VF were influenced heavily by the design of the Emulab system. In this section, we discuss how this affects the applicability of VF to other environments and what changes might be required to increase its generality.

A major assumption in the design of Frisbee is that the complete resources of the client machine are available during the image-deployment process. This ability to dedicate full CPU, RAM, and network bandwidth resources to image deployment, coupled with the use of commodity SATA hard drives, leads to the situation where the hard drive is clearly the bottleneck. We believe this situation is not unique to Emulab and is true for many image deployment systems, whether in a network testbed or a cloud infrastructure. Even using today's commodity Solid State Drives (SSDs) is not likely to move the bottleneck, since

even a 1 Gbs network is capable of transmitting data more quickly, given a reasonable (3×) compression ratio. However, continuing improvements in SSD performance, coupled with increased prevalence of 10 Gbs Ethernet will increase pressure on the Frisbee server. The read performance of Venti will need to be revisited along with other aspects of the server, such as the system used to store disk images.

The collection of disk images we used in this study was created between 2002 and 2011 and covers a diverse assortment of Linux distributions, including RedHat, Fedora, CentOS and Ubuntu. These represent a mix of system-provided and user-customized versions. The provenance of this data set raises the question as to whether it is representative of today's images. If anything, we believe the images chosen for this study are *more* "conservative" with respect to the potential for deduplication than are today's images. This is largely because, at least in Emulab, we have increasingly "incentivized" the use of custom images through newer, more convenient interfaces. These include single-click snapshotting of nodes and explicit versioning of images. Images created this way are more likely to be small variations of previous versions, and therefore deduplicate even better.

A final question is whether it is feasible to apply techniques used in VF (and Frisbee) in other environments, specifically OpenStack [76], one of the most popular Cloud software platforms. The image service in OpenStack is provided by Glance [77], which provides a RESTful API to add, retrieve, and query images. Glance provides a very basic image distribution service: the image data is retrieved from Glance by HTTP requests and responses. We believe the set of techniques used in Frisbee can be applied to Glance. These include transmission of image data in compressed format, use of multicast for scalable image deployment, partitioning of image data into independently installable data units, and the pipelining design in image deployment. We can also use data deduplication to store virtual machine images.

4.7 Related Work

The work related to VF falls into four categories: analysis of duplication in disk images, use of deduplicating storage for storing disk images, storage for virtual machines, and scalable and efficient disk-image deployment.

Several studies have analyzed data across disk images and have found significant amounts of duplicate data; this work supplies the basic motivation for VF. Jin and Miller [60] ana-

lyzed deduplication for a set of 52 disk images, applying various strategies for partitioning the images into fixed- and variable-length blocks. For a set of fourteen related images, they showed that deduplication could reduce storage space demand by up to 78%. They also showed that fixed-size chunking does slightly better than variable-size chunking at small block sizes. Meyer and Bolosky [61] compared block-based and whole-file deduplication over a dataset collected from 850 Windows desktop PCs in production use. They observed that there exists up to 72% and 83% duplicate data for file-based and block-based deduplication. Jayaram et al. [72] analyzed a set of 525 virtual machine images from their production cloud datacenter and found more than 30% of blocks appear at least twice. Smaldone et al. [46] and Lin et al. [37] analyzed virtual machine backup files and found there exists a significant amount of duplicate data. Atkinson et al. [70] found that disk images derived from a "base" image commonly share 60–80% of blocks with their base image.

Others have designed storage systems with deduplication to store disk images. Both the Mirage image format (MIF) [78] and the Marvin Image Store (MIS) [79] utilize file-based deduplication to improve storage efficiency. However, these stores optimize only for storage size, and are not designed specifically for scalable and efficient image deployment. Liguori et al. [80] used Venti to host *live* disk images for running virtual machines. Since most reads and writes inside the client virtual machine must communicate over the network with the Venti server, performance is poor and does not scale well. On contrast, VF installs an image on the client's local disk, giving consistent, high performance.

Some storage systems, like Parallax [81], Capo [82] and Lithium [83], were built for running virtual machines. With Parallax, Meyer et al. proposed to run a storage virtual machine at each host, to provide access to a shared block device. This approach will not scale well because it relies on centralized, shared storage. To improve scalability, Capo uses the disk at each host machine as the persistent cache for disk images. However, it requires a special block device in the hypervisor layer. I/O performance depends on whether an I/O request hits in the cache, and it becomes less consistent and predictable overall. Lithium is a distributed storage system utilizing local disks at each host. It uses peer-to-peer sharing to replicate per-VM storage volumes. The problem with some of these approaches is that they only work for running virtual machines; they are not applicable when users need

physical machines (which testbed environments like Emulab must support). Others require running storage services in the background, which could interfere with user applications. Our approach works for both virtual machines and physical machines, and no background storage service is needed.

Other related work looks at scalable image deployment. VMTorrent [84] and VMThunder [85] use peer-to-peer (P2P) sharing for image deployment. To further improve the opportunity to share data blocks, VDN [86] and Liquid [87] use deduplication in P2P sharing by using block IDs based on block content, rather than the combination of image name and offset. P2P imaging is unsuitable for testbed environments, as machines are unavailable to participate most of the time: running an intensive data-transfer application would risk interfering with active experiments. Instead, Frisbee [26] uses an application-level multicast protocol for scalable image deployment.

The work most similar to VF is LiveDFS [88], which examined both the deduplication effect and image-deployment performance. However, LiveDFS scales poorly: distribution time increases linearly with the number of virtual machine instances, whereas VF inherits Frisbee's flat scaling [26]. LiveDFS also did not explore the trade-off between disk-space saving and image-deployment performance, using a single block size for experiments. VF explores the role that compression plays in saving disk space, considering several options for how to include it. We also evaluated various block sizes to strike a balance between storage savings and image-deployment performance.

Versioning is another technique to store multiversion data efficiently, where the new version is stored as a delta (the difference between the new version and the original version), with a reference to the original version. A common use case is to implement efficient snapshots (e.g., LVM [89], WAFL [90], and Parallax [81]). While versioning might be effective to reduce storage space, it has several limitations. First, the complexity and the execution time to construct the latest version increases linearly as the number of versions increases. This is undesirable, since the latest version is likely to be used more frequently and should get the best performance. Second, the comparison in versioning is only between two versions, while deduplication can be done globally across all stored disk images and within a single image. We leave a full comparison between versioning and deduplication as future work.

Pullakandam [91] made an early attempt at using Venti to store disk images for Frisbee. It was not optimized in two ways. First, the earlier design stored raw image data into Venti, and compression of image data was done before image distribution (Approach (b), as shown in 4.3(b)). At a block size of 32 KB, the image construction time in the earlier work took 80 seconds, while it only took 10 seconds with a warm cache in VF. Given that it took only 22 seconds to deploy an image, image construction in the earlier work clearly becomes a significant bottleneck. Second, the chunking algorithm used in the earlier work was fixed-size chunking at the disk-level, while we propose a new chunking algorithm (AFC), achieving a similar deduplication effect with a shorter runtime.

4.8 Summary

This chapter has presented the design and evaluation of a system that couples deduplicating storage with a high-performance disk-deployment system. Integrating these components effectively requires an end-to-end view; the use of deduplicating storage in our complete VF system balances the goals of storage reduction and fast image transmission. The principles and trade-offs in the design of VF, which balances these goals, are the primary contributions of this work. By optimizing the storage for deduplicating data in this context and architecting an efficient pipeline, VF produces substantial image-storage savings while incurring very modest costs in the time required for image deployment. These properties are valuable for “infrastructure as a service” (IaaS) facilities, including both clouds and network testbeds, that must manage large catalogs of disk images and deploy images quickly to produce a good user experience.

CHAPTER 5

DIFFERENTIAL IO SCHEDULING FOR REPLICATING CLOUD STORAGE

Previous chapters have shown that similarity in content can be used to improve space efficiency. The same principle can also be applied in IO scheduling: specifically, we can schedule IO requests based on their characteristics, achieving predictable, consistent performance with high disk utilization. We present one such example Differential IO Scheduling in this chapter[1].

5.1 Overview

A common problem with disk-based cloud storage services is that performance can vary greatly and become highly unpredictable and inefficient in a multitenant environment. A fundamental reason for this is interference between workloads co-located on the same physical disk. We observe that different IO patterns interfere with each other significantly. As we introduce more random workloads into a system, the bandwidth received by sequential workloads degrades significantly. To deal with this problem, we propose Differential IO Scheduling to improve predictability and efficiency for read-intensive workloads. The key observation is that replication is commonly used for cloud storage systems, and thus we can dedicate each disk for serving only one type of read requests. This separation prevents interference between random requests and sequential ones, leading to consistent and efficient performance for sequential workloads. A prototype implementation based on Ceph shows the new scheduling algorithm provides more predictable and efficient performance for sequential workloads. It achieves a consistent aggregate bandwidth of 120 MB/s for 20 sequential workloads, regardless how many random read workloads are running concurrently. On the other hand, in the standard Ceph, the aggregate bandwidth drops from

---

[1]This work was published at HotStorage in 2012 [92].

240 MB/s when run alone, to only 18.54 MB/s when run concurrently with 40 random workloads.

5.2 Introduction

Infrastructure-as-a-Service (IaaS) cloud computing services offer virtual machines (VMs), that provide elastic computing, storage, and network resources. Exemplified by Amazon EC2 [93] and OpenStack [76], IaaS clouds are attractive because they are cost-effective and simple to manage.

There are several types of storage service abstractions in the cloud, including object stores (e.g., Amazon S3), block stores (e.g., Amazon EBS), and databases (e.g., Amazon RDS, Google Cloud SQL, and Microsoft SQL Azure). Among these options, block-level storage provides the most flexibility: because a block device is attached as a conventional disk to a VM, applications do not need to be modified to "port" them to the cloud. A block device can be mapped to a partition of a local attached hard drive, a logical volume from a Storage Area Network (SAN), or a customized driver that leverages a storage cluster. For example, Amazon EBS not only provides block storage, but also offers high reliability by performing replication in the storage cluster.

However, because of their multitenant nature, a large number of diverse workloads are running concurrently by different tenants, leading to competition for resources and interference among different workloads. The performance experienced by end-users in cloud storage environments varies unpredictably, sometimes more than an order of magnitude, compared with a dedicated cluster [33, 34]. For example, Schad et al. [34] observed that the performance of CPU, memory speed, I/O, and network bandwidth in Amazon EC2 is at least an order of magnitude *less stable* than a physical cluster. In another report Shripad and Radu pointed out that at different times of day, the performance of Amazon EBS and S3 also varied significantly [94]. Such performance fluctuations are a natural consequence of sharing servers, networks, and storage among many different users. For block storage, when two or more tenants share the same physical disk, they compete for the disk head position for I/O accesses. For instance, random workloads from one tenant can destructively interfere with sequential workloads of another tenant [35], and reads may conflict with writes [36]. Such interference makes the performance experienced by applications highly unpredictable. Attackers may also use the performance anomaly as a

covert channel between VMs to perform co-residence checks [95].

We are targeting a cloud environment under heavy use by a large number of tenants running a diverse set of workloads. The workload may be database queries made of many small random reads, or MapReduce [96] jobs with mostly sequential IOs and some random IOs. In this work, we focus on solving one of the interferences that could happen in such environments: the interference between random and sequential read workloads. When more random workloads are introduced into a system, more seeks are required from the disk, and it becomes much less efficient in serving sequential workloads. Ideally, we want to build a system that stays at a high utilization for sequential workloads, even when there are a large number of random workloads running concurrently.

For better reliability, replication is commonly used in cloud storage systems, such as Google File System [31] and Ceph [32]. We propose to take advantage of replication to isolate the interference between random and sequential read workloads by dedicating each disk to serve only one type of workloads. To demonstrate the problem, we first present interference analysis between random and sequential workloads for both a single hard disk drive and Ceph, a popular distributed storage system. Then, we present Differential IO Scheduling (DIOS for short): it detects types of read requests and intelligently schedules read operations to different replica disks, to avoid co-locating random and sequential reads and thus the unpredictable interference between these types of workloads. We present an evaluation of our prototype implementation based on Ceph, validating our hypothesis that DIOS provides a more predictable and efficient performance (up to 6× better) for sequential workloads, with a reasonable degradation in performance (within 50%) for random workloads as the tradeoff.

5.3 Interference Analysis

To motivate the design of DIOS, we first present undesirable interference between sequential and random read workloads when they are concurrently executed. We use the *FIO* tool [97] to generate workloads directly at the block storage level. For higher-level abstractions, we use the TPC-H benchmark [98] to simulate real-world application scenarios. Specifically, our objective is to evaluate the performance interference between random and sequential read workloads. We present the study of the interference for a single physical disk and Ceph, on which our prototype implementation is based.

5.3.1 Benchmark with FIO

We use FIO to investigate the performance interference between random read (RR) and sequential read (SR) workloads. Each workload was set to run for 120 seconds, and direct I/O was used to bypass operating system caches and examine disk behavior directly. We measured throughput of random workloads in terms of I/O operations per second (IOPS), and used bandwidth (MB/s) to measure sequential workloads[2]. For the single disk tests, we used a TOSHIBA 10K RPM 600 GB SCSI disk (product ID MBF2600RC). This disk is attached to a RAID controller but is exported to the operating system directly as a single SCSI target. The controller is a MegaRAID SAS 2008 [Falcon] from LSI Logic / Symbios Logic. A different 10 GB physical region of the disk is used for each workload.

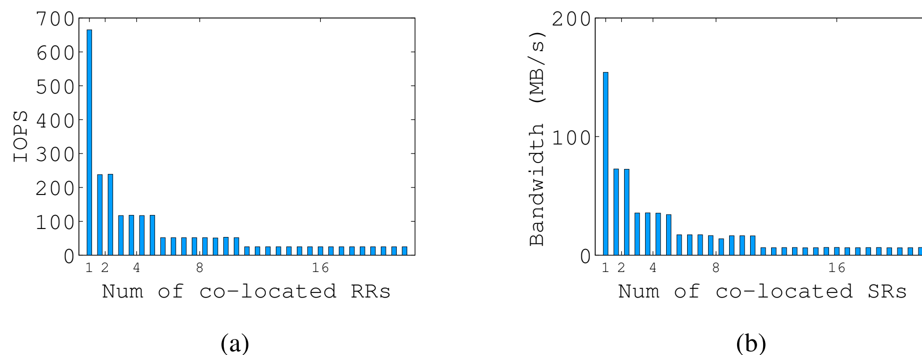
- **Co-locating same type of workloads.** Our first set of experiments investigated the interference between workloads of the same type. We varied the number of concurrent workloads from 1 to 16, doubling it at each step. The default I/O size is 8 KB for random workloads and 64 KB for sequential workloads. We kept 32 outstanding I/Os by default. The throughput for each workload is measured and we present the results in Figure 5.1.

  The figures show clearly that as we co-locate same type of workloads, each workload gets similar throughput, implying a fair sharing among them. The performance is also predictable. Hence, we have our first observation.

  Observation 1: When co-locating workloads with the same I/O request characteristics, each workload gets a fair share in performance and system resources.

- **Co-locating random and sequential read workloads.** Next, we examine effects when we co-locate RR workloads with sequential ones. The same settings for RR and SR workloads were used as the previous experiment. In the baseline, we keep adding more SR workloads to the disk until we reach a total of 17 SR workloads, adding one SR workload at a time. For comparison, we run a single SR workload and keep adding one RR workload at each step until 16 RR workloads are added.

[2]For random workloads, IOPS shows the seek interval directly while for sequential workloads, the throughput is not sensitive to seeks.



(a)

(b)

Figure 5.1. Co-locating same type of workloads. (a) random read. (b) sequential read.

We report aggregated bandwidths of SR workloads. The results are presented in Figure 5.2(a).

The figure shows that the bandwidth the disk can deliver dropped rapidly as we added the first few random workloads (∼10 MB/s when four RR workloads were introduced) and became extremely small by the end. On the other hand, the disk sustained to deliver ∼100 MB/s when co-locating sequential workloads, even when we added 16 sequential workloads.

We also did a similar experiment with Ceph. The Ceph cluster was configured with 12 disks, with a replication factor of 2. We ran the same set of experiments, except that they were scaled up: in the baseline, we experimented with up to 200 SR workloads, by adding 2 at each step. For comparison, we ran 20 SR workloads and added 2 RR workloads at each step until we had a total of 200 workloads. The result is presented in 5.2(b). The first green triangle from the left represents the aggregated bandwidth for 20 SRs when we added the first 2 RR workloads, with the second triangle standing for the aggregated bandwidth for SR workloads when 4 RRs were added, and the same applies for the rest. We see a similar trend that, as we introduce more random workloads into the system, bandwidths for sequential workloads drop significantly. From these experiment, we know the following.

Observation 2: RR workloads can destructively impact a disk's effective utilization. We should try to avoid co-locating random workloads with sequential workloads, in order to get high disk utilization.

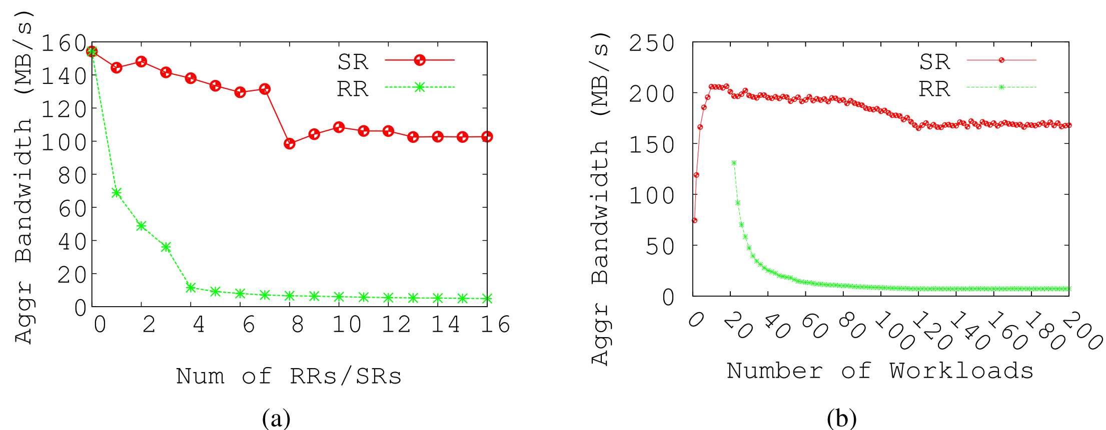


Figure 5.2. Co-locating random read workloads with sequential read workloads, for a disk or Ceph. (a) disk. (b) Ceph.

5.3.2 Higher-level experiments

Our previous results demonstrate the interference between workloads at the block storage level; we now show that such interference propagates up to the application layer. To measure this effect, we used the query 7 (dominated by random IOs) provided with TPC-H as a random workload, and a scan of the *lineitem* table as a sequential workload. We populated two mysql databases with the same dataset at the scale factor of 1. The data of these two databases is stored in two partitions in the same disk. We ran these two workloads against each database, either in isolation or concurrently. When both workloads are run concurrently, we re-start the workload that finishes earlier (with another mysql database) to ensure that the other workload is interfered across its whole execution. We compare the total execution time for each workload, averaged across three runs.

Figure 5.3 presents the results. Running the two workloads concurrently on the same disk increases the runtime of the random workload by only 16%: when run by itself, it takes 113 seconds, and when co-located with the sequential workload, it takes 131 seconds. However, co-location increases the runtime of the sequential workload much more substantially, from 73 seconds to 172, an increase of 135%. This clearly demonstrates that random workloads do interfere with sequential workloads (destructively) for real-world applications.

We also scaled this experiment up to using 20 scans with 20 queries and ran it with the Ceph cluster. The increase in runtime for scan workloads is 86%, from an average of

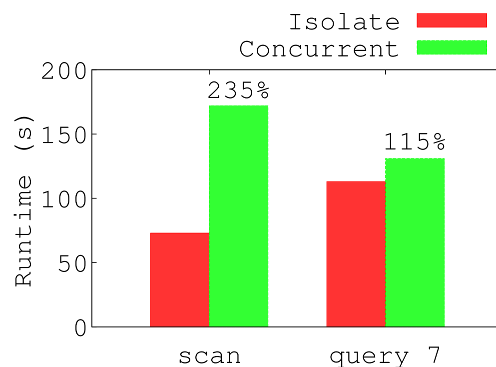


Figure 5.3. Performance interference in TPC-H, running with a physical disk

187 seconds to an average of 348 seconds, while for query workloads, it is a 94% increase, from 562 seconds to 1088 seconds. In this case, we did not see a destructive impact from co-locating these two types of workloads. This is likely because we have not generated enough load for the system, and disk IO might not be a significant bottleneck for this workload: each query was only issuing 1 or 2 outstanding IOs, while we have 6 disks and each IO request needs to traverse the IO stack at the client machine, transmitted across the network, served by the IO stack at the server machine, and finally transmitted back to the client and returned to the application. Running TPC-H queries with mysql databases at this scale does not seem to be the suitable workload for demonstrating our point. In the following evaluations, we focus on FIO and leave exploration of a larger scale of TPC-H queries and other high-level workloads for Ceph as future work.

5.3.3 Discussion

We now discuss effective disk utilization and our rationale for optimizing for sequential workloads. For serving an IO request, the disk needs to seek to the correct cylinder, wait for the right sectors to be rotated under the disk read/write header, and finally transfer data. Seeking and rotation are necessary but do not effectively contribute to data transfer. So, one metric to measure the effective utilization of a disk is the ratio of time when it is transferring data to the total time. The formal definition of effective disk utilization is as follows ($T1$: transfer time, $T2$: seek time, $T3$: rotation time, R: disk transfer rate).

$$disk\,util = \frac{T1}{T1+T2+T3}$$

$$Bandwidth = \frac{T1*R}{T1+T2+T3} = \frac{T1}{T1+T2+T3} * R = disk\,util * R$$

Measuring disk transfer time and seek time directly is challenging. A reasonable metric to use is the disk bandwidth, since there is a linear relation between disk utilization and bandwidth, as shown by the definitions above.

By definition, sequential workloads are using a disk at high utilization, since only a single seek is needed for the first read request, and after that, the disk can start to transfer data at its full capability. On the other hand, random workloads lead to poor disk utilization, since for almost every read request, the disk needs to do a seek and wait for the rotation. As we can see from previous analysis, introducing random read workloads has a destructive impact on bandwidths for sequential workloads. To maintain high disk utilization, we should isolate these two types of workloads to prevent the interference. By optimizing the bandwidths sequential workloads receive, we are optimizing the disk bandwidth and thus utilization.

5.4 Differential IO Scheduling

In designing a cloud storage system offering block-level storage abstraction, we have the following assumptions. First, the system is built from many inexpensive commodity components that have a failure rate high enough that replication is necessary to offer high availability and durability. Second, all nodes in the cloud are within a single data center, interconnected by high speed Ethernet with sufficient bisection bandwidth that this is not a major concern. Third, no assumption is made on the storage workload from VMs. The workload may be database queries made of many small random reads, or map-reduce jobs with mostly sequential IOs and some random IOs. A particular VM may change its workload characteristics over time. Finally, given the large user base and diverse workloads, we assume new random and sequential workloads constantly come into the system and at any given time, the system needs to serve both types of workloads.

The basic idea in DIOS is to partition the data nodes in a storage cluster into replication groups, each replica within a replication group serving a particular type of read requests[3]: each disk serves either random read requests or sequential read requests, but not both

[3] We do not deal with write requests in this work and leave it as future work.

simultaneously. Since we isolate random read requests from sequential ones, data nodes serving sequential workloads remain high disk utilization, leading to good and predictable performance for sequential workloads. Since we partition the cluster with some data nodes reserved for serving only sequential workloads, the performance for random workloads may not be as good as the baseline where all data nodes can be used for serving random read requests.

Figure 5.4 presents the overall architecture for a cloud storage system, providing block-level abstraction with DIOS. Data nodes are partitioned into replication groups of 2 data nodes each, assuming a replication factor of 2. The cloud storage system exports a block-level interface for upper-layer applications, and the block device driver is running at each compute node. DIOS is implemented within the block device driver. For each write, it is stored in a replication group by replicating the write request at each data node. When the driver receives a read IO request from an application, it detects the type of this request and schedules this request based on its type: if it is a random read request, it is sent to the first data node. Otherwise, the read request is sent to the second data node. The data is returned by the corresponding data node, after it receives the read request. Note that this scheduling algorithm can be extended for other replication factors as well. For example, given a replication factor of 3, we could reserve the first two to serve random requests and the last for serving sequential ones.

The architecture of Ceph [32] resembles the overall architecture we have designed for DIOS, and thus we modified Ceph by adding the support for DIOS.

5.4.1 Ceph Background

Ceph is a unified distributed storage system that provides storage service with three abstractions: files, block devices, and objects. All these abstractions are built on a common subsystem: its object store, called RADOS [99]. RADOS implements the primary-copy replication scheme to improve the reliability of objects stored in the system. RADOS is also scalable in that there is no centralized metadata service for managing the location of a data block. Instead, the topology map of the storage cluster is used at each node, and this map is also used to determine the location of a data block independently. Whenever there are new nodes added into the cluster or faulty nodes removed from the cluster, the topology map is updated. Data is redistributed/replicated automatically to reach a predefined level

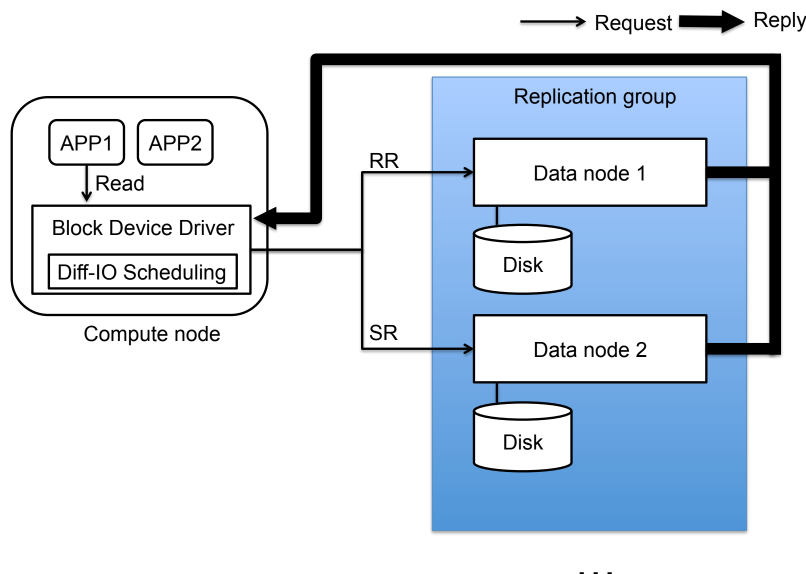


Figure 5.4. Overall system architecture.

of redundancy.

Each data node in Ceph is called an Object Storage Device (OSD). A set of OSDs is grouped into a placement group (PG, similar to a replication group, as we discussed previously), within which objects are replicated. An OSD can be a member in multiple PGs, in which the role of that OSD may be different: in one PG, an OSD might be the primary replica for an object, while in another PG, it might be the second replica or the third. For an object, it is first mapped to a placement group, mainly based on its name. Then the placement group is mapped to an order list of r distinct OSDs, using a pseudo-random replica distribution algorithm, called CRUSH [100]. For a replication factor of 3, the mapping from objects to PGs and PGs to OSDs are presented in Figure 5.5.

The first replica in a placement group is called the primary replica. In Ceph, a read request is always sent to and served by the primary replica. Since each data node can be the primary replica in some placement groups, they can receive both types of read requests, leading to poor disk utilization. For a write request, it is sent to the primary replica and the primary replica forwards this write request to other replicas in the same placement group for replication.

To support DIOS in Ceph, we extended CRUSH by adding a new replica placement algorithm called *direct-map* and changed the driver so that it schedules read requests to

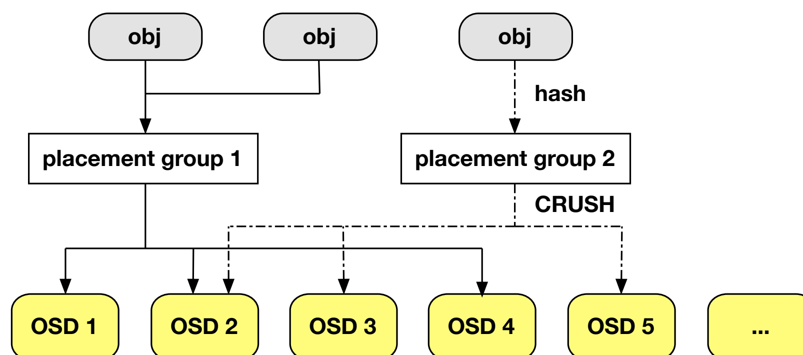


Figure 5.5. The mapping among objects, placement groups (PGs), and object storage devices (OSDs).

different replicas based on request bytes. We discuss each in turn.

5.4.2 Deterministic Replica Placement

At a high level, CRUSH works as follows. The cluster topology map is organized in a hierarchy: the root node stands for the cluster, which contains C server cabinets. Each cabinet in turn contains S disk shelves and each shelf contains D storage devices. When CRUSH is asked to find r distinct storage devices, it starts the iteration from the root node: find r storage device from this root node. Then, it finds this node contains cabinets, instead of storage devices. It continues the iteration for each cabinet contained in that root node: find r/C distinct storage devices randomly from this cabinet. Shelves are found to be contained in that cabinet. Then, for each shelf, it finally finds storage devices and returns $r/C/S$ distinct ordered storage devices randomly. Finally, an ordered list of r replicas are aggregated at the root node. This is simplified in that we assume equal weights for each cabinet and shelf. Depending on the cluster's configuration, we could assign weights for cabinets and shelves appropriately.

For load-balancing, CRUSH randomly selects some number of items at each level. As a result, a data node could appear as any replica in a placement group. However, in DIOS, each data node is supposed to have one designated position in a placement group. To achieve that determinism, we modified CRUSH to support a new replica selection algorithm, which we called *direct-map*. It works as following: to find m items from a set with n items, it first sorts and partitions n items into groups of m items. Then it randomly picks one group and returns all m items in order.

direct-map is a general replica selection algorithm that is implemented within the CRUSH framework and could be applied at any level. But to support DIOS, we have to use it at the root level. For levels below the root node, standard random replica selection algorithms can be used. This is sufficient for our environment since there are only two levels in our environment: the cluster contains data nodes directly. We apply *direct-map* at the root level and it returns a pair of OSDs (e.g., {OSD1, OSD2} or {OSD3, OSD4}) for a placement group.

5.4.3 Workload Type Detection

We also need a way to detect sequential and random workloads. We implemented the most intuitive one: when the logical block address of a request is contiguous with the previous one, the request is treated as sequential. To deal with the interleaves from multiple access streams, we use a similar approach as Gulati et al. [35] took in calculating seek distance. Instead of remembering only one previous request, we remember an array of n noncontiguous requests (precisely, we remember n next expected offsets. The value of n is configurable; we used 16 for our experiments). When the current request is contiguous to one of expected offsets, the value for that expected offset is updated based on the current request. Otherwise, we calculate the expected offset based on this request and add this to the array (replace an existing one in round robin when the array is full).

5.5 Evaluation

Table 5.1 shows the configuration of a single d820 server we used in our experiments. These machines are hosted in the Emulab network testbed, hosted by the FLUX research group at the University of Utah. Each disk can provide 154 MB/s for 64 KB sequential reads and 668 IOPS for 8 KB random reads at an iodepth of 32. Two d820 machines are used as data nodes and another machine is used as the client machine to generate workloads. At each data node, we use the first 3 as data disks and the remaining 3 as journal disks.[4] In total, we have 6 data disks. We used ext4 as the file system for each disk. We implemented our *direct-map* replica selection algorithm in Ceph at version 0.56.3 (the second stable release - Bobtail). The sequential detection algorithm was implemented in the RADOS

[4]Ceph implements journaling for consistency.

Table 5.1. The configuration of a d820 machine

Processors	4 × 2.2 GHz 8-Core Intel(R) Xeon(R) E5-4620
Memory	128 GB DDR3 RAM
Disk	6× 10K RPM TOSHIBA MBF2600RC SCSI disks

block driver included in the Linux kernel 3.2.6.

To demonstrate the benefit of DIOS, we ran 20 sequential read workloads, with 10, 20, and 40 random read workloads respectively. We reported aggregated bandwidths for 20 sequential read workloads and aggregated IOPSs for random workloads. The workloads are simulated with FIO, and IO requests are issued to a 2.6 GB file (the data file of the mysql database, for TPC-H at scale of 1). The IO size is 64 KB for SR and 8 KB for RR. We maintain 32 outstanding IOs. The results are presented in Figure 5.6.

Figure 5.6(a) shows that in the baseline Ceph, aggregated bandwidths for sequential workloads dropped significantly, as we added more random workloads into the system: it started with an aggregated bandwidth of 240 MB/s when there are no random workloads running concurrently and dropped to be only 18.54 MB/s when 40 RRs were added. Bandwidths for sequential workloads were impacted significantly by co-located random workloads. On the other hand, since we partition the disks among random and sequential workloads in DIOS, such interference is prevented, and sequential workloads received consistent bandwidths, with an aggregated bandwidth stable at about 120 MB/s. Sequential workloads got higher bandwidths in DIOS than the baseline Ceph when there were 10, 20, or 40 random workloads running concurrently (e.g., it is more than 6× higher than the baseline Ceph at 40 RRs). This is the type of workload mixes that DIOS is designed for and from the experiment, we did see that DIOS provides consistent and high performance for sequential workloads.

Figure 5.6(b) shows the aggregated IOPS for random workloads. With DIOS, IOPSs for random workloads were lower than the baseline. This is expected, since we now only have half of the disks serving random workloads. When 40 random workloads were running concurrently, there was a drop in the aggregate IOPS in both the baseline Ceph and Ceph with DIOS. We believe this is because when 40 random workloads are introduced into the system, disks need to serve random IOs across a larger area, leading to longer seek times

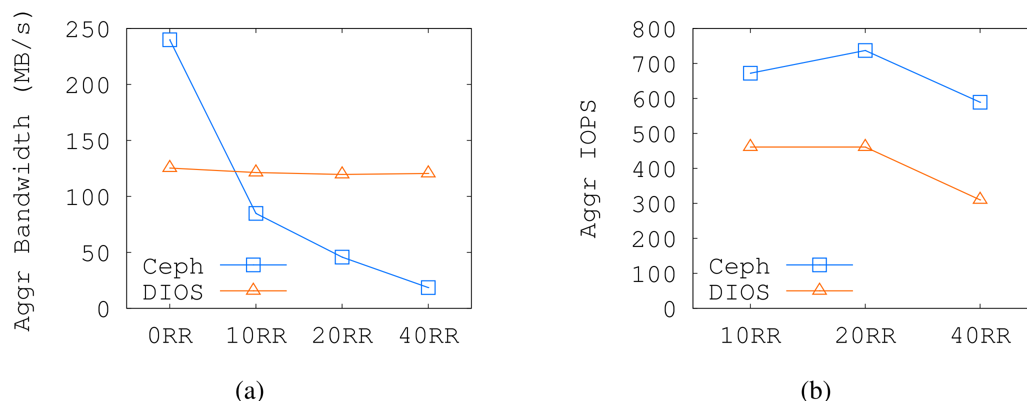


Figure 5.6. Comparison between the baseline Ceph and Ceph with DIOS. In (a), we showed aggregated bandwidths of 20 sequential workloads when they were run concurrently with 0, 10, 20, or 40 random workloads respectively. In (b), we showed aggregated IOPSs of random workloads, when they were run concurrently with 20 sequential workloads.

and a lower IOPS.

Overall, we show that for cloud environments that a large number of diverse workloads are running concurrently, random workloads could lead to destructive interference for sequential ones, leading to inefficient performance. DIOS successfully isolates random requests from sequential ones and provides consistently high bandwidths for sequential workloads, leading to high disk utilization and efficient performance.

5.6 Related Work

Performance variation and unpredictability is a well-known problem in shared cloud environments. Shripad et al. [94] pointed out the performance of Amazon EBS and S3 could vary greatly at different times of day. Gulati et al. [35] noticed that random workloads interfered with sequential workloads significantly. Some attacks, such as resource-freeing attacks [101], are also possible to improve a tenant's performance at a co-located tenant's expense.

There is much related work on providing QoS-based resource allocation for storage, such as Stonehenge [102], Argon [103], and Aqua [104]. In particular, Stonehenge provides multidimensional storage performance virtualization by translating other metrics into disk bandwidth guarantees. Bandwidth reservation is guaranteed with a novel disk scheduler using virtual clock request scheduling. To provide performance insulation, Argon partitions

the cache and schedules disk requests based on time slicing. All of these algorithms are to allocate throughput or bandwidth in proportion to the pre-defined weights.

Further proposals provide additional support for latency-sensitive applications (BVT [105], SMART [106], pClock [107]). Furthermore, mClock [108] borrows the concept of reservation and limit from CPU and memory scheduling to storage, in addition to the proportional weight-based allocation. mClock uses two schedulers: the constraint-based scheduler ensures no variation regarding minimum reservation and upper limit; the weight-based scheduler tries to allocate resources in proportion to weights.

Some other work has looked into proportional resource sharing and load balancing problems in distributed storage systems. PARDA [109] used the AIMD approach used for congestion control in TCP to achieve proportional resource allocation for distributed storage systems. In BASIL [110], the authors proposed a linear model for characterizing the load of a workload and device capacity modeling. Their system migrates workloads across devices to achieve load-balancing.

These works typically abstract the storage device to a single block device, such as a physical disk, a LUN, or a RAID device. They rely on the lower layer to deal with replications, and replication is a black-box to these approaches. In contrast, we propose to leverage the replication information during scheduling to isolate the interference between random and sequential workloads, improving effective disk utilization.

Providing predictability and performance isolation is an important subject in multi-tenant cloud database management systems (DBMSs), as seen in recent works from the database community [111, 112]. Nevertheless, these works studied these problems from a higher-level abstraction, and in particular, at the DBMSs layer, which is clearly different from our approach focusing on the low-level block level storage. DIOS clearly has important applications in those domains.

One recent work that came very close to DIOS is Flash on Rails [113]. While flash-based Solid-state drives provide high performance, read requests could be blocked by write requests, leading to unpredictable performance. For achieving consistent performance for flash storage, they proposed utilizing replication: data is replicated at several drives and during each interval, a certain number of drives within a sliding window serve only read requests while the remaining drives serve writes. For drives that serve only read requests,

write requests are buffered in memory. The sliding window is moved at every interval so that one drive is moved out of the sliding window and a new drive is added. The new drive starts to serve read requests, while write requests buffered in memory for the old drive are then flushed. This approach is very similar to what we did in DIOS: we also partition the hard drives to serve different types of read requests while they partition SSDs to serve read or write requests.

5.7 Summary

In this chapter, we reveal the performance interference problem among different tenants when co-locating random and sequential read workloads in the cloud. Then we propose a new scheduling algorithm, DIOS, which partitions a replicating storage cluster into two groups, with each serving one particular type of requests. Our prototype implementation based on Ceph demonstrated that DIOS can provide a strong isolation between sequential and random workloads, achieving consistent and high performance for sequential workloads. By carefully separating and scheduling different types of workloads in a multitenant cloud environment, the efficiency of the whole system is improved. Future work includes the evaluation of DIOS for different cloud applications, such as multitenant cloud database management systems and multitenant cloud hosting services and extension for different replication factors.

CHAPTER 6

CONCLUSION

6.1 Summary of the Dissertation

Digital data is continuing to grow exponentially, and it is important to investigate techniques to improve space efficiency for storage systems. By improving space efficiency, one can store more information without increasing investment in storage hardware. Compression and deduplication are two methods commonly used to improve space efficiency. This dissertation addresses some limitations in compression and deduplication, improving their effectiveness in data reduction by using similarity in content. Cloud storage is being widely used. However, users suffer from unpredictable and inefficient performance because of interference between co-located workloads. This dissertation proposes a new scheduling algorithm to address this problem by using similarity in access patterns. Overall, this dissertation has showed that similarity in content and access patterns can be utilized to improve space efficiency and performance for storage systems.

In Chapter 2, we address the scalability limitation in detecting redundant information for traditional compressors. Traditional compressors search for redundancy in a fine granularity (the string level), and this approach does not scale: the larger the window size we use, the longer the compressor takes to detect redundancy. The maximal window size that is in practical use is only 1 GB. To deal with limitation, we propose Migratory Compression: it works by detecting similarity in the block level and then grouping similar blocks together. After this data reorganization, traditional compressors can detect redundancy with small window sizes. With experiments using real-world datasets, we have shown that Migratory Compression improves compressibility and frequently runtime. Migratory Compression is also generic: it can be used for any standard compressor to improve their compression performance.

In Chapter 3, we identify many file formats used in deduplication environments that are sub-optimal and suffer from a common problem: metadata is interleaved with data.

Metadata changes more frequently, and this introduces many unnecessary unique chunks, reducing the benefit from deduplication. We propose to separate metadata from data and store metadata separately. When it is possible to change file formats, we recommend designing and using deduplication-friendly formats. When it is challenging to change file formats because of compatibility issues, we propose doing application-level postprocessing to transform the input from its original format into a deduplication-friendly format. When either of these approaches does not work, we can modify deduplication systems to be aware of input file formats. The last approach requires more engineering efforts, and we recommend to take this approach as a last resort. For each approach, we present real-world case studies and evaluate the benefit from separating metadata from data. Significant improvements in deduplication have been demonstrated.

In Chapter 4, we present a case study where we use a deduplicating storage system to improve space efficiency in storing disk images. One of the main challenges is how to use compression in the integrated system to achieve efficient storage and image deployment simultaneously. We analyze three alternatives and propose to store compressed chunks into the deduplicating storage system. We also propose a new chunking algorithm called Aligned Fixed-size Chunking (AFC). AFC allows us to do fixed-size chunking for allocated sectors for disk images. It provides better deduplication than traditional fixed-size chunking and runs considerably faster than variable-size chunking, providing another option when selecting a chunking algorithm. With a combination of using four techniques, we demonstrate that it is possible to use a deduplicating storage system to store disk images, achieving similar high performance in image deployment as the original system does.

Lastly, we applied the same idea of utilizing similarity, in the context of IO scheduling in Chapter 5. Our analysis reveals random workloads have a significant impact on disk bandwidth and utilization. Co-locating random workloads with sequential ones leads to unpredictable and inefficient performance. To deal with this problem, we propose to schedule different types of requests to different replica disks so that each disk is serving only the same type of requests. This separation prevents interference between random and sequential read workloads, resulting in consistent and efficient performance for sequential read workloads.

To summarize, Migratory Compression utilizes *similarity* in content to improve com-

pression. To improve the effectiveness of deduplication, we propose separating metadata from data and storing the *same* type of data together. Furthermore, we exploit *similarity* across disk images and present a use case of using deduplication storage systems to improve space efficiency in storing disk images. From the above three pieces of work, we show that similarity in content can be used to improve space efficiency. In Differential IO Scheduling, we schedule *same* type of requests to the same disk, achieving consistent and more efficient performance. We show similarity in access patterns can be utilized to improve performance for storage systems.

6.2 Future Research Directions

We believe techniques proposed in this dissertation are practical and can be used in future file systems and storage systems. We discuss briefly a few topics that are of particular interests to explore.

- **Migratory Compression**: In this work, we use an existing tool from a prior project [114], to do chunking and produce superfeatures. It uses a specific algorithm in calculating superfeatures. We have not examined other algorithms in calculating superfeatures. It would be interesting to know the interaction between different compression algorithms and superfeature calculation algorithms. Specifically, we are interested to know which superfeature algorithm can bring in the greatest improvement in compression for a specific compressor.

  We also present a case study using Migratory Compression in data migration between a backup system and an archive system. In general, storage systems that use some kinds of staging area to buffer writes temporarily can be enhanced with Migratory Compression to improve space efficiency. It has become common to use nonvolatile memory in storage systems to absorb write requests [4, 90]. Shingled Magnetic Recording (SMR) drives do not support random writes. Instead, they use a persistent write cache internally [115] to cache write requests and move them to their permanent locations later. For such systems, we can add Migratory Compression when we move data from the staging area to their permanent locations. Read-back performance needs to be considered when evaluating Migratory Compression in such systems. Solid-State Disks may help in improving read-back performance. It is interesting to

explore what the performance degradation is when we add Migratory Compression, for SSD- or HDD- based storage systems respectively.

We mainly use backup datasets to evaluate Migratory Compression. It would be interesting to evaluate Migratory Compression with other types of data, as well. A few examples include scientific datasets [116,117], large corpora of web pages [118], images and videos, and human genomes. It would also be interesting to develop some sampling techniques [119–121] to estimate the potential benefit of Migratory Compression for a particular given input, without actually running it.

Lastly, `mzip` also has potential performance improvements, such as multithreading and reimplementing in a more efficient programming language. Using GPUs [122–126] to accelerate the process in chunking, calculating fingerprints and super-features, and detecting similar chunks would also be an interesting area to explore.

- **Separating Metadata From Data**: We use semantic information from file formats to distinguish metadata from data and then store metadata separately from data. A more general approach would be to detect data hotness: design some algorithms to detect hot data that is more likely to change and then store hot data separately from cold data. Separating metadata from data can be considered as a special case for this principle: metadata is changed more frequently, and can be classified as hot data and we can consider data blocks as cold data. A few hot and cold data classification algorithms have been proposed and applied in improving the performance of storage systems [127, 128]. We can possibly apply these techniques to detect data hotness and use that information to reorganize data to improve deduplication.

  We also mainly examine file formats used in backup environments. Other file formats, such as video formats, suffer from a similar problem. Dewakar et al. [129] identified a video file interleaves text, audio, and video information. When we encode a video file in different languages (English, Chinese, etc.), the video data remains the same but the text and audio data needs to be changed. This leads to poor deduplication across different versions of the same video. In their work, they proposed two new chunking algorithms that utilize file format information to determine chunk boundaries. Another possible approach is to design some data

transformation technique, similar to Migratory Compression and Migratory Tar, to group the same type of data together. We believe this will improve deduplication and is worth exploring.

This study also indicates that a more thorough investigation is needed to re-examine a few use cases (databases and HTML web pages, etc.) that have shown poor deduplication. We might be able to identify the root cause for poor deduplication in these use cases and then be able to construct our solution so that we can also get good deduplication.

- **Deduplicating Storage for Efficient Disk Image Deployment** We present a case study of using deduplication for high-performance image deployment in a large network testbed. It would be very interesting to apply the same set of design principles in other image deployment systems, such as Glance [77] in OpenStack. We believe we could be able to improve the space efficiency in storing virtual machine images and performance in image deployment significantly, if we use the techniques we have proposed in our study.

- **Differential IO Scheduling** In this study, we assume cloud storage systems are based on hard disk drives. We find random workloads have a significant performance impact on disk utilization. This will remain true for the new generation of hard disk drives based on Shingled Magnetic Recording (SMR). SMR drives use the same mechanical mechanism as traditional hard disk drives and the only difference is tracks overlap with each other in SMR drives. DIOS could be applied for cloud storage systems based on SMR drives as well.

  As technologies for flash memory continue to advance and the cost continues to drop, flash-based Solid State Drives (SSDs) are getting popular. SSDs are based on circuits and are different from traditional mechanical hard disk drives. Our initial measurement shows that a consumer-level Samsung SSD provides reasonably good performance even for random workloads: 200 MB/s for random read workloads, versus 400 MB/s for sequential ones. We believe random workloads have much less impact on disk utilization for SSDs, and the benefit from DIOS may become limited. However, SSDs suffer from another problem: write requests can stall subsequent

read requests to the same flash chip, and there are some proposals [113], very similar to DIOS, which separate read requests from write ones to achieve predictable performance.

Ceph is not performing well for sequential workloads in baseline. It only achieves a maximum bandwidth of 250 MB/s, given that we have 6 data disks and each disk can give us a 150 MB/s bandwidth. So, we are only getting a maximum of 27% disk utilization in Ceph, even when we are running just sequential workloads. This leaves us a very small space to improve. We believe we can achieve better results if Ceph could provide much better performance for sequential workloads in its baseline. We instrument Ceph and find it spends a considerable amount of time in concurrency control (locking). We believe as Ceph becomes mature and more optimization techniques, such as lock-free algorithms, are introduced, we can see performance improvements, and by that time, the benefit from DIOS will become more significant. It would also be interesting to apply DIOS in other replication systems as well, such as RAID0.

REFERENCES

[1] I. D. Corporation, "The digital universe of opportunities: Rich data and the increasing value of the internet of things," Apr. 2014. [Online]. Available: http://www.emc.com/leadership/digital-universe/2014iview/index.htm

[2] I. Data Corporate, "Worldwide purpose-built backup appliance 2012-2016 forecase and 2011 vendor shares," 2011. [Online]. Available: http://www.emc.com/collateral/analyst-reports/idc-worldwide-purpose-built-backup-appliance.pdf

[3] Amazon Web Services LLC, "Amazon machine images (AMIs)," 2015. [Online]. Available: http://aws.amazon.com/

[4] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, pp. 18:1–18:14, [copyright 2008] © [USENIX]. [Online]. Available: http://dl.acm.org/citation.cfm?id=1364813.1364831

[5] I. C. Tuduce and T. Gross, "Adaptive main memory compression," in *Proceedings of the 2005 USENIX Annual Technical Conference (USENIX ATC '05)*, Anaheim, CA, Apr. 2005, pp. 237–250.

[6] Z. Li, M. Chen, A. Mukker, and E. Zadok, "On the trade-offs among performance, energy, and endurance in a versatile hybrid drive," in *the ACM Transactions on Storage*, vol. 11, no. 3, Jul. 2014, pp. 13:1–13:27. [Online]. Available: http://doi.acm.org/10.1145/2700312

[7] Z. Li, A. Mukker, , and E. Zadok, "On the importance of evaluating storage systems' costs," in *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '14)*, [copyright 2014] © [USENIX]. [Online]. Available: http://dl.acm.org/citation.cfm?id=2696578.2696584

[8] Z. Li, "GreenDM: A versatile tiering hybrid drive for the trade-off evaluation of performance, energy, and endurance," Ph.D. dissertation, State University of New York at Stony Brook, 2014.

[9] Z. Li, K. M. Greenan, A. W. Leung, and E. Zadok, "Power consumption in enterprise-scale backup storage systems," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, Feb. 2012, pp. 65–72.

[10] Z. Li, R. Grosu, P. Sehgal, S. A. Smolka, S. D. Stoller, and E. Zadok, "On the energy consumption and performance of systems software," in *Proceedings of 4th Annual International Systems and Storage Conference (SYSTOR '11)*, [copyright 2011] © [ACM]. [Online]. Available: http://doi.acm.org/10.1145/1987816.1987827

[11] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace, “Nitro: A capacity-optimized ssd cache for primary storage,” in *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC ’14)*, Philadelphia, PA, Jun. 2014, pp. 501–512.

[12] J. Ziv and A. Lempel, “A universal algorithm for sequential data compression,” in *IEEE Transactions on Information Theory*, vol. 23, no. 3, May 1977, pp. 337–343.

[13] T. Makatos, Y. Klonatos, M. Marazakis, M. D. Flouris, and A. Bilas, “Using transparent compression to improve SSD-based I/O caches,” in *Proceedings of the 5th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys ’10)*, [copyright 2010] © [ACM]. [Online]. Available: http://doi.acm.org/10.1145/1755913.1755915

[14] M. Burrows, C. Jerian, B. Lampson, and T. Mann, “On-line data compression in a log-structured file system,” in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS ’92)*, [copyright 1992] © [ACM]. [Online]. Available: http://doi.acm.org/10.1145/143365.143376

[15] P. Deutsch, “DEFLATE Compressed Data Format Specification version 1.3,” Internet Engineering Task Force, 1996. [Online]. Available: http://www.ietf.org/rfc/rfc1951.txt

[16] J. Gilchrist, “Parallel data compression with bzip2,” in *Proceedings of the 16th International Conference on Parallel and Distributed Computing and Systems*, vol. 16, Cambridge, MA, Nov. 2004, pp. 559–564.

[17] I. Pavlov, “7-zip,” 2013. [Online]. Available: http://www.7-zip.org/

[18] L. Collin, “XZ Utils,” 2013. [Online]. Available: http://tukaani.org/xz/

[19] S. Quinlan and S. Dorward, “Venti: A new approach to archival data storage,” in *Proceedings of the Conference on File and Storage Technologies (FAST ’02)*, [copyright 2002] © [USENIX]. [Online]. Available: http://dl.acm.org/citation.cfm?id=1083323.1083333

[20] B. H. Bloom, “Space/time trade-offs in hash coding with allowable errors,” in *Communications of the ACM*, Jul. 1970, pp. 422–426.

[21] W. Xia, H. Jiang, D. Feng, and Y. Hua, “Silo: A similarity-locality based near-exact deduplication scheme with low ram overhead and high throughput,” in *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC ’11)*, Portland, OR, Jun. 2011, pp. 285–298.

[22] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezise, and P. Camble, “Sparse indexing: Large scale, inline deduplication using sampling and locality,” in *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST ’09)*, San Francisco, CA, Feb. 2009, pp. 111–123.

[23] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, F. Huang, and Q. Liu, "Accelerating restore and garbage collection in deduplication-based backup systems via exploiting historical information," in *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*, Philadelphia, PA, Jun. 2014, pp. 181–192.

[24] M. Fu, D. Feng, Y. Hua, X. He, Z. Chen, W. Xia, Y. Zhang, and Y. Tan, "Design tradeoffs for data deduplication performance in backup workloads," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, Santa Clara, CA, Feb. 2015, pp. 331–344.

[25] M. Lillibridge, K. Eshghi, and D. Bhagwat, "Improving restore speed for backup systems that use inline chunk-based deduplication," in *Proceedings of the 11th USENIX Conference on File and Storage Technologies (FAST '13)*, San Jose, CA, Feb. 2013, pp. 183–197.

[26] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb, "Fast, scalable disk imaging with Frisbee," in *Proceedings of the 2003 USENIX Annual Technical Conference (USENIX ATC '03)*, San Antonio, TX, Jun. 2003, pp. 283–296.

[27] Amazon Web Services LLC, "Amazon Web Services," 2015. [Online]. Available: http://aws.amazon.com

[28] Google, "Google Cloud Platform," 2015. [Online]. Available: https://cloud.google.com/

[29] Microsoft, "Microsoft Azure," 2015. [Online]. Available: http://azure.microsoft.com/en-us/

[30] E. Pinheiro, W.-D. Weber, and L. A. Barroso, "Failure trends in a large disk drive population," in *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 2007)*, San Jose, CA, Feb. 2007, pp. 17–28.

[31] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, Bolton Landing, NY, Oct. 2003, pp. 29–43.

[32] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, WA, Nov. 2006, pp. 307–320.

[33] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," Electrical Engineering and Computer Sciences, University of California at Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb. 2009.

[34] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," in *Proceedings of the 36th International Conference on Very Large Data Bases*, Singapore, Sep. 2010, pp. 460–471.

[35] A. Gulati, C. Kumar, and I. Ahma, "Storage workload characterization and consolidation in virtualized environments," in *Proceedings of the 2nd International Workshop on Virtualization Performance: Analysis, Characterization, and Tools*, [copyright 2009] © [IEEE]. [Online]. Available: https://labs.vmware.com/download/10/

[36] I. Ahmad, J. M. Anderson, A. M. Holler, and V. M. Rajit Kambo, "An analysis of disk performance in VMware ESX server virtual machines," in *Proceedings of the 6th Annual IEEE Workshop on Workload Characterization*, [copyright 2003] © [IEEE]. [Online]. Available: https://www.vmware.com/pdf/wwc_performance.pdf

[37] X. Lin, G. Lu, F. Douglis, P. Shilane, and G. Wallace, "Migratory compression: Coarse-grained data reordering to improve compressibility," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '13)*, Santa Clara, CA, Feb. 2014, pp. 256–273.

[38] E. R. Fiala and D. H. Greene, "Data compression with finite windows," in *Communications of the ACM*, vol. 32, no. 4, Apr. 1989, pp. 490–505.

[39] A. Tridgell, "Efficient algorithms for sorting and synchronization," Ph.D. dissertation, Australian National University Canberra, 1999.

[40] J. J. Hunt, K.-P. Vo, and W. F. Tichy, "Delta algorithms: an empirical analysis," in *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 2, Apr. 1998, pp. 192–214.

[41] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey, "Redundancy elimination within large collections of files," in *Proceedings of the 2004 USENIX Annual Technical Conference (USENIX ATC '04)*, Boston, MA, Jun. 2004, pp. 59–72.

[42] M. Burrows and D. J. Wheeler, "A block-sorting lossless data compression algorithm," Digital Equipment Corporation, Tech. Rep. SRC-RR-124, 1994.

[43] S. Mathew, "Overview of amazon web services," 2014. [Online]. Available: https://d0.awsstatic.com/whitepapers/aws-overview.pdf

[44] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *SIGOPS Operating System Review*, vol. 35, no. 5, Oct. 2001, pp. 174–187.

[45] P. Shilane, G. Wallace, M. Huang, and W. Hsu, "Delta compressed and deduplicated storage using stream-informed locality," in *Proceedings of the 4th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '12)*, [copyright 2012] © [USENIX]. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342806.2342816

[46] S. Smaldone, G. Wallace, and W. Hsu, "Efficiently storing virtual machine backups," in *Proceedings of the 5th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '13)*, [copyright 2013] © [ACM]. [Online]. Available: http://dl.acm.org/citation.cfm?id=2534861.2534871

[47] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings of the Compression and Complexity of Sequences (SEQUENCES '97)*, Washington, DC, Jun. 1997, pp. 21–29.

[48] N. Jain, M. Dahlin, and R. Tewari, "Taper: Tiered approach for eliminating redundancy in replica synchronization," in *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST '05)*, San Francisco, CA, Dec. 2005, pp. 281–294.

[49] J. MacDonald, "File system support for delta compression," Master's thesis, Department of Electrical Engineering and Computer Science, University of California at Berkeley, 2000.

[50] D. G. Korn and K.-P. Vo, "Engineering a differencing and compression data format," in *Proceedings of the 2002 USENIX Annual Technical Conference (USENIX ATC '02)*, Monterey, CA, Jun. 2002, pp. 219–228.

[51] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu, "Characteristics of backup workloads in production systems," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, Feb. 2012, pp. 33–49.

[52] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," in *IEEE Transactions on Information Theory*, vol. 24, no. 5, 1978, pp. 530–536.

[53] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy, "Potential benefits of delta encoding and data compression for http," in *Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '97)*, Cannes, French Riviera, FRANCE, Sep. 1997, pp. 181–194.

[54] L. Aronovich, R. Asher, E. Bachmat, H. Bitner, M. Hirsch, and S. T. Klein, "The design of a similarity based deduplication system," in *Proceedings of the 2009 Israeli Experimental Systems Conference (SYSTOR '09)*, pp. 6:1–6:14, [copyright 2009] © [ACM]. [Online]. Available: http://doi.acm.org/10.1145/1534530.1534539

[55] X. Lin, F. Douglis, J. Li, X. Li, R. Ricci, S. Smaldone, and G. Wallace, "Metadata considered harmful ... to deduplication," in *Proceedings of the 7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '15)*, [copyright 2015] © [USENIX]. [Online]. Available: http://dl.acm.org/citation.cfm?id=2813749.2813760

[56] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "Hydrastor: A scalable secondary storage," in *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST '09)*, San Francisco, CA, Feb. 2009, pp. 197–210.

[57] K. Srinivasan, T. Bisson, G. Goodson, and K. Voruganti, "iDedup: Latency-aware, inline data deduplication for primary storage," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, Feb. 2012, pp. 299–312.

[58] S. Microsystems, "ZFS," 2008. [Online]. Available: http://en.wikipedia.org/wiki/ZFS

[59] W. J. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur, "Single instance storage in windows 2000," in *Proceedings of the 4th Conference on USENIX Windows Systems Symposium (WSS '00) - Volume 4*, [copyright 2000] © [USENIX]. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267102.1267104

[60] K. Jin and E. L. Miller, "The effectiveness of deduplication on virtual machine disk images," in *Proceedings of the 2009 Israeli Experimental Systems Conference (SYSTOR '09)*, pp. 7:1–7:12, [copyright 2009] © [ACM]. [Online]. Available: http://doi.acm.org/10.1145/1534530.1534540

[61] D. Meyer and W. Bolosky, "A study of practical deduplication," in *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST '11)*, San Jose, CA, Feb. 2011, pp. 1–14.

[62] Oracle Corp., "Database backup and recovery user's guide," 2015. [Online]. Available: http://docs.oracle.com/cd/E11882_01/backup.112/e10642/

[63] F. Douglis, D. Bhardwaj, H. Qian, and P. Shilane, "Content-aware load balancing for distributed backup," in *Proceedings of the 25th International Conference on Large Installation System Administration*, Boston, MA, Dec. 2011, pp. 151–168.

[64] gnu tar, "Basic Tar Format," 2015. [Online]. Available: http://www.gnu.org/software/tar/manual/html_node/Standard.html

[65] File systems and Storage Lab from Stony Brook University, "fs-hasher," 2015. [Online]. Available: http://tracer.filesystems.org/

[66] F. Guo and P. Efstathopoulos, "Building a high-performance deduplication system," in *Proceedings of the 2011 USENIX Annual Technical Conference (USENIX ATC '11)*, Portland, OR, Jun. 2011, pp. 271–284.

[67] C.-H. Ng and P. P. C. Lee, "Revdedup: a reverse deduplication storage system optimized for reads to latest backups," in *Proceedings of the 4th Asia-Pacific Workshop on Systems (APSys '13)*, Singapore, Singapore, pp. 15:1–15:7, [copyright 2013] © [ACM]. [Online]. Available: http://doi.acm.org/10.1145/2500727.2500731

[68] B. Sung, S. Park, Y. Oh, J. Ma, U. Lee, and C. Park, "An efficient data deduplication based on tar-format awareness in backup applications," [copyright 2013] © [USENIX]. [Online]. Available: https://www.usenix.org/system/files/fastpw13-final24.pdf

[69] X. Lin, M. Hibler, E. Eide, and R. Ricci, "Using deduplicating storage for efficient disk image deployment," in *EAI Endorsed Transactions on Scalable Information Systems*, vol. 15, no. 6. EAI, Aug 2015. [Online]. Available: http://eudl.eu/doi/10.4108/icst.tridentcom.2015.259963

[70] K. Atkinson, G. Wong, and R. Ricci, "Operational experiences with disk imaging in a multi-tenant datacenter," in *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, Seattle, WA, Apr. 2014, pp. 217–228.

[71] K. Sklower, Personal communication, Feb. 2015.

[72] K. R. Jayaram, C. Peng, Z. Zhang, M. Kim, H. Chen, and H. Lei, “An empirical analysis of similarity in virtual machine images,” in *Proceedings of the Middleware 2011 Industry Track Workshop*, [copyright 2011] © [ACM]. [Online]. Available: http://doi.acm.org/10.1145/2090181.2090187

[73] S. Rhea, R. Cox, and A. Pesterev, “Fast, inexpensive content-addressed storage in Foundation,” in *Proceedings of the 2008 USENIX Annual Technical Conference (USENIX ATC ’08)*, Boston, MA, Jun. 2008, pp. 143–156.

[74] L. P. Cox, C. D. Murray, and B. D. Noble, “Pastiche: Making backup cheap and easy,” in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI ’02)*, Boston, MA, Dec. 2002, pp. 285–298.

[75] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, “An integrated experimental environment for distributed systems and networks,” in *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI ’02)*, Boston, MA, Dec. 2002, pp. 255–270.

[76] OpenStack Foundation, “OpenStack,” 2015. [Online]. Available: http://www.openstack.org/

[77] ——, “OpenStack Glance,” 2015. [Online]. Available: http://docs.openstack.org/developer/glance/

[78] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala, “Opening black boxes: Using semantic information to combat virtual machine image sprawl,” in *Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE ’08)*, Seattle, WA, Mar. 2008, pp. 111–120.

[79] R. Schwarzkopf, M. Schmidt, M. Rüdiger, and B. Freisleben, “Efficient storage of virtual machine images,” in *Proceedings of ScienceCloud*, Delft, The Netherlands, Jun. 2012, pp. 51–60.

[80] A. Liguori and E. Van Hensbergen, “Experiences with content addressable storage and virtual disks,” in *Proceedings of the 1st Workshop on I/O Virtualization (WIOV ’08)*, San Diego, CA, Dec. 2008, pp. 51–60.

[81] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield, “Parallax: Virtual disks for virtual machines,” in *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys ’08)*, San Jose, CA, Mar. 2008, pp. 41–54.

[82] M. Shamma, D. T. Meyer, J. Wires, M. Ivanova, N. C. Hutchinson, and A. Warfield, “Capo: Recapitulating storage for virtual desktops,” in *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST ’11)*, San Jose, CA, Feb. 2011, pp. 31–46.

[83] J. G. Hansen and E. Jul, “Lithium: Virtual machine storage for the cloud,” in *Proceedings of the 1st ACM Conference on Cloud Computing (SoCC ’10)*, Indianapolis, Indiana, USA, Jun. 2010, pp. 15–26.

[84] J. Reich, O. Laadan, E. Brosh, A. Sherman, V. Misra, J. Nieh, and D. Rubenstein, "VMTorrent: Scalable P2P virtual machine streaming," in *Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '12)*, Nice, France, Dec. 2012, pp. 289–300.

[85] Z. Zhang, Z. Li, K. Wu, D. Li, H. Li, Y. Peng, and X. Lu, "VMThunder: Fast provisioning of large-scale virtual machine clusters," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 12, 2014, pp. 3328–3338.

[86] C. Peng, M. Kim, Z. Zhang, and H. Lei, "VDN: Virtual machine image distribution network for cloud data centers," in *IEEE International Conference on Computer Communications*, Orlando, FL, Mar. 2012, pp. 181–189.

[87] X. Zhao, Y. Zhang, Y. Wu, K. Chen, J. Jiang, and K. Li, "Liquid: A scalable deduplication file system for virtual machine images," in *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, 2014, pp. 1257–1266.

[88] C.-H. Ng, M. Ma, T.-Y. Wong, P. P. C. Lee, and J. C. S. Lui, "Live deduplication storage of virtual machine images in an open-source cloud," in *Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware (Middleware '11)*, Lisboa, Portugal, Dec. 2011, pp. 81–100.

[89] A. Burtsev, P. Radhakrishnan, M. Hibler, and J. Lepreau, "Transparent checkpoints of closed distributed systems in Emulab," in *Proceedings of the 4th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '09)*, Nuremberg, Germany, Mar. 2009, pp. 173–186.

[90] D. Hitz, J. Lau, and M. Malcolm, "File system design for an NFS file server appliance," in *Proceedings of the USENIX Winter 1994 Technical Conference (WTEC '94)*, [copyright 1994] © [USENIX]. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267074.1267093

[91] R. Pullakandam, "EmuStore: Large scale disk image storage and deployment in the Emulab network testbed," Master's thesis, University of Utah, Aug. 2014.

[92] X. Lin, Y. Mao, F. Li, and R. Ricci, "Towards fair sharing of block storage in a multi-tenant cloud," in *Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud '12)*, [copyright 2012] © [USENIX]. [Online]. Available: http://dl.acm.org/citation.cfm?id=2342763.2342778

[93] Amazon Web Services LLC, "Amazon elastic compute cloud (Amazon EC2)," 2011. [Online]. Available: http://aws.amazon.com/ec2/

[94] S. J. Nadgowda and R. Sion, "Cloud performance benchmark series: Amazon EBS, S3, and EC2 instance local storage," Cloud Commons Online, State University of New York at Stony Brook, Tech. Rep. [Online]. Available: http://digitalpiglet.org/research/sion2010cloud-storage.pdf

[95] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Securitys (CCS '09)*, Chicago, IL, Nov. 2009, pp. 199–212.

[96] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, [copyright 2004] © [USENIX]. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251254.1251264

[97] J. Axboe, "Flexible IO tester," 2012. [Online]. Available: http://freecode.com/projects/fio

[98] Transaction Processing Performance Council, "The TPC-H benchmark," 2012. [Online]. Available: http://www.tpc.org/tpch/

[99] S. A. Weil, A. W. Leung, S. A. Brandt, and C. Maltzahn, "RADOS: A scalable, reliable storage service for petabyte-scale storage clusters," in *Proceedings of the 2nd International Workshop on Petascale data storage (PDSW '07)*, Reno, Nevada, Nov. 2007, pp. 35–44.

[100] S. A. Weil, S. A. Brandt, E. L. Miller, and C. Maltzahn, "CRUSH: Controlled, scalable, decentralized placement of replicated data," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '06)*, [copyright 2006] © [ACM]. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188582

[101] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift, "Resource-freeing attacks: Improve your cloud performance (at your neighbors expense)," in *Proceedings of the 19th ACM Conference on Computer and Communications Securitys (CCS '12)*, Raleigh, NC, Oct. 2012, pp. 281–292.

[102] L. Huang, "Stonehenge: a high-performance virtualized ip storage cluster with qos guarantees," Ph.D. dissertation, State University of New York at Stony Brook, 2003.

[103] M. Wachs, M. Abd-El-Malek, E. Thereska, and G. R. Ganger., "Argon: Performance insulation for shared storage servers," in *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST '07)*, [copyright 2007] © [ACM]. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267903.1267908

[104] L. Krishnamurthy, "Aqua: an adaptive quality of service architecture for distributed multimedia applications," Ph.D. dissertation, University of Kentucky, 1997.

[105] K. J. Duda and D. R. Cheriton, "Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler," in *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, Charleston, SC, Dec. 1999, pp. 261–276.

[106] J. Nieh and M. S. Lam, "A smart scheduler for multimedia applications," in *ACM Transactions on Computer Systems*, vol. 21, no. 2, May 2003, pp. 117–163.

[107] A. Gulati, A. Merchant, and P. J. Varman, "pClock: an arrival curve based approach for QoS guarantees in shared storage systems," in *Proceedings of the ACM International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*, San Diego, CA, Jun. 2007, pp. 13–24.

[108] ——, "mClock: handling throughput variability for hypervisor IO scheduling," in *Proceedings of the 9th Symposium on Operating Systems Design and Implementation (OSDI '10)*, Vancouver, Canada, Oct. 2010, pp. 437–450.

[109] A. Gulati, I. Ahmad, and C. A. Waldspurger, "PARDA: Proportional Allocation of Resources for Distributed Storage Access," in *Proceedings of the 7th USENIX Conference on File and Storage Technologies (FAST '09)*, San Francisco, CA, Feb. 2009, pp. 85–98.

[110] A. Gulati, C. Kumar, and I. Ahmad, "BASIL: Automated IO load balancing across storage devices," in *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST '10)*, San Jose, CA, Feb. 2010, pp. 169–182.

[111] A. J. Elmore, S. Das, A. Pucher, D. Agrawal, A. El Abbadi, and X. Yan, "Characterizing tenant behavior for placement and crisis mitigation in multitenant dbmss," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*, New York, NY, Jun. 2013, pp. 517–528.

[112] V. R. Narasayya, S. Das, M. Syamala, B. Chandramouli, and S. Chaudhuri, "SQLVM: Performance isolation in multi-tenant relational database-as-a-service," in *Proceedings of the 6th Biennial Conference on Innovative Data Systems Research (CIDR '13)*, [copyright 2013] © [CIDR '13]. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=184020

[113] D. Skourtis, D. Achlioptas, N. Watkins, C. Maltzahn, and S. Brandt, "Flash on rails: Consistent flash performance through redundancy," in *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC '14)*, Philadelphia, PA, Jun. 2014, pp. 463–474.

[114] P. Shilane, M. Huang, G. Wallace, and W. Hsu, "WAN optimized replication of backup datasets using stream-informed delta compression," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, Feb. 2012, pp. 49–64.

[115] A. Aghayev and P. Desnoyers, "Skylight: A window on shingled disk operation," in *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST '15)*, Santa Clara, CA, Feb. 2015, pp. 135–149.

[116] Y. Su, "Big data management framework based on virtualization and bitmap data summarization," Ph.D. dissertation, The Ohio State University, 2015.

[117] Y. Su, G. Agrawal, J. Woodring, A. Biswas, and H.-W. Shen, "Supporting correlation analysis on scientific datasets in parallel and distributed settings," in *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC '14)*, Vancouver, Canada, Jun. 2014, pp. 191–202.

[118] C. Crawl, "Common crawl web crawling corpus," 2015. [Online]. Available: https://commoncrawl.org

[119] Y. Su, G. Agrawal, J. Woodring, K. Myers, J. Wendelberger, and J. Ahrens, "Effective and efficient data sampling using bitmap indices," in *Cluster Computing*, vol. 17, no. 4. Springer US, 2014, pp. 1081–1100.

[120] Y. Su, Y. Wang, and G. Agrawal, "In-situ bitmaps generation and efficient data analysis based on bitmaps," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '15)*, Portland, Oregon, Jun. 2015, pp. 61–72.

[121] F. Xie, M. Condict, and S. Shete, "Estimating duplication by content-based sampling," in *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC '13)*, San Jose, CA, Jun. 2013, pp. 181–186.

[122] P. Bhatotia, R. Rodrigues, and A. Verma, "Shredder: GPU-accelerated incremental storage and computation," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)*, San Jose, CA, Feb. 2012, pp. 171–186.

[123] W. Sun, R. Ricci, and M. L. Curry, "GPUstore: Harnessing GPU computing for storage systems in the os kernel," in *Proceedings of 5th Annual International Systems and Storage Conference (SYSTOR '12)*, [copyright 2012] © [ACM]. [Online]. Available: http://doi.acm.org/10.1145/2367589.2367595

[124] W. Sun and R. Ricci, "Fast and Flexible: Parallel packet processing with GPUs and Click," in *Proceedings of the 9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '13)*, San Jose, CA, Oct. 2013, pp. 25–36.

[125] W. Sun, "Harnessing GPU computing in system level software," Ph.D. dissertation, the University of Utah, 2014.

[126] W. Sun and R. Ricci, "Augmenting operating systems with the GPU," arXiv preprint arXiv:1305.3345.

[127] D. Park, "Hot and cold data identification: Applications to storage devices and systems," Ph.D. dissertation, The University of Minnesota, 2012.

[128] J. Lee and J.-S. Kim, "An empirical study of hot/cold data separation policies in solid state drives (SSDs)," in *Proceedings of 6th Annual International Systems and Storage Conference (SYSTOR '13)*, [copyright 2013] © [ACM]. [Online]. Available: http://dx.doi.org/10.1145/2485732.2485745

[129] S. Dewakar, S. Subbiah, G. Soundararajan, M. Wilson, M. Storer, K. K. Udayashankar, K. Voruganti, and M. Shao, "Storage efficiency opportunities and analysis for video repositories," in *Proceedings of the 7th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage '15)*, [copyright 2015] © [USENIX]. [Online]. Available: http://dl.acm.org/citation.cfm?id=2813749.2813761