Deduplication on Virtual Machine Disk Images

Technical Report UCSC-SSRC-10-01
September 2010

Keren Jin
kjin@cs.ucsc.edu

Storage Systems Research Center
Baskin School of Engineering
University of California, Santa Cruz
Santa Cruz, CA 95064
http://www.ssrc.ucsc.edu/

Filed as a masters thesis in the Computer Science Department of the University of California, Santa Cruz in September 2010. Some of the material in this thesis was published in SYSTOR 2009.

UNIVERSITY OF CALIFORNIA
SANTA CRUZ

DEDUPLICATION ON VIRTUAL MACHINE DISK IMAGES

A thesis submitted in partial satisfaction of the
requirements for the degree of

Master of Science

in

COMPUTER SCIENCE

by

Keren Jin

September 2010

The Thesis of Keren Jin
is approved:

Professor Ethan L. Miller, Chair

Professor Darrell D. E. Long

Doctor Mark W. Storer

Tyrus Miller
Vice Provost and Dean of Graduate Studies

Table of Contents

List of Figures **v**

List of Tables **viii**

Abstract **ix**

Dedication **xi**

Acknowledgments **xii**

1 Introduction **1**

2 Related Work **4**

2.1 Virtual Machines 4

2.2 Deduplication 5

2.3 Delta encoding 8

3 Deduplication **11**

3.1 Chunking 12

3.2 Deduplication 15

3.3 Experiments 18

3.3.1 Overall Impacts 22

3.3.2 Impact of Specific Factors 26

3.3.3 Impact of Package Management 33

3.3.4 Chunk Compression 40

4 Delta Encoding **42**

4.1 Similarity detection 43

4.2 Experiments 48

4.2.1 Overall Effectiveness 48

4.2.2 Effectiveness of Specific Parameters 51

4.2.3 Time consumption 54

5 Future work **57**
5.1 User generated data 57
5.1.1 Identical instances 57
5.1.2 Homogeneous instances 58
5.1.3 Heterogeneous instances 59
5.2 Security 60

6 Conclusion **61**

A Duplicate chunk sequences **64**
A.1 Motivation 65
A.2 Design 66
A.3 Discussions on the greedy algorithm 73
A.3.1 Postprocessing or inline deduplication 73
A.3.2 Minimum unit (postprocessing dedup only) 75
A.3.3 Detection order 76
A.3.4 Multi-threading 77

Bibliography **79**

List of Figures

3.1 Chunking a file with size 0xF300 bytes. 13

3.2 Share categories of chunks. Chunks seen for the first time must be stored; subsequent occurrences need not be stored. 16

3.3 Growth of data in different categories for 14 different Ubuntu 8.04 LTS instances. Stored data grows very slowly after the second VM is integrated. In these figures, stored data includes the first instance of each chunk, regardless of how it is shared. The sharp increase in size at the 5th VM is due to the use of a flat disk image, in which there are a large number of zero-filled (empty) sectors. 21

3.4 Growth of category data for 13 Unix and Linux virtual machines, using variable-size chunking, with an average chunk size of 1 KB. Unique data grows significantly as each disk image is added. As in Figure 3.3, the sharp increase in size at the 3rd VM is due to a large number of empty, zero-filled sectors in a flat disk image. 23

3.5 Effects of varying operating system type on deduplication. All experiments used 512 B variable-size chunks, except for the Linux experiment that uses 4 KB variable-size chunks. There is more intra-inter sharing in Linux than in BSD, indicating that the former is more homogeneous. . . 24

3.6 Cumulative distribution of chunks by count and total size. The upper line shows the cumulative number of chunks with total count of n or less, and the lower line shows the total space that would be consumed by chunks with count n or less. The data is from the *Linux* chunk store with chunk size 512 B. As the graph shows, most chunks occur fewer than 14 times, and few non-zero chunks appear more than 50 times in the set of disk images. 25

3.7 Effects of chunk size and fixed versus variable-size chunking on deduplication for a set of disk images including Ubuntu Server 6.10, 7.04, 7.10 and 8.04. In each group of two bars, the left bar measures fixed-size chunking and the right bar indicates variable-size chunking. The graph shows that smaller chunk sizes result in more effective deduplication. 26

3.8 Deduplication on different releases of a single Linux distribution. The left bar in each pair is fixed-size chunking and right bar is variable-size chunking; chunk size is 512 B. Consecutive releases have only slightly lower deduplication ratio than non-consecutive releases. 28
3.9 Locale versions (variable-size chunking, chunk size 512 B). The left bar in each pair only measures the English version and the right bar measures both English and French versions. Storing different locale versions of same distribution produces high deduplication. 29
3.10 Distribution lineage. Debian series comprises Debian, Ubuntu and Knoppix. Red Hat series comprises Fedora, CentOS and Mandriva. Variable-size chunking, chunk size 512 B. Despite the common lineages, there is a relatively low level of deduplication. 30
3.11 Deduplication of Web appliance VMs (variable-size chunking, chunk size 512 B). *AMP* stands for Apache, MySQL and PHP. Since the deduplication ratios for both cases are low when zero chunks are excluded, we can conclude that diverse operating systems with similar goals do not have many identical chunks of code. 30
3.12 Virtual machines created by different VMMs deduplicate well. The bar on the left measures fixed-size chunking, and the right-hand bar measures variable-size chunking. 32
3.13 Differing package installation orders. *p1a* denotes package set 1, ascending order, and others follow the same naming rule. Variable-size chunking, chunk size 512 B. Different package installation orders generate nearly identical disk images. 36
3.14 Package systems comparison on Ubuntu Server 8.04 (U), CentOS 5.2, no desktop (C) and Fedora 9, no desktop (F). Both install package set 1 in ascending order. Variable-size chunking, chunk size 512 B. Package system's contribution to deduplication is outweighed by OS heterogeneity. 37
3.15 Package removal orders. Variable-size chunking, chunk size 512 B. *rp* denotes removing only target packages. *rr* denotes removing target packages and all dependencies (revert to original). *ori* denotes the original disk image before package installation. Data of the removed packages are not erased from disk image, which still resemble the image with the installed packages. 39
3.16 Chunk-wise compression. Variable-size chunking, chunk size 512 B, except the rightmost bar for 4 KB. Chunk-wise compression is effective, but limited by small window size. 41

4.1 Example of delta encoding on two different chunk sequences. The window size is 3. **A**, **B**, **C** and **D** are data blocks in chunks. Each chunk is formed by concatenating two data blocks. The duplicate chunk **AA** acts as the center of the locality. The head of the arrows indicate the target of encoding, while the tail indicate the source. Chunks **CA**, **CB** and **CD** are similar as they sharing the common **C** data block. Chunks such as **BD** are not encoded because their group signatures are unique in the locality. 46
4.2 Effects of varying operating system type on delta encoding. In each group of two bars, the left bar measures the window size of 4 chunks and the right bar measures the window size of 8 chunks. 49
4.3 Effectiveness of delta encoding for varying the number of groups, number of signatures and number of candidates. All experiments are taken on the Ubuntu Server 6.10 through 8.04 chunk store. The size of signature blocks is fixed to 64 bytes. The pattern is similar to Difference Engine, except less distinctive. 51
4.4 Effectiveness of delta encoding for varying window sizes and size of signature blocks. The rest of the three parameters are fixed at $(2, 1), 1$. Small block size does not decrease space saving. 53
4.5 Relative ratios of the time consumption in the deduplication and delta encoding phases. Three chunk stores are experimented, regarding the block size and window size combinations in the parentheses on the X-axis. The rest parameters are fixed at $(2, 1), 1$. The letter before parentheses abbreviates the name of the chunk store: **l** for *linux*, **u** for *ubuntu6778* and **t** for *ubuntuLTS*. 55

A.1 Examples of level one locality. 69
A.2 Illustration of the progressive strategy. The longest sequence group has length 3. 70
A.3 Detection result of the first sequence group with the greedy strategy. In the second extending, 1:48:80 hits the implicit boundary, and becomes a sequence subgroup. 74

List of Tables

3.1 Parameters for variable-size chunking. 14
3.2 Virtual machine disk images used in the study. Under disk type, *M* is a monolithic disk image file, *X* is a disk image file split into 2 GB chunks, *S* is a sparse image file, and *F* is a flat image file. For example, *MS* would correspond to a single monolithic disk image formatted as a sparse image. 20
3.3 Package installation and removal sets. "asc" and "desc" refer to the order in which packages are installed. Image size is post-installation / removal. All sizes are in megabytes; the "before" and "after" columns reflect the number of installed packages before and after the installation / removal action. 33
3.4 Two package sets. 35
3.5 Compression parameters. 40

4.1 Meaning of the scheme parameters in our similarity detection method. . 46

Abstract

Deduplication on Virtual Machine Disk Images

by

Keren Jin

Virtual machines are becoming widely used in both desktop and variable servers to efficiently provide many logically separate execution environments while reducing the need for physical machines. While this approach utilizes free physical CPU resources, it still consumes large amounts of storage because each virtual machine (VM) instance requires its own multi-gigabyte disk image. Moreover, existing systems do not support *ad hoc* block sharing between disk images, instead relying on techniques such as overlays to build multiple VMs from a single “base” image.

Deduplication is a commonly used technique in archival storage systems and virtualization architectures. The concept of deduplication is similar to data compression, that finds identical instances of data blocks in a storage repository, and removes all such instances but one. Indexing is also employed to enable high-performance global identity detection. In an archival storage system, deduplication is an ideal approach to save disk I/O as well as storage space. In virtual machine host centers, deduplication causes homogeneous operating systems to share not only file system data, but also common memory pages. Furthermore, by identifying duplicate data within a spatial locality of each chunk, extra space saving can be achieved without much extra time invested.

To test the effectiveness of deduplication, we conducted extensive evaluations on different sets of virtual machine disk images with different chunking strategies. Our experiments found that the amount of stored data grows very slowly after the first few virtual disk images if only the locale or software configuration is changed, with the rate of compression suffering when different versions of an operating system or different operating systems are included. We also show that fixed-length chunks work well, achieving nearly the same compression rate as variable-length chunks. We also show that simply identifying zero-filled blocks, even in ready-to-use virtual machine disk images available online, can provide significant savings in storage. Finally, we propose an approach to incorporate delta encoding into regular deduplication as a post-processing step. Experimental results indicate as much space savings from delta encoding as from deduplication for certain virtual machines, while the extra time consumption is low.

To my beloved mother

Acknowledgments

First, I thank my dear mother for raising and supporting me through all the adventures of my life. She is always the first one to share the joys whenever I achieve even a tiny success, the first one to relieve me from depression whenever I'm stuck, and the first one to give me advices that help me get over difficulties. She is the mother of this thesis.

Second, I thank professor Ethan Miller and professor Darrell Long for their priceless advices in my academic field. Not only have they enlightened me while morphing from an inexperienced Bachelor of Science to a qualified computer science researcher, but also they imbued me with the knowledge that empowers me with the ability to make discoveries in the past, present and future.

Third, I thank Mark Storer, Kevin Greenan, Andrew Leung and all the colleagues from SSRC. Whenever Whenever discussed my work, their questions, answers and suggestions always inspired me to think differently and to explore a new way to solve problems. The creative atmosphere of the group was one of the essentials of the work.

Last, I thank my other friends, first, Peggy Pollard, who helped greatly with proofreading to the thesis. I thank Carl Bocchini, one of the my friends who introduced me both the American way of life and the American way of thinking. And I thank all the other friends I've met in this most important part of my study life.

Chapter 1

Introduction

In modern server farms, virtualization is being used to provide ever-increasing numbers of servers on virtual machines (VMs), reducing the number of physical machines required while preserving isolation between the machines. This approach better utilizes server resources, allowing many different operating system instances to run on a small number of servers, saving both hardware acquisition costs and operational costs such as energy, management and cooling. Individual VM instances can be separately managed, allowing them to serve a wide variety of purposes and preserving the level of control that many users want. However, this flexibility comes at a price: the storage required to hold hundreds or thousands of multi-gigabyte VM disk images and the inability to share identical data pages between VM instances.

One approach to saving disk space when running multiple instances of operating systems on multiple servers, whether physical or virtual, is to share files between them; *i. e.*, sharing a single instance of the `/usr/local/` files via network mount. This approach is incompatible with VM disk images, however, since they are usually monolithic space hogs, constantly changing, and their internal file structures are invisible to the underlying file system. Standard compression such as that provided by the Lempel-Ziv compression used in `gzip` [47], is ineffective because, while it can reduce the storage space used by a single disk image, it cannot eliminate commonalities between files.

Instead, others have proposed the use of deduplication to reduce the storage space required by the many different VM disk images that must be stored in a medium to large scale VM hosting facility [30]. While it seems clear that deduplication is a good approach to this problem, our research quantifies the benefits of using deduplication to reduce the storage space needed for multiple VM disk images. Our experiments also investigate *which* factors impact the level of deduplication available in different sets of VM disk images, some of which are under system control (*e. g.*, fixed versus variable-sized chunking and average chunk size) and some of which are dependent on the usage environment (*e. g.*, operating system version and VM target use). By quantifying the effects of these factors, our results provide guidelines for both system implementers and sites that host large numbers of virtual machines, showing which factors are important to consider and the costs of making design choices at both the system and usage level.

In reality, the chunk size is usually large enough to fit multiples of the block size of the underlying operating system. This can be a major threat to the overall space saving as deduplication occurs in granularity of chunks. We investigate the possibility of combining conventional compression and deduplication by doing delta encoding within a spatial locality of duplicate chunks, thereby identifying the duplicate sub-chunk data in “distinct” chunks. As the delta encoding is limited within short chunk sequences, real-time update has lower overhead, and the scheme could be further optimized per VM by bookkeeping statistical data. We also care about the extra-saving / extra-time ratio for the delta encoding. It is not worth investing so much time for small saving.

The paper is organized as follows: Chapter 2 reviews related work and background material on deduplication and virtual machines. Chapter 3 introduces our chunking and deduplication approaches to analyze VM disk images as well as evaluation of deduplication on sets of VM disk images for different purposes. Chapter 4 discusses and evaluates the delta encoding approach. Chapter 5 possible discusses directions for possible future work, and Chapter 6 summarizes our conclusions.

Chapter 2

Related Work

Our research studies the effectiveness of applying deduplication to virtual machine environments. In this section, we provide some background on both technologies.

2.1 Virtual Machines

Virtual machine monitors provide a mechanism to run multiple operating system instances on a single computer system. Systems such as Xen [7], VMware [41], and QEMU [8] provide a full execution environment to "guest" operating systems, using many techniques to convince each guest operating system that it has control of a full computer system.

One technique used by virtual machine monitors is the use of a file in the underlying "host" operating system to hold the contents of the guest's disk. These virtual disk images, whose sizes are specified when the virtual machine is created, can be either *flat* or *sparse*. Flat images are fixed-size files, with one block for each block in the guest's disk. Initially, unused blocks are zero-filled; thus, flat disk images tend to have a lot of zero-filled blocks, particularly if they are relatively large to allow the guest operating system space in which to store additional data. Sparse images, unlike flat images, only contain virtual disk blocks that have been written to at least once. Thus, sparse images can occupy relatively little space when created, and only grow as the guest operating system uses more and more of its virtual disk. Note, however, that sparse images *can* contain zero-filled blocks, since the guest operating system may write a block containing all zeros; such a block would be stored by the virtual machine in the disk image file. While there is no global standard for virtual machine disk images, specifications for one file format is available from VMware [42].

2.2 Deduplication

Deduplication is a technology that can be used to reduce the amount of storage required for a set of files by identifying duplicate "chunks" of data in a set of files and storing only one copy of each chunk [28, 31]. Subsequent requests to store a chunk that already exists in the chunk store are done by simply recording the *identity* of the chunk in the file's inode or block list; by not storing the chunk a second time, the system stores less data, thus reducing cost.

Different implementations of deduplication use different techniques to break files into chunks. Fixed-size chunking, such as that used in Venti [32] simply divides files at block boundaries. Variable-size chunking, used in systems such as LBFS [28] and Deep Store [45], computes a Rabin fingerprint [33] or similar function across a sliding window to place chunk boundaries, resulting in blocks that may have different lengths. This approach typically provides better deduplication, since it is more resistant to insertion or deletion of a few bytes in the middle of a file; however, it may negatively impact performance by requiring non-block aligned I/O requests.

While their approaches to identifying chunks differ, both fixed-size and variable-size chunking use cryptographically-secure content hashes such as MD5 or SHA1 [3] to identify chunks, thus allowing the system to quickly discover that newly-generated chunks already have stored instances. Even for a 128 bit MD5 hash, the chance of a single collision among 10^{15} chunks—10^{18} bytes, assuming an average chunk size of 1 KB—is about 10^{-9} [24]. By using a 160 bit hash such as SHA1, we can reduce the probability of a *single* collision in an exabyte-scale chunk store to about 7×10^{-18}. Collisions in SHA1 have been reported [43], and intentionally forging different but still equivalently meaningful independent chunks is a potential security breach, though it is not yet practical [15].

While these two issues might prevent a system implementer from using SHA1 [23], we chose SHA1 to name chunks in our experiments because these issues clearly do not impact the statistical results gathered from our experiments—one or two "false collisions" would not significantly alter our findings.

Deduplication and similar technologies have already been used to reduce bandwidth and storage demands for network file systems [28, 2, 18], reduce the storage demands created by VM checkpoints [30], store less data in backup environments [14, 46, 32], and reduce storage demands in archival storage [45, 44]. Using deduplication in an online system requires fast identification of duplicate chunks; techniques such as those developed by Bhagwat, *et al.* [9] and Zhu, *et al.* [46] can help alleviate this problem. Moreover, Nath, *et al.* found that deduplication was sufficiently fast for storing checkpoints of VM images [30] and the Difference Engine [21] used deduplication to share in-memory pages between different virtual machine instances. While these uses are not identical to read-write sharing of VM disk image chunks, the relatively low performance overhead for other uses of deduplication in VM images suggests that a file system for VM disk images should be sufficiently fast to use in a production environment.

Several projects have investigated the use of deduplication in virtualization environments. Nath, *et al.* used content-addressable storage to store multiple checkpoints from a hosting center running multiple virtual machines [30]. Their experiments covered 817 checkins across 23 users and 36 virtual machines. However, this study created "gold images" for each operating system (Windows XP and Linux) upon which each VM was based; while the gold images received security patches, users did not have a choice of which Linux distribution to run. Our research explores a much wider range of Linux configurations, exposing the factors that affect deduplication. The Difference Engine project [21] used deduplication to allow multiple virtual machines to share in-memory pages. This approach proved effective, though the level of deduplication was lower for memory page deduplication than for storage deduplication. Moreover, the VMs were limited in sample size; in contrast, the virtual disks we studied showed a large amount of variation. Liguori and Van Hensbergen explored the use of content addressable storage

(CAS) with virtual machines [4]. They examined overlap between pairs of VM disk images for both Linux and Windows XP, measuring the amount of deduplication possible between them. They then experimented with an implementation of CAS in Qemu, showing that Venti, a content-addressable store, performs worse than "vanilla" systems, though this may be due to Venti's inefficiency. Their deduplication results confirm some of our findings, but they do not exhaustively study different operating system characteristics and their impact on deduplication effectiveness. `DeDe` [10] exhibits a deduplication system in SAN, which is integrated into the VMware ESX Server [41] atop VMware VMFS. While files and index data are managed independently on each host on regular basis, the index data are synchronized on schedule. The overall evaluation shows high deduplication ratio for 113 Windows XP VMs. Similar to previous findings [30], these VMs are originated from a small set of standardized images.

2.3 Delta encoding

Delta encoding [1] is a widely-used technology for minimizing the data stored and transmitted in various cases. Unlike the normal applications that handle data at the granularity of complete objects, delta encoding processes data in the form of states and differences. It is mainly based on the following assumption: when modifying a data object, the size of the difference between the original state and the current state is less than the size of the data object itself. Therefore, when the new states of the object need to be stored or transmitted, it is beneficial to represent them with the original state and the corresponding differencing data rather than storing them in full. The new states can be restored by applying the differencing data to the original state.

Delta encoding is particularly useful for situations where data is being updated across a network with limited bandwidth [28]. Web sites and email servers are good examples to illustrate its usage. Web sites are regularly replicated in different web servers both for high performance and availability. These web servers may be physically located in different spots, even separated by continents. It is crucial to minimize the synchronization data transmitted from any source server to any target server for keeping every visitor updated. Email system users often replicate their email messages locally. Many email softwares store all the messages of each user in single file. When these files grow large, or the users connect to the email servers with a slow link, the size of transmitted data is desired to be minimized.

The most accessible implementation of delta encoding is probably the “diff” utility available in the Unix systems. It takes two file names as input and generates the differencing data as output. The size of the output should be mathematically minimal. Users could use the “patch” utility to make restorations. Alternatively, Myers [29] described a fast algorithm to find such differencing data, which was based on the concept of “edit script.” An edit script is an ordered set of insertion and deletion instructions that transforms sequence A to B. [26] defines VCDIFF as a generic format for the differencing data. Xdelta and open-vcdiff [38, 39] are two implementation libraries of the VCDIFF format.

Since the concept of delta encoding is common in conventional compression such as 7-zip and bzip2 [12, 13], people are motivated to adopt it in the deduplication area. Difference Engine [21] was implemented to identify and compress memory pages in virtual machines. The authors proposed a parameterized scheme to discover similar chunk candidates, and make patches for the best compressed chunk pairs. The delta encoding implementation they used is Xdelta. In this paper we use a modified version of the Difference Engine algorithm to find encoding candidates. We are also interested in comparing the effect of scheme parameters on the VM disk image case to the memory page sharing case. The system designed by Aronovich *et al.* [5] was built with a similarity based deduplication that also uses the signature mechanism to identify similar chunks. However, the signature generation algorithm they chose was different, and the authors provided mathematical discussions to support their choice. The two systems both have good results in saving storage space.

Chapter 3

Deduplication

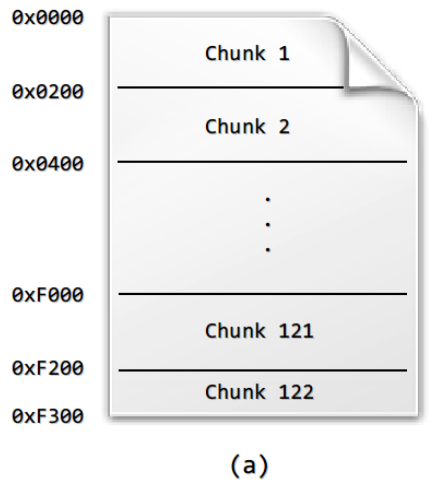
Since our experiments were focused on the amount of deduplication possible from sets of VM disk images, we first broke VM disk images into chunks, and then analyzed different sets of chunks to determine both the amount of deduplication possible and the *source* of chunk similarity. Section 3.1 discusses the techniques we used to generate chunks from VM disk images, and Section 3.2 discusses the approach we used to measure deduplication effectiveness.

We use the term *disk image* to denote the logical abstraction containing all of the data in a VM, while *image files* refers to the actual files that make up a disk image. A disk image is always associated with a single VM; a *monolithic* disk image consists of a single image file, and a *spanning* disk image has one or more image files, each limited to a particular size, typically 2 GB. When we refer to "disk images," we are referring to multiple VM disk image files that belong to multiple distinct VMs. Finally, we use the term *chunk store* to refer to the system that stores the chunks that make up one or more disk images.

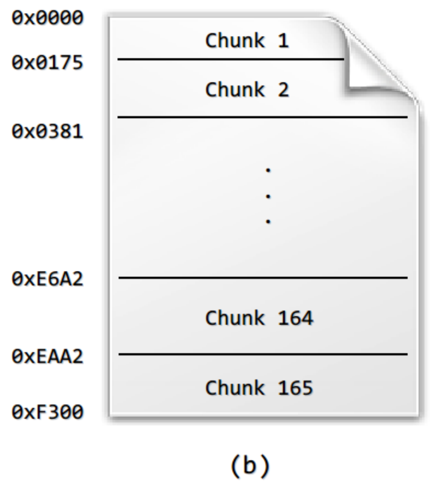
3.1 Chunking

In order to locate identical parts of disk images, we divide the image files into chunks to reduce their granularities. This is done by treating each image file as a byte stream and identifying boundaries using either a “constant” function (for fixed-size chunking) or a Rabin fingerprint (for variable-sized chunking). A chunk is simply the data between two boundaries; there are implicit boundaries at the start and end of each image file. Chunks are identified by their SHA1 hash, which is calculated by running SHA1 over the contents of the chunk. We assume that chunks with the same chunk ID are identical; we do not do a byte-by-byte comparison to ensure that the chunks are identical [22].

We implemented both fixed-size and variable-size chunking to test the efficiency of each approach in deduplicating disk images. Fixed-size chunking was done by reading an image file from its start and setting chunk boundaries every N bytes, where N is the chunk size. For variable-size chunking, we calculated a 64-bit Rabin fingerprint using the irreducible polynomial from Table 3.1 for a fixed-size sliding window, slid the window one byte at a time and updated the fingerprint until the modulo of the fingerprint and the divisor became equal to the residual; at this point a boundary was created at the start of the window, as shown in Figure 3.1. Of course, this approach resulted in chunks that may have greatly varying sizes, so we imposed minimum and maximum chunk sizes



(a) Fixed-size chunking with chunk size 512 B.



(b) Variable-size chunking with expected average chunk size 512 B.

Figure 3.1: Chunking a file with size 0xF300 bytes.

on the function to reduce the variability, as is usually done in real-world systems [45]. The specific parameters we used in variable-size chunking are shown in Table 3.1; These parameters were chosen to ensure that the chunking algorithm generates chunks with the desired average chunk size. We conducted experiments for average chunk sizes of 512, 1024, 2048, and 4096 bytes for both fixed-size and variable-size chunking.

We chunked each image file separately because fixed-size chunking exhibits the "avalanche effect:" although altering bytes in the file only changes the corresponding chunk IDs, inserting or removing bytes before the end of an image file changes *all* of the remaining chunk IDs, unless the length of insertion or deletion is a multiple of the chunk size for fixed-size chunking. Thus, if image file sizes are not multiples of the chunk size, the result of chunking across files is different than that of chunking each separately.

Name	Value
divisor	512, 1024, 2048, 4096 (bytes)
maximum chunk size	divisor $\times$ 2
minimum chunk size	divisor / 16
irreducible polynomial	0x91407E3C7A67DF6D
residual	0
sliding window size	minimum chunk size / 2

Table 3.1: Parameters for variable-size chunking.

Also, because both monolithic and spanning image files have a header specific to the VM instance, chunking sequentially across the spanning files does not restore the original guest file system because the last chunk of each file could be shorter than the specified chunk size.

Zero-filled chunks are common in VM disk images, and come from three sources. One source is VM-specific: disk images can contain zero blocks corresponding to space not yet used by the virtual machine. Another source is runs of zeroes in the file system, caused by space that has been zeroed by the operating system running in the VM. The third source is application-generated zero-filled blocks, as are sometimes generated by databases and other applications. The relative frequency of the three sources of zeroed blocks varies in different VMs. While the first source is VM-generated, different types of disk images (flat versus sparse) can have different numbers of zero blocks in them. Decisions such as the maximum disk image size can influence this number as well. The other two sources of zero blocks are due to the guest operating system and applications, which are less affected by the choice of virtual disk size. In fixed-size chunking, all zero-filled chunks are identical. In the variable-size chunking experiments, runs of zeros do not generate boundaries, and thus result in chunks of the maximum chunk size. Since all zero-filled chunks are the same (maximal) size (except perhaps for a run at the end of an image file), they are all identical to one another.

To further reduce space, we compress each chunk using `zip` after hashing it to generate the chunk ID. Since the `zip` compression is deterministic, additional space saving can be achieved with no loss of fidelity. It is a trade off between time, space saving and the size of chunks.

3.2 Deduplication

The deduplication process is simple: for each chunk being stored, we attempt to locate an existing instance in the chunk store. If none is found, the new chunk is added to the chunk store; otherwise, the new chunk is a shared chunk. As described in Section 3.1, nearly all zero chunks are identical, except for a non-maximal length zero chunk at the end of an image file. Since even large spanning disk images have relatively few files, most of which end with non-zero chunks, an optimization that recognizes such non-maximal length zero chunks would provide little benefit.

The chunk ID, which is generated from the SHA1 hash of the chunk's content, is the only value used to look up existing chunks. Even for an exabyte-scale chunk

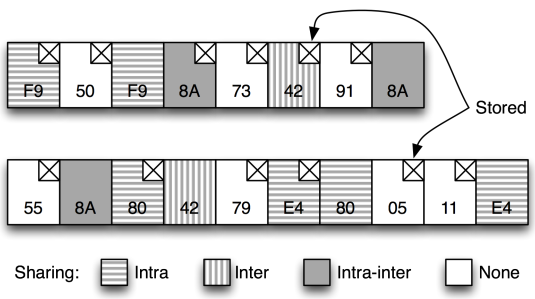


Figure 3.2: Share categories of chunks. Chunks seen for the first time must be stored; subsequent occurrences need not be stored.

store, collisions would be highly unlikely; for the multi-terabyte chunk store in our experiments, the chances of collision are even smaller. The maximum number of possible distinct chunks in an exabyte file is 2^{54}, with fixed-size chunking and 8 byte chunk size employed; this is far more less than the $1/2^{80}$ collision likelihood of SHA-1.

We calculate the deduplication ratio for a given chunk store and chunking method by:

$$1 - \frac{Stored\ bytes\ of\ all\ disk\ images}{Original\ bytes\ of\ all\ disk\ images}$$

The deduplication ratio is a fraction in $[0, 1)$, since there is at least one stored chunk in a chunk store, and the worst possible case is that there are no duplicate chunks. We exclude per-chunk overhead in our studies, which is 20 bytes for SHA-1 hash and varied size of implementation-specific bookkeeping information.

We classify each occurrence of a chunk into one of four categories, shown in Figure 3.2. When a chunk appears exactly once in the entire set of VM disk images, it is called an *unshared chunk*, labeled "none" in Figure 3.2. Chunks that appear in more than one disk image are termed *inter-image shared chunks*, and chunks that appear multiple times, within a single disk image but not elsewhere, are called *intra-image shared chunks*. Chunks that appear in multiple disk images and appear more than once in at least one of those images are *inter-intra-image shared chunks*. Zero-filled chunks are tracked separately; however, they are typically inter-intra-image shared chunks in sets with more than one disk image because zero-filled chunks appear in every disk image that we examined.

As Figure 3.2 shows, all chunks must be stored the first time they are seen. Subsequent occurrences of each chunk are not stored, reducing the total storage space required to store the set of disk images. All stored chunks are grouped together for our experiments, while non-stored chunks are classified by the disk images in which other occurrences of the chunks are found. Thus, a chunk c that occurs one time in disk image A and then two times in disk image B would result in one *stored* chunk and two *inter-intra shared* chunks because there are occurrences of chunk c in two *separate* disk images, and multiple occurrences of chunk c in at least one disk image. The total size of a chunk store before deduplication is thus the sum of the sizes of the stored chunks and three shared categories. This notation differs from the metrics used by another study [4], which only concentrates on "duplicate" chunks. In their study, two identical disk images would be 100% similar, and half (only inter-share) or less (inter- and intra-share) chunks would typically need storage.

Changing the processing order for a set of disk images can produce different intermediate results for the number and type of shared chunks and the deduplication ration, but the final result will always be the same for a given set of disk images. For example, processing disk images in the order $\langle A1, A2, B\rangle$ would result in a high level of inter-image sharing after the second disk image was added, assuming that images $A1$ and $A2$ are very similar and both are dissimilar to image B. However, processing the files in the order $\langle A1, B, A2\rangle$ would result in a much lower deduplication ratio after the second disk image was added, but the final result would be the same as for the first processing order.

3.3 Experiments

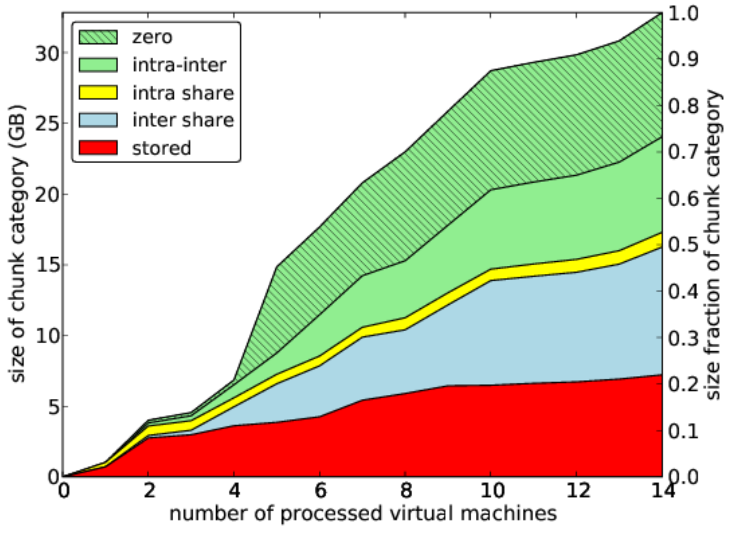
To determine which factors affect deduplication ratios for sets of disk images, we obtain the pre-made disk images listed in Table 3.2 from Internet web sites as data source, including VMware's Virtual Appliance Marketplace [40], Thoughtpolice [37], and bagvapp's Virtual Appliances [6] site. Most of the VMs were created in VMware's format, and were compatible with VMware Server 2.0. The VirtualBox VMs were compatible with VirtualBox 2.0 and onward.

For the figures in this section, *stored* chunks are counted the first time they are seen during the deduplication process, each of which must be stored. Each of the other chunk categories is reduced in size by its remaining instances. Zero chunks are isolated as a separate chunk class, not part of the inter-intra shared chunk.

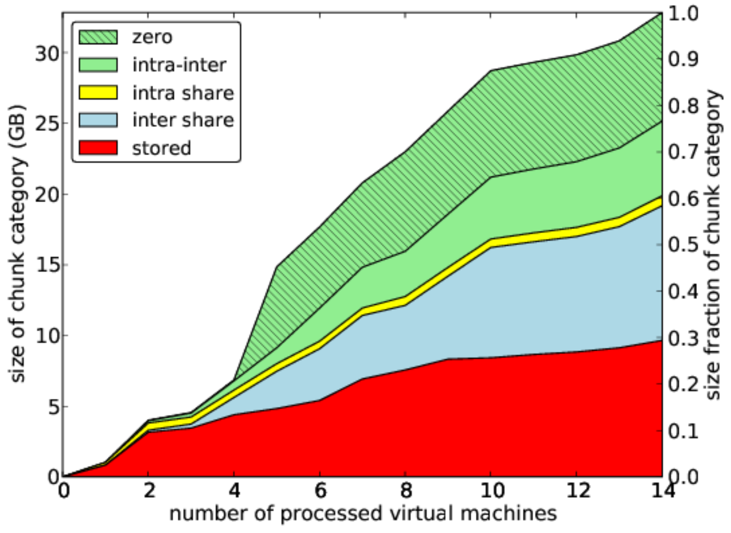
Index	Name and version	Kernel	File system	Desktop	Image size	Disk type
1	Arch Linux 2008.06	Linux 2.6.25-ARCH	ext3	GNOME	3.5G	XS
2	CentOS 5.0	Linux 2.6.18-8.el5	ext3	None	1.2G	XS
3	CentOS 5.2	Linux 2.6.18-92.1.10.el5	ext3	GNOME	3.3G	MS
4	DAMP	Dragonfly 1.6.2-RELEASE	ufs	None	1.1G	MF
5	Darwin 8.0.1	Darwin 8.0.1	HFS+	None	1.5G	MF
6	Debian 4.0.r4	Linux 2.6.18-6-486	ext3	None	817M	XS
7	Debian 4.0	Linux 2.6.18-6-686	ext3	GNOME	2.5G	MS
8	DesktopBSD 1.6	FreeBSD 6.3-RC2	ufs	KDE	8.1G	XF
9	Fedora 7	Linux 2.6.21-1.3194.fc7	ext3	GNOME	2.9G	XS
10	Fedora 8	Linux 2.6.23.1-42.fc8	ext3	GNOME	3.4G	XS
11	Fedora 9 en-US	Linux 2.6.25-14.fc9.i686	ext3	GNOME	3.4G	XS
12	Fedora 9 fr	Linux 2.6.25-14.fc9.i686	ext3	GNOME	3.6G	XS
13	FreeBSD 7.0	FreeBSD 7.0-RELEASE	ufs	None	1.2G	XS
14	Gentoo 2008.0	Linux 2.6.24-gentoo-r8	ext3	Xfce	5.5G	XS
15	Gentoo 2008.0 with LAMP	Linux 2.6.25-gentoo-r7	ext3	None	8.1G	XF
16	Knoppix 5.3.1	Linux 2.6.24.4	ext3	KDE	13G	XS
17	Kubuntu 8.04.1	Linux 2.6.24-19-generic	ext3	KDE	2.6G	MS
18	Mandriva 2009.0	Linux 2.6.26.2-desktop-2mnb	ext3	GNOME	3.3G	XS
19	NAMP	NetBSD 3.1 (GENERIC)	ffs	None	1.1G	MF
20	OAMP	OpenBSD 4.0	ffs	None	804M	MS
21	OpenBSD 4.3	OpenBSD 4.3	ffs	None	558M	MS
22	OpenSolaris 2008.5	SunOS 5.11	ZFS	GNOME	3.8G	XS
23	openSUSE 11.0	Linux 2.6.25-1.1-pae	ext3	KDE	8.1G	MF
24	PC-BSD 1.5	FreeBSD 6.3-RELEASE-p1	ufs	KDE	2.2G	MS
25	Slackware 12.1	Linux 2.6.24.5-smp	ext3	KDE	3.5G	MS
26	Ubuntu 8.04 en-US	Linux 2.6.24-19-generic	ext3	GNOME	3.5G	MS
27	Ubuntu 8.04 fr	Linux 2.6.24-19-generic	ext3	GNOME	2.5G	XS
28	Ubuntu 8.04 JeOS	Linux 2.6.24-16-virtual	ext3	None	293M	MS

Index	Name and version	Kernel	File system	Desktop	Image size	Disk type
29	Ubuntu 6.10 Server	Linux 2.6.17-10-server	ext3	None	520M	XS
30	Ubuntu 7.04 Server	Linux 2.6.20-15-server	ext3	None	557M	XS
31	Ubuntu 7.10 Server	Linux 2.6.22-14-server	ext3	None	543M	XS
32	Ubuntu 8.04 Server	Linux 2.6.24-16-server	ext3	None	547M	XS
33	Ubuntu 8.04 LTS (1)	Linux 2.6.24-16-server	ext3	GNOME	3.0G	XS
34	Ubuntu 8.04 LTS (2)	Linux 2.6.24-16-server	ext3	GNOME	2.9G	MS
35	Ubuntu 8.04 LTS (3)	Linux 2.6.24-16-server	ext3	GNOME	2.2G	XS
36	Ubuntu 8.04 LTS (4)	Linux 2.6.24-16-server	ext3	GNOME	2.9G	MS
37	Ubuntu 8.04 LTS (5)	Linux 2.6.24-16-server	ext3	GNOME	2.9G	MS
38	Ubuntu 8.04 LTS (6)	Linux 2.6.24-16-server	ext3	None	547M	XS
39	Ubuntu 8.04 LTS (7)	Linux 2.6.24-16-server	ext3	GNOME	2.1G	XS
40	Ubuntu 8.04 LTS (8)	Linux 2.6.24-16-server	ext3	None	1.1G	MS
41	Ubuntu 8.04 LTS (9)	Linux 2.6.24-16-server	ext3	None	559G	MS
42	Ubuntu 8.04 LTS (10)	Linux 2.6.24-16-server	ext3	GNOME	3.2G	MS
43	Ubuntu 8.04 LTS (11)	Linux 2.6.24-16-server	ext3	None	604M	MS
44	Ubuntu 8.04.1 LTS (12)	Linux 2.6.24-16-server	ext3	GNOME	2.4G	MS
45	Ubuntu 8.04.1 LTS (13)	Linux 2.6.24-16-server	ext3	GNOME	8.1G	XF
46	Ubuntu 8.04.1 LTS (14)	Linux 2.6.24-16-server	ext3	None	1011M	XS
47	Xubuntu 8.04	Linux 2.6.24-16-generic	ext3	Xfce	2.3G	XS
48	Zenwalk 5.2b	Linux 2.6.25.4	ext3	Xfce	2.5G	XS
49	Ubuntu 8.04 Server on VMware	Linux 2.6.24-16-server	ext3	None	1011M	MS
50	Ubuntu 8.04 Server on VirtualBox	Linux 2.6.24-16-server	ext3	None	969M	MS
51	Ubuntu 8.04 Server on VMware	Linux 2.6.24-16-server	ext3	None	4.1G	XF
52	Ubuntu 8.04 Server on VirtualBox	Linux 2.6.24-16-server	ext3	None	4.1G	MF

Table 3.2: Virtual machine disk images used in the study. Under disk type, *M* is a monolithic disk image file, *X* is a disk image file split into 2 GB chunks, *S* is a sparse image file, and *F* is a flat image file. For example, *MS* would correspond to a single monolithic disk image formatted as a sparse image.



(a) Average chunk size = 1 KB.



(b) Average chunk size = 4 KB.

Figure 3.3: Growth of data in different categories for 14 different Ubuntu 8.04 LTS instances. Stored data grows very slowly after the second VM is integrated. In these figures, stored data includes the first instance of each chunk, regardless of how it is shared. The sharp increase in size at the 5th VM is due to the use of a flat disk image, in which there are a large number of zero-filled (empty) sectors.

3.3.1 Overall Impacts

Before going into detail on how much specific factors impact deduplication, we evaluated deduplication for closely related disk images—a set of images all based on Ubuntu 8.04 LTS—whose result was then compared against a set of disk images from widely divergent installations, including BSD, Linux, and OpenSolaris VMs. The results are shown in Figures 3.3 and 3.4. As these figures show, operating systems with similar kernel versions and packaging systems deduplicate extremely well, with the system able to reduce storage for the 14 disk images by over 78% and 71% for 1 KB and 4 KB chunk size, respectively. Of the 22% of the chunks that are stored, 12% are chunks that occur exactly once and 10% are chunks that occur more than once, with only a single instance stored.

Given this trend, it is likely that additional Ubuntu 8.04 disk images would add little additional operating system data. It is important to note that these images were pre-compiled and gathered from various sources; they were not created by a single coordinated entity, as was done by Nath, *et al.* [30], suggesting that hosting centers can allow users to install their own VMs and still gain significant savings by using deduplication. Since it takes more time and storage overhead to generate smaller chunks, 4 KB chunk size might be a good idea if space allows.

On the other hand, a set of disk images consisting of 13 operating system images including various versions of BSD, OpenSolaris, and Linux did not fare as well, as Figure 3.4 shows. In such a case, unique data is the largest category, and deduplication

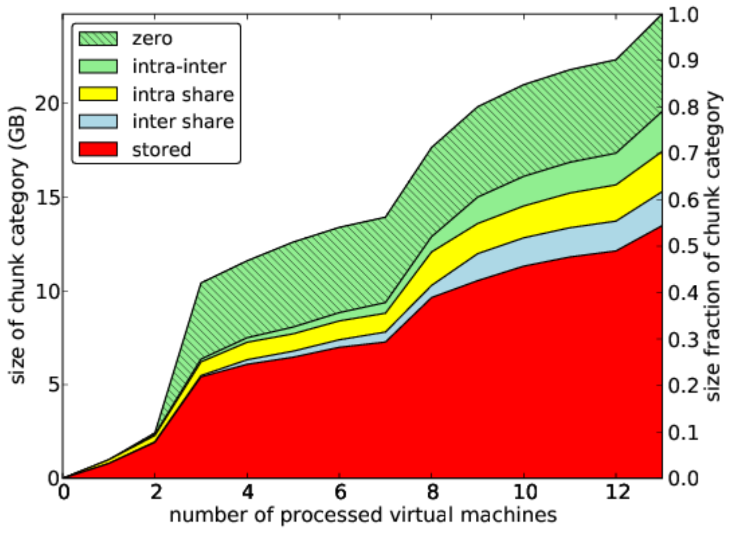


Figure 3.4: Growth of category data for 13 Unix and Linux virtual machines, using variable-size chunking, with an average chunk size of 1 KB. Unique data grows significantly as each disk image is added. As in Figure 3.3, the sharp increase in size at the 3rd VM is due to a large number of empty, zero-filled sectors in a flat disk image.

saves less than half of the total space, with zero-filled blocks second. This is close to the worst-case scenario for deduplication, since the disk images differed in every possible way, including operating system and binary format. That this approach was able to achieve a space savings of close to 50% is encouraging, suggesting that adding further VMs will result in more space savings, though not as much as for the case in which all VMs are highly similar.

In our next experiment, we compared deduplication levels for chunk stores consisting only of VMs with a single operating system—BSD or Linux—to that of two more heterogeneous chunk stores: one with BSD, OpenSolaris, and Darwin (labeled "Unix"), and another with all types of operating systems (labeled "All"). The *All* chunk

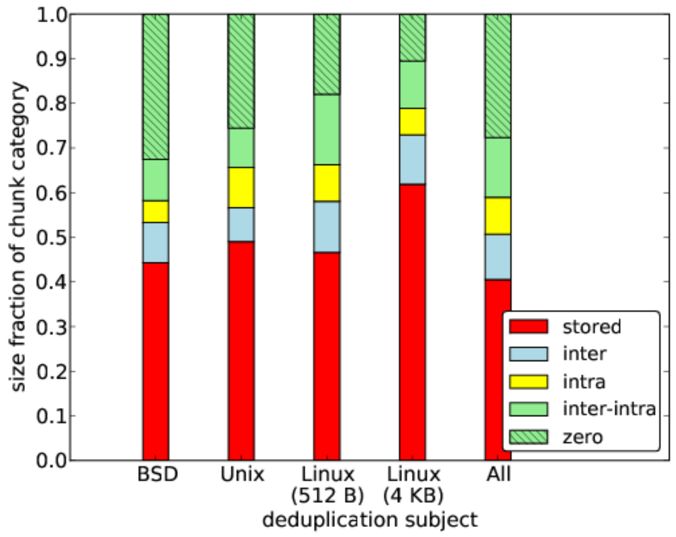


Figure 3.5: Effects of varying operating system type on deduplication. All experiments used 512 B variable-size chunks, except for the Linux experiment that uses 4 KB variable-size chunks. There is more intra-inter sharing in Linux than in BSD, indicating that the former is more homogeneous.

store is a super set of the other three chunk stores, and contains more flat disk images. As Figure 3.5 shows, the *All* chunk store achieved the best deduplication ratio, indicating that, even for operating systems of different lineage, there is redundancy available to be removed by deduplication. This seemingly contradictory high deduplication level comes from two sources. The first is zero chunks, as the *All* chunk store includes more flat disk images than any other chunk stores. The second is additional inter chunks, as they appeared only once in *BSD* or *Linux* chunk stores and were considered as stored chunks. Nevertheless, the higher levels of sharing for non-zero chunks in the *Linux* chunk store indicates that the Linux disk images were more homogeneous than the BSD images and *All* images.

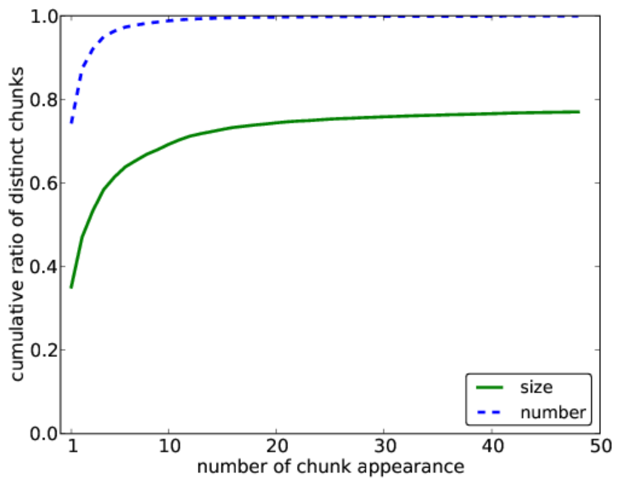


Figure 3.6: Cumulative distribution of chunks by count and total size. The upper line shows the cumulative number of chunks with total count of n or less, and the lower line shows the total space that would be consumed by chunks with count n or less. The data is from the *Linux* chunk store with chunk size 512 B. As the graph shows, most chunks occur fewer than 14 times, and few non-zero chunks appear more than 50 times in the set of disk images.

Another notable feature in Figure 3.5 is that the fraction of zero chunks decreases markedly from the 512 B Linux case to the 4 KB Linux case. While the overall number of chunks decreases by a factor of 8 (4096/512), the fraction of chunks that contain all zeros shrinks, with a corresponding increase in unique chunks, indicating that most of the zero chunks are 2 KB or less in length. The figure also shows that, while the relative fractions of zero chunks and unique chunks changes from 512 B chunks to 4 KB chunks, the amount of sharing changes little, indicating that chunk size may have relatively little impact on the relative frequency of shared chunks.

Figure 3.6 shows the cumulative distribution of chunks by both count and total size in the *Linux* chunk store. Chunks that appear only once are *unique* chunks and

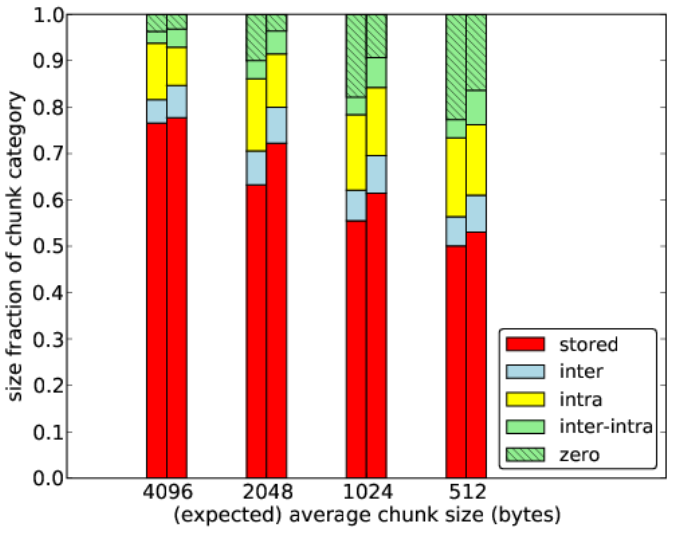


Figure 3.7: Effects of chunk size and fixed versus variable-size chunking on deduplication for a set of disk images including Ubuntu Server 6.10, 7.04, 7.10 and 8.04. In each group of two bars, the left bar measures fixed-size chunking and the right bar indicates variable-size chunking. The graph shows that smaller chunk sizes result in more effective deduplication.

must be stored; for other chunks, the chunk store need only contain one copy of the chunk. As the figure shows, over 70% of chunks occur exactly once, and these chunks make up about 35% of the undeduplicated storage. The rest 20% storage is occupied by zero-filled chunks, as shown in Figure 3.5; they are particularly common in flat disk images.

3.3.2 Impact of Specific Factors

The next set of experiments focus on many factors that might affect deduplication, identifying those that are most critical and quantifying their effects on the deduplication ratio.

We first examined the effect of chunk size and boundary creation technique on deduplication ratio. Figure 3.7 shows the effect of deduplicating a set of disk images for different versions of Ubuntu Server using fixed and variable-size chunking for different average chunk sizes. As expected, smaller chunk sizes result in better deduplication ratios; this is done by converting unique chunks into shared chunks and zero chunks. Interestingly, the number of zero chunks grows significantly as chunk size decreases, indicating that zero chunks are more likely to be generated by the operating system or applications than by the VM software writing out the virtual disk, since zeros in the virtual disk would likely be 4 KB or larger. It is also important to note that fixed-size chunking is even more effective than variable-size chunking in this experiment, suggesting that fixed-size chunking may be appropriate for VM disk images.

We next examined the effects of different releases on deduplication effectiveness. Figure 3.8 shows the deduplication ratios for consecutive and non-consecutive releases of Ubuntu and Fedora. *Ubuntu7-8* is the product of deduplicating Ubuntu 7.10 against Ubuntu 8.04, and *Ubuntu6-8* is built with Ubuntu 6.10 and Ubuntu 8.04. *Fedora8-9* is the result generated from Fedora 8 and Fedora 9, while *Fedora7-9* is the story of Fedora 7 and Fedora 9. Deduplication patterns between consecutive and non-consecutive releases of a single distribution appear similar, and deduplication is only slightly less effective when skipping a release. This experiment shows that when a mature operating system such as Ubuntu updates its major version, most of the data, *e. g.*, base system or software architecture, remains unchanged. Another interesting point from Figure 3.8 is that variable-size chunking does much better on Fedora than

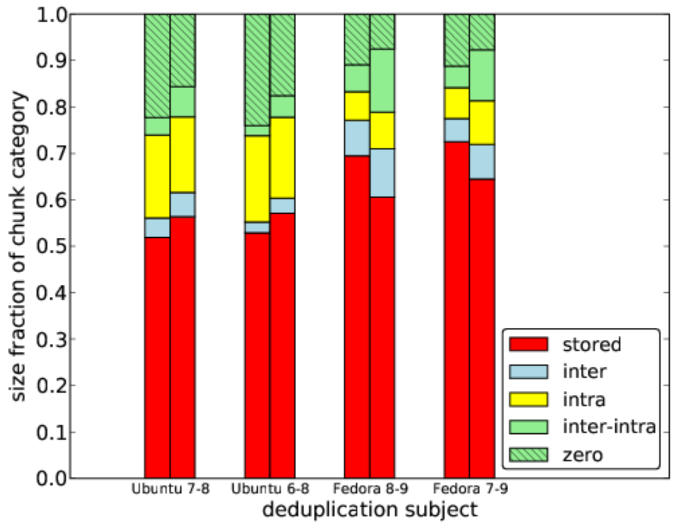


Figure 3.8: Deduplication on different releases of a single Linux distribution. The left bar in each pair is fixed-size chunking and right bar is variable-size chunking; chunk size is 512 B. Consecutive releases have only slightly lower deduplication ratio than non-consecutive releases.

does fixed-size chunking, in large part because it is able to convert unique blocks into inter-intra shared blocks. We do not know why this is; however, it is one of the only cases in which variable-size chunking significantly outperforms fixed-size chunking in our experiments, confirming that deduplication for VMs should use fixed-size chunking rather than adding the complexity of variable-size chunking.

Figure 3.9 shows the impact of OS locale upon deduplication. We deduplicated English and French versions of Ubuntu Server 8.04 and Fedora 9 separately; the only differences between the two versions were a few files relating to software interfaces and keyboard layouts. As the figure shows, deduplicating the French version of each OS against its English version adds few stored chunks, indicating that changing the

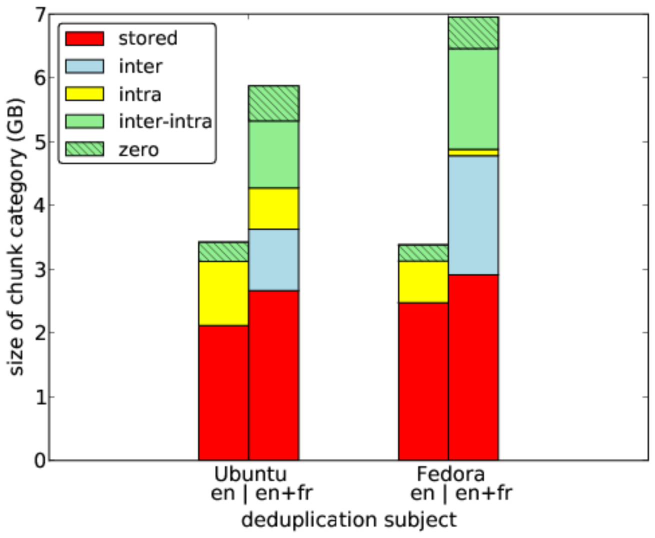


Figure 3.9: Locale versions (variable-size chunking, chunk size 512 B). The left bar in each pair only measures the English version and the right bar measures both English and French versions. Storing different locale versions of same distribution produces high deduplication.

localization of an operating system introduces very few unique blocks.

We next evaluated the effect of deduplicating Linux versions that derive from a common root. Ubuntu and Knoppix are both based on Debian, and Fedora, CentOS, and Mandriva descend from Red Hat Linux. The result, shown in Figure 3.10, is surprising: despite their common lineage, Linux versions derived from a single root do not deduplicate well against each other. While mature releases do not change much, as Figure 3.8 showed, there are significant changes when a new Linux distribution "forks off." Thus, it is necessary to consider these distributions as distinct ones when considering deduplication effectiveness.

We next evaluated the effects of varying the operating system while keeping

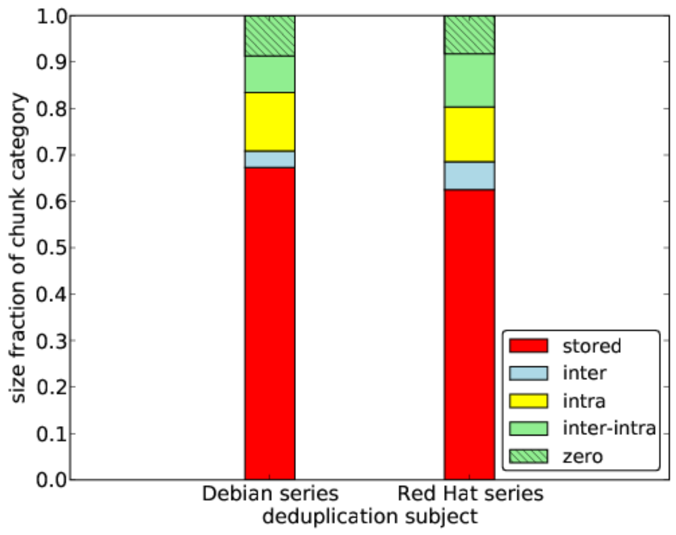


Figure 3.10: Distribution lineage. Debian series comprises Debian, Ubuntu and Knoppix. Red Hat series comprises Fedora, CentOS and Mandriva. Variable-size chunking, chunk size 512 B. Despite the common lineages, there is a relatively low level of deduplication.

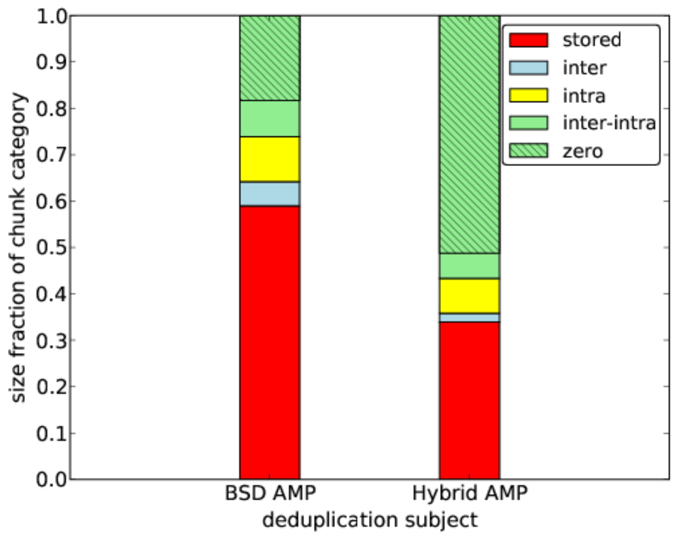
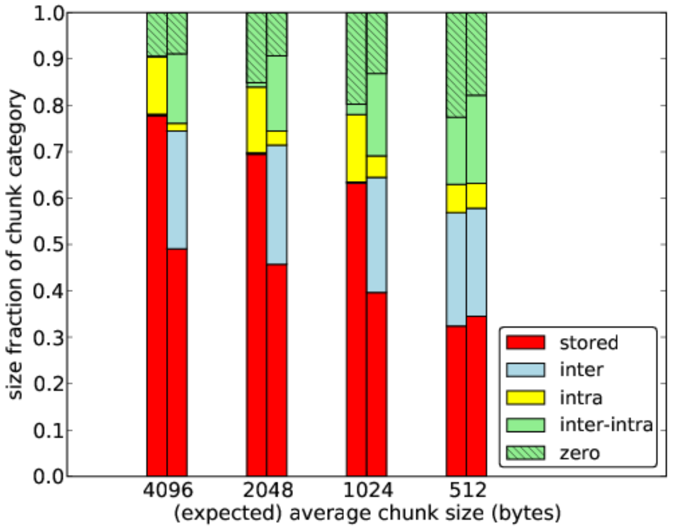


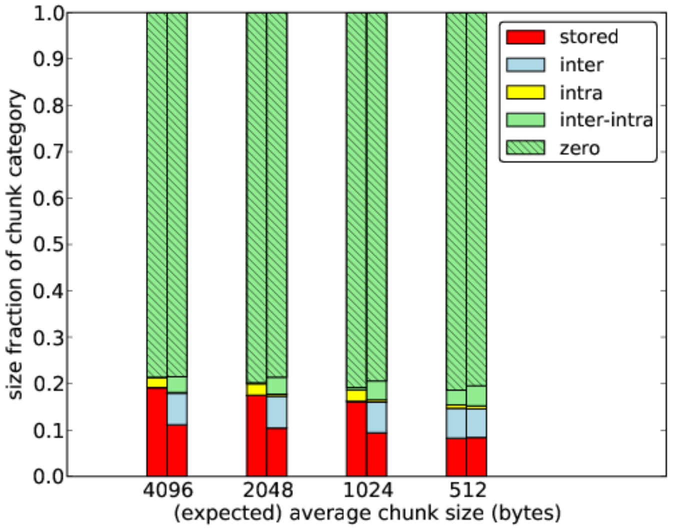
Figure 3.11: Deduplication of Web appliance VMs (variable-size chunking, chunk size 512 B). *AMP* stands for Apache, MySQL and PHP. Since the deduplication ratios for both cases are low when zero chunks are excluded, we can conclude that diverse operating systems with similar goals do not have many identical chunks of code.

the purpose of the distribution—in this case, serving Web pages—constant. We built a chunk store from disk images of DragonflyBSD, NetBSD and OpenBSD with Apache, MySQL, PHP and all dependent packages, as described in Section 3.3.3. We then added similar configurations of CentOS, Gentoo and Ubuntu into the chunk store; the results are shown in Figure 3.11. Excluding the large number of zero chunks, the deduplication ratios are not high, showing that the common purpose of the systems does not improve the deduplication ratio. While it may appear that diversifying the chunk store with Linux distributions helps, this is an illusion caused by the large number of zero chunks; the levels of sharing remain low.

The choice of virtual machine monitor (VMM) does not significantly change the virtual disk image, as Figure 3.12 shows. The same virtual machine, Ubuntu Server 8.04.1, on VirtualBox 2.0 and VMware 2.0 deduplicates extremely well, saving about half of the space, as long as variable-size chunking is used. The large variation in deduplication effectiveness for fixed-size chunking is due to different offsets in the disk image for the actual disk blocks (the first part of the file is occupied by the VMM-specific header). Since the offset of the actual virtual disk data in the two disk images files have the same value modulo 512 but not modulo 1024, fixed-size chunking is only effective for 512 B chunks. Figure 3.12(b) shows that, not surprisingly, pre-allocated disk images contain a lot of zeros. Since most modern file systems support *sparse files*, the VMMs which only support flat disk images may utilize the functionality to simulate sparse disk images, thus eliminate space requirement dramatically.



(a) Ubuntu Server 8.04.1 on VMware and VirtualBox (sparse disk image, 4 GB maximum)



(b) Ubuntu Server 8.04.1 on VMware and VirtualBox (flat disk image, pre-allocate 4 GB space)

Figure 3.12: Virtual machines created by different VMMs deduplicate well. The bar on the left measures fixed-size chunking, and the right-hand bar measures variable-size chunking.

Name	Image	Packages		
	size	Size	# Before	# After
Ubuntu, package set 1, asc, instance1	879	332	255	391
Ubuntu, package set 1, asc instance 2	883	336	255	391
Ubuntu, package set 1, desc	885	338	255	392
Ubuntu, package set 2, asc	873	326	255	397
Ubuntu, package set 2, desc	867	320	255	402
Ubuntu, remove no dependency	885	-	391	377
Ubuntu, remove all dependencies	881	-	391	256
CentOS, package set 1, asc	1435	387	359	443
Fedora, package set 1, asc	1307	405	194	306

Table 3.3: Package installation and removal sets. "asc" and "desc" refer to the order in which packages are installed. Image size is post-installation / removal. All sizes are in megabytes; the "before" and "after" columns reflect the number of installed packages before and after the installation / removal action.

3.3.3 Impact of Package Management

Users of modern Unix-like operating systems can modify their functionality by installing and removing software packages as well as by storing user-related data. In this section, we experimented with the impact on deduplication effectiveness of adding and removing packages. The packages used in the experiments are described in Table 3.4; all experiments were conducted on a fresh Ubuntu Server 8.04 and CentOS 5.2 installation, which use `aptitude` and `yum`, respectively, for package management. Table 3.3 lists the sizes of the VMs and the packages.

Figure 3.13 shows the deduplication effectiveness for different sets of disk im-

(a) Package set 1. Some packages only have a `deb` version or an `rpm` version.

Index	Package name		Software	Application
	Ubuntu	CentOS	detail	
1	apache2 apache2-doc	httpd httpd-manual	Apache 2.2	web server
2	php5 libapache2-mod-php5	php	PHP 5.2.4	language
3	mysql-server mysql-doc-5.0	mysql-server	MySQL Server 5.0	database
4	exim4	exim	Exim MTA 4.69	email service
5	mediawiki php5-gd	mediawiki	MediaWiki 1.11.2	wiki
6	vsftpd	vsftpd	Very Secure FTP 2.0.6	FTP server
7	subversion libapache2-svn	subversion	Subversion 1.4.6	version control system
8	bacula	bacula-client bacula-console bacula-director bacula-storage	Bacula 2.2.8	backup
9	rpm	-	RPM 4.4.2.1	package manager

(b) Package set 2

Index	Package name	Software detail	Application area
1	lighttpd lighttpd-doc	lighttpd 1.4.19	Web Server
2	rails	Ruby on Rails 2.0.2	Language
3	postgresql postgresql-contrib postgresql-doc	PostgreSQL 8.3.1	database
4	postfix	Postfix 2.5.1	Email service
5	python-moinmoin	Moin Moin 1.5.8	Wiki Application
6	cupsys	CUPS 1.3.7	Print Server
7	cvs xinetd	CVS 1.12.13	Version Control System
8	backuppc	BackupPC 3.0.0	Backup
9	yum	Yum 2.4.0	Package Management

Table 3.4: Two package sets.

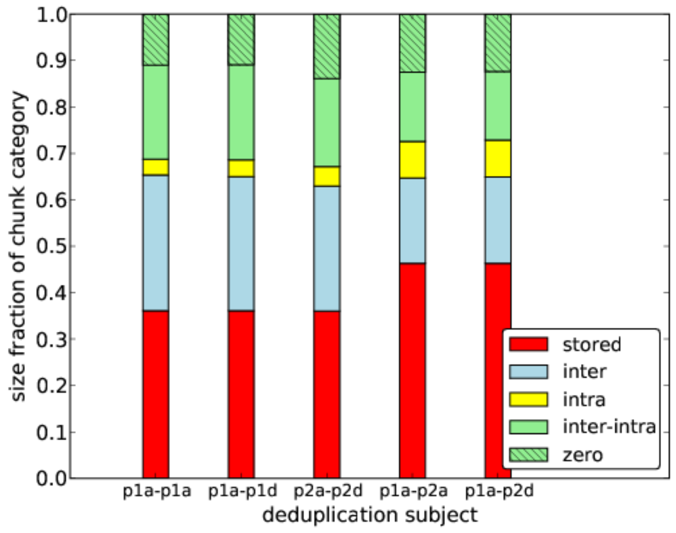


Figure 3.13: Differing package installation orders. *p1a* denotes package set 1, ascending order, and others follow the same naming rule. Variable-size chunking, chunk size 512 B. Different package installation orders generate nearly identical disk images.

ages with differing installation orders. Deduplication tests on these images show two important facts. First, installation order is relatively unimportant for deduplication; all cases have very high deduplication ratios, indicating that each disk image in a pair is nearly identical to the other. Second, increasing the number of packages installed reduces the level of deduplication but, again, installation order is relatively unimportant. This is an important finding for hosting centers; it shows that hosting centers can allow users to install their own packages without fear that it will negatively impact the effectiveness of deduplication. Again, this shows that it is not necessary to base disk images on a "gold" image; it suffices to use deduplication to detect identical disk blocks in the images.

When we calculated the deduplication ratio for the second case, we found that

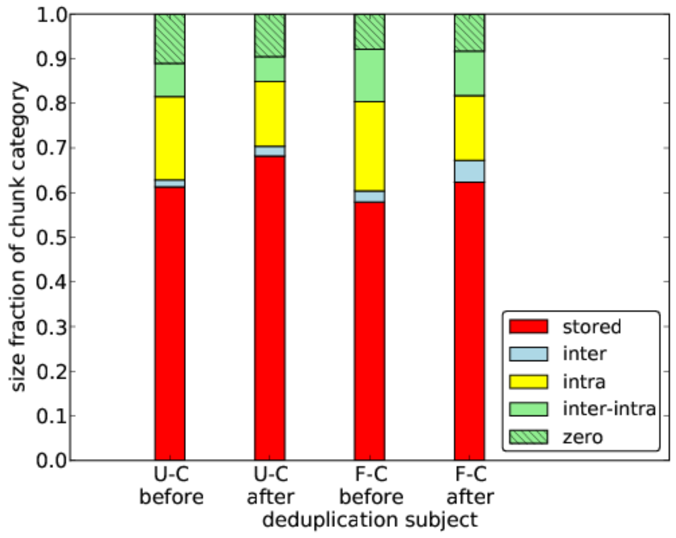


Figure 3.14: Package systems comparison on Ubuntu Server 8.04 (U), CentOS 5.2, no desktop (C) and Fedora 9, no desktop (F). Both install package set 1 in ascending order. Variable-size chunking, chunk size 512 B. Package system's contribution to deduplication is outweighed by OS heterogeneity.

the amount of newly introduced unique data is far less than the absolute size of the package sets. Applications in the same interest areas usually share common dependencies. For example, both `apache2` and `lighttpd` depend on `mime-support`, a commonly used MIME [19] support library for web server and email service. Although there is no common package in the two sets, the common dependent packages in each area outnumber the listed packages, allowing the potential of high deduplication; moreover, from Table 3.3 we see the total package numbers after installation are approximately equal for both package sets, implying that the dependency package numbers of are also approximately equal.

We also experimented with installing the same packages in the same order on different operating systems. The Debian distribution uses `deb` packages and manages them using `aptitude`. Red Hat distributions use the RPM package system, which uses `yum` to install and update packages. Some packages are system related, either with different binary or different library and dependencies. Figure 3.14 shows the result of installing package set 1 on Ubuntu, CentOS and Fedora, revealing two points. First, by installing the same set of packages, the deduplication ratio for different package systems drops. This is not surprising—the binary data for exactly the same software packages in `deb` and `rpm` format are different. Second, the deduplication ratio for the *same* package system also drops. This indicates software packages are primarily OS dependent, not package system dependent. The conclusion is that packaging system has little effect on deduplication ratio, particularly when compared to the diversity of the underlying operating systems.

The final factor we examined was the effect of removing packages. We compared the resulting disk images with the original and dirty disk images from earlier experiments. "Removing" was done by executing the `apt-get purge` command; as a result, the package name is removed from the list generated by `dpkg --get-selections`. We can see from Figure 3.15 that, as expected, removed packages are not actually wiped off the disk; rather, the files are marked as deleted. This is evident from the high deduplication ratio between the installed and removed images and lower ratio for the other two pairs. We can also see that the removal order is relatively unimportant as well. In order to preserve a high deduplication ratio, there should be as little difference in package installation as possible because it is only package installation, not removal, that

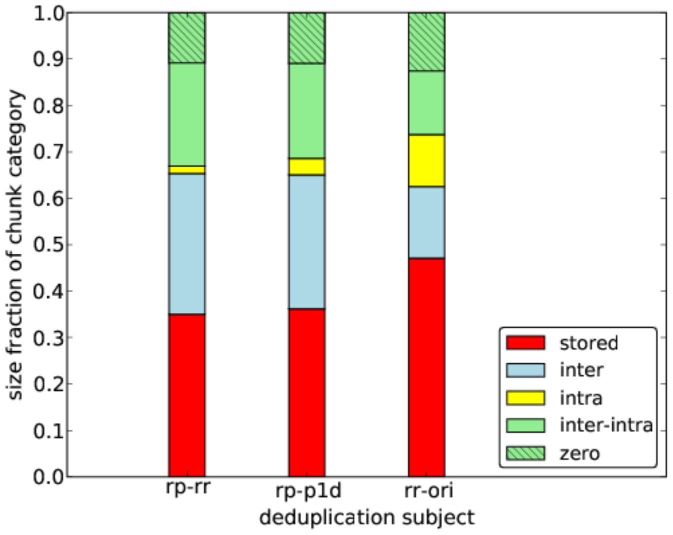


Figure 3.15: Package removal orders. Variable-size chunking, chunk size 512 B. *rp* denotes removing only target packages. *rr* denotes removing target packages and all dependencies (revert to original). *ori* denotes the original disk image before package installation. Data of the removed packages are not erased from disk image, which still resemble the image with the installed packages.

Method	Size
tar+gzip, level 9	21152781050 (19.7 GB)
7-zip, level 9	15008385189 (14.0 GB)
var, chunk size 512 B, level 9	29664832343 (27.6 GB)
var, chunk size 4 KB, level 9	23994815029 (22.3 GB)

Table 3.5: Compression parameters.

alters the disk image significantly. It is a good idea to leave all dependencies (orphan or not) installed rather than removed because the file system will allocate different regions for these dependencies when reinstalling them. Moreover, on a deduplicated system, unused packages require no additional space, so there is no reason to remove them.

3.3.4 Chunk Compression

In this section, we discuss the impact of compressing chunks on space reduction. We employed the `zlib` compression method, using the DEFLATE algorithm [17]), which is essentially the same as that used in Linux's `zip` and `gzip` code. As a good comparison group, we used `7-zip` (LZMA algorithm) to archive and compress the original disk images, because various benchmarks [11, 25] have shown that `7-zip` is capable of higher compression than `zip`. We used the *Linux* chunk store from Section 3.3.1 for our compression experiments. The final sizes of the chunk store are shown in Table 3.5.

Figure 3.16 tells us three things. First, chunk-wise compression can reduce the

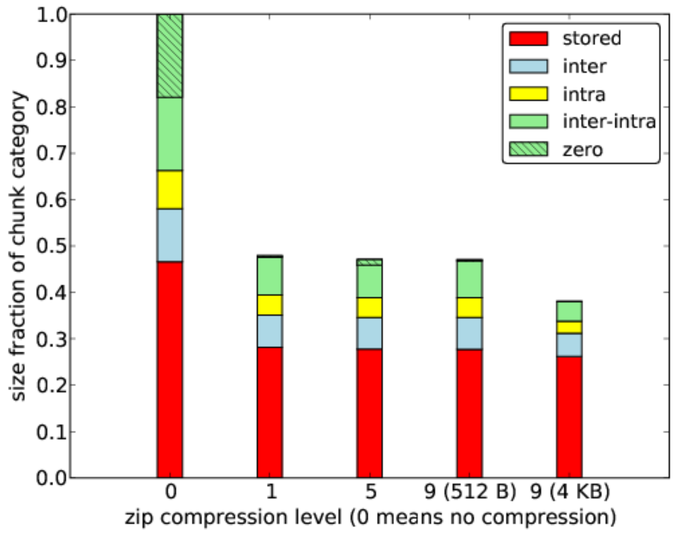


Figure 3.16: Chunk-wise compression. Variable-size chunking, chunk size 512 B, except the rightmost bar for 4 KB. Chunk-wise compression is effective, but limited by small window size.

size of the stored chunks by 40%. Most of the saving is contributed by chunks that occur exactly once; shared chunks are unlikely to be further compressed. Since we only store a single instance of zero chunks, the saving from such chunks is no more than 1 KB. Second, for 512 B chunks, increasing compression level yields negligible benefit because the 512 B chunks do not fully utilize `zip`'s 32 KB sliding window. Third, while larger chunk size yields better compression, this effect does not counteract the lower deduplication ratio generated by using larger chunks, and may deteriorate real-time performance for spending more time on decompression.

Chapter 4

Delta Encoding

The duplicate detections discussed in Chapter 3 are all based on an assumption that the minimum processing unit is measured as a chunk. If two chunks have exactly the same size and data but vary by only several bytes, they are considered distinct. This scenario usually happens when several characters are inserted into one of two identical files. While this granularity is acceptable in some cases, when large chunk sizes are employed for higher system throughput, or the subject virtual machines differ slightly in most portions (*e. g.* branched from a golden base virtual machine), the deduplication ratio would not be optimal. We are especially interested to know what is the cause of the low deduplication ratio exhibited in figure 3.7: are these Ubuntu VMs really differ in large scale, or seemingly different because few bytes such as version numbers in the guest file systems are changed.

Delta encoding is an ideal way to solve the dilemma between chunk size and deduplication ratio. When doing delta encoding on two "distinct" chunks, the outcome space saving is determined only by their relative entropy, not by the layout of the bytes in the chunks. However, delta encoding usually takes more time than content hashing, and blindly selecting candidate chunks from the whole chunk store is unwise. In this chapter, we propose one should only do delta encoding in a spatial locality of similar chunks to achieve space saving. The locality is defined as the range of n sequential chunks, where n is usually small. We call the sequential chunks *ordered chunk sequences*, or *sequence* in short. The fixed-sized or variable-sized chunks forming a sequence must be next to each other in the image file and any skip within the image files is not allowed. Similar as chunk, sequences do not span image files. Therefore chunks from different image files never form sequences. Two sequences are duplicate if their order, the numbers of chunks, and the corresponding chunk data are exactly the same, but appear at different locations.

4.1 Similarity detection

We use the same heurisitic as used in Difference Engine and the system designed by Aronovich *et al.* [21, 5] to identify similar chunks: if the same portions of data in two chunks are identical, then the two chunks are likely to be similar to each other, and worth delta encoding to find out the actual duplicate data. For memory similarity detection, the authors of Difference Engine introduce a parameterized scheme with four parameters. Divide the *page*-byte memory pages into *page*/*block* blocks, each with *block* bytes. Randomly select $k \times s$ unique blocks from a page, and evenly distribute to k sample groups. Hash all s blocks in each sample group and concatenate the digests as the sample group signature. Use all the k signatures as the entries into a hash table with chaining. All the memory pages in a same chain are comparison candidates. Difference Engine chooses the best scheme to make delta encoding between the current chunk and the first c candidates. In Difference Engine, the *page* size is 4 KB and the *block* size is fixed to 64 bytes; the number of groups, number of signatures and number of candidates fully explain a scheme pattern.

We do not adopt the implementation of memory similarity detector directly into the VM locality detection, instead, we modify it to better fit the VM disk image use cases. The first change in our implementation is, while Difference Engine compares the group signatures in the global scope, we do the comparisons only in local scope, thus introduce the scope window size w in the unit of chunk. The group signatures of the chunks inside the window are kept and compared, while the group signatures of the chunks outside the window are abandoned. There is no logical or physical evidence showing that arbitrary nearby data are likely to be related or similar. On the other hand, the neighborhood of duplicate chunks could be good candidates for delta encoding. Viewing the neighborhood as whole, it is likely that the relative entropy of these chunks is lower than arbitrary ones. Given a set of duplicate chunks, we do similarity detection on the two sides of each chunk, with the range from 1 to $w + 1$. The nearby chunks on the different sides are not compared. Thus, the spatial locality size is $2w + 1$.

The reason of this change is because candidate chunk data must be in memory before the delta encoding takes place. In Difference Engine, chunk data are always available in memory. In VM disk image deduplication, chunk data are on disk. Large chunk store and/or big c value could result in numerous disk read and cache operations, thus dramatically deteriorate system performance. Moreover, memory pages are usually allocated randomly to different applications with varied usage patterns. Nearby memory pages do not necessarily have spatial relationships. On the contrary, studies show that internal fragmentation in some file systems is relatively rare [35] and the mean fragmentation path length is small [16]. Data in disk images are mostly allocated sequentially and accessed by applications with certain purposes.

The second change is, we introduce the sample block size b into the scheme parameters. Smaller block size increases detection performance which would be critical in real-time use cases. Bigger block size increases the likelihood of two "similar" chunks being actually similar and saves more data. Unlike memory pages, chunk sizes in disk-based deduplication could vary from 1 KB to as high as 64 KB, and the fixed 64-byte sample block size may not be optimal for all cases. The question naturally arises as to whether it is beneficial to save time on detection by cutting block size in half which would result in numerous 64 KB chunks with average similarity of only 10%? Since delta encoding takes much more time than generating signatures, the answer is usually "no." Our modified scheme pattern is $(k, s), c, b, w$. Table 4.1 summarizes the meaning of each parameter, and Figure 4.1 shows an example of delta encoding on similar chunks.

Because of the extra scope parameter, and because the chunk sizes in variable-size chunking are content based and could vary dramatically within a sequence, we only do research of delta encoding on fixed-size chunking. In fact, Chapter 3 shows that

Parameter	Meaning
k	Number of sample groups for each chunk
s	Number of data blocks for each sample group whose hash values are concatenated as the group signature
c	Number of candidates where the best one is chosen when comparing the group signatures
b	Size of the hashed data blocks for each sample group
w	Size of the adjacency of duplicate chunks where delta encoding is applied

Table 4.1: Meaning of the scheme parameters in our similarity detection method.

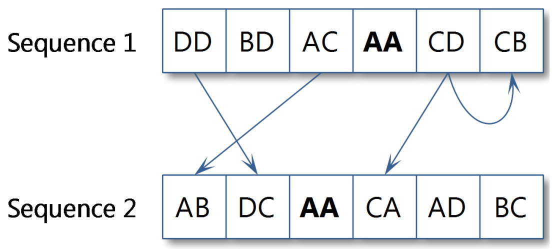


Figure 4.1: Example of delta encoding on two different chunk sequences. The window size is 3. **A**, **B**, **C** and **D** are data blocks in chunks. Each chunk is formed by concatenating two data blocks. The duplicate chunk **AA** acts as the center of the locality. The head of the arrows indicate the target of encoding, while the tail indicate the source. Chunks **CA**, **CB** and **CD** are similar as they sharing the common **C** data block. Chunks such as **BD** are not encoded because their group signatures are unique in the locality.

fixed-size chunking works well for virtual machine disk images. As a result, we initialize the random block numbers for each sample group before the detection begins, since all scheme parameters are determined then, and we use the block numbers throughout the detection.

As the indexing of group signatures is restricted within sequences, even the extreme schemes like $(4,1), 8, 1024, 64$ limit the number of input strings of the block hash function up to 2^{15}. To optimize performance, we use the SuperFastHash [34] as the block hash function, which produces 32-bit digests. The input strings in the worst case are in length of $64 \times 8 = 512$ bits, and 2^{15} is far less than 2^{512}; the lower bound of the probability having no collision is

$$p_{nc} = \prod_{i=0}^{2^{15}-1} (1 - \frac{i}{2^{32}}) \approx 0.8825$$

Although the probability decreases sharply as a small digest length is chosen, unlike chunk deduplication, false positives are merely candidates and will be additionally compared. It is beneficial if the time wasted on comparing false positives is less than the extra time SHA-1 hash takes.

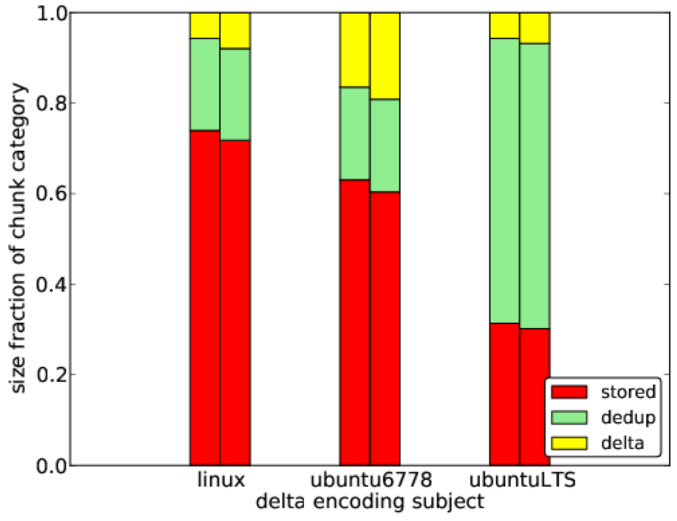
Unlike the chunk compression discussed in Section 3.3.4 that all chunks are self-compressed, the result of delta encoded chunks depend on not only themselves, but also the source chunk of each encoding operation. Since the source is required to be loaded prior to any request to its dependants, the main penalty of the extra space saving shifts from CPU to I/O cost, which is a function of both the chunk size and the window size w. The the VM hypervisor may want to cache the source chunk more often than other chunks. Once the source chunk is updated, all the encodings for its dependent chunks must be recalculated.

4.2 Experiments

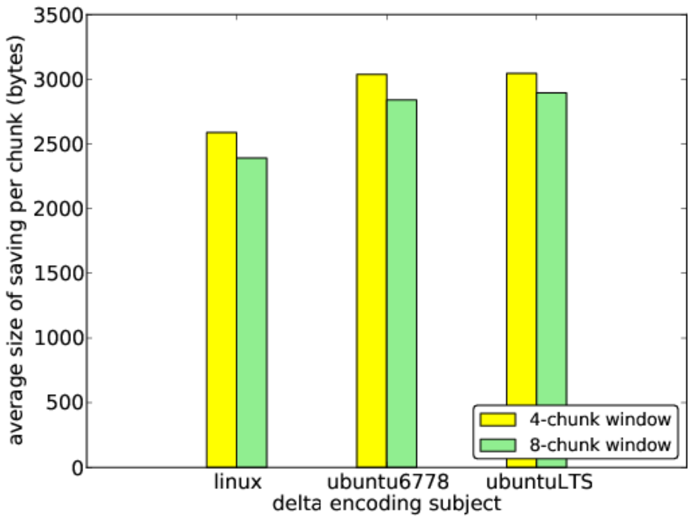
In this section, our goal is to evaluate the effectiveness of the delta encoding applied on the similar chunks. For the consideration of maintaining comparability to the last section, we still use the same set of VMs listed in Table 3.2. For the delta encoding algorithm, we use the open-vcdiff library [39]. All experiments are taken on chunk stores with 4 KB fixed-size chunking, and all zero-filled chunks are excluded in all time.

4.2.1 Overall Effectiveness

Again, we evaluate the overall effectiveness of delta encoding in locality before going into detail on specific factors. The subject VMs in the experiments are the 14 Ubuntu 8.04 LTS (*ubuntuLTS*), Ubuntu Server 6.10 through 8.04 (*ubuntu6778*) and the mixture of various Linux distributions (*linux*). The first 3 parameters in the used scheme pattern are $(2, 1), 1$. We compare both the window size of 4 chunks and 8 chunks for each subject. The signature block size is 64 bytes. Note that if a chunk is identified as duplicate, it is not further been processed by the delta encoding library. The results are shown in figure 4.2(a). As the figure shows, compared to the significant savings that deduplication has in the homogeneous Ubuntu LTS distributions, delta encoding's effect is almost negligible. In contrast, the Ubuntu Server chunk store has as much



(a) Space saving



(b) Average saving per chunk

Figure 4.2: Effects of varying operating system type on delta encoding. In each group of two bars, the left bar measures the window size of 4 chunks and the right bar measures the window size of 8 chunks.

space saving from delta encoding as from deduplication. There are two possible ways to explain the differences in the results. Firstly, the Ubuntu Server chunk store has lower intra relative entropy than the Ubuntu LTS chunk store. To prove or disprove this, we compare the average size of savings per chunk, as shown in figure 4.2(b). Clearly, both chunk stores have almost the same average per-chunk saving for either scheme pattern, indicating that the average impact of delta encoding is equal. This fact disproves the first hypothesis. Secondly, the saving ratio from delta encoding is connected to the deduplication ratio. Since the Ubuntu LTS chunk store is highly duplicated, there are not many distinct chunks left for delta encoding to compress. If a length-w sequence is *duplicate*, none of its chunks is subjected to be delta encoded. On the other hand, the deduplication ratio of the Ubuntu Server chunk store is significantly smaller. Slightly changed chunks, which are not captured by deduplication, have more opportunities to be delta encoded. In short, the delta encoding saving ratio has negative correlation to the deduplication ratio. It is advised to conduct delta encoding in chunk stores where deduplication is ineffective.

Also, while the average per-chunk saving for 4-chunk windows is constantly higher than 8-chunk windows, the corresponding total saving has the opposite results. This pattern proves one of our hypotheses in Section 4.1 that the closer distance between the candidate chunks and duplicate chunk, the more similarity they have, therefore the delta encoding is more effective on average. However, in the larger window sizes, there are more opportunities to do delta encoding, and this becomes the dominating factor in the overall results. The Linux mixture shows the lowest average per-chunk saving,

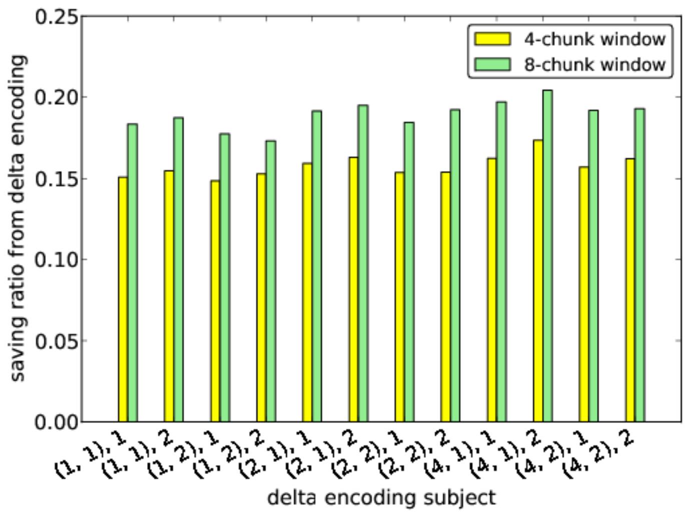


Figure 4.3: Effectiveness of delta encoding for varying the number of groups, number of signatures and number of candidates. All experiments are taken on the Ubuntu Server 6.10 through 8.04 chunk store. The size of signature blocks is fixed to 64 bytes. The pattern is similar to Difference Engine, except less distinctive.

as it is the most heterogeneous one among the three chunk stores. This heterogeneity applies to not only its deduplication ratio, as shown in figure 3.5, but also to the delta encoding approach.

4.2.2 Effectiveness of Specific Parameters

In the next experiments, we focus our attention on specific parameters, and examine which are the most critical ones in affecting the delta encoding.

Figure 4.3 shows the results of delta encoding on the Ubuntu Server 6.10, 7.04, 7.10 and 8.04 chunk store. In these experiments, we fix the size of signature blocks to 64 bytes, and vary the rest of the parameters. At the first glance, the presented pattern is similar to its counterpart in Difference Engine. More signature groups yield a noticeably higher saving ratio, while larger candidates numbers affect the saving ratio to the least scale. If, instead of one, two or more signature groups are independently generated and compared, this will help the similarity detector filter the false positive chunks. More candidates will definitely help increase the saving ratio; however, considering the extra encoding time it consumes, compared with the little gain it brings, it may not be recommended to increase this number in most cases. The parameters with 1 sample group per chunk and 2 sample data blocks per group produce the lowest saving ratio both in memory page sharing and VM disk image deduplication. The effectiveness pattern of the parameter combinations is preserved between different window sizes, meaning our modification of the similarity detector is compatible and independent to Difference Engine's original algorithm.

Examining the absolute values of the saving ratio, the difference between each parameter set is less distinctive than Difference Engine. Possibly it is a result of this specific chunk store, meaning the VMs are so highly similar that even bad parameters are capable of producing high saving ratios. It is also possible that the small distinction is a characteristic of delta encoding on VM disk images. As mentioned before, memory pages are allocated randomly and serve for varied applications and usage patterns. There is no obvious spatial relationship between adjacent memory pages. On the other hand, the sizes of VM disk images are generally much larger than the sizes of installed memory. Same sets of data blocks are more probable to be copied and modified by multiple applications. Considering that memory is temporary while the disk images are permanent, such similarities can be preserved and accumulated.

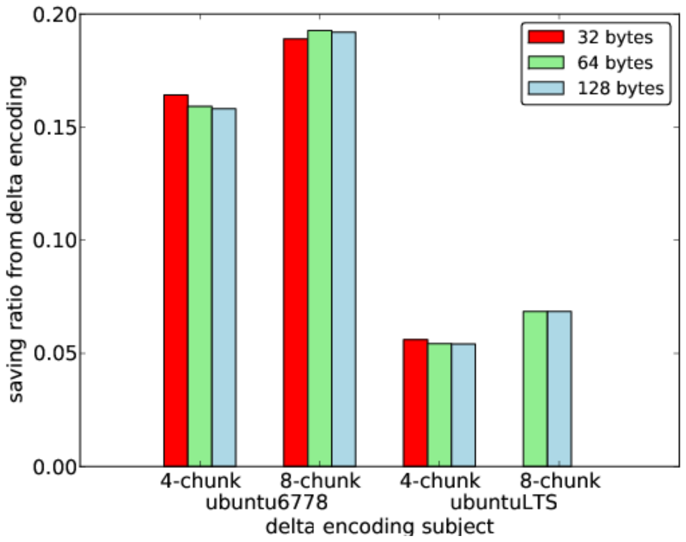


Figure 4.4: Effectiveness of delta encoding for varying window sizes and size of signature blocks. The rest of the three parameters are fixed at $(2, 1), 1$. Small block size does not decrease space saving.

Alternatively, we fix the number of sample groups, sample data blocks and group candidates in the next experiments, and vary the size of signature blocks. For each experiment, two variations with different window sizes are done for reference. The results are presented in figure 4.4. The pattern in the figure is obvious: small signature block size neither decreases the saving ratio with a same window size applied nor affects the saving ratio when the window size is changed. The size of signature block controls the likelihood of the captured similarity for each chunk. When the block size is 128 bytes, the two chunks with identical signature groups are more likely to be similar than the ones when the block size is 32 bytes. However, since the signature groups are selected randomly, large block size may also increase false negative rate. When using a fast hash function like SuperFastHash, the overhead of hashing the extra 96 bytes is trivial comparing to the total delta encoding time. When using a canonical hash function like SHA-512, such overhead may become noticeable.

4.2.3 Time consumption

Although delta encoding is capable of producing significant extra space saving for certain virtual machine groups, the overhead of such operations is not yet clear. It is not economical to gain less saving with delta encoding than with deduplication given that the time consumption of the former is even more than the latter, except when deduplication is ineffective or in limited storage environments. During the experiments in the last sections, we count the time consumed in the deduplication and delta encoding phases, and show the results in Figure 4.5. Because the absolute numbers are highly dependent on the hardware and software configurations, as well as the algorithm implementation and run-time system state (CPU load, memory and I/O usage), we only show the relative ratio of each phase for better comparability.

The pattern from the figure is analogous to what we have observed in Figure 4.2(a); The chunk store with higher saving ratio takes less time to finish delta encoding, while the chunk stores with lower saving ratio consumes more. As the roster of duplicate chunks is prerequisite of our delta encoding algorithm, all the chunks in the chunk store have to be traversed, thus the time consumption is proportional to the total number of chunks. On the other hand, since our algorithm only make patches for the chunks near duplicate chunks, mathematically the number of delta encoding operations is $2 \times w \times n$, where n is the number of duplicate chunks discovered in the deduplication

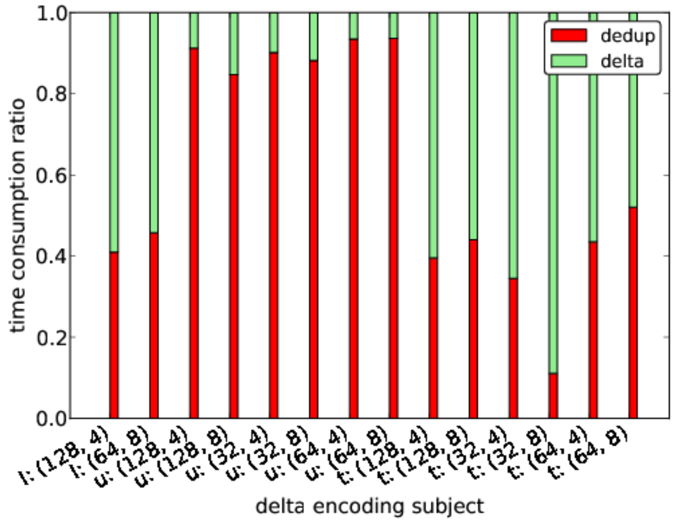


Figure 4.5: Relative ratios of the time consumption in the deduplication and delta encoding phases. Three chunk stores are experimented, regarding the block size and window size combinations in the parentheses on the X-axis. The rest parameters are fixed at $(2, 1), 1$. The letter before parentheses abbreviates the name of the chunk store: **l** for *linux*, **u** for *ubuntu6778* and **t** for *ubuntuLTS*.

phase, and the time consumption is approximately proportional to n. It can be inferred that the time consumption goes up when the deduplication ratio is high. Also, it is not surprising that one delta encoding operation is more costly than one deduplication operation on an average basis. The overhead can be accumulated to a noticeable level if such operations are massive. This could be the explanation of the significant time consumption for the *linux* chunk store, since the size of this chunk store is far bigger than the other two.

Specifically, the time consumption is approximately proportional to the reciprocal of the signature block size. This occurs not because the larger signature blocks take more time to hash, but narrow down the similar chunk candidates. As we have seen, smaller size does not necessarily generate higher saving ratio, it is crucial to carefully choose an appropriate size before doing encoding. Generally, the larger the window size is, the more time it consumes. The counter-examples in the *ubuntuLTS* chunk store can be explained by content-based characteristics or simply by variances of the experiments.

Chapter 5

Future work

5.1 User generated data

Beside the areas of deduplication we investigated in the previous chapters, another important factor for deduplication is user generated data. While all the VMs in our experiments were not used by real users, involving user data in VMs may impact deduplication ratios [30, 31]. Different results may be discovered when identical, homogeneous or heterogeneous VMs are used and deduplicated.

5.1.1 Identical instances

Prepare n identical copies of a single base virtual machine and distribute to n users. In these VMs, users may be granted the super-user privileges, thus be able to install softwares on per-machine basis. Collect the deduplication ratio of such instances over time and examine the trend of ratio change. This is an experiment to reveal whether or not user-related data have correlations and similarities, and how would user-related actions change the original virtual machine data. This is the ideal case for deduplication, though not quite practical for real host centers.

Although it has been shown that package installation order has neglible effect on deduplication ratio, the packages were carefully chosen. It is also good to know if there is an impact on the ratio when users install their own favorite packages. Predictably, popular packages might be installed on almost all the instances while cold packages might appear only once. One can expect to have the deduplication from those popular ones between user and user, especially if they are large.

Apart from package, users may install software, which are not obtained as distribution packages. In this case, users configure the development environment, generate Makefile to compile and install. Since all virtual machines are from the same configuration, it may be expected that the compiled binary are highly duplicated.

Because users have super-user privileges, they may make some changes on system configurations. It will be useful to know if such changes will propagate and accumulate. For example, suppose the base virtual machine is equipped with an X window environment. If the user disables the X window environment, all subsequent software installations may not compile X related files.

5.1.2 Homogeneous instances

Instead of n copies from a same base virtual machine, prepare copies from n homogeneous virtual machines with the same distribution. This might be all Ubuntu

8.04 LTS obtained from different sources. There are some differences in the original system configurations. Distribute them to m users and observe the trend of deduplication ratio. If $n < m$, some of the virtual machines are duplicated instances. We can expect the deduplication ratio in the case of $n < m$ to be between the identical instances case and the case of $n = m$, in average. This experiment is more practical for host centers. They may trust a few certain sources of virtual machines, but not necessarily the same version. It would be good to know whether or not the trend of deduplication ratio in this experiment is comparable to the last section. This would prove or disprove that the impact of user-related actions is independent from system configurations and minor differences.

5.1.3 Heterogeneous instances

As different distributions act in different roles, it is more realistic to survey the trend of the deduplication ratio from heterogeneous instances that are distributed to users. We hope to simulate the actual environment of host centers, since they rarely install only one or two distributions.

Apart from the previous experiments, we would like to collect not only the intra-VM share and inter-VM share, but also the intra-distribution share and inter-distribution share. If two heterogeneous distributions have high inter-distribution share, they may be good choices for host centers. If one particular distribution has high intra-distribution share, space savings can be achieved by hosting it in large amount.

It is also noted that because different distributions have different system components and package systems, users on such VMs might exhibit different kinds of behaviors. For example, users of BSD may do more scientific and computational tasks, while users of Ubuntu may be more interested in multimedia entertainment. This is also a potential source of intra-distribution share and a barrier to inter-distribution share.

5.2 Security

Issues such as privacy and security can be important for virtual machines, particularly when a hosting center contains thousands of disk images. Thus, it can be useful to investigate the use of secure deduplication [36] to combine encryption with deduplication for virtual machines. Doing so promises to provide not only privacy, but also better security since it will be more difficult for an intruder from system A to compromise encrypted chunks from system B.

Chapter 6

Conclusion

Deduplication is an efficient approach to reduce storage demands in environments with large numbers of VM disk images. As we have shown, deduplication of VM disk images can save 80% or more of the space required to store the operating system and application environment. It is particularly effective when disk images correspond to different versions of a single operating system "lineage," such as Ubuntu or Fedora.

We explored the impact of many factors on the effectiveness of deduplication. We showed that package installation and language localization have little impact on deduplication ratio. However, factors such as the base operating system (BSD versus. Linux) or even the Linux distribution can have a major impact on deduplication effectiveness. Thus, we recommend that hosting centers suggest "preferred" operating system distributions for their users to ensure maximal space savings. If this preference is followed subsequent user activity will have little impact on deduplication effectiveness.

We found that, in general, 40% is approximately the highest deduplication ratio if no obviously similar VMs are involved. However, while smaller chunk sizes provide better deduplication, the relative importance of different categories of sharing is largely unaffected by chunk size. As expected, chunk-wise compression dramatically reduces chunk store size, but compression level has little effect on the amount of space saved by chunk compression.

We also noted that fixed-size chunking works very well for VM disk images, outperforming variable-sized chunking in some cases, thus confirming earlier findings [31] stated. In particular, in small chunk stores such as those in the Ubuntu Server series experiment in Section 3.3.1, fixed-size chunking results in better deduplication than in variable-size chunking. This is good news for implementers of deduplication systems, since fixed-size chunking is typically easier to implement, and performs considerably better.

Surprisingly, the deduplication ratio of different releases within a given lineage does not depend heavily on whether the releases are consecutive. We expected to find that, the further away two releases are, the less effective deduplication becomes. We suspect that this effect is not seen because the same parts of the operating system are changed at each release, while large portions of the operating system remain unchanged for many releases. We also noted that, while different releases of a given lineage are similar, large changes are made in operating systems when they are "forked off" from existing distributions.

Finally, we further examine the possibility and effectiveness of incorporating delta encoding into VM disk image deduplication. We use a parameterized algorithm to identify similar chunks, limit the candidate chunks in spatial locality of duplicate chunks, and apply delta encoding on the selected candidates. Fine-grained adjustments can be made on the parameter combinations to meet different usage requirements. The experiments exhibit high effectiveness for homogeneous but poorly duplicated chunk stores, and ignorable impact on chunk stores with good deduplication ratio. For heterogeneous chunk stores with fair deduplication ratio, the delta encoding approach still produce some space saving. Fortunately, the overall time consumptions for just the delta encoding phase in the experiments are small. Therefore this approach can be a considerable post-processing step in a deduplication system.

Throughout all of our experiments, we found that the exact effectiveness of deduplication is data-dependent—hardly surprising, given the techniques that deduplication uses to reduce the amount of storage consumed by the VM disk images. However, our conclusions are based on real-world disk images, not images created for deduplication testing; thus, we believe that these findings will generalize well to a wide array of VM disk images. We believe that the results of our experiments will help guide system implementers and VM users in their quest to significantly reduce the storage requirements for virtual machine disk images for hosting centers with large numbers of individually managed virtual machines.

Appendix A

Duplicate chunk sequences

In this chapter, we introduce the duplicate chunk sequences in chunk stores, and discuss how to identify them. Virtual machine hypervisor could examine the locality information and focus on storing and caching the hottest ones. By doing so, the sequential access pattern on the sequence chunks are preserved, thus improve VM performance, in exchange of less space saving and more memory consumption.

The following definitions are essential for the rest of this section:

Chunk A data block between two specific boundaries in a file. With fixed-size chunking, the offset distance between every pair of boundaries is fixed. With variable-size chunking, the boundaries are content-based [28, 45].

Ordered chunk sequence A set of chunks ordered accordingly to a specific trait in the disk images, whose cardinality is greater than or equal to 2.

Sequence containment Sequence A is contained in sequence B if their intersection equals to A while A and B have different appearances.

Sequence overlap Sequence A and B are overlapped if their intersection is a proper subset of either sequence A or B.

Sequence group Group of fully duplicate sequences with different appearances plus its subgroups.

Sequence subgroup Fully duplicate sequence group that some of its member sequences are contained in another sequences group.

Hotness Measurement of how duplicate an object is, defined as the number of its appearance in the chunk store.

A.1 Motivation

Generally guest operating system disk I/O on 32KB sequential data blocks is highly possible to be mapped to 32KB sequential data blocks in the virtual machine disk image, and then mapped to sequential geometric locations on the physical disk. In deduplicated virtual machine with 4KB fixed-size chunking, this 32KB sequential data blocks may be mapped to 8 different chunks. Because the chunk server stores chunk data according to the chunk IDs, and the chunk IDs are uniformly distributed by the SHA-1 hash function, it is highly possible that the original sequential disk I/O is eventually transformed to discrete I/O, results in performance deterioration.

To address this problem, the chunk server would store single chunks as well as sequences of chunks. Sequences are considered as big “chunks,” and the data are stored in one piece. Therefore the sequentiality of the 32K sequence data in the last paragraph is preserved. This sequence ends up with multiple “chunk sizes” even if it is a fixed-size chunk store. While the single chunks are responsible for storing data, the sequences increase I/O performance. The disadvantages are that duplicate chunks are stored and more memory are needed for the sequence IDs. The trade off between the overhead and I/O performance is a function of the hotness of the stored sequences. An alternative approach is to omit the single chunks if they are part of a stored sequence, with the penalty of performance that, when accessing the chunks in a sequence, the whole sequence must be read first. This approach might be good for deduplication systems whose chunk sizes are smaller than the guest file system’s block size, since it is the nature of guest OS to request data blocks in the granularity of block size.

A.2 Design

The first level locality is fully duplicate sequences in a chunk store, forming sequence groups. We use the quaternary

$$< id, length, size, appearances >$$

to formally define a sequence in the group, where *id* is a SHA-1 digest calculated by running SHA-1 over the concatenation of all the affiliated chunk IDs, *length* is the number of chunks in the sequence, *size* is the size of the sequence in byte and *appearances* is the set of locations the sequence appears. *id* uniquely identifies the content of a sequence for the purpose of deduplication. According to the definition of sequence, *length* is always greater than or equal to 2. Specifically for fixed-size chunking, *size* equals to *length* times the chunk size, though the last locality of an image file could be an exception. For variable-size chunking, there is no necessary relationship between *size* and *length*. Obviously, every chunk in a sequence group is duplicated in the chunk store.

We always favor long sequences over short sequences. That is, if every sequence in sequence group A is contained in some sequences of sequence group B, we only consider B and ignore A, because the locality information in A is redundant. However, if A is a subgroup of B, we do not ignore A and consider A as a independent sequence group, since some sequences in A are not part of B per definition and bear unique information. Finally, overlapped sequence groups are considered no exception as normal sequence groups with the same reason as sequence subgroups.

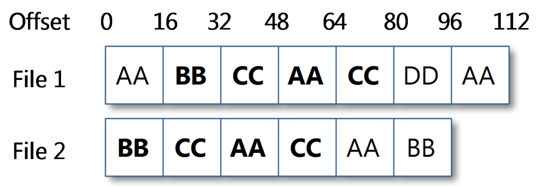
Figure A.1 shows a segment of chunks in a chunk store. In the context, we use *file:start:end* to specify a sequence starting at offset *start* and ending at offset *end* in file *file*. For example, *1:16:48* indicates the sequence starting at offset 16 and ending at offset 48 in file 1, with chunk **BB** and **CC**. Figure A.1(a) is an example of a sequence group. Note that even though 1:16:64 and 2:0:48 are eligible to form a sequence group with length 3, both of them are contained in a longer sequence group with members 1:16:80 and 2:0:64 respectively, and thus ignored. Figure A.1(b) is a case of overlapped sequence groups. Chunk 1:16:32 is shared by both sequence groups, but none of them is contained by the other. Figure A.1(c) exhibits sequence subgroup. Sequence 1:32:64

and 2:16:48 are duplicate with sequence 2:48:80, where the chunk 2:64:80 is independent from the sequence group in Figure A.1(a). Therefore sequence **CC AA** bears unique information and is kept in record. Since chunk **DD** is not duplicated, any sequence includes it does not form sequence group. There are totally 3 sequence groups with 1 subgroup in this chunk segment.

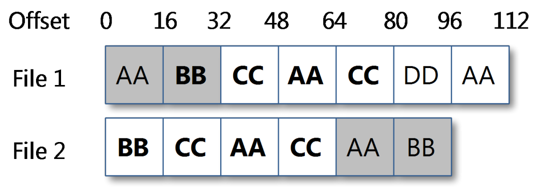
We implemented two strategies to detect level one locality: progressive and greedy. In general, the progressive algorithm deduplicates sequence groups of one specific length in each run. Each run involves two phases. The first phase detects all the duplicate length l localities in the chunk store. The second phase removes all the length $l-1$ sequence groups contained in any of the length l sequence groups and leave subgroups untouched, if there is any $l-1$ sequence groups detected so far. After each run, all the sequence groups with length less than l are conclusive.

Deduplicating sequences is similar to detecting duplicate chunks. Instead of reading one chunk, for every location the algorithm reads in l chunks, concatenates the chunk IDs into a string with length $20 \times l$, applies SHA-1 hash on the string to generate 20 byte sequence ID, and does deduplication. The algorithm moves to the next chunk and process in the same manner, until there are not enough chunks to form a length l sequence in the disk image.

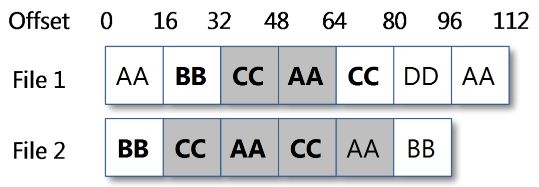
The progressive algorithm begins the first run at length 2 sequences, and progressively increases the sequence length until less than 2 sequences can be formed. To detect all possible duplicate sequences, this algorithm needs to run $N-2$ times, assuming N is the number of chunks in the largest image file. Each run reads every chunk



(a) A sequence group **BB CC AA CC** in bold text

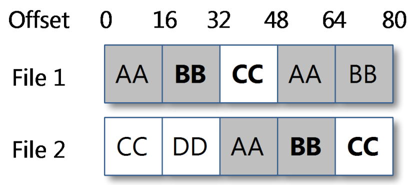


(b) Two overlapped sequence groups **AA BB** in shade and **BB CC AA CC** in bold text

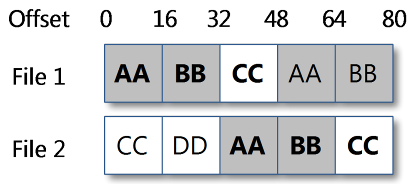


(c) A sequence group **BB CC AA CC** with subgroup **CC AA**, since the chunk **AA** at 2:64:80 is not a part of the former sequence group

Figure A.1: Examples of level one locality.



(a) The first run detects all the length 2 sequence groups **AA BB** and **BB CC**



(b) The second run detects the length 3 sequence group **AA BB CC**, leaving the subgroup **AA BB** untouched

Figure A.2: Illustration of the progressive strategy. The longest sequence group has length 3.

ternary record in the chunk store at least once, therefore a large number of duplicate disk I/Os are inevitable. The progressive algorithm is simple and straightforward to implement, and ensures output of specific length l in merely 2 runs (length l and length $l+1$). However since virtual machine disk images are generally big files, detecting all sequence groups may take a long time. Figure A.2 illustrates the progressive strategy.

We use the concatenation of chunk IDs instead of chunk data to identify sequences by means of reducing disk I/O. We show that this approach does not dramatically increase the probability of hash collision by using the shorter input strings. Assume we have a perfect hash function that generates uniformly distributed b-bit digests for any input string no more than m bits long. This hash function is capable of producing 2^b distinct digests, each of which has $\frac{2^m-1}{2^{b-1}}$ collisions in the input space. Consider the fixed-length input strings with l bits each, given $b < l \leq m$. The total number of distinct strings is 2^l, and the collision number for each digest is reduced to 2^{l-b} for this specific input space. If $b \geq l$, the collision probability of perfect hash function is constantly 0.

Now we have n distinct random strings from the length-l input space, satisfying $n < 2^l$. We consider the shift of the probability having no collision in all strings, because any pair of data blocks from the disk image is correlated. To compute it, each time we hash one string, compute its probability of having collision with any of the previous digests, make the complement probability, and follow the procedure n times. It differs from the probability of the birthday problem [20] since each input string is counted only once. The probability of having no collision in all strings p_{nc} is

$$\begin{aligned} p_{nc} &= \prod_{i=1}^{n} \frac{(2^l - 2^{l-b}) \cdot i}{2^l - i} \\ &= \prod_{i=1}^{n} (\frac{2^l}{2^l - i}(1 - \frac{i}{2^b})) \end{aligned}$$

This is the upper-bound of all possible input, since duplicate strings may be involved in actual practice. Obviously the probability becomes 0 when n goes to 2^b. By fixing b and n, decreasing l results in increasing p_{nc}. However, if n is far less than 2^l, the equation is approximated by

$$p_{nc} = \prod_{i=1}^{n} (1 - \frac{i}{2^b}) \tag{A.1}$$

a value not related to l.

For SHA-1 hash, b is constantly 160 [3], therefore the collision probability is bounded by the number of chunks n and chunk size l. In our experiments, the smallest chunk size is 8192 bits (1KB), and the smallest chunk ID concatenations are $2 \times 40 = 80$ bytes (we use the ASCII representation of the SHA-1 hashes), or 640 bits. 1TB chunk store generates 2^{30} 1KB chunks and at most $2^{30} - 1$ chunk ID concatenations. It is safe to consider that $2^{30} - 1$ is far less than either 2^{640} or 2^{8192}. Thus, unless the SHA-1 digests are badly distributed in the space, the concatenation approach will not dramatically change the collision probability.

On the other hand, the greedy algorithm finishes detecting all sequence groups in one run, with the prerequisite that all chunk records are loaded in memory (alternative method is discussed in section A.3.1). The algorithm visits each chunk in a specific order (discussion on detection order is in section A.3.3) in the index. For each of the visited chunk, locate its other duplicate instances in the chunk store, and greedily extend the sequences starting from these chunks. We call the first chunk of each sequence the *head chunks*, the first such sequence in the detection order the *head sequence*, and the rest of such sequences the *candidate sequences*. The extending procedure is repeated on the head sequence and all the candidate sequences. Once the last extended chunk differs from the one in the head sequence or the end of chunk store is encountered, the extending is stopped on that candidate sequence.The extending on the head sequence stops when all candidate sequences have stopped extending. Obviously, if the head chunk of the head sequence is an unique chunk, the extending procedure does not happen. Otherwise, at the end of extending, there is at least one candidate sequence whose length is as long as the head sequence, all of which form a sequence group with the head sequence. All the candidate sequences whose lengthes are shorter than the head sequence, and no less than two, form sub-localities. All the candidate sequences whose lengths are less than 2 are ignored.

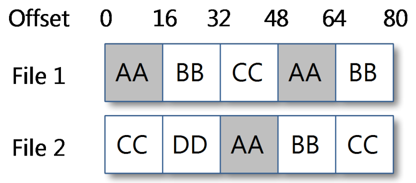
To avoid duplicate sequence groups in the detection result, the head chunks of group members are recorded, and will be skipped in the rest of the second run. To avoid contained sequences, every processed head chunk is assigned a non-negative *minimum sequence length* after extending or skipping, defined as $l - d$ with l be the length of the most recent sequence and d be the chunk distance between the head chunks of the current and the most recent sequence. Whenever the current sequence is concluded with a length less than its head chunk's minimum sequence length, this sequence is contained in a previous sequence group and should be ignored. The initial minimum sequence length for all chunks is 0. Figure A.3 illustrates the greedy strategy.

A.3 Discussions on the greedy algorithm

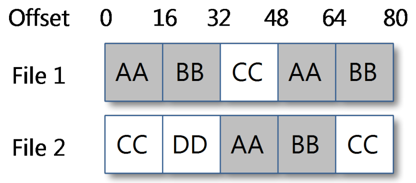
There are four significant factors to be considered when implementing the greedy algorithm:

A.3.1 Postprocessing or inline deduplication

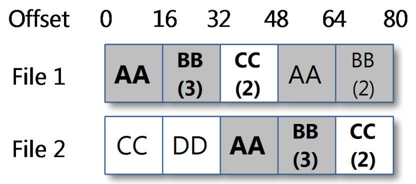
Postprocessing deduplication means all data for the deduplication process are ready before it starts. Inline deduplication means while some or no data is available when the deduplication process starts, more data will come during the deduplication



(a) Before extending the current head chunk **AA**



(b) Extend the head chunk **AA** once to **BB**



(c) Extend twice to **CC**. The numbers in parentheses are the minimum sequence lengthes.

Figure A.3: Detection result of the first sequence group with the greedy strategy. In the second extending, 1:48:80 hits the implicit boundary, and becomes a sequence subgroup.

process. The former mode is usually used in archive system backup, and the latter one is best for realtime data backup. In postprocessing deduplication, full data analysis and sorting are possible, and the index and/or content (if memory size permits to fit) of the data can be loaded prior to the deduplication process. In inline deduplication, only parts of the total data are accessible at all times, and the deduplication algorithm can only run progressively. A queue of the most recent chunks are kept on record, and the oldest ones are abandoned when the queue is full. Therefore, the inline deduplication can be seen as multiple local version postprocessing deduplication, which is updated progressively. In our experiments, the subject virtual machine disk images are static and always fully available. The extra flexibility of postprocessing deduplication simplifies the algorithm logics and increases its performance by keeping less footprints and utilizing more memory on hash caching, especially for our Berkeley DB implementation. Therefore, we mainly focus on postprocessing deduplication in our experiments. However, when implementing on-the-fly VM deduplication systems, inline implementation would be the only choice, since the disk data are constantly changing. More related information can be found in [27].

A.3.2 Minimum unit (postprocessing dedup only)

In the description of the algorithm above, the Minimum unit is the native chunk, and the algorithm identifies duplicate sequences by comparing the ID of chunks. Chunk IDs are always available before the locality detection starts, therefore no pre-processing is needed. However, since a single chunk does not form a sequence, locating a duplicate chunk does not necessarily mean to locating a locality. Therefore, many random disk I/Os would be necessary since the duplicate chunk instances are usually scattered in the chunk store, and an *initial extend* must be always included in the algorithm, which slightly perplexes the implementation. On the other hand, using length 2 sequence as the Minimum unit requires joining every two adjacent chunks into sequences before the locality detection starts. Fortunately, deduplication can be applied on these length 2 sequences, and only the duplicate sequences are locality candidates. This alternative would greatly shrink the detection time (depending on the workload) and simplify the implementation.

A.3.3 Detection order

There are two detection orders of the locality detection. *By offset* is defined as visiting every chunk from the first bit of the first image to the last one. It is the nature order because the locality is a set of sequences. Since the original chunks are usually processed and stored with this order, this order requires no additional sorting procedure. As it is guaranteed that all the visited head chunks will not be revisited again, one can remove the bookkeeping information of such chunks to save memory utilization and improve detection performance. On the contrary, *by ID* is a detection order that takes all chunk instances of a same ID into account at once, processes them as the head chunks, and moves to the next chunk ID in specific sorting direction (dictionary, alphabetical, etc.). The number of distinct chunks are usually much less than the number of chunk appearances, thus, traversing through all the targets could be faster than the first detection order, making it efficient in highly duplicate chunk stores. However, as it does *not* guarantees that all the head chunks will be visited only once, postprocessing to remove contained sequences will be required.

A.3.4 Multi-threading

Multi-threaded locality detection is feasible. One approach is to start all n threads at the beginning of the chunk store, and follow the *by offset* detection order until there are not enough chunks left in the chunk store to form a sequence. Each of them uses different but adjacent chunks as the head chunks, off by one chunk. When thread t_1 finishes its detection at chunk c_1, the detection result of thread $t_1 + 1$ at chunk $c_1 + 1$ could be used to verify if the latter is a contained sequence of the former with short-term memory bookkeeping. Although this is not always true for another instance of chunk c_1 and $c_1 + 1$, it is highly possible if the sequence starting from c_1 is a locality. Moreover, since all the head chunks are logically adjacent, a cache system could improve the detection performance with high hit rate (depending on the workload). On the other hand, if the threads detect locality *by ID*, logically adjacent chunks are not necessarily to be visited sequentially, and a cache system cannot help improving performance no matter whether it is in context of prefetching future chunk data or by verifying contained sequences.

It is also natural to consider the divide-and-conquer strategy, which evenly divides the work load into n parts, assign each part to one thread, and merges the result after all the threads finish. This approach could yield higher performances by taking advantage of parallelism when the chunk store are stored in a multi-disk environment, *e. g.* RAID-0 or RAID-5 disk arrays. However, bookkeeping is necessary for all the processed chunks in all threads, since some localities may be scattered across threads. In case of large chunk stores, the memory capacity would be easily used up, and the performance deteriorates as more memory pages are flushed to disk.

Bibliography

[1] M. Ajtai, R. Burns, R. Fagin, D.D.E. Long, and L. Stockmeyer. Compactly encoding unstructured inputs with differential compression. *Journal of the ACM (JACM)*, 49(3):318–367, 2002.

[2] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: Scaling file servers via cooperative caching. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, pages 129–142, 2005.

[3] Anonymous. Secure hash standard. FIPS 180-1, National Institute of Standards and Technology, April 1995.

[4] IBM Anthony Liguori and IBM Eric Van Hensbergen. Experiences with content addressable storage and virtual disks. In *First Workshop on I/O Virtualisation*, 2008.

[5] Lior Aronovich, Ron Asher, Eitan Bachmat, Haim Bitner, Michael Hirsch, and Shmuel T. Klein. The design of a similarity based deduplication system. In *SYSTOR '09: Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, pages 1–14, New York, NY, USA, 2009. ACM.

[6] `http://bagside.com/bagvapp/`.

[7] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, 2003.

[8] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[9] Deepavali Bhagwat, Kave Eshghi, and Pankaj Mehra. Content-based document routing and index partitioning for scalable similarity-based searches in a large corpus. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '07)*, pages 105–112, August 2007.

[10] Austin Clements, Irfan Ahmad, Murali Vilayannur, and Jinyuan Li. Decentralized deduplication in san cluster file systems. In *Proceedings of the USENIX Annual Technical Conference*, June 2009.

[11] Lasse Collin. A quick benchmark: Gzip vs. Bzip2 vs. LZMA, 2005.

[12] 2005. `http://www.7zip.org`.

[13] 2005. `http://www.bzip.org`.

[14] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: Making

backup cheap and easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 285–298, Boston, MA, December 2002.

[15] M. Daum and S. Lucks. Hash Collisions (The Poisoned Message Attack) “The Story of Alice and her Boss”. *Presentation at Rump Sessions of Eurocrypt 2005*, 5, 2005.

[16] Gaurav Deshpande and Sachin Manpathak. A revaluation of modern file systems.

[17] P. Deutsch. Deflate compressed data format specification version 1.3. 1996.

[18] John R. Douceur, Atul Adya, William J. Bolosky, Dan Simon, and Marvin Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS ’02)*, pages 617–624, Vienna, Austria, July 2002.

[19] N. Freed and N. Borenstein. Multipurpose Internet Mail Extensions (MIME) Part One: Format Of Internet Message Dodies, 1996.

[20] M. Girault and M. Campana. A generalized birthday attack. In *Lecture Notes in Computer Science on Advances in Cryptology-EUROCRYPT’88*, pages 129–156, New York, NY, USA, 1988. Springer-Verlag New York, Inc.

[21] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference Engine: Harnessing memory redundancy in virtual machines. In *Proceedings of the 8th Sym-*

posium on Operating Systems Design and Implementation (OSDI), pages 309–322, December 2008.

[22] Val Henson. An analysis of compare-by-hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, 2003.

[23] Val Henson. An analysis of compare-by-hash. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS-IX)*, May 2003.

[24] Jeffrey Hollingsworth and Ethan Miller. Using content-derived names for configuration management. In *Proceedings of the 1997 Symposium on Software Reusability (SSR '97)*, pages 104–109, Boston, MA, May 1997. IEEE.

[25] Jr. Kingsley G. Morse. Compression tools compared. *Linux J.*, 2005(137):3, 2005.

[26] D. Korn, J. MacDonald, J. Mogul, and K.V. RFC. 3284: The vcdiff generic differencing and compression data format. *IETF, June*, 2002.

[27] NetApp Larry Freeman. Looking beyond the hype: Evaluating data deduplication solutions. *NetApp White Paper*, WP-7028-0907, 07, May, 2007.

[28] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 174–187, October 2001.

[29] Eugene W. Myers. An O (ND) difference algorithm and its variations. *Algorithmica*, 1(1):251–266, 1986.

[30] Partho Nath, Michael A. Kozuch, David R. O'Hallaron, Jan Harkes, M. Satyanarayanan, Niraj Tolia, and Matt Toups. Design tradeoffs in applying content addressable storage to enterprise-scale systems based on virtual machines. In *Proceedings of the 2006 USENIX Annual Technical Conference*, 2006.

[31] Calicrates Policroniades and Ian Pratt. Alternatives for detecting redundancy in storage systems data. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 73–86, Boston, MA, June 2004. USENIX.

[32] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, pages 89–101, Monterey, California, USA, 2002. USENIX.

[33] Michael O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[34] `http://www.azillionmonkeys.com/qed/hash.html`.

[35] K. Smith and M. Seltzer. File layout and file system performance. Technical Report TR-35-94, Harvard University, 1994.

[36] Mark W. Storer, Kevin M. Greenan, Darrell D. E. Long, and Ethan L. Miller. Secure data deduplication. In *Proceedings of the 2008 ACM Workshop on Storage Security and Survivability*, October 2008.

[37] `http://www.thoughtpolice.co.uk/vmware/`.

[38] `http://xdelta.org/`.

[39] `http://code.google.com/p/open-vcdiff/`.

[40] `http://www.vmware.com/appliances/`.

[41] VMware, Inc. Introduction to VMware Infrastructure, 2007. `http://www.vmware.com/support/pubs/`.

[42] VMware, Inc. Virtual Disk Format. VMware web site, http://www.vmware.com/interfaces/vmdk.html, 11 2007.

[43] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding collisions in the full SHA-1. *Lecture Notes in Computer Science*, 3621:17–36, 2005.

[44] Lawrence L. You. *Efficient Archival Data Storage*. PhD thesis, University of California, Santa Cruz, June 2006. Available as Techncial Report UCSC-SSRC-06-04.

[45] Lawrence L. You, Kristal T. Pollack, and Darrell D. E. Long. Deep Store: An archival storage system architecture. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*, Tokyo, Japan, April 2005.

[46] Benjamin Zhu, Kai Li, and Hugo 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)*, February 2008.

[47] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, May 1977.