IZO: Applications of Large-Window Compression to Virtual Machine Management

Mark A. Smith, Jan Pieper, Daniel Gruhl, and Lucas Villa Real – IBM Almaden Research Center

ABSTRACT

The increased use of virtual machines in the enterprise environment presents an interesting new set of challenges for the administrators of today's information systems. In addition to the management of the sheer volume of easily-created new data on physical machines, VMs themselves contain data that is important to the user of the virtual machine. Efficient storage, transmission, and backup of VM images has become a growing concern. We present IZO, a novel large-window compression tool inspired by data deduplication algorithms, which provides significantly faster and better compression than existing large-window compression tools. We apply this tool to a number of VM management domains, including deep-freeze, backup, and transmission, to more efficiently store, administer, and move virtual machines.

Introduction

Virtual machines are becoming an increasingly important tool for reducing costs in enterprise computing. The ability to generate a machine for a task, and know it has a clean install of a given OS can be an invaluable time saver. It provides a degree of separation that simplifies support (a major cost) as people all get their "own" machines [26].

A side effect of this, however, is that there can be many machines created in a very short time frame, each requiring several gigabytes of storage. While storage costs are falling, terabytes are not free.

We believe that large-window compression tools provide much needed relief in this domain, because they often perform extremely well on large data sets. Large-window compression has not received much attention in the past because archiving tasks have traditionally involved only hundreds of megabytes. Moreover, traditional tools such as gzip perform consistently well for small and large data sets. However, falling storage costs and larger storage applications (e.g., virtual machines) have driven archiving to hundreds of gigabytes or even hundreds of terabytes. Large-window approaches provide significant additional compression for data sets of these sizes, and complement small-window compression.

One of the most exciting results of our experiments is that large-window compression does not impact the compression factor achieved by a small-window compressor. The combined compression ratio is close to the product of the individual compression ratios. In some cases we even noticed small improvements in the compression ratio of gzip when applied to data that had been processed by our large-window compression algorithm. Moreover, the total time required for large-window and small-window compression combined is often smaller than that of the small-window compression alone.

We find this property analogous to the way a suitcase is packed. It is possible to simply stuff all of one's clothes into a suitcase, sit on the lid, and zip it up. However, if the clothes are folded first, and then compressed, they will fit in a smaller space in less time. Large-window compression can be thought of as the folding step, while small-window gzip style compression is the sitting and zipping. Using the term compression for both techniques can become confusing, so in such cases, we refer to large-window compression as data folding.

In this paper, we present three applications of large-window compression for the efficient management of virtual machines. We also introduce IZO, a novel large-window compression tool using data deduplication, which accomplishes significantly better data folding than existing large-window compression tools.

The first application is the use of large-window compression in cases where long-term storage of "retired" machines is desired. This is a common case when a group of machines have been generated for tasks that are no longer active, but which may become active again. Examples include yearly activities, or machines for people who are only present periodically (e.g., summer students). In this scenario a number of similar machines are to be placed in "deep freeze."

The second task is that of distributing machine images. Core images of machines are often used for the "base machine" with users having their own mounted filesystems. The core image is usually maintained by experienced IT staff, e.g., ensuring that the latest security patches and drivers are installed properly. In a global organization these new images need to be distributed to remote sites. For example, in the financial sector it may be necessary to distribute updated machine images to all bank branches. Unfortunately, these images can be quite large. A problem that can be compounded by low or moderate Internet connectivity for remote sites. We observe that each image tends to be very similar to the previous one. We therefore investigate whether a smaller patch file can be created by calculating the binary delta between similar virtual machines. We use the series of RedHat Enterprise Linux installs from version 4 to create a set of virtual machines from the initial release to updates 1-6 and calculate binary deltas between these machines.

The last task is in creating regular backups of these systems. Traditional backup tools need to be installed on the actual machine and require the machine to be running during the backup. They are often specialized on a specific operating system and filesystem. One benefit of these specialized solutions is that they provide incremental backups: the ability to identify which files were modified since the last backup, in order to save storage space on the backup device. This can be difficult if the virtual machines are running a variety of (potentially arcane) operating systems or filesystems. We instead look at creating backups of the whole virtual machine image, and using large-window deduplication to identify the changes between different revisions of the backup, to achieve the footprint of incremental backups without specialized knowledge of the inner workings of these virtual machines.

In all these cases there is considerable redundancy or duplication of data – for example, files that appear on multiple filesystems. These duplicates may occur gigabytes apart in the archive, and thus traditional streaming short-window compression schemes such as gzip are unlikely to "see" both duplicates within their window, and therefore cannot compress them.

We are working on a novel compression tool called IZO (Information Zipping Optimizer), which was initially targeted at compressing ISO CD images. IZO uses data deduplication algorithms to identify and remove global redundancies. Because large-window compression tools are expected to be applied to hundreds of gigabytes of data, we desire reasonable memory requirements and fast data processing. Moreover, a scriptable interface is important to allow automation of the different compression scenarios. For IZO, the command line interface has been modeled after tar, providing intuitive ease-of-use to users that are familiar with this tool. IZO also provides the same streaming semantics as tar – it requires a set of files or directories as input, and can stream the compressed output to stdout, or to a specified archive. IZO only removes large-window redundancies, therefore the output should be streamed into a short-window compressor of the user's choice.

In the remainder of this paper we will examine some background on the compression and data deduplication approaches used (Section Background), outline the scenarios we are examining (Scenarios), provide a brief tutorial on large-window compression and outline our implementation (Tutorial and Implementation), discuss the experiments and their results (Experiments) before concluding with some thoughts on how these tools can be extended even further (Conclusions and Future Work).

Background

Virtual machines are an emerging tool for allowing multiple "guest machines" to be run on a single piece of hardware. Each guest thinks it has a whole machine to itself and the host machine juggles resources to support this. A full discussion of the challenges and opportunities of virtual machines are beyond the scope of this paper, but the reader is encouraged to consult [19, 28, 21] for a good overview. A good introduction to the open source virtual machine monitor Xen can be found in [5, 6] and the commercial product VMWare in [27, 22].

We are working on a novel large-window compression tool called IZO. IZO draws on concepts from existing tools and technologies including tar [7], gzip [9], rzip [24, 25] and data deduplication [29, 14]. It is invoked on the command line using parameters identical to the well-known tar utility to produce an archive of multiple files or directories.

Large Window Compression

Traditional LZW-style compression techniques are notoriously poor at handling long-range global redundancy. This is because they only look for redundancy within a very small input window (32KB for gzip, and 900KB for bzip2 [20]). To address this shortcoming, rzip [24] was developed to effectively increase this window to 900 MB. Rzip generates a 4-byte hash signature for every offset in the input file and stores selected hashes as {hash, offset} tuples in a large hash table. Data at offsets that produce the same hash signature may be identical. The data is re-read and a longest-matching byte sequence is calculated. If a matching byte sequence is found, a redirecting (offset, length) record is written to the rzip file instead of the duplicate data. This method achieves extremely high compression ratios, but suffers from two main drawbacks as described by rzip's author Andrew Tridgell in [24]. First, rzip has large memory requirements for storing a hash table entry for every offset in the input file. Second, it cannot operate on input streams because it requires random access to the input data to re-read possible matching byte sequences.

Rzip also compresses the deduplicated archive using bzip2 at the highest compression level. Unfortunately, this step may require significant resources and time. Lrzip [10] provides an extended version of rzip, allowing the user to chose between different short-range compression schemes or to execute rzip's long-range compression only. In the experimental evaluation, we compare IZO to lrzip, because lrzip allows us to assess the benefit of rzip's long-range compression in isolation.

Even though IZO achieves large-window compression as well, its roots are quite different. It was inspired by our work on the StorageNet deduplication filesystem [18, 23]. Data deduplication filesystems are a recent trend in enterprise class storage systems, such as NetApp ASIS, DataDomain, Sepaton DeltaStor and Diligent HyperFactor. IZO was created on the insight that the deduplication algorithms used in these filesystems could provide data compression as a command line tool similar to tar or gzip as well.

CZIP [13] takes a similar approach, storing compressed data in a single archive file. But it focuses on efficient transmission of data over a network by maintaining a "Content-Based Named" CBN dictionary of chunks, which has to be stored in the archive as well. CZIP enables Servers to prevent cache-pollution by using this dictionary to only load a unique instance of a particular chunk, while clients can decrease network transmission by re-using already-sent duplicate data chunks. IZO, in contrast, is designed as an offline compression tool. Creating the smallest possible archive from its input is its primary goal and thus we perform optimizations such as discarding the CBN dictionary, etc. to reduce the total compressed size. The use of a CBN dictionary also precludes CZIP from being able to stream both into its compressor and decompressor. To stream into the compressor, the CBN dictionary must be written to the end of the archive; while to stream into the decompressor, the CBN dictionary must be at beginning of the archive.

Data Deduplication

Data deduplication can eliminate global redundancies across multi-terabyte-scale corpora [11]. Early deduplication filesystems include Deep Store [30] and Venti [16], which are also called content-addressable storage systems.

These systems identify duplicate data when it is written to the filesystem. They then reference the existing data instead of storing another copy. Duplicates can be detected on a whole-file level, but current solutions usually detect sub-file redundancies. This is done by breaking files into chunks. A chunk signature is created using cryptographically strong hashing of the chunk content to avoid collisions. Chunks with the same hash value are considered duplicate, and only a single instance is stored. Files are comprised of a list of unique data chunks and are reconstructed during read. The list of unique hashes needs to be kept in memory, to provide fast lookup for newly incoming data. Therefore, chunk sizes in these systems are generally large, typically between 4KB and 32KB.

Different methods exist to split the data into chunks. The chunking method and chunk size strongly influence the speed of the system and the overall data compression. A detailed discussion of chunking approaches can be found in the data chunking section below.

Commercially available deduplication products use a wide variety of chunking approaches. NetApp ASIS uses a modified version of WAFL to deduplicate at a 4K fixed-block granularity [8, 12]. Fixed-block chunking is very fast, but provides comparatively low folding factors. Data Domain's Global Compression uses content-agnostic chunking to produce chunks of 8KB average size [2]. Content-agnostic chunking is based on a rolling hash function, such as Rabin Fingerprinting [17] and provides better folding factors, but at the expense of speed. Sepaton's DeltaStor is content-aware, with built-in knowledge of common data formats for optimal chunking [3]. This is the slowest method because data must be interpreted, but can provide the highest folding factors. HyperFactor, by Diligent, departs from the typical chunking methodology, and relies instead on using data fingerprints to identify similar data objects, and stores the differences between them [1]. Computing the differences requires a read of the most-similar identified data object, but because chunk hashes are neither computed nor compared, this technique does not suffer from the possibility of hash collisions.

Storing virtual machines on a deduplication filesystem would provide similar benefits as the ones we hope to accomplish in our scenarios. Moreover, live filesystems allow virtual machines to be deduplicated even when they are running. However, these benefits are lost when a set of machines is moved to a non-deduplication storage device, such as tape or a different disk array. We are interested in evaluating the usefulness of a simple deduplication compression tool that can be used on a standard desktop and doesn't require the acquisition of specialized storage hardware. IZO bridges this gap by providing a deduplication archive, which can be stored on any storage device without loss of compression.

Binary Diff

It is curious to note that the algorithms employed by rzip and data deduplication can also be used to calculate the differences between two files. The difference is defined by all data in the second file that is found to be non-duplicate with regards to the first one. Binary deltas between two versions of an executable are useful to create patch files that are significantly smaller than the new executable. Our VM distribution scenario in Section "Scenarios, VM Distribution" evaluates the ability of IZO to create binary diffs. The equivalent tool from the rzip family is xdelta [24, 4].

Scenarios

In all the following scenarios we are trying to put ourselves in the "mindset" of a systems administrator for a virtual machine farm. We want approaches that reduce the cost/effort of maintaining such systems, are simple enough to be useful within seconds of reading the man page, are scriptable, and hopefully work well with existing infrastructure. To this end we use gzip for small-window compression, and IZO with its "tar"-like syntax for large-window. As such it can be used as simply as

```
izo -cv /mnt/vmimages | \
        gzip > freeze.igz
```

Deep Freeze

A not uncommon scenario with virtual machine management is to have a number of machines that are no longer being used. However, the data on them is too valuable to just throw away. We've begun to archive working demo systems at the end of project life cycles as virtual machines. These systems provide self-contained "time capsules" of code, libraries and configuration that can be "revived" at a later time.

The desire is to take a set of similar machines and deep freeze them – that is compress the set as much as possible, and then place them on low-cost storage (potentially near or off line). If they are ever needed again it is understood that it may take a day to restore them, and the move to deep freeze can be done in batch during "off time." If there are a number of machines that need to be frozen they may be done in batches, or they may trickle out (e.g., as summer students leave). Thus we would also like to be able to "append" new machines to the end of a freeze.

As a proxy for this, we use images of machines from a set of computers generously provided by a local after school program. The 11 machines all started life with similar installs of Windows XP, but have "evolved" over the course of a year of student use. We look at this set for examples of how a group of similar but not identical machines can be deep frozen together.

Deep freeze is most effective if the machines in each freeze set are reasonably similar. They should, for example, have the same underlying operating system and preferably be used for similar tasks. Fortunately this is usually the case in enterprises.

VM Distribution

One common use of virtual machines is to provide many employees with their "own" machines, on which their home directory is mounted but on which they have limited permissions to modify the rest of base machine. This simplifies the support task, as a central IT department can create and test the image which is then distributed. For global organizations this last step is non-trivial. If the images are generated in South East Asia, the testing is done in North America, and the images are deployed to company branches across Europe, sending multi-gigabyte images around can become quite cumbersome – especially when network connectivity to some sites is slower than may be hoped.

We find, however, that the differences between these base images can be quite small. Using the original image as a "template," the updates then become "deltas." We use RedHat Enterprise Linux 4 WS as our test image for this scenario. As a framework, we used Xen on a Fedora Core 7 machine, but since we were using hardware virtualization from the processor and raw files for disk images the same results can be achieved using other VM solutions. We installed each operating system on a 10 GB partition answering "yes" to all questions, starting with the base RHEL4 and then doing the same for the updates u1 through u6. We then create the binary delta between the initial RHEL4 image and the RHEL4U1 image; the RHEL4U1 image and the RHEL4U2 image; and so on.

The goal then is to create incremental patches that can be as small as possible, reducing the "cost" of an update and thus encouraging shorter and less cumbersome update cycles – critical in the case of security or machine stability issue patches.

Backup

Virtual machines give tremendous freedom to IT professionals. With the emergence of full hardware virtual machine support in both Intel and AMD chip sets, it is possible for users to run whatever operating system and programs best support their tasks. This freedom comes at a price. Inevitably someone will be running OS/2 Warp with the HPFS filesystem.

Despite the wide variety of operating systems, versions, usage scenarios, etc. there is still an expectation that system administrators will provide basic support (such as system backup and recovery). Support for modern backup clients in OS/2 Warp is unfortunately limited, and even mounting the HPFS filesystem can prove problematic. We thus need a way to backup such machines and not rely on the users to maintain copies of important files.

This is facilitated by the proliferation of snapshot functionality – either at the storage level in NAS/SAN products, or in software such as the LVM system. These facilities allow a snapshot to be taken of a running system which can then be saved as a backup.

This snapshot is essentially a disk image file, often many gigabytes large. Keeping a snapshot from every week is an expensive proposition. What is desired is the ability to do a binary "diff" of each image and store only the changes. This is thus similar to the former task, although the data evolution is quite different. Additionally, real savings can be obtained by performing snapshots of a number of machines (as per the "deep freeze" scenario).

We will look at backing up a single user's machine every week for a number of weeks. The backups are full filesystem images of a machine running Windows XP, including all data, application, and system files. The machine is one of the authors' actual workstations, used every day during the work week for email, office applications, and software development, and thus is typical of the kind of machine one might find in a corporate environment.

We repeat this experiment with a virtual machine running OS/2 Warp 4. This machine is not actively being used, but different snapshots are taken after modifying the filesystem by adding and removing files.

Tutorial and Implementation

Large-window compression/deduplication is a fairly new technology to most IT professionals, so we will review the basic idea behind it, and the choices we made for our IZO implementation. For those already familiar with large-window compression (and/or not interested in the implementation details) skipping to the next Section might be advisable.

IZO operates like an archiver with built-in data deduplication facilities. To perform data deduplication and create archives, IZO implements three core components: a chunking algorithm, a chunk hash table, and an archive format generator.

Data Chunking

As described earlier, the core technology choice in a deduplication implementation is the method by which the data is chunked. The major chunk generation schemes are as follows:

Fixed-Size Chunking: Fixed-size chunking breaks data into chunks of a specific size. It is very simple, fast, and the resulting chunk size can be selected to optimally align with the physical block size of the underlying storage device to maximize storage utilization. However, a major drawback of fixed-sized chunking is that shifting the data by some offset may result in completely different chunks. This issue is illustrated in Figure 1.

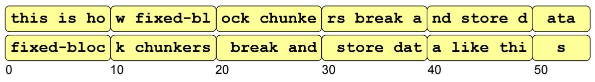


Figure 1: Fixed-size chunking.

Content-Aware Chunking: This method, illustrated in Figure 2, generates chunks by parsing the input files and understanding their formats. Chunk boundaries can be custom generated for the highest probability of redundancy. In this example, boundaries are created between words. This method does not suffer from byte-shifting issues, and can produce larger chunks, but is useful only for known data types and can be very expensive because the data must be interpreted.

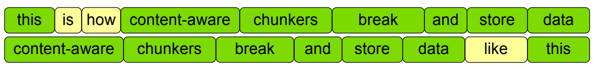


Figure 2: Content-aware chunking.

Content-Agnostic Chunking: Content-agnostic chunkers use the content to determine chunk boundaries, but do not understand the content format. Instead, they consider the features of the byte sequence to deterministically identify boundaries.

This is usually done by generating a rolling hash over a window of the input stream, e.g., using Rabin fingerprinting [17]. Each hash is masked and the resulting value is placed through a modulus operation. If the result of the modulus is zero, then a chunk boundary is created. The modulus value determines the frequency of chunk generation. A modulus of 16, for example, would produce an average chunk size of 16 bytes. Chunks produced this way have a high probability of being duplicates with other chunks produced using the same method. This is because all such chunks are guaranteed to end with a byte sequence that generated modulus-value bits of zeros in the rolling hash. This method eliminates the byte-shifting issue that fixed-size chunking suffers from, but is more expensive because of the hashing.

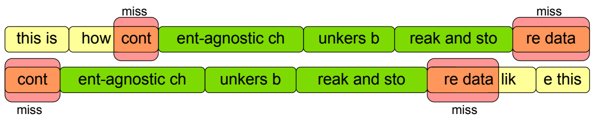


Figure 3: Content-agnostic chunking.

This method is also subject to "external fragmentation" wherein matching chunks may have a small number of bytes either before or after the chunk boundaries that are also identical, but not necessarily detected. This can be seen in Figure 3 in which the missed duplicate byte fragments are highlighted. Finally, because the chunks are variable size, this can lead to wasted storage on the physical level, for example, a 3KB chunk requires a whole block in a 4KB filesystem, wasting 1KB.

For IZO, fixed-block chunking may not provide enough deduplication, and content-aware chunking is too expensive and does not generalize well. Aligning chunk sizes with the physical disk is unnecessary, because IZO produces just a single output stream, which will be optimally stored by the underlying filesystem. Our prototype therefore uses content-agnostic chunking in order to provide maximum deduplication. Although still subject to external fragmentation, we implement an optimization (described in detail in the data format section) in which contiguous chunks can be merged into superchunks in the reconstruction metadata.

Chunk Hash Lookup

Deduplication filesystems require the fast lookup of chunk hashes to determine whether a data chunk exists in the system or not. This is one reason why these systems generally use large chunk sizes: the storage overhead of this metadata increases rapidly for smaller chunk sizes and lookup performance plummets as the data cannot be cached in memory anymore. On the other hand, smaller chunks allow the detection of more duplicates, increasing the deduplication ratio.

For the implementation of IZO it is possible to discard the in-memory hash data once the compression is completed. The main drawback of not storing this metadata in the archive is that adding additional files to the archive is not possible or computationally expensive. Without this information, new chunks cannot be directly compared to the existing chunks. However, the original hash table can be rebuilt by re-chunking and re-hashing the existing archive. Another option is to store this metadata in a separate file so that additions can be made quickly, but consumers of the archive are not burdened by the size overhead.

Not having to store the hash data reduces the final archive size. For example, a 1 GB input file deduplicated at a 512-byte average chunk size and achieving a folding factor of 2 (deduplicated to 500 MB) would produce one million 24-byte {hash, offset} tuples, accounting for 24 MB or 5% of the archive size. Not storing this metadata also allows us to use much smaller chunk sizes because we won't have to worry about "polluting" the output file with inflated hash data. However, the reconstruction metadata (the {hash, offset} tuple list) still grows with decreased chunk size. For very small chunk sizes it is worse than we would wish, due to increased segmentation of the data. Figure 4 shows the amount of metadata from our deep freeze experiment for a single VM image (37.3 GB) as input. The upper line shows the expected growth of a factor of two. The reconstruction metadata (lower line) roughly follows this line until a chunk size of around 512 bytes. At 256 byte the metadata grows even faster, because increased segmentation requires more reconstruction information.

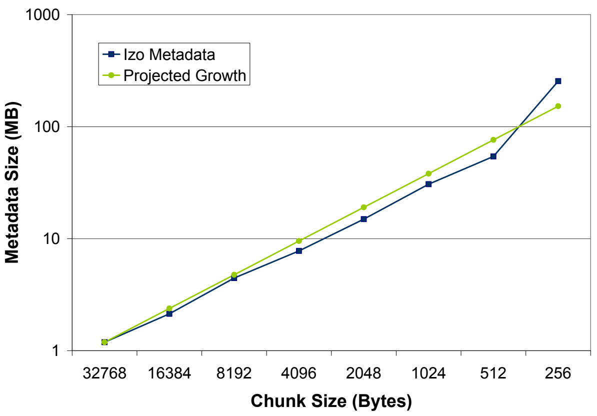


Figure 4: Effects of chunk size on reconstruction metadata.

Chunk Hash Collisions

By relying on the chunk hash to guarantee chunk matches, IZO is able to operate on sequential input streams. This is because IZO does not need to seek back in the stream to verify that chunks are identical. Unlike rzip, which uses a 4-byte hash as a hint that the indicated data may be identical, IZO uses a 16-byte MD5 hash to probabilistically guarantee that the indicated chunks are the same. A detailed account of the probability of a hash collision in a data deduplication system is provided for EMC's Centera [15].

The problem of a hash collision occurring between two chunks in a deduplication system simplifies to the birthday paradox (see Appendix). In the case of IZO, the namespace n is the number of unique 128-bit hashes (2^{128}), while the number of things to be named c is the number of chunks in the archive. With an archive of 1 TB and an average chunk size of 1 KB, c is 2^{30}. For large c's, the collision formula simplifies: $c^2/(2n - c)$. For an archive of 1 TB using an average chunk size of 1 KB, the probability of collision is 1 in 2^{68}. For a 1 PB archive, the probability is 1 in 2^{48}. For comparison, the non-recoverable Bit Error Rate (BER) of an ATA hard drive is 1 in 10^{15} (2^{50}) [15].

The IZO Data Format

The design of the IZO data format was driven by the desire to pack data and metadata as tightly as possible into the resulting archive. Instead of storing a list of chunk identifiers for each file, we use offsets and lengths within the archive. In particular, if a sequence of chunks from the original file is new to the system, then it is being stored as the same sequence into the output file, requiring only one {offset, length} tuple. Because we keep track of all individual segments during compression, it is still possible to match subsequences within this data.

The hash metadata is not being persistently stored. The only metadata that is required is the path and filename for each file in the archive, along with file-specific information such as ctime, mtime, etc., and a list of {offset, length} tuples to reconstruct the constituent files.

It should be noted that we create an IZO archive in segments. A segment remains in memory until it is completely prepared, and is then streamed out. This allows us to perform random updates within the segment (such as to the segment length) without the need to seek on disk. A typical segment size is 8 MB. File data can span segments, but each new file to the archive begins a new segment. Reconstruction references in a given segment are always relative to the beginning of the segment in which they sit. These references always refer either to chunks in the same segment (positive values), or to chunks in a previous segment (negative values), but never to chunks in a future segment. This property was designed into our data format to provide for appendability of archives. Just as with tar, one can concatenate two IZO archives to form another valid IZO archive.

We demonstrate the IZO file format with an example of processing two input files shown in Figure 5. The processing steps are illustrated in Figure 6. The first four bytes of the IZO archive store a 32-bit segment length. The segment length is not known until the entire segment is populated. Immediately following the segment length is the reconstruction metadata. The reconstruction metadata contains information such as the file name and a list of {offset, length} tuples to reconstruct the file. Following the reconstruction metadata are the actual data chunks from the first input file (see Figure 6(a)). As unique chunks are encountered, their hashes and offsets are stored in the in-memory hash table and the actual chunk data is written to the current segment in the archive.

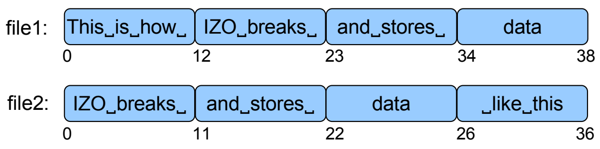


Figure 5: Example input files for IZO.

Once the first file has been processed, the lengths of both the reconstruction metadata and data are known, and we can update the segment length field and stream out our segment (see Figure (b)). In this example, the reconstruction metadata is placed at offset 4, and contains only one {offset, length} tuple, because the first file did not provide any duplicate chunks and was therefore written as-is. The file1 reconstruction metadata specifies that to recreate file1, read 38 bytes from offset 22 relative to the beginning of the segment (0+22=22).

Now, IZO processes the second file. The first chunk matches the chunk stored at offset 34 with length 11. The next chunk matches at offset 45 with length 11. Because these two chunks are in sequence, IZO uses the superblock optimization and only modifies the length of the matching data section from 11 to 22. The third chunk is also a contiguous duplicate of four bytes at offset 56, resulting in another update to the length to 26. The last chunk is new to the system and is added to the archive, and its hash is added to the in-memory hash table. Finally, the metadata length for the second file is known and the segment is finalized and streamed out. Looking at the reconstruction metadata for file2, it specifies reconstruction of the file by reading 26 bytes from an offset 26-bytes before the current segment (60-26=34) and then 10 bytes from an offset 34 bytes after the beginning of the current segment (60+34=94) (see Figure 6(c)).

Experiments

We performed our experiments on a 2-way 2.0 GHz 64-bit Intel Pentium Xeon server with 2 GBs of DDR RAM with four 500 GB 7200 RPM UDMA/133 SATA disks of storage. This appliance hosts a SUSE SLES64 version 10.1 operating system which is built upon Glibc 2.4-31.30 and GCC 4.1.2. The operating system and all executable were installed on the first disk, while the remaining three disks were used for data storage.

Tuning IZO

We first examined the effect of the block size that IZO would use for the deep freeze experiment. We tried average variable-sized blocks ranging from 32 KB in size down to 256 bytes. We ran each block size against first a single 40 GB virtual machine image, then added the next, and so on up to all 11 images.

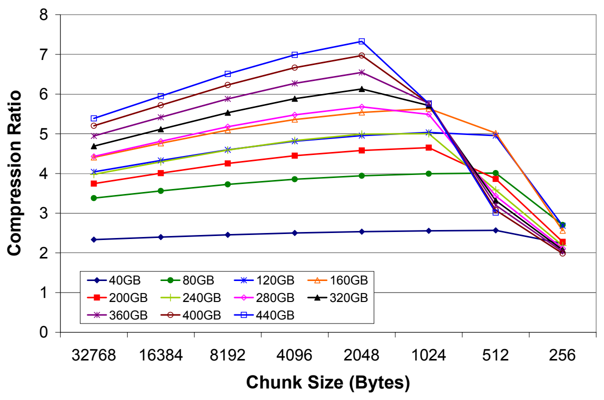
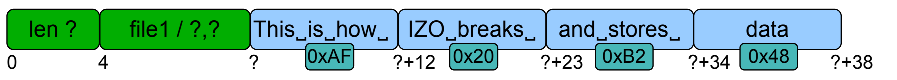
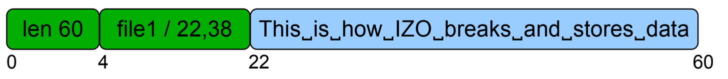


Figure 7: Effects of chunk size on combined compression ratio.

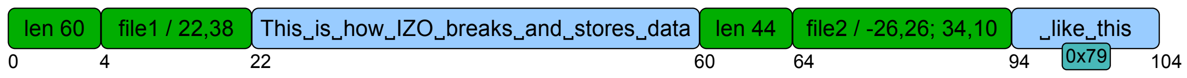
Figure 7 shows the overall fold+compress ratios for these image sets at each block size. The best overall compression is garnered from 2 KB block sizes. In our experience with this tool, generally we find that with large datasets, in the hundreds of GBs for



(a) Step 1: Filling the First Segment with Metadata and Unique Data Chunks



(b) Step 2: Determining Superblocks, Finalizing Offsets, Streaming Segment Out



(c) Step 3: Working on and Finishing Segment for file2

Figure 6: Output creation.

example, block sizes in the range of 2 KB to 8 KB yield a good folding factor. When dataset size is smaller than 10 GB or so, the block size can be reduced, with 128 bytes as the lower limit.

Smaller blocks can have a negative impact on the folding factor for two reasons. First, smaller blocks can greatly increase the amount of metadata required to store references to previously-seen blocks. Second, smaller blocks more quickly fill the in-memory hash table we use to store references to all of the blocks in the system. With a large dataset, the hash table becomes full, and we must start evicting block references, becoming effectively blind to a number of previously seen blocks. For these reasons, in our subsequent experiments we use a 2KB block size for IZO.

Deep Freeze

The deep freeze experiment uses a number of similar but not identical machines and tries to reduce the amount of storage the set of images will require. For this experiment we look at large-window deduplication (IZO), large-window compression (lrzip) and small-window compression (gzip).

The dataset for this experiment was eleven 40 GB Windows XP drive images of machines that started with a similar install but then diverged for a year in a community after-school program.

Table 1 shows the archive size and time required for gzip, IZO and lrzip. Using the large-window deduplication tuned to operate at a 2KB block size on a typical XP system results in a folding factor of 1.92 for just a single machine. This compares to the gzip-only compression ratio of 1.40. These two techniques are more or less orthogonal and can therefore be combined. Applying gzip to the IZO-archive provides an overall compression ratio of 2.54.

Timewise it is clear that gzip is fairly intensive – running it on the smaller IZO file instead of on the original data results in time savings of 22%. For a single image we see that deep freezing (fold+compress) takes 43 minutes, and will result in an overall storage savings of 61%. This compares to gzip alone on the original data, which takes 55 minutes and yields a savings of only 28%. lrzip does not find much duplication, and hands a large output to gzip, resulting in a 97 minute effort to save a combined 36%. So, not only does the fold+compress technique yield the best overall compression ratio, it is also much faster than the next fastest technique.

Where deep freezing really shines is when we examine freezing more than one image at the same time. In these cases IZO can exploit cross-image similarities to garner much higher folding factors. For five images, IZO+gzip affects a 78% storage reduction in 144 minutes. Applying lrzip+gzip yields a data reduction of 36% and takes 527 minutes, longer than an 8-hour working day.

In the case of 11 images IZO achieves a folding factor of 5.47, and gzip is able to push it 7.32, a storage savings of 86%. This takes only 262 minutes, more than 2 times faster than running lrzip+gzip on only 5 images. We do not provide measurements for lrzip on eleven images because of the long run-time this would require.

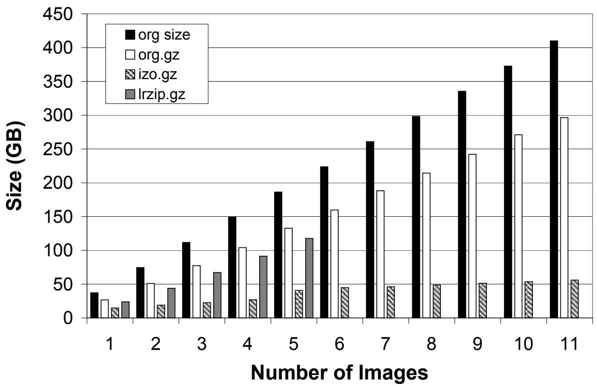


Figure 8: Compression of multiple Windows XP images.

Figure 8 illustrates these finding across all of the machine images. While the original size (in black) and gzip size (in white) continue to increase rapidly with the number of images, IZO+gzip (in stripes) grows much more slowly. lrzip+gzip does slightly better than gzip alone, but does not approach the IZO+gzip ratios.

The reason that IZO is able to achieve such high folding factors on this data, and lrzip is not, has to do with the effective window size of the two large-window compression tools. In order to take advantage of any inter-image duplication at all, a tool must have a window at least as large as a single image, in this case, roughly 40 GB. To eliminate duplication across all 11 images, this window must encompass every image, over 400 GB in our case. lrzip has an effective window of around 900 MB, and is therefore unable to "see" and eliminate the inter-image duplication. IZO, with a 2 KB average block size, has an effective window in the TB range, and is therefore able to deduplicate across the entire set of 11 images.

Number	Size (GB)						Time (minutes)				
Images	orig	gzip	izo	izo.gz	lrzip	lrzip.gz	gzip	izo	izo.gz	lrzip	lrzip.gz
1	37.3	26.7	19.4	14.7	30.6	23.9	55	14	43	49	97
5	186.4	132.8	51.7	40.7	151.4	117.7	282	67	144	269	527
11	410.0	296.6	74.9	56.0	–	–	556	153	262	–	–

Table 1: Deep freeze scenario compression numbers using 2KB variable chunks.

VM Distribution

This scenario simulates the case when a company needs to update a set of standard images, which may be distributed globally to different remote offices or branches. Traditionally, images are either simply gzip compressed, or a delta is computed between the two images, and the delta is compressed. In the case of creating deltas, the original image is assumed to be at the destination already. We investigate the use of IZO to produce a folded delta that can be applied to the original image to produce an updated image.

For this scenario we used seven "clean installs" of RedHat Enterprise Linux Workstation version release 4 to simulate incremental upgrades of a base system that might be used in a virtual machine environment. We have one install for each update (u1-u6) plus the original release. Each install is done on a 10 GB partition, the majority (roughly 7.5 GB) of which is sparse.

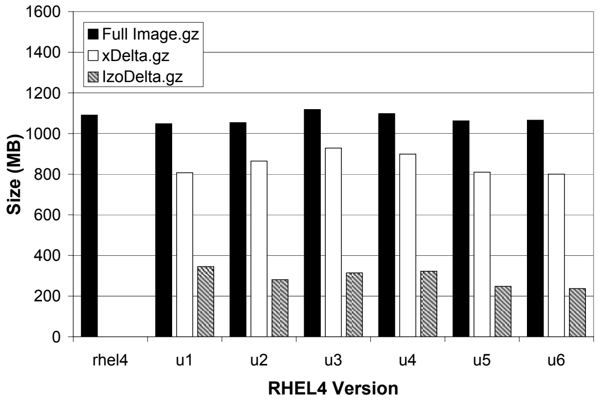


Figure 9: Binary deltas between incremental VM versions.

Figure 9 shows the smallest deltas that can be achieved between successive updates to the previous image. u1 represents the delta to the base, u2 represents the delta to u1, and so on. As illustrated, the overall image size after gzip compression is just over a gigabyte. Already this is a significant savings due primarily to the sparseness of the 10 GB image. When xdelta is applied to the images first, and then gzip is applied, the data size is reduced to an average size of 851 MB, or an additional 21% on average. IZO, however, provides an additional 73% storage savings over gzip alone. This reduces the amount of data to an average of 290 MB that is needed to transform one 10 GB image into its subsequent version – just 3% of the entire image size, and roughly three times smaller than xdelta+gzip .

Backup

In this experiment, we look at the application of large-window compression techniques to weekly backups of a single user's machine. The goal is to provide a traditional incremental backup of the machine without requiring any end user action, nor the ability to install clients on the end user machine.

The backups in question are full filesystem backups of a machine running Windows XP, including all data, application, and system files. The machine is one of the authors' actual workstation, used every day during the work week for email, office applications, and software development.

The backup schedule used by this machine was very simple. It completes the first backup of roughly 23 GB on Sept 8, and then performs a full weekly backup for eight weeks. This scenario is analogous to the scenario described in the introduction, wherein a user may be running an operating system in a VM without palatable means of backup from within the system. By running large-window compression over the entire system periodically, the system VM administrator is able to effectively perform multiple full system backups in an operating-system agnostic way, while achieving smart incremental backup space efficiency.

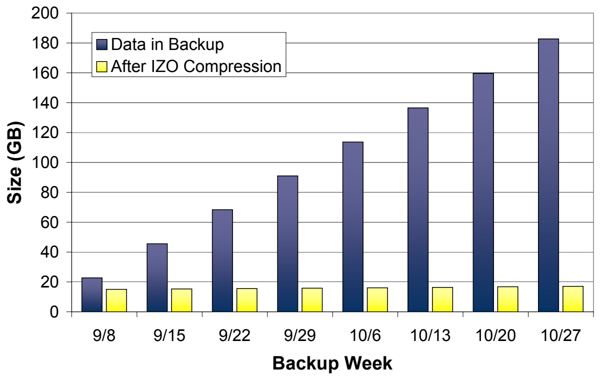


Figure 10: Compression of weekly system backups.

Based on our previous experiments, we chose to use a content-agnostic chunker based on Rabin fingerprinting to produce blocks of variable size of around 2 KB for this test. The results we observed from these experiments were very encouraging for this method of backup (see Figure 10). The first observation was that even for the very first backup, the size of the data was reduced from 23 GB to 15 GB (note, for these experiments we have not gziped the resulting IZO files). Adding successive backups (using the append functionality of IZO) consistently showed that very little additional data was added to the system. In fact, during the two months of use of this machine, the user added just less than 1 GB of unique data. Including all data and metadata, IZO is able to store all 8 full filesystem backups in 17 GB, which is smaller than the original 23 GB image itself.

It is interesting to note where the growth in the IZO archive actually occurs (see Figure 11). The IZO archive grows from 15 GB to store a single image to 17 GB to store all 8 images. However, the growth of the actual data stored in the archive grows even more slowly. The predominant growth of the data can be attributed to the reconstruction metadata used to point back to the data in previous backups. We are currently investigating ways to compress the metadata itself, by looking at similarities in the metadata between different backups.

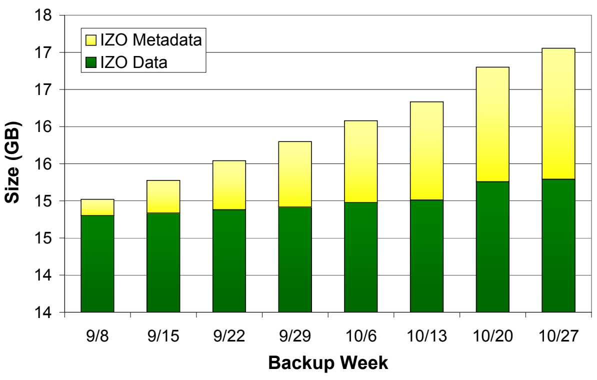


Figure 11: Increase in the amount of meta data stored in IZO for weekly system backups.

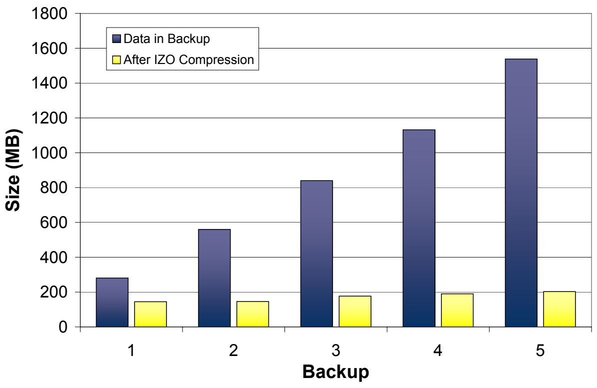


Figure 12: Compression of a collection of OS/2 VM images.

These numbers are for a single machine. The advantages to performing multiple machine backups into the same IZO archive should be comparable to the advantages seen in the deep freeze experiments.

Next, we studied as a proof of concept the use of deduplication to enable backup of a VM running a somewhat uncommon OS. The idea is to verify that such machines can be backed up, even if there is no client or filesystem support for them. We created a virtual machine using Microsoft VirtualPC and OS/2 Warp 4. We then ran the machine and performed simple filesystem modifications in OS/2 by adding data to the disk or removing files from disk. Once in a while, we stopped the machine and created a "backup" by copying the VM image. It is clear that this approach does not fully capture the change that would occur during real-world use of an OS/2 installation, but it provides some insight in whether IZO can achieve incremental backup space efficiency for OS/2 as well.

Figure 12 shows the results of compressing up to five backups. The initial VM image is only 281 MB, and IZO is able to compress the image to 145 MB already. Adding additional backups slowly increases the overall archive size to 203 MB. During the experiment, we added a total of 93 MB of new data and removed 32 MB of data from the OS/2 filesystem. Our backup #5 encodes all five backups and only requires 58 MB of additional space over the base image, clearly providing incremental backup functionality.

Conclusions and Future Work

Large-window compression is a fairly new technology that has many applications to the domain of virtual machine administration. Initial experiments show storage savings of up to 86% for some scenarios. These compression savings can enable new approaches to tasks such as machine image distribution and weekly virtual machine backups.

Our IZO compression prototype uses algorithms that originate from online deduplication filesystems. We find that these methods are well-suited for offline large-window compression as well. They require relatively little memory and little time when processing large data sets. While IZO removes global redundancies, it relies on small-window tools such as gzip to compress the remaining data. We observe that these two approaches are orthogonal to each other – the combined compression ratio is very close to the product of the individual compression ratios. An additional benefit is that the combined processing time of IZO+gzip is generally less than the time it takes gzip to process the original input file alone. This is explained by the fact that IZO quickly removes global redundancies and gzip is left to compress a much smaller amount of data afterwards.

We are quite excited about the results of our initial experimental evaluation, but we also found several shortcomings of our prototype that we plan to address in our ongoing research.

First, we are concerned with the rapid growth of metadata in the backup scenario in Figure 11. Because the incremental changes between backups are small, we expect that the metadata between the two versions should be very similar as well. One way of confirming this would be to apply IZO to the archive yet again, but we'd rather store the metadata more efficiently in the first place.

Another area of interest is to determine optimal chunk size automatically. We currently require larger chunk sizes for larger input sets, because IZO eventually cannot keep all chunk hashes in memory, reducing the overall fold factor. It may be possible to begin compression with smaller chunks, and then increasing chunk size over time. A different approach that we would like to investigate is the encoding of superchunks – sequences of chunks that occur repeatedly because the underlying data is always contiguous as well (e.g., a large music file that is chunked into a sequence of hundreds of chunks). Lastly, we may be able to modify our hash eviction strategy to yield better results or lower memory requirements.

Finally, we are planning to add some convenience features to IZO, such as integration with gzip and bzip2, to increase the ease-of-use even further.

Acknowledgements

We wish to thank our shepherd, Derek Balling, for his guidance and comments, which improved our paper tremendously. Andrew Tridgell provided valuable insights into the algorithms and implementation of rzip. We would like to thank Simon Tsegay of the San Jose Eastside Boys & Girls club for the machine images used in the deep freeze experiments. Lastly, we are grateful to our anonymous reviewers for their comments and hope that the final version of our paper addresses their feedback.

Author Biographies

Mark A. Smith finished his graduate work at the University of California at San Diego with an M.S.-C.S., where he had worked in the Cardiac Mechanics Research Group on computer-aided visualization of heart mechanics. After graduation, he joined the Systems Research team at the IBM Almaden Research Center. Current areas of focus include filesystems, data deduplication, distributed security, and archival.

Jan Pieper received a Diploma Engineering Degree in Computer Science from the University of Applied Sciences in Hamburg, Germany in 2000, where he designed a web-based content management system. He joined IBM Almaden Research after graduation and has worked on distributed systems, web-scale data mining, file systems and social networking applications. He is currently also a part-time graduate student at UC Santa Cruz.

Daniel Gruhl is a research staff member at the IBM Almaden Research Center. He earned his Ph.D. in Electrical Engineering from the Massachusetts Institute of Technology in 2000, with thesis work on distributed text analytics systems. His interests include steganography (visual, audio, text and database), machine understanding, user modeling, distributed systems and very large scale text analytics.

Lucas Villa Real received a B.Ch.-C.S. in 2005 at the Unisinos University in Brazil, where he had worked with parallel computing and general purpose computation on GPUs. After graduation, he started his M.S.-C.S. program at the Polytechnic School at the University of Sao Paulo and joined their digital TV lab, focusing his research on demultiplexers and filesystems. In the meantime, he spent a few months doing an internship at the IBM Almaden Research Center, also working in filesystems.

Bibliography

[1] *Hyperfactor: A Breakthrough in Data Reduction Technology*, Technical report.

[2] Data Domain SISL, *Scalability Architecture*, Technical report, May, 2007.

[3] *Scalable, Highly Automated Virtual Tape Library Technology Reduces the Cost of Storing, Managing and Recovering Data*, Technical report, January, 2007.

[4] Tridgell, J. M. Andrew, *xdelta*, 2008, http://xdelta.org/ .

[5] Barham, P., B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pp. 164-177, 2003.

[6] Clark, B., T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. N. Matthews, "Xen and the Art of Repeated Research," *USENIX Annual Technical Conference 2004, FREENIX Track*, pp. 135-144, 2004.

[7] Gilmore. J., *Gnu tar*, 2008, http://www.gnu.org/software/tar/ .

[8] Hitz, D., J. Lau, and M. Malcolm, "File System Design for an NFS File Server Appliance," *1994 Winter USENIX*, pp. 235-246, 1994.

[9] Gailly, J. and M. Adler, *gzip*, 2004, http://www.gzip.org/ .

[10] Kolivas, C., *lrzip*, 2008, http://ck.kolivas.org/apps/lrzip/README .

[11] Kulkarni, P., F. Douglis, J. LaVoie, and J. Tracey, "Redundancy Elimination Within Large Collections of Files," *USENIX 2004 Annual Technical Conference*, pp. 59-72, 2004.

[12] May, B., *NetApp A-SIS Deduplication Deployment and Implementation Guide*, Technical Report TR-3505, 2007.

[13] Park, K., S. Ihm, M. Bowman, and V. S. Pai, "Supporting Practical Content-Addressable Caching with czip Compression," *USENIX 2007 Annual Technical Conference*, pp. 185-198, USENIX, 2007.

[14] Policroniades, C. and I. Pratt, "Alternatives for Detecting Redundancy in Storage Systems Data," *Proceedings of the USENIX 2004 Annual Technical Conference*, pp. 73-86, 2004.

[15] Primmer, R. and C. D. Halluin, *Collision and Preimage Resistance of the Centera Content Address*, Technical report, 2005.

[16] Quinlan, S. and S. Dorward, "Venti: A New Approach to Archival Storage," *FAST 2002 Conference on File and Storage Technologies*, pp. 89-102, 2002.

[17] Rabin, M. O., *Fingerprinting by Random Polynomicals*, Technical Report TR-15-81, 1981.

[18] Reed, B., M. A. Smith, and D. Diklic, "Security Considerations When Designing a Distributed File System Using Object Storage Devices," *IEEE Security in Storage Workshop*, pp. 24-34, 2002.

[19] Rosenblum, M. and T. Garfinkel, "Virtual Machine Monitors: Current Technology and Future Trends," *Computer*, Vol. 38, Num. 5, pp. 39-47, 2005.

[20] Seward, J., *bzip2*, 2007, http://www.bzip.org/ .

[21] Smith, J. E. and R. Nair, "The Architecture of Virtual Machines," *Computer*, Vol. 38, Num. 5, pp. 32-38, 2005.

[22] Sugerman, J., G. Venkitachalam, and B.-H. Lim, "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor," *USENIX 2001 Annual Technical Conference, General Track*, pp. 1-14, 2001.

[23] Tang, J. C., C. Drews, M. Smith, F. Wu, A. Sue, and T. Lau, "Exploring Patterns of Social Commonality Among File Directories at Work," *CHI '07: Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, pp. 951-960, 2007.

[24] Tridgell, A., *Efficient Algorithms for Sorting and Synchronization*. Ph.D. thesis, The Australian National University, 1999.

[25] Trigell, A., *rzip*, 2004, http://rzip.samba.org/ .

[26] Villanueva, B. and B. Cook, "Providing Students 24/7 Virtual Access and Hands-On Training Using VMware GSX Server," *SIGUCCS '05: Proceedings of the 33rd Annual ACM SIGUCCS Conference on User Services*, pp. 421-425, 2005.

[27] Waldspurger, C. A., "Memory Resource Management in VMware ESX Server," *OSDI '02: Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, pp. 181-194, 2002.

[28] Whitaker, A., R. S. Cox, M. Shaw, and S. D. Gribble, "Rethinking the Design of Virtual Machine Monitors," *Computer*, Vol. 38, Num. 5, pp. 57-62, 2005.

[29] You, L. and C. Karamanolis, "Evaluation of Efficient Archival Storage Techniques," *21st IEEE/12th NASA Goddard Conference on Mass Storage Systems and Technologies*, pp. 227-232, 2004.

[30] You, L. L., K. T. Pollack, and D. D. Long, "Deep Store: An Archival Storage System Architecture," *21st International Conference on Data Engineering (ICDE'05)*, pp. 804-815, 2005.

Appendix: Birthday Paradox

The birthday paradox asks how many people need to be in a room together before the probability of two of them sharing a birthday is 50%. The formula to describe the probability is well known: $1 - n!/(n^c(n-c)!)$ where n is the namespace (365 days) and c are the number of things (people) with names (birthdays). In the case of the birthday paradox, there are 365 days, and by setting the probability to 50%, we arrive at the answer that it requires 23 people in a room before the likelihood of two of them sharing a birthday is 50%.