# Using Deduplicating Storage for Efficient Disk Image Deployment

Xing Lin Mike Hibler Eric Eide Robert Ricci
University of Utah, School of Computing
Salt Lake City, UT USA
{xinglin, hibler, eeide, ricci}@cs.utah.edu

## ABSTRACT

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

**Categories and Subject Descriptors** H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*performance evaluation*; C.2.2 [**Computer-Communication Networks**]: Network Protocols—*applications*

**General Terms** Design, Measurement, Performance

**Keywords** deduplication; image deployment

## 1. INTRODUCTION

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

IaaS facilities generally make a large collection of disk images available to their users; these images may contain a variety of operating systems and sets of standard software. In addition, most allow users to create disk images of their own. The Amazon EC2 Web site [1], for example, lists more than 37,000 public Amazon Machine Images. The Utah Emulab testbed (which we operate) manages more than 1,000 images—public and private—for its users [2], and the DETER testbed manages more than 400 [26]. These catalogs represent large amounts of data (21 TB for Emulab), and moreover, they grow steadily over time [2]. A facility's operators and users continually create new images, while old images need to be retained to support existing users or the reproducibility of previous results. It becomes important to store these large numbers of disk images efficiently.

*Data deduplication* has been shown to be an efficient way to save disk space for storing disk images. In a deduplicating storage system, large pieces of data—e.g., disk images—are divided into units, and every unique unit is stored exactly once. If two images have a unit in common, they share the single copy of that unit. Because disk images are typically derived from other images by making small changes, there is significant duplication between an image and its "children." Previous work has shown that deduplication can greatly reduce the storage requirements of disk-image catalogs across virtual machines [8, 9, 27] and across machines in a commercial environment [13].

To support efficient and scalable image deployment, systems like Emulab have designed sophisticated mechanisms. Frisbee [7], used in Emulab, includes the following features. First, image data is compressed before it is stored, and it is transferred in compressed format during image deployment. Second, Frisbee utilizes filesystem information to skip unallocated disk sectors. This reduces the amount of data to store during image creation. More importantly, less data needs to be transferred across the network and fewer disk writes are needed during image installation. Third, the image file created by Frisbee is composed of independently installable chunks. Each chunk can be requested and installed independently. Last, Frisbee uses pipelining so that chunks at different stages in the pipeline can be processed in parallel. To get the highest possible performance, the pipeline is designed so that the last stage (writing image data to disk) is the bottleneck. This ensures that Frisbee can install the image at the full speed of the disk. To be scalable, it implements its own application-level multicast protocol.

This paper presents Venti-Frisbee (*VF*), our new image-deployment system that utilizes a deduplicating storage system to reduce the amount of physical storage while maintaining Frisbee's high performance in image deployment. We use Venti [20] as our deduplicating storage system, but any similar system should work.

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

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

- Frisbee skips unallocated sectors and concatenates allocated sectors. This implies that, in the face of sector allocation and deallocation, the positions of sectors in the output image data will not remain the same. Fixed-size chunking may thus become less effective. We propose a new chunking algorithm, called Aligned Fixed-size Chunking (AFC). It utilizes disk offsets to pad the start and the end of each contiguous allocated sector range to ensure full blocks from each allocated range.

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

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

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

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

## 2. FOUNDATION: FRISBEE AND VENTI

VF is built on top of two existing systems: Frisbee [7], a scalable, high-performance disk deployment system, and Venti [20], a deduplicating storage system. In its original design, Frisbee stores disk images as files in a regular filesystem on the Frisbee server; VF replaces this back-end storage with Venti. While this change is conceptually simple, Frisbee's design for efficient image deployment have four implications for VF. We discuss each in turn.

### 2.1 Frisbee

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

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

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

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

**Compress image data.** The data read from allocated disk sectors is compressed as it is added to the image file. As with filesystem-awareness, data compression reduces storage requirements and network bandwidth during image distribution. The additional decompression step added during image installation does not introduce a significant overhead: decompressing image data can be done twice as fast as writing decompressed image data to disk and the two tasks can be pipelined. On the other hand, doing compression for image data is significantly slower than any stage in the image deployment pipeline.

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

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

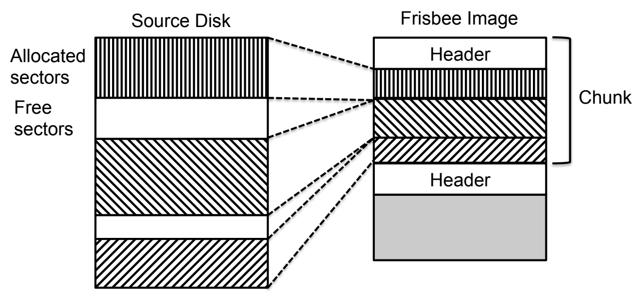

Figure 1: Frisbee identifies allocated disk sectors, compresses them, and concatenates them into 1 MB chunks.

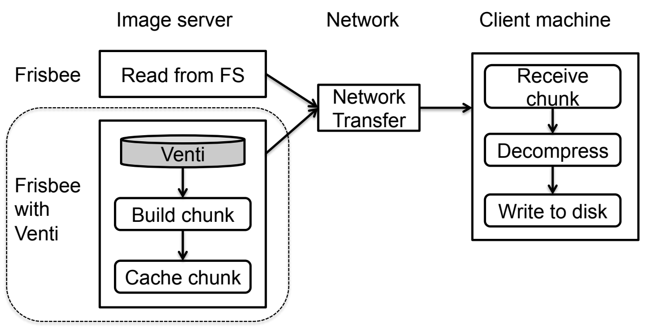

Figure 2: Frisbee's two-level pipelining design and the design of VF. In the new design of VF, the first stage (Read from FS) is replaced with chunk construction from Venti.

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

**Pipelining.** The design of independently installable image chunks also enables pipelining: the installation of a chunk can be pipelined with the transmission and decompression of other chunks. In Frisbee, there are two levels of pipelining, shown in Figure 2. The image-deployment pipeline has three stages: image data is *read* from the Frisbee server's disks, *transmitted* on the network, and *installed* on the target disk. Image installation at the client machine is further decomposed into three pipeline stages: *receiving* chunks from the network, *decompressing* the data in those chunks, and *writing* the decompressed data to the target disk. These stages are handled by separate threads so that they can proceed in parallel. The pipeline is designed such that the last stage (writing to disk) is the bottleneck of the pipeline overall. This results in a highly efficient disk-deployment system that succeeds in writing at the full speed at the target disk during image installation.

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

◇ *Implication 4:* To get performance comparable to the original Frisbee, the chunk-construction stage in VF must be faster than the slowest stage (writing to disk) in the image-deployment pipeline.

## 2.2 Venti

Our second building block is Venti, a deduplicating storage system by Quinlan and Dorward [20], with enhancements from the Foundation [23] system. It has been used for daily archival snapshots of filesystems in the Plan 9 operating system. We use the Venti archival storage server, which provides a large data repository and exposes a simple object interface for clients to read and write variable-size blocks. A block can be any size from 512 B–56 KB. When a block is written to Venti, it returns a handle to retrieve that block. The handle includes the fingerprint (the SHA–1 hash of its content) for that block and it uniquely identifies a data block within the storage system. Venti skips writes of duplicate blocks and stores only unique ones. When compression is turned on, each unique block is compressed and then written to disk.

Venti is publicly available and it served our purposes in developing the VF prototype. As long as it is "fast enough" to not be a new bottleneck, VF should get similar high performance as the original Frisbee. The lessons we learned in this paper are independent of the particular deduplicating system used. Other deduplicating systems, including commercial ones such as the EMC Data Domain Deduplication Storage System [31], could be used in place of Venti.

## 3. DESIGN AND IMPLEMENTATION

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

### 3.1 Compression

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

Figure 3a presents the most straightforward approach to integrating Frisbee and Venti: storing compressed Frisbee images directly into Venti, by first partitioning them into blocks and then storing those blocks in Venti. This approach has the advantage that, at image-deployment time, chunk construction requires only concatenating the data retrieved from Venti to reform chunks. However, it requires that deduplication be done on compressed data. This yields a low deduplication ratio since compressors have already identified and replaced repeated strings with more compact encodings and the resulting compressed data has very little duplication. Moreover, even small changes to a disk (e.g., the allocation of a single sector) can produce a dramatically different compressed image compared to the original, leading to poor deduplication across multiple disk images. Overall, this approach has high image-deployment performance but little savings from deduplication.

Figure 3b shows another option, which improves deduplication. It uses the Frisbee image-creation tool to identify allocated ranges on the disk, breaks uncompressed data from these ranges into blocks, and stores them in Venti. Venti fingerprints the blocks and then compresses and stores them, one copy of each unique block. This approach achieves efficient deduplication, but it incurs major overhead on the image-deployment path: data must now be compressed after retrieval and before sending it on the wire. This scheme meets our storage-saving goals, but falls short on high-performance image deployment.

A third approach, shown in Figure 3c, performs compression

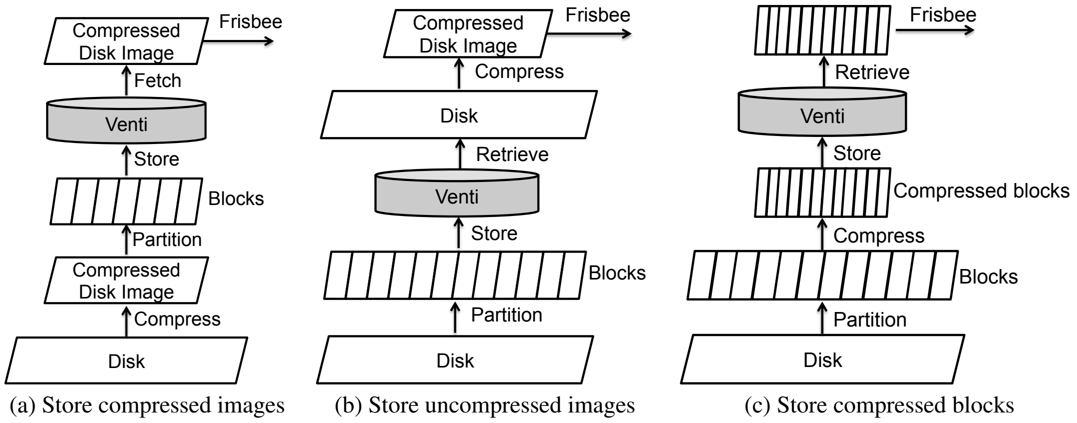

Figure 3: Three possible compression schemes.

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

## 3.2 Chunking

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

The first is the type of chunking algorithm to use. There are two types of chunking algorithms: fixed-size [20] or variable-size [4, 14, 31].[1] Fixed-size chunking determines block boundaries based on data offsets while variable-size chunking is based on data content and is more resistant to content shifts from data insertions and deletions. Fixed-size chunking is straightforward and requires low computational overhead while variable-size chunking has considerably higher computational overhead.

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

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

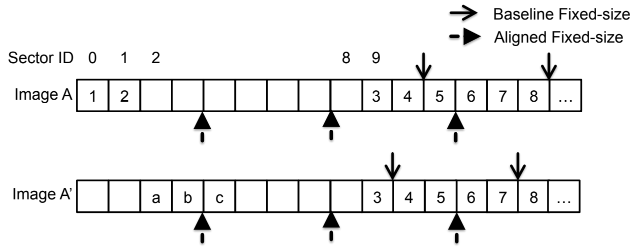

Figure 4: Comparison between baseline fixed-size chunking and aligned fixed-size chunking (AFC).

Figure 4 shows a comparison between using conventional ("baseline") fixed-size chunking on a Frisbee stream and AFC. Image A is the base image and we create a new image A' by freeing the first two sectors (1 and 2) and allocating the next three ("a", "b", and "c"). Assume we are partitioning this image into fixed-size blocks of four sectors each. Frisbee will concatenate the second allocated range starting from the ninth sector with the first allocated range. Thus, in the baseline fixed-size chunking, for Image A, the first block will contain [1,2,3,4] and the second block will contain [5,6,7,8]. However, for image A', the first block will contain [a,b,c,3] and the second block will contain [4,5,6,7]. Sector allocations and deallocations cause block boundary shifts resulting in no deduplication between the images.

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

[1]We use "chunks" for Frisbee chunks and "blocks" for deduplication units. The term "chunking" is borrowed from the deduplication literature, to denote the process of partitioning data into deduplication units.

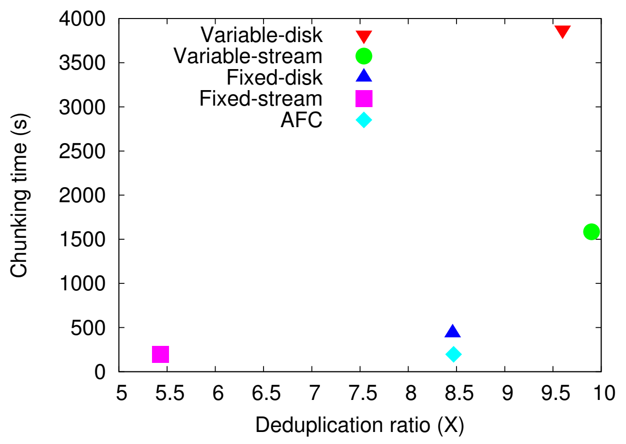

Figure 5: Comparison of different chunking options in terms of deduplication and runtime.

following blocks identical.

We performed an experiment measuring the deduplication ratio (defined as $\frac{original\_size}{deduplicated\_size}$) and runtime for these five chunking options, over the 430 Linux images used in our study. We used fs-hasher [5], a chunking and hash tool developed by Stony Brook University, with hash calculations disabled when measuring the chunking time. The result is presented in Figure 5. Here we can see that the variable-size chunking options (Variable-disk and Variable-stream) yield the best deduplication but perform poorly, with chunking times about 10× longer than their corresponding fixed-size alternatives (Fixed-disk and Fixed-stream). We also observe that chunking at the disk level (Variable-disk and Fixed-disk) increases chunking time by more than 2× compared with chunking at the stream level (Variable-stream and Fixed-stream). Finally we note that AFC has performance similar to fixed-size chunking at the stream level while achieving the same level of deduplication as fixed-size chunking at the disk level.

## 3.3 Block Size

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

The possible block sizes for Venti (and therefore VF) range from 512 B to 56 KB. Because we are dealing primarily with filesystem data in our images, we consider the lower bound to be 4 KB, which is the minimum block size used in many OS filesystem implementations. In Section 4.3, we compare five candidate block sizes from the range 4 KB–48 KB, and find 32 KB to work best in practice.

## 3.4 Frisbee Chunk Construction

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

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

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

## 3.5 Design Summary

The new image-deployment pipeline of the VF system is shown in the bottom half of Figure 2. Though similar to the original pipeline, the whole system incorporates four main considerations to make the new system as efficient as the original one for image deployment while also improving storage efficiency significantly. The considerations include the informed choice of the deduplication block size, careful alignment of block boundaries, precompression of data blocks, and precomputation of chunk header metadata.

## 4. EVALUATION

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

Following that, our evaluation of VF is presented in three parts. The first presents an empirical analysis of the three alternatives for performing compression and validates our choice for VF. The second describes the experiments we performed to measure storage savings and the image-construction time as a function of deduplication block size (mentioned in Section 3.3). The third compares our VF system against the standard Emulab Frisbee implementation (hereafter referred to as "baseline Frisbee"), measuring both their storage demands and their image-deployment performance.

All experiments were performed on the Utah Emulab testbed [28]. The infrastructure for our evaluation consists of one server machine, acting as both a Venti archival storage system and a Frisbee image server, and 20 client machines all connected via dedicated 1 Gbps switched Ethernet. All machines are Dell PowerEdge R710s: each machine has a single quad-core 2.4 GHz 64-bit Xeon processor, 12 GB RAM, and two 250 GB, 7200 RPM Seagate SATA disks each capable of sustained sequential read and write throughput of up to 110 MB/s. The server has one additional 1.5 TB, 7200 RPM Western Digital Caviar Black SATA disk hosting the Venti repository. This disk can perform sequential reads and writes at rates up to 150 MB/s. All machines run a 64-bit version of the Ubuntu 10.04 operating system.

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

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

Table 1: Average throughput rate and execution time of each stage in the baseline Frisbee pipeline. Throughput values are measured relative to the uncompressed data, except for network transfer, which reflects compressed data (with uncompressed rate in parentheses).

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

## 4.1 Frisbee Pipeline Measurements

One thesis of our work is that extracting an image from Venti and constructing a Frisbee image needs to be fast, but ultimately just "fast enough" to not be the bottleneck for image deployment. To support this, we empirically measured the stages of the Frisbee pipeline. The results are shown in Table 1.[2] The last three lines show the stages of the image-deployment pipeline. (The first, compression, is performed at image-creation time.) The network transfer rate of 54 MB/s appears to be the bottleneck, but this is a compressed data rate. The effective (uncompressed) data rate delivered to subsequent stages is actually 165 MB/s.

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

## 4.2 The Impact of Compression

In Section 3.1 we presented three alternatives for where to do compression in VF (see Figure 3) and argued that the third alternative (c) was best. Here we present the empirical data to support our conclusion.

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

The second alternative of storing uncompressed image data and letting Venti compress it (b) introduces image compression in the deployment path. As we see in Table 1, image-data compression is much slower than disk write and would make image construction the new bottleneck. This would seriously impact the end-to-end performance of image deployment. We want to emphasize that this conclusion holds, independently of what deduplication storage systems are used.

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

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

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

Table 2: The effect of different Venti block sizes for deduplicating disk data. These storage figures are for disk data only, and do not account for image metadata.

| Repository format | Total (GB) | Metadata (GB) | Savings vs. `ndz` (%) |
|---|---|---|---|
| `ndz` | 233.391 | 0.912 | – |
| Venti 4 KB | 55.456 | 5.146 | 76.24 |
| Venti 8 KB | 63.085 | 3.060 | 72.97 |
| Venti 16 KB | 69.534 | 2.010 | 70.21 |
| Venti 32 KB | 75.103 | 1.485 | 67.82 |
| Venti 48 KB | 77.486 | 1.314 | 66.80 |

Table 3: Total storage space required for storing images in different repository formats, including metadata. `ndz` is for baseline Frisbee images stored in a filesystem.

## 4.3 The Space/Time Trade-off

As mentioned in Section 3.3, the choice of a block size for Venti storage can impact not only the storage savings but also the time required to retrieve and construct an image from Venti. To explore this trade-off, we populated five Venti repositories with all Linux images using 4 KB, 8 KB, 16 KB, 32 KB, and 48 KB block sizes and measured the effect on deduplication and image construction performance. Table 2 summarizes the data deduplication achieved at the various block sizes. Overall, we achieve a 3–5$\times$ deduplication ratio. The results also show that the deduplication ratio increases as we decrease the block size. This is not a surprising result, because intuitively, smaller block sizes tend to increase opportunities for deduplication. Finally we note that a larger deduplication block size improves compression and thus leads to a smaller image size (237.3 GB for the 48 KB block size versus 263.8 GB for 4 KB).

Whereas Table 2 shows just the image data stored in Venti, Table 3 shows the total amount of storage required, including the image metadata. The table includes baseline Frisbee '`.ndz`' image files as the basis for computing storage savings. For baseline Frisbee, metadata consists of the per-chunk headers that record the ranges present in each chunk; the "Total" column shows the total size of the '`.ndz`' files for our 430-image collection. For VF images, metadata includes the fingerprints (SHA–1 hashes) for retrieving data from Venti and chunk headers for chunk construction, while "Total" is the sum of metadata size and the size of the Venti repository. This table shows that we can reduce the total storage space by more than 60%. That also means that we can store more disk images, given a fixed storage space.

Based on Table 2 and Table 3, it is tempting to choose the smallest Venti block size for VF in order to maximize storage savings. This is the decision one would make if only considering storage savings. However, it is also important to consider the effect on image construction.

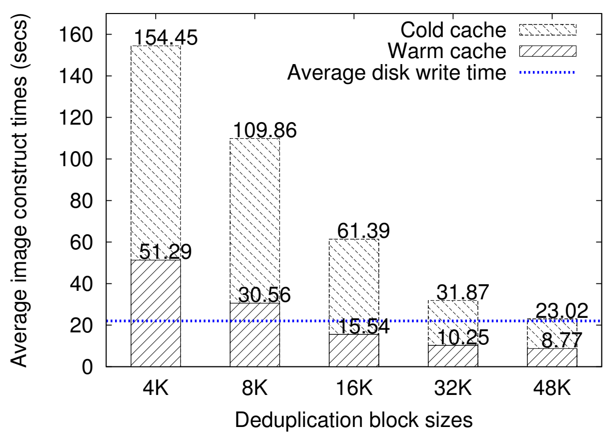

Figure 6: Average image-construction time for different Venti block sizes.

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

Figure 6 presents the results of this experiment. The stacked bars represent the average time required to construct an image in both the cold and hot cases. The horizontal line shows the average disk write time when installing an image at the target disk (from Table 1). This time is the "goal" that we must beat in order to avoid becoming the bottleneck in the image-deployment pipeline. From this figure it is clear that increasing the block size significantly decreases the image-construction overhead, especially for Venti with a cold cache. It is also clear that we can not use 4 KB and 8 KB as the block size because even with a hot cache, image-construction time exceeds the time required to write the image to disk. Of the remaining block sizes, it is tempting to use 48K since the image construction time, even with a cold cache, can match the disk-write time. However, for the purposes of this work, we chose to use 32K given that the majority of the time, the Venti cache will not be empty and we do gain slightly better deduplication. We note that the optimal block size is, in large part, an artifact of the specific performance characteristics of the deduplicating store. If we were to use a storage system other than Venti, we would need to re-evaluate the exact optimal block size, though the fundamental tradeoffs would remain the same.

## 4.4 Delivering a Large Catalog

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

### *4.4.1 Storage Savings*

To explore storage savings at scale, we loaded our corpus of 430 disk images into the Venti repository in 32 KB blocks and measured the storage size of the repository after adding each image. We loaded the images into Venti in order of their creation times, oldest to newest, to obtain a realistic sense of the growth rate for an image repository over time. As we did this, we also tracked the storage that would be required by a conventional Emulab image store—a directory of '.ndz' image files—in which the images are added in

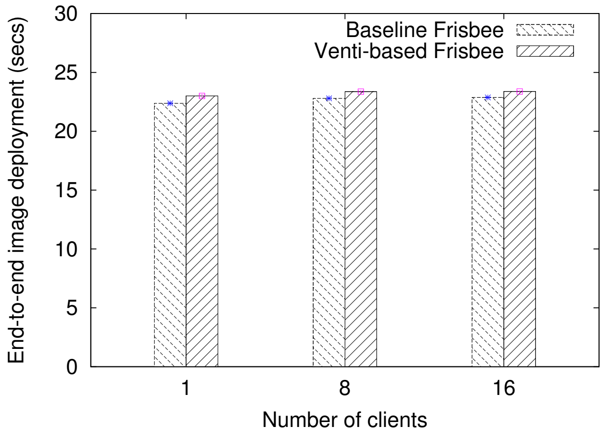

Figure 7: Average time to deploy an image to 1, 8, or 16 clients with synchronized start times

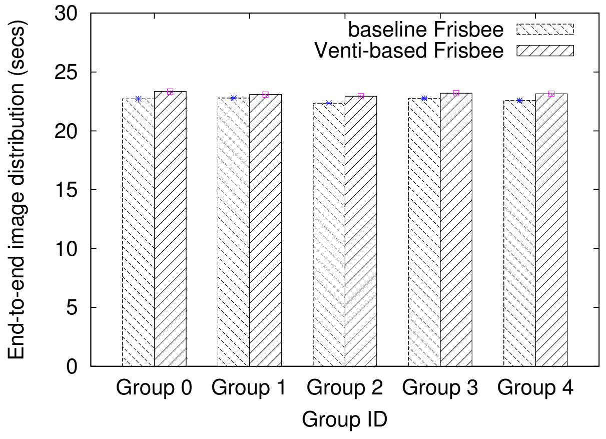

Figure 8: Average time to deploy an image to 20 clients with staggered start times. Clients start in five groups of four; for each group, we measure the time from client start to client finish.

the same order.

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

### *4.4.2 End-to-End Image Deployment*

To determine how well VF works in a production environment, we performed end-to-end tests, deploying an image from the Frisbee server to one or more client machines using both baseline Frisbee and VF with a 32 KB block size. Frisbee scales quite well with an increasing number of clients [7]. We therefore ran tests to measure image deployment with 1, 8, and 16 simultaneous clients. Running with simultaneous clients increases the request load on the server and also tends to randomize and duplicate requests—situations which the VF server must be able to handle efficiently to compete with baseline Frisbee.

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

Figure 7 summarizes our results. The bars in the figure show the average time to deploy over the ten runs of each configuration. (The figure plots the standard deviation of the time in each configuration, but these are so small that they are hardly visible.) The results show that for 1, 8, or 16 clients, VF has just over a 2% increase in run time compared to baseline Frisbee.

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

Figure 8 shows the results. The times shown are the average elapsed time for each group over the ten trials. Clients in all groups took a similar amount of time to finish. The difference between baseline Frisbee and VF is always less than 3%. These results show that VF performance is very close to that of baseline Frisbee.

### *4.4.3 Pipelined Distribution*

The experiments in Section 4.4.2 show that VF suffered only a small performance impact compared to baseline Frisbee. Yet the most significant source of additional overhead in VF is the chunk-construction process (described in Section 4.3) which could take significant time. To understand how this overhead is masked, we instrumented the end-to-end distribution process and analyzed the steps involved.

Figure 9 presents the timeline for deploying the first 10 chunks of an image using both the baseline Frisbee and VF. Each vertical bar represents the time taken to deploy a single chunk, with each pair of bars showing the time for the same chunk using both Frisbee implementations. Within each bar, the time is divided into seven categories:

1. *image construction*: the time to read (baseline) or read and construct (VF) the chunk.
2. *network transfer*: the time to transfer the entire chunk over the network, measured from the server sending the first packet of the chunk to the client receiving the final packet of the chunk.

3–5. *decompress*, *decompress + disk write*, and *disk write*: because chunk decompression is overlapped with writing decompressed data to the disk, we break down time to show when only the decompressor is active, both are active, or only the disk writer is active.

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

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

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

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

## 5. DISCUSSION

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

A major assumption in the design of Frisbee is that the complete resources of the client machine are available during the image deployment process. This ability to dedicate full CPU, RAM, and network bandwidth resources to image deployment, coupled with the use of commodity SATA hard drives, leads to the situation where the hard drive is clearly the bottleneck. We believe this situation is not unique to Emulab and is true for many image deployment systems, whether in a network testbed or a cloud infrastructure. Even using today's commodity Solid State Drives (SSDs) is not likely to move the bottleneck, since even a 1 Gbs network is capable of transmitting data more quickly, given a reasonable ($3\times$) compression ratio. However, continuing improvements in SSD performance coupled with increased prevalence of 10 Gbs Ethernet will increase pressure on the Frisbee server. The read performance of Venti will need to be revisited along with other aspects of the server such as the system used to store disk images.

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

A final question is whether it is feasible to apply techniques used in VF (and Frisbee) in other environments, specifically OpenStack [16]—one of the most popular Cloud software platforms. The image service in OpenStack is provided by Glance [17] which provides a RESTful API to add, retrieve, and query images. Glance provides a very basic image distribution service: the image data is retrieved from Glance by HTTP requests and responses. We believe the set of techniques used in Frisbee can be applied Glance. These include transmission of image data in compressed format, use of multicast for scalable image deployment, partitioning of image data into independently installable data units and the pipelining design in image deployment. We can also use data deduplication to store

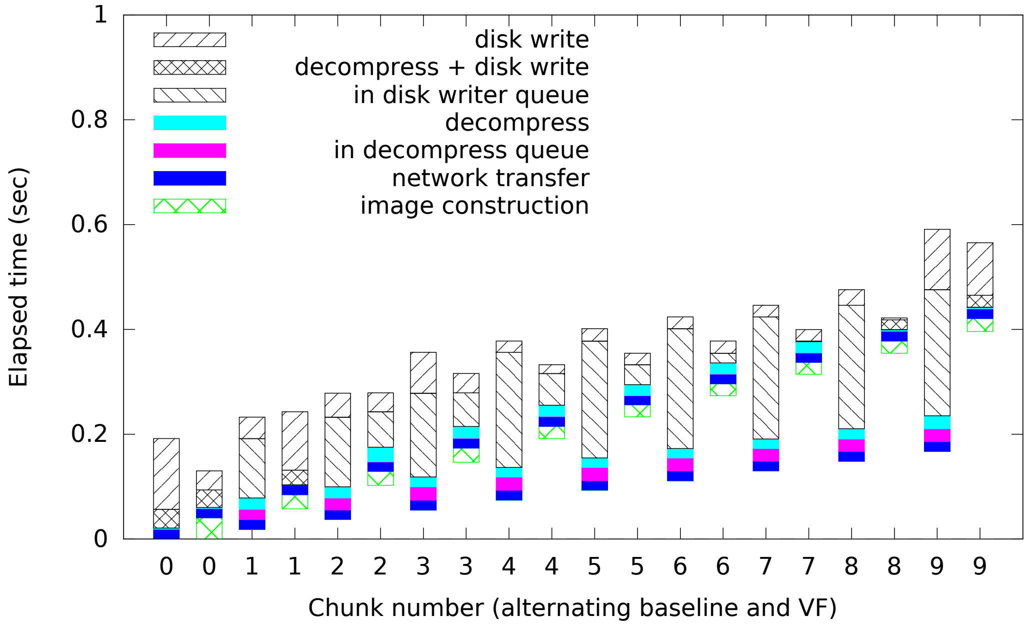

Figure 9: Timeline of chunk deployment for baseline Frisbee and VF. The x-axis interleaves per-chunk measurements from runs of baseline Frisbee and VF for the first 10 chunks of the same image. The y-axis shows elapsed time (relative to the first read of the first chunk) for each of the chunks broken down by categories.

virtual machine images.

## 6. RELATED WORK

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

Several studies have analyzed data across disk images and have found significant amounts of duplicate data; this work supplies the basic motivation for VF. Jin and Miller [9] analyzed deduplication for a set of 52 disk images, applying various chunking algorithms. For a set of fourteen related images, they showed that deduplication can reduce storage space by up to 78%. Meyer and Bolosky [13] compared block-based and whole-file deduplication over a dataset collected from 850 Windows desktop PCs in production use. They observed that there exists up to 72% and 83% duplicate data, for file-based and block-based deduplication. Jayaram et al. [8] analyzed a set of 525 virtual machine images from their production cloud datacenter and found more than 30% blocks appear at least twice. Smaldone et al. [27] and Lin et al. [11] analyzed virtual machine backup files and found there exists significantly amount of duplicate data. Atkinson et al. [2] found that disk images derived from a “base” image commonly share 60–80% of blocks with their base image.

Others have designed storage systems with deduplication to store disk images. Both the Mirage image format (MIF) [22] and the Marvin Image Store (MIS) [24] utilize file-based deduplication to improve storage efficiency. However, these stores optimize only for storage size, and are not designed specifically for scalable and efficient image deployment. Liguori et al. [10] used Venti to host *live* disk images for running virtual machines. Since most reads and writes inside the client virtual machine must communicate over the network with the Venti server, performance is poor and does not scale well. On contrast, VF installs an image on the client’s local disk, giving consistent, high performance.

Systems like Parallax [12], Capo [25] and Lithium [6] were built for running virtual machines. With Parallax, Meyer et al. proposed to run a storage virtual machine at each host, to provide access to a shared block device. This approach will not scale well because it relies on centralized, shared storage. To improve scalability, Capo uses the disk at each host machine as the persistent cache for disk images. However, it requires a special block device in the hypervisor layer. I/O performance depends on whether an I/O request hits in the cache, and it becomes less consistent and predictable. Lithium is a distributed storage system, built from local disks at each host. It uses peer-to-peer sharing to replicate per-VM storage volumes. The problem with some of these approaches is that they only work for running virtual machines; they are not applicable when users request physical machines (which testbed environments like Emulab must support). Others require running storage services in the background, which could interfere with user applications. Our approach works for both virtual machines and physical machines, and no background storage service is needed.

Other related work looks at scalable image deployment. VM-Torrent [21] and VMThunder [29] use peer-to-peer (P2P) sharing for image deployment. To further improve the opportunity to share data blocks, VDN [18] and Liquid [30] use deduplication in P2P sharing, by using block IDs based on block content, rather than the combination of image name and offset. P2P imaging is unsuitable for testbed environments, as machines are unavailable to participate most of the time: running an intensive data-transfer application would risk interfering with active experiments. Instead, Frisbee [7] uses an application-level multicast protocol for scalable image deployment.

The work most similar to VF is LiveDFS [15], which examined both the deduplication effect and image-deployment performance. However, LiveDFS scales poorly: distribution time increases linearly with the number of virtual machine instances, whereas VF inherits Frisbee’s flat scaling [7]. LiveDFS did not consider compression and failed to explore the trade-off between disk-space saving and image-deployment performance, using a single block size for experiments. VF used compression and considered several options for how to include it. We also evaluated various block sizes.

Versioning is another technique to store multi-version data efficiently, where the new version is stored as a delta (the difference between the new version and the original version), with a reference to the original version. A common use case is to implement efficient snapshots (e.g., LVM [3] and Parallax [12]). While versioning might be effective to reduce storage space, it has several limitations. First, the complexity and the runtime to retrieve data from the latest version increases linearly with the number of versions. This is undesirable since the latest version is likely to be used more frequently and should get the best performance. Second, the comparison in versioning is only between two versions while deduplication can be done globally across all stored disk images and within a single image.

Pullakandam [19] made an early attempt at using Venti to store disk images for Frisbee. It was not optimized in two ways. First, the earlier design stored raw image data into Venti, and compression of image data was done before image distribution (Approach (b) as shown in Figure 3b). At a block size of 32 KB, the image construction time in the earlier work took 80 seconds while it only took 10 seconds with a warm cache in VF. Given that it took only 22 seconds to deploy an image, image construction in the earlier work clearly becomes a significant bottleneck. Second, the chunking algorithm used in the earlier work was fixed-size chunking at the disk-level while we propose a new chunking algorithm (AFC), achieving a similar deduplication effect with a shorter runtime.

## 7. CONCLUSION

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

## 8. ACKNOWLEDGMENTS

We thank Fred Douglis, Eddie Kohler, Raghuveer Pullakandam, John Regehr, Gokul Soundararajan, and the anonymous TRIDENT-COM '15 reviewers for their comments on drafts of this paper. This material is based upon work supported by the National Science Foundation under Grant Nos. 0709427 and 1314945.

## 9. REFERENCES

[1] Amazon Web Services LLC. Amazon machine images (AMIs). `http://aws.amazon.com/`, Feb. 9, 2015. VM image data retrived from author's AWS console.

[2] K. Atkinson, G. Wong, and R. Ricci. Operational experiences with disk imaging in a multi-tenant datacenter. In *Proc. NSDI*, Apr. 2014.

[3] A. Burtsev, P. Radhakrishnan, M. Hibler, and J. Lepreau. Transparent checkpoints of closed distributed systems in Emulab. In *Proc. EuroSys*, Mar. 2009.

[4] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. In *Proc. OSDI*, Dec. 2002.

[5] File systems and Storage Lab from Stony Brook University. fs-hasher. `http://tracer.filesystems.org/`. Retrieved March 9, 2015.

[6] J. G. Hansen and E. Jul. Lithium: Virtual machine storage for the cloud. In *Proc. SOCC*, June 2010.

[7] M. Hibler, L. Stoller, J. Lepreau, R. Ricci, and C. Barb. Fast, scalable disk imaging with Frisbee. In *Proc. USENIX ATC*, June 2003.

[8] K. R. Jayaram, C. Peng, Z. Zhang, M. Kim, H. Chen, and H. Lei. An empirical analysis of similarity in virtual machine images. In *Proc. Middleware 2011 Industry Track Workshop*, Dec. 2011.

[9] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proc. SYSTOR*, May 2009.

[10] A. Liguori and E. Van Hensbergen. Experiences with content addressable storage and virtual disks. In *Proc. WIOV*, Dec. 2008.

[11] X. Lin, G. Lu, F. Douglis, P. Shilane, and G. Wallace. Migratory compression: Coarse-grained data reordering to improve. In *Proc. FAST*, Feb. 2014.

[12] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Parallax: Virtual disks for virtual machines. In *Proc. EuroSys*, Mar. 2008.

[13] D. T. Meyer and W. J. Bolosky. A study of practical deduplication. In *Proc. FAST*, Feb. 2011.

[14] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proc. SOSP*, Oct. 2001.

[15] C.-H. Ng, M. Ma, T.-Y. Wong, P. P. C. Lee, and J. C. S. Lui. Live deduplication storage of virtual machine images in an open-source cloud. In *Proc. Middleware 2011*, Dec. 2011.

[16] OpenStack Foundation. OpenStack. `http://www.openstack.org/`, May 2015.

[17] OpenStack Foundation. OpenStack Glance. `http://docs.openstack.org/developer/glance/`, May 2015.

[18] C. Peng, M. Kim, Z. Zhang, and H. Lei. VDN: Virtual machine image distribution network for cloud data centers. In *IEEE INFOCOM*, Mar. 2012.

[19] R. Pullakandam. EmuStore: Large scale disk image storage and deployment in the Emulab network testbed. Master's thesis, University of Utah, Aug. 2014. `http://www.flux.utah.edu/paper/pullakandam-thesis`.

[20] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proc. FAST*, Jan. 2002.

[21] J. Reich, O. Laadan, E. Brosh, A. Sherman, V. Misra, J. Nieh, and D. Rubenstein. VMTorrent: Scalable P2P virtual machine streaming. In *Proc. CoNEXT*, Dec. 2012.

[22] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala. Opening black boxes: Using semantic information to combat virtual machine image sprawl. In *Proc. VEE*, Mar. 2008.

[23] S. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in Foundation. In *Proc. USENIX ATC*, June 2008.

[24] R. Schwarzkopf, M. Schmidt, M. Rüdiger, and B. Freisleben. Efficient storage of virtual machine images. In *Proc. ScienceCloud*, June 2012.

[25] M. Shamma, D. T. Meyer, J. Wires, M. Ivanova, N. C. Hutchinson, and A. Warfield. Capo: Recapitulating storage for virtual desktops. In *Proc. FAST*, Feb. 2011.

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

[27] S. Smaldone, G. Wallace, and W. Hsu. Efficiently storing virtual machine backups. In *Proc. HotStorage*, June 2013.

[28] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, Dec. 2002.

[29] Z. Zhang, Z. Li, K. Wu, D. Li, H. Li, Y. Peng, and X. Lu. VMThunder: Fast provisioning of large-scale virtual machine clusters. *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[30] X. Zhao, Y. Zhang, Y. Wu, K. Chen, J. Jiang, and K. Li. Liquid: A scalable deduplication file system for virtual machine images. *IEEE Transactions on Parallel and Distributed Systems*, 2014.

[31] B. Zhu, K. Li, and H. Patterson. Avoiding the disk bottleneck in the Data Domain deduplication file system. In *Proc. FAST*, Feb. 2008.