

Opening Black Boxes: Using Semantic Information to Combat Virtual Machine Image Sprawl

Darrell Reimer

IBM T. J. Watson Research Center
dreimer@us.ibm.com

Arun Thomas

University of Virginia
arun@cs.virginia.edu

Glenn Ammons

IBM T. J. Watson Research Center
ammons@us.ibm.com

Todd Mummert

IBM T. J. Watson Research Center
mummert@us.ibm.com

Bowen Alpern

IBM T. J. Watson Research Center
alpernb@us.ibm.com

Vasanth Bala

IBM T. J. Watson Research Center
vbala@us.ibm.com

Abstract

Virtual-machine images are currently distributed as disk-image files, which are files that mirror the content of physical disks. This format is convenient for the virtual machine monitors that execute these images. However, it is not well-suited for administering images because storing images as disk-image files forces administrators to maintain the software on images with the same tools that they use to maintain the software on physical machines. Already, these tools cannot cope with “physical server sprawl”; in the future, because images can be snapshotted and cloned easily, enterprises that migrate from physical machines to images will need tools that scale to cope with the larger problem of “virtual-machine image sprawl”.

To address this problem, this paper proposes the *Mirage image format (MIF)*, a new storage format that exposes the rich semantic information currently buried in disk-image files. Disk-image files contain a mapping from file name to file content (and file metadata). MIF decouples this mapping into a *manifest* that maps file names to *content descriptors* (and file metadata) and a *store* that holds the content. Each image has its own manifest and a store may contain content for many images. As with disk-image files, images in MIF fully encapsulate application state including all software dependences. In addition, conversion between MIF and traditional disk-image formats is easy.

This paper shows, through examples, that MIF makes some typical software management tasks—inventory control, customized deployment, and image update—faster and easier. The general technique is to operate on manifests instead of on content whenever possible. These tasks can be performed without starting images and, because manifests are simpler and orders of magnitude smaller than disk-image files, without accessing large amounts of data.

Categories and Subject Descriptors K.6.2 [Management of Computing and Information Systems]: Installation Management; K.6.3

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

VEE '08 March 5–7, 2008, Seattle, Washington, USA.
Copyright © 2008 ACM 978-1-59593-796-4/08/03...\$5.00

[Management of Computing and Information Systems]: Software Management

General Terms Management.

Keywords Virtualization, deployment, management, installation.

1. Introduction

Virtualization is widely touted as a solution to both client-side and server-side problems in large enterprises. On the server side, the problem is *physical server sprawl*: the tendency for enterprises to accumulate underutilized, heterogeneous, power-hungry, unmanageable servers. The virtualization solution is to replace each server with a virtual machine image (perhaps a software appliance) and run these images on a smaller number of well-utilized, homogeneous, thrifty, and centrally managed machines. Clients suffer from their own kind of sprawl, because, unless clients are centrally managed, no two clients are exactly alike, yet all must be kept up-to-date with the latest software. A virtualization solution is to distribute client-side software as software appliances, which encapsulate an application together with a complete, configured environment.

Virtual machine images are convenient because they can be treated as data, but they also are subject to sprawl. As data, images can be cloned, versioned, stored in archives, and transformed; while physical servers cost money, creating a new image is “free”. However, these images must still be stored, and worse, because each image contains a full stack of software, each image must be maintained. This problem has been called *virtual machine image sprawl* or just *image sprawl*. Of course, an enterprise could avoid sprawl by creating only a few virtual machines, but this strategy foregoes some advantages of virtualization, such as stronger isolation between applications and a record of changes to applications.

This paper addresses image sprawl by treating images as structured data, stored in a centrally managed repository. A new storage format, the *Mirage image format (MIF)*, exposes the rich semantic information currently buried in disk-image files. Disk-image files contain an implicit mapping from file name to file content (and file metadata). To access this mapping, one must have the complete image — for some tasks, the image must be started. By contrast, MIF decouples this mapping into a *manifest* that maps file names to *content descriptors* (and metadata) and a *store* that holds content.

MIF has several advantages. It saves space because a file’s content is stored only once, even if that content appears in more than one image or in multiple files of the same image. Some software

management tasks that used to require complete images now require only manifests and/or partial images. These tasks can be performed without starting the image and, because manifests are simpler and orders of magnitude smaller than disk-image files, without transferring large amounts of data. Section 5 presents examples, modeled on real-world usage scenarios, that include searching a repository for images that contain a particular file, deploying customized “clones” of a base image, and upgrading a large number of similar images with a new software package. For these tasks, operating on manifests and partial images is up to two orders of magnitude faster than operating on disk images.

This paper makes the following contributions:

- A new image format, MIF, that exposes the semantic information implicit in virtual machine images. Images in MIF can be stored compactly. As with current formats, images in MIF fully encapsulate application state including all software dependencies. Furthermore, it is easy to convert between MIF and traditional disk-image formats.
- Analyses and optimizations that exploit MIF to improve software management tasks. By operating on manifests and partial images, we obtain orders-of-magnitude speedups for inventory control, customized deployment, and software updates.

The remainder of the paper is organized as follows. Section 2 explains the software management tasks that motivate this work. Section 3 discusses the design and implementation of MIF, the repository, and the analyses and optimization that rely on them. Section 4 presents three real-world software management scenarios and explains how we modeled them in our experiments. Section 5 evaluates the performance of the proposed system on those models. Section 6 reviews related work. Finally, Section 7 concludes the paper and discusses future work.

2. Motivation

The work reported upon in this paper is motivated by three software tasks: inventory control, customized deployment, and updates. This section explains these tasks with a focus on how *sprawl*—both physical and virtual—makes them more problematic.

Inventory control is the problem of determining what software is installed where. Inventory control is important for many reasons: enterprises pay for most commercial software on a per-installation basis; for reliability, installed software must be kept at known version levels and updated or patched systematically; installing some software, such as file-sharing software, at the enterprise is legally risky or simply unethical; and installing other software, such as viruses, can damage the enterprise’s systems.

There are many commercial products for inventory control of physical machines, including virus scanners and compliance checkers such as IBM’s Tivoli License Compliance Manager [7], all of which work similarly. These products install an agent on each machine that periodically scans the filesystem, determines which software is installed, and sends a report to a central server. In many cases, the software in question could be installed anywhere on the system, so these products do not search for files by name. Instead, they compute a cryptographic checksum of each file’s contents and compare the checksum to a database of known hashes.

Our design (see Section 3) solves two problems with this method of inventory control. First, ensuring that the agent is installed properly on each machine is itself an inventory control problem. Second, scanning every filesystem is expensive, even when optimized. Our design scans each image once only, as it is added to a centrally managed repository. The scan is “for free” because the image must be transferred to the repository anyway. After the scan,

inventory control queries become queries over manifests, which can be evaluated quickly.

Deploying software in the face of *sprawl* is a huge challenge for most enterprises. The enterprise must determine which machines or images are eligible for the deployment; this is an inventory control problem. Next, the enterprise must install and validate the software on each server or image. Installing enterprise software can be an arduous task.

Virtualization should help, because the software could be installed and validated on one “master” image, which could then be cloned to run on many virtual machines. However, *sprawl* says that the enterprise needs many heterogeneous images. Even in the simplest environment, each instance of the master needs a unique hostname and IP address. Therefore, either the new software must be deployed to many images, or the master image must be customized to produce many instances, each with a slightly different configuration.

Sprawl causes two problems here. First, changing many images takes time. Second, storing multiple images is costly unless their similarity is exploited. As Section 3 explains, our system addresses the first problem through optimizations of deployment to or customization of many similar images. The second problem is addressed through a space-efficient encoding of images.

The final motivating task is updating software. The effect of *sprawl* on this task is obvious: the greater the variety of machines or images that must be updated, the more difficult the task. As in customized deployment, eligible machines or images must be identified and the update must be installed on each eligible machine. However, unlike customized deployment, updating software makes a single big change to many different machines or images. The same features of our system that support customized deployment also support updates, but updates require a further optimization: identifying a generic portion of the update and executing it on manifests instead of on individual images.

3. Design and Implementation

This section describes the design and implementation of MIF and the Mirage repository. We explain the goals of the design, its implementation, and how it improves three software management tasks: inventory control, customized deployment, and updates. Finally, we discuss limitations of the design and its implementation.

Two important goals conflict. The chief goal of MIF is to represent files explicitly, since files are the building blocks of any virtual machine image. However, a second goal is that there be few constraints on the images that can be represented in MIF. In particular, the image’s operating system or target virtual machine monitor should not matter.

The first goal requires that, on converting an image to MIF, each filesystem in the image must be traversed. So, one constraint on images is that they must contain only filesystems that the current implementation can traverse. At this time, only the Ext2 and Ext3 filesystems [21] are supported (see Section 3.2).

An assumption of the Mirage project is that many images in the repository will be similar to one another. Three goals rely, at least in part, on this assumption:

- Storage efficiency. Many of the images in the repository will share files, and this redundancy must be exploited.
- Fast retrieval. Retrieving an image must take time proportional to the size of the image’s difference from previously retrieved images.
- Support for analyses and optimizations. For example, it should be possible to analyze the effect of an update and use that information to speed updates to similar images.

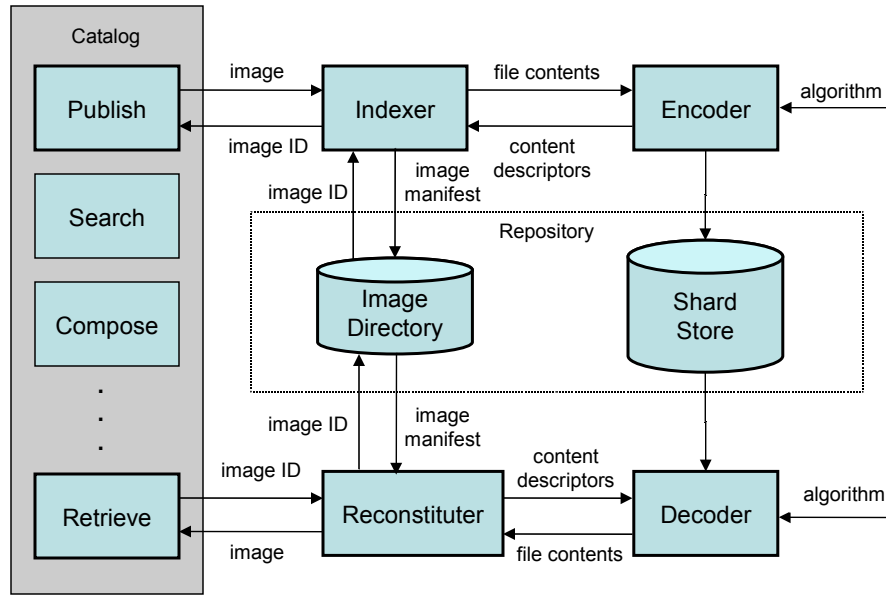


Figure 1. Architecture of Mirage. The catalog (left) is the interface to the system. The repository (middle) consists of an image directory and a shard store. Publishing an image uses modules for indexing images and encoding files (top); retrieving an image uses modules for reconstituting images and decoding files (bottom).

Figure 1 shows the architecture of the system. Each block in the figure corresponds to a library (not a process). To interact with the system, clients use the *catalog* interface (left side of the figure), which has operations for publishing images, listing images (not shown), retrieving images or parts of images, searching for images that contain particular files, and composing images (see below).

The repository, which consists of two parts, is in the center of Figure 1. The *image directory* lists the unique identifier of each image in the repository. This identifier also identifies the image’s *manifest* (see below). The content-addressed *shard store* holds the rest of the data in the system: manifests, persistent data structures for the catalog, and of course the contents of files. The term “shard” is our name for an atomic chunk of data and was inherited from an earlier project, PDS [1]. Shards are added to the store by the *encoder* and retrieved by the *decoder*, each of which is parametrized by an algorithm that determines how shards are identified; currently, the identifier of a shard is its SHA1 checksum [11].

In between the catalog and the shard store, the *indexer* converts images to manifests and the *reconstituter* converts manifests back to images or to file trees. Manifests are the central data structure of the system and there are three kinds, whose structure is as listed in Figure 2.

A *virtual machine image manifest* or *image manifest* represents an image and stores three kinds of information. First, an image manifest describes the image, such as its size and the type, size, and location of each of its partitions. Second, an image manifest specifies the algorithm used to encode the image. Finally, the image manifest stores references to a *file manifest* and a *content manifest*.

The file manifest lists each filesystem path in the image along with its metadata¹ and, for paths to regular files, its *file checksum* and *content-descriptor index*. The file checksum, which varies with the encoding method, is a collision-resistant hash (currently

```

struct VirtualImageManifest{
    int manifestVersion;
    int encodeMethod;
    u64 imageSize;
    int numberOfPartitions;
    PartInfo partitionInfo;
    Checksum fileManifestChecksum;
    ContentDesc fileManifestContentDesc;
    Checksum contentManifestChecksum;
    ContentDesc contentManifestContentDesc;
};
struct FileManifest{
    int manifestVersion;
    int encodeMethod;
    FileMfstEntry ** fileMfstEntry;
};
struct FileManifestEntry{
    int entrySize;
    Checksum fileChecksum;
    Metadata fileMetadata;
    u32 contentDescriptorIndex;
    char * fileName;
};
struct ContentManifest{
    int manifestVersion;
    int encodeMethod;
    ContentDesc **contentMfstEntries;
};

```

Figure 2. Manifests in MIF, in C-like pseudocode.

¹On Unix operating systems, the metadata includes the data returned by the `stat` system call.

SHA1): with very high probability, two files that have the same file checksum have identical contents.

The content-descriptor index is used to find the *content descriptor* of the file's content in the content manifest. The structure of content descriptors will vary with the encoding method. The current method stores the contents of each (unique) file as a single shard. For this method, the content descriptor is the identifier of that shard.

Separating file manifests and content manifests allows encoding methods and shard-store formats to evolve separately. For example, the shard store could be migrated to use a new kind of shard identifier without changing file manifests. Similarly, if an encoding method that stores deltas between similar file contents were desired, the new encoding method could be implemented by augmenting the content descriptors without changing the code that interacts with the shard store.

3.1 Analyses and optimizations

We have built several analyses and optimizations that exploit MIF. These extensions address the inventory control, customized deployment, and image update problems introduced in Section 2.

The inventory control problem is “given a list of file contents, find all images that contain a file with content from the list”. Traditionally, this is done by scanning each image (physical or virtual) for a match. By contrast, for images in MIF, the problem can be solved by searching each file manifest for checksums that match the target content's checksum. Essentially, MIF records the result of scanning each file's content so that the scan need not be repeated.

There are a number of ways to implement the search. One easy way is as follows. First, build a hash table that stores a cryptographic checksum of each content in the list. Retrieve the manifest of the image to be searched from the repository. Finally, walk the file manifest and, for each file, report a match if the file's checksum is in the hash table. A more elaborate implementation would avoid a linear search of each manifest by building a map of checksums to manifests and reusing the map for searches of multiple images. Section 5 shows that the simple implementation can search a single image in one or two seconds.

MIF also enables custom deployment of virtual-machine images. If multiple instances of an image are to be deployed on a network of virtual machines, each instance must be assigned a unique hostname. On a Unix system, changing the hostname may require changing a single file: `/etc/hostname`. To support such small customizations efficiently, Mirage implements *selective retrieval* and *manifest composition*.

Selective retrieval retrieves selected files from an image instead of the image itself. For the hostname example, one would retrieve the file `/etc/hostname` and no other files. Note that the time for a selective retrieval is proportional to the size of the image's manifests and the size of the files requested and not to the size of the image.

Manifest composition adds or replaces files in an image with files from another manifest, producing a new image in the repository. For hostname customization, a unique hostname would be stored in `/etc/hostname` and a tiny file manifest that contains just this file would be published. Such tiny file manifests are called *overlay manifests*. Finally, the customization would compose the overlay manifest with the original manifest to create the customized image in the repository. The time for manifest composition is proportional to the size of the manifests used to create the new image.

Section 4 shows that selective retrieval and image composition result in significant speedups for installation of Debian packages as well as for hostname customization.

The image update optimization exploits the assumption that many images in the repository will be similar to one another. Updates fall into two classes: image-specific updates and generic updates. An image-specific update depends on details of the image that are unlikely to recur in other images, while a generic update depends on features that are common to many images.

This optimization uses *memoization* to speed up generic updates. In general, to memoize a function, one maintains a table that maps function arguments to function results. Before the function is computed for a given argument, the table is consulted to see if it already contains a result for that argument. If so, the computation need not be repeated. If not, the computation is performed and a new argument-result pair is added to the table.

In this case, the function is a generic update script, which must be deterministic. Given a command line, the script reads and writes files. A “function argument” in the memoization table is a command line together with that part of the image state that the script reads when given that command line. In the current implementation, this part of the image state is encoded as a list of paths (of files read by the script) together with the checksums for (the initial contents of) these files. A “function result” in the memoization table records the list of paths of files altered by the script together with checksums of their final contents. The results are stored as overlay manifests. Note that the argument does not depend on files that the script does not read (with a given command line), nor does the result list files that the script does not alter.

To apply an update script to an image, the memoization table (a hash table that is stored in a well-known location in the repository) is checked for an earlier run of the script with the same command line on files that exactly match those in the image. If a match is found, then a new image is created by composing the image with the match's overlay manifest, without actually running the update script. The time for this operation is proportional to the size of the table entry (both the argument and the result) and not to the size of the input files that the script would have accessed.

If the desired entry is not found in the table, then the update script is run on the image. After running the script, the appropriate entry is added to the memoization table.

Mirage uses Strace [19] to identify files read and written during script executions.

Memoization does not apply to image-specific updates. However, some image-specific updates can be split into two updates, one of which is generic. Section 4 describes how we performed such a split of Debian's update tool, `Dpkg`.

3.2 Limitations

Both the design and implementation of MIF and the Mirage repository have limitations.

The design assumes that cryptographic checksums of file contents capture similarities among images. This assumption fails when a file's contents do not depend solely on the file's “meaning”. For example, a digital rights management scheme may add a watermark to the file, where the watermark varies from image to image even though the semantics of the file remains the same. Similarly, a virus that morphs itself in each generation cannot be detected by searching for a known checksum.

The implementation is limited in that it supports only Ext2 and Ext3 filesystems on Linux. Since the indexer traverses each filesystem with the kernel's own drivers, it would be easy to support any filesystem that has a Linux driver. Indexing filesystems without a Linux driver would require more work, as would indexing other hierarchical namespaces, like the Windows registry.

Memoization is unsafe when the update is non-deterministic; Mirage relies on the user to certify that a given script is determinis-

tic. The current implementation does not support scripts that delete or rename files.

4. Usage Scenarios

For each software-management task listed in Section 2, this section describes a real-world scenario and explains how the scenario is modeled for the evaluation in Section 5. In the inventory control scenario, images are searched for particular programs; in the customized deployment scenario, a cluster of servers is set up; and in the update scenario, a new package is installed on a number of clients.

4.1 Scenario: search images for programs

In this scenario, images are searched for certain programs and a report is generated that lists which images contain which programs. Enterprises need such scans for a number of reasons: the programs may be commercial software with a per-seat license, the programs may need an upgrade, or the programs may be viruses or forbidden software that must be removed.

This scenario is modeled as a file-content search. For each program, there is an associated content. If an image contains a file with that content, then the program is installed on the image; otherwise, the program is not installed. The search is performed by comparing the content descriptors in image manifests with checksums of the associated contents, as described in Section 3.

The goal here is to show that file-content searches are about as fast with MIF as they would be with a traditional inventory control system. Traditional inventory control systems scan images (physical or virtual) and build a database of checksums; later, the database is used for searches. With MIF, searches are no faster but the database is built “for free” as a side-effect of adding images to the repository. Another benefit is that the database is always synchronized with the repository.

4.2 Scenario: deploy a cluster of servers

In this scenario, a cluster of virtual servers is deployed to a cluster of physical machines. The new cluster might serve a new application at the enterprise or be used to increase the throughput of an old application. In either case, the repository stores a *master image* that contains the application and its environment. A number of instances are cloned from the master and customized to fit the physical cluster. For example, the master image might be fully configured except for its network settings.

The model for this scenario replaces the content of five network-configuration files of the master image. For a typical installation, the total size of these files is on the order of 50 kilobytes. Each instance is customized in four steps:

1. From the master, retrieve the five network-configuration files.
2. Change the five files.
3. Publish the customized files, creating an overlay manifest.
4. Compose the overlay manifest with the master to create the instance in the repository.

By contrast, a “traditional” approach would retrieve the entire master image and change the five files to create an instance, and publish the instance to the repository. With this approach, it is tempting to deploy the instances without publishing them; however, this is cheating, because deployed instances are then not tracked by the repository. Section 5 shows that the Mirage approach achieves the same performance as the cheat while keeping the repository up-to-date; both the cheat and the Mirage approach are orders of magnitude faster than the traditional approach without the cheat.

4.3 Scenario: install a package on clients

In this scenario, new software is installed on a number of clients. The software might support a new application at the enterprise or replace an old application’s client-side code with the “latest and greatest” code. Large enterprises support many clients; often, no two clients are exactly alike.

The model assumes that it is enough to update the repository with new client images; some other mechanism must ensure that users run appropriate images. The model also assumes that the enterprise distributes software as *packages*, such as is common practice for Linux distributions and other systems. A package bundles related programs, data, and documentation into one file. The distribution’s maintainers manage a repository of packages and each package lists its dependences on other packages in the repository. Packages are installed by running a package installation tool; for example, on Debian [18], packages are installed with Dpkg. Dpkg maintains a *package database* on each image, which lists the packages installed on that image.

When a simple package is installed on Debian, Dpkg must

1. Verify that the package’s dependences are satisfied by the packages already installed on the image.
2. Unpack the new package’s files.
3. Configure the package by running the new package’s post-installation script.

More complicated packages have a lengthier install sequence that involves running more scripts, but the above is common. Two of these steps will vary from one image to another. Step 1 varies with the image’s package database, and step 3 varies with the new package’s configuration.

By contrast, step 2 is the same for every image because the package’s files are always the same. Also, step 2 is the slowest step for all but small packages. Therefore, if the system could remember the outcome of step 2 from an install on one image and reuse that outcome when installing on other images, the latter installs would be significantly faster.

In the experiments of Section 5, a Debian package is installed on images using an optimized Dpkg² that exploits memoization, selective retrieval, and manifest composition. The optimized Dpkg splits installation into a generic part, which is memoized, and an image-specific part, which is not memoized. To install the package on the first image, the user tells Dpkg to execute these steps:

1. Create an overlay manifest for the unpacked package.
 - (a) Unpack the package’s files.
 - (b) Publish the unpacked files, creating an overlay manifest.
2. Compose the overlay manifest with the first image, creating (in the repository) an image with the unpacked package.
3. Create an overlay manifest for the installed package.
 - (a) Retrieve selected files from the image for the unpacked package. Only the files necessary to complete the installation are retrieved—typically, this does *not* include most of the files from step 1(a).
 - (b) Configure the package by running the package’s post-installation script in the context of the retrieved files.

²The optimized Dpkg does not support some of Dpkg’s features. In particular, scripts that change the state of a running system (for example, start or stop services) and upgrades of certain already installed packages do not work properly. Handling state of a running system is a research challenge, but we believe that all of Dpkg’s other features could be supported efficiently, with more work.

(c) Publish the resulting files, creating an overlay manifest.

4. Compose the overlay manifest for the installed package with the first image, creating (in the repository) an image with the installed package.

Step 1 is the generic part of the installation and is memoized. In this case, memoization is trivial, since the first step depends on none of the image’s files. To install the package on subsequent images, the user tells Dpkg to execute steps 2 through 4.

In step 3a, the list of files to retrieve depends on the package and (rarely) on the image. To construct the list, Strace [19] was used to track all file accesses during a normal installation, a method that works only when the list does not depend on the image. A more general alternative is to declare this information within packages.

By contrast, a “traditional” approach would retrieve the entire image, install the package, and publish the image to the repository. Section 5 shows that (for a large package) the speedup of the above approach over the traditional approach is 2.3 on the first image and 21 on subsequent images. As with the last scenario, the traditional approach could cheat: instead of running the installation at the repository, force clients to run the installation on their own machines. This cheat distributes some (but not all) of the load but is harder to manage, especially when the clients are heterogeneous.

Speedup is not the only benefit of our approach. Package management tools for physical machines (such as Dpkg) assume that installing and removing packages are dangerous operations. These tools contain many guards against corrupting the system; for example, they ensure that certain operations are atomic. Because our approach stores images in a repository, with snapshots of each stage of the installation, corruption is not a concern, so package management tools can be simpler and perhaps amenable to more optimizations.

5. Results

5.1 Experimental Methodology/Setup

All experiments were run on an IBM X Series Blade, with IBM ESXS disks (146.8GB, 16MB buffer, 10000rpm, 3.8ms average seek time), 2 CPU Intel Xeon E5345 processors (2.33GHz, 1333HMz FSB, L1 Cache 32K, L3 Unified Cache), and 4GB RAM. The operating system was Linux, Ubuntu 7.04 (kernel 2.6.20-1).

The methodology is as follows. All reported times are the median of three test runs; the variance was not significant in any of the experiments. Before each timed run, all pending writes are forced to disk with `sync` and the filesystem caches flushed with `echo 3 > /proc/sys/vm/drop_caches`. Whenever data is written to disk, the time to force pending writes to disk is included in the measurements.

The disk-image files for these experiments are raw disk images. 5 different images were used, all based on the Debian Linux distribution and created as sparse files:

Small A minimal install.

Base A typical non-desktop install.

Wiki An image with Apache, PHP, MySQL, and MediaWiki.

Big A desktop image with X Windows and standard desktop productivity tools.

IDE An image with a large, commercial, Eclipse-based development environment.

In addition to these images, experiments on multiple experiments used 40 images, all similar to IDE, generated from 40 successive builds of the development environment.

Table 1 lists each image and its characteristics, including the number of files it contains and its disk usage when mounted.

Name	Files (10 ³)	Size (GB)	Manifest sizes (MB)			Time (s)	
			Image	File	Con.	Pub.	Ret.
Small	20	0.28	0.004	3.0	0.5	34	21
Base	27	0.45	0.004	4.0	0.7	49	28
Wiki	39	0.84	0.004	6.1	1.2	137	102
Big	66	1.67	0.004	10.5	2.2	309	246
IDE	79	2.24	0.004	13.1	2.4	451	353

Table 1. For each image: its number of files (in thousands); mounted size; the size of its image, file, and content manifests; and the time to publish and retrieve the image to the Mirage repository.

5.2 Repository Performance

5.2.1 Performance on a single image

Table 1 also shows the sizes of the 3 manifests created for each image by Mirage. Image manifests are very small, roughly 4KB and independent of image size or number of files. The sizes of the file manifest is roughly proportional to the number of files in the image. It is not exactly proportional because file names vary in length. In fact, the file manifest implementation stores the complete path name for every file, so there is redundancy that a smarter scheme could remove. Finally, the size of the content manifest is also roughly proportional to the number of files in the image. This is an artifact of the implementation’s encoding method, which stores each file’s content as a shard. Other encoding methods scale differently: for example, if the encoding method broke each file into chunks (all of about the same size) and stored each chunk as a shard, then the size of the content manifest would be roughly proportional to the size of the mounted image.

Finally, Table 1 shows the time to publish each image to the Mirage repository and to retrieve the image from the same repository. As an optimization, the publishing implementation avoids sending shards that already exist in the shard store by asking the shard store if it already has a given shard identifier before sending the shard. Publish times shown are for publishing the image to an empty repository.

Both publish and retrieve times are roughly proportional to the size of the mounted image. The times depend on how the filesystem underlying the shard store lays out files, how the operating system caches disk blocks, how the hard drive caches blocks, the pattern of store accesses, and so forth, so the times should not be expected to be exactly proportional to image size. The times in Table 1 are similar to the times for simply copying the mounted image’s files with `cp -a`, as one would expect.

5.2.2 Performance on multiple images

A repository that performs well when storing one image is of no interest if it does not also perform well when storing many images. We do not have large image repositories yet, but we have evaluated Mirage on modestly-sized repositories. In the following, Mirage is compared with two other image-encoding schemes:

Raw The repository stores images in a traditional “raw” disk format, with no compression.

Gzip The repository stores images in a traditional raw disk format and compresses each image with Gzip [5].

Figure 3 and Figure 4 show how the space required by small repositories varies with how the images are stored. On a repository that contains only the 5 Debian images (Figure 3), MIF does almost as well as Gzip, even though MIF does not compress individual files; instead, MIF achieves its space savings by storing each file content only once, even if it appears in multiple files. This ad-

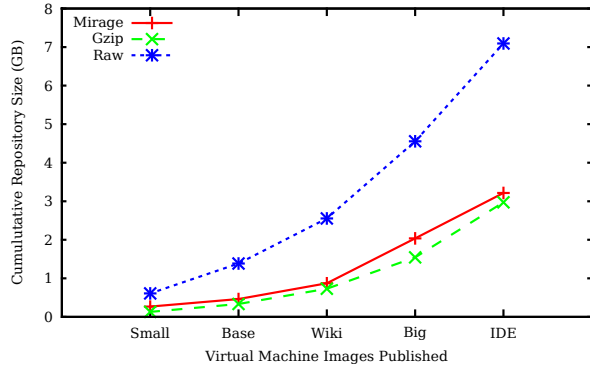


Figure 3. Growth of the repository as five images are added successively, for the Raw, Gzip, and Mirage encoding schemes.

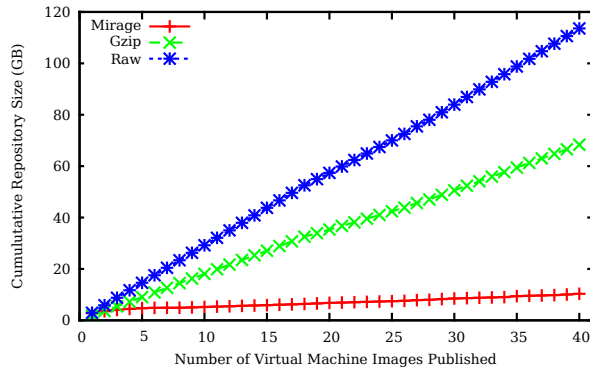


Figure 4. Growth of the repository as 40 successive IDE images are added, for the Raw, Gzip, and Mirage encoding schemes.

Operation	Time (s)	
	Average	Std. Dev.
Standard Gzip	248	12
Standard Gunzip	146	12
Mirage Publish	109	35
Mirage Retrieve	599	21

Table 2. Average and standard deviation of times to publish and retrieve each of 40 successive IDE images, with Gzip and Mirage.

vantage of MIF is clear in Figure 4, which shows the storage cost of storing 40 successive IDE images in the repository. In this scenario Mirage requires 9.9GB, while Gzip requires 65GB. Raw’s storage cost is even higher, 108GB. Here, Mirage is 6.5 times better than Gzip and 10.9 times better than Raw.

Table 2 compares the time required to publish and retrieve each of the 40 IDE images with Mirage to the time required with Gzip. On average, it takes Mirage 146s to publish an image, compared to Gzip’s 247s³. Thus, Mirage not only requires less space, but also publishes faster. However, retrieving images with Mirage is significantly slower than retrieving images with Gunzip, because

³ The high standard deviation is due to the cost of publishing the first image: publishing subsequent images is faster because many of their file contents already exist in the shard store.

Name	Size (GB)	Checksum Time (s)	
		1 File	1000 Files
small	0.28	0.5	1.2
base	0.45	1.1	1.3
wiki	0.84	1.6	1.9
big	1.67	2.2	3.0
IDE	2.24	2.6	3.2

Table 3. For each image, its mounted size and the time to search the MIF-encoded image for one file checksum and successively for each of 1000 file checksums.

Mirage reads more data from disk and reads from many files instead of linearly through one file.

5.3 Virtual Image Search / Inventory Control

Table 3 shows the times to search the 5 different virtual image stored in the Mirage repository for certain file checksums. The first test was to search each image for a single file checksum, the second test to search for 1000 different checksums. Since the current implementation performs a linear scan of the file manifest, the time to search an image depends on the number of files in the image and the number of file checksums sought. Even with this straightforward implementation, searching the largest image (IDE) for 1000 different file checksums takes only 3 seconds.

5.4 Customized Deployment

This section evaluates the performance of MIF on the customized deployment scenario from Section 4.2. In this scenario, a master image is customized by replacing its network configuration files with new files. The size of the new files totaled 32KB. As the master image, we used Base (a small image) and IDE (a large image).

Figure 5 shows the customization time for each master image with the traditional approach and Figure 6 shows the time with the MIF-optimized approach described in Section 4.2. Compared with the traditional approach, the MIF optimizations yield a speedup of 120 on Base and 507 on IDE. With the optimized approach, storing each image in the Mirage repository consumes 40KB of disk space; this includes storage for the new files, the overlay manifest, and the new image manifests. The traditional approach consumes more storage because, instead of storing an overlay manifest, it stores a complete new file manifest; the traditional approach consumes 3MB of disk space.

Figure 6 shows that, even though selective retrieval fetches very few files from the repository, the time for selective retrieval exceeds the time to create and compose the overlay manifest. This is because selective retrieval scans the entire (large) file manifest, while the overlay manifest operations scale with the number of files in the overlay manifest. Thus, selective retrieval from Base takes 0.4s, while selective retrieval from IDE, which has 3.3 times as many file manifest entries, takes 1.3s.

5.5 Updates

This section evaluates the performance of MIF on the package-installation scenario from Section 4.3. In this scenario, a new package is installed on an image. We used two different packages: Wine, a Windows emulator, and Email, a large email/office productivity system. The Wine package is 8.6MB and the Email package is 295MB. In each case, the base image was Desktop.

Figure 7 shows the time for both the traditional and MIF-optimized installs. On Wine, the MIF-optimized install gives a speedup of 19 on the first install; for subsequent installs, memoization further increases the speedup to 23. On Email, the MIF-

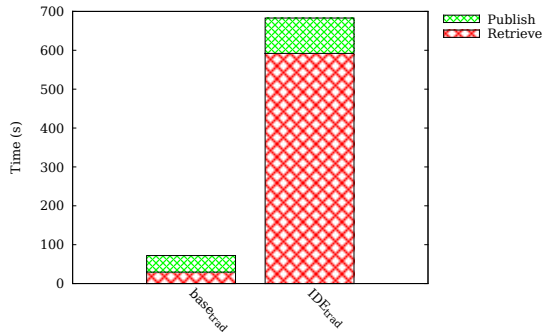


Figure 5. Time to customize the Base and IDE images the traditional way, by retrieving the image, customizing it, and publishing the customized image. The customization time is insignificant and not shown.

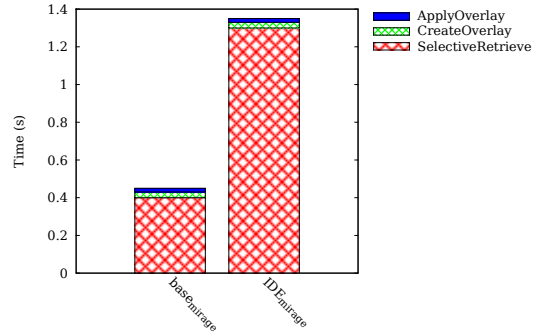


Figure 6. Time to customize the Base and IDE images in MIF format by retrieving selected files, customizing those files, creating an overlay manifest, and applying the overlay to the original image to produce a new image in the repository. The customization time is insignificant and not shown.

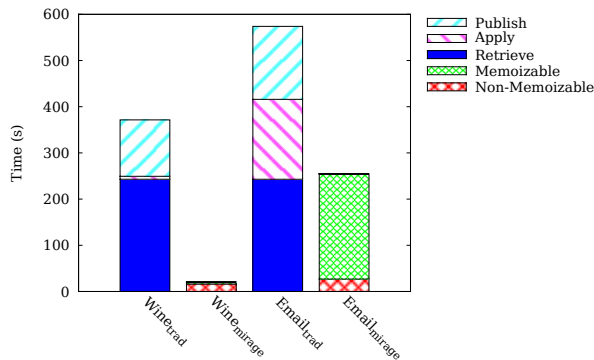


Figure 7. Time to install Wine and a large Email package on the Desktop image. For traditional installs ($\text{Wine}_{\text{trad}}$ and $\text{Email}_{\text{trad}}$), the install consists of retrieving the image, installing the software, and publishing the resulting image to the repository. For Mirage installs ($\text{Wine}_{\text{mirage}}$ and $\text{Email}_{\text{mirage}}$), the install is as described in Section 4.3 and consists of a memoizable part and a non-memoizable part.

optimized install gives a speedup of 2.3 on the first install; subsequent installs see a speedup of 21.

For both packages, selective retrieval and manifest composition are critical for performance because retrieving and publishing the images account for most of the traditional install time. For Wine, memoization is not very important: only 18 percent of the optimized install is memoizable. However, memoization is critical for Email: 89 percent of that install is memoizable.

Selective retrieval greatly reduces the amount of data that must be retrieved from the repository. With selective retrieval, installing Wine requires 12MB of data and installing Email requires 28MB of data. Without selective retrieval, both installs require the entire image.

6. Related Work

There are too many configuration-management systems for physical machines to mention here. In systems such as Debian [18] and the Comprehensive Perl Archive Network [4], a central repository stores software packages, including both the software itself and metadata describing how each package fits in to a complete sys-

tem. Programs like Cfengine [2] and Puppet [15] read high-level descriptions of a machine’s desired state and take action to move the machine to the desired state. Rather than replace packages or other high-level descriptions, in which much effort has been invested, this paper proposes changing the programs that operate on those descriptions to scale better to large numbers of images.

The configuration-management system Nix [6] obtains some of the benefits of virtual-machine images without relying on a hypervisor. Nix is a purely functional package management system that stores all installed components in a per-machine repository called the Nix store. Nix uses cryptographic checksums to ensure that different versions of the “same” component have different names in the Nix store. Each user selects a Nix profile, which determines the components visible to that user; profiles are implemented through symbolic links into the Nix store. Profiles are roughly analogous to virtual-machine images and Nix optimizes profile construction by not rebuilding or reinstalling components that a profile requires and which already exist in the store; this is similar to our memoization optimization. Unlike virtual-machine images, Nix profiles are not well-encapsulated, so Nix’s mechanisms for dealing with shared state such as the list of system users are awkward.

There are many systems for managing repositories of virtual-machine images. Ventana, a “virtualization aware” filesystem [13] replaces disk-image files with a distributed, versioned filesystem. This changes not just how an image’s files are stored but also how the image accesses those files. By contrast, our approach changes how images are stored but not how images access their files.

Moka5’s Engine [9, 17], its predecessor the Collective [3], and Microsoft’s Machine Bank [20] break machine images into disk blocks and store those blocks in a content-addressable store. This architecture enables images to be streamed from a server to a client and also saves space when updated images are added to the store by only storing new blocks. However, this architecture is not aware of files and so cannot support the memoization and search optimizations described in this paper.

VMware Lab Manager [22] stores a modified image as a differencing disk relative to a base image. Differencing disks are block-based, not file-based, and so Lab Manager also cannot support our optimizations.

Lutterkort and McLaughlin [8] distinguish between *appliance images* and *appliance recipes*. The former are disk-image files; the latter are instructions for constructing a software appliance at its site of use. An example of the recipe approach is rPath’s rBuilder [16], which allows appliance consumers to build unique

appliances by selecting from a menu of software components. Appliance recipes are smaller than appliance images, easier to distribute, and easier to modify but the cost of instantiation is high. Mirage's memoization optimization could be used to amortize the cost of instantiation.

PDS [1] (prior work by some of the authors of this paper) used a similar manifest and content-addressable repository as that described here. One difference is that, while PDS stored virtualized applications so that they could be streamed to unmodified Windows hosts, the repository in this paper stores entire virtual-machine images. Also, PDS did not address the configuration-management tasks which were the focus of this paper.

Other filesystems have used cryptographic checksums to support new filesystems. The first was Plan 9's Venti [14], which supported applications like backup and snapshots. More recently, the low-bandwidth filesystem [10] used checksums to achieve high-performance over slow networks.

CZIP [12] is a content-based naming compression scheme which also reduces data sizes by eliminating redundant chunks via content based cryptographic hash techniques. The paper mentions a use of this technique for compressing virtual machine images but does not address the configuration-management tasks addressed here.

7. Conclusion and Future Work

Disk-image files are an ideal format for managing the provisioning and execution of virtual machines, because the entire encapsulated state of the machine is kept as a single unit. However, such a representation is not well suited for the software management tasks that need to be performed on images when they are not executing.

Because virtual machine images are treated as data, they are easy to clone, extend, and snapshot, making virtual machine image sprawl a fast growing problem. As the number of virtual machine images that need to be maintained grows, the conventional disk image format for representing such images becomes cumbersome. The answer is not to constrain the creation of more virtual machine images, which is part of what makes these images so appealing relative to physical machines. Rather, we need to address the problem of maintaining large virtual machine image repositories in a way that makes virtual machine image sprawl manageable.

This paper presents the Mirage image format for virtual-machine images, which is optimized for storing large numbers of images and for performing tasks such as search, update, and compose without having to start up each image. The Mirage image format also allows images to be reconstituted as conventional disk images prior to execution, so adopting this format does not disrupt established production environments.

A key concept underlying the Mirage image format is the decoupling of the file name to file content (and file metadata) mapping. By representing file content with compact content descriptors, we split the mapping into two parts: an image manifest that precisely describes the image and a store that stores the contents of files. Such a design naturally exploits redundancies within and across images, enabling better storage scaling than conventional disk image formats.

In addition, the image manifest makes the image's files explicit, enabling many image operations to be performed merely by scanning or manipulating the image manifest. As demonstrated in this paper, substantial performance improvements for inventory control, customized deployment, and image upgrade tasks can be realized using this format.

The long term goal of the Mirage project is to build a scalable repository for efficiently storing and managing large numbers of virtual machine images. To this end, we have started automatically

indexing the daily builds of an IBM software product as fully configured virtual machine images.

The repository will provide utilities for efficiently manipulating these images. An example, which we are building now, is a versioning system that provides the functions typically provided by source-code control systems (check-in, check-out, update, merge, and so forth), but on virtual machine images instead of on source code.

References

- [1] B. Alpern, J. Auerbach, V. Bala, T. Frauenhofer, T. Mummert, and M. Pigott. PDS: a virtual execution environment for software deployment. In *Proceedings of the First ACM/USENIX International Conference on Virtual Execution Environments (VEE '05)*, pages 175–185, Chicago, IL, USA, June 2005.
- [2] M. Burgess. A tiny overview of cfengine: Convergent maintenance agent. In *Proceedings of the First International Workshop on Multi-Agent and Robotic Systems (MARS/ICINCO)*, Barcelona, Spain, Sept. 2005.
- [3] R. Chandra, N. Zeldovich, C. Sapuntzakis, and M. S. Lam. The collective: A cache-based system management architecture. In *Proceedings of the Second Symposium on Networked Systems Design and Implementation (NSDI '05)*, pages 259–272, Boston, MA, USA, May 2005.
- [4] Comprehensive Perl Archive Network. <http://www.cpan.org/>, Jan. 2008.
- [5] P. Deutsch. GZIP file format specification version 4.3. RFC 1952, Internet Engineering Task Force, May 1996.
- [6] E. Dolstra and A. Hemel. Purely functional system configuration management. In *Proceedings of the Eleventh Workshop on Hot Topics in Operating Systems (HOTOS XI)*, San Diego, California, USA, May 2007. USENIX.
- [7] IBM. IBM Tivoli License Compliance Manager. <http://www.ibm.com/software/tivoli/products/license-mgr/>, Jan. 2008.
- [8] D. Lutterkort and M. McLouglin. Manageable virtual appliances. In *Proceedings of the 2007 Ottawa Linux Symposium*, pages 293–302, Ottawa, Ontario, Canada, June 2007.
- [9] moka5. Engine. <http://www.moka5.com/>, Jan. 2008.
- [10] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 174–187, Banff, Alberta, Canada, Oct. 2001.
- [11] National Institute of Standards and Technology. *FIPS PUB 180-1: Secure Hash Standard*. Apr. 1995. Supersedes FIPS PUB 180 1993 May 11.
- [12] K. Park, S. Ihm, M. Bowman, and V. S. Pai. Supporting practical content-addressable caching with czip compression. In *Proceedings of the 2007 USENIX Annual Technical Conference (USENIX '07)*, pages 185–198, Santa Clara, CA, USA, June 2007.
- [13] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization aware file systems: getting beyond the limitations of virtual disks. In *Proceedings of the Third Symposium on Networked Systems Design and Implementation (NSDI '06)*, pages 353–356, San Jose, CA, USA, May 2006.
- [14] S. Quinlan and S. Dorward. Venti: A new approach to archival storage. In *Proceedings of the First USENIX conference on File and Storage Technologies (FAST '02)*, Monterey, California, USA, Jan. 2002.
- [15] Reductive Labs, LLC. Puppet. <http://reductivelabs.com/trac/puppet/>, Jan. 2008.
- [16] rPath. rBuilder. <http://www.rpath.com/rbuilder/>, Jan. 2008.
- [17] C. Sapuntzakis, D. Brumley, R. Chandra, N. Zeldovich, J. Chow,

- M. S. Lam, and M. Rosenblum. Virtual appliances for deploying and maintaining software. In *Proceedings of the Seventeenth Large Installation Systems Administration Conference (LISA '03)*, pages 181–194, San Diego, CA, USA, Oct. 2003.
- [18] Software in the Public Interest. Debian: The universal operating system. <http://www.debian.org/>, Jan. 2008.
- [19] Strace, version 4.5.8. <http://sourceforge.net/projects/strace/>.
- [20] S. Tang, Y. Chen, and Z. Zhang. Machine bank: Own your virtual personal computer. In *Proceedings of the Twenty-first IEEE International Parallel and Distributed Processing Symposium (IPDPS '07)*, pages 1–10, Long Beach, California, USA, Mar. 2007.
- [21] T. Y. Ts'o and S. Tweedie. Planned extensions to the linux ext2/ext3 filesystem. In *Proceedings of the USENIX 2002 Annual Technical Conference, Freenix Track (FREENIX '02)*, pages 235–244, Monterey, California, USA, June 2002.
- [22] VMware. Lab Manager. <http://www.vmware.com/products/labmanager/>, Jan. 2008.