Virtual machine images as structured data: the Mirage image library

Glenn Ammons, Vasanth Bala, Todd Mummert, Darrell Reimer, Xiaolan Zhang

IBM Research

The rise of infrastructure-as-a-service (IaaS) clouds, both private and public, has created new problems in managing large collections of virtual-machine (VM) images. VM images must be kept up-to-date with security patches and scanned for malicious or improperly licensed software. Because images are bulky, attention must be paid to the latencies of deployment and capture.

Large collections of VM images also present new opportunities which, while not completely realized today, do not exist with collections of physical machines. Users can search for images that contain the software they want, configured as they want it. Collections can be mined for patterns in order to answer questions like "which database management systems does our company use?" Images can be compared with one another.

This paper describes Mirage, an image library that addresses these problems and opportunities and is pluggable into various clouds. Mirage stores images in a format that indexes their filesystem structure, instead of as opaque disk images. Like other libraries, Mirage provides features for image capture and deployment. In addition, Mirage maintains a provenance tree that records how each image was derived from other images; allows operations, like patching and scanning, that normally require a VM instance to execute offline; and enables analyses such as image search and comparison.

Other papers have used Mirage [16, 13, 17] or described parts of the system [8]. This paper is the first overview of the complete system. It also describes three novel features that reduce the costs of translating between disk images and the Mirage format: a content-addressed store optimized for VM image data; a simple yet flexible indexer that supports a variety of filesystem and image types; and *delta deployment*, which uses precomputed deltas between images to speed up format conversions. Finally, we relate our experience with Mirage in the IBM Workload Deployer product (IWD) [3], which serves images to a customer's own private cloud, and in the Research Compute Cloud (RC2) [10], which is a production IaaS cloud.

1 Overview of Mirage

Mirage is more sophisticated than the image libraries typically used in IaaS clouds. In addition to classic functions like image browsing, access control and deployment, Mirage provides offline image content introspection, and manipulation capabilities. This is enabled by indexing the file system contents of an image when it is added to the Mirage library. The indexing process converts the image to a format that exposes the internal structure of the image, such as its partition layout, filesystem hierarchy, and digests of its files (see Figure 1). This file-aware format is a departure from the unstructured block-based disk representations used in other image libraries, including Moka5's Engine [5, 11], its predecessor the Collective [2], Microsoft's Machine Bank [15], and the Internet Suspend/Resume system [12]. The advantage of a file-aware format is that it naturally supports useful administrative services such as governance (who did what to which image when), software maintenance (offline security scan and certain kinds of patch operations), and analytics (versioning, comparison, search).

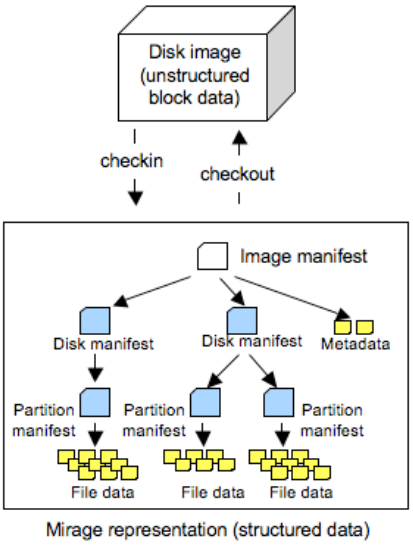

Figure 1: Mirage image format

Mirage provides a simple "checkin" and "checkout" interface to transfer conventional disk images into and out of the Mirage system. This interface is tuned for high performance in several storage environments (e.g. SAN/iSCSI/DAS with or without a clustered file system like GPFS/VMFS), with overhead roughly equivalent to a disk copy operation; some of our techniques for reducing translation costs are described in Section 2. The checkin/checkout interface makes it easy to integrate Mirage into an existing data center or cloud.

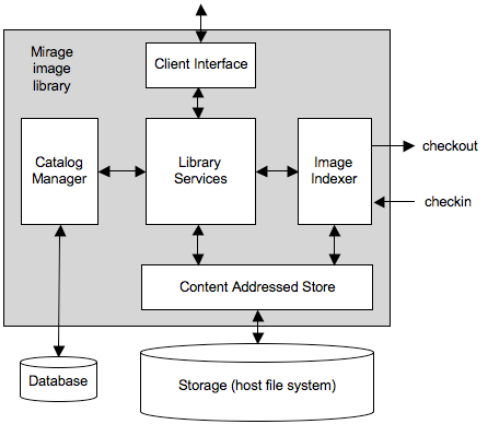

Figure 2: Mirage architecture

Figure 2 shows the components, described below, that make up the Mirage system.

Content-addressed store The content-addressed store (CAS) provides scalable, concurrent, garbage-collected storage for arbitrary data. A CAS identifies each item in the store by a cryptographically secure digest of its contents (Mirage currently uses SHA1 [6] to compute digests). Identical items have identical identifiers and therefore are only stored once. As the number of images in the store grows, storage requirements grow more slowly with Mirage than they do with libraries that store monolithic disk images; for example, in the IBM Research Compute Cloud (RC2), storage requirements are reduced by a factor of 5 (see Section 3). The storage de-duplication is done at the granularity of files instead of disks, which has the benefit of not incurring overheads for storing disk blocks that are unreachable from the file systems contained within the disk.

The semantics of a CAS are write-once, as an item cannot be modified without changing its identifier. Storing a modified item creates a new item. Mirage's version-control system exploits the write-once semantics of the CAS to retain old versions of images, although obsolete versions can be explicitly removed from the CAS, which makes them eligible for garbage collection.

Figure 1 shows the structure of an image as it is stored in Mirage. Each node of the structure is stored as a separate item in the CAS. The leaves are the contents of files and interior nodes list the files in a filesystem partition (partition manifests), describe a disk image (disk manifests), or describe a VM image (image manifests). Manifests reference other nodes by their CAS identifier.

As in a Merkle tree [4], the identifier of the root of the structure (an image manifest) securely identifies the entire structure. Mirage uses the image manifest's identifier as a serial number for the image, which fully determines the software, disk layout, and other metadata of an image.

Image indexer The image indexer converts, in both directions, between the disk images required by standard hypervisors and the Mirage image format. The design of the indexer addresses two technical challenges. First, the indexer must support a number of disk and filesystem formats. For this reason, the indexer has a plugin architecture; Section 2 describes a technique called hybrid indexing, that reduces the effort of developing filesystem plugins.

Another challenge is that converting a disk image to the Mirage format and back does not produce a bit-for-bit identical copy of the original. Because the indexer restores the image by creating one or more empty filesystems and then recreating the image's files, the restored file data can appear at different disk blocks than it did in the original. This confuses boot loaders that load data from particular disk blocks. One solution we use is for the indexer to reconfigure affected boot loaders at checkout-time. Hybrid indexing offers a more general solution (see Section 2).

We have also experimented with block-based representations of disk images. Block-based representations allow images to be reproduced faithfully but make fast file-level operations on disk images harder to implement, because filesystems generally do not allow applications to inspect or control the assignment of file data to disk blocks; see our earlier paper on the Mirage image format [8] for a discussion of how our file-aware format enables fast file-level operations.

Catalog manager The catalog manager keeps metadata about images in a relational database. For each image, this metadata includes the image's CAS identifier; the state of the image (either "active" or "deleted from the CAS"); the image's creation time; if the image was derived from another image, the identifier of the parent; and other fields.

The catalog manager also keeps version-control data. Version control is based on *named images*. A named image has a user-understandable name (as opposed to just a CAS identifier) and a mutable list of *image versions* called the *version chain*. Each image version is immutable and has an associated image and a version number. At any time, one image version is designated as the default version: this is the version used by default when users check out a named image. When a user creates a new version of a named image, the image version is added to the version chain and becomes the default version.

Access control operates at the level of named images. Access control lists specify who may check out the named image or check in new versions. Only the named image's owner or an administrator can alter the access control lists or mark images in the version chain as "deleted".

A user creates a new named image either by checking in a brand new image or by deriving from an existing named image. The creator is the owner of the new named image.

Unwanted images can be marked as "deleted", in which case the image's CAS data becomes eligible for garbage collection. However, in order to preserve provenance trees, the catalog never discards metadata.

Library services The library services component implements the user-visible and administrator-visible functions of Mirage. The user-visible functions include checkin, checkout, version control, virtual mount (see Section 2), and analytics functions (describing, listing, comparing, and searching images). The administrator-visible functions include starting or stopping the Mirage server, controlling garbage collection, locking selected images for maintenance, and other administrative tasks.

2 Mitigating translation costs

Because Mirage's image format differs from the formats that hypervisors require, Mirage imposes translation costs. The runtime costs of indexing filesystems and recreating disk images are obvious, but there are also the development costs of supporting a multiplicity of image formats and filesystem types. This section describes three ways that we reduce runtime costs — a structure-aware CAS, virtual mount, and delta deployment — and one way, hybrid indexing, that we reduce development costs.

The cost of recreating disk images can sometimes be avoided by caching images. Although Mirage does cache popular images, we focus here on optimizations that apply even when such a cache would have a low hit rate.

Structure-aware CAS Mirage stores indexed images in a CAS whose architecture is broadly similar to that of Plan 9's Venti [7] and Foundation [9]. Content items, which in Mirage are variably sized, are stored sequentially in large volumes. An on-disk index maps each item's content identifier to the item's location in a volume. In Mirage, volumes are sparse files and the on-disk index is a directory-based search tree whose leaves are symlinks that hold item locations. The index is slow but easy to update concurrently because creating symlinks is atomic.

Indexed images are highly structured: an image manifest references a disk manifests, each disk manifest references partitions, some partitions reference filesystem manifests, and each filesystem manifest references a large number of file-content items. This structure is communicated to the CAS; for example, when a filesystem manifest is added to the CAS, the CAS also receives a list of the file-content items referenced by the manifest.

Awareness of the structure allows the CAS to avoid almost all index lookups when retrieving an image. This is an important optimization because index lookups, which are essentially random, are costly for any on-disk index [9] and especially costly for our simple implementation. When an item that references other items is added to the CAS, the CAS creates and stores a location table that lists the locations of the item's references. Later, when a client fetches an item from the CAS, it tells the CAS if it intends to follow that item's references. If so, the CAS reads the item's location table and caches the locations of all referenced items in memory, so that subsequent lookups avoid the on-disk index.

Foundation has a similar optimization, but Foundation associates lookup tables with content volumes instead of with items. For image retrieval, our scheme has the advantage: the indexer can tell the CAS exactly which items will be fetched next and the CAS can cache the locations of exactly those items; by contrast, Foundation would cache locations for items that will not be fetched. On the other hand, Foundation's scheme is more robust when fetches do not follow the structure closely.

Structure-awareness has advantages beyond reducing translation costs. The CAS uses the structure information to garbage-collect obsolete images. Also, a generic "push/pull" program copies images and other data structures between CAS instances by walking the structure; this is useful for backup and for increasing availability.

Virtual mount Many operations on images, such as offline patching, do not require entire disk images. Virtual mount allows users to mount an image's filesystems without reconstructing the image's disks. A FUSE [14] daemon fetches items from the CAS on-demand. Writes are buffered in a scratch area on disk; reading a special file attribute flushes the buffered writes to the CAS, creates a new filesystem manifest, and returns the new manifest's identifier. Another special file attribute exposes operations on the filesystem manifest: reading this attribute returns the identifier of the file's contents and writing an identifier to this attribute atomically replaces the file's contents. Depending on the patch, Virtual mount yields speedups between 2 and 8.5 for offline patching [17].

Delta deployment There is often a great deal of commonality among VM images in the library, in part because most images are produced by modifying existing images. Delta deployment reduces disk-recreation costs by exploiting this similarity.

To prepare for delta deployment, a preprocessing step computes file-level deltas between pairs of similar disk images. A file-level delta lists files that should be added, modified, and deleted in a source image in order to produce a target image. For each added or modified file, the delta specifies the file's target metadata and, if the file is a regular file, the identifier of its target contents.

As noted above, Mirage keeps a cache of popular disk images. Normally, when a disk image is retrieved, Mirage consults the cache: if the image is in the cache, a clone is returned immediately; otherwise, the image must be reconstructed. Delta deployment behaves differently on cache misses. If the target image is not in the cache, the cache is searched for a source image for which the preprocessing step computed a delta that produces the tar-

get image. If such a source image exists, it is cloned, the delta is applied to the clone, and the clone is returned. The cost of applying the delta is proportional to the size of the delta, not to the size of the target image.

Deltas that involve deleting files create an opportunity and a security problem. Normally, deleting a file is a cheap operation because it changes the filesystem's namespace without overwriting the file's data blocks. By populating the cache with "superimages", which do not correspond to any image in the library but do contain large numbers of popular files, and precomputing deltas between superimages and real images, we can improve the odds of being able to deploy via a delta without much increasing the cost of applying deltas.

The security problem is that the target image will contain data blocks that belonged to files that were deleted from the source image. That is, information leaks from the source to the target. Because of this problem, Mirage applies deltas only when the user has permission to read both the source and the target image.

Hybrid indexing Indexing refers to the process of checking an image into the image library. Mirage supports a variety of filesystems, including NTFS, Ext2 and its successors, Reiserfs, and AIX's Mksysb. Most image library implementations use a simple image indexing technique that stores the image disk as an array of fixed size disk blocks. A benefit of this approach is not having to deal with the complexity of extracting individual files from the disk and later re-assembling them when the image is checked out. But the drawback of such an approach is that access to the filesystem contents requires mounting each of the image disks (to access the filesystem), every time the contents of an image need to be scanned, queried, or manipulated.

Hybrid indexing is aimed at getting the best of both worlds: access to individual files within an image without having to re-assemble or mount it, but without the complexity of dealing with the idiosyncracies of every filesystem we may encounter in customer environments. It lowers the development costs of supporting a broad range of filesystems by offloading much of the work to off-the-shelf backup and restore tools while still indexing the filesystem structure.

Hybrid indexing processes a filesystem partition as follows. First, a clone of the partition is mounted. The indexer walks the filesystem and adds each regular file it finds to the CAS, recording the file's identifier in a filesystem manifest. The indexer walks the filesystem again, this time truncating every regular file to size zero, taking care not to disturb modification times. The result is a *filesystem skeleton*. The skeleton is then backed up with an off-the-shelf, filesystem-specific program, the backup is added to the CAS, and its identifier is recorded in the manifest. Finally, the manifest is added to the CAS and its identifier is returned.

Retrieval reverses the process. The skeleton is restored with an off-the-shelf, filesystem-specific program. The indexer walks the filesystem and restores the contents of each regular file, taking care not to disturb modification times.

Note that, because the backup and restore programs faithfully record and reproduce all filesystem metadata, the indexer need not understand the special features of each filesystem (such as extended filesystem attributes in Ext3 or directory junctions in NTFS). Also, if a certain file's data blocks should not be moved (perhaps because the boot loader expects them at a certain location), then that file can be added to a list of files that should not be truncated. The backup and restore programs will ensure that its blocks are restored correctly.

The Mirage image format normally records filesystem metadata in the filesystem manifest, which makes it accessible to analyses and tools. With hybrid indexing, most metadata is now in the backup of the skeleton. Fortunately, skeleton backups are about the same size as filesystem manifests; tools that require metadata can get it cheaply by restoring this backup.

3 Experience

IBM Workload Deployer The Mirage image library is used in the recently announced IBM Workload Deployer (IWD) product, which is an enterprise cloud management appliance [3]. Mirage provides a central image library that is pluggable into the hypervisor platforms that customers already have in their data centers. The use of Mirage in these environments provided interesting lessons and insights.

A first lesson is the importance of transport costs. Although the diversity of hypervisor platforms poses many technical challenges, the main issue is that the image library and the hypervisors do not necessarily share storage. Potentially, each checkout moves a multigigabyte image between library-reachable and hypervisor-reachable storage. This adds latency and can overwhelm the available network infrastructure. These problems are especially acute in development and test environments, where checkouts are more frequent than in production environments.

Fortunately, caching and delta deployment can lower transport costs as well as translation costs. The key is to cache images on hypervisor-reachable storage. This is tricky in some environments and delta deployment requires that some Mirage code run in the customer's data center, but the payoff is worth the complications.

A second lesson is the need for collaborative image development. Enterprises often split IT responsibilities among several teams. For example, at one customer, one team maintains the operating system, another handles middleware, and a third installs applications. Currently, only the operating system team produces images.

To avoid a hard-to-manage profileration of images, the other teams work on running instances: if the operating system team releases a new image, the other teams must create a new instance and reconfigure it from scratch.

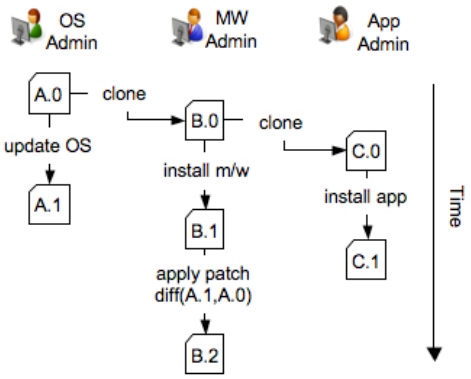

Figure 3: A version control scenario

The version control features of Mirage can extend the benefits of image-centric management to all three teams. Figure 3 illustrates our approach. The OS team creates an image A.0 (image A, version 0) and shares it with the middleware team. The middleware team clones A.0 to create B.0 (a private copy), checks out an instance of B.0, installs and tests the middleware, then checks in the resulting image as B.1. The application team follows a similar process to create image C.1.

Mirage tracks the links between these images, which makes updates more manageable. If the OS team were to update A.0 to A.1, the middleware team can be notified. Using Mirage, the middleware team can compute a file-level delta between A.1 and A.0, create B.2 by applying the delta to B.1, test B.2 to verify that the patch worked properly, and finally release B.2 to the application team.

RC2 Mirage serves as the image manager for the IBM Research Compute Cloud (RC2) [10], a production cloud used by IBM researchers and other employees worldwide. From its general release in early April 2009 to early March 2011, RC2 served 458 unique users. The image repository had grown to 4890 images, of which 2170 were active (available for use).

RC2 does not currently expose Mirage's version control facility to users of the cloud - it is only for internal administrative use in situations like image migration, which we elaborate on later. However, there is evidence that version control would be welcome for some cloud users: inspection of the provenance tree shows that some users roll their own version control system by embedding version numbers in image descriptions. However, three-fifths of all images derive directly from a base image; as images are immutable, this implies that most images are created once and never updated. Note that users are encouraged to store data (as opposed to software and configuration information) outside of images, for which purpose RC2 provides block-level storage volumes similar to Amazon's EBS volumes [1]. The next release of RC2 plans to expose the Mirage version control features to cloud users in addition to cloud administrators.

The repository occupied 7.29 TB of storage, an average of 1.5 GB per image (active or inactive). We checked out a sample of active images and found that, on average, checked-out images occupy 8.9 GB of disk space. The fivefold compression comes from CAS deduplication and from not storing disk blocks that are unreachable from an image's filesystems. Note that RC2 does not run garbage collection, so inactive images also consume space.

However, the main benefit of Mirage for RC2 is in maintaining images, not in saving space. Images are not static objects; like physical machines, they require periodic maintenance. The following example shows how RC2 used virtual mount and version control to migrate their entire image collection from one hypervisor to another.

After a year of operation, RC2 decided to switch x86 hypervisors from Xen to KVM. Because RC2 is a production system, the conversion needed to cause minimal disruption to operations and to users. The goals were to have no downtime, to minimize resource use, and to be transparent to users. Ideally, users would not notice that their images had been migrated from Xen to KVM.

The migration mounted each Xen image with virtual mount, converted it to run on KVM, and saved the result as a new image. A total of 419 Xen images were converted to KVM. During this period, RC2 ran with full functionality and no noticeable performance degradation.

It took several tries to convert some images. Unexpected kernel versions, missing drivers, and other bugs caused conversions to fail; after each failed conversion, bugs were fixed and the conversion was rerun. Version control was very useful in this migration scenario: each patch created a new version (which could be analyzed) without replacing the original. In addition to the rollback capability, understanding what files were added, modified and deleted between any pair of versions was valuable in troubleshooting failures.

A total of 1120 images were created during the conversion period (419 converted images, 355 failed conversions, and 346 user-created images). Thanks to the CAS, the repository grew a modest 293 GB. Storing each image as disk images would have required 9.5 TB (8.5 GB average image content * 1120).

4 Conclusion

We presented Mirage, an image library that, by storing images in a format that indexes their filesystem structure, speeds up image deployment and supports advanced fea-

tures such as version control, fast offline operations, efficient search, image comparison, and other analyses of images. Experience with deployments shows that this view of images as structured data has real-world value.

References

[1] AMAZON. Amazon Elastic Block Store (EBS). `http://aws.amazon.com/ebs/`.

[2] CHANDRA, R., ZELDOVICH, N., SAPUNTZAKIS, C., AND LAM, M. S. The collective: A cache-based system management architecture. In *Proceedings of the Second Symposium on Networked Systems Design and Implementation* (May 2005).

[3] IBM. IBM Workload Deployer. `http://www-01.ibm.com/software/webservers/workload-deployer/`.

[4] MERKLE, R. C. Protocols for public key cryptosystems. In *Proceedings of the IEEE Symposium on Security and Privacy* (1980), pp. 122–133.

[5] MOKA5. Engine. `http://www.moka5.com/`, Jan. 2008.

[6] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. *FIPS PUB 180-1: Secure Hash Standard*. Apr. 1995.

[7] QUINLAN, S., AND DORWARD, S. Venti: A new approach to archival storage. In *Proceedings of the FAST '02 Conference on File and Storage Technologies* (2002), pp. 89–101.

[8] REIMER, D., THOMAS, A., AMMONS, G., MUMMERT, T., ALPERN, B., AND BALA, V. Opening black boxes: Using semantic information to combat virtual machine image sprawl. In *VEE '08: Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (2008), pp. 111–120.

[9] RHEA, S., COX, R., AND PESTEREV, A. Fast, inexpensive content-addressed storage in foundation. In *Proceedings of the 2008 USENIX Annual Technical Conference* (2008), pp. 143–156.

[10] RYU, K. D., ZHANG, X., AMMONS, G., BALA, V., BERGER, S., DA SILVA, D. M., DORAN, J., FRANCO, F., KARVE, A., LEE, H., LINDEMAN, J. A., MOHINDRA, A., OESTERLIN, B., PACIFICI, G., PENDARAKIS, D., REIMER, D., AND SABATH, M. RC2 - A living lab for cloud computing. In *Proceedings of the 24th International Conference on Large Installation System Administration* (2010), LISA'10, pp. 1–14.

[11] SAPUNTZAKIS, C., BRUMLEY, D., CHANDRA, R., ZELDOVICH, N., CHOW, J., LAM, M. S., AND ROSENBLUM, M. Virtual appliances for deploying and maintaining software. In *Proceedings of the Seventeenth Large Installation Systems Administration Conference (LISA '03)* (San Diego, CA, USA, Oct. 2003), pp. 181–194.

[12] SATYANARAYANAN, M., GILBERT, B., TOUPS, M., TOLIA, N., SURIE, A., O'HALLARON, D. R., WOLBACH, A., HARKES, J., PERRIG, A., FARBER, D. J., KOZUCH, M. A., HELFRICH, C. J., NATH, P., AND LAGAR-CAVILLA, H. A. Pervasive personal computing in an internet suspend/resume system. *IEEE Internet Computing 11*, 2 (March/April 2007), 16–25.

[13] SATYANARAYANAN, M., RICHTER, W., AMMONS, G., HARKES, J., AND GOODE, A. The case for content search of VM clouds. *Computer Software and Applications Conference Workshops 0* (2010), 382–387.

[14] SZEREDI, M. Fuse: Filesystem in userspace. `http://fuse.sourceforge.net/`, 2010.

[15] TANG, S., CHEN, Y., AND ZHANG, Z. Machine bank: Own your virtual personal computer. In *Proceedings of the Twenty-first IEEE International Parallel and Distributed Processing Symposium (IPDPS '07)* (Long Beach, California, USA, Mar. 2007), pp. 1–10.

[16] WEI, J., ZHANG, X., AMMONS, G., BALA, V., AND NING, P. Managing security of virtual machine images in a cloud environment. In *Proceedings of the 2009 ACM Workshop on Cloud Computing Security (CCSW '09)* (2009), pp. 91–96.

[17] ZHOU, W., NING, P., ZHANG, X., AMMONS, G., WANG, R., AND BALA, V. Always up-to-date: scalable offline patching of VM images in a compute cloud. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC '10)* (2010), pp. 377–386.