Logical Image Migration Based on Overlays

Joe Meehean
Greg Quinn

Technical Report #1564

June 2006

Logical Image Migration Based on Overlays

Joe Meehean and Greg Quinn
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706
{jmeehean|gquinn}@cs.wisc.edu

May 12, 2005

Abstract

Virtual machine technology is becoming an increasingly popular vehicle for enabling process and service migration. Many migration frameworks rely on a distributed file system to avoid worrying about the burden of migrating a large virtual disk. While this assumption is valid in some contexts, there are other environments where it is not feasible. In these cases, disk migration becomes the primary bottleneck for achieving efficient migration. We discuss the design, implementation, and evaluation of LIMBO, a system for efficient migration of VM local file systems. We've found that overlay-based migration using a copy-on-write block device can significantly improve migration costs while imposing a small amount of overhead on file system operations.

1 Introduction

Process migration has been an active topic in operating systems research for several decades. Until recently, most of this research has focused on enabling support for migration within the operating system. An alternative approach, which is now receiving a lot of attention, is to enable migration by executing a system within a virtual machine (VM).

The virtual machine approach has several advantages. The main challenges that a system must consider in order to enable migration are those of resource naming and handling a process's external dependencies. A virtual machine ensures that naming conflicts cannot occur, as resource names like process IDs are private to a particular VM. In addition, migrating a process along with its enclosing virtual machine significantly reduces the types of external dependencies that must be dealt with when migrating. For example, IPC mechanisms like shared memory or pipes do not present problems since both endpoints are migrated together. An additional advantage of using virtual machines for migration is portability. Migrating individual processes will require that the migration target is running a sufficiently similar software environment to the source machine in order for the migration to succeed. Migrating a virtual machine, however, will require only that the migration target has the same hardware architecture, a significantly more relaxed constraint.

VM-based approaches for problems like migration are likely to continue to gain momentum in coming years. Hardware technologies are in development that will provide support for virtualization at the CPU level [11, 1]. Furthermore, newer systems like Xen [12], based on the idea of paravirtualization, reduce the runtime cost of executing inside a VM by placing virtual machine monitor awareness in the guest OS kernel.

Process migration has been suggested as a mechanism for several purposes, including fault tolerance, improving network locality, load balancing, and user mobility. It is important to note that a single migration strategy may not be suitable for all of these purposes. For example, the VM approach has many nice properties as described above, but may not be a realistic option for achieving load balancing. The way in which a migration scheme will be used should dictate the set of assumptions the scheme is based on. In particular, many migration systems assume file system migration can be handled by mandating the use of a distributed file system [19, 14]. This assumption disallows migration in many circumstances, and may result in unacceptable performance in others. Specifically, the distributed file system assumption is not well suited to the mobile user and grid computing environments where a process may migrate beyond the efficient operating range of many distributed file systems.

The idea of using virtual machine-based migration to allow mobile users to access their personal computing environments anywhere is described in Internet Suspend/Resume [14]. This work breaks the bond between a user's customized environment and a particular piece of hardware. With such a wide range of possible migration targets, it is unlikely that a distributed file system will always be available.

Grid computing is another area in which migrating virtual machines are becoming useful. Virtual machines make it possible to create a sandbox to protect a system from any untrusted grid jobs it may be running. Virtual machines are also useful for grids because they allow a user to specify a well-defined environment in which a job will be executed.

Our goal is to facilitate efficient migration in a wider range of environments by dealing with situations where the ability to use a distributed file system cannot be assumed. Removing this assumption presents an imposing challenge. In a typical VM configuration, virtual disk size is at least an order of magnitude larger than main memory. In this paper, we describe and evaluate the LIMBO system, which reduces the amount of data that must be transmitted when migrating a virtual machine's local file system. In Section 2, we present properties of file system data that can be leveraged in achieving our goal. Section 3 presents the LIMBO architecture. In implementing LIMBO for Linux, we've taken advantage of some existing technologies, which we detail in Section 4. Section 5 describes our implementation, which we evaluate in Section 6. Section 7 describes related work, and Section 8 concludes.

2 Leverage Points

There are two leverage points that allow for optimization when migrating a large system image. First, typically a large portion of a disk image is comprised of standard operating system and application files (Section 2.1). Second, a user is likely to migrate within a particular subset of machines (Section 2.2).

2.1 Image Commonality

A typical disk image contains a large number of files that are common to many users. This includes operating system files, application binaries, libraries, and documentation. In a typical Linux installation this includes the files located in the `/bin`, `/sbin`, `/lib`, `/boot`, and portions of `/usr` directories [20]. Many distributed computing environments take advantage of this commonality by placing binaries and libraries in a shared distributed file system directory.

If all users shared the same unmodified operating systems and applications, full VM migration would be unnecessary. A much simpler solution would be provided by installing this common image on all machines and using process migration frameworks such as Zap [19]. Users, however, do not perform the same functions uniformly across all desktop systems. Even between software engineers on the same project one may prefer `emacs` and while another may use `vi`. Typically a user will customize their particular system installation to maximize the efficiency of their given work. A dedicated `emacs` user may have a `emacs` configuration file that is tens of KB. Additionally, some users require special applications or libraries. For example, a user wishing to build the Xen source code would require the development version of the `libcurl` library. In a typical Linux installations these customizations are sprinkled across the previously mentioned directories as well as populating directories created specifically for customization such as `/etc`, `/usr`, and `/home`.

Much of this customization is encapsulated in relatively small configuration files and updated application binaries. These customizations generally account for a relatively small portion of the total disk size. A solution to the problem of migrating system images should exploit the large commonality of system images while still allowing user customization.

2.2 Migratory Patterns

Similar to migratory birds, the mobile user is likely to migrate between a small set of locations. Typically, this will include the user's work desktop, home desktop, and laptop computers. The reduction in the number of probable migration hosts allows these hosts to perform prefetching and caching without a large penalty in either storage or bandwidth. However, users may also migrate to machines they rarely use. Teaching assistants may work in a common lab, rather than in their offices, to answer student questions. Similarly, many software development companies provide "bull pens" where developers can temporarily work together in small groups. Without careful design, prefetching and caching become difficult, costly, and ultimately useless at unlikely migration hosts.

A mobile user migrating to an uncommon place should not increase the overhead of migration to a degree that makes it undesirable. However, a good solution should also take advantage of the likelihood that the user will migrate within a particular subset of machines.

3 Architecture

In this section we discuss how image commonality and migratory patterns can be leveraged to develop an efficient architecture in both storage space and migration time. We present several design alternatives to illustrate the benefits of our technique. Section 3.1 discusses how an immutable disk image can facilitate prefetching, while Section 3.2 presents a range of techniques for encapsulating user configuration and customization. Section 3.3 provides an overview of our architecture for efficient virtual machine file system migration.

3.1 Pristine Image

By analyzing the leverage points discussed in the previous section one can see that it is possible to capture a majority of the disk contents in a shared *pristine* disk image that contains the operating system and standard applications. Our definition of pristine image implies that any two instances of the same pristine image are exactly the same. This allows a pristine image to be safely prefetched and cached at the likely migration points because it is guaranteed not to change. Additionally, the overhead of prefetching and caching for a subset of probable users is reduced because several users may share the same pristine image.

Throughout this paper we refer to *the* pristine image, however, there may be several. For maximum flexibility there may be single pristine image for each operating system and application working set. For example within the same company there may be a *Fedora2 with Standard Utilities* image for software developers, a *Windows XP with Microsoft Office* image for management, and a *RedHat Enterprise Linux 3 with IBM's WebSphere* for the web servers. Two issues arise when dealing with multiple images. First, each image must be assigned a unique name to prevent a host from fetching the wrong image. Second, the benefits of pristine images are reduced as the number of images approaches the number of users. We do not explicitly deal with either of these issues in this paper.

3.2 Customization Encapsulation

Pristine images do not solve the problem of migrating user customizations or configurations, which is key for functional VM migration. These changes must be captured using a separate mechanism. If we can assume that there is a mechanism to encapsulate these changes and that the destination host has already prefetched the pristine image, when a particular user migrates to a host only the user's changes must be retrieved. In the case where a user migrates to a machine that does not have a copy of the correct pristine image, the image does not have to be fetched from the same site as the user's changes. When using VM migration for mobile computing, pristine images may be burned to CD or DVD and carried with the user. When migrating a VM in a grid environment an image server can be set up in each grid to efficiently distribute pristine images at runtime. The network connection between a grid computing resource and a grid image server will likely have better bandwidth and lower latency than the connection between the grid resource and the submission site.

An issue of using pristine images and a set of changes for virtual disk migration is that encapsulating customizations cannot continue indefinitely because the change representation may grow as large as the disk image. Some technique must be devised for merging changes back into a pristine state to prevent migrating customizations from becoming as cumbersome as migrating entire disk images.

Removing the assumption that there is some mechanism for encapsulating changes to a pristine image, we must look at possible solutions. We detail several techniques in the following sections to give the reader an understanding of the advantages of our technique.

3.2.1 System Scan

The most straight forward approach to capturing modifications is to determine what the user has modified after the fact. Each user begins with an instance of a pristine system image which they may modify. During migration the user's modified image is scanned and compared to a pristine version. The differences can be captured in a redo log and sent to the destination site. These changes are then reapplied to a pristine instance at the destination site. The fundamental disadvantage to this approach is that it is computationally expensive to scan an entire disk image.

3.2.2 Change Log

Improving upon the previous approach, it may be possible to record the changes made by the user when they occur. These changes can be captured with small overhead by appending to a log. At migration time no computation is required at the source machine. The log is sent to the destination host and replayed there to bring the instance of pristine image up to date. For simple, non repetitive changes this approach may work well. However, users often truncate a file and rewrite it differently or delete a file and replace it with one that has the same name. An example of this is a user who incrementally updates some software installation several times over the course of weeks. Capturing this work flow in a log would include capturing file contents that were later overwritten. Logs containing overwritten data are inefficient in both time and space, as they can be large and the entire log must be replayed at the remote site. To help ease this problem, a log can be compressed to remove wasted entries before the log is sent to the destination host. However, this compression increases the computational penalty for migrating at the source host.

3.2.3 Distributed File System

An obvious improvement on the log approach is to use write-back caching to dispense with the need for a log altogether. Many distributed file systems already provide a mechanism for performing write-back disk caching to aggregate the number of updates send to the remote server. When a given file page is written, it is marked dirty so that it can eventually be flushed to the server. A distributed file system meshes well with many of the requirements of virtual disk migration. Distributed file systems can perform prefetching, long-term caching, and provide a single access point for fetching disk images. Leveraging the functionality already provided by distributed systems is tempting as it simplifies implementation of customization encapsulation. Using an technique built on a distributed file system, a system image can be prefetched from the file server, the VM can run over the distributed file system client with little overhead, and only changed portions of the image will be migrated back to the server. An additional optimization could be added to stream back modified pages at a low bandwidth while the VM is still running. The primary drawback to this approach is that distributed file systems do not easily integrate with the abstraction of an image linked with a set of changes. The distributed file system cache allows us to determine which parts of the pristine image have been changed but still leaves the problem of determining how to efficiently represent these changes.

The AFS [7] distributed file system has a mechanism for creating backups that may be leveraged to tackle the problem of representing changes. This mechanism creates a read-only backup copy of a given AFS volume (file hierarchy). This backup volume does not store an actual copy of the data, but rather stores pointers to data that is in the original volume. When the original volume is updated, the backup volume maintains the pointers to the old data and therefore remains unchanged. The backup is essentially a copy-on-write volume that copies the old data only when new data is written. One could create a pristine image by making a backup of a system image. All modifications must be made to the original volume because the backup is read only. This implies only a single modifiable image for given pristine image, although there can be several backup images for a single modified image. Unfortunately, this is relationship is opposite of what is required. Ideally, a single pristine image might have multiple changed representations, one for each user.

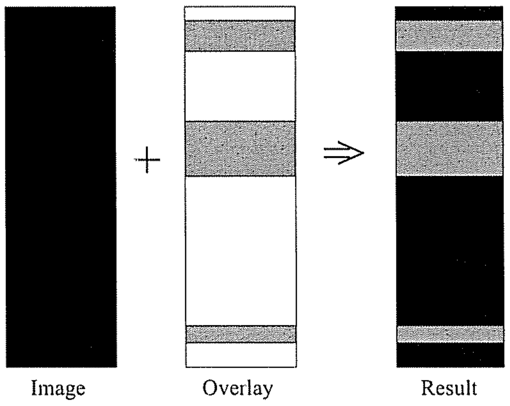


Figure 1: *Illustration of images and overlays.*

3.2.4 Copy-on-Write Overlays

Although AFS backup volumes do not sufficiently solve the problem of capturing modifications to a pristine image, it does provide an interesting abstraction. AFS allows the pristine image to occupy a small amount of space by sharing data with the modified image until a write occurs. Additionally, AFS presents the backup as a full image to the user. From the AFS abstraction we move naturally to the idea of images and overlays. In a reversal of how AFS provides sharing, an implementation of image and overlays should have the pristine image maintaining the unmodified pristine data. The overlay will share this data until a modification is made, at which time, the modified data is stored in the overlay. The overlay is presented to the user as a stand-alone file system when in reality it is only the changes made to the pristine image, see Figure 1. Unlike AFS backup volumes, this allows many overlays to be associated to a single pristine image. Each overlay maintains a single user's modifications to a pristine image.

An overlay must contain not only the changed data, but also metadata describing where in the pristine image this changed data belongs. In this regard the overlay is similar to a compressed log with no overwritten data. The advantage overlays have over logs is not in data, but in the layering of an overlay on top of an image that allows changes to be written in place as opposed to appended to a log.

Another advantage of overlays is the ability apply them recursively, creating a tiered architecture. For example a student may capture the changes made to a pristine image for an entire semester into a pristine overlay. The next semester the student could start with a fresh, empty overlay layered on top of the pristine overlay from the previous semester, layered on top of the pristine image. A multi-tiered approach allows the set of new changes that have stricter consistency requirements to be maintained in a smaller subset. The larger set of changes captured by a pristine overlay can be prefetched and cached indefinitely since it is immutable. In this way overlays can more efficiently capture the relatively large `/home` directory. Pristine overlays may also allow a set of users to share a modification to a pristine image, such as an update to a standard application or library.

Without breaking the abstraction, overlays may be implemented at either the file or block level. There are advantages and disadvantages to a block level implementation of overlays. The primary advantage is simplicity; a block level implementation is not required to perform any complicated manipulation of file system metadata. Further, recording changes at the block level does not bind an implementation to a particular file system, maximizing its generality. The primary disadvantage to a block level implementation is that it is difficult to determine if a given modified block has been deallocated by the file system. Therefore, block level overlays may introduce added overhead by migrating blocks that are no longer in use. To prevent this problem one could add an explicit `free block` command to the device or implement gray-boxing techniques at the device level to determine which blocks are free [22].

3.3 LIMBO

We chose to use an image and overlay technique as we feel that it best meets the requirements of mobile users and grid computing. We have named our technique for migrating VM disk images *Logical Image Migration Based on Overlays* or LIMBO. In LIMBO the pristine image is not viewable by the user except through an overlay, even if the overlay contains no changes. This allows us to ensure that all changes are correctly recorded. Additionally, we use a block level implementation of overlays because we want to support multiple pristine images regardless of the image's file system. Unfortunately, we were not able to explore multi-tiered overlays because the underlying mechanism for layering an overlay on an image does not support recursive layering.

4 Enabling Technologies

In our implementation of an overlay-based approach to disk migration, we require a level of indirection on top of the block device that will store the actual data. This provides hook for intercepting attempts to change the data on disk and thus allows for a copy-on-write block device. Such a CoW block device already exists in Linux as part of the Device-Mapper (DM) and Logical Volume Manager (LVM) systems. We detail these essential components to our LIMBO implementation in this section.

4.1 Device Mapper

Linux's DM is a kernel subsystem that allows for the creation of logical block devices. Once a logical device is enabled, a file system may be built on top of it and the file system code can treat it in the same way as a physical block device. However, any time an I/O request is made using the logical block device, the DM may modify the request as required by the semantics of the logical device. Examples of simple functionality enabled by DM include disk striping and mirroring.

Creation and management of logical devices is done through a logical device driver represented by `/dev/mapper/control`. User code can make `ioctl` system calls to this device file directly, or use higher-level interfaces provided by the `libdevmapper` library or `dmsetup` command-line utility.

A logical device is specified and represented by a table, with each entry mapping a contiguous region of logical disk blocks to a DM *target*. A target is a kernel module that defines the translation of requests on a logical device into requests on the underlying physical device(s).

Table 1 shows an example DM table. The logical device represented by this table uses two physical disks for its underlying storage. The disk sizes are 80GB and 60GB, and combined to form a 140GB logical disk. The first table entry maps the first 120GB of the logical device to the *striped* target. The effect is that the first 120GB of the logical disk are striped across both physical hard drives to improve performance. The last 20GB of the logical device are

```
0 251658240 striped 2 32 /dev/hda 0 /dev/hdb 0
251658240 83886080 linear /dev/hda 125829120
```

Table 1: *A DM table showing the use of some simple targets. The first two numbers on each line show the starting logical block and number of blocks, respectively, for that entry. The third token is the name of the target that will handle those blocks. The rest of the line consists of arguments specific to the the given target.*

```
vg-lvsnap:  0 1024000 snapshot 254:1 254:2 P 16
vg-lv:  0 1024000 snapshot-origin 254:1
```

Table 2: *DM tables for a snapshot device and it's origin. The arguments to the snapshot target include: (1) The origin block device major and minor numbers, (2) the exception store device numbers, (3) "P" to designate a persistent snapshot, and (4) the chunk size. The snapshot-origin target takes the original device's numbers as its only argument.*

mapped by the second table entry to the remaining part of the first hard drive, using the basic *linear* target.

The specific target that we leverage in LIMBO is the *snapshot* target. The original motivation for the snapshot target is to make backups of a system while it continues normal operation. A system administrator can create a snapshot of a logical device prior to any backup operation. This snapshot device will not reflect any changes that are made to the original device, referred to as the origin, after the snapshot has been made. This ensures that the backup operation captures a consistent state of the file system.

Table 2 shows two example logical device tables that represent a snapshot device and its associated origin. A logical device table is needed for the origin for cases where a block that has not yet been copied to the exception store is written in the original. The physical disk that underlies a snapshot logical device is referred to as an *exception store*. An exception store only stores the contents of blocks that have been modified on the original device since the snapshot was created. Thus, the exception store behaves like a copy-on-write block device and stores significantly less data than the original device, provided the set of changes is relatively small.

When a read request is made to a snapshot logical device, metadata in the exception store is consulted to determine if the requested block is present. If it is, the data is retrieved directly from the exception store. If not, the request is forwarded to the origin because the data has not been modified. The solid lines in

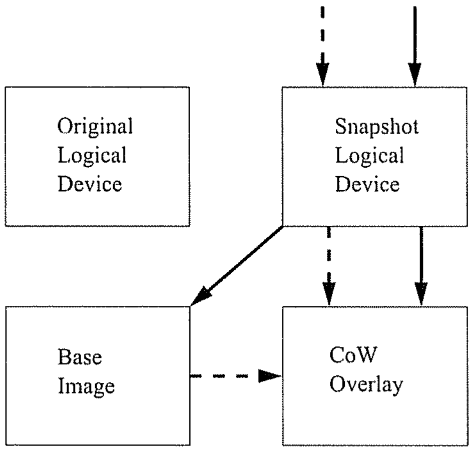


Figure 2: *Data paths for snapshot operations. Solid lines indicate reads, dashed represent writes*

Figure 2 show the possible data paths on a snapshot read request.

The snapshot target also implements *writable snapshots*, which means that the snapshot logical device does not have to be immutable and thus snapshots are suitable for uses other than backup. Like read requests, snapshot write requests consult the exception store metadata to see if the block is present. If it is present, the write is executed in the obvious fashion. If the block is not present, it is first copied from the origin device into the exception store, and then the write proceeds. It is this writable snapshot capability that enables us to treat an exception store as an overlay on top of a larger pristine image. The write data paths for a snapshot are depicted as dashed lines in Figure 2.

There are two main sources of overhead when using DM snapshots for normal file system activity as required by LIMBO. The first is the mapping function that occurs on any access to a snapshot device. The exception store metadata that maps exception blocks to blocks in the original device is read into a hash table in memory when the snapshot is activated. We do not anticipate the cost of these lookups to be prohibitive.

The other source of overhead is the copy operation that must occur when a block is modified for the first time. These copies are performed by a kernel thread, `kcopyd`, that is created by DM. Rather than copying the modified block, a larger unit of granularity called a *chunk* is transferred. Chunk size is a configurable parameter during snapshot creation, which defaults to 8KB (16 blocks).

4.2 Logical Volume Manager

As mentioned in the previous section, DM is a kernel-level system with a low-level, table-based interface. DM state resides entirely in memory and is thus not persistent across machine reboots. Persistence and a higher-level user interface are provided by a user-level system called the Logical Volume Manager (LVM).

The first-class objects in LVM include volume groups (VGs), physical volumes (PVs), and logical volumes (LVs). Volume groups represent pools of allocatable disk space. This space is provided by some number of physical volumes, which are physical disks or disk partitions formatted for use with LVM. Logical volumes appear in the system as logical block devices using the DM mechanism described earlier. Each logical volume is created by allocating some amount of free space from a volume group. Logical volumes can make use of various features present at the DM layer, such as striping, mirroring, and snapshots.

Persistence is provided by dedicating a configurable portion of each physical volume to volume group metadata. This metadata provides all the information necessary to reconstruct the logical volumes within a volume group when the system restart. By default, the metadata is replicated on all physical volumes in the VG for fault tolerance. Each change to the VG metadata also invokes a backup of the previous version to enable recovery from mistakes.

Interaction with LVM is done via a large set of user-level commands, with some examples being `lvcreate`, `lvremove`, and `vgcreate`. These commands offer a convenient interface and also implement the metadata backup feature. Metadata corruption is further avoided in these tools via a locking mechanism to ensure updates are serialized. Based on these factors, we chose to implement our system as a set of extensions to the LVM tool suite. The details of these extensions are discussed in the following section.

5 Implementation

As a means for evaluating the idea of overlay-based virtual disk migration, we've implemented the ability to migrate logical volumes representing both pristine images and snapshot-based overlays. This section describes our mechanisms for both.

While migrating pristine images is a situation that should be feverishly avoided, it becomes a necessary evil when the appropriate image is not present on a migration target. Migrating an image based on a logical volume is straightforward. Exporting an image to a file involves copying directly from the device's file handle, e.g. `/dev/vg/lv`, to a destination file. On some Linux systems, it may be necessary to break the image into multiple files, as files over 2GB are not supported. Our implementation does not handle this case, and the largest images we've dealt with so far are 1.5GB in size. Importing an image is also a simple copy, this time from an image file to a logical volume device file. An extra step of creating a LV for the image is also needed for import.

Migrating a snapshot-based overlay involves a few more steps than simple image migration. The process is detailed in Figure 3. The `lvunbind` and `lvexport` operation combine to checkpoint an overlay on the migration source. At this point, any file transfer mechanism may be used to move the checkpoint to the migration destination. On the destination, `lvimport` and `lvbind` are performed to complete the disk migration. These overlay migration components form the bulk of our implementation and are detailed in the following subsections.

5.1 `lvunbind`

In migrating a snapshot-based overlay we only wish to transfer the contents of the exception store. Performing a copy directly from the snapshot logical device, in a way similar to what was done for image migration, would not produce the desired effect. Although the snapshot device occupies a small amount of disk space relative to its origin, the snapshot device behaves like a block device of the same size as its origin. Thus, a direct copy would produce a file the size of the entire system image.

In order to access the exception store directly, we first must tear down part of the snapshot DM state. `lvunbind`, run on an active snapshot, breaks the association between an exception store and the origin it refers to. The result is that the exception store now appears as normal logical volume. This situation is shown as state 2 in Figure 3.

5.2 `lvexport`

With the exception store now directly accessible as a logical volume, we can simply copy the volume's contents to a file as was done for full system images. Such a copy would result in a file as big as the full capacity of the exception store. This capacity is specified by an administrator at the time when the snapshot is created, and is a conservative estimate of the size that the snapshot may grow to. We may, however, further reduce the size of our checkpoint file by

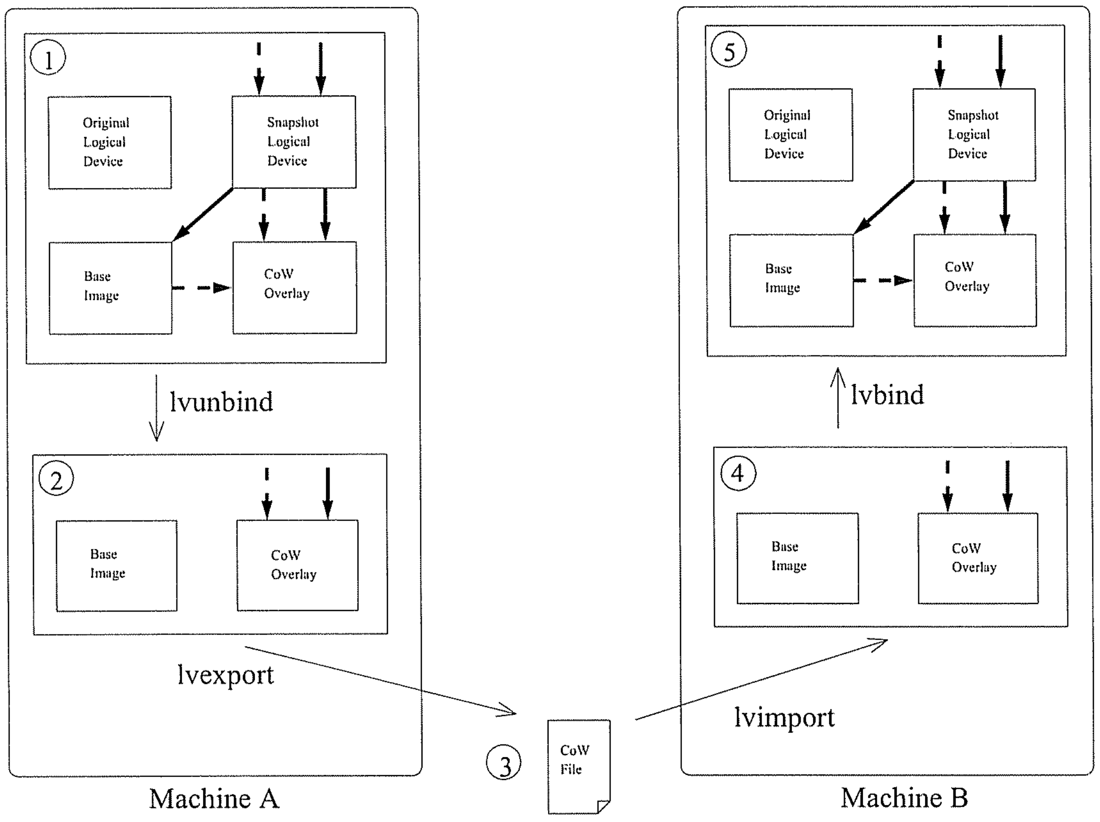
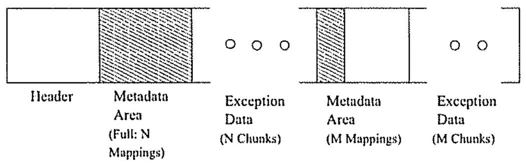


Figure 3: *Overlay migration flow chart*

only exporting the blocks that are actually in use by the exception store.

In order to throw away unneeded data, we use the internal metadata of the exception store to determine which blocks are *live* data. The exception store is treated as an array of chunks, the chunk size being a configurable parameter at snapshot creation time. The first chunk on the device contains some basic header information. The next chunk is a *metadata area*, which stores an array of mappings from blocks in the original device to blocks in the exception store. The next set of blocks contain the data that corresponds to these mappings. This pattern of metadata areas followed by the corresponding data chunks continues until a special end marker appears in a metadata area. This exception store format is illustrated in Figure 4.

In order to export only the needed blocks from the exception store, our `lvexport` tool scans the metadata areas for the end marker. The result is that our exported overlay consists only of the chunks that have been changed from the original device, plus a small amount of additional metadata. The exported overlay can then be transferred and used on any migration target that contains a matching pristine image.



Header
Metadata Area (Full: N Mappings)
Exception Data (N Chunks)
Metadata Area (M Mappings)
Exception Data (M Chunks)

Figure 4: *Example exception store layout*

5.3 `lvimport`

Once the overlay file is transferred to the destination target, we use the `lvimport` utility to create and populate a logical volume to serve as the exception store. The user may specify the maximum capacity of this new exception store, but the chunk size must match that of the source's, since the overlay file format depends on this parameter. The situation after `lvimport` is depicted as state 4 in Figure 3.

5.4 `lvbind`

`lvbind` takes as input two logical volumes: an origin and an exception store, and creates a working snapshot. This process sets up the DM tables and makes the appropriate changes to the volume group metadata. After `lvbind` is run, the disk is fully migrated and the overlying file system may be used. `lvbind` is the complement of the `lvunbind` command introduced earlier.

In addition to implementing `lvbind` as a standalone operation, we also added an option to the stock `lvcreate` command to create a snapshot using a disk file as the initial contents of the exception store. This essentially combines the `lvimport` and `lvbind` operations, and we used this version of `lvcreate` in evaluating our LIMBO prototype.

We found that the `lvbind` process takes significantly longer to run than `lvunbind`. This is because binding an exception store to an origin causes the DM to read in all metadata from the exception store and use it to populate a hash table. This table is then used to perform lookups to determine whether a given block is in the exception store or only exists in the origin.

6 Evaluation

In evaluating our prototype, we set out to answer the following questions:

1. How large can we expect overlays to be?
2. How long does it take to checkpoint and restart overlays as compared to entire disk images?
3. What is the performance cost of running a file system on top of an LVM snapshot versus running directly over a physical partition?

The experiments presented here were all run on Pentium III machines with 256MB of RAM. Since our goal is to use LIMBO for migration of virtual machines, the machines ran the Xen 2.0.5 VMM, with Linux 2.6.10 as the domain-0 operating system kernel. The domain-0 OS was allocated 128MB of RAM, and no guest OSs were running while the experiments were conducted, unless noted otherwise.

6.1 Overlay Size

Before comparing overlay migration times times to image migration times, it is useful to know roughly how large an overlay will get after it's in use for some time. In this experiment, we try to represent the typical amount of change that a user may make to their system in a few weeks time. After this time, we expect most users to update their pristine state to include these changes, allowing them to restart with an empty overlay. Supporting this operation efficiently is a necessary complement to LIMBO which we were unable to explore due to time constraints.

	Chunk Size			
	4KB	8KB	16KB	Ideal
coreutils-5.2.1	17.98	18.08	18.39	16.74
emacs-21.3	86.66	88.49	89.20	79.60
openssh-3.9	88.09	90.02	90.83	80.93
jre-1.4.2	149.35	151.18	152.20	141.06

Table 3: *Cumulative effective of software installations on overlay size (in MB). Ideal is the cumulative disk space the installations would require on a raw ext2 partition.*

Starting with a Gentoo Linux installation with about 1GB of file system data installed, we consecutively installed the packages shown in Table 3. We see that the net effect was an overlay size of about 150MB. Varying the chunk size parameter of the snapshot did not significantly effect the size of the overlay - larger chunk sizes resulted in only slightly larger files. From this experiment, we reason that it is possible to take advantage of the space savings of overlay-based migration for a reasonable amount of time before overlays must be merged back into a base system image.

We expect the bulk of changes to a system to be results of software installations, like those tested here, and configuration changes. Since configuration changes are small compared to software installations, we've ignored their effect on overlay size for the purposes of this experiment.

6.2 Checkpoint/Restart Performance

We locally checkpointed and restarted several images and overlays of varying sizes to illustrate the benefits of migrating an overlay instead of an entire image. Figure 5, illustrates that the time to both checkpoint and restart an image scales linearly with the size of the image. Further, restarting is only marginally slower than checkpointing. Figure 6, displays the times for checkpointing and restarting overlays. This also scales linearly with size (note the log-scale in the x-axis), but the disparity between the checkpoint and restart times is larger. Restarting an overlay is more expensive because the DM must reconstruct the

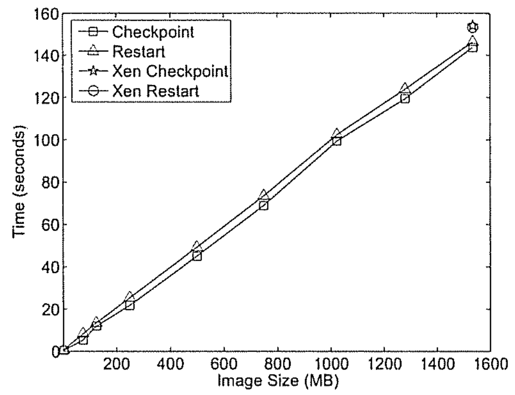


Figure 5: *Checkpoint and restart times for full system images.*

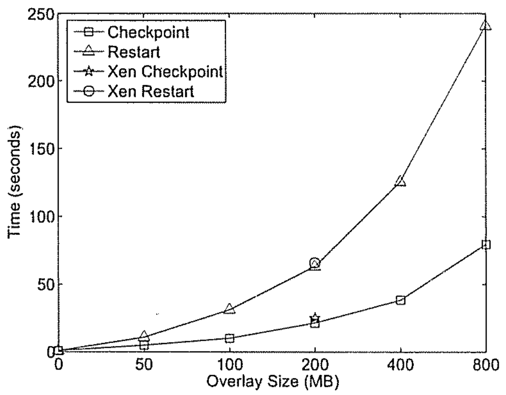


Figure 6: *Checkpoint and restart times for system overlays.*

in-memory metadata mapping from the checkpointed CoW file. For example, an 800MB overlay with 8KB chunks has 102,400 map entries or about 1600KB of map data. From our previous experiment (Section 6.1) we expect a reasonable amount of modifications to a 1.5GB image to total less than 200MB in overlay size. This experiment shows that checkpointing a 200MB overlay is nearly seven times faster than checkpointing the entire 1.5GB image, even disregarding network transmission. Despite the expense of reconstructing the chunk mappings, restarting a 200MB overlay is more than twice as fast as restarting an entire 1.5GB image. We checkpointed and restarted a Xen guest OS running on a 1.5GB pristine image with a 200MB overlay to further illustrate the performance increase when migrating an entire virtual machine using overlays. As a proof of concept, in this experiment we migrated the Xen guest OS to another machine with a copy of the same pristine image. Checkpointing the Xen guest OS with overlays was over six times faster than checkpointing Xen with the entire image. Restarting Xen with overlays was twice as fast as restarting Xen with the entire image, even excluding the overhead of network transmission.

6.3 Snapshot Overhead

We also performed a series of benchmarks comparing Linux file systems running directly on the device to file systems running over a writable-persistent snapshot to ensure the overhead of running a file system over a snapshot is not overly restrictive. Additionally, we varied the snapshot chunk size in these experiments to determine whether the 8KB default is an appropriate granularity. A large chunk size can be more efficient than a smaller chunk size because performing a single 16KB copy between the origin device and snapshot device is likely more efficient than performing four 4KB copies. However, this depends largely on the locality of writes. If a series of writes only modifies a single 4KB range of data, it is far more efficient to perform a 4KB copy than a 16KB copy.

	ext2	ext3
Phase I	0.023179	0.024229
Phase II	0.240448	0.245333
Phase III	0.624104	0.630803
Phase IV	0.879889	0.892818
Phase V	75.720034	76.034853
Total	77.493653	77.828036

Table 4: *Modified Andrew benchmark times (in seconds) for ext2 and ext3 running on raw disk partitions.*

In our first experiment we used a modified Andrew Benchmark [5]. This benchmark consists of five phases, each stressing a different component of the file system. Phase 1 recursively creates directories, phase 2 performs a series of file copies, phase 3 examines the metadata of several files, phase 4 performs reads, and phase 5 compiles the OpenSSH source tree. Table 4 shows the baseline benchmark results for ext2 [3] and ext3 [4] running directly on the device. Figure 7 and Figure 8 illustrate the overheads incurred when using snapshot volumes with a varying chunk size. The largest overhead using ext2 is incurred when performing copies, slightly over 25%. This is expected because writes to the snapshot device cause two writes to occur, one from the origin to

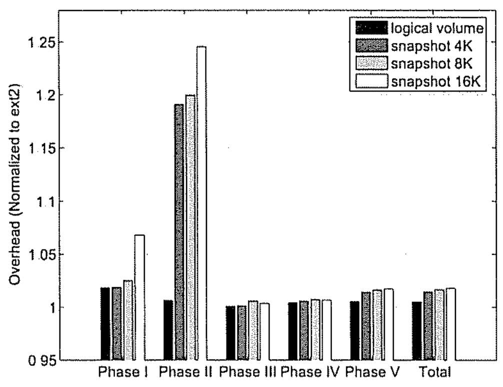


Figure 7: *Overhead of running ext2 over a snapshot. Normalized to ext2 running on a raw disk partition.*

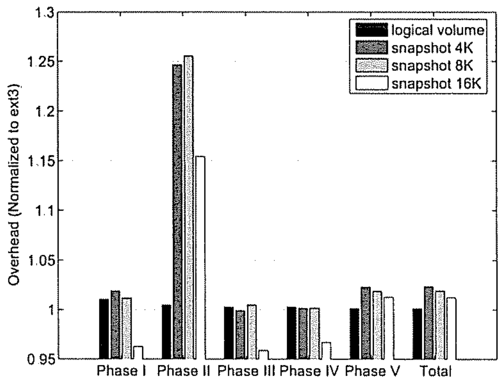


Figure 8: *Overhead of running ext3 over a snapshot. Normalized to ext3 running on a raw disk partition.*

the snapshot and one from the user to the snapshot. However, the total overhead for this benchmark using ext2 over a snapshot is still less than 2%. Also note that in this experiment a 4KB chunk size performed better than either the 8KB or 16KB sizes. This would imply that this benchmark does not stress non-locality.

Like ext2, in ext3 experiment the largest overhead is incurred when performing copies. An interesting phenomenon we discovered from this benchmark, is that using a snapshot with 16KB chunks provides better performance for creating directories, reads, and compiling than raw ext3. This may be caused by ext3's journaling facility, where metadata changes are written to a journal before being written to the inode to ease crash recovery. The first update to the journal causes the entire journal chunk to be copied to the CoW device. The associated disk update also causes a chunk to be written to the CoW device. It is likely that the data chunk and the journal chunk are now much nearer on the CoW device then they were on the origin device. This should significantly reduce seek times for further updates to both the journal and the data chunk. We are uncertain what causes an increase in read performance. Snapshots perform close to 2% worse for the entire benchmark even with these performance improvements. This is due, primarily, to the poor write performance when compared to a raw ext3 partition. Also note that in this experiment a 16KB chunk size performed better than either the 8KB or 4KB sizes and in some cases even the raw partition. We believe this may be caused by the larger preallocation of log space on the CoW device provided by a 16KB chunk.

The results of the previous section made us doubt the quality of the modified Andrew Benchmark for evaluating snapshot performance using the ext3 file system. We ran the PostMark [8] benchmark with an ext3 file system using both a raw disk partition and snapshots to provide a counterpoint to the above test. In our experiment the PostMark benchmark created 761 files with sizes ranging from 500 bytes to 500KB and performed roughly 250 reads and as many writes. We disabled Unix standard I/O buffering for these tests to remove any possible advantages this buffering may provide to snapshots. Table 5 shows the results of this benchmark for both cold and warm snapshot performance. The cold experiments were run with a fresh snapshot containing no CoW exception chunks. The warm experiments were performed after a single run of the benchmark on the snapshot. The performance penalty for using a cold snapshot is large, read and write throughput is less than a quarter of a raw disk throughput. This is likely caused by the additional bandwidth consumed copying chunks from the origin to the snapshot. However, after this initial penalty the reduction in throughput caused by using a snapshot is less than 12% for writes and 6% for reads.

7 Related Work

Other work has been motivated by migrating virtual machines to facilitate mobile and grid computing. Internet Suspend/Resume (ISR) [14] is an abstraction comparing migrating virtual machines for user mobility to the suspend and resume functionality of laptops. The implementation of this abstraction uses VMWare [9] and NFS [6] to provide

		ext3	lv	snapshot(4KB)	snapshot(8KB)	snapshot(16KB)
Cold	Transactions (#/s)	35	35	12	10	10
	Read (MB/s)	3.73	3.73	1.02	0.87	0.94
	Write (MB/s)	11.69	11.69	3.19	2.66	2.88
Warm	Transactions (#/s)	35	35	33	35	33
	Read (MB/s)	3.73	3.73	3.36	3.36	3.53
	Write (MB/s)	11.69	11.69	10.52	10.52	11.07

Table 5: *PostMark benchmark results for an ext3 file system running on a raw partition, a logical volume, and snapshots using varying chunk sizes*

checkpoint functionality and distributed storage respectively. This approach stores and migrates an entire disk image for each user. The Grid Virtual File System (GVFS) [24] is user-level modification to NFS to provide on-demanding paging to grid systems. GVFS also provides some optimizations for migrating virtual machines, including using metadata to store which memory pages are empty and do not require migrating. GVFS advocates using the pristine image and change log technique to migrate VM disk images.

There has been related work focusing on synchronizing a pair of file systems. `diff` [2] is a GNU program that can generates a list of differences between two files or file hierarchies. However, `diff` is too computationally expensive even for a scan implementation of customization encapsulation. Each pair-wise byte of the two file systems is examined and compared. This would mean that the entire image would have to be sent and compared if the pristine image were on a remote image server. `rsync` [23] is another tool for synchronizing a pair of file systems. It can also be used as a scan technique, but more efficiently than `diff`. `rsync` breaks a file into fixed-size blocks and computes a hash for each block using a rolling checksum. This checksum is then sent to the site of the other file system and compared to every block-sized segment of data in the other file. When this completes for every block in the file, only data that is not on the other file system is sent. This algorithm prevents sending the complete file when non-appending writes shift the block alignment by comparing the hash against all *possible* blocks.

A follow up paper [16] to ISR replaces NFS with a modified Coda [21]. Coda already performs prefetching and long-term caching because it is designed to work in a disconnected state or on a low-bandwidth network. In the updated implementation of ISR, disk images are prefetched and cached using Coda and only changes to the image are written back to Coda server. This implementation does not use an abstraction of a pristine images so each user must store a large disk image on the Coda server. Additionally, there is a large restart penalty for migrating to an unlikely host as the entire disk image must be fetched over a possibly high-latency, low-bandwidth link. The modified ISR uses demand paging to offset the penalty for a cold cache, but this introduces a performance penalty while the VM waits for requested pages to be transmitted across the network. A further follow up [15] to ISR allows the user to carry a memory device to be used as a look-aside cache when performing restarts with a cold cache. However, because the contents of this device are not pristine the migration infrastructure must still query the server to ensure cache consistency.

LBFS [18] is a distributed file system designed to work in a low-bandwidth WAN environment. Like `rsync`, LBFS sends hashes prior to sending data blocks to prevent transmitting blocks that are already at the remote side. Also like `rsync` LBFS uses a mechanism to prevent insertion at the front or in the middle of a file from requiring the entire file to be retransmitted. Additionally, LBFS stores its data blocks in a hash-indexed database to reduce the impact of redundant data on storage space. LBFS could be used effectively as distributed file system implementation of customization encapsulation because, although each user would have a separate non-pristine system image, redundant blocks among the images would only be stored once. Additionally, storing a pristine image on LBFS would require little overhead and would allow efficient prefetching. A pristine image would not be linked in a logical way to a given user's system image, but the images would share a multitude of blocks. A host can prefetch the pristine image, and when a user migrates to that host LBFS will only transmit the blocks that differ from the pristine image. One drawback to using LBFS for VM disk migration is that block hashing may decrease file system performance.

VMWare Workstation 5 [10] can capture a snapshot of a running guest operating system, including its virtual disk. Further, this version of Workstation has a management tool that allows snapshots to be versioned either linearly or in a tree structure, much like code versioning systems. However, we were unable to find any technical documentation detailing whether commonality between snapshot versions is exploited or whether there is a merging feature for branches of the same version tree.

Other related work has explored providing a layer of indirection between block devices and the file system. The Logical Disk [13] is a logical layer that manages allocation of space on the device. This abstraction removes the responsibility of disk layout and allocation from the file system. The focus of this work is to provide an interface to the logical disk that is simple, but still allows the file system to optimize disk layout for performance. LIMBO uses a device driver abstraction that implements the same interface as the actual device driver. This interface does not allow the file system to give hints about block layout so the layout of blocks on the CoW device may not be optimal. Petal [17] provides a layer of indirection between file systems and block storage devices to implement network distribution at the block level. Again this abstraction allows the file system to focus on managing files, while the underlying abstraction handles distribution, replication, and fault tolerance.

8 Conclusion

We have demonstrated the utility of enabling local file system migration for virtual machines by representing disk contents as a set of changes to a pristine image. We have also shown that using a copy-on-write block device is an effective means of capturing these changes with an acceptable amount of overhead during normal operation. The overlay checkpoint and restart times for LIMBO are significantly faster than those for full system images.

This work lays a foundation for overlay-based migration. Making this system fully usable would require adding the ability to merge overlays into a user's pristine state (either a user-custom image or a pristine overlay) in order to bound overlay growth. Additionally, using pristine overlays to create a multi-tiered customization encapsulation framework is also an area of future work.

References

[1] *Amd multi-core white paper*, Tech. report, AMD.

[2] *diff*, http://www.gnu.org/software/diffutils/diffutils.html.

[3] *ext2*, http://e2fsprogs.sourceforge.net/ext2intro.html.

[4] *ext3*, http://www.redhat.com/support/wpapers/redhat/ext3/.

[5] *Modified andrew benchmark*, http://www.citi.umich.edu/projects/nfs-perf/results/cmarion/.

[6] *Nfs*, http://sourceforge.net/projects/nfs.

[7] *Open afs*, www.openafs.org.

[8] *Postmark*, http://www.netapp.com/tech_library/3022.html.

[9] *Vmware*, http://www.vmware.com.

[10] *Vmware snapshot version manager*, http://www.vmware.com/support/ws5/doc/preserve_snapshot_ws.html.

[11] *Enhanced virtualization on Intel architecture-based servers*, Tech. report, Intel, 2005.

[12] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield, *Xen and the art of virtualization*, SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, 2003.

[13] Wiebren de Jonge, M. Frans Kaashoek, and Wilson C. Hsieh, *The logical disk: A new approach to improving file systems*, Proceedings of the 14th ACM Symposium on Operating Systems Principles, 1993.

[14] Michael Kozuch and M. Satyanarayanan, *Internet suspend/resume*, Fourth IEEE Workshop on Mobile Computing Systems and Applications, April 2002.

[15] Michael Kozuch, M. Satyanarayanan, Thomas Bressoud, Casey Helfrich, and Shafeeq Sinnamohideen, *Seamless mobile computing on fixed infrastructure*, Tech. Report IRP-TR-04-28, Intel Research Pittsburgh, 2004.

[16] Michael Kozuch, M. Satyanarayanan, Thomas Bressoud, and Yan Ke, *Efficient state transfer for internet suspend/resume*, Tech. Report IRP-TR-02-03, Intel Research Pittsburgh, May 2002.

[17] Edward K. Lee and Chandramohan A. Thekkath, *Petal: Distributed virtual disks*, Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (Cambridge, MA), 1996, pp. 84–92.

[18] Athicha Muthitacharoen, Benjie Chen, and David Mazieres, *A low-bandwidth network file system*, Symposium on Operating Systems Principles, 2001, pp. 174–187.

[19] S. Osman, D. Subhraveti, G. Su, and J. Nieh, *The design and implementation of Zap: A system for migrating computing environments*, 5th USENIX Symposium on Operating Systems Design and Implementation, December 2002, pp. 361–376.

[20] Daniel Quinlan, Paul Russell, and Christopher Yeoh, *Filesystem hierarchy standard*, Tech. report, Filesystem Hierarchy Standard, 2004.

[21] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere, *Coda: A highly available file system for a distributed workstation environment*, IEEE Transactions on Computers **39** (1990), no. 4, 447–459.

[22] Muthian Sivathanu, Lakshmi Bairavasundaram, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau, *Life or Death at Block-Level*, Proceedings of the Sixth Symposium on Operating Systems Design and Implementation (OSDI '04) (San Francisco, CA), December 2004.

[23] Andrew Tridgell and Paul Mackerras, *The rsync algorithm*, Tech. report, Australian National University, 1998, http://samba.anu.edu.au/rsync/tech_report.

[24] Ming Zhao, Jian Zhang, and Renato Figueiredo, *Distributed file system support for virtual machines in grid computing*, Proceedings of HPDC-13, 2004.