Optimizing the Migration of Virtual Computers

Constantine P. Sapuntzakis Ramesh Chandra Ben Pfaff Jim Chow
Monica S. Lam Mendel Rosenblum
Computer Science Department
Stanford University

{csapuntz, rameshch, blp, jimchow, lam, mendel}@stanford.edu

"Beam the computer up, Scotty!"

Abstract

This paper shows how to quickly move the state of a running computer across a network, including the state in its disks, memory, CPU registers, and I/O devices. We call this state a *capsule*. Capsule state is hardware state, so it includes the entire operating system as well as applications and running processes.

We have chosen to move *x*86 computer states because *x*86 computers are common, cheap, run the software we use, and have tools for migration. Unfortunately, *x*86 capsules can be large, containing hundreds of megabytes of memory and gigabytes of disk data. We have developed techniques to reduce the amount of data sent over the network: copy-on-write disks track just the updates to capsule disks, "ballooning" zeros unused memory, demand paging fetches only needed blocks, and hashing avoids sending blocks that already exist at the remote end. We demonstrate these optimizations in a prototype system that uses VMware GSX Server virtual machine monitor to create and run *x*86 capsules. The system targets networks as slow as 384 kbps.

Our experimental results suggest that efficient capsule migration can improve user mobility and system management. Software updates or installations on a set of machines can be accomplished simply by distributing a capsule with the new changes. Assuming the presence of a prior capsule, the amount of traffic incurred is commensurate with the size of the update or installation package itself. Capsule migration makes it possible for machines to start running an application within 20 minutes on a 384 kbps link, without having to first install the application or even the underlying operating system. Furthermore, users' capsules can be migrated during a commute between home and work in even less time.

1 Introduction

Today's computing environments are hard to maintain and hard to move between machines. These environments encompass much state, including an operating system, installed software applications, a user's individual data and profile, and, if the user is logged in, a set of processes. Most of this state is deeply coupled to the computer hardware. Though a user's data and profile may be mounted from a network file server, the operating system and applications are often installed on storage local to the computer and therefore tied to that computer. Processes are tied even more tightly to the computer; very few systems support process migration. As a result, users cannot move between computers and resume work uninterrupted. System administration is also more difficult. Operating systems and applications are hard to maintain. Machines whose configurations are meant to be the same drift apart as different sets of patches, updates, and installs are applied in different orders.

We chose to investigate whether issues including user mobility and system administration can be addressed by encapsulating the state of computing environments as first-class objects that can be named, moved, and otherwise manipulated. We define a *capsule for a machine architecture* as the data type encapsulating the complete state of a (running) machine, including its operating system, applications, data, and possibly processes. Capsules can be bound to any instance of the architecture and be allowed to resume; similarly, they can be suspended from execution and serialized.

A computer architecture need not be implemented in hardware directly; it can be implemented in software using virtual machine technology[12]. The latter option is particularly attractive because it is easier to extract the state of a virtual computer. Virtual computer states are themselves sometimes referred to as "virtual machines." We introduce the term "capsule" to distinguish the contents of a machine state as a data type from the machinery that can execute machine code. After all, we could bind these machine states to real hardware and not use virtual machines at all.

To run existing software, we chose the standard *x*86 architecture[11, 32] as the platform for our investigation. This architecture runs the majority of operating systems and software programs in use today. In addition, commercial *x*86 virtual machine monitors are available, such as VMware GSX Server (VMM)[28] and Connectix Virtual PC[7], that can run multiple virtual *x*86 machines on the same hardware. They already provide the basic functions of writing out the state of a virtual *x*86 machine, binding the serialized state onto a virtual machine, and resuming execution.

The overall goal of our research project is to explore the design of a capsule-based system architecture, named *the Collective*, and examine its potential to provide user mobility, recovery, and simpler system management. Computers and storage in the Collective system act as caches of capsules. As users travel, the Collective can move their capsules to computers close to them, giving users a consistent environment. Capsules could be moved with users as they commute between home and work. Capsules can be duplicated, distributed to many different machines, and updated like any other data; this can form the basis for administering a group of computers. Finally, capsules can be moved among machines to balance loads or for fail-over.

1.1 Storing and Migrating Capsules

Many challenges must be addressed to realize our goals of the Collective project, but this paper focuses on one simple but crucial one: can we afford the time and space to store, manipulate and migrate *x*86 capsules? *x*86 capsules can be very large. An inactive capsule can contain gigabytes of disk storage, whereas an active capsule can include hundreds of megabytes of memory data, as well as internal machine registers and I/O device states. Copying a gigabyte capsule over a standard 384 kbps DSL link would take 6 hours! Clearly, a straightforward implementation that copies the entire capsule before starting its computation would take too long.

We have developed a number of optimizations that reduce capsules' storage requirements, transfer time and start-up time over a network. These techniques are invisible to the users, and do not require any modifications to the operating system or the applications running inside it. Our techniques target DSL speeds to support capsule migration to and from the home, taking advantage of the availability of similar capsules on local machines.

To speed up the transfer of capsules and reduce the start-up times on slow networks, our system works as follows:

1. Every time we start a capsule, we save all the updates made to disk on a separate disk, using copy-on-write. Thus, a snapshot of an execution can be represented with an incremental cost commensurate with the magnitude of the updates performed.
2. Before a capsule is serialized, we reduce the memory state of the machine by flushing non-essential data to disk. This is done by running a user "balloon" process that acquires memory from the operating system and zeros the data. The remaining subset of memory is transferred to the destination machine and the capsule is started.
3. Instead of sending the entire disk, disk pages are fetched on demand as the capsule runs, taking full advantage of the operating system's ability to tolerate disk fetch latencies.
4. Collision-resistant hashes are used to avoid sending pages of memory or disk data that already exist at the destination. All network traffic is compressed with gzip[8].

We have implemented all the optimizations described in this paper in a basic prototype of our Collective system. Our prototype's platform uses VMware GSX Server 2.0.1 running on Red Hat Linux 7.3 (kernel 2.4.18-10) to execute *x*86 capsules. Users can retrieve their capsules by name, move capsules onto a file system, start capsules on a computer, and save capsules to a file system. We have run both Linux and Windows in our capsules.

Our results show that we can move capsules in 20 minutes or less across 384 kbps DSL, fast enough to move users' capsules between home and work as they commute. Speed improves when an older version of the capsule is available at the destination. For software distribution, we show that our system sends roughly the same amount of data as the software installer package for newly installed software, and often less for upgrades to already installed software. The results suggest that capsule migration offers a new way to use software where machines can start running a new application within a few minutes, with no need to first install the application or even its underlying operating system.

1.2 Paper Organization

Section 2 describes how we use a virtual machine monitor to create and resume capsules. Section 3 motivates the need for optimizations by discussing the intended uses of capsules. Section 4 discusses the optimizations we use to reduce the cost of capsules. In Section 5 we describe some experiments we performed on a prototype of our system. The paper discusses related work in Section 6 and concludes in Section 7.

2 Virtual Machine Monitors

A virtual machine monitor is a layer of software that sits directly on the raw hardware and exports a virtual machine abstraction that imitates the real machine well enough that software developed for the real machine also runs in the virtual machine. We use an *x*86 virtual machine monitor, VMware GSX Server, to generate, serialize, and execute our *x*86 capsules.

Virtual machine monitors have several properties that make them ideal platforms for supporting capsules. The monitor layer encapsulates all of the machine state necessary to run software and mediates all interactions between software and the real hardware. This encapsulation allows the monitor to suspend and disconnect the software and virtual device state from the real hardware and write that machine state to a stream. Similarly, the monitor can also bind a machine state to the real hardware and resume its execution. The monitor requires no cooperation from the software running on the monitor.

Migration is made more difficult by the myriad of hardware device interfaces out there. GSX Server simplifies migration by providing the same device interfaces to the virtual machine regardless of the underlying hardware; virtualization again makes this possible. For example, GSX Server exports a Bus Logic SCSI adapter and AMD Lance Ethernet controller to the virtual machine, independent of the actual interface of disk controller or network adapter. GSX in turn runs on a *host operating system*, currently Linux or Windows, and implements the virtual devices using the host OS's devices and files.

Virtual hard disks are especially powerful. The disks can be backed not just by raw disk devices but by files in the host OS's file system. The file system's abilities to easily name, create, grow, and shrink storage greatly simplify the management of virtual hard disks.

Still, some I/O devices need more than simple conversion routines to work. For example, moving a capsule that is using a virtual network card to communicate over the Internet is not handled by simply remapping the device to use the new computer's network card. The new network card may be on a network that is not able to receive packets for the capsule's IP address. However, since the virtualization layer can interpose on all I/O, it can, transparent to the capsule, tunnel network packets to and from the capsule's old network over a virtual private network (VPN).

3 Usages and Requirements

The Collective system uses serialization and mobility of capsules to provide user mobility, backup, software management and hardware management. We describe each of these applications of capsules and explain their requirements on capsule storage and migration.

3.1 User Mobility

Since capsules are not tied to a particular machine, they can follow users wherever they go. Suppose a user wants to work from home on evenings and weekends. The user has a single active work capsule that migrates between a computer at home and one at work. In this way, the user can resume work exactly where he or she left off, similar to the convenience provided by carrying a laptop. Here, we assume standard home and office workloads, like software engineering, document creation, web browsing, e-mail, and calendar access. The system may not work well with data-intensive applications, such as video editing or database accesses.

To support working from home, our system must work well at DSL or cable speeds. We would like our users to feel that they have instantaneous access to their active environments everywhere. It is possible to start up a capsule without having to entirely transfer it; after all, a user does not need all the data in the capsule immediately. However, we also need to ensure that the capsule is responsive when it comes up. It would frustrate a user to get a screen quickly but to find each keystroke and mouse click processed at glacial speed.

Fortunately, in this scenario, most of the state of a user's active capsule is already present at both home and work, so only the differences in state need to be transferred during migration. Furthermore, since a user can easily initiate the capsule migration before the commute, the user will not notice the migration delay as long as the capsule is immediately available after the commute.

3.2 Backups

Because capsules can be serialized, users and system administrators can save snapshots of their capsules as backups. A user may choose to checkpoint at regular intervals or just before performing dangerous operations. It is prohibitively expensive to write out gigabytes to disk each time a version is saved. Again, we can optimize the storage by only recording the differences between successive versions of a capsule.

3.3 System Management

Capsules can ease the burden of managing software and hardware. System administrators can install and maintain the same set of software on multiple machines by simply creating one (inactive) capsule and distributing it to all the machines. This approach allows the cost of system administration to be amortized over machines running the same configuration.

This approach shares some similarities with the concept of disk imaging, where local disks of new machines are given some standard pre-installed configuration. Disk imaging allows each machine to have only one configuration. On the other hand, our system allows multiple capsules to co-exist on the same machine. This has a few advantages: It allows multiple users with different requirements to use the same machine, e.g. machines in a classroom may contain different capsules for different classes. Also, users can use the same machine to run different capsules for different tasks. They can have a single customized capsule each for personal use, and multiple work capsules which are centrally updated by system administrators. The capsule technique also causes less disruption since old capsules need not be shut down as new capsules get deployed.

Moving the first capsule to a machine over the network can be costly, but may still be faster and less laborious than downloading and installing software from scratch. Moving subsequent capsules to machines that hold other capsules would be faster, if there happen to be similarities between capsules. In particular, updates of capsules naturally share much in common with the original version.

We can also take advantage of the mobility of capsules to simplify hardware resource management. Rather than having the software tied to the hardware, we can select computing hardware based on availability, load, location, and other factors. In tightly connected clusters, this mobility allows for load balancing. Also, migration allows a machine to be taken down without stopping services. On an Internet scale, migration can be used to move applications to servers that are closer to the clients[3].

3.4 Summary

The use of capsules to support user mobility, backup, and system management depends on our ability to both migrate capsules between machines and store them efficiently. It is desirable that our system works well at DSL speed to allow capsules be migrated to and from homes. Furthermore, start-up delays after migration should be minimized while ensuring that the migrated capsules remain responsive.

4 Optimizations

Our optimizations are designed to exploit the property that similar capsules, such as those representing snapshots from the same execution or a series of software upgrades, are expected to be found on machines in a Collective system. Ideally, the cost of storing or transferring a capsule, given a similar version of the capsule, should be proportional to the size of the difference between the two. Also, we observe that the two largest components in a capsule, the memory and the disk, are members of the memory hierarchy in a computer, and as such, many pre-existing management techniques can be leveraged.

Specifically, we have developed the following four optimizations:

1. Reduce the memory state before serialization.
2. Reduce the incremental cost of saving a capsule disk by capturing only the differences.
3. Reduce the start-up time by paging disk data on demand.
4. Decrease the transfer time by not sending data blocks that already exist on both sides.

4.1 Ballooning

Today's computers may contain hundreds of megabytes of memory, which can take a while to transfer on a DSL link. One possibility to reduce the start-up time is to fetch the memory pages as they are needed. However, operating systems are not designed for slow memory accesses; such an approach would render the capsule unresponsive at the beginning. The other possibility is to flush non-essential data out of memory, transfer a smaller working set, and page in the rest of the data as needed.

We observe that clever algorithms that eliminate or page out the less useful data in a system have already been implemented in the OS's virtual memory manager. Instead of modifying the OS, which would require an enormous amount of effort per operating system, we have chosen to use a *gray-box* approach[2] on this problem. We trick the OS into reclaiming physical memory from existing processes by running a *balloon* program that asks the OS for a large number of physical pages. The program then zeros the pages, making them easily compressible. We call this process "ballooning," following the term introduced by Waldspurger[29] in his work on VMware ESX server. While the ESX server uses ballooning to return memory to the monitor, our work uses ballooning to zero out memory for compression.

Ballooning reduces the size of the compressed memory state and thus reduces the start-up time of capsules. This technique works especially well if the memory has many freed pages whose contents are not compressible. There is no reason to transfer such data, and these pages are the first to be cleared by the ballooning process. Discarding pages holding cached data, dirty buffers and active data, however, may have a negative effect. If these pages are immediately used, they will need to be fetched on demand over the network. Thus, even though a capsule may start earlier, the system may be sluggish initially.

We have implemented ballooning in both the Linux and Windows 2000 operating systems. The actual implementation of the ballooning process depends on the OS. Windows 2000 uses a local page replacement algorithm, which imposes a minimum and maximum working set size for each process. To be most effective, the Windows 2000 balloon program must ensure its current working set size is set to this maximum.

Since Linux uses a global page replacement algorithm, with no hard limits on the memory usage of processes, a simple program that allocates and zeros pages is sufficient. However, the Linux balloon program must decide when to stop allocating more memory, since Linux does not define memory usage limits as Windows does. For our tests, the Linux balloon program adopts a simple heuristic that stops memory allocation when free swap space decreases by more than 1MB.

Both ballooning programs explicitly write some zeros to each allocated page so as to stop both OSes from mapping the allocate pages to a single zero copy-on-write page. In addition, both programs hook into the OS's power management support, invoking ballooning whenever the OS receives a suspend request from the VMM.

4.2 Capsule Hierarchies

Capsules in the Collective system are seldom created from scratch, but are mostly derived from other capsules as explained in Section 3. The differences between related capsules are small relative to the total size of the capsules. We can store the disks in these capsules efficiently by creating a hierarchy, where each child capsule could be viewed as inheriting from the parent capsule with the differences in disk state between parent and child captured in a separate copy-on-write (COW) virtual disk.

At the root of the hierarchy is a *root disk*, which is a complete capsule disk. All other nodes represent a COW disk. Each path of COW disks originating from the root in the capsule hierarchy represents a capsule disk; the COW disk at the end of the path for a capsule disk is its *latest disk*. We cannot directly run a capsule whose latest disk is not a leaf of the hierarchy. We must first derive a new child capsule by adding a new child disk to the latest disk and all updates are made to the new disk. Thus, once capsules have children, they become immutable; this property simplifies the caching of capsules.

Figure 1 shows an example of a capsule hierarchy illustrating how it may be used in a university. The root capsule contains all the software available to all students. The various departments in the university may choose to extend the basic capsules with department-specific software. The department administrator can update the department capsule by deriving new child capsules. Students' files are assumed to be stored on networked storage servers; students may use different capsules for different courses; power users are likely to maintain their own private capsule for personal use. Each time a user logs in, he looks up the latest department capsule and derives his own individual capsule. The capsule migrates with the student as he commutes, and is destroyed when he logs out. Note that if a capsule disk is updated, all the derived capsules containing custom installed software have to be ported to the updated capsule. For example, if the University capsule disk is updated, then each department needs to re-create its departmental capsule.

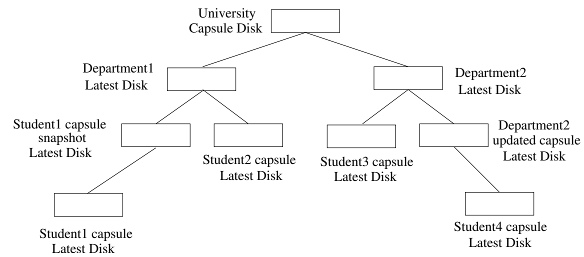

Figure 1: An example capsule hierarchy.

Capsule hierarchies have several advantages: During the migration of a capsule disk, only COW disks that are not already present at the destination need to be transferred. Capsule hierarchies allow efficient usage of disk space by sharing common data among different capsule disks. This also translates to efficient usage of the buffer cache of the host OS, when multiple capsules sharing COW disks simultaneously execute on the same host. And finally, creating a new capsule using COW disks is much faster than copying entire disks.

Each COW disk is implemented as a bitmap file and a sequence of *extent* files. An extent file is a sequence of blocks of the COW disk, and is at most 2 GB in size (since some file systems such as NFS cannot support larger files). The bitmap file contains one bit for each 16 KB block on the disk, indicating whether the block is present in the COW disk. We use sparse file support of Linux file systems to efficiently store large yet only partially filled disks.

Writes to a capsule disk are performed by writing the data to the latest COW disk and updating its bitmap file. Reads involve searching the latest COW disk and its ancestor disks in turn until the required block is found. Since the root COW disk contains a copy of all the blocks, the search is guaranteed to terminate. Figure 2 shows an example capsule disk and the chain of COW disks that comprise it. Note that the COW disk hierarchy is not visible to the VMM, or to the OS and applications inside the capsule; all of them see a normal flat disk as illustrated in the figure.

The COW disk implementation interfaces with GSX Server through a shim library that sits between GSX Server and the C library. The shim library intercepts GSX Server's I/O requests to disk image files in VMware's "plain disk" format, and redirects them to a local disk server. The plain disk format consists of the raw disk data laid out in a sequence of extent files. The local disk server translates these requests to COW disk requests, and executes the I/O operations against the COW disks.

Each suspend and resume of an active capsule creates a new active capsule, and adds another COW layer to its disks. This could create long COW disk chains. To avoid accumulation of costs in storing the intermediate COW disks, and the cost of looking up bitmaps, we have implemented a *promote* primitive for shortening these chains. We promote a COW disk up one level of the hierarchy by adding to the disk all of its parent's blocks not present in its own. We can delete a capsule by first promoting all its children and then removing its latest disk. We can also apply the promotion operations in succession to convert a COW disk at the bottom of the hierarchy into a root disk.

On a final note, VMware GSX Server also implements a copy-on-write format in addition to its plain disk format. However, we found it necessary to implement our own COW format since VMware's COW format was complex and not conducive to the implementation of the hashing optimization described later in the paper.

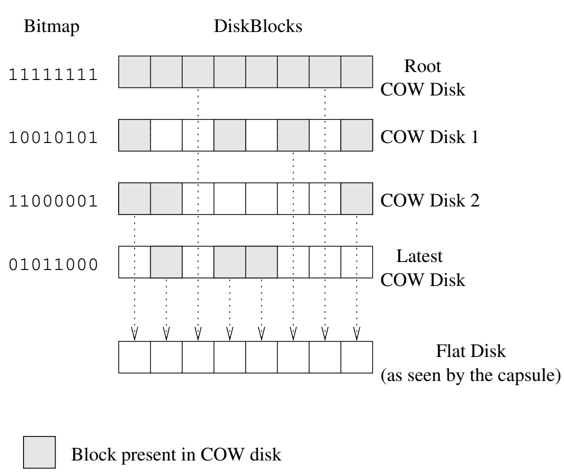

Figure 2: An example capsule disk.

4.3 Demand Paging of Capsule Disks

To reduce the start-up time of a capsule, the COW disks corresponding to a capsule disk are read page-by-page on demand, rather than being pre-fetched. Demand paging is useful because COW disks, especially root disks, could be up to several gigabytes in size and prefetching these large disks could cause an unacceptable start-up delay. Also, during a typical user session, the working set of disk blocks needed is a small fraction of the total blocks on the disk, which makes pre-fetching the whole disk unnecessary. Most OSes have been designed to hide disk latency and hence can tolerate the latency incurred during demand paging the capsule disk.

The implementation of the capsule disk system, including demand paging, is shown in Figure 3. The shim library intercepts all of VMware's accesses to plain disks and forwards them to a disk server on the local machine. The disk server performs a translation from a plain disk access to the corresponding access on the COW disks of the capsule. Each COW disk can either be local or remote. Each remote COW disk has a corresponding local *shadow COW* disk which contains all the locally cached blocks of the remote COW disk.

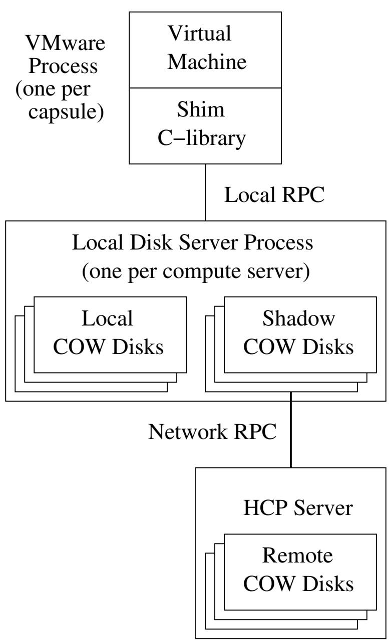

Figure 3: Implementation of capsule disks and demand paging.

Since the latest COW disk is always local, all writes are local. Reads, on the other hand, could either be local or remote. In the case of a remote read, the disk server requests the block from the shadow COW disk. If the block is not cached locally, it is fetched remotely and added to the shadow COW.

Starting a capsule on a machine is done as follows: first, the memory image and all the bitmaps of the COW disks are transferred if they are not available locally. Then the capsule is extended with a new, local latest COW disk. For each remote COW disk, the corresponding shadow COW disk is created if it does not already exist. GSX Server can now be invoked on the capsule. Note that since remote COW disks are immutable, the cached blocks in the shadow COW disks can be re-used for multiple capsules and across suspends and resumes of a single capsule. This is useful since no network traffic is incurred for the cached blocks.

The Collective system uses an LDAP directory to keep track of the hosts on which a COW disk is present. In general, COW disks of a capsule disk could be distributed across many hosts since they were created on different hosts. However, the disks are also uploaded (in the background) to a central storage server for better availability.

4.4 Hash-Based Compression

We use a fourth technique to speed up data transfer over low-bandwidth links. Inspired by the low-bandwidth file system (LBFS[19]) and rsync[27], we decrease transfer time by sending a hash of data blocks instead of the data itself. If the receiver can find data on local storage that hashes to the same value, it copies the data from local storage. Otherwise, the receiver requests the data from the server. We call this technique HCP, for Hashed Copy. The Collective prototype uses HCP for demand paging disks and copying memory and disk images.

We expect to find identical blocks of data between disk images and memories, even across different users' capsules. First, the memory in most systems caches disk blocks. Second, we expect most users in the Collective to migrate between a couple of locations, e.g. home and work. After migrating a couple of times, these locations will contain older memory and disk images, which should contain blocks identical to those in later images, since most users will tend to use the same applications day to day. Finally, most users run code that is distributed in binary form, with most of this binary code copied unmodified into memory when the application runs, and the same binary code (e.g. Microsoft Office or the Netscape web browser) is distributed to millions of people. As a result, we expect to find common blocks even between different users' capsules.

Like LBFS, HCP uses a strong cryptographic hash, SHA-1[1]. The probability that two blocks map to the same 160-bit SHA-1 hash is negligible, less than the error rate of a TCP connection or memory[5]. Also, malicious parties cannot practically come up with data that generates the same hash.

Our HCP algorithm is intended for migrating capsules over low bandwidth links such as DSL. Because HCP involves many disk seeks, its effective throughput is well under 10 Mbps. Hence, it is not intended for high-bandwidth LAN environments where the network is not the bottleneck.

4.4.1 Hash Cache Design

HCP uses a *hash cache* to map hashes to data. Unlike rsync, the cache is persistent; HCP does not need to generate the table by scanning a file or file system on each transfer, saving time.

The cache is implemented using a hash table whose size is fixed at creation. We use the first several bits of the hash key to index into the table. File data is not stored in the table; instead, each entry has a pointer to a file and offset. By not duplicating file data, the cache uses less disk space. Also, the cache can read ahead in the file, priming an in-memory cache with data blocks. Read-ahead improves performance by avoiding additional disk accesses when two files contain runs of similar blocks.

Like LBFS, when the cache reads file data referenced by the table, it always checks that it matches the 20-byte SHA-1 hash provided. This maintains integrity and allows for a couple of performance improvements. First, the cache does not need to be notified of changes to file data; instead, it invalidates table entries when the integrity check fails. Second, it does not need to lock on concurrent cache writes, since corrupted entries do not affect correctness. Finally, the cache stores only the first 8 bytes of the hash in each table entry, allowing us to store more entries.

The hash key indexes into a bucket of entries, currently a memory page in size. On a lookup, the cache does a linear search of the entries in a bucket to check whether one of them matches the hash. On a miss, the cache adds the entry to the bucket, possibly evicting an existing entry. Each entry contains a use count that the cache increments on every hit. When adding an entry to the cache, the hash cache chooses a fraction of the entries at random from the bucket and replaces the entry with the lowest use count; this evicts the least used and hopefully least useful entry of the group. The entries are chosen at random to decrease the chance that the same entry will be overwritten by two parallel threads.

4.4.2 Finding Similar Blocks

For HCP to compress transfers, the sender and receiver must divide both memory and disk images into blocks that are likely to recur. In addition, when demand paging, the operating system running inside the capsule essentially divides the disk image by issuing requests for blocks on the disk. In many systems, the memory page is the unit of disk I/O and memory management, so we chose memory pages as our blocks.

The memory page will often be the largest common unit between different memory images or between memory and disk. Blocks larger than a page would contain two adjacent pages in physical memory; since virtual memory can and does use adjacent physical pages for completely different objects, there is little reason to believe that two adjacent pages in one memory image will be adjacent in another memory image or even on disk.

When copying a memory image, we divide the file into page-sized blocks from the beginning of the image file. For disk images, it is not effective to naively chop up the disk into page-size chunks from the start of the disk; file data on disk is not consistently page aligned. Partitions on *x*86 architecture disks rarely start on a page boundary. Second, at least one common file system, FAT, does not start its file pages at a page offset from the start of the partition. To solve this problem, we parse the partition tables and file system superblocks to discover the alignment of file pages. This information is kept with the disk to ensure we request properly aligned file data pages when copying a disk image.

On a related note, the ext2, FAT, and NT file systems all default to block sizes less than 4 KB when creating smaller partitions; as a result, files may not start on page boundaries. Luckily, the operator can specify a 4 KB or larger block size when creating the file system.

Since HCP hashes at page granularities, it does not deal with insertions and deletions well as they may change every page of a file on disk or memory; despite this, HCP still finds many similar pages.

4.4.3 HCP Protocol

The HCP protocol is very similar to NFS and LBFS. Requests to remote storage are done via remote procedure call (RPC). The server maintains no per-client state at the HCP layer, simplifying error recovery.

Figure 4 illustrates the protocol structure. Time increases down the vertical axis. To begin retrieving a file, an HCP client connects to the appropriate HCP server and retrieves a file handle using the LOOKUP command, as shown in part (a). The client uses READ-HASH to obtain hashes for each block of the file in sequence and looks up all of these hashes in the hash cache. Blocks found via the hash cache are copied into the output file, and no additional request is needed, as shown in part (b). Blocks not cached are read from the server using READ, as in part (c). The client keeps a large number of READ-HASH and READ requests outstanding in an attempt to fill the bandwidth between client and server as effectively as possible.

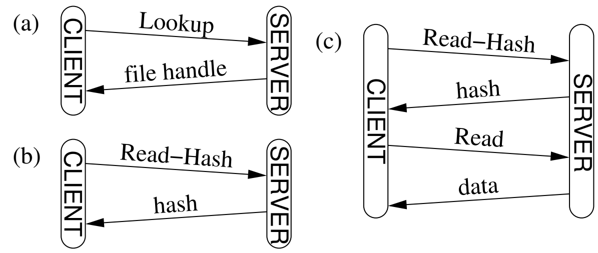

Figure 4: Typical HCP session: (a) session initiation, (b) hash cache hit, (c) hash cache miss.

5 Experimental Results

Our prototype system is based on VMware GSX Server 2.0.1 running on Red Hat Linux 7.3 (kernel 2.4.18-12). Except for the shim library, we wrote the code in Java using Sun's JDK 1.4.0. The experiments ran on 2.4 GHz Pentium 4 machines with 1GB memory.

A separate computer running FreeBSD and the dummynet[23] shaper simulated a 384 kbps symmetric DSL link with 20 ms round-trip delay. We confirmed the setup worked by measuring ping times of 20ms and a TCP data throughput of 360kbps[20]. We checked the correctness of our HCP implementation by ensuring that the hash keys generated are evenly distributed.

We configured the virtual machines to have 256 MB memory and 4 GB local disk. Along with the operating system and applications, the local disk stored user files. In future versions of the system, we expect that user files will reside on a network file system tolerant of low bandwidths.

To evaluate our system, we performed the following four experiments:

1. Evaluated the use of migration to propagate software updates.
2. Evaluated the effectiveness and interaction between ballooning, demand paging, and hash compression.
3. Evaluated the trade-offs between directly using an active capsule versus booting an inactive capsule.
4. Simulated the scenario where users migrate their capsules as they travel between home and work.

5.1 Software Management

Software upgrades are a common system administration task. Consider an environment where a collection of machines maintained to run exactly the same software configuration and users' files are stored on network storage. In a capsule-based system, the administrator can simply distribute an updated capsule to all the machines. In our system, assuming that the machines already have the previous version of the capsule, we only need to send the latest COW disk containing all the changes. Our results show that using HCP to transfer the COW disks reduces the transfer amounts to levels competitive or better than current software install and update techniques. We consider three system administration tasks in the following: upgrading an operating system, installing software packages, and updating software packages.

5.1.1 Operating System Upgrade

Our first experiment is to measure the amount of traffic incurred when updating Red Hat version 7.2 to version 7.3. In this case, the system administrator is likely to start from scratch and create a new root disk, instead of updating version 7.2 and capturing the changes in a COW disk. The installation created a 1.6 GB capsule. Hashing this capsule against a hash cache containing version 7.2 found 30% of the data to be redundant. With gzip, we only need to transfer 25% of the 1.6 GB capsule.

A full operating system upgrade will be a lengthy operation regardless of the method of delivery, due to the large amount of data that must be transferred across the network. Use of capsules may be an advantage for such upgrades because data transfer can take place in the background while the user is using an older version of the capsule being upgraded (or a completely different capsule).

5.1.2 Software Installations and Updates

For this experiment, we installed several packages into a capsule containing Debian GNU/Linux 3.0 and upgraded several packages in a capsule containing Red Hat Linux 7.2. Red Hat was chosen for the latter experiment because out-of-date packages were more readily available.

In each case, we booted up the capsule, logged in as root, ran the Debian apt-get or Red Hat apt-rpm to download and install a new package, configured the software, and saved the capsule as a child of the original one. We migrated the child capsule to another machine that already had the parent cached. To reduce the COW disk size, software packages were downloaded to a temporary disk which we manually removed from the capsule after shutdown.

Figure 5 shows the difference in size between the transfer of the new COW disk using HCP versus the size of the software packages. Figure 5(a) shows installations of some well-known packages; the data point labeled “mega” corresponds to an installation of 492 packages, including the X Window System and TeX. Shown in Figure 5(b) are updates to a number of previously installed applications; the data point labeled “large” corresponds

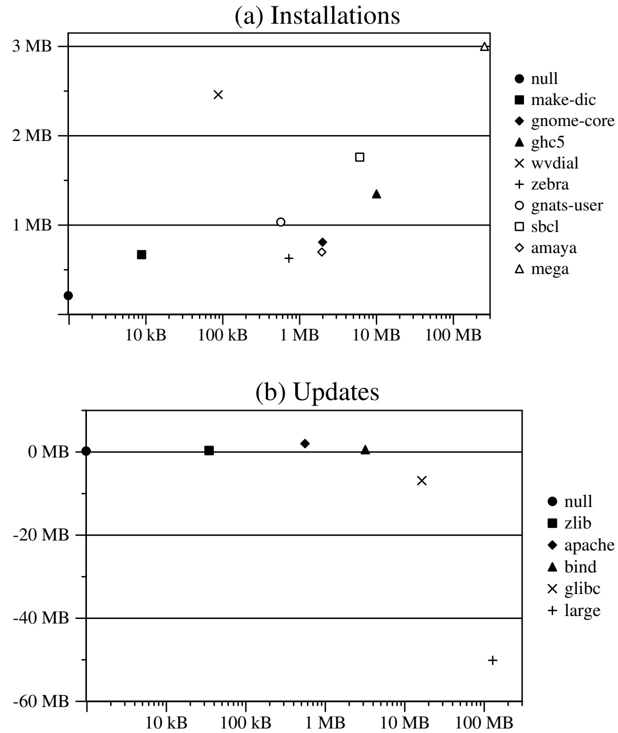

Figure 5: Difference in size between the HCP transfer of the COW disk holding the changes and (a) the installed packages, (b) the update packages.

to an update of 115 packages installed previously and 7 new packages pulled in by the updates. The software updates used were not binary patches; as with an install, they included new versions of all the files in a software package, a customary upgrade method for Debian and Red Hat systems.

For reference, we also include the “null” data point which corresponds to the size of the COW disk created by simply logging in as root and shutting down the capsule without updating any software. This amounts to about 200 KB after HCP and gzip, consisting of i-nodes written due to updated access times, temporary files written at boot, and so on.

As shown in the figure, transfers of small installations and updates are dominated by the installer rewriting two 6 MB text databases of available software. Hashing sometimes saves us from having to send the entire database, but not always, due to insertions that change all the pages. The different results for make-dic and wvdial illustrate this effect. On larger installs, the cost of transferring the disk via HCP is near that of the original package; the overhead of the installer database is bounded by a constant and gzip does a good job of compressing the data. For larger updates, HCP sent less data than the packages because many of these updates contained only minor changes from previous versions (such as security patches and bug fixes), so that hashing found similarities to older, already installed packages. In our experiment, for updates over 10 MB, the savings amount to about 40% in each case.

The results show that distributing COW disks via HCP is a reasonable alternative to current software install and update techniques. Package installations and upgrades incur a relatively low fixed cost, so further benefits can be gained by batching smaller installs. In the case of updates, HCP can exploit similarities between the new and old packages to decrease the amount of data transferred. The convenience of a less tedious and error-prone update method is another advantage.

5.2 Migration of Active Capsules

To show how capsules support user mobility, we performed two sets of experiments, the first on a Windows 2000 capsule, the second on a Linux capsule.

First, we simulated the workload of a knowledge worker with a set of GUI-intensive applications on the Windows 2000 operating system. We used Rational Visual Test software to record user activities and generate scripts that can be played back repeatedly under different test conditions. We started a number of common applications, including Microsoft Word, Excel, and PowerPoint, plus Forte, a Java programming environment, loaded up some large documents and Java sources, saved the active capsule, migrated it to another machine, and proceeded to use each of the four applications.

On Linux, we tested migration with less-interactive and more CPU- and I/O-bound jobs. We chose three applications: processor simulation with `smttls`, Linux kernel compilations with GCC, and web serving with Apache. We imagine that it would be useful to migrate large processor simulations to machines that might have become idle, or migrate fully configured webservers dynamically to adjust to current demand. For each experiment, we performed a task, migrated the capsule, then repeated the same task.

To evaluate the contributions of each of our optimizations, we ran each experiment twelve times. The experimental results are shown in Figure 6. We experimented with two network speeds, 384 kbps DSL and switched 100 Mbps Ethernet. For each speed, we compared the performance obtained with and without the use of ballooning. We also varied the hashing scheme: the experiments were run with no hashing, with hashing starting from an empty hash cache, and with hashing starting from a hash cache primed with the contents of the capsule disk. Each run has two measurements: "migration," the data or time to start the capsule, and "execution," the

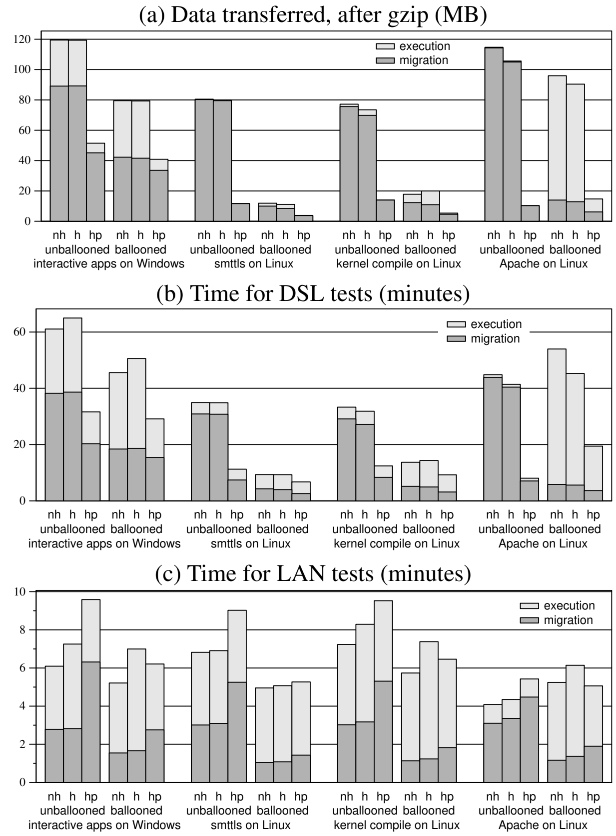

Figure 6: Migration experiments. Data transferred for remote activations and executions are shown after gzip in (a). Time to activate and run the experiments are shown for 384 kbps DSL in (b) and 100 Mbps switched Ethernet in (c). Labels "nh", "h", and "hp" denote no hashing, hashing with an empty hash cache, and hashing with a primed cache, respectively.

data or time it took to execute the task once started.

Figure 6(a) shows the amounts of data transferred over the network during each migration and execution after gzip. These amounts are independent of the network speed assumed in the experiment. The memory image is transferred during the migration step, and disk data are transferred on demand during the execution step. Gzip by itself compresses the 256 MB of memory data to 75–115 MB. Except for the Windows interactive benchmark, none of the applications incurs uncached disk accesses during unballooned execution.

Hashing against an empty cache has little effect because it can only find similarities within the data being transferred. Our results show that either hashing against a primed disk or ballooning alone can greatly reduce the amount of memory data transferred to 10–45 MB. By finding similarities between the old and new capsules, primed hashing reduces the amount of data transferred both during migration and execution. While ballooning reduces the amount of memory data that needs to be transferred, it does so with the possible expense of increasing the data transferred during the execution. Its effectiveness in reducing the total amount of data transferred is application-dependent; all but Apache, which has a large working set, benefit tremendously from ballooning. Combining ballooning with primed hashing generally results in the least amount of data transferred.

The timing results obtained on a 384 kbps DSL link, shown in Figure 6(b), follow the same pattern found in Figure 6(a). The execution takes longer proportionally because it involves computation and not just data transfer. With no optimization, it takes 29–44 minutes just to transfer the memory image over before the capsule can start executing. Hashing with priming reduces the start-up to less than 20 minutes in the Windows interactive experiment, and less than 6 minutes in all the other applications. Ballooning also reduces the start-up time further to 3–16 minutes. Again, combining both ballooning and priming yields the best result in most cases. As the one exception here, Apache demonstrates that ballooning applications with a large working set can slow them down significantly.

Hashing is designed as an optimization for slow network connections; on a fast network, hashing can only slow the transfer as a result of its computational overhead. Figure 6(c) shows this effect. Hashing against a primed cache is even worse because of the additional verification performed to ensure that the blocks on the destination machine match the hash. This figure shows that it takes only about 3 minutes to transfer an unballooned image, and less than 2 minutes ballooned. Again, except for Apache which experiences a slight slowdown, ballooning decreases both the start-up time as well as the overall time.

The Windows experiment has two parts, an interactive part using Word, Excel, and PowerPoint on a number of large documents, followed by compiling a source file and building a Java archive (JAR) file in Forte. The former takes a user about 5 minutes to complete and the latter takes about 45 seconds when running locally using VMware. In our test, Visual Test plays back the keystrokes and mouse clicks as quickly as possible. Over a LAN with primed hashing, the interactive part takes only 1.3 minutes to complete and Forte takes 1.8 minutes. Over DSL with primed hashing, the interactive part takes 4.4 minutes and Forte takes 7 minutes. On both the DSL and LAN, the user sees an adequate interactive response. Forte is slower on DSL because it performs many small reads and writes. The reads are synchronous and sensitive to the slow DSL link. Also, the first write to a 16 KB COW block will incur a read of the block unless the write fills the block, which is rarely the case.

The processor simulation, kernel compile, and Apache tasks take about 3, 4, and 1 minutes, respectively, to execute when running under VMware locally. Without ballooning, these applications run mainly from memory, so remote execution on either LAN or DSL is no slower than local execution. Ballooning, however, can increase run time, especially in the case of Apache.

Our results show that active capsules can be migrated efficiently to support user mobility. For users with high-speed connectivity, such as students living in a university dormitory, memory images can be transferred without ballooning or hashing in just a few minutes. For users with DSL links, there are two separate scenarios. In the case of a commute between work and home, it is likely that an earlier capsule can be found at the destination, so that hashing can be used to migrate an unballooned memory image. However, to use a foreign capsule, ballooning is helpful to reduce the start-up time of many applications.

5.3 Active Versus Inactive Capsules

The use of capsules makes it possible for a machine in a Collective system to run an application without first having to install the application or even the operating system on which the application runs. It is also possible for a user to continue the execution of an active capsule on a different machine without having first to boot up the machine, log on, and run the application.

We ran experiments to compare these two modes of operation. These experiments involved browsing a webpage local to the capsule using Mozilla running on Linux. From the experiment results, we see that both active and inactive capsules are useful in different scenarios and that using capsules is easier and takes less time than installing the required software on the machine.

The different scenarios we considered are:

1. We mounted the inactive capsule file using NFS over the DSL link. We booted the inactive capsule, ran Mozilla, and browsed a local webpage. The results for this test are shown in Figure 7 with the label NFS.

2. We used demand paging to boot the inactive capsule and ran Mozilla to browse the local webpage. We considered three alternatives: the machine had not executed a similar capsule before and therefore had an empty hash cache, the machine had not executed the capsule before but the hash cache was primed with the disk state of the capsule, and the machine had executed the same capsule before and hence the capsule's shadow disk had the required blocks locally cached. The results are shown in Figure 7 un-

der the labels boot, boot-p, and boot2 respectively.

3. We activated an active remote capsule that was already running a browser. We ran it with and without ballooning, and with and without priming the hash cache with the inactive capsule disk. The results are shown in the figure under the labels active, active-b, active-p, and active-bp.

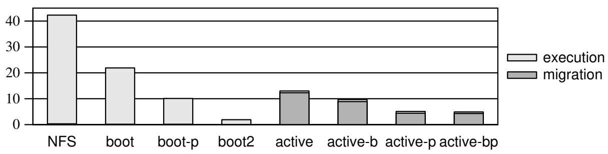

Figure 7: Times for activating a browser capsule (in minutes). The capsules are NFS, booted with an NFS-mounted disk; boot, a remote capsule booted with demand paging and unprimed database; boot2, the same remote capsule booted a second time; and active, migration of a capsule with a running browser. Suffix "b" indicates that ballooning was done and suffix "p" indicates that a primed database was used.

The bars in Figure 7 show the time taken while performing the test. The times are split into execution and migration times. As expected, the four inactive capsules in the first two scenarios have negligible migration times, while execution times are negligible for the four active capsules in the last scenario. When comparing the different scenarios we consider the total times as a sum of migration time and execution time.

In this experiment, demand paging, even with an empty hash cache, performed much better than NFS. Demand paging brought down the total time from about 42 minutes for NFS to about 21 minutes. When the cache was warmed up with a similar capsule, the total time for the inactive capsule dropped to about 10 minutes. When the inactive capsule was activated again with the required blocks locally cached in the capsule's shadow disk, it took only 1.8 minutes, comparable to boot of a local capsule with VMware. Using an active capsule with no ballooning or priming required about 12 minutes. Ballooning the active capsule brought the time down to about 10 minutes, and priming the hash cache brought it down further to about 4 minutes, comparable to the time taken to boot a local machine and bring up Mozilla. These times are much less than the time taken to install the required software on the machine.

These results suggest that: (a) if a user has previously used the inactive capsule, then the user should boot that capsule up and use it, (b) otherwise, if the user has previously used a similar capsule, the user should use an active capsule, and (c) otherwise, if executing the capsule for the first time, the user should use an active ballooned capsule.

5.4 Capsule Snapshots

We simulate the migration of a user between work and home machines using a series of snapshots based on the Business Winstone 2001 benchmark. These simulation experiments show that migration can be achieved within a typical user's commute time.

The Winstone benchmark exercises ten popular applications: Access, Excel, FrontPage, PowerPoint, Word, Microsoft Project, Lotus Notes, WinZip, Norton AntiVirus, and Netscape Communicator. The benchmark replays user interaction as fast as possible, so the resulting user session represents a time-compressed sequence of user input events, producing large amounts of stress on the computer in a short time.

To produce our Winstone snapshots, we ran one iteration of the Winstone test suite, taking complete images of the machine state every minute during its execution. We took twelve snapshots, starting three minutes into the execution of the benchmark. Winstone spends roughly the first three minutes of its execution copying the application programs and data it plans to use and begins the actual workload only after this copying finishes. The snapshots were taken after invoking the balloon process to reduce the user's memory state.

To simulate the effect of a user using a machine alternately at work and home, we measured the transfer of snapshot to a machine that already held all the previous snapshots. Figure 8 shows the amount of data transferred for both the disk and memory images of snapshot 2 through 12. It includes the amount of data transferred with and without hashing, and with and without gzip.

The amount of data in the COW disk of each snapshot varied depending on the amount of disk traffic that Winstone generated during that snapshot execution. The large size of the snapshot 2 COW disk is due to Winstone copying a good deal of data at the beginning of the benchmark. The size of the COW disks of all the other snapshots range from 2 to 22 MB after gzip, and can be transferred over completely under about 8 minutes. Whereas hashing along with gzip compresses the COW disks to about 10–30% of their raw size, it compresses the memory images to about 2–6% of their raw size. The latter reduction is due to the effect of the ballooning process writing zero pages in memory. The sizes of ballooned and compressed memory images are fairly constant across all the snapshots. The memory images require a transfer of only 6–17 MB of data, which takes no more than about 6 minutes on a DSL link. The results suggest that the time needed to transfer a new memory image, and even the capsule disk in most cases, is well within a typical user's commute time.

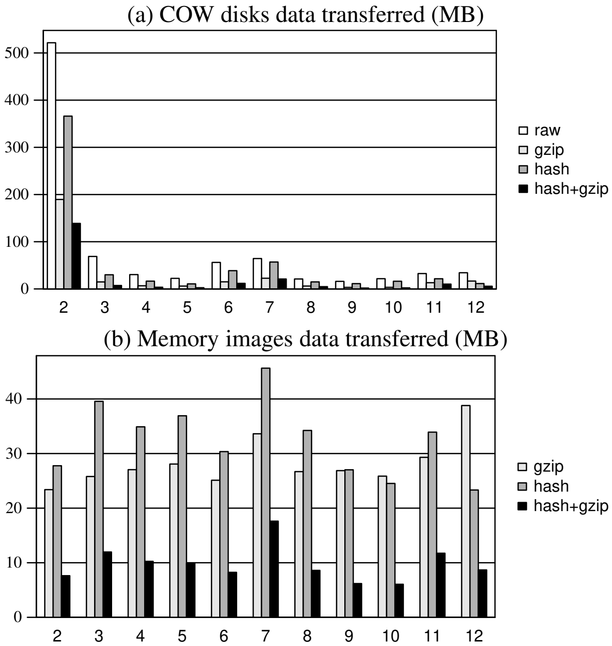

Figure 8: Snapshots from the Winstone benchmark showing (a) disks and (b) memory images transferred. Raw sizes not shown in (b) are constant at about 256 MB.

6 Related Work

Much work was done in the 1970s on virtual machines at the hardware level[9] and interest has recently revived with the Disco[4] and Denali[30] projects and VMware GSX Server[28]. Earlier work demonstrated the isolation, performance, and economic properties of virtual machines. Chen and Noble suggested using hardware-level virtual machines for user mobility[6]. Kozuch and Satyanarayanan independently came up with the idea of using VMware's *x*86 VMMs to achieve mobility[15].

Others have also looked at distributing disk images for managing large groups of machines. Work by Rauch et al. on partition repositories explored maintaining clusters of machines by distributing raw disk images from a central repository[22]. Rauch's focus is on reducing the size of the repository; ours is on reducing time spent sending disk images over a WAN. Their system, like ours, reduces the size of successive images by storing only the differences between revisions. They also use hashes to detect duplicate blocks and store only one copy of each block. Emulab[31], Cluster-on-Demand[18], and others, are also distributing disk images to help maintain groups of computers.

The term *capsule* was introduced earlier by one of the authors and Schmidt[24]. In that work, capsules were implemented in the Solaris operating system and only groups of Solaris processes could be migrated.

Other work has looked at migration and checkpointing at process and object granularities. Systems working at process level include V[26], Condor[16], libckpt[21], and CoCheck[25]. Object-level systems include Legion[10], Emerald[14], and Rover[13].

LBFS[19] provided inspiration for HCP and the hash cache. Whereas LBFS splits blocks based on a fingerprint function, HCP hashes page-aligned pages to improve performance on memory and disk images. Manber's SIF[17] uses content-based fingerprinting of files to summarize and identify similar files.

7 Conclusions

In this paper, we have shown a system that moves a computer's state over a slow DSL link in minutes rather than hours. On a 384kbps DSL link, capsules in our experiments move in at most 20 minutes and often much less.

We examined four optimization techniques. By using copy-on-write (COW) disks to capture the updates to disks, the amount of state transferred to update a capsule is proportional to the modifications made in the capsule. Although COW disks created by installing software can be large, they are not much larger than the installer and more convenient for managing large numbers of machines. Demand paging fetches only the portion of the capsule disk requested by the user's tasks. "Ballooning" removes non-essential data from the memory, thus decreasing the time to transfer the memory image. Together with demand paging, ballooning leads to fast loading of new capsules. Hashing exploits similarities between related capsules to speed up the data transfer on slow networks. Hashing is especially useful for compressing memory images on user commutes and disk images on software updates.

Hopefully, future systems can take advantage of our techniques for fast capsule migration to make computers easier to use and maintain.

8 Acknowledgments

This research is supported in part by National Science Foundation under Grant No. 0121481 and Stanford Graduate Fellowships. We thank Charles Orgish for discussions on system management, James Norris for working on an earlier prototype, our shepherd Jay Lepreau, Mike Hilber, and David Brumley for helpful comments.

References

[1] FIPS 180-1. Announcement of weakness in the secure hash standard. Technical report, National Institute of Standards and Technology (NIST), April 1994.

[2] A. C. Arpaci-Dusseau and R. H. Arpaci-Dusseau. Information and control in gray-box systems. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 43–59, October 2001.

[3] A. A. Awadallah and M. Rosenblum. The vMatrix: A network of virtual machine monitors for dynamic content distribution. In *Seventh International Workshop on Web Content Caching and Distribution*, August 2002.

[4] E. Bugnion, S. Devine, and M. Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM Transactions on Computer Systems*, 15(4):412–447, November 1997.

[5] F. Chabaud and A. Joux. Differential collisions in SHA-0. In *Proceedings of CRYPTO '98, 18th Annual International Cryptology Conference*, pages 56–71, August 1998.

[6] P. M. Chen and B. D. Noble. When virtual is better than real. In *Proceedings of the 8th IEEE Workshop on Hot Topics on Operating Systems*, May 2001.

[7] http://www.connectix.com/.

[8] P. Deutsch. Zlib compressed data format specification version 3.3, May 1996.

[9] R. P. Goldberg. Survey of virtual machine research. *Computer*, 7(6):34–45, June 1974.

[10] A. Grimshaw, A. Ferrari, F. Knabe, and M. Humphrey. Legion: An operating system for wide-area computing. Technical Report CS-99-12, Dept. of Computer Science, University of Virginia, March 1999.

[11] IA-32 Intel architecture software developer's manual volumes 1-3. http://developer.intel.com/design/pentium4/manuals/.

[12] *IBM Virtual Machine/370 Planning Guide*. IBM Corporation, 1972.

[13] A. Joseph, J. Tauber, and M. Kaashoek. Mobile computing with the Rover toolkit. *IEEE Transactions on Computers*, 46(3):337–352, March 1997.

[14] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transaction on Computer Systems*, 6(1):109–133, February 1988.

[15] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 40–46, June 2002.

[16] M. Litzkow, M. Livny, and M. Mutka. Condor – a hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, June 1988.

[17] U. Manber. Finding similar files in a large file system. In *Proceedings of the USENIX Winter 1994 Technical Conference*, pages 1–10, 17–21 1994.

[18] J. Moore and J. Chase. Cluster on demand. Technical report, Duke University, May 2002.

[19] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, pages 174–187, October 2001.

[20] M. Muuss. The story of T-TCP.

[21] J. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: Transparent checkpointing under Unix. In *Proceedings of the USENIX Winter 1995 Technical Conference*, pages 213–224, January 1995.

[22] F. Rauch, C. Kurmann, and T. Stricker. Partition repositories for partition cloning—OS independent software maintenance in large clusters of PCs. In *Proceedings of the IEEE International Conference on Cluster Computing 2000*, pages 233–242, 2000.

[23] L. Rizzo. Dummynet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):31–41, Jan. 1997.

[24] B. K. Schmidt. *Supporting Ubiquitous Computing with Stateless Consoles and Computation Caches*. PhD thesis, Computer Science Department, Stanford University, August 2000.

[25] G. Stellner. CoCheck: Checkpointing and process migration for MPI. In *Proceedings of the 10th International Parallel Processing Symposium*, pages 526–531, April 1996.

[26] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable remote execution facilities for the V-system. In *Proc. 10th Symposium on Operating Systems Principles*, pages 10–12, December 1985.

[27] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, April 2000.

[28] "GSX server", white paper. http://www.vmware.com/pdf/gsx_whitepaper.pdf.

[29] C. A. Waldspurger. Memory resource management in VMware ESX server. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December 2002.

[30] A. Whitaker, M. Shaw, and S. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. Technical report, University of Washington, February 2001.

[31] B. White et al. An integrated experimental environment for distributed systems and networks. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December 2002.

[32] Wintel architecture specifications. http://www.microsoft.com/hwdev/specs/.