

**Optimizing Live Virtual Machine Migrations
using Content-based Page Hashes**

by

Jacob A. Stultz

S.B. EECS, M.I.T., 2008

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

August, 2008

©2008 Massachusetts Institute of Technology

All rights reserved.

Author _____
Department of Electrical Engineering and Computer Science
August 29, 2008

Certified by _____
Carl Waldspurger
Principal Engineer
VMware Inc.
VI-A Company Thesis Supervisor

Certified by _____
Larry Rudolph
Principal Research Scientist
Computer Science & Artificial Intelligence Lab
M.I.T. Thesis Supervisor

Accepted by _____
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

Optimizing Live Virtual Machine Migrations using Content-based Page Hashes

by

Jacob A. Stultz

Submitted to the

Department of Electrical Engineering and Computer Science

August 29, 2008

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Virtualization systems such as VMware ESX Server use content-based page hashing to help identify duplicate memory pages within virtual machines. Such duplicate pages, once identified, can then be mapped copy-on-write to a single page in physical RAM, thus saving memory in a manner that is transparent to the virtual machine. The page hashes that are collected in this process can be further utilized. This thesis demonstrates how these page hashes can be used to reduce the time required to migrate running virtual machines from one physical machine to another. Normally, this is done by sending the virtual machine state and all memory contents from the source to destination machine. However, since some memory pages may already be present on the destination, it is possible to reduce the number of pages sent (and therefore total migration time and bandwidth used), by sending only a compact hash instead of the full contents for each page likely to be present on the destination machine. This thesis accomplishes this by creating a database of canonical or “standard” pages to determine which pages can be safely sent as only a hash. Tests of the system demonstrate up to 70% reduction in migration time in idealized workloads, 40% reduction in some realistic workloads, and minimal slowdown in some pathological cases.

Thesis Supervisor: Larry Rudolph

Title: Principal Research Scientist, Computer Science & Artificial Intelligence Lab

Acknowledgments

This thesis was the result, in part, of an internship at VMware, Inc. as part of MIT's VI-A program, so I owe a lot of thanks to all of those at VMware who have helped me along the way.

In particular, I would like to thank Carl Waldspurger and Rajesh Venkatasubramanian for their guidance and assistance from the initial conception to final completion of this project. Their help was invaluable in getting me up to speed with VMware's software and brainstorming ideas, particularly in the kernel and memory management areas.

Ali Mashtizadeh was a great help with the VM migration component of the project. He made himself available as an expert on migration and did his best to help me at every turn.

Thanks to everyone else at VMware who has helped me at any point, as well as the company itself, for providing me with the resources and environment with which to pursue this thesis.

Larry Rudolph, my thesis supervisor, has been instrumental in project planning and the writing of this document, in addition to frequently providing valuable technical input from a different perspective.

Anne Hunter, the course administrator, has been absolutely wonderful and understanding throughout my undergraduate and graduate education here, and particularly helpful and encouraging in the final few weeks.

The MIT Ski Team has been one of my favorite activities throughout my years at MIT, allowing me to continue competing in my favorite childhood sport; a part of my life that I thought was over after high school. I'm particularly grateful to my coach, Todd Dumond, and teammate Tim Pier for convincing me to continue racing through my graduate year. Skiing is one of the few things keeping me sane.

Beta has always provided a welcoming home for me, and I'm extremely glad to have so many lasting friendships as a result. Seeing all of the amazing things that Betas have done over the years has kept me motivated more than anything else.

Josh Wilson has also been a great motivator over the past term, always with an open ear to listen to issues I've had and sympathize with frustrations.

Finally, I would like to thank my family for their unending support and encouragement throughout my life, but especially during the past 5 years of my education at MIT.

Table of Contents

1	Introduction.....	8
2	Background.....	9
	2.1 Virtualization Overview.....	9
	2.2 Memory Management.....	10
	2.3 Migration.....	13
	2.4 Extending Page Sharing to Migrations.....	16
3	Design.....	18
	3.1 Standard Pages.....	18
	3.2 VMFS Database Module.....	20
	3.2.1 File Structure.....	21
	3.2.2 Interface and Implementation of VMkernel module.....	23
	3.3 VMkernel and Page Sharing Changes.....	31
	3.3.1 Standard Page Daemon Thread.....	32
	3.4 Migration Changes.....	37
	3.4.1 Migration Source.....	37
	3.4.2 Migration Destination.....	38
4	Evaluation.....	40
	4.1 Factors affecting performance.....	40
	4.1.1 Costs of standard page discovery.....	40
	4.1.2 Costs and benefits to migrations.....	41
	4.2 Benchmark Design.....	42
	4.3 Benchmark Results.....	44
5	Related Work.....	48
6	Summary.....	51
7	Bibliography.....	55

List of Figures

Figure 1.	Layout of a virtualized system.....	9
Figure 2.	Virtualized memory.....	11
Figure 3.	Page sharing.....	13
Figure 4.	Migration.....	14
Figure 5.	Migration precopy stage.....	16
Figure 6.	Database file layout.....	22
Figure 7	<i>Contains</i>	28
Figure 8.	<i>Write</i>	30
Figure 9.	<i>Read</i>	30

1. Introduction

The independence of virtual machines from physical hardware gives them great flexibility, allowing them to be transparently migrated between host machines even while running. The performance of these migrations can be improved by reducing the amount of data sent during the migration. This thesis exploits existing page sharing infrastructure to avoid sending contents of pages that are already present on the destination, thereby sending less total data. The general concept of this improvement, VMOSH (Virtual machine Migration Optimization using Standard Hashes) is straightforward, but of course there are trade-offs involved. There are costs associated with determining which pages might not need to be sent, and in order for this scheme to be beneficial, the savings must outweigh the costs. This balance is tested by implementing the concept in existing software capable of performing migrations. VMware ESX Server was chosen for this purpose because it has the necessary patented page sharing infrastructure.

Relevant details about virtualization design, particularly in VMware ESX Server, and the page sharing and migration systems are described in section 2. This provides the necessary background information to understand the design and implementation details of this thesis project, which are explained in section 3. Section 4 contains an evaluation of the VMOSH system, including benchmark comparisons and analysis. Previous related work is described in section 5, followed by a summary and description of potential future work in section 6. Section 7 contains a bibliography.

It may be helpful to think of this work as a type of data compression; the source has a large set of unencoded data that must be sent to the destination. If the destination already has a particular page of data, then it is only necessary to send some sort of identifier for that page, which can be thought of as encoded data.

2. Background

There are two key concepts enabled by virtualization that are central to this thesis: transparent content-based page sharing, and live virtual machine migration. First, however, a broad overview of virtualization technology is necessary to lay the groundwork for page sharing and migration. With that in place, an in depth description of page sharing and other relevant memory management techniques followed by a similar explanation of virtual machine migration are given. These are the existing mechanisms that lead into the design of VMOSH.

2.1 Virtualization Overview

Virtualization allows users to run multiple operating systems concurrently on the same physical hardware, effectively acting as a multiplexer of hardware resources. A software program called a hypervisor presents each operating system (guest OS) with a virtual hardware interface (a virtual machine, or VM), and isolates each operating system from the others. The hypervisor does not simply emulate the hardware, it allows most instructions to run natively on the underlying hardware, trapping and handling privileged instructions, much in the same way that an operating system does for applications.

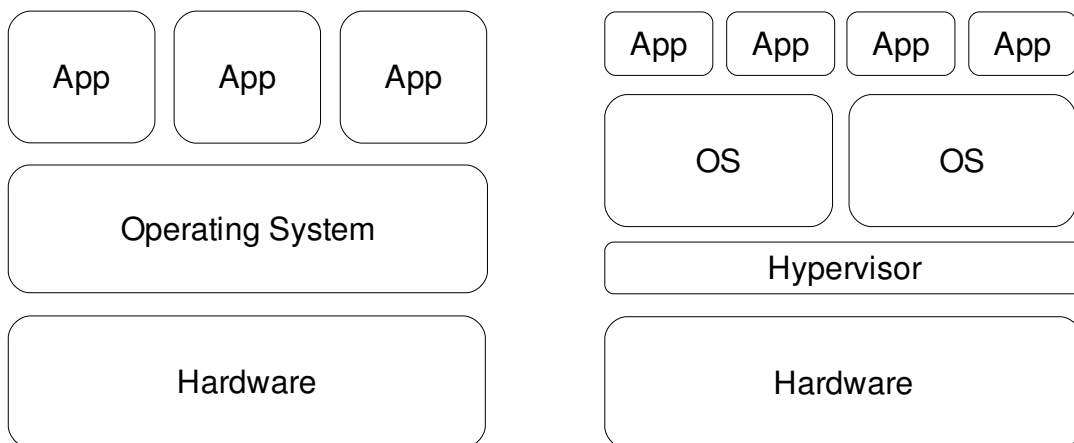


Figure 1: Layout of a non-virtualized system (left) compared with a type 1 hypervisor (right).

The hypervisor layer allows for control over operating systems in an analogous manner to the control an operating system has over applications. This control leads to potentially more efficient and flexible use of hardware resources in terms of scheduling, load balancing, energy usage, and memory management. There are two main categories of hypervisors (Figure 1). Type 1 hypervisors, also known as native or bare-metal, run directly on the hardware, and act as the native operating system of the computer, directly interacting with hardware devices. Type 2 hypervisors, on the other hand, run as an application within some other operating system.

This research was conducted in the context of VMware ESX Server, which is a type 1 hypervisor. Running directly on the hardware with its own kernel (VMkernel) allows VMware ESX Server to be used efficiently as a hypervisor in enterprise server environments with large numbers of virtual machines running unmodified guest operating systems, such as Windows, Linux, and Solaris. Because the hypervisor controls the hardware instead of the normal operating system, the hypervisor must implement ways to manage memory and other hardware so that the guest OS still functions normally without modification.

2.2 Memory Management

Hypervisors typically insert an extra layer of memory management [2,8] to the well established virtual memory architecture (Figure 2). For example, the x86 architecture contains a memory management unit (MMU) that translates the virtual addresses to physical addresses, used directly by hardware, with a page table that is maintained by the operating system. The hardware translates a virtual address by splitting the address into a virtual page number (VPN) and an offset into the page. The VPN is translated into a physical page number (PPN) via the page table data structure. In a virtualized system, the guest operating system still maintains these page tables, however the physical page numbers that the guest OS sees (guest physical page numbers, or GPPNs) do not directly correspond to actual physical

memory. Instead, they correspond to the virtualized memory that the hypervisor presents. The hypervisor then translates the GPPNs that the guest operating systems use into what are termed “machine page numbers” or MPNs, which in turn directly correspond to host physical pages in hardware memory.

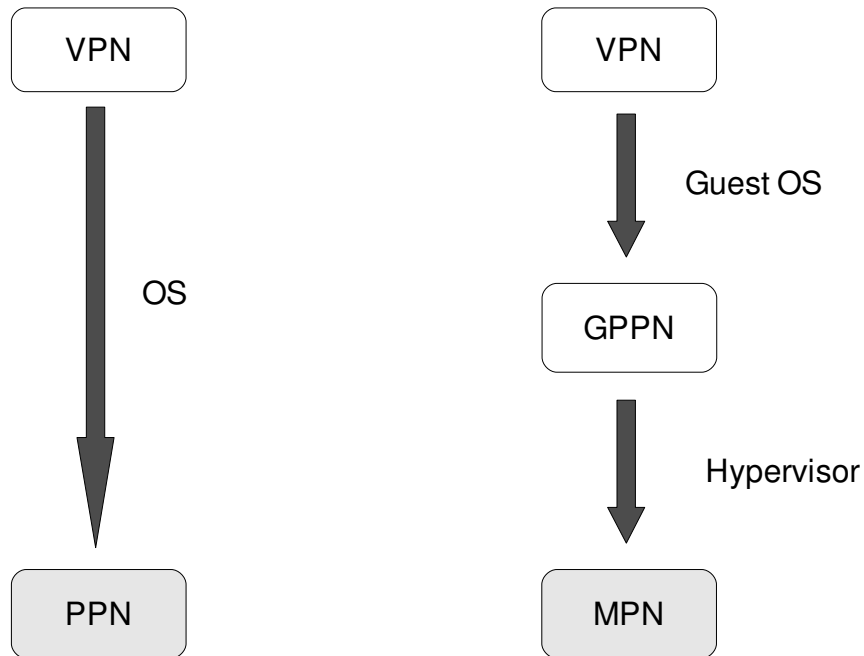


Figure 2: In a non-virtualized machine, the operating system translates virtual page numbers (VPNs) to physical page numbers (PPNs), representing hardware addresses. In the virtualized system on the right, however, the guest operating system translates the VPns to guest physical page numbers (GPPNs), which are then translated into machine page numbers (MPNs) by the hypervisor, which correspond to hardware addresses.

This extra layer of indirection in memory allows for some useful memory management techniques, including transparent swapping, page remapping, and page sharing. On any given physical machine running a hypervisor, there may be many different virtual machines running the same guest operating system, running the same application, or working with the same set of data. As such, there may be many pages of memory across these virtual machines with identical content. There is no need to maintain this many redundant copies of pages. This idea is exploited by VMware ESX Server's content-based page sharing, which identifies these identical pages, and maps the corresponding guest physical

pages to the same hardware machine page, thus allowing VMware ESX Server to conserve a significant amount of machine memory [8,9]. This assists other memory management techniques in allowing VMware ESX Server to overcommit memory; that is, to run multiple virtual machines on one system with lower total hardware RAM than the sum of all memory configured for each individual VM.

The page sharing component of the VMware ESX Server kernel detects these identical pages via periodic memory sampling. During the sampling, a hash value of a GPPN's contents is generated using an efficient 64-bit hash function [5], and then used as an index into a table containing hashes of pages that have already been discovered. For each match, a full byte-for-byte comparison of the two pages is done to handle the possibility of a hash collision. If identical, the new page is mapped to the same MPN as the one already in the hash table, and is marked copy-on-write, to prevent data loss or corruption. If a page hash is not found in the table, it is added. Marking the page copy-on-write could result in unnecessary copying if the page is modified, so it is left writable and marked as a "hint." If a duplicate page is found before the first page is modified, it will then be marked copy-on-write. In case the page has been modified, if a hash match is discovered, the hash of the hint page is recalculated. If the hash has changed, then the page has been modified and it is removed from the table. The paper that introduced content-based page sharing demonstrated that in idealized situations, memory savings of up to two-thirds can be seen, and real world deployments can save as much as a third of total memory[8].

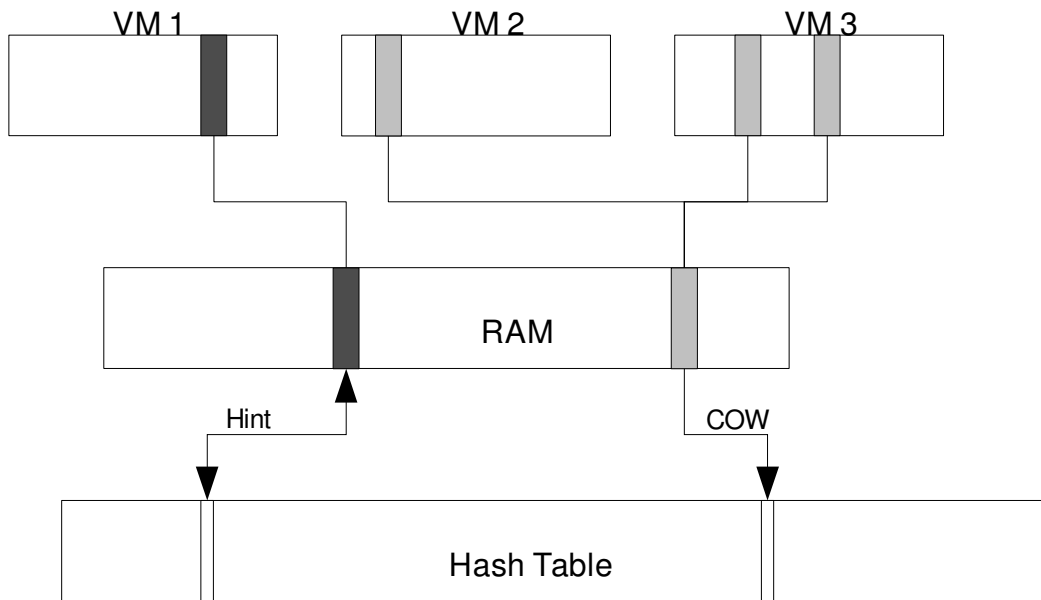


Figure 3: Pages with a single reference are not marked copy-on-write (COW), but are instead marked as a “hint” in the hash table with a reference back to the MPN, which is necessary to verify that the page contents haven’t changed if a second page with the same hash is found. Pages with multiple references are all mapped to the same COW page in hardware, and their page table entries are marked read-only.

2.3 Migration

Another feature enabled by virtualization is live migration of running virtual machines between servers. This can be done quickly enough and with sufficiently short downtime that there is no noticeable interruption of service in a production environment [3,6]. Migration can be used for a number of things, including load balancing (automated or otherwise), power management, or shutting down machines for maintenance without powering down the virtual machines running on them.

On VMware ESX Server, migration is called vMotion, and can be done between any two physical hosts with sufficiently similar hardware. This feature is also present on other virtualization technologies such as Xen and KVM, but VMOSH is implemented with vMotion. The disk images for any virtual machines to be migrated, representing the virtual disks attached to those VMs, must be located on storage shared between the two hosts. This is typically over iSCSI or a Fibre Channel SAN, and the

partition is formatted with the Virtual Machine File System (VMFS), a clustering file system designed for VMware ESX Server [10]. When a migration is initiated, the source sends the memory contents and machine state of the VM to the destination host, and the VM continues to access the disk on the shared storage from the destination host, eliminating the need for the transfer of the VM's virtual disk contents.

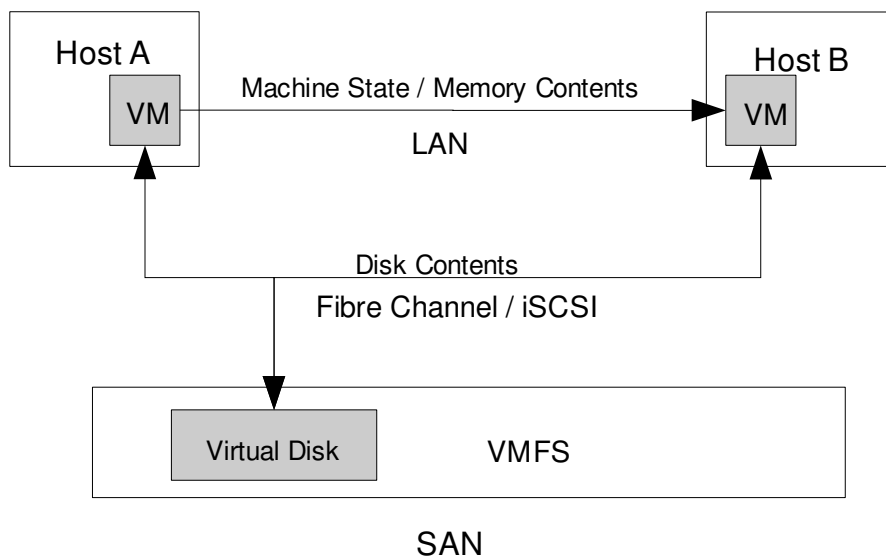


Figure 4: When a VM is migrated (in this case from Host A to Host B), memory contents and machine state are copied over the network, whereas the VM's disk contents are not copied at all. Instead, the virtual disk image for the VM is located on shared storage accessible to both hosts, and can be utilized as necessary by both hosts during the migration process.

The transfer of the memory contents is the interesting part of this process. Machine state (register contents, virtualized device states, etc) is small enough that it can be read and transferred without a noticeable delay. The virtual disks are easily taken care of by placing them on shared storage. Memory contents, however, are large enough that they cannot be sent instantaneously; typical VMs are configured with several gigabytes of memory. In order for the migration to be transparent to the VM and to appear as if the VM has not been interrupted, the VM is left running while the memory contents are transferred. This poses the additional issue that the memory contents are constantly changing while being copied. To solve this problem, vMotion occurs largely in two stages: precopy and checkpoint/transfer.

The precopy stage is iterative. To begin, pages are sent from the source to the destination, but first a trace [1] is placed on each page to detect if it has been modified. Placing a trace on a page is essentially marking it read-only; if the VM attempts to write to that page after it is copied, it will be marked dirty again, and therefore must be re-copied. After all pages are sent, a new phase is started wherein all newly marked dirty pages are sent and traced again. This continues iteratively until the set of dirty pages is small enough that they can all be sent quickly enough that operation of the virtual machine does not appear to be interrupted; typically less than a second [6].

At this point, the VM is quiesced (halted on the source after all in-flight I/O is completed), and the remaining dirty pages are transferred to the destination host along with the machine state. The destination host then has all of the necessary components to start the virtual machine back up as if it was never stopped or relocated.

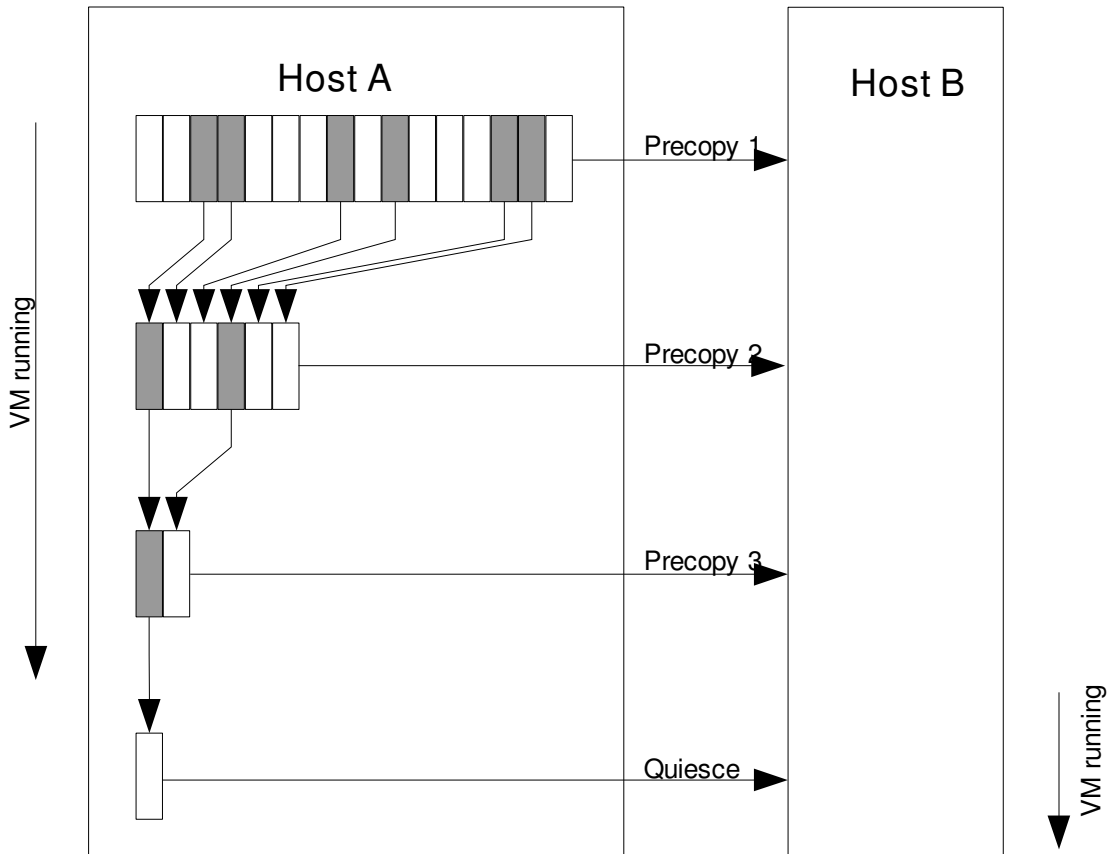


Figure 5: In the first precopy stage, a trace is placed on each page before they are all sent to the destination host. After all pages are sent, each page that has been dirtied (gray) is sent again and a trace is re-installed on those pages. This process continues iteratively until the number of dirty pages is small enough that the VM can be halted on the source and started on the destination after the remaining pages are sent without a noticeable interruption.

2.4 Extending Page Sharing to Migrations

Presently, in VMware ESX Server 3.x, the full contents of every page are sent except for pages containing entirely zeros (the zero page). However, just as multiple virtual machines on one host may be running the same guest operating system or applications, the same is true across hosts. If a Windows XP virtual machine is being migrated to a host running multiple other instances of Windows XP, it is likely that a large number of pages in the virtual machine being migrated may already exist on the destination host.

In light of this, it should be beneficial to send only page hashes (and not full contents) for pages that are shared on the source host and are therefore considered likely to be present and shared on the

destination host as well. The transparent page sharing already generates hashes for all of the shared pages, so those do not have to be recalculated for a migration. A similar idea has been proposed [7], but the potential issue of hash collisions was not addressed, which is essential for preventing data corruption, an important issue for production servers. This is explained in more detail along with a more complete discussion of related work in Section 5.

3. Design

Ideally, the source host in a migration would only send pages to the destination that were not already there. A simple attempt at this would be to start a migration by sending a full set of hashes for all pages in the migrating VM. For any page hash that the destination does not have, it could send a request back to the source for the full contents, thus ensuring that no redundant pages are sent.

However, there is one major obstacle to this straightforward process of simply sending hashes instead of contents for shared pages. A 64-bit hash cannot uniquely represent a 4 kilobyte page, so there is the possibility of a collision. If different pages on the source and destination hosts have the same 64-bit hash, measures need to be taken to ensure that the source page being sent is not mistaken as the different page on the destination. Using a larger cryptographic hash would make such a hash collision even more unlikely, but would require more computation, memory, and would still not guarantee against corruption. There must therefore exist some mechanism to ensure that the hosts involved in a migration share a consistent view of the hashes being sent.

3.1 *Standard Pages*

The central concept of this thesis is the idea of a canonical or *standard* page for a given hash. The purpose of the standard page is to establish a consistent view of hashes sent during a migration. As such, there is at most one standard page that hashes to any given value. If all hosts agree on this standard page for each hash, hash collisions can be detected and dealt with appropriately. The standard pages for each hash are chosen from the page sharing tables. If page is shared, it is most likely the most common page with that hash value, and is therefore most likely the best standard page candidate, especially since hash collisions should be extremely rare in practice. Additionally, shared pages are relatively unlikely to change; if a given page changed frequently, it would be much less likely that another

identical page would be found. Using the page sharing tables to find standard pages is convenient, since it does all of the VM memory scanning and page hashing. It does have the disadvantage of limiting standard pages to those that have been shared. There are likely to be some pages that are unique within a host but not across all VMs that might be involved in a migration; these pages will be missed by VMOSH. Extending the search to every potential standard page, however, would require significantly more overhead, since significantly more pages would be involved, and hashes would have to be computed for them.

A database is used as the definitive authority on standard pages. The database, located on shared storage like the virtual machine disk images, stores mappings between page hashes and the full contents of these standard pages. This database and its interface together are one of three main components to this migration optimization. The second and third components to the optimization are changes to the kernel page sharing and memory management code to account for the concept of the standard page, and modifications to the migration module to send hashes instead of contents for standard pages.

On each host, there is a background thread that chooses standard page candidates from that host's page sharing tables, which contain shared page information for all of the VMs on that host. These standard page candidates are sent to the database module, which either checks the database for that page's presence, or attempts to add it; the module attempts to add the more highly shared pages while only querying for the rest. All pages that the database module indicates are present in the database (whether newly added or already there) are marked in the page sharing tables as standard. This ensures that later, when a migration is started, the source host knows which pages are standard and can safely be sent only as hash values. The destination host may not already have a particular standard page sent by the source, or may not yet have it marked standard. If this is the case, it requests the full page contents from the source, but could alternatively read the page contents directly from the database.

This scheme may result in an overall increase in cumulative data transfer, both between hosts over the network and between the hosts and the shared storage, compared with simply sending all pages over the network during a migration. This is primarily because of the increased volume of data transfer to and from the shared storage. It is a trade-off, however: the primary goal of VMOSH is to decrease the latency of live migrations, not to reduce overall network traffic. Even still, data transfer will likely be reduced over time. Each page only needs to be checked against the database once per host; once a shared page is marked standard on a host, additional GPPNs from any VM on the host can be later mapped to it without being checked against the database. If the number of migrations is sufficiently high, the savings made by transferring only the hash for each standard page may fully amortize or even significantly exceed the cost of the database accesses.

3.2 VMFS Database Module

The standard page database takes the form of a set of files on shared storage, and a dynamically loadable VMkernel module to provide an interface to them. The primary purpose of the module is to ensure that all hosts involved in a migration agree on which full pages are represented by the standard page hashes. This is a relatively simple task in the data complexity sense, so the module itself is also fairly simple. There is a file to store page hashes, a file to store page contents, and a lock file. The module provides an interface to query for the presence of a page in the database, write a page to the database, and read a page out from the database.

Pages can only be added to the database, not modified or removed. The complexity of allowing modifications or deletions is significantly greater than only allowing additions. Allowing pages to be removed from the standard page database, or allowing the standard page for a given hash to be changed would create a coherency issue, if that page has been marked standard by another host. Each host has a local representation of a subset of the database in the form of pages marked standard in the page sharing

tables. If a page is removed from the database, this must be communicated to every host so that they can mark the page as non-standard. If one host changed a standard page in the database, and then began the migration of a virtual machine containing that page before the destination host was notified of the change in the database, the destination host would interpret the hash sent as the stale standard page in the database, and not the newly updated page. Since hash collisions that would make deletions or modifications to the database necessary are extremely rare, and most environments have an abundance of disk storage, it has been decided to only allow the addition of new pages to the database. Another improvement that could be made that would help deal with the lack of deletion and modification ability is multiple databases. Instead of marking pages as standard, they could be marked with a standard database identifier, making it possible to have multiple different standard pages with the same hash.

3.2.1 File Structure

The general structure of the standard page database is an on-disk hash table. The page hash file serves as the hash table, and stores mappings between the page hashes and indices into the page contents file. The file is of a fixed size, and uses the lower order bits of the page hash as an index to each pair in the table. However, since the page contents are stored separately, it would be relatively simple to grow the hash table dynamically, if necessary. Each hash table entry consists of the 64-bit hash of that standard page, and a 64-bit index into the page contents file. The size of the file is determined by the number of lower order bits used as an index, which is a configurable option. If there are k bits used as an index, then there are 2^k entries in the file, and the total file size is $16(2^k)$ bytes. Hash table collisions are resolved by open-addressing with linear probing, with a predetermined probe limit [4]. Open addressing is used in this implementation because it is much simpler to write the logic for a flat, fixed-size file using open addressing rather than an alternative such as chaining, though chaining would eliminate the need for the probe limit. If the probe limit is actually hit, it would be a good indication that the hash table size should be increased if dynamically growing the hash table is implemented as discussed above.

files are closed, the lock file is closed. This is necessary for writes to ensure that two hosts do not attempt to add a hash table entry in the same slot, or append page contents to the page file at the same time. In the case of a read, the lock file is still opened, but in read only mode, instead of exclusive mode. This way, many hosts can read the database at the same time, but the database is protected against a reader seeing partial writes to the database. Whenever a host is writing to the database files, it is guaranteed that no other hosts have the files open at that time. An alternative method, not used here, to deal with these concurrency issues would be to have a single server running a database process instead of having each host access the database files directly. This would simplify the locking process, and eliminate the need for disk-level locks, which could improve performance if the overhead of an intermediary was small enough.

3.2.2 Interface and Implementation of VMkernel module

The standard page database module makes five functions externally accessible: module initialization and cleanup routines, and three database access functions. The access functions are *Contains*, *Write*, and *Read*.

- The *Contains* function checks for the presence of a specific page. It is used for standard page candidates with a low degree of sharing that should not necessarily be added to the database; if the reference count for the page is below the write threshold value, it is not written and instead *Contains* is called.
- The *Write* function attempts to add a new page to the database. It is called for standard page candidates with a high degree of sharing, where the reference count is at least as high as the write threshold value.
- The *Read* function, given a page hash, will retrieve the page contents associated with that hash,

if available.

The distinction between the *Contains* and *Read* function is that *Contains* is used to determine if a given known page is already present in the database, and simply returns true or false. *Read*, however, retrieves from the database the contents of an unknown page associated with a given hash.

In order for the three database interface functions to be made accessible, the module initialization function (*init_module*) must provide the kernel with pointers to those functions. The kernel contains a structure with function pointers to each of the three functions, initially unset. When the module is loaded, the *init_module* function populates the struct with these function pointers. If the module is later unloaded, the *cleanup_module* function sets all of the function pointers back to null. Whenever the kernel attempts to call one of the module functions, the appropriate function pointer value is checked; if null, the function returns immediately, otherwise the module function is executed. Module unloading makes use of synchronization necessary to ensure that once the module is unloaded, the function pointers will be null on all processors, to prevent a function from being called after unload.

The module has one main internal structure, containing file handles for all three database files, as well as a boolean variable indicating whether the database is already present on the disk, and a variable containing the current number of entries in the database (equivalent to the index of the next page to be added to the page contents file.) The *init_module* function sets these values to false and zero, respectively. Performing the blocking disk accesses necessary to determine appropriate values for these variables is not feasible in the context in which the *init_module* function is called. Instead, determining those values is deferred to the first call of any of the three database access functions. All three of the functions first check if the structure indicates that the database is present. If it doesn't, they look for the database files on disk. If the files are not there, a function is called to create them (*CreateDB*), and then in either case the structure is updated to reflect that the database is present. All hosts use the same

filename and path for all three files.

CreateDB first creates the lock file, and then attempts to open it in exclusive mode, waiting until it successfully does so (to prevent another host from trying to create the database files at the same time). The hash table file is created and populated with zeros, the page contents file is created and left empty, and then all files are closed in the reverse order in which they were opened.

The three externally accessible database interface functions make use of four core internal functions which actually perform the file input and output: *ReadHash*, *WriteHash*, *ReadPage*, and *WritePage*. Their uses and operation are relatively straightforward; the specifics are detailed below. All four functions return a return status value, which is OK if the function executes successfully. If there is an error reading from the database files, the return status is given by the kernel file I/O function.

ReadHash (*index* IN, *hash* OUT, *pageIndex* OUT) :

This function provides the direct read interface to the hash table file. Given a 32-bit hash table index, it reads the 64-bit hash value and page file index for that entry and returns both values. If the entry for the given hash table index is not populated, zero is returned for both values, indicating an available hash table entry. Since the hash table uses open addressing and linear probing, there will frequently be consecutive calls to *ReadHash* on contiguous portions of the hash table. It would therefore be inefficient for *ReadHash* to open and close the hash table file every time it is called. Instead, a precondition for calling *ReadHash* is having the hash table file open for reading. This way, the file is opened, and a number of consecutive reads made before the file is closed, instead of opening and closing the file for each individual query of the hash table.

WriteHash (*index* IN, *hash* IN, *pageIndex* IN) :

WriteHash is the complementary function to *ReadHash*. Given a hash table index, a hash value,

and a page file index, the hash value and page file index are written to the specified index in the hash table file. The disk sector size is 512 bytes, however, so disk I/O must be performed on 512 byte chunks. To deal with this, the disk sector containing the entry to be written to is read from the disk into memory, modified to contain the new hash and page file index pair, and written back to disk. Any write to the hash table file is inevitably preceded by a corresponding read (to ensure that the hash table entry is available), so like *ReadHash*, *WriteHash* also requires that the hash table file be opened (in this case, for writing) before it is called.

ReadPage (*pageIndex* IN, *page* OUT) :

Given a page file index, *ReadPage* reads and returns the 4-KB of page data stored at that index in the file. It is a much simpler function than *ReadHash* and *WriteHash*, as the page size is an integer multiple of the disk sector size and can be read into memory from the disk with one command. Another key difference, since consecutive page reads won't occur frequently compared to consecutive hash table reads, is that *ReadPage* does not assume that the page file is already opened; instead, it opens it for reading and closes it once the page contents are read.

WritePage (*pageIndex* IN, *page* IN) :

Naturally, this is the complementary function to *ReadPage*. *WritePage* appends the full contents of a given page to the end of the page file. It first opens the page file for writing, then reads the file attributes to determine the current length of the file, and therefore the appropriate index and file offset to write the new page. The file size is incremented by 4-KB (one page), and the new page is written in the newly created space. Here again, since the page size is an integer multiple of the disk sector size, it is not necessary to read any data (besides the length metadata) from the file before writing, as it was with *WriteHash*.

With the core file input and output functions explained in depth, the three database module

interface functions can be detailed as well. The four file I/O functions handle mostly just that: interaction with the file system; whereas these functions handle the higher level logic of the hash table, including collision resolution, locking protocol, and coordination between the hash table file and the page contents file. *Contains* and *Write* both return boolean values indicating whether the page is present in the database, whereas *Read* returns a status value.

Contains (*hash IN*, *page IN*) :

As previously mentioned, *Contains* is used to check the database for the presence of a specified hash and page value pairing; if a specified pair is present in the database, then it can be considered standard, and safely sent as only a hash value during a migration instead of sending the full page contents. *Contains* first creates all three database files, if they are not already present. Then, it attempts to open the lock file read only. Once successful, the hash table index is taken from the lower order bits of the given page hash, then the hash table file is opened, and traversed starting at that index until the page is found or determined to be not present in the database.

This is an iterative process, which is repeated until the probe limit is reached. The probe limit is implemented to ensure that if a large contiguous region of the hash table is full, the kernel does not spend too much time searching for an available hash table entry. It is configurable, since the ideal value would vary depending on the size of the hash table. The default value is 15. If the probe limit is reached, therefore, that new page cannot be added to the database. On each iteration, *ReadHash* is called, and the returned page hash is compared with the hash value that *Contains* was called with. If they are equal, there is a potential page match, so the appropriate page contents are retrieved by calling *ReadPage* with the page file index returned by *ReadHash*. The atomicity of these operations is not a concern, since entries cannot be removed from the database; it is guaranteed that the page in the contents file corresponds to the hash in the hash file. Additionally, the page contents are written before the hash table

entry, so it is also guaranteed that the page contents exist at the page index given in the hash table. Once the page is read by *ReadPage*, A full memory comparison is performed between it and the page given to *Contains*. The result of this comparison is returned by *Contains*: the page is standard if and only if their contents are identical.

Additionally, if the hash returned by *ReadHash* is equal to zero (*INVALID*), then it is also certain that the page is not present in the database, and a value of false is returned. The *Contains* and *Write* functions share the same probe limit for hash collisions, and entries cannot be removed from the hash table. Therefore, if the page had been previously added by the *Write* function, it would have written it into an entry with a lower index, or into the blank entry encountered by *Contains*.

If the hash value returned by *ReadHash* is not *INVALID* or the given page hash value, and the probe limit has not yet been reached, then the hash may still be present in an entry with a higher index; the index is then incremented and another iteration is begun. If the probe limit is reached, the page is not present in the database, and *Contains* returns false. In any case, before any value is returned, the hash table file is closed.

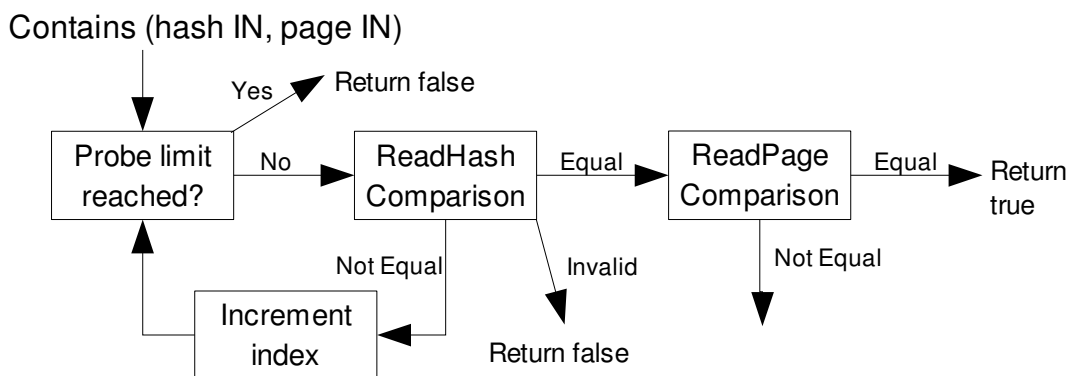


Figure 7: Contains iteratively searches for an entry in the hash table with a hash value equal to the hash given as input until a match or a value of zero is found, or the probe limit is exceeded. If there is a match, the page contents given to the function are compared with the page contents in the database and the result of that comparison is returned.

Write (hash IN, page IN) :

The operation of *Write*, used to add new standard pages to the database, is much the same as *Contains*, with a few key differences; specifically, the actions taken based upon the value returned by *ReadHash*, and the use of the lock file. *Write* attempts to open the lock file in exclusive mode and then opens the hash table file for writing. This may result in heavy lock contention, but if the exclusive lock is not acquired before the hash table file is read, there is a potential race condition where two hosts may attempt to write to the same hash table entry. Additionally, the lock is only ever acquired exclusively for *Write*, it is never acquired for a reading operation (by *Contains* or *Read*). Furthermore, read operations from the database should be much more common than write operations; each standard page is only ever written to the database once across all hosts, but it can be read many times, and likely at least once by each host.

If an empty entry is found in the hash table file (*i.e.*, the hash value is equal to zero), then the new page can be added to the database there. *WritePage* is called to add the page to the page file, and get the appropriate page file index value to be written to the hash table. Once this is done, *WriteHash* is called, and the page hash is added to the hash table along with the index of the associated page in the page file. If these writes complete successfully, *Write* returns true after closing the files to indicate that the page was successfully added to the database, and can now be considered standard.

If, on the other hand, an entry is found in the hash table file with a matching hash value, then either: the page is already present in the database, and can be marked standard, or a different page is already present in the database with the same hash value, implying a hash collision. To determine which of these is the case, *ReadPage* is called to retrieve the page contents from the database, which are compared to the page contents requested to be written. The result of the comparison is returned after the database files are closed. As with *Contains*, if the probe limit is reached, the page cannot be added to the database and is not already present, and a result of false is returned.

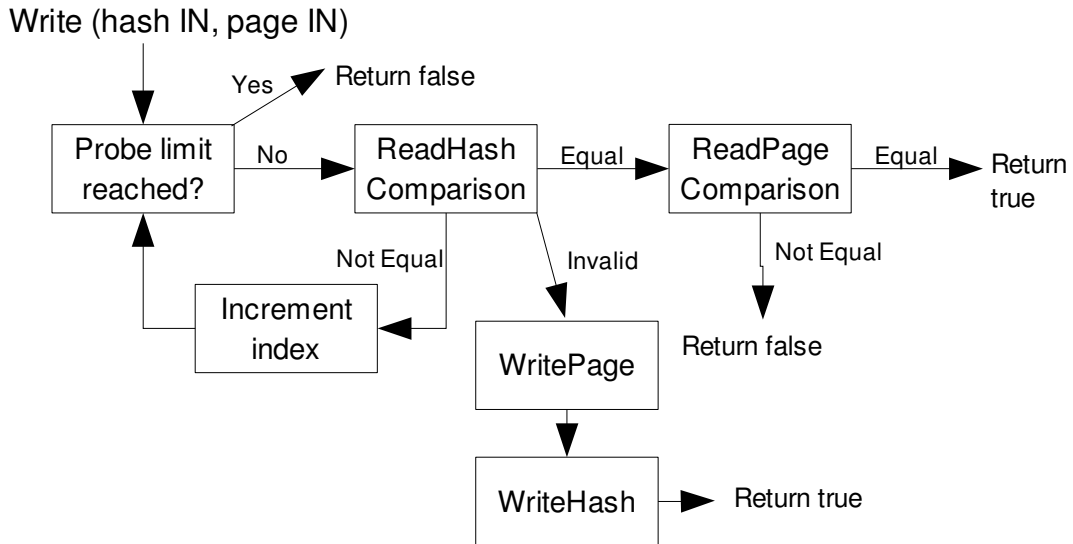


Figure 8: Write functions in nearly the same way as Contains. The primary change in behavior is when a hash table entry with a hash value of zero is found: this indicates an empty slot in the table that the new standard page can be placed in. The page contents are written to the page contents file, and the new hash value is added to the hash table with a page index pointing to the just-added page.

Read (hash IN, page OUT) :

Read is nearly identical to Contains with one exception: instead of taking a page as input, it gives a page as output. If a hash match is found in the database, instead of doing a full page comparison between the page it was given and the page in the database, it simply returns the page read from the database at that page index with a call to ReadPage.

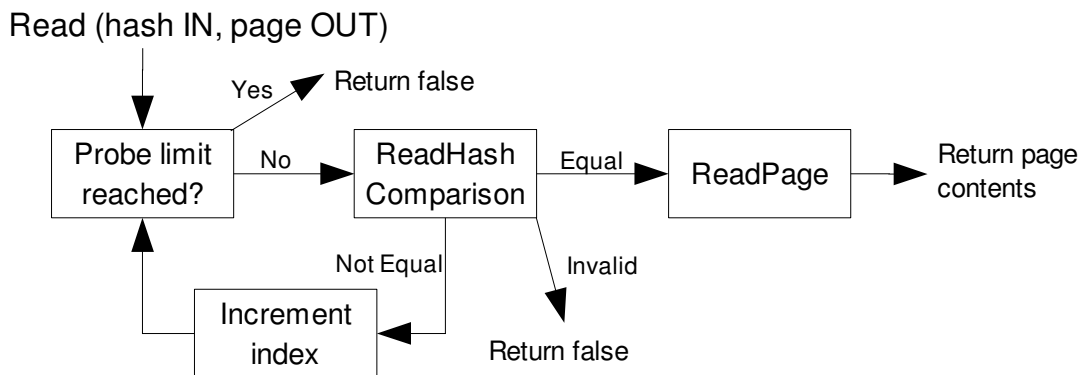


Figure 9: Read is simply used to retrieve database contents, so when a hash match is found, there is no need for a page contents comparison. It simply returns the contents of that database entry.

3.3 VMkernel and page sharing changes

With a database and matching interface available to determine which pages are standard, the next essential element is to find standard pages in guest memory. This is implemented with a set of changes to the portion of the VMkernel that handles page sharing. Potential standard pages are found by scanning the page sharing tables and looking for shared pages which are not already marked as standard. Each potentially standard page is checked against the database, and if present, is set as a standard page and will remain standard as long as it is present in the page sharing tables.

The key element to this process is the ability to tag a page to indicate that it is standard. This is enabled by the existing structure of the page sharing tables: each entry, in addition to the 64-bit hash key, the MPN, and a reference count, contains a tag indicating information about the type of the page. This tag can indicate that it is a regular shared page (*PSHARE_REGULAR*), or a hint (*PSHARE_HINT*), or a few other types. A *PSHARE_STANDARD* tag has been added as well, and the tag value is set to this for any page that's been checked against the database and is definitely a standard page. Utility operations were added to facilitate the use of this tag: *IsStandardPage* and *SetStandardPage*, which perform those standard tag operations on a given MPN.

There are two other utility functions necessary for the overall operation of this procedure: two wrappers for the database module functions, *DBWrite* and *DBContains*. These functions are necessary to handle interaction with the database module, which may or may not be loaded, and to provide the module functions with a copy of the contents of the page in question.

DBWrite (*hash IN, MPN IN*)

DBContains (*hash IN, MPN IN*) :

These functions are identical except for the database module function that they call and refer to.

First, the function pointer exported by the database module for that function is checked. If it is equal to NULL, the module is not loaded, and therefore standard page checking is disabled, so a result of false is returned. If not, the module has been loaded, and the page contents for that MPN are copied into a temporary variable. The appropriate module function (*Contains* or *Write*) is then called with the page hash and the temporary variable holding the page contents. Once the result of the database function call is returned, the temporary page is freed and the result is returned.

These functions provide the necessary interface to the page sharing table entries and the standard page database. A separate daemon thread is created and started on system bootup when page sharing is initiated to perform the scanning of the page sharing tables and call the database functions as necessary. There are three stages in this process, which are repeated periodically. First, a *DBScan* function traverses a preset number of page sharing table entries, adding each potential standard page (those tagged *PSHARE_REGULAR*) to a buffer. Second, a *DBQuery* function performs the database queries on all of the buffer entries, refilling the buffer with those entries that were either found in or successfully added to the database. Finally, a *DBSetStandard* function sets the *PSHARE_STANDARD* page on all of those pages in the page sharing table which *Query* determined to be standard.

3.3.1: Standard Page Daemon Thread

A background daemon that performs the above operations is started on boot at the same time that the page sharing is initialized. An initialization function creates the new system thread, creates a heap for it large enough to allocate memory for the aforementioned buffer, and adds it to the scheduler with a starting function (*DBLoop*) that loops and makes calls to the *DBScan*, *DBQuery*, and *DBSetStandard* functions.

DBLoop:

This function performs general control over the standard page process. It is a nested loop; the outer loop is an infinite while loop, while the inner loop calls the three other functions sequentially in a preset number of iterations (*PAGEDB_BATCHES*), and then sleeps for a preset amount of time (*PAGEDB_SLEEP_TIME*). Both of these things are done in an effort to prevent constant disk accesses to the shared storage, while still attempting to process a large number of potential standard pages. Additionally, before the *DBQuery* function is called, since the thread is non-preemptible, *DBLoop* voluntarily yields to the scheduler before the disk accesses are performed to avoid monopolizing CPU time.

The page sharing tables are arranged as an array of linked lists containing entries for each shared page. Only a small portion of the page sharing tables are scanned on each iteration of *DBLoop*, so it keeps track of the current index into the array between iterations. This value is passed to *DBScan* as the index to start scanning from, ensuring that over time, all linked lists in the array will be scanned with regular frequency.

The aforementioned buffer is part of a global page sharing structure, which is accessible by all of the page database functions in the kernel. Each entry in the buffer consists of a hash key, the MPN of the shared page, and a boolean value indicating whether the page should be added to the database, or just checked for presence in the database. The size of the buffer is set to a constant value, *MAX_ENTRIES*, which also represents the maximum number of page table entries that will be scanned on each iteration.

DBLoop pseudocode:

```
WHILE TRUE

  FOR batch = 0 to PAGEDB_BATCHES
    Acquire page sharing lock
    FOR chain = 0 to MAX_CHAINS
      WHILE end of chain has not been reached
        IF current page is not zero or standard
          IF page reference count > write threshold
            Set MPN to be added to database
          ENDIF
          Add page MPN to buffer
          Increment MPN reference count
        ENDIF
        IF buffer is full, exit FOR loop
          Set current page to the next page in the chain
        ENDWHILE
      ENDFOR
      Release page sharing lock
      Yield to scheduler
      FOR all MPNs in buffer
        IF MPN is set to be added
          IF DBwrite(MPN) == TRUE
            Reinsert MPN into buffer
          ENDIF
        ELSE
          IF DBContains(MPN) == TRUE
            Reinsert MPN into buffer
          ENDIF
        ENDIF
      ENDFOR
      Acquire page sharing lock
      FOR all MPNs in buffer
        Lookup MPN in page sharing tables
        IF reference count == 1
          Remove page from tables
        ELSE
          Set page as standard
          Decrement MPN reference count
        ENDIF
      ENDFOR
      Release page sharing lock
    ENDFOR
    Sleep for PAGEDB_SLEEP_TIME
  ENDWHILE
```

DBScan (currentIndex IN/OUT) :

The general flow of *DBScan* is a nested loop; the outer loop iterates over the array in the page sharing table, while the inner loop traverses the linked list that each entry in the array refers to. The outer loop is run a preset maximum number of times, *MAX_CHAINS*. The inner loop traverses the linked list of page sharing entries until the end of the list is reached. If at any point, the buffer becomes completely full, execution jumps outside both loops. Before the loops are entered, however, the lock for the page sharing tables is acquired, to ensure that the page sharing tables aren't modified while *DBScan* is reading from them. Since this thread is sleeping significantly more often than it is running, the contention for the page sharing lock shouldn't be problematic. Before the function returns, the lock is released; it cannot be held while the blocking disk access is performed during the next function *DBQuery*.

The inner loop examines each entry in the linked list. If a page that an entry refers to is shared and not already marked as standard or zero, an entry is added to the buffer for it with the page's MPN and key. If the reference count is above the preset threshold, the entry is marked to be added to the database. The default reference count threshold is 2, so all shared pages will meet this requirement, but the value is configurable. At this point, because the page sharing lock is released while the database queries are performed, a measure must be taken to ensure consistency. It is possible, in the period of time after a page is added to the buffer and before it is later marked standard in the page sharing tables, while the lock isn't held, that all references to that particular page may be dropped, and that MPN may at that point have been recycled and contain entirely different page contents. To prevent this from happening, the reference count for the page is incremented by one to prevent the MPN from being recycled while the lock isn't held by the standard page daemon.

After each completion of the inner loop, the current array index is incremented, and the next linked list in the array is traversed. However, if the buffer is filled before a linked list is completely scanned, the array index is not incremented before the function returns. This prevents the pages towards

the end of a list from being repeatedly skipped over on each pass over the page sharing tables. There is one exception to this: it is possible that an extremely long linked list could exist with more entries than the buffer being filled. If this was the case, and the loops were terminated before the end of the linked list was reached, the function may end up scanning that list repeatedly every time it is called. In light of this, if the buffer is filled and the array index hasn't changed since the function was called (indicating that the buffer filled before completely traversing one linked list), the index is incremented. However, since the page sharing table happens to be configured to be at least as large as the number of potential entries, and the hash function used has good uniformity, this special case is very unlikely. It is, nevertheless, important to handle the possibility that it might occur. Finally, the *currentIndex* variable passed originally to *DBScan* is updated to reflect the new array index value, and the number of entries that were added to the buffer is returned.

DBQuery (numPages IN) :

This function is given the number of entries that were added to the buffer by *DBScan*, and iterates over all of the entries that were added to the buffer. For each entry, either *DBWrite* or *DBContains* is called, depending on the boolean value set by *DBScan*. If the database function returns true, the page can be marked as standard. The buffer is reused here; while it is traversed, a second index value is maintained, initialized to zero. Each page determined to be standard is added to the buffer at that index, which is subsequently incremented. After all database queries have been made, the value of this second index is returned, indicating the number of standard page entries present in the buffer.

DBSetStandard (numPages IN) :

DBSetStandard is given the number of standard pages in the buffer, acquires the page sharing lock, and calls *SetStandardPage* on each of the pages. Before a page is set as standard, however, its reference count in the page sharing tables is checked. If the reference count is equal to one, that

indicates that the only reference is the false reference added by *DBScan*, and all true references to the page have been dropped while the page sharing lock was unheld. In this case, the page's entry is removed from the page sharing table and the MPN is freed. Otherwise, the page is marked standard and the reference count is decremented. After the entire buffer has been processed, the page sharing lock is released and *DBSetStandard* returns.

3.4 Migration Changes

As with the modifications made to page sharing, the central aspect of the modifications done to the migration system is the addition of the standard page type. The rest of the changes are focused around handling the new page type properly in different situations. During the precopy stage of migration, the source host sends the page hash with no accompanying page contents for any pages that are marked as standard. The destination host, when it receives a standard page hash from the source, simply maps that new page copy-on-write to the already-shared standard page, if it is present. If it is not, the destination host sends a request to the source for the full page contents.

3.4.1 Migration Source

The most important task on the source side of a migration related to VMOSH is retrieving the relevant information about a page to be sent to the destination host. This is accomplished by *GetPageInfo*, which is given a GPPN and the associated VM, and returns a page information structure containing metadata about the page, as well as the page contents in some cases. For standard pages, the important piece of metadata is the page type. Where *GetPageInfo* normally determines the page type, a new call to *IsStandardPage* (the same function as above in the page sharing modifications) has been added. If it returns true for the page in question, the page type field in the metadata structure is set to standard.

For a normal non-zero page, the page data would then be copied and returned. For a standard page, however, this may or may not be the case. If a standard page is being sent for the first time, it is not necessary to copy the page contents. However, if *GetPageInfo* has been called to send a standard page that has been requested by the destination, the page contents must be copied. This uncertainty is handled by a new boolean input parameter to *GetPageInfo*: *sendStandard*. If *sendStandard* is true, it is because the call to *GetPageInfo* originated from a request from the destination, and the page data is copied. If it is false, then it is safe not to copy the page contents. *GetPageInfo* is called from two places: the precopy routine, and the function that handles page requests from the destination. The precopy routine sets *sendStandard* to false, while the page request handler sets it to true. In this way, it is assured that the page contents are only copied and sent when specifically requested by the destination host. Any page that *GetPageInfo* indicates is standard is then handled by the precopy routine in the exact same way that zero pages are treated: no actual page contents are sent.

3.4.2 Migration Destination

The changes to the code for the migration destination are similarly localized, to a function *AddPage* which handles each page type received as is necessary. It is given the GPPN and associated VM for each page received, as well as the page metadata and page contents given by *GetPageInfo* on the source host. For a standard page, *AddPage* first performs a lookup in the page sharing hash tables for the page hash sent by the source. If a matching MPN is found, it is passed to *IsStandardPage* to determine if the incoming page can be safely shared with the pre-existing page. If it is standard, it then attempts to share it and if successful, the migration of that page is complete.

If any of those conditions are not met, the reason is either: no matching page hash was found in the page sharing tables, a matching page hash was found but that page was not marked as standard, or a matching standard page was found but sharing the page failed for some reason. In all of these cases,

the required action is to request the page contents from the source. This is done by calling a *PageFault* function which handles network messaging to and from the source host, and returns the page contents and metadata. Once this completes, the page hash is again looked up in the page sharing tables. This time, however, if a match is found, the page contents are compared. If they are equal, the incoming page is shared, and the shared page is marked standard. If they are not equal, or there was no matching page hash found, then the incoming page is added but not shared.

4. Evaluation

The feasibility of VMOSH hinges entirely on whether or not the benefits gained are greater than the overhead incurred. To do so, performance testing was necessary. In order to make it worthwhile, four separate steps were made, and are laid out in the section. First, an analysis of potential performance effects on different aspects of the system in different situations determined what should be measured. Then a set of benchmarks were designed to test and exhibit the predicted behaviors. The full set of tests was run and the desired data collected and combined into meaningful numbers. The last step was a post-analysis of the benchmark results in light of the predictions made.

4.1 Factors affecting performance

There are two distinct operations at work in VMOSH that are largely independent of each other in terms of the immediate effect on the overall performance of the system. The first is the background daemon thread that scans the page sharing tables searching for standard pages and performing database accesses. The second is the actual migration, which of course only comes into play when some policy dictates that a virtual machine must be migrated to another host. Because of this separation, the theoretical performance impact of each can be examined separately; the migration modifications should not affect the impact of the standard page scanning at all except during migrations.

4.1.1 Costs of standard page discovery

Though the results of the standard page scanning can improve the performance of VM migrations, the scanning process itself offers no potential improvement to normal operation of the server. Therefore, the primary consideration for this process is the performance cost, rather than any potential benefit. The first of two potential concerns is the CPU time spent running the daemon thread. Since the

thread sleeps for extended periods of time (on the order of seconds at a time), and voluntarily yields to the scheduler frequently, the direct CPU usage of the thread should be minimal. Furthermore, the scanning parameters are all configurable, so they can be modified in conjunction with kernel resource controls in order to manage the threads resource consumption. The second and more significant performance penalty will be the quantity of disk I/O to the shared storage with the batched database reads and writes.

4.1.2 Costs and benefits to migration

Optimistically sending only page hashes for standard pages could have a widely varying effect on the bandwidth usage and net latency of migrations depending on the workload running in the migrating VM and the other virtual machines running on both the source and destination machine. If the VM has few pages in common with other VMs on the source host, not many pages will be marked standard and little if any speedup could be expected. Worse, if the VM has many pages in common with other source host VMs but not with destination host VMs, there is the potential for a large percentage of standard pages sent not being present on the destination. This would result in the destination requesting many pages in full from the source host, implying an unnecessary round trip communication for each of those pages. If there were enough such pages, the end result could be a significant overall performance degradation. The first case might occur for data intensive workloads such as database systems where relatively few duplicate pages would be expected across VMs. The second, worse scenario might occur if migrating a Linux VM from a host full of other Linux VMs to a host running all Windows VMs.

However, those are near worst-case scenarios. A more homogeneous virtual machine environment has the potential to benefit from significant improvements in migration performance and efficiency. If the migrating VM has many pages in common with VMs on both the source and destination hosts, as could be expected if the VMs are running similar workloads, the results should be very positive.

The most immediate benefit to the migration process is the first order benefit of less overall network transfer and elapsed time associated with the precopy stage of migration. Additionally, less time spent in each precopy stage of migration should result in fewer dirty pages for the next stage of precopy to transfer as a second order benefit. This has the potential to even reduce the number of precopy iterations necessary and might result in shorter VM downtime during the final transfer period of the migration.

4.2 Benchmark design

The tests to determine the performance of VMOSH should take into account the potential effects predicted above. There are three different decisions to be made to accomplish this: what measurements should be taken, what workloads should be run, and what testing environments should be chosen to draw performance comparisons between.

Since the overarching goal is to speed up the migration of VMs between hosts, the most important measurement to make is the total elapsed time of the migrations. Its also useful to know how this elapsed time breaks down, in particular the effect on the precopy stage, and whether or not any of the other stages of the migration sped up or slowed down; downtime, for instance, could be potentially affected as discussed above. In addition to potentially improving the elapsed time of a migration, using standard pages might also result in a decrease in network bandwidth usage, so that should be measured as well.

There are other aspects of migration statistics that are introduced by the use of a standard page database. It should be useful to see the breakdown of page types transferred, in order to see to how many pages are standard in different workloads. This would allow for a comparison between the migration time improvements and the percentage of standard pages in the VM to see how closely they match. Ideally, the reduction in precopy time might be very close to the percentage of standard pages present

and marked in the migrating VM and on the destination host. Of course, the VM will have some standard pages that the destination host doesn't, and those will weigh against any reduction in elapsed time, so the number of standard pages that the destination requests from the source should be measured and factored into that comparison. Finally, the performance impact of the standard page scanning process should be measured. Since the database interface is a loadable kernel module, the effect on migration time can be easily measured by running tests with the module loaded, then running them again immediately afterwards with the module unloaded. In that time the set of standard pages won't have changed significantly, so a valid comparison can be made.

Workloads have been chosen to attempt to exhibit the predicted behaviors. A best case scenario workload, a worst case scenario workload, and a more realistic workloads to look at precopy speedup as well as any potential second order benefit in VM downtime resulting from fewer dirty pages.

Best Case: Identical Ubuntu 8.04 VMs configured with 512MB of RAM (booted up but not logged in) should minimize data unique to each VM and maximize potentially shareable read-only executable pages.

Worst Case: Memtest86 should do well to show worse case results; constantly changing data throughout all of the VM's memory should prevent many standard pages from being successfully transferred, if any.

VDI Workload: Desktop VMs like those running in thin-client environments like VMware's Virtual Desktop Infrastructure (VDI) are a realistic workload that should aptly demonstrate the benefits of the standard page database; there should be a lot of executable read-only pages that are sharable between VMs if clients are running the same desktop software, with relatively few unique writeable pages. Precopy speedup should be high. This workload is approximated with Windows XP Pro SP2 VMs, configured with

256MB of RAM, running Internet Explorer, Microsoft Word, Microsoft Excel, and Outlook Express; a likely typical office workload.

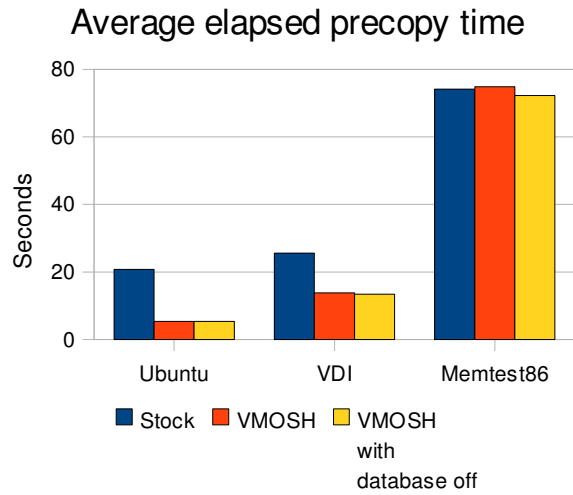
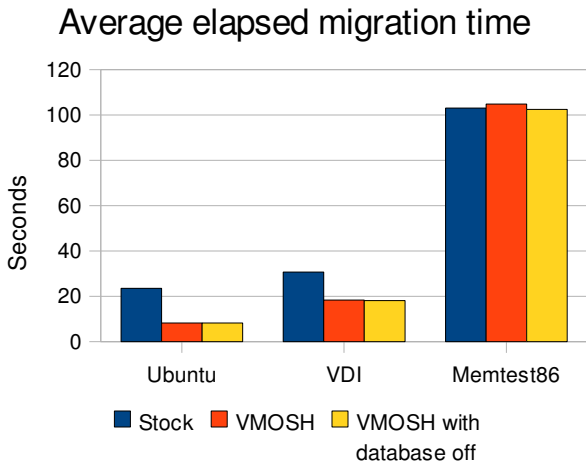
All tests are run on a pair of Dell PowerEdge 2850 servers with dual-core Intel Xeon 2.8GHz processors and 4GB of RAM. Each server runs two of the VMs, while a fifth VM is migrated back and forth between them. Measurements are taken from each of the migrations, and each set of migrations is run in three different environments: one set running on an unchanged kernel with no standard page modifications, one set on the modified kernel accessing the standard page database, and then one more set on the same kernel after turning off the standard page database kernel module. Measurements discussed above were taken from migration logs during ten repeated migrations of each VM between two hosts running each version of the kernel.

4.3 Benchmark results

We present and discuss the experimental results in this subsection. Each chart, except where specified, shows individual measurements of each VM.

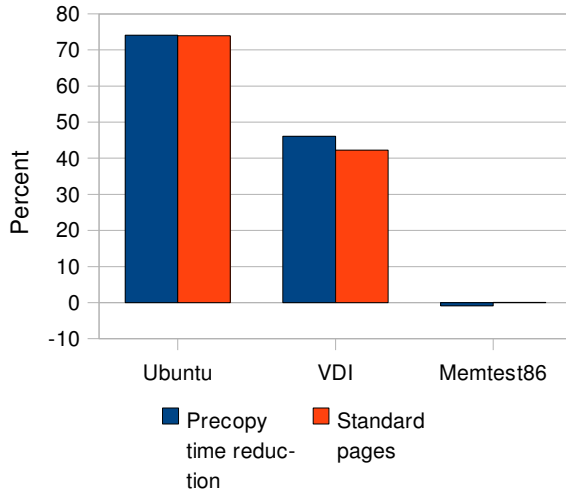
- “Stock” measurements represent those taken on an unmodified VMware ESX Server system.
- “VMOSH” measurements are those taken with the standard page optimization running.
- “VMOSH with database off” represents measurements taken on the modified kernel, but with the page database module turned off to test the overhead of disk accesses.

These results confirm the initial hypothesis that sending page hashes instead of full pages where possible significantly improves migration performance.

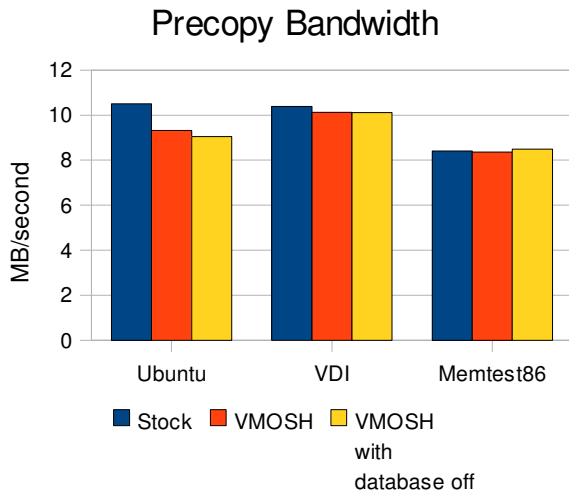
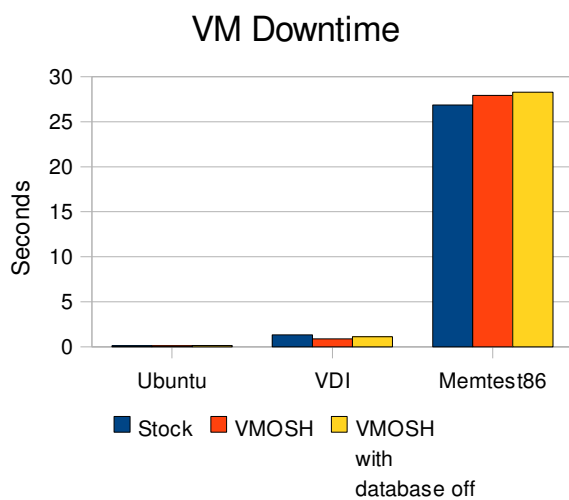


As expected, the best case Ubuntu VM had the most significant reduction in migration time, with an improvement of nearly a factor of 3. Also as expected, the Memtest86 VM performed the worst under VMOSH, since the memory contents are unpredictable and constantly changing, standard pages were not a useful concept, and the overhead did slow down the migration by a slight amount. The VDI workload was also impressive, with a 40% reduction in net migration time, showing that there were plenty of identical pages across the hosts.

The average precopy time shows a similar story, with the Ubuntu VM again showing the biggest reduction at nearly 4 times faster, Memtest86 performance degraded, and VDI showing a 46% reduction. Of course, it is not surprising to see how these three workloads performed compared to each other; the Ubuntu VM was expected to have significantly more standard pages than any of the others. It is more interesting to examine how closely the speedup is correlated with the percentage of standard pages in the VM.



The performance improvement for the Ubuntu and VDI VMs actually exceeded the percentage of standard pages by a slight amount, which cannot be accounted for by the first order precopy benefit. It is likely, therefore, especially in the VDI case, that this discrepancy was caused by the predicted second order benefit: the less time each precopy stage takes, the fewer pages will be dirtied and needed to be sent during the next stage. This second order benefit, if present, would also appear as predicted in the VM downtime and bandwidth measurements:



The Ubuntu VM showed no difference in downtime. This is most likely because the VMs were so small, similar, and easily copied that the data transferred during the downtime was insignificant compared to the overhead of the operation. At 0.13 seconds, a reduction in downtime shouldn't be necessary. The VDI workload, on the other hand, saw a 33% reduction in downtime, from 1.35 seconds to 0.9 seconds, further reinforcing the idea of the second order benefit. Similarly, modest improvements were seen to the precopy bandwidths on both the Ubuntu VM and VDI workloads. As expected, the Memtest86 VM performed poorly; exhibiting a slight but insignificant slowdown.

5. Related Work

This thesis does not present the first system to make use of page hashes to optimize virtual machine migration. There is a body of existing work related to shared pages, virtual machine migration, and accelerating the migrations with various methods, including page hashes. The published work that is closely related to the concepts presented in this thesis is discussed below.

Disco: Running Commodity Operating Systems on Scalable Multiprocessors

Disco[2] introduced the idea of transparent page sharing that is central to the feasibility of the standard page database, which basically amounts to transparent page sharing between hosts and not simply between VMs. In contrast to the periodic scanning used in VMware ESX Server's page sharing, Disco's sharing mechanism was based around disk accesses. If a VM attempted to read a block from disk that was already loaded into memory, the hypervisor would simply share it in memory and mark that disk block copy-on-write instead of making an unnecessary disk read. This had the advantage that many disk reads were very fast, and that many files could be shared between VMs, saving disk space. However, it was only able to share data that appeared exactly the same in memory as on disk. Many executables, ideal candidates for sharing since they are generally read-only, are loaded into memory differently than the on-disk contents, for instance. Furthermore, Disco was unable to perform this sharing mechanism with unmodified guest operating systems.

Memory Resource Management in VMware ESX Server

Memory Resource Management in VMware ESX Server[8] introduced the page sharing scheme that is used for the standard page database. Unlike the Disco implementation, it is able to share pages transparently between unmodified guest operating systems. Scanning and hashing page contents

enabled this improvement, and allow for the sharing of all identical pages, regardless of how they originated. Since it isn't disk-based like Disco, it doesn't offer the improvement in disk access speed or the disk space savings. The hashes gathered to implement it, however, are the key to making VMOSH function efficiently.

Fast Transparent Migration for Virtual Machines

VMware ESX Server's VMotion feature[6] was, as far as we know, the first system able to transparently migrate entire VMs with unmodified operating system and application code. This is the motivation behind the use of page sharing hashes to transfer VMs more efficiently between hosts. It utilizes the precopy and checkpoint/transfer stages already discussed. However, in terms of reducing the number of full pages of memory sent, it only optimized for zero pages, and not any other shared pages.

Optimizing the Migration of Virtual Computers

The Collective[7] is a system somewhat similar to VMotion for transferring virtual machines transparently. However, unlike VMotion, it is not intended to be used with server machines on a local area network with shared storage. Instead, it is designed to transfer entire virtual machine states, including disk contents, over long distances on relatively slow connections. To do this in a reasonable amount of time, it utilizes a variety of techniques to reduce the amount of data that must be sent between the hosts. One of them is very similar to the standard page hash method used here; sending only a hash instead of full page contents. However, it does not have pre-generated hash values available for the pages, nor do the hosts involved have high speed access to a shared database of pages to verify the hashes. Instead, 160-bit SHA-1, a cryptographic hash, is sent for each page. If the destination can find a page with the same hash, it uses that page. Otherwise, it sends a request back to the source for the full contents. *The Collective* would not work well in a server environment; SHA-1 is computationally intensive, and

computing a full hash of every page sent during a migration would likely negate any performance benefits from reduced network traffic. Computing the hashes in the background would help mitigate this, but there is still a possibility, however minute, of a harmful hash collision which would not be acceptable in an enterprise environment. Utilizing the 64-bit hashes already generated for page sharing and shared storage to store a database of pages is more appropriate in a local area network server context.

6. Summary

Overall, the performance outcome of VMOSH was quite good, an average reduction in elapsed time of 40% on a fairly realistic workload is significant, and is also fairly close to the performance improvement that one might expect simply based on the similar gains in memory savings provided by content-based page sharing. The worst case scenario VM showed slowdown, but not as much as might have been expected. In retrospect, it likely wasn't really the worst case scenario. Since Memtest86 has very few shared pages, there weren't many pages in the page sharing tables for VMOSH to check against the database, so disk access overhead was minimal. As a result of this, there were also very few standard pages, so most of the pages in the VM transferred no differently than normal.

A worse scenario comes to light when the number of standard pages that the destination VM requested from the source is examined. There were only a few migrations in which these pages exceeded even one tenth of one percent of the total pages migrated; clearly, it did not have much of an impact. A pathological case for these instances would be a source host with a high rate of sharing, and many standard pages, migrated to a destination with minimal standard pages in common with the migrating VM. If, for example, three of the Ubuntu VMs had been running on one of the hosts, and one was migrated to the other host, running a set of Memtest86 VMs, a majority of the standard pages (comprising 70% of the VM) would be requested in full from the source host. A more realistic example would be two hosts running VMs with different workloads on different operating systems, and migrating between them anyway. That is likely the real-life worst case scenario, and it would be a valuable test to perform and weigh against the improvements seen in the VDI workload to determine if VMOSH would be beneficial in a production environment.

Since page sharing and migration were already implemented, the initial implementation of VMOSH did not require excessive changes. All of the changes together consist of approximately one

thousand lines of code, which is fairly small compared to the migration and page sharing systems themselves. However, though this first effort is fairly compact, there are a number of improvements to VMOSH that could be made in the future.

Database Improvements

The database designed for VMOSH is quite simple; while fully functional, there are some changes and additions that could be made to make it faster, more efficient, and more capable. Two features that are lacking in the current implementation are the ability to resize the database and delete or replace entries. These both limit the scalability of the database and the hash table's ability to handle high load factors. Resizing the database could be accomplished by implementing a policy to recreate the hash table in a significantly larger file whenever the load factor reaches some preset threshold. Allowing entries to be deleted or replaced is a more difficult task, because of the necessity of having all hosts agree on the standard pages. Changes made to the database would have to be communicated back to every host affected by that change. This could be done by keeping track of which hosts have each standard page, and having them remove the standard page tag if its database entry is deleted or replaced. Alternatively, the database could handle modifications by periodically creating a new epoch and making sure that all hosts updated their standard page information to match, and sending only hashes for standard pages if both hosts are operating in the same epoch.

There are also some performance improvements that could be made to the database, primarily by reducing the quantity of costly disk accesses to shared storage. This can be done by modifying the way that the hash table is utilized. Reads from the hash table can be reduced by retrieving contiguous groups of hash table entries instead of reading them one at a time; because the hash table uses open addressing, the sequential locality of the entries means that if one entry is accessed, the next is likely to be accessed as well. Additionally, a sort of in-memory precheck could be implemented for reads to determine whether it is likely that a page is in the database without accessing the disk. A local cache of

the hash table file that is periodically updated, or a bloom filter representing the set of hashes in the database are both options. If a hash is present in the hash table cache or the bloom filter, then a page read from the database should be performed.

Tuning

There are many configurable numerical settings that determine the behavior of VMOSH as a whole. While all of these values default to reasonable values, little research has gone into determining optimal values for any of them; the time and resources required to do so were not within the scope of this project. However, in the future it would be useful to run further experiments to determine if there are better values than the defaults. The configurable values include database size, page sharing table scanning rate and buffer size (broken down into the maximum number of table chains and frames, the number of batches run between sleeps, and the amount of time to sleep for), and the reference count thresholds for database reads and writes. Some of them are likely to vary depending on the machine and the workload, and some are dependent upon each other, so testing for this purpose would have to be rather extensive.

Other Potential Applications

Much like the page hashes stored in the page sharing tables were useful for other purposes as has been shown, a central database of common pages could be quite useful in contexts other than virtual machine migrations. For example, the standard page database could be used when swapping pages to disk; if a page being swapped happens to be a standard page, the host could safely store only the hash and then recover the full page contents when necessary from the database instead of making an unnecessary copy on disk. On a more general level, the same concept could be used for a more universal standard page database. Instead of only enforcing consistency within a local area network, a

system might be devised to accelerate transfers across longer distances if a VM needs to be migrated to another server facility. The same database could be used to distribute pre-made virtual machines; an initial image with hashes instead of standard pages in the VM could be provided, and any standard page that the user does not have could be requested. In essence, the use of the standard page database is a hash-based compression scheme; viewed in that light, it wouldn't be surprising if there were many more potential uses.

7. Bibliography

1. Keith Adams, Ole Agesen. "A comparison of software and hardware techniques for x86 virtualization," In *Proceedings of the Twelfth International Conference on Architectural Support for Programming Languages and Operating Systems*, November 2006.
2. Edouard Bugnion, Scott Devine, Mendel Rosenblum. "Disco: Running Commodity Operating Systems on Scalable Multiprocessors," *ACM Transactions on Computer Systems*, 15(4), November 1997.
3. Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, Andrew Warfield. "Live migration of virtual machines," In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, May 2005.
4. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms*. Cambridge: MIT Press, 2001.
5. Bob Jenkins. "Algorithm Alley," *Dr. Dobbs Journal*, September 1997. Source code available from <http://burtleburtle.net/bob/hash/>
6. Michael Nelson, Beng-Hong Lim, Greg Hutchins. "Fast Transparent Migration for Virtual Machines," In *Proceedings of the USENIX Annual Technical Conference 2005 on USENIX Annual Technical Conference*, April 2005.
7. Constantine P. Sapuntzakis et al. "Optimizing the Migration of Virtual Computers", In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December 2002.
8. Carl A. Waldspurger. "Memory resource management in VMware ESX Server," In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, December 2002.
9. Carl A. Waldspurger, 2004. Content-based, transparent sharing of memory units. U.S. Patent 6,789,156, filed July 25, 2001, and issued September 7, 2004.
10. VMware, Inc. VMware Infrastructure 3 Documentation, <http://www.vmware.com/support/pubs>, 2007.