University of Alberta

HIGH PERFORMANCE LIVE MIGRATION OVER LOW-BANDWIDTH, HIGH-DELAY NETWORK WITH LOSS PREVENTION

by

Zhu Pang

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Fall 2010
Edmonton, Alberta

Examining Committee

Mike MacGregor, Computing Science

Bruce Cockburn, Electrical & Computer Engineering

Paul Lu, Computing Science

Abstract

Virtualization technology has attracted considerable interest. It allows several virtual machines to run concurrently inside a physical host, which brings multiple advantages. One of the most useful features is called live migration, during which a virtual machine can be migrated over network with minimal disruption. So far, most existing migration algorithms are focused on transferring the runtime state over high-speed, low-delay network. They all require shared storage for file systems. However, this sharing sometimes becomes impossible because of performance loss. Thus, the whole system needs to be transferred during migration.

In this thesis, we introduce a Virtual Machine Management System which contains a block-level solution. Combined with pre-copying the runtime state, we can migrate an entire virtual machine over low-bandwidth, high-delay network with minimum service downtime. We show that this is sufficient even for interactive workloads. We also provide snapshots and full backup for the virtual machine.

Keywords Virtual machine, live migration, low-bandwidth network, snapshot, loss prevention

Table of Contents

List of Symbols

VMMS	Virtual Machine Management System
VM	Virtual Machine
SBD	Synchronized Block Device
WWS	Writable Working Set
VMM	Virtual Machine Monitor
VNODE	Virtual Machine Management System Node
CMS	Centra Management Service
OSPF	Open Shortest Path First

Chapter 1

Introduction

Recently, virtualization is gaining more and more interest in both industry and academia, mainly because of its capabilities of doing load balancing, full system isolation, fault tolerance, and server consolidation. With virtualization, it is possible to run several OS instances concurrently, sharing the same physical host. It provides efficient and secure computing resource containers for cluster environments, data centres, and many other distributed infrastructures ([4] [16], [12]).

Live migration, a key benefit brought by virtualization technologies, refers to the process of transferring the runtime data of a virtual machine from one physical host (the source) to another machine (the destination). The execution of the virtual machine is not interrupted during most of the transfer. This allows administrators to manage system resources on-the-fly and simplifies maintenance. Most migration technologies ([1], [3], [8]) only focus on the runtime state of virtual machines within a local network, assuming that the file system could be provided by some shared storage over the LAN. Unfortunately, although necessary for some important solutions, such as disaster recovery, live migration through low bandwidth network is not as easily achievable. The major reason is that shared storage is no longer realistic. Besides, a virtual machine running on local storage has better performance, security and privacy.

In this thesis, we propose a Virtual Machine Management System (VMMS), which uses an application running inside a specific user domain to provide the migration services. This application manages and migrates the disks for all other virtual machines. VMMS also provides loss prevention with the migration. By abstracting the interface between our system and the hypervisor, this system can work with more than one hypervisor, such as Xen, KVM, etc. This system is part of the MAVIS project which aims to provide live migration over low-bandwidth, high-delay networks. Other work in this project proposes an algorithm ([18], [24]) to allocate bandwidth for different physical hosts and internal storages.

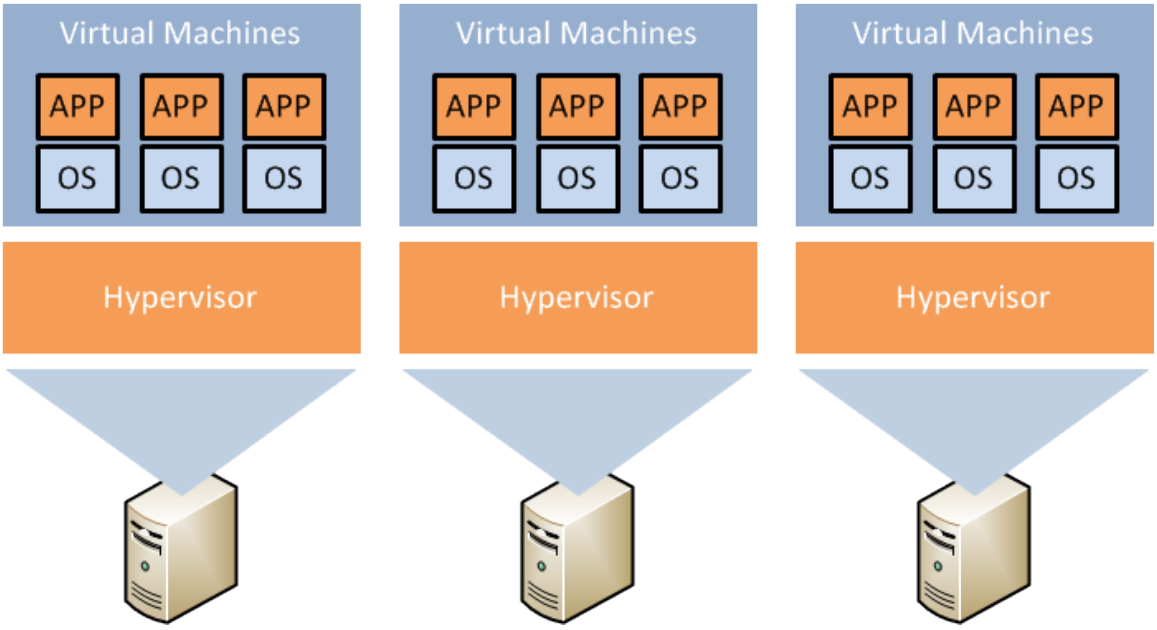

Figure 1.1: Virtualization Environment

1.1 Definitions

In the following, we will frequently refer to several components in the virtualization environment. However, different names have been used to describe these components, which may cause confusion. Thus, we give a list of the definitions that we use. The virtualization environment is shown in Figure 1.1.

- **Hypervisor:** Hypervisor is Virtual Machine Monitor. It allows several operating system instances to run concurrently inside a physical host. The hypervisor provides a virtual platform for the guest operating systems and monitors their execution. All the physical resources, such as CPU and memory, are managed by the hypervisor.

  According to the hardware access schema, hypervisors could be grouped into two categories: the native hypervisor which runs directly on bare hardware, and the hosted hypervisor which runs on top of another hosting operating system. However, these two types are treated the same in our system.

- **Physical Host:** A physical host refers to the physical machine which has a hypervisor running on it. Several virtual machines could exist on a single physical host, and this physical host includes all physical resources shared between different virtual machines.

- **Domain:** A domain refers to a guest operating system that is installed inside the hypervisor and runs in an isolated environment from other domains. Each domain could use a guest OS that is different from the physical host and provide its own services. In this thesis, we do not

differentiate a domain and a virtual machine.

- **Administrative Domain:** This is a special domain and is sometimes named as Dom 0 [1]. It refers to a guest operating system which has several special management privileges. Usually, several administrative tools are built inside this domain, and users can log into it to manage user domains or access the hardware directly. A physical host is allowed to run only one administrative domain. Currently, all existing migration solutions require some applications running inside the administrative domain.

- **User Domain:** A user domain refers to a guest operating system that runs on simple, virtualized hardware abstractions. Unlike the administrative domain, a user domain can only access the virtualized hardware, unless specified by the administrator, which isolates it from other user domains. When we talk about migration, the virtual machine being migrated must be a user domain.

- **Runtime State:** refers to the CPU State, memory data, I/O state, network connections, etc of a virtual machine. These data are small enough so that they can be transferred quickly. Currently, most hypervisors provides a built-in tool to migrate the runtime state.

- **Persistent State:** Persistent state refers to the file system data for a virtual machine. Any disk attached to a virtual machine may hold persistent state. Unlike the runtime state, it is usually quite large, and with the built-in migration tool, the persistent state is not transferred because of its large size.

1.2 Problem Definition

The problem we would like to solve is high performance live migration and the provision of loss prevention functions over low-bandwidth, high-delay networks.

The solution must include the migration of persistent state as well as runtime state because a low-speed, high-delay interconnect makes shared storage infeasible. To make this process transparent for client applications running inside the migrated virtual machine, we need to minimize the service downtime and also the performance impact during the process. In Chapter 2, we analyze this problem by giving some metrics frequently used in the migration and build a migration model to evaluate the performance of our system. In addition to the migration, our system needs to provide loss prevention. Users are allowed to create snapshots and full backups for a virtual machine.

To make our system available for more than just one specific hypervisor, we need to abstract a general virtual machine management interface, and the rest of our system should only talk with the hypervisor by accessing this interface. Another important requirement for our system is that it should not involve any modification to the hypervisor administrative domain or the guest operating system.

1.3 Our Contribution

Our contribution includes the design and implementation of a system-level solution, VMMS, for the management of virtual machines in a given virtualization environment. Users are allowed to manage all the virtual machines running on different physical hosts through our system, and it can migrate a live virtual machine, including its runtime state and persistent state, through a low-bandwidth, high-delay network. To render it practical, instead of modifying the administrative domain or deploying some extra components, this system only involves applications inside a specific user domain, called the management virtual machine, to provide services for management and migration.

To support live migration through a low-bandwidth, high-delay network, the key problem is the transfer of the persistent data (the disk images). The essence of our approach is a new block device driver, the Synchronized Block Device (SBD), which runs in our management virtual machine and provides storage functions for other virtual machines running on the same physical host. It uses a local bitmap to track the dirty disk data. During the migration, it supports state transition within a very short time, and then uses a hybrid push-pull mechanism to make sure that the disk data are synchronized. We also discuss two methods to perform network redirection after the virtual machine is running at the new site, which keeps existing network connections alive.

Unlike previous live migration research work, our system does not contain extra components to process the runtime state. Instead, it works as a cooperative utility with the migration tool built inside the administrative domain, which makes the persistent state migration procedure occur in parallel with runtime state transfer. We invoke a ballooning action inside the migrated virtual machine before the runtime state migration to zero the domain memory pages that are not used by the guest OS. We then intercept and compress the migration data to reduce its size for transfer over the network. This compression operation needs to be finished very quickly, so that it does not increase the migration time.

In addition to the live migration, we also provide loss prevention function, including snapshots and full backup support. The snapshot function allows users to record the system state changes since a particular time point for virtual machines, while the full backup saves the whole virtual machine state. Later on, users can use these saved data to recover a specified virtual machine state. This operation is supported by the SBD.

1.4 Related Work

First we describe some frequently used techniques in migration. Then we categorize current migration algorithms into three groups: cold migration, live migration with shared storage, and live migration with local storage.

Writable Working Set: In [3], the authors define the Writable Working Set (WWS). During a live migration of a virtual machine, the runtime state, especially the memory data, is transferred

while the operating system and all services on this system continue to run. This causes the problem, however, that some transferred memory pages are subsequently modified. These pages then must be transferred again. For most workloads, there exists a set of memory pages that are updated very frequently, and it is not worth attempting to transfer these pages before the virtual machine is suspended. This set of pages is called the WWS.

Stop-and-copy: This is the basic migration method. It first stops the migrated virtual machine. Then all the memory pages and other necessary device metadata are transferred over the network. After that, the virtual machine is started on the destination. Stop-and-copy transfers only the necessary data, but the virtual machine is down during the whole process, which is unacceptable for live migration.

Pre-copy: Pre-copy [20] involves a bounded iterative push phase and then a typically very short stop-and-copy phase. It first transfers the memory pages iteratively, in which the pages modified in a certain round will be transferred later in the next round. This iterative push phase continues until the WWS is reached. Then the virtual machine is suspended and the remaining dirty data are transferred. Pre-copy is widely used in runtime state migration ([2], [3], [13], [6])

Post-copy: Post-copy [8] also refers to pure demand-migration. It starts a short stop-and-copy phase at the beginning of the migration, which transfers the essential kernel data and other necessary structures to the destination. The destination virtual machine is then brought up, and other memory pages are pulled from the source host across the network on their first use. This usually results in a very small service downtime. However, the performance after the migration can be unacceptable because the virtual machine will have a high proportion of memory access faults. This situation lasts until a large set of pages has been fetched. In this thesis, to simplify the problem, we assume that the essential data to be transferred during the stop-and-copy phase is WWS.

1.4.1 Cold Migration

The *Internet Suspend/Resume* project [11] explores the new capability for mobile computing which simulates the opening and closing of a laptop computer without physical transport of the real hardware. With this system, users are allowed to suspend work at one machine and then resume it at another. This capability is achieved based on virtual machine technology with a distributed file system. The execution state of a virtual machine is mapped to files in the local file system by the virtual machine monitor (VMM). Once the virtual machine is suspended, these files are updated to reflect the latest state. Then a distributed file system can serve as the transport layer for pushing the suspended virtual machine state across the network.

This explicit copy in/copy out approach allows virtual machines to access the local file system before and after the migration. To improve performance, the authors use differential file transmission techniques to reduce the amount of data to be transferred. Also the system allows the VMM to begin execution before the reconstruction of the input files is complete, which is called incremental

reconstruction.

Different from *Internet Suspend/Resume*, our efforts are concerned with the migration of live, running virtual machines. However, we also improve the the service downtime using the same idea as incremental reconstruction during the persistent state transfer. The virtual machine is allowed to run even before all the disk data are synchronized.

Zap [14] is a system based on partial OS virtualization which allows the migration of processes on the system. It introduces pods, which are groups of processes that are provided a consistent, virtualized view of the system. Based on a thin virtualization layer, *Zap* separates the processes from the host operating system and other processes. However, during the migration, pods are entirely suspended, copied and resumed, so *Zap* does not support live migration.

1.4.2 Live Migration with Shared Storage

NomadicOS [6] project builds a virtualization and migration system on top of the L4 microkernel [7]. It creates a fundamental program which works like the BIOS or the firmware installed in computers today. Based on the microkernel, this program encapsulates the specifics of installed hardwares and also the processing units and memory to construct a distributed infrastructure. Then to migrate a running guest OS instance, NomadicOS implements a pre-copy mechanism to transfer the runtime state. It can achieve a very short service downtime. However, it makes no usage of the WWS behaviour of the migrating OS.

The authors in [8] propose a system which is based on post-copy to migrate a virtual machine across a Gigabit LAN. They show improvements in several migration metrics. They also use adaptive pre-paging in order to eliminate all duplicate page transmissions. To eliminate the transfer of free memory pages during the migration, they start a self-ballooning mechanism which periodically releases free pages in the guest VM back to the hypervisor. This self-ballooning mechanism significantly speeds up the migration with negligible performance degradation.

The post-copy mechanism proposed is the basis of our hybrid push-pull synchronization for the block device data. After migration, our system starts the active pushing and demand paging to synchronize the dirty data, which is a combination of two variants of post-copy.

1.4.3 Live Migration with Local Storage

Xeno Server [2] has a component that supports the live migration of a virtual machine with its local persistent state. It is based on the split-driver architecture in Xen for block devices. This architecture is a front-end that provides a raw block device in the guest VM communicating with a backend in the administrative domain through a write buffer. Xeno Server pre-copies the disk image at the beginning of migration, and for any data dirtied later, it will intercept the write operations and transfer the delta data to the destination. It uses Xen built-in migration program to transfer the runtime state.

Like Xeno Server, our implementation uses the Xen built-in program to transfer the runtime state. However, SBD does not require the split-driver architecture and uses a different mechanism to synchronize the disk data.

In [13], the authors propose a whole-system live migration scheme. To minimize the huge service downtime caused by the disk data synchronization, a three-phase migration algorithm is used. The disk data is transferred to the destination in a pre-copy process before the migration. When the virtual machine is suspended, only a block-bitmap is transferred and the remaining dirty data will be synchronized after the virtual machine is running on the destination. They implemented a prototype of this algorithm using the Xen split driver. Our implementation uses the same idea to migrate the disk data but with a totally different system structure to support a variety of hypervisors and loss prevention.

1.5 Organization of the Thesis

The rest of this thesis is organized as follows: In section 2 we give the performance metrics to evaluate the migration system. Then we analyze the problem by proposing the migration model, and describe our goal. With this analysis, we can have a general view of our migration scheme. In section 3, we explain our system design and the general structure of VMMS. In section 4, we talk about the implementation details of the memory migration and disk data transfer. Then we discuss the network redirection and loss prevention functions in our system. In section 5, we show the performance results of our system. Finally, in section 6, we conclude and outline future work.

Chapter 2

Problem Definition

Our system is designed to migrate the whole virtual machine from the source to the destination host, including both runtime state and the persistent state. The target virtual machine should keep running during most of the migration time. In this section, we describe some metrics to evaluate the performance of migration algorithms, and based on these metrics, we propose a model of the migration process and discuss our goals.

2.1 Performance Metrics

The following metrics are frequently used to evaluate the performance of a migration algorithm:

- **Migrated Data Size** is the total amount of data that is transferred through the network during the whole migration process. Usually, it includes the memory data, CPU state, and in our system, the file system data. However, it may be larger than the sum of these data, because most migration schemes need transfer some extra data according to their protocols. The only exception is the stop-and-copy method.

- **Service Downtime** refers to the time interval after the source virtual machine is suspended and before the new virtual machine is brought up on the destination host. During this time, services provided by the virtual machine will be entirely unavailable. The length of this period depends on the data size that cannot be transferred when the virtual machine is still running.

- **Total Migration Time** is the time period from when the user triggers the migration command to the time when the whole system state has been transferred and all the data is synchronized. The total migration time depends on the migrated data size.

- **Bandwidth Usage** is the usage of network bandwidth during the migration. As several service applications may run inside the target virtual machine, and these services usually share the same network interface with our migration system, we would like to limit the usage of network bandwidth by our migration system to minimize the impact on those services.

Based on the metrics described above, our migration system should transfer the whole virtual machine state with a very short service downtime and reasonable migration time. Moreover, during the migration, the performance overhead should be negligible, and the system should use only a small part of the network bandwidth. Thus, our goals include the following:

- **Data Consistency:** this guarantees that all the system data of the target virtual machine, including its runtime state, file system data, network connections and external devices, are consistent and identical before and after the migration. This is the basic requirement for migration.
- **Minimal Service Downtime:** because during this time period, all the services are unavailable, and no currently executing virtual machine is available, our system must minimize the amount of data transferred during the downtime.
- **Live Migration:** based on the minimum service downtime, we must provide a live migration, which means the virtual machine should keep running during most of the migration. Clients accessing services provided by this virtual machine should not notice any interruption during migration.
- **Minimal Migration Time:** A long migration time makes the migrated virtual machine dependent on the source host for a long period. Thus, the total migration data size should be reduced. We achieved this by using memory ballooning and compression of the migration data.
- **Finite Migration Process:** this requires that at the end of migration, the whole system is running on the destination host, which means that the migration process must be finished in a finite period of time. After this time, we can shut down the virtual machine on the source host and close all the connections.

With the above metrics and requirements, we give a deterministic analysis of the migration problem. In the following sections, we present the design and implementation of the system to satisfy these requirements.

2.2 Migration Model

In this section, we analyze the live migration problem and propose a model for the whole process to estimate the performance of our system working with different runtime migration schemes. This allows us to have a better knowledge about the whole migration procedure, and it is used to estimate the performance of the migration system.

As stated before, our system needs to process both the runtime state and persistent state, so we include them in this model to calculate the time and bandwidth impact of migration. Assuming that

the memory and disks could be processed in our system as pages with a fixed size equal to P bytes, we treat a virtual machine as a set of memory pages $\{m_1, m_2, ..., m_M\}$, and a set of disk pages, $\{d_1, d_2, ..., d_D\}$, where M and D are the memory size and disk size in pages. As we state above, the migration system should use a limited network bandwidth to lessen the impact on other services. We assume the bandwidth available for the transfer is constant and equal to B bytes per second. Therefore, the time cost to transfer a single page is:

$$T_{page} = \frac{P}{B}$$

if no compression is used. To make a state transition for the disk when all the bandwidth is available, the time interval is:

$$T_{Tran} = \frac{D/8}{B}$$

where $D/8$ is the bitmap size we need to transfer during the transition. Because the virtual machine keeps running during most of the migration time, we define the memory dirty rate as W_M, and disk dirty rate W_D, showing how many pages are dirtied per second during the migration. Then we define the average compression ratio C_M for memory pages and C_D for disk pages. To simplify the problem, we assume all of these rates have constant values. This is acceptable because the total migration requires a relatively short period. At last, we define the preparation time for the migration as T_{PREP}. With these definitions, we analyze the performance of several runtime migration schemes, including stop-and-copy, pre-copy, post-copy. The process of the migration is as follows:

1. **Preparation**

   After the user issues the migration command, the system starts the preparation stage, where SBD starts to push dirty data to remote site. During this time period, only the persistent state is transferred, so the network bandwidth it can utilize is B. We calculate the new memory size and disk size after the preparation time:

   $$\begin{aligned} M_1 &= M \\ D_1 &= D - (B/P - W_D) \times T_{PREP} \times C_D \end{aligned}$$

   which means the memory size does not change. For the disk data, some pages are dirtied during the synchronization, so the actual transfer rate is $(B/P - W_D)$.

2. **Pre-Stage**

   The Pre-Stage refers to the time period before the virtual machine is stopped. During this time, pre-copy will copy the memory pages in several iterations until WWS is reached. We assume the network bandwidth for the migration of runtime state and persistent state are both $B/2$. Then for pre-copy:

   $$T_{PRE-STAGE} = \frac{M_1 - WWS}{(B/2P - W_M) \times C_M}$$

$$
\begin{aligned}
M_2 &= WWS \\
D_2 &= D_1 - (B/2P - W_D) \times T_{PRE-STAGE} \times C_D
\end{aligned}
$$

where the $T_{PRE-STAGE}$ shows how long the pre-stage will last for the pre-copy scheme. The system transfers the runtime data at rate $(B/2P - W_M)$, until WWS is reached. Then we calculate the remaining memory and disk data.

For the stop-and-copy and post-copy schemes, this stage involves no operation, so in these two cases:

$$
\begin{aligned}
T_{PRE-STAGE} &= 0 \\
M_2 &= M_1 \\
D_2 &= D_1
\end{aligned}
$$

3. **Stop-Stage**

   After the pre-stage, the virtual machine on the source host will be stopped. Then depending on the specific migration scheme, different data will be transferred across the network to the destination host, and the new virtual machine will be brought up. For stop-and-copy and pre-copy, we need to transfer all the remaining runtime data, so the new state will be as follows:

$$
\begin{aligned}
T_{STOP-STAGE} &= \frac{M_2}{(B/2P) \times C_M} \\
M_3 &= 0 \\
D_3 &= D_2 - (B/2P) \times T_{STOP-STAGE} \times C_D
\end{aligned}
$$

   which means we need to stay in this stage until all the runtime data are transferred. Notice that because the virtual machine is stopped, there will be no dirty pages generated. As a result, the transfer rates for both runtime state and persistent state become $B/2P$. For post-copy, this stage involves only a transfer of WWS, so it generates the following new state:

$$
\begin{aligned}
T_{STOP-STAGE} &= \frac{WWS}{(B/2P) \times C_M} \\
M_3 &= M_2 - WWS \\
D_3 &= D_2 - (B/2P) \times T_{STOP-STAGE} \times C_D
\end{aligned}
$$

4. **Post-Stage**

   Now the new virtual machine is running on the destination host. For stop-and-copy and pre-copy, the whole runtime state is migrated. Nothing needs to be done in this stage. So we have:

$$
\begin{aligned}
T_{STOP-STAGE} &= 0 \\
M_4 &= M_3
\end{aligned}
$$

$$D_4 = D_3$$

For post-copy, the remaining runtime data will be pulled from the source host. SBD will also pull dirty pages in this stage.

$$\begin{aligned} T_{STOP-STAGE} &= \frac{M_3}{(B/2P - W_M) \times C_M} \\ M_4 &= 0 \\ D_4 &= D_3 - (B/2P - W_D) \times T_{STOP-STAGE} \times C_D \end{aligned}$$

5. **Final-Stage**

   At this point, the migration of the runtime state is finished for all migration schemes, so in this stage, we will wait for SBD to synchronize its persistent data.

$$\begin{aligned} T_{FINAL-STAGE} &= \frac{D_4}{(B/P - W_D) \times C_D} \\ M_5 &= 0 \\ D_5 &= 0 \end{aligned}$$

Summary

With the above analysis, we can get a general view of how the migration works. According to the the performance metrics we defined in section 2.1, we get the general results for VMMS working with the three runtime state migration schemes:

$$\begin{aligned} \text{Total Migration Time} &= T_{PREP} + T_{PRE-STAGE} + \\ &\quad T_{STOP-STAGE} + T_{POST-STAGE} + \\ &\quad T_{FINAL-STAGE} \\ \text{Service Down Time} &= T_{STOP-STAGE} \\ \text{Migrated Data Size} &= \text{Total Migration Time} \times B \\ \text{Bandwidth Usage} &= B \end{aligned}$$

The migrated data size and bandwidth usage are the same for all three schemes. To conclude the remaining metrics, we simplify the migration model by assuming that there is no preparation stage and $W_M = W_D$, $C_M = C_D$. Then results for the three migration schemes are given in Table 2.1.

It is obvious that with stop-and-copy, the total migration time depends on the memory and disk size of the virtual machine, but it requires a long service downtime to transfer all the memory data. With pre-copy and post-copy, more time is required because the virtual machine is still running during memory migration. It is the memory and disk dirty rate that decide the extra time we need.

Migration Scheme	Total Migration Time	Service Down Time
stop-and-copy	$\frac{M}{B/2P \times C_M} + \frac{D-M}{(B/P-W_D) \times C_D}$	$\frac{M}{B/2P \times C_M}$
pre-copy	$\frac{M-WWS}{(B/2P-W_M) \times C_M} + \frac{WWS}{B/2P \times C_M} + \frac{D-M}{(B/P-W_D) \times C_D}$	$\frac{WWS}{B/2P \times C_M}$
post-copy	$\frac{WWS}{B/2P \times C_M} + \frac{M-WWS}{(B/2P-W_M) \times C_M} + \frac{D-M}{(B/P-W_D) \times C_D}$	$\frac{WWS}{B/2P \times C_M}$

Table 2.1: Performance Metrics of Different Schemes

The dominant factor for the service downtime is the WWS size. In [3], the authors propose a method to measure the WWS size.

Example: VMMS with Xen

Xen has a built-in migration tool which uses pre-copy scheme for runtime state migration. It requires a shared device to provide file system for the migrated virtual machine. In this section, according to the previous migration model, we would like to compare the performance of VMMS to the Xen built-in migration. The time costs of each stage for the two approaches are given in Table 2.2.

Stage	Xen	VMMS (Shared Storage)	VMMS (Local Storage)
$T_{PRE-STAGE}$	$\frac{M-WWS}{B/P-W_M}$	$\frac{M-WWS}{(B/P-W_M) \times C_M}$	$\frac{M-WWS}{(B/2P-W_M) \times C_M}$
$T_{STOP-STAGE}$	$\frac{WWS}{B/P}$	$\frac{WWS}{B/P \times C_M}$	$\frac{WWS}{B/2P \times C_M}$
$T_{FINAL-STAGE}$	0	0	$\frac{D-M}{(B/P-W_D) \times C_D}$

Table 2.2: Performance Comparison

Here we give a specific, numerical example to evaluate the performance of VMMS. Based on the experiment with FTP server workload described in chapter 5, we make the assumption that before migration, the virtual machine has the following states: M=131072 pages, WWS=512 pages, B=5.625 MB/s, W_M=W_D=512 pages/s, C_M=C_D=4, and the disk has 262144 pages to be synchronized. Then we have the results in Table 2.3.

Stage	Xen	VMMS (Shared Storage)	VMMS (Local Storage)
$T_{PRE-STAGE}$	110.27s	27.56s	70.34s
$T_{STOP-STAGE}$	350ms	88.9ms	177.78ms
$T_{FINAL-STAGE}$	0s	0s	27.67s
Total	110.61s	27.64s	98.19s

Table 2.3: Performance Example

With shared storage, VMMS saves about 80s for $T_{PRE-STAGE}$ because of the ballooning and compression. It requires 70.34s to complete the pre-stage when the virtual machine is migrated with local storage, due to less bandwidth being available at this time. In addition, VMMS with local storage needs another 27.67s to synchronize the remaining dirty data for the disk. In this stage, applications accessing disk data will get lower performance until the synchronization is finished. From these results, we expect that VMMS can migrate the runtime and persistent state of a virtual machine in less time than the Xen built-in migration tool requires to migrate just the runtime state. We then compare the experiment results in chapter 5.

Chapter 3

System Design

In this section, we summarize the design decisions for our migration system. We start by describing the system structure and then the general steps of the migration procedure. To support our migration algorithm, a migration client must run somewhere inside the source host, communicating with the daemon on the destination host. Also, the new block device driver, SBD, must be deployed on both sides to provide local support for migration of the persistent state.

3.1 Operating Environment

VMMS is designed to provide a front-end for users to manage all the virtual machines running on the current physical host. With this system, the virtual machine can be migrated through a low-bandwidth, high-delay network with its persistent state, involving only a small service downtime. We deploy a subsystem, VMMS Node (VNODE) inside a virtual machine running a patched CentOS 5.4 Linux guest system. The virtual machine to be migrated must be a user domain, and the physical hosts involved in the migration must be connected by a TCP/IP network. VNODE allows users to import block devices for SBD to use, and these devices will be managed by SBD to provide iSCSI targets for the administrative domain. Virtual machines to be migrated with their persistent state should use only block devices from these iSCSI targets for storing root file systems and swap files.

Generally, VMMS tries to avoid accessing the internal data of the guest operating system. Thus, it could be Windows, Linux, etc. However, some hypervisors do have constraints on the guest operating system option. For instance, without Hardware Assisted Virtualization Technology (e.g. VT from Intel, or Pacific from AMD), Xen only supports para-virtualized Linux. VMMS does not add any more constraints and thus should work with any guest operating system that the underlying system supports.

3.2 VMMS Architecture

The general structure of VMMS is shown in Figure 3.1. It contains two major subsystems: Central Management Service (CMS) and VNODE. CMS runs as a Web server on a separate host, either a

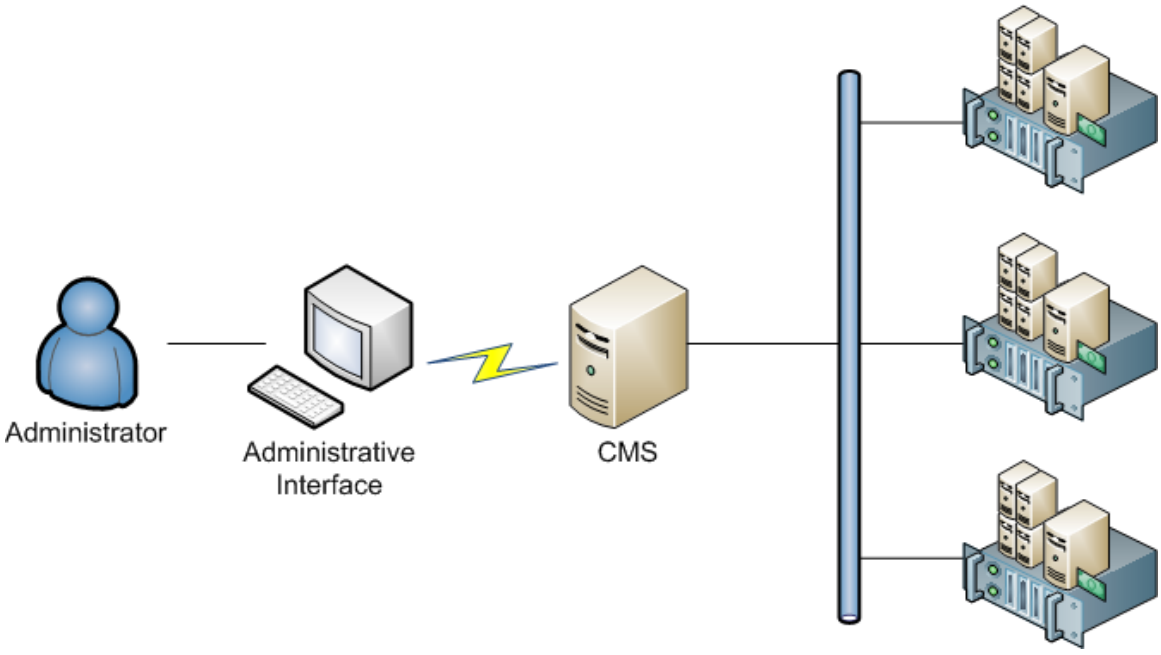

Figure 3.1: VMMS Structure

virtual machine or physical machine, to provide the user interface. Users need to log into CMS to operate on the virtual machines that are managed by the current CMS. For each hypervisor, we have a VNODE running on top of it as a virtual machine. VNODE receives commands from CMS and complete these requests via the hypervisor-specified API.

As we stated before, to render it practical, we install the VNODE subsystem without making any change to the administrative domain or the guest operating systems. This is important to keep the whole system stable. To achieve this goal, VNODE is deployed inside a virtual machine. We will refer to this virtual machine as VNODE Virtual Machine. The other benefit from this architecture is that we can design our system in such a way that it will work with all hypervisors. The system structure of VNODE is shown in Figure 3.2. It contains five major components to interact with CMS and the hypervisor:

- **Management Service:** Listens for management requests from CMS. These requests will be finished by using the Abstract Hypervisor API. This service provides an administrative interface for CMS to manage the virtual machines running in current physical host. CMS should provide security options so that only trusted objects can talk with this system.

- **Data Transfer Service:** Provides a tunnel for original data transfer. This service intercepts all data that should be transferred to the remote host during migration, so these data can be processed to meet our system requirements. Data Transfer Service also works as the target for full backup operation. Because different hypervisors use different data transfer protocols, Data Transfer Service needs to use the Hypervisor Data Pattern (see below) to analyze the

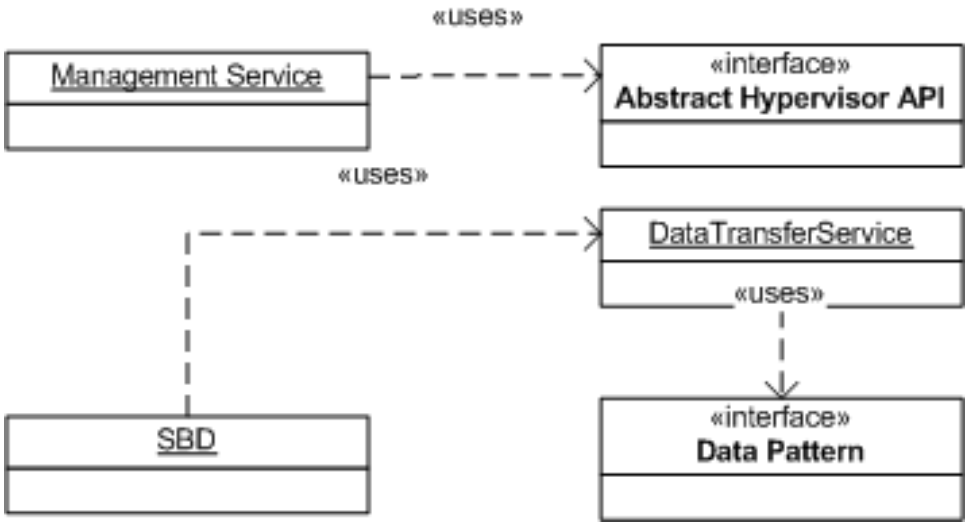

Figure 3.2: VNode Structure

data received.

- **Synchronized Block Device:** This component is used to provide file systems for all virtual machines in the current physical host. VMMS integrates an interface to manage the disks of the virtual machines, and these disks are exposed to the Administrative Domain as iSCSI targets. SBD also supports fast state transition to allow live migration of persistent state. During this transition, only a bitmap needs to be transferred. The other dirty data will be synchronized through a hybrid push-pull mechanism.

- **Abstract Hypervisor API:** Our system can communicate with different hypervisors using the same protocol. Abstract Hypervisor API talks directly with the hypervisor. It handles the differences between hypervisors and provides a unique interface for VMMS. Based on this abstract interface, VMMS can manage different hypervisors. During the installation of VMMS, users will be asked to provide enough information for this component to talk with the hypervisor. This information typically includes, for example, hypervisor type, IP address, management account, etc.

- **Hypervisor Data Pattern:** Describes the data transfer pattern of the underlying hypervisor. This pattern will be used by the Data Transfer Service for analysis and processing. The Hypervisor Data Pattern needs to enable the Data Transfer Service to detect the virtual machine suspension operation and the migration completion boundary for other components.

Two major components, the Abstract Hypervisor API and Hypervisor Data Pattern, are related to the underlying hypervisor. When we deploy VNODE on a new type of hypervisor, these components need to be rewritten so that the other elements will work properly. Because CMS only interacts with VNODE, it can work with all types of hypervisors supported by VNODE.

With VMMS, users can manage all virtual machines through CMS, and these management requests will be processed by VNODE using the Abstract Hypervisor API. However, the migration procedure involves several steps. On receiving a migration command, VNODE will trigger the hypervisor built-in migration application to transfer the runtime state to Data Transfer Service. The data will be processed and transferred to the remote Data Transfer Service. When the virtual machine is suspended, VNODE will notify SBD to make a state transition to migrate the persistent state.

Many types of business system could potentially take advantage of the infrastructure of our virtual machine management solution. For instance, in a clustering environment, administrators could manage all the virtual machines through a consistent front-end, without knowing the underlying hypervisors. In other words, with this structure, we provide a unified user interface for different hypervisors.

3.3 Migration Procedure

For CMS, requests from the users can be completed by VNODE through the Abstract Hypervisor API, and this usually involves a simple system invocation. However, the actual details of the migration process are complex. In this section, we describe the operation of the system during the migration process with Xen platform.

1. **Initialization** CMS listens for the migration command and handles the authentication and access control of the requests. After users submit a migration request, CMS finds out which VNODE is managing the target virtual machine and sends an internal migration message to the Management Service in that VNODE. Then CMS records all the information about this migration request, including the virtual machine ID, source and destination host IP addresses, in a local database. These will be used later to intercept the migration process.

   There are two options to start the SBD synchronization. If this migration has been previously scheduled, the synchronization could be started long enough before the migration so that most of the disk data will be synchronized at this point. Alternatively, if this migration request is an emergency, the synchronization will be started automatically during this stage, leaving most of the disk data to be synchronized later through the push-pull mechanism.

2. **Runtime State Pre-copy** VNODE then triggers a ballooning program inside the target virtual machine to zero all the free memory. This removes the transfer of meaningless data through the network and thus reduces the total migration time. Then a request is sent to the hypervisor built-in migration program to start the runtime state transfer. However, instead of the original destination, we use VNODE Data Transfer Service as the target host. Data Transfer Service will analyze the data using the Hypervisor Data Pattern. For each data segment, some specified operations will be applied. Usually this involves compression and encryption. Because all

the free memory has been cleared, we expect a high compression ratio for these data and a greatly reduced transfer time. Then the data will be transferred to the VNODE on the real destination host, processed by the inverse operations, and sent to the hypervisor. Notice that SBD synchronization is also running during this stage.

In Xen, the runtime state transfer is based on the shadow page table [1], which is created below the running guest OS. It is used to track pages that dirtied during the transfer procedure. All entries are first initialized as read-only, and upon guest system writing operations, the page fault is trapped by Xen and the corresponding bit is set in the dirty bitmap. Then the migration tool uses this bitmap to decide which pages should be transferred again in the next iteration.

3. **Role Transition** After WWS is reached during the runtime state transfer in the hypervisor built-in migration program, the target virtual machine will be suspended and WWS will be transferred. Data Transfer Service analyzes the data, detects the suspension message, and notifies SBD to make a state transition. This involves only a dirty bitmap transfer and requires only a small amount of time. After WWS is transferred, the source virtual machine will be destroyed, and the new VM is brought up on the destination host. At this time, SBD enters the push-pull phase to synchronize the remaining dirty data.

4. **Finalization** At this stage, the virtual machine is running on the destination host. From the user's perspective, the migration is already completed. However, SBD still needs some time to finish the synchronization. Source SBD device will keep pushing dirty disk pages to the target host. Any dirty disk page that is accessed by the virtual machine on the destination host will be pulled from the source device. After all dirty pages are synchronized, SBD will make another state transition to terminate the push-pull phase. After that, the whole migration is finished.

Chapter 4

Implementation Details

In the following sections, we describe the details of our system implementation. We first present the ballooning technique used during the memory migration and the memory data path in our system. Then we give a detailed description of the structure of our Synchronized Block Device. After that, we discuss possible ways to handle network connections after the migration. Finally, we describe our method to create a snapshot and full backup for a virtual machine.

As for each function, several design choices may exist. Here we list our implementations and talk about its advantages:

1. **Ballooning:** Instead of talking with the hypervisor and returning the unused memory, our ballooning program simply allocates all unused memory and clears its contents, and the compression in the migration data path will remove this part of memory. It does require some extra time but makes our ballooning process independent on the hypervisor.

2. **Migration System Deployment:** Our system deploys all components in a special user domain, instead of Dom 0. This involves more components and causes some problems to process the migration data. However, it also removes the dependency on the hypervisor, and makes the hypervisor stable. Even if the migration system fails, it will not have any impact on the hypervisor and other virtual machines.

3. **Snapshot Scheme:** The snapshot in our implementation is based on Copy-on-Write scheme. It has several advantages over the Redirect-on-Write. The disk always contains the latest status of data. If all the snapshot data files are lost, the disk can still work normally. Besides, only first write involves extra I/O operations, which brings better performance. To save storage, we use nested snapshot, which means to rollback a snapshot, users need to restore disk data from all the snapshots after the desired one. However, this is acceptable because rollback is a offline operation and it's not time critical.

4.1 Ballooning

We implemented runtime state migration using hypervisor built-in program. However, today's computer systems typically have several gigabytes of memory, which may require a long time to transfer through a low-bandwidth network. This will result in an unacceptable waste of network bandwidth during the migration. To reduce the amount of data we need to transfer during this process, one useful technique is to eliminate or page out the less useful data in the guest operating system. In other words, unused memory pages will be returned to the hypervisor. This method requires cooperation between the guest operating system and the hypervisor, and thus, it involves some modification of the guest operating system.

The other possibility proposed in [17] is to clear all unused memory pages of the migrated virtual machine. Then before transfer to the destination host, these pages will be compressed. This is accomplished by triggering a ballooning program that allocates all unused physical memory pages in the guest operating system and then zeroes these pages, making them easily compressible. The ballooning process involves no interaction with the hypervisor, and can be completed by a user-space program. This ballooning technique works especially well for virtual machines with a large number of unused memory pages. Transferring these data over the network to the destination makes no sense for the migration, and should be avoided. The performance impact of this ballooning technique is presented in Chapter 5.

We implemented this ballooning program for both Linux and Windows operating systems. For these two systems, we use the specified API to allocate memory. For Windows, there might be a limitation on the maximum working set for each process. To make our program effective, we must ensure that the limitation for our ballooning process is set to the maximum. For Linux, our program reads the /proc/meminfo file to decide when to stop the ballooning process. Currently, once we detect that the free swap space has decreased by more than 1MB, which means most of the physical memory is allocated and the system is starting to use the virtual memory, the ballooning program will exit.

The other requirement for this ballooning technique is that the compression process must be finished in a much shorter time than the period to transfer a memory page over network, or it will make no sense to do the ballooning and compression. Our primary target is to remove the all-zero memory pages, so we use a simple method to do the compression. The LZW algorithm [22] is used to compress each page of data. We avoid the usage of a compression stream, because for some hypervisors, command messages require immediate response and the stream library may delay them. This implementation generates results with a lower compression ratio than the compression stream, but it will remove the all-zero pages efficiently.

4.2 Synchronized Block Device

To support full migration of a virtual machine, including the persistent data, we designed a Synchronized Block Device component. This component is implemented as a Linux kernel module and is deployed inside the VNODE virtual machine. Like the loop device in the Linux kernel, SBD acts as a normal block device and requires a backing device to actually store the disk data. For all I/O requests, SBD will pass them to the backing device. This backing device could be another block device or just a plain file. In addition to the ordinary block device access functions, SBD allows users to set up synchronization relationships between two block devices before migration. With this component, VNODE provides a storage environment for virtual machines, and a unified interface to manage all disks.

Once the synchronization relationship has been created, two SBD devices will be at either the synchronization stage or the migration stage. Most of the time, they stay at the synchronization stage. The primary task here is to keep data consistent on the remote site, while the virtual machine is still accessing the local disk. During this stage, these two devices have the roles of master and slave:

- **Master:** works as an ordinary block device. A master SBD runs on the source host and allows normal read and write operations on this device to provide a file system for the virtual machine. However, once the synchronization relationship is set up, the device will create a kernel thread named sync-thread, working iteratively to push dirty pages to the slave. Inside SBD, a local bitmap which is initialized as all dirty is maintained to track dirty pages. Sync-thread transfers these dirty pages to the slave and clears the corresponding bit in the bitmap. During this procedure, any write operation from the virtual machine will set the corresponding bit as dirty again.

- **Slave:** only allows read operations on the device, meaning all the write operations will be ignored and the backing device will not be updated. However, no exception will be raised when a write request is received, so a slave SBD can be used as an iSCSI target. Once the synchronization relationship is set up, it also creates the sync-thread which keeps receiving dirty pages from the master side and updates these data on the backing device. Thus, this is the device running on the destination host.

During the live migration of a virtual machine, after the pre-stage, VNODE will notify SBD to make a state transition once the virtual machine is suspended. Thus, it is guaranteed that no write operation will be made during this process. After transferring some metadata, the master device will change its role into mig-slave device, and the slave device will change into mig-master device. We then implement a push-pull mechanism to achieve better disk performance after the migration. Even with some dirty pages unsynchronized after the migration, the migrated virtual machine can still access all data on the disk at almost local performance. The functions of these two roles are:

- **Mig-Master:** is the original slave device. Mig-master device is running on the destination host. However, this device now begins to allow both normal read and write operations to the disk. After the migration, the new VM will operate on this device. During the synchronization stage, most of the disk data should already be transferred. When reading these data, the behaviour of mig-master disk is exactly the same as a normal block device, and we expect local performance for such disk operations. mig-master also keeps receiving synchronization data for any remaining dirty pages from the source host. For any operations involving dirty data that has not been transferred yet, mig-master will generate a PULL message and send it to the source site with its page id. This page will be synchronized in the next round.

- **Mig-Slave:** is the original master device. According to current live migration solutions, after the virtual machine is suspended and moved to the destination host, no more disk operations will be allowed on the source host, so mig-slave device does not support write requests, which is the same as the original slave disk. However, the sync-thread is still running to push the remaining dirty pages to the mig-master. Once a PULL message is received from mig-master, sync-thread will change the synchronization sequence and jump to that data area. This dirty page will be pushed to mig-master immediately. Because most operations involve neighbouring disk pages, sync-thread will start to push this data area in the next iterations.

In addition to synchronization, SBD also allows users to make snapshots or full backups of virtual machine disks. To meet these requirements, SBD device should first contain a component which handles normal block read and write functions. If users set up the synchronization relation between SBD devices, there must be some other components tracking and transferring the dirty data. The general structure of SBD device is shown in Figure 4.1. Here the R/W Handler processes the normal read and write requests from applications, working as a normal block device. However, every I/O request is intercepted by a chain of DataPath objects. These objects are of different uses. In this section, we only focus on the SyncPath, which is used during the synchronization stage to transfer data. We will talk about loss prevention via SnapshotPath and BackupPath in a subsequent section.

4.2.1 Synchronization Stage

When there is a plan to migrate a virtual machine, users could schedule a synchronization relationship between two SBD devices on the source and destination hosts. We expect that most of the dirty disk data could be synchronized before the runtime state transfer starts. During this stage, SBD has the structure shown in Figure 4.2

For the master device, after the synchronization relationship is set up, SBD initializes a SyncPath object in its DataPath list, so that all the write operations are intercepted and this SyncPath object is notified for each request. A bitmap is maintained inside the SyncPath object to track the status of

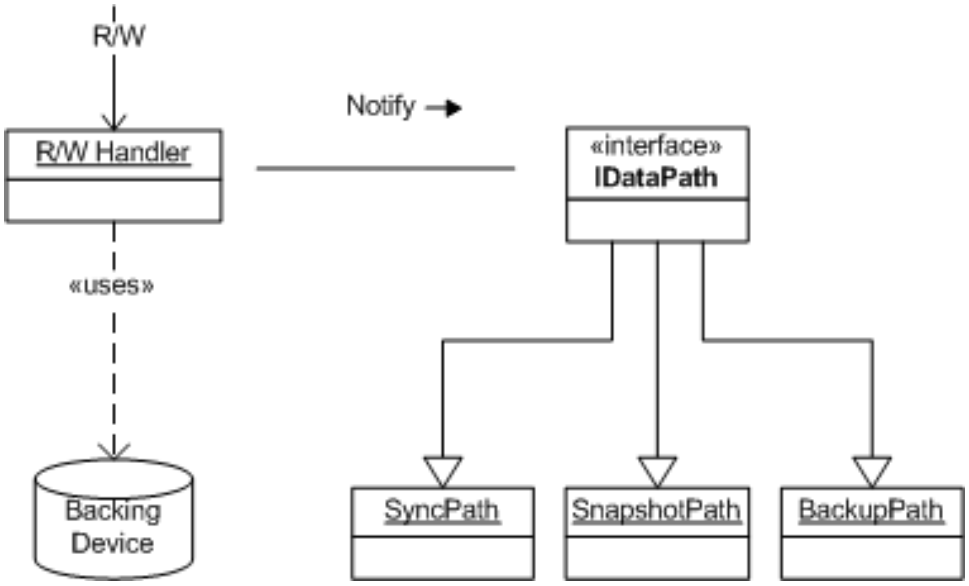

Figure 4.1: SBD Device Structure

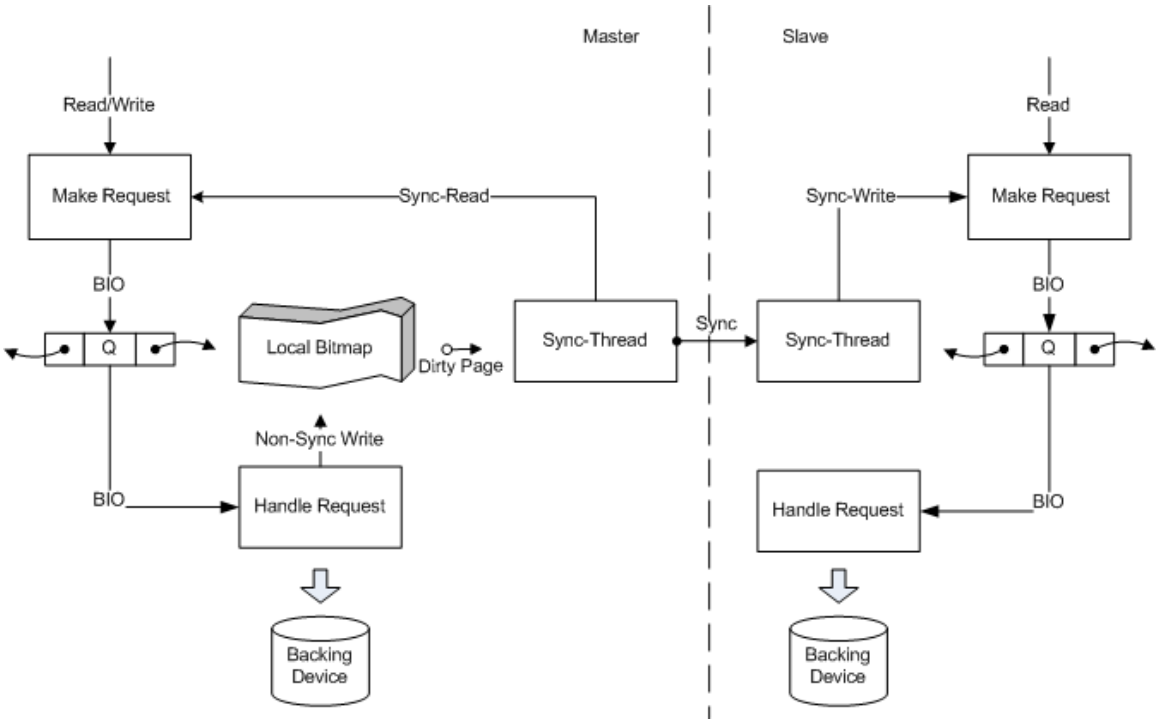

Figure 4.2: SBD Structure During Synchronization Stage

each disk page. If a disk page is modified by a write operation, it is marked as dirty in the bitmap. This dirty bitmap is used in the sync-thread created by SyncPath. It runs repeatedly scanning the bitmap and talking with the remote side. For each dirty page in the bitmap, it posts a read operation to retrieve the data from the disk. Then the page is transferred over a network connection to the remote device. The slave device is relatively simple. It also contains a SyncPath object but with a different use. This object creates a sync-thread which keeps receiving synchronization data from master device. For any page received, it posts a sync-write operation to update the local disk. Because the slave does not support write operations, all write requests other than sync-write are ignored. The details of this procedure are as follows:

1. During the synchronization stage, the virtual machine is still running and accessing the disk, so normal read and write operations are processed. Disk operations generate several I/O requests to the R/W Handler and these requests are saved in a queue managed by SBD. Because there might be several requests happening at the same time, this queue is protected by a global lock, which serializes all the requests in a multi-threaded environment.

2. SBD contains a kernel thread to process these I/O requests. It retrieves a request from the queue and completes its data operation using the backing device, so read and write operations are completed just as for an ordinary block device. In fact, with our implementation, these requests are simply passed on to the backing device, which guarantees efficiency and safety.

3. For any non-sync write (ordinary write) request, which means this write operation is generated by the virtual machine, the SyncPath object is notified by the request handler. The disk pages written are marked dirty in the local bitmap.

4. Sync-thread in SyncPath object keeps scanning the local bitmap. When it meets a dirty page, a sync-read request is generated to read the page from the backing device. Once this sync-read is completed, the dirty page is transferred over the network to the remote Sync-Thread running in the slave device.

5. On the slave side, when a sync packet is received, the Sync-Thread extracts the page ID and data from the packet and generates a Sync-Write request to the R/W Handler. This request is then processed and the dirty page is updated on the local backing device. After that, a response is sent back to the master device to confirm the synchronization.

6. After receiving the response, the Sync-Thread on the master clears the corresponding bit in the local bitmap and continues scanning. To avoid impacting the performance of normal disk operations, the scheduler in SBD runs the scanning process only when the disk is idle.

The scanning and synchronization procedures require CPU and extra disk requests, so they will have an impact on the I/O performance. To avoid this performance loss, we implement a synchronization scheduler. The scheduler tracks the normal disk operations and calculates a busy level for

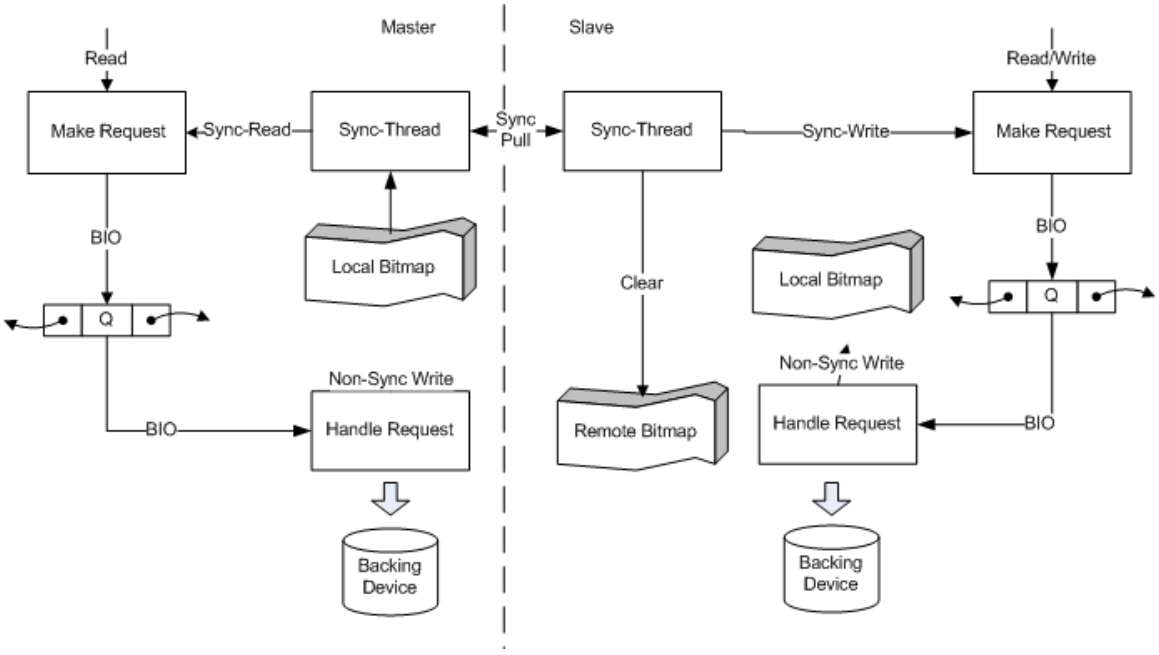

Figure 4.3: SBD Structure During Migration Stage

the disk. The SyncPath object is activated only when the disk is judged as idle according to the busy level.

4.2.2 Migration Stage

During migration, once the virtual machine is suspended, a notification from the Data Service in VNODE is sent to the master SBD through IOCTL command. This device then finishes any current block I/O request and suspends the request handler. This is safe because the virtual machine is not running now and no more write requests could be made at this time. After that, it sends a migration command to the slave device and then transfers the local dirty bitmap. Finally, the master device changes state to mig-slave and resumes running.

After receiving the dirty bitmap, the slave saves it as the remote bitmap which records the remaining dirty pages. It also creates its own bitmap to track local write operations. Then it changes the state to mig-master, and both read and write operations are allowed on this device. At this time, the virtual machine should be running using the mig-master device on the destination host. These two devices have the structures shown in Figure 4.3.

If the user creates the synchronization relationship long enough before the virtual machine migration, at the time of migration, most of the disk data should already be synchronized, and any operations on these data should get local performance. However, if an operation involves some remaining dirty data, the corresponding disk pages need to be available as soon as possible. This is achieved by using the push-pull mechanism, described it in detail here.

1. As for the original master device, mig-slave still has its sync-thread pushing dirty pages according to its local bitmap. This thread posts a sync-read request to its backing device for every dirty page, Once this request is completed, the data is transferred to the mig-master device. Note that non-sync write operations are not allowed any more on the mig-slave device, so no dirty pages will be generated.

2. After receiving a dirty page, the mig-master generates a sync-write request to its backing device. The dirty data is written to the disk and the corresponding bit in the remote bitmap is cleared. Then a response is sent to the mig-slave to clear the dirty flag of this page in its local bitmap.

3. Upon a normal read or write request on the mig-master, the request handler needs to check the status of each page using the remote bitmap. For any page that is marked as dirty, the request handler needs to suspend the current request and notify the sync-thread to pull that page. This PULL message is sent to the mig-slave immediately.

4. After receiving a PULL message, mig-slave completes its current push operation and jumps to the desired data area. The required dirty page is pushed in the following iteration to the mig-master. Once the desired page is received, mig-master resumes the last request. Note that if it is a non-sync write operation, these pages will be marked as dirty in the local bitmap for later synchronization.

   All the requests are put into a queue to serialize them. However, when we receive a dirty page, we insert the sync-write request at the front of the queue. This avoids meaningless pull operations if this dirty page happens to be accessed in the following requests that are already in the queue. During the migration stage, the sync-request has a higher priority over ordinary read/write operations.

5. During an iteration, if mig-slave detects that no more pages are marked as dirty in the local bitmap, it sends a notification to mig-master to complete the migration stage. The remote bitmap in mig-slave will be destroyed as it is no longer useful. These two devices will make another state transition and the migration is complete.

One optimization for this pull process is that on the mig-master device, for any write operation that involves dirty data, we check if some pages are fully overwritten, which means the original page data is no longer useful. In this situation, we simply clear the bit in the remote bitmap and update the local backing device. No notification even needs to be sent to the mig-slave device. When dirty data arrive, if we detect that the corresponding bits in the remote bitmap are already cleared, these data will be dropped. This optimization results in local performance for write operations on dirty data if they are aligned with the page size.

4.3 Wide-Area Network Redirection

In this section, we discuss the possible solutions to the addressing problem faced when migrating a virtual machine across different networks, especially in a Wide-Area Network (WAN). After the migration, the virtual machine may have to obtain a new IP address, which breaks existing network connections. Clients that are already connected as well as new clients need know the new IP address and domain name of the migrated virtual machine, and we would like to make this process transparent. As far as we know, there are basically two methods to do the network redirection: IP tunneling with Dynamic DNS, and IP Anycast.

- **IP tunneling with Dynamic DNS:** In [2], the authors propose this method for network redirection. During the runtime state migration, once the migrated virtual machine is suspended at the source, the redirection scheme is started. iproute2 is used to create an IP tunnel between the source and the destination. At this time, the virtual machine is not running, so we need to drop or queue all the packets to avoid connection resets. After the runtime state migration, the virtual machine is running at its new network location, so dynamic DNS entries are updated, which ensures that any new connection to this virtual machine is directed to the new IP address. All packets arriving at the old IP address are forwarded to the destination through the tunnel. [15] also uses a similar method to find the new network location of the virtual machine.

  With this mechanism, the virtual machine has two IP addresses after the migration. The old one is kept to forward packets from connections before the migration. When no connections use the virtual machine's old IP address, the tunnel can be destroyed.

- **IP Anycast:** Anycast is a network addressing and routing scheme whereby the data is routed to the "nearest" or "best" destination as viewed by the routing topology. It is widely deployed today to reduce latency and increase reliability. To support virtual machine migration over WAN using IP Anycast, the virtual machine needs to keep the same IP address before and after the migration. It is up to the network to decide where the active virtual machine is and how clients will locate it. In [5], the authors use IP Anycast to achieve the shortest latency of services after migration.

  After the new virtual machine is brought up and running on the destination host, we could shut down the source host. Then this node is not available and the network redirects all the data traffic to new location. However, it requires some time to detect the topological changes, such as 30 seconds for OSPF routing. There will be some performance loss for existing network connections before the network converges.

4.4 Loss Prevention

In current storage systems, users rely on periodic backup and remote replications to protect their data. Usually this involves a full backup of a large amount of data, such as the virtual machine disk, which will take a long time to complete and is not acceptable for most business environments. Thus, the snapshot feature is frequently used to preserve disk state. It allows users to save the disk at some specified time by using differential copy. To achieve this, we need to store the difference (delta) data of both the runtime state and the disk data for the given virtual machine. This operation must be a live process so it does not have any impact on the running virtual machine. In this section, we will describe the way SBD implements the snapshot and full backup functions.

Basically, there are two types of differential snapshot techniques [23] frequently used in modern data protection techniques: copy-on-write and redirect-on-write.

Copy-on-Write snapshot: When the user plans to create a snapshot, a data file is specified as the target snapshot volume. Upon the first write operation on a disk page, the original data (before the snapshot) will be read from the primary disk and written into the snapshot volume. After that, the write operation will be processed and the new data are written to the primary disk. For the following operations on this disk page, no copy operation is needed. New data will be written immediately, so only the first write of a disk page involves a copy operation. Thus, the disk state at the time when the snapshot is created is saved in the combination of the snapshot volume and the primary disk. Because the page is copied on a write operation, this technique is called Copy-on-Write snapshot [19].

The problem with Copy-on-Write is that for the first write request, three I/O operations are required: (1) read the original page data from the primary disk. (2) write the page data to the snapshot volume. (3) write new data into the primary disk. The first two extra I/O operations may decrease disk performance.

Redirect-on-Write snapshot: Using Redirect-on-Write, a data file is used as the target snapshot volume. For any write operation, the new data will be written into the snapshot, instead of the primary disk. This eliminates the extra I/O operations in the copy process of Copy-on-Write. Thus, once a snapshot is created, all the subsequent write operations are performed on the snapshot volume. To read a page from this disk, there must be some metadata to track its current location. It may be in the primary disk or one of the snapshot volumes, depending on whether this page was modified after the snapshot was created.

Redirect-on-Write has better write performance than Copy-on-Write, because it only involves one I/O operation for the first write. However, the current state of the disk is preserved in the primary disk and all the snapshots. It also involves some extra work for every read operation to locate the desired pages.

In our implementation, we use Copy-on-Write to do the snapshot. However, to improve the performance of the initial write, we provide a cache disk for all SBD devices. For the first write

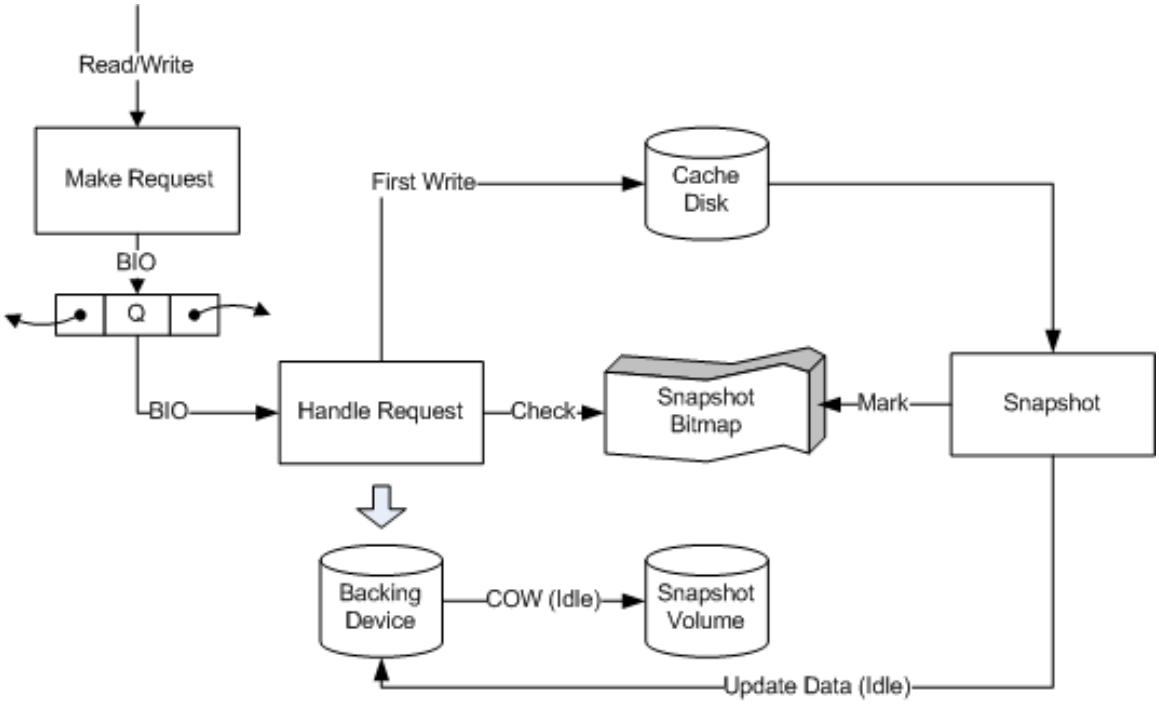

Figure 4.4: SBD Snapshot Structure

of a page after the snapshot, the data is first written into the cache disk with its metadata. Later, when the disk becomes idle, the original page data is copied from the primary disk to the snapshot volume, and data in the cache disk is synchronized to the primary disk. From the users' perspective, the first write request involves only one I/O operation on the cache disk, and this should have better performance than pure Copy-on-Write. A read operation should involve at most one check over the cache bitmap. Our snapshot implementation in SBD has the structure shown in Figure 4.4

Note that for a given VNODE, there will be only one cache disk for each SBD device. Thus, users could use a high-performance disk drive for the cache disk to get better performance. The snapshot function involves the following steps:

1. The user creates a snapshot. If this is the first one, a snapshot volume is allocated, and a SnapshotPath object is allocated in SBD and appended into the IDataPath list. For the following snapshots, it creates some metadata in the snapshot volume to record the current state.

2. For a read request, SnapshotPath checks the bitmap of the cache to see if this data page is synchronized to the primary disk. Any page in the cache disk would cause the request to be modified and the read operation to be redirected to get right data. Otherwise, this read request is completed as in a normal block device using the backing device. In the worst case, this read operation involves different parts of both the primary disk and the cache. We then split this request to read all of these data. Normally, this should not happen, because the file system uses aligned blocks to read and write disk.

3. When a write request is received, the request handler checks the snapshot bitmap to see if the original data is already saved in the snapshot volume. If so, this write request is completed directly on the primary disk. Otherwise, this request is modified to add some metadata and the new data are written into the cache disk. This request will be marked as completed, and from the users' view, this operation is finished.

4. There is a snapshot-sync thread running in the background. It scans the cache disk and for every page, it first triggers a copy-on-write process for this page between the backing device and the snapshot volume. Then the thread updates the data from cache disk to the backing device. This process is activated when the disk becomes idle or the cache disk is full. So in most situations, it should have little impact on normal IO operations.

During the snapshot process, there is a chance that the system might crash before a page is synchronized to the primary disk. Under this condition, when we restart the system, SBD needs to guarantee that all the write operations that receive responses from our device actually have their data consistent. Thus, SBD starts a recovery process. When a page is written into the cache disk, some metadata is appended to the original data, which allows us to save the exact runtime state when the system crashed. The SnapshotPath object scans the cache disk and collects all the metadata to generate the cache bitmap, and this cache bitmap is used for the subsequent I/O operations. SBD marks a page as SYNC in cache bitmap only after this page is synchronized to primary disk, which assures us that even if the system crashes, all the new data in the cache will be synchronized. The worst case is that some data in the last write operation might be synchronized for a second time after the recovery, but we can keep all the data consistent.

We also need to restore the system status from the snapshot metadata on system failure. The snapshot bitmap, which is stored in memory during runtime, needs to be rebuilt, so we can know which pages are already in the snapshot. This rebuild operation involves scanning the snapshot data and reading the metadata of each page. From these metadata, we can get the page id and set the corresponding bit in the bitmap. So this guarantees that even the system fails to shutdown normally, the snapshot data is still consistent.

With the above process, we can generate local snapshots for SBD devices. To test the system's ability to recover from a power failure, we shutdown the system in the middle of a snapshot, and then rebooted it. The system was able to recover its status from the cache disk and snapshot metadata and then continued to run properly.

Although users could make a snapshot with smaller cost and impact on the running virtual machine, a full backup is still needed, as a snapshot requires both the primary disk and the snapshot pools to represent the virtual machine state. SBD implements a BackupPath to do the full backup of disk data. It works similarly to the snapshot component. The only difference is that BackupPath object copies a data page from the primary disk to destination storage even if no write operation involves this page, whereas in snapshot, this page will only be copied for the first write.

Chapter 5

Performance Evaluation

Our prototype VMMS is based on the Xen virtualization platform in CentOS 5.4. We ran experiments between two identical Dell PowerEdge server-class machines, each with dual-core 2GHz AMD Opteron CPU and 2GB DDR RAM. These machines have Intel gigabit network interface cards (NICs) and are connected through switched gigabit Ethernet. tc is used to simulate a 45 Mbps link with 100ms round-trip delay. We checked the traffic shaping by measuring ping times of 100ms and a TCP data throughput of 45 Mbps.

A separate computer running Ubuntu 8.10 64-bit server provided the central management service. We allocated 512MB memory and 8GB disk for the VNODE virtual machine. We configured the migrated virtual machine to have 512 MB memory and 8GB local disk. To evaluate our system, we performed four experiments: 1) memory ballooning and compression, 2) SBD read and write performance, 3) runtime migration with shared disk, and 4) system migration with local disk. Each experiment was repeated five times and the final result reported is the average of the five values.

5.1 Memory Ballooning and Compression

We first evaluated the impact memory ballooning has on the compression ratio. For this experiment, we installed our memory ballooning application in the guest system. Then different workloads were used to access the memory:

- **No Work Load:** Power on the guest system and wait until the OS is ready. There is no service running inside the guest system.
- **Random Write (100MB):** A running program keeps writing random data to a 100MB memory block every 10 seconds.
- **Random Write (200MB):** The same as above, but with 200MB memory block.
- **FTP Server:** A normal FTP server with two concurrent clients uploading files. We use vsftp as the ftp server, and create our own client application which uploads a 100MB data file every 10 seconds.

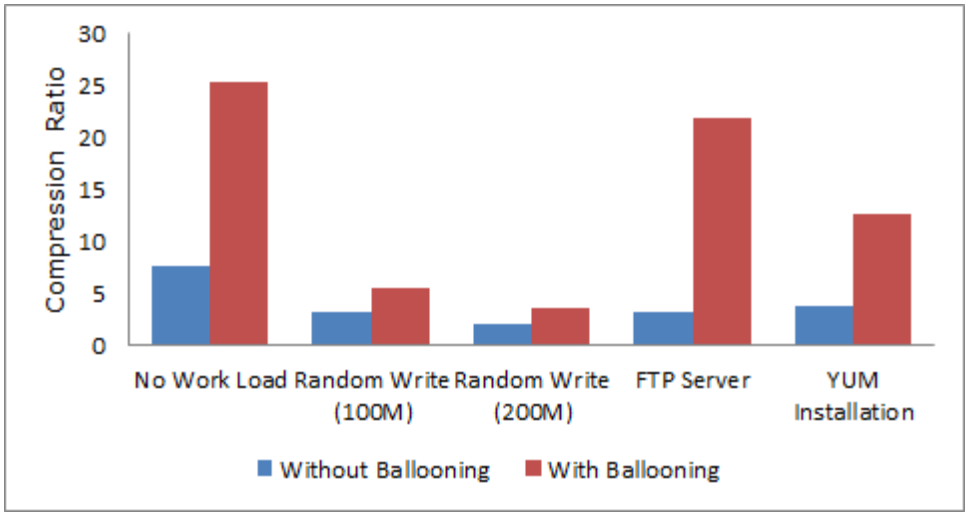

Figure 5.1: Compression Ratio with and without Ballooning

- **YUM Installation:** Execution of YUM to install packages with 320MB data. This process involves the installation of X Window System and Eclipse development environment.

For each workload, we ran it for one minute and made a checkpoint of the memory data. Then we restored the virtual machine and executed the memory ballooning application. After the ballooning, another memory checkpoint was made. We used our compression program to process these two checkpoint files and then we compared the compressed memory size with and without ballooning.

The compression ratio is calculated as $CompressionRatio = \frac{OriginalSize}{CompressedSize}$. A higher compression ratio leads to a smaller compressed size. As shown in Figure 5.1, ballooning results in a much higher compression ratio.

5.2 SBD Performance

We then evaluated the impact that migration has on the SBD read and write operations. We focused on the throughput after migration and compared it with local disk performance. The system structure used in this experiment is shown in Figure 5.2. Two servers running SBD inside were connected with a 45 Mbps link. This link was used by SBD to do the synchronization. We also added another 1 Gbps link to create an iSCSI target. We used our own benchmark tool which ran on top of the iSCSI disk in this experiment. A random scheduler was used to decide when to read or write a block of the disk. For each operation, the benchmark tool accessed the disk sequentially. With the above structure, the benchmark tool talked with the disks via the 1Gbps network connection, but the disk was migrated over the 45 Mbps link. First, we ran the benchmark tool for 10 seconds, which generated several data blocks. Then we paused this tool, migrated the disk, and rebuilt the iSCSI target. After this, the benchmark tool was running on the destination side, and we resumed its running with read/write mode to test the performance after migration.

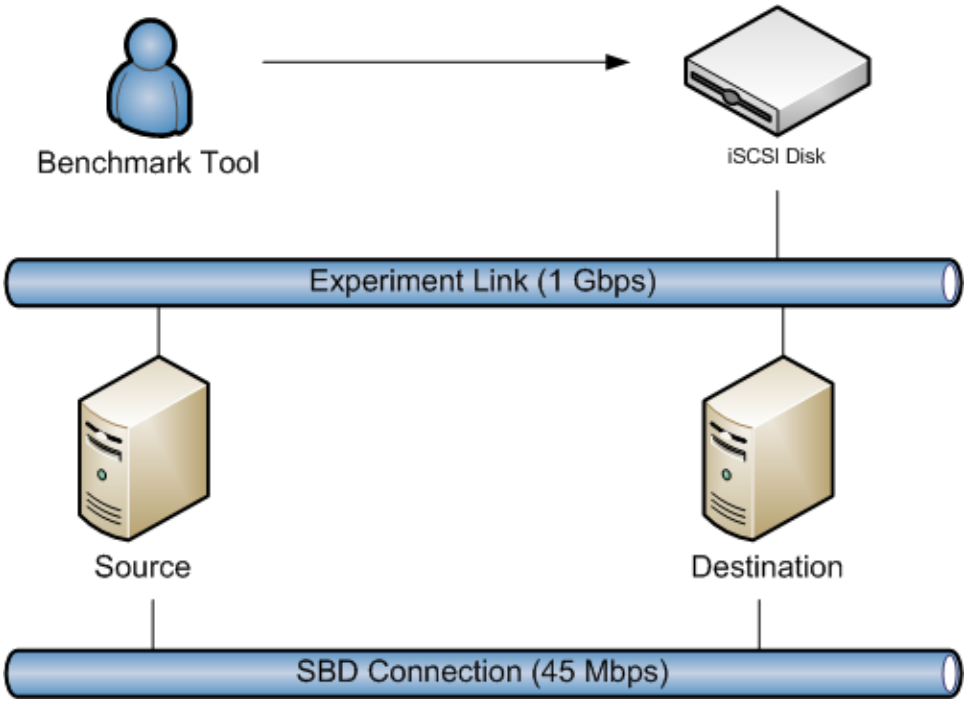

Figure 5.2: Experiment Configuration for SBD

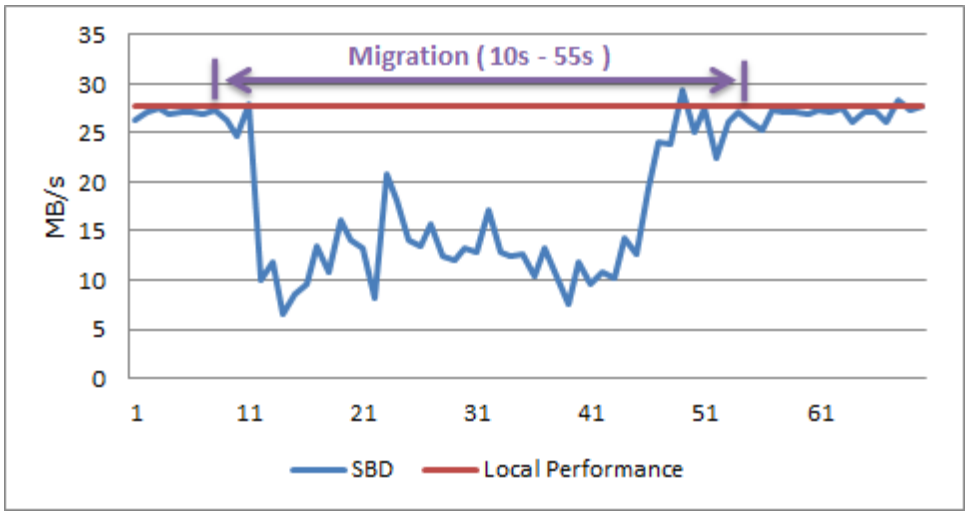

Figure 5.3: SBD Write Performance

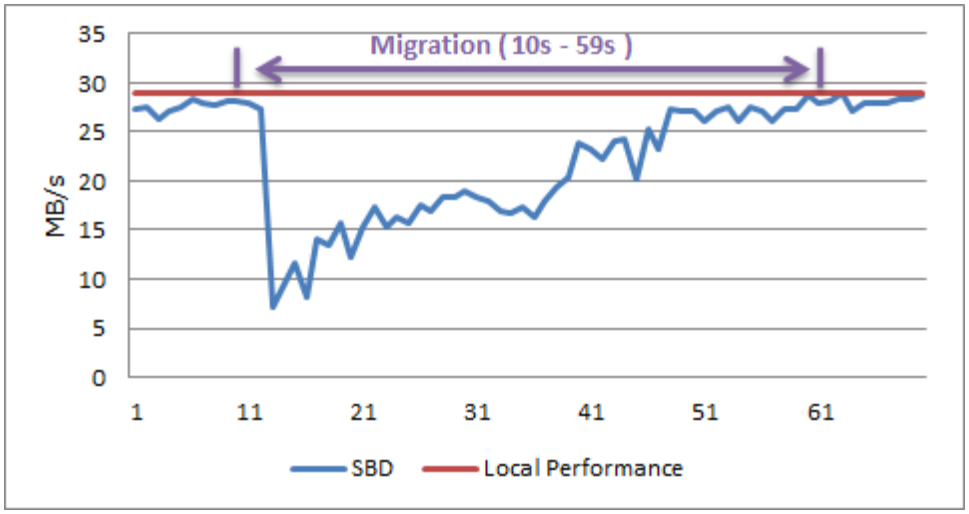

Figure 5.4: SBD Read Performance

Figure 5.3 shows the write performance of disk before and after migration. At the time we migrated the SBD, the write performance dropped from 28 MB/s to 10 MB/s. After the migration, SBD still has some dirty data unsynchronized. However, as we mentioned before, for write requests that are aligned, SBD updates the local disk and clears the bitmap, so some write operations before the synchronization was finished receive 21 MB/s. When the synchronization finished at 55s, SBD went back to local performance. We also present the read performance in Figure 5.4. The migration completes at 59s at which time the performance of SBD went back to normal local rates.

From these two figures, the worst case during migration is to read or write completely dirty blocks, which requires a PULL operation from the source through the network. In this case, the read/write rate is the same as the available network bandwidth (in our experiment, this was 45Mbps). In practice, most systems access the block device through a memory buffer, so we expect better performance for real applications.

5.3 Runtime Migration

To examine the performance of VMMS for runtime state migration, we migrated virtual machines with the following workloads :

- **No Work Load:** Power on the guest system and wait until the OS is ready. There is no service running inside the guest system.
- **Video Stream Server:** Guest OS runs a VLC streaming server hosting a 45MB video. We also have two video clients connecting to this server and playing the video.
- **FTP Server:** A normal FTP server with two concurrent clients uploading files. We use the same clients described before.

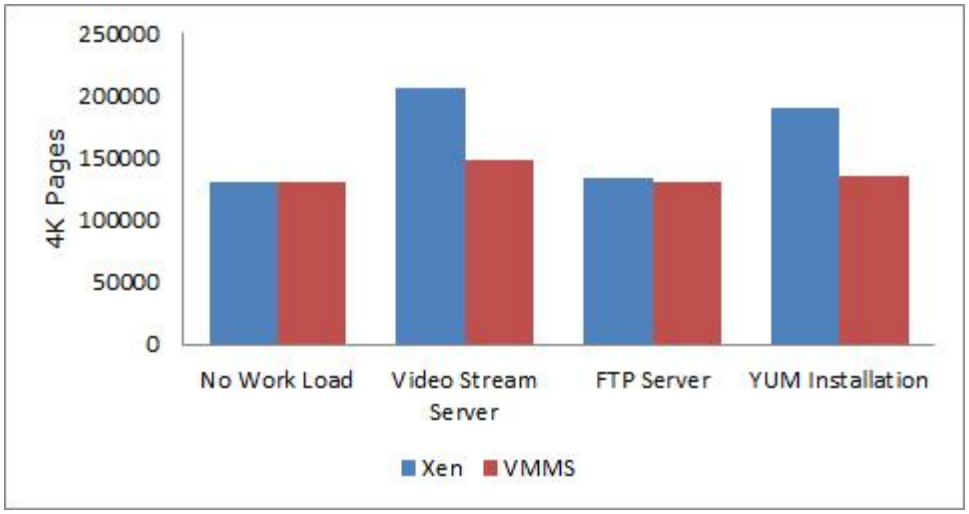

Figure 5.5: 4K Pages Transferred During the Migration

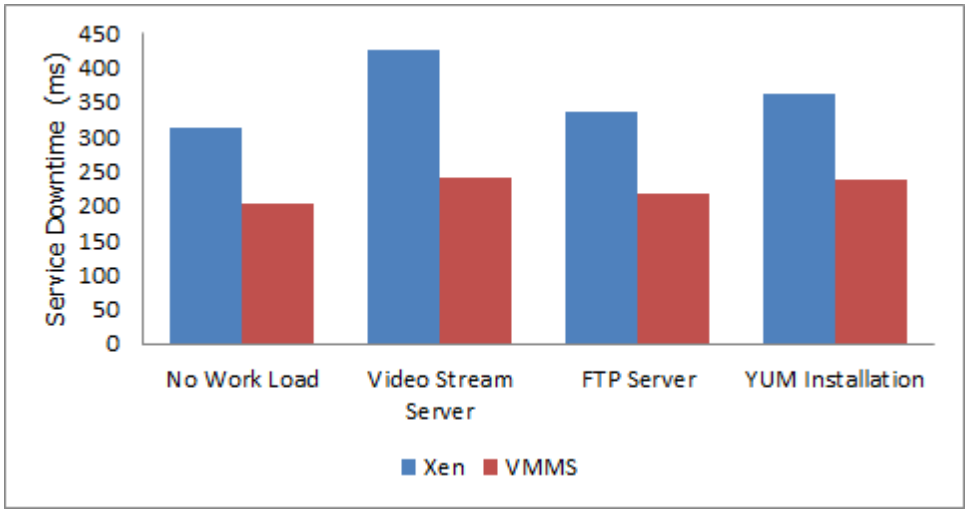

Figure 5.6: Service Down Time with Shared Device

- **YUM Installation:** Execution of YUM to install packages with 320MB data. We install the same packages described before.

Like the Xen built-in migration program, this experiment requires a shared device used by both source and destination host to provide the file system for the migrated virtual machine. We created an 8 GB iSCSI target on a separate machine and connected both hosts to this target via the 1Gbps network. The disk was not migrated in this experiment. We focus on the improvement of runtime state migration by using VMMS.

We compare our system to the Xen built-in migration program. Figure 5.5 shows that VMMS decreases the number of memory pages that need to be transferred during the migration. Figure 5.6 shows the service downtime during the migration of runtime state. For all four workloads, VMMS

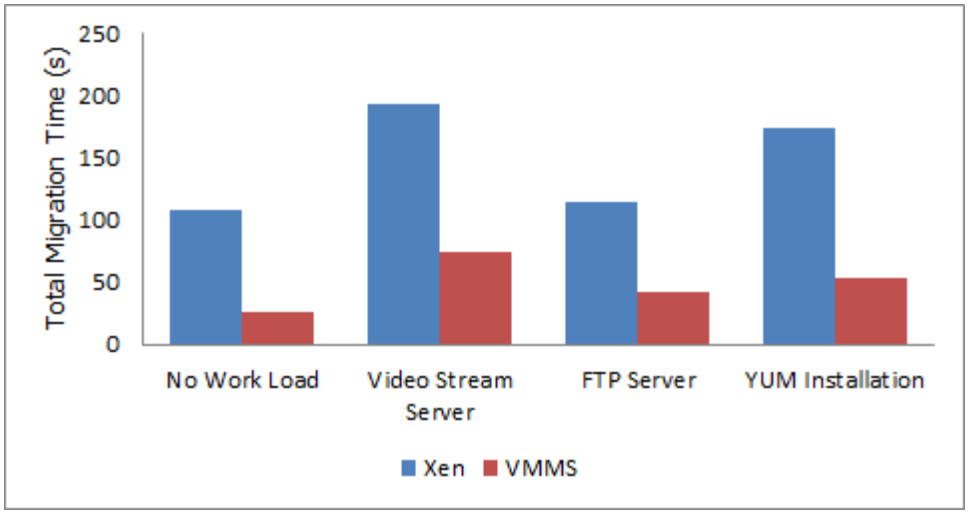

Figure 5.7: Total Migration Time with Shared Storage

reduces downtime by over 50%. Figure 5.7 shows that the total migration time is also greatly decreased. In Figure 5.8, we compare the actual experiment results with the numerical analysis in chapter 3. It is clear the experiment results are consistent with our migration model. The differences between these two results are primarily due to the assumptions we made.

This experiment proves that VMMS can successfully live migrate a virtual machine with practical workloads over a 45Mbps link, and that VMMS has better performance over Xen built-in migration.

5.4 System Migration

Finally, we consider the performance of VMMS to migrate the whole virtual machine, including both the runtime state and its persistent state over a 45Mbps link. The same workloads as the previous experiment were used. None of the existing live migration systems deal with disk data, so we only compare the results of VMMS to itself with and without disk migration.

Figure 5.9 illustrates that even with disk migration, our system achieves a very short service downtime. It only leads to an extra 20ms, due to less bandwidth being available during the runtime state transfer. The test results in Figure 5.10 show that, compared with the previous experiment, more time is required to complete the system migration process. The main reason is that after the runtime state migration, SBD still has some disk pages that are not synchronized. For FTP server and YUM Installation, SBD needs more time to finish the synchronization, as these two workloads involve more write operations during the migration. From Figure 5.11, we can see that the experiment results are consistent with the numerical analysis in chapter 3.

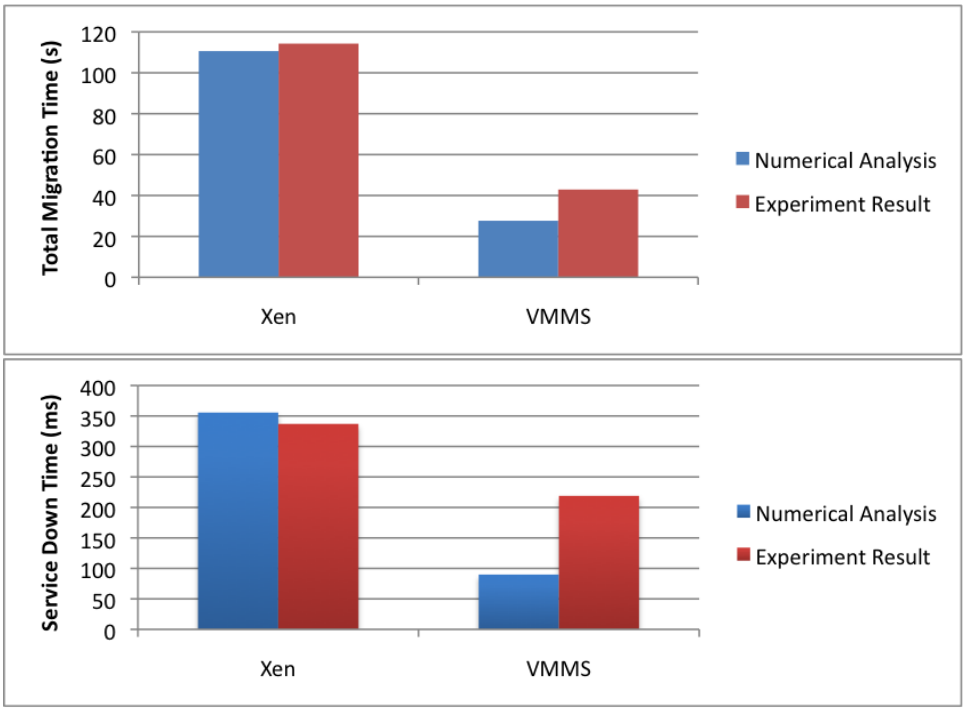

Figure 5.8: Runtime State Migration with FTP Workload

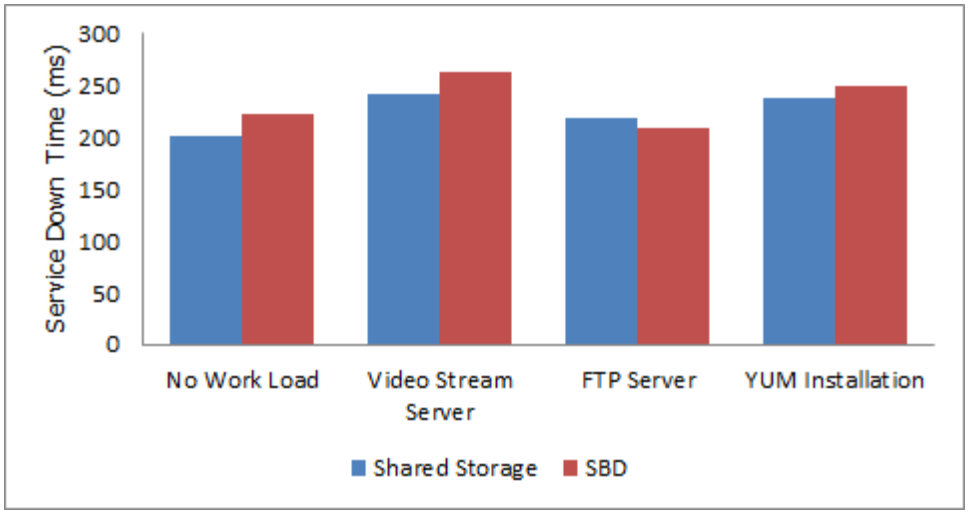

Figure 5.9: System Migration Down Time

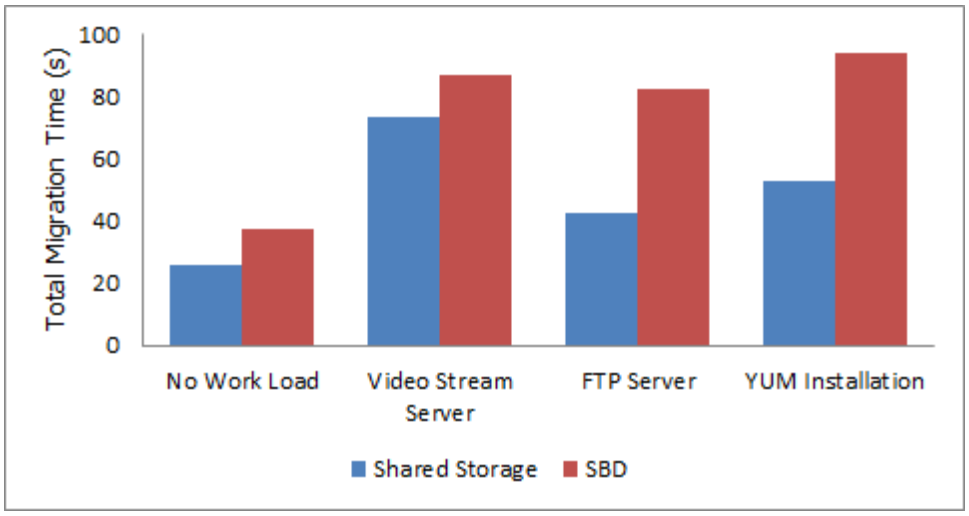

Figure 5.10: System Total Migration Time

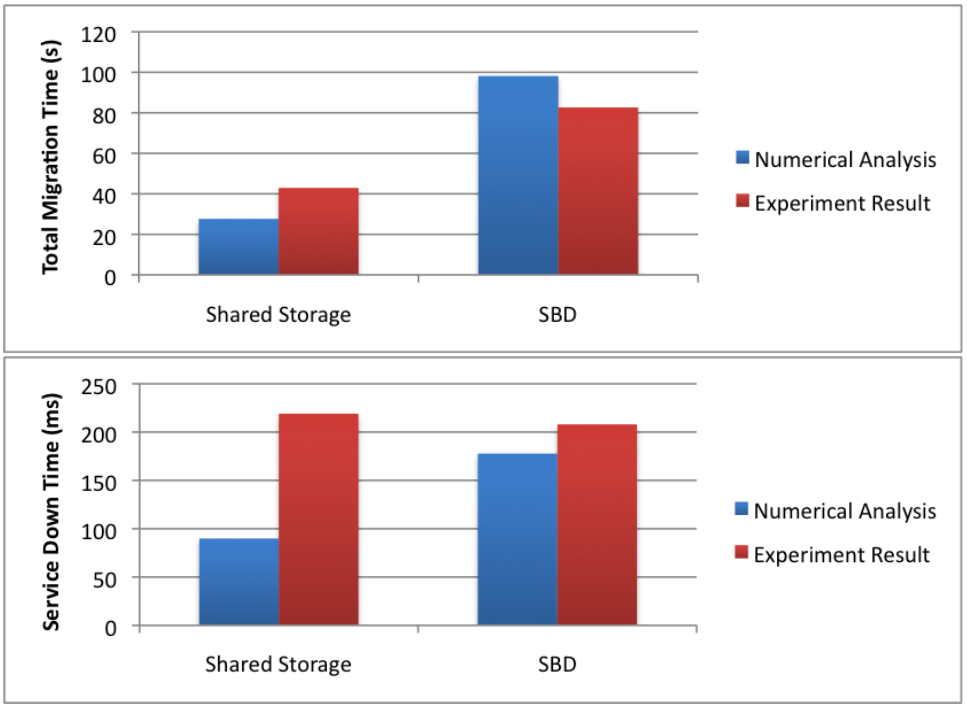

Figure 5.11: Runtime State Migration with FTP Workload

Chapter 6

Conclusions and Future Work

In this thesis, we have presented the design, implementation and evaluation of VMMS, a high performance live migration system. It handles both runtime and persistent state during migration, which allows users to transfer the whole virtual machine to the target host. VMMS is a combination of two major sub-systems: the Central Management Service and VNODE for each physical host. The implementation of VNODE takes the migrated virtual machine and also the administration domain as a black-box. Every request is completed using an abstract layer, which helps us to implement our system for different hypervisors. Currently, we have implemented VMMS on the Xen platform.

We applied memory ballooning and compression to improve the migration of runtime state. Unused memory of the migrated virtual machine is cleared and dropped, saving more than 50% of the total migration and service downtime. This makes it possible to migrate the virtual machine over low-bandwidth networks. We also implemented a block device driver, SBD, to support the migration of persistent state. Based on the block-bitmap, our system implements a hybrid push-pull mechanism to synchronize the disk data. With fast role-transition support, VMMS can migrate the whole virtual machine and minimize the total migration and service downtime. Experimental results show that the virtual machine can achieve local performance before and after the migration.

The most important benefit from VMMS is that we can now migrate a virtual machine, including both its runtime state and its persistent state, over long distance, while the service downtime is minimized. VMMS ensures local system performance before and after migration. SBD also allows administrators to create remote backups of disk data in case of physical disaster. With the built-in scheduler, this synchronization has little performance impact on the virtual machine. Unlike other implementations, our system is not bound to any paticular platform. The abstract layer allows us to support a variety of hypervisors, and users can manage several physical hosts in one place, without knowing their underlying systems.

In the future, we will test our system with different hypervisors, such as ESX Server [21], KVM [10], etc. We also plan to improve the compression algorithm used during the runtime state migration. In [9], the authors present a novel approach to optimize live virtual machine migration using a zero-aware *characteristics-based compression* (CBC) algorithm. The overhead of CBC memory

compression is quite small and they invoke the adaptive compression mode based on the rate at which memory is dirtied. To improve the migration process of SBD, we would like to do more experiments with different scheduling methods. If the synchronization thread can decide the optimal sequence in which to transfer the disk pages, the total migration time could be further decreased and disk performance after the transition could be improved. Finally, we would like to extend our system to migrate a whole virtualization cluster.

Bibliography

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[2] R. Bradford, E. Kotsovinos, A. Feldmann, and H. Schiöberg. Live wide-area migration of virtual machines including local persistent state. In *VEE '07: Proceedings of the 3rd International Conference on Virtual Execution Environments*, pages 169–179, New York, NY, USA, 2007. ACM.

[3] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation*, pages 273–286, Berkeley, CA, USA, 2005. USENIX Association.

[4] R.J. Figueiredo, P. A. Dinda, and J.B. Fortes. A case for grid computing on virtual machines. In *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems*, page 550, Washington, DC, USA, 2003. IEEE Computer Society.

[5] K. Han. Anycast network solutions for DataGardens' Syntropy implementation and their performance investigations. Technical report, Edmonton, AB, Canada, 2009.

[6] Jacob Gorm Hansen and Asger Kahl Henriksen. Nomadic operating systems, 2002.

[7] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. The performance of microkernel-based systems. In *SOSP '97: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 66–77, New York, NY, USA, 1997. ACM.

[8] M. R. Hines and K. Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 51–60, New York, NY, USA, 2009. ACM.

[9] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan. Live virtual machine migration with adaptive memory compression. In *CLUSTER*, pages 1–10. IEEE, 2009.

[10] A. Kivity. kvm: the linux virtual machine monitor. In *OLS '07: The 2007 Ottawa Linux Symposium*, pages 225–230, July 2007.

[11] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *WMCSA '02: Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, page 40. IEEE Computer Society, 2002.

[12] I. Krsul, A. Ganguly, J. Zhang, J. A. B. Fortes, and R. J. Figueiredo. Vmplants: Providing and managing virtual machine execution environments for grid computing. In *SC '04: Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, page 7, Washington, DC, USA, 2004. IEEE Computer Society.

[13] Y. Luo. A three-phase algorithm for whole-system live migration of virtual machines. In *CHINA HPC '07: Proceedings of the 2007 Asian Technology Information Program's (ATIP's) 3rd Workshop on High Performance Computing in China*, pages 172–173, New York, NY, USA, 2007. ACM.

[14] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: a system for migrating computing environments. *SIGOPS Oper. Syst. Rev.*, 36(SI):361–376, 2002.

[15] K. K. Ramakrishnan, P. Shenoy, and J. Van der Merwe. Live data center migration across WANs: a robust cooperative context aware approach. In *INM '07: Proceedings of the 2007 SIGCOMM Workshop on Internet Network Management*, pages 262–267, New York, NY, USA, 2007. ACM.

[16] P. Ruth, X. Jiang, D. Xu, and S. Goasguen. Virtual distributed environments in a shared infrastructure. *Computer*, 38(5):63–69, 2005.

[17] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. *SIGOPS Oper. Syst. Rev.*, 36(SI):377–390, 2002.

[18] Y. Sheng. Dynamic network resource allocation. Master's thesis, University of Alberta, Edmonton, AB, Canada, 2010.

[19] M. H. Sun and D. Blough. Fast, lightweight virtual machine checkpointing. Technical report, Atlanta, GA, USA, 2010.

[20] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable remote execution facilities for the V-system. Technical report, Stanford, CA, USA, 1985.

[21] vSphere Hypervisor. http://www.vmware.com/products/vsphere-hypervisor/.

[22] T. A. Welch. A technique for high-performance data compression. *Computer*, 17(6):8–19, 1984.

[23] W. Xiao, Y. Liu, and Q. Yang. Implementation and performance evaluation of two snapshot methods on iSCSI target storages. In *Proceedings of NASA/IEEE 14th Conf. on Mass Storage Systems and Technologies*, May 2006.

[24] Y. Zhang. Networked storage system (Open-iSCSI) traffic control in virtual machines and its performance enhancement. Master's thesis, University of Alberta, Edmonton, AB, Canada, 2010.