國立交通大學

資訊科學與工程研究所

碩士論文

廣域網路中的虛擬機器傳送

Teleporting Virtual Machine over Wide-Area Network

研究生：賴敘方

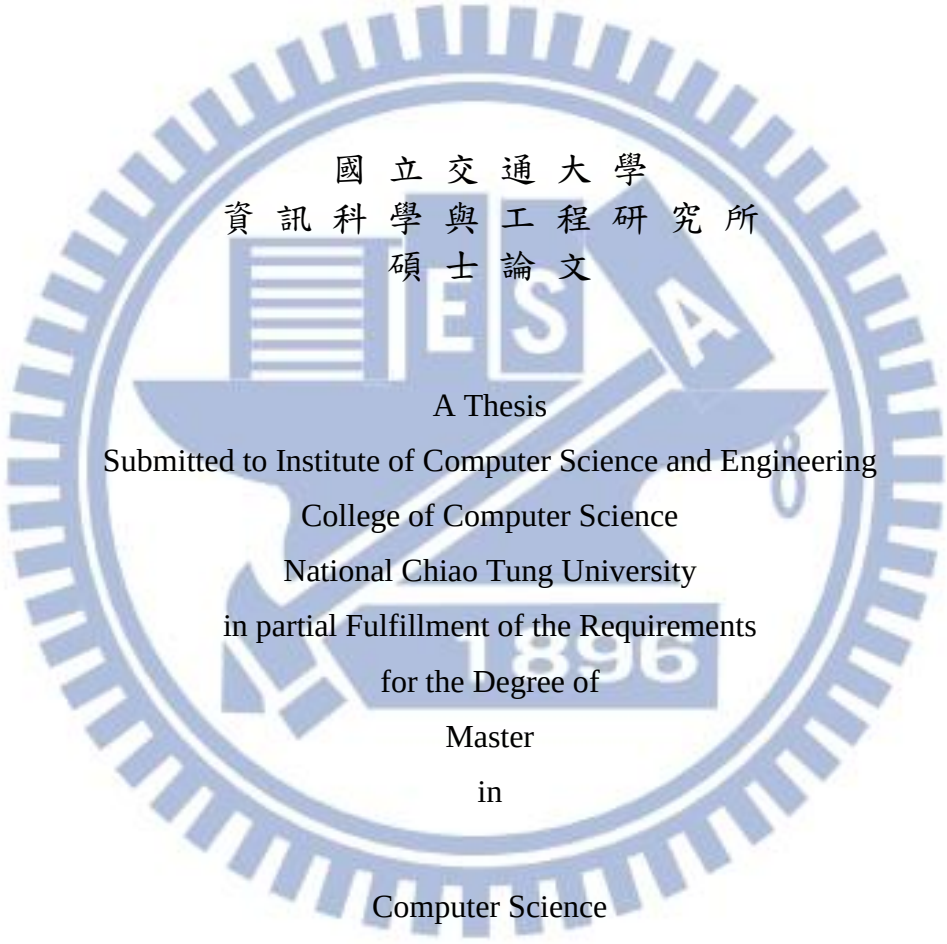
指導教授：吳育松　教授

中華民國　101　年　10　月

廣域網路中的虛擬機器傳送
Teleporting Virtual Machine over Wide-Area Network

研 究 生：賴敘方　　Student：Hsu-Fang Lai

指導教授：吳育松　　Advisor：Yu-Sung Wu



國 立 交 通 大 學

資 訊 科 學 與 工 程 研 究 所

碩 士 論 文

A Thesis

Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

June 2012

Hsinchu, Taiwan, Republic of China

中華民國 101 年 10 月

廣域網路中的虛擬機器傳送

學生：賴敘方　　指導教授：吳育松

國立交通大學資訊科學與工程研究所碩士班

摘　　要

虛擬機器搬移為基礎設施即服務(IaaS)雲端運算環境中的重要功能之一，傳統的虛擬機器搬移專注在宿主機器(Host machine)間轉移記憶體及處理器之狀態，虛擬機器的硬碟即必需要在搬移時讓來源與目地宿主機器都能夠存取。如此一來傳統虛擬機器搬移的範圍即被限制於區域網路(LAN)中，因為跨越廣域網路(WAN)地存取存儲裝置的效能並不理想。然而有越來越多的基礎設施即服務雲被建構在全球各地，虛擬機器搬移將很快地需要被使用在這些距離太遠只能藉由廣域網路連結的宿主機器上。我們提出一個有效率的跨廣域網路虛擬機器搬移系統，其關鍵技術在於一個運用預先整理好的索引來重組存儲裝置的機制。我們的實驗結果顯示此機制不只減少在廣域網路中的資料傳輸量同時也縮短了搬移所花費的時間。我的實驗結果指出平均減少66%的資料傳輸量以及節省 59%的搬移時間。大大地增進搬移的效能。

關鍵字： 線上搬移、儲存資料重複刪除、廣域網路

Teleporting Virtual Machine over Wide-Area Network

Student: Hsu-Fang Lai Advisor: Dr. Yu-Sung Wu

Institutes of Computer Science and Engineering
National Chiao Tung University

Abstract

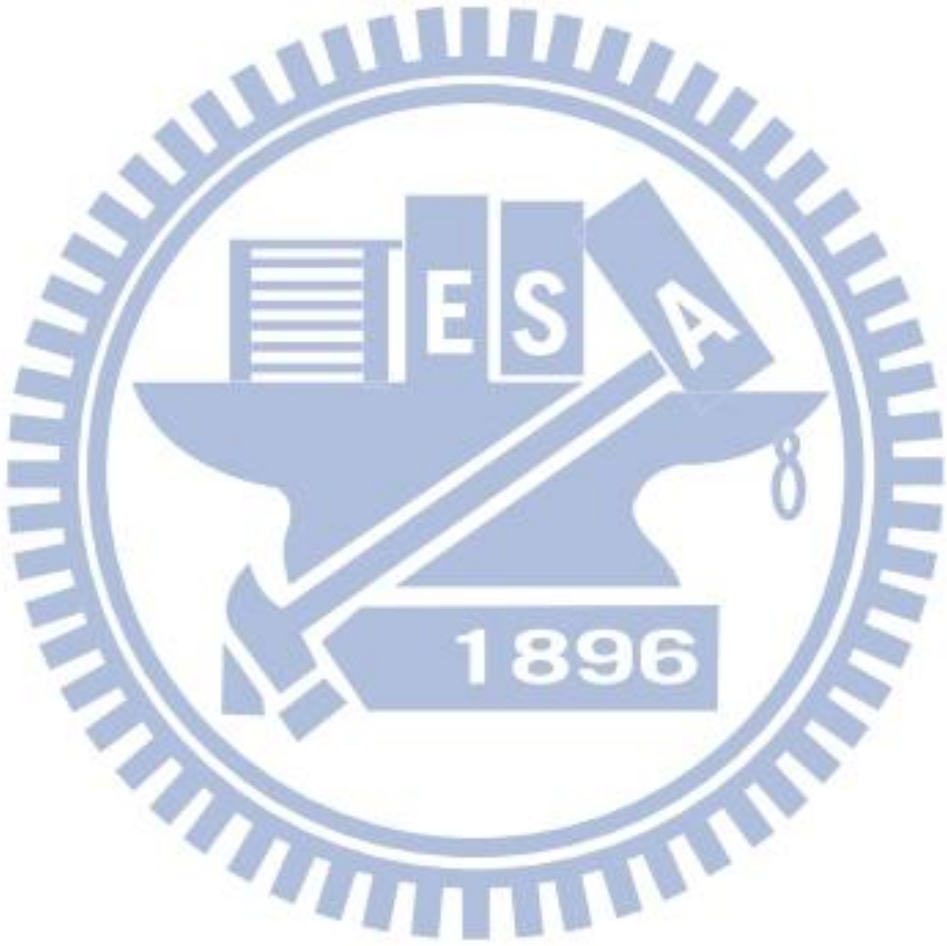
Virtual machine (VM) migration is one of the key features of infrastructure-as-a-service (IaaS) cloud computing. Conventional VM migration focuses on transferring a VM's memory and CPU states across host machines. The VM's disk image has to be accessible to both the source and destination host machines during the migration. Therefore, conventional virtual machine migration is limited to host machines that reside on the local area network (LAN) since sharing storage across wide-area network (WAN) is very inefficient. However, as more IaaS clouds are being constructed around the globe, VM migration will soon be needed for host machines that are far apart and can only be reached from each other over the wide-area network. We propose a system to allow efficient VM migration over WAN. The key technique is a mechanism to rebuild VM storage by using pre-calculated indexes. This mechanism not only reducing amount of data transferring over WAN but also decrease total migration time. Our experiment result indicates that about average 66% of data is reduced on transferring and 59% time is saved during migration. Improve the migration performance greatly.

Keywords: Live migration, Storage de-duplication, Wide-area network

Contents

List of Figures

List of Tables



Chapter 1 Introduction

Virtualization is now popularly employed in cloud datacenter to allow multiple virtual machines (VMs) to run on a single host machine in the hope of more efficient and cost-effective resource utilization. Virtualization also allows the migration of virtual machines across host machines, which makes the utilization of system resource more dynamic. For instance, we can evenly distribute VMs with heavy workloads across host machines for load balancing. Conversely, we can aggregate idle VMs together onto a few host machines and put other host machines into sleep mode for power saving. Also if there is an imminent hardware failure on a host machine, we can move the VMs running on it to another host machine to prevent a disastrous system crash from happening.

VM Migration in Cloud Computing Environment

Conventional virtual machine migration transfers the memory and CPU states of a VM from a source host machine to a destination host machine. The storage of the VM is usually exported by a storage server and attached to both host machines so the storage can be accessed from the virtual machine no matter which host machine the VM is running on. The VM only needs be suspended for a brief moment in the migration process to hold up the generation of dirty memory pages when the generation rate exceeds a given threshold.

Conventional virtual machine migration requires a shared storage. This is not an issue if the migration only takes place within a local-area network (LAN) environment, as sharing storage in local area network environment can be

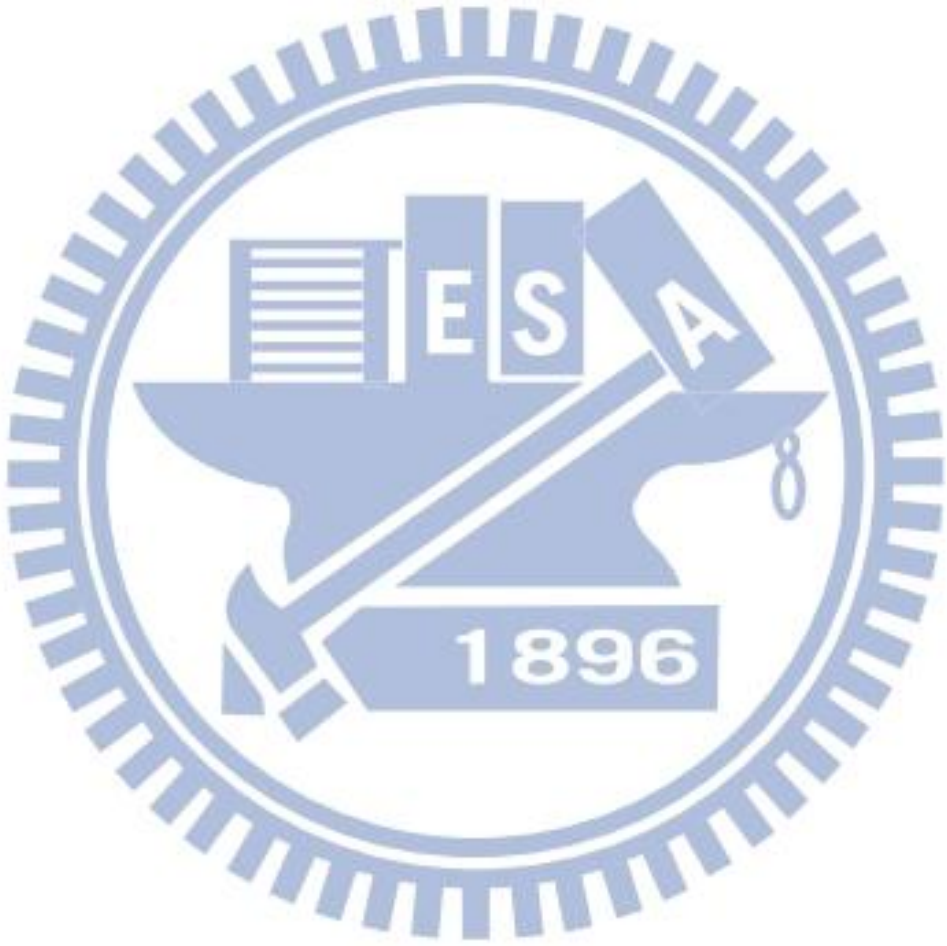
implemented by well-established NAS or SAN solutions. However, if we want to do VM migration in the wide-area network environment, there is unfortunately no well-established solution for sharing storage in the WAN environment. In fact, sharing storage in the WAN environment does not make much sense due to the limited bandwidth, long transmission latency, and the unreliable nature of WAN. Thus for VM migration in the WAN environment, the conventional approach of copying memory and CPU states have to be extended to copy the storage across storage servers on the WAN as well. However, the size of storage is usually much larger than the size of memory and the size of CPU registers. If we are to use the same way of migrating memory and CPU states to migrate the storage, we have to take a lot of time and transfer a lot of data via WAN. This is very inefficient.

We anticipate the need for VM migration over WAN in foreseeable future due to the fact there are many existing IaaS (Infrastructure as a Service) cloud service providers, and each provider may operate multiple datacenters around the globe. For the purpose of load balancing and disaster recover, it will be useful to be able to relocate VMs across datacenters. As datacenters are spread around the globe, the migration of VMs will have to carry out in the WAN environment.

In this work, we propose a novel approach to VM migration in WAN environment by indexing the VM storage content and exploring the similarity across VM storages to greatly reduce the time and network transmission required for migrating the storage of a VM across WAN. A prototype system is built for the Xen virtualization platform on Linux. The experiment results show that the total migration time is reduced by 59% on average, and the data transmission rate is reduced by 66% on average.

The rest of the work is organized as follows. Chapter 2 gives a brief

introduction of conventional virtual machine migration and a survey of related work. Chapter 3 gives the design of the proposed VM migration system for WAN environment. Chapter 4 describes some of the key implementation details. Chapter 5 presents the experiment results and discussions. Finally, Chapter 6 concludes this work with some discussion on potential future work.



Chapter 2 Background

Migration is a technique commonly employed in distributed system such as clusters for load balancing. Before platform virtualization becomes popular, migration is mostly performed at the level of process. We can move a running process from a busy machine to another idle machine without resetting the execution state [1]. Nowadays migration can be performed at the platform level with the help of virtualization, which moves virtual machine from one host machine to another. When virtual machine runs on a hypervisor, it needs a non-volatile storage space (such as a physical or logical block device or just an image file) to serve as its virtual hard disk and memory space to serve as its virtual memory. Most hypervisor assumes the storage is shared between host machines on the same local area network (LAN) when performing migration. Thus the migration of a virtual machine only needs to transfer CPU and memory states of the VM to the destination host machine and then continue the VM's execution there. However, if we have to do migration over wide-area network (WAN), where a shared storage is not available, then the migration of VM will also have to move the storage states from the source storage to the destination storage.

2.1 Virtual Machine Migration

There are two key tasks in a VM migration. The first is to transfer the VM states including CPU, memory, and possibly disk states. The second is to transfer execution to the new VM on the destination host machine. Both the original VM

and the new VM must be temporarily paused while transferring execution. The time during which the VMs are paused is called the downtime, as the service running the VM will be unavailable. Depending on which of the two tasks goes first, there are two categories of VM migration mechanisms.

Pre-copy mechanism transfers VM states first and will transfer the execution at the last moment (i.e. when most the VM states have been copied over to the destination host machine). Since the virtual machine is still running on the source host machine during state transfer, some of the states may have changed and have to be re-transferred. This actual situation may depend on the loading of the virtual machine and transfer speed.

Post-copy mechanism transfers execution first and then transfers VM states. In this case, the new VM on the host machine begins execution almost right after the migration takes place (the source VM is paused simultaneously). As a result, the new VM may not have all the memory and disk states synchronized as the source VM yet. Some of the states will have to be transferred immediately on demand as required for the execution.

Both migration mechanisms have their pros and cons. Pre-copy can be rolled back any moment before transferring execution, but it costs more transferring bandwidth and has possible longer downtime. Post-copy can minimize downtime, but it can't be rolled back and the VM may run slowly until all the states have been copied from the source. Most hypervisors use pre-copy as their default mechanisms for VM migration [2][3][4].

Storage migration moves the storage of a VM from one storage server to another storage server. It also consists of two tasks similar to VM migration. During the transfer of execution, storage migration must coordinate with the

host machine on which the VM is running to switch the storage. Similarly, depending on whether the transfer of storage states occurs before or after the transfer of execution, there is the distinction of pre-copy and post-copy storage migration mechanisms.

The discussion of the VM migration mechanism above is centered on synchronizing the VM states between the source host machine and the destination host machine. In practice, VM migration may also include the migration of existing network connections. Solutions such as mobile IP [5] and network virtualization techniques [6] can be used to deal with this. Our focus here is to accelerate the speed of VM migration over wide-area-network.

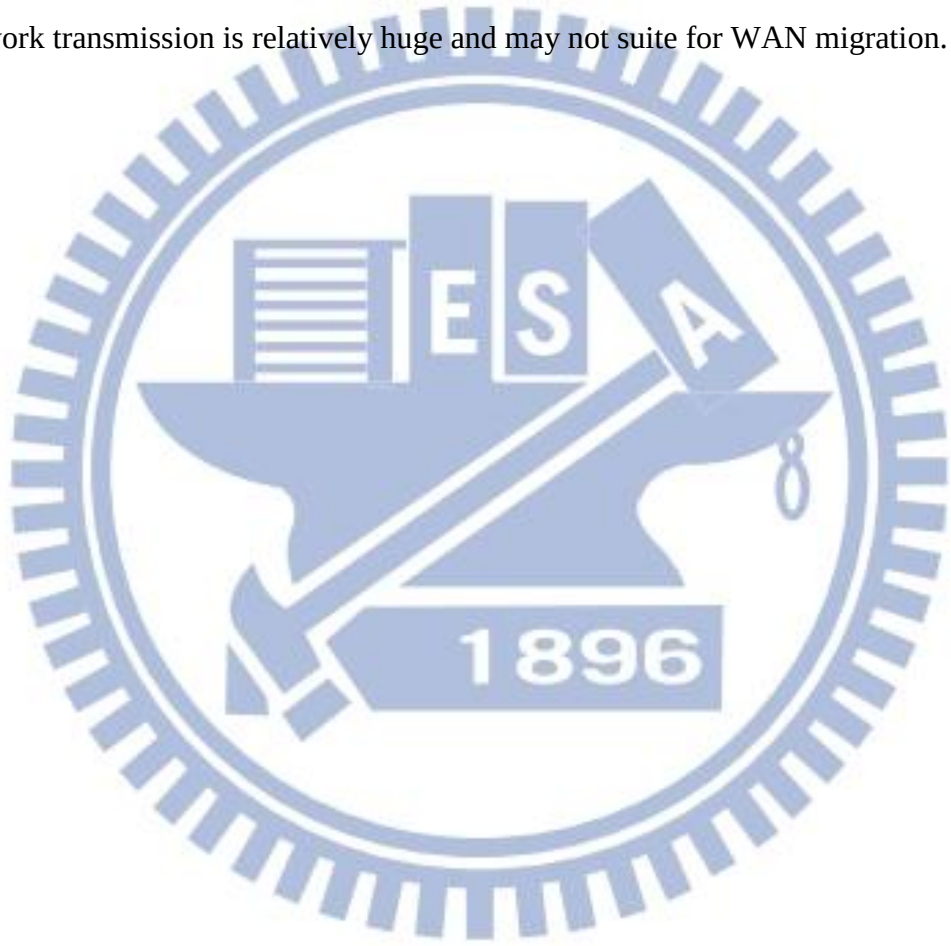
2.2 Related Work

Since migration is important in virtualization technology and the limitation of the conventional virtual machine migration, there are many results working on it. In the conventional memory migration, some works use compression [7] or de-duplication [8] to optimize data transmission and reduce migration time. Other work shortens the downtime by changing the data moving order [9].

There are also works on storage migration. The work that is most related to ours is K. Haselhorst et al. [10]. They use DRBD to synchronize the storage data. DRBD is a Linux kernel module which can do storage mirroring via network. The work [11] by T. Hirofuchi et al. uses NBD building a post-copy storage migration called xNBD. They minimize the downtime but the VM would have performance degradation time after the storage is switched. S. Akoush et al. propose another storage migration way by synchronizing storage in advance of migration [12].

Although it has good performance on synchronization and storage switching, it cannot do migration on demand.

T. Wood et al. combine the storage migration and virtual network technique making a total solution of WAN migration [13]. But they still use DRBD to migrate storage. In enterprise, VMware provide a storage migration solution called storage vMotion [14]. It uses the conventional way to migrate storage so the amount of network transmission is relatively huge and may not suite for WAN migration.



Chapter 3 Teleportation of Virtual Machine over Wide-Area Network

Migrating a VM over wide-area network requires not only moving the memory states of the VM but also the data on its disk storage. This is because accessing the disk storage over wide-area network will serverly impair the performance of VM due to the limited network bandwidth and non-negligible network transmission latency. The size of the disk storage is typically in the range of serveral giga-bytes or more, which makes moving the data over wide-area network a very challenging task. However, as cloud computing advocates the aggregation of computing resource towards the cloud, it is typical for some machines in the cloud to have similar applicatoins running on them. The disk storages of these virtual machines are like to have a significant amount of data in common. By a process that quikcly identifies the portion of data that are in common and efficiently reassembles the original storage from pieces of data from available sources, we develop a system that can support the migration of VM over wide-area network.

In this chapter, we will present a system that is designed to assist disk storage migration by using indexing information. We first give the overview in section 3.1 and later the details of the proposed system in section 3.2~3.4.

3.1 Virtual Machine Migration and Wide-Area-Network

Conventional virtual machine migration consists of copying the memory and CPU states of a VM from a source host machine to a destination host machine. The disk storage of the VM has to be shared by both hosts (i.e. attaching to the same storage area network). The migration begins by copying the memory pages of the VM to the destination host machine, while the VM is kept running on the source host machine. The running VM may modify some of the memory pages that have been copied to the destination. These pages will be marked dirty and need to be copied again. The process of copying dirty pages will continue till the dirty page yield rate hit a threshold, which typically set to the network transfer rate, is reached. At that time, the virtual machine will be paused and the remaining dirty pages will be copied to the destination in a single last round. Finally, the VM would be resumed execution on destination host machine. VM migration is restricted in a LAN environment, where the storage can still be accessed from the destination host machine over the LAN and do not require migration. However, storage migration is necessary when moving a VM over wide-area network. To build a storage migration system, we need to overcome three problems. First we have to detect which data blocks are changed after last time we handled since the VM would probably write some data to storage during migration process. Second, we cannot detach the storage from a running VM so we must have another way to switch the storage from source storage to destination storage. Third, the amount of data is often huge in storage. It's not only waste time but also waste bandwidth to transfer the data directly. Thus we need a mechanism to help our system optimizing for this problem.

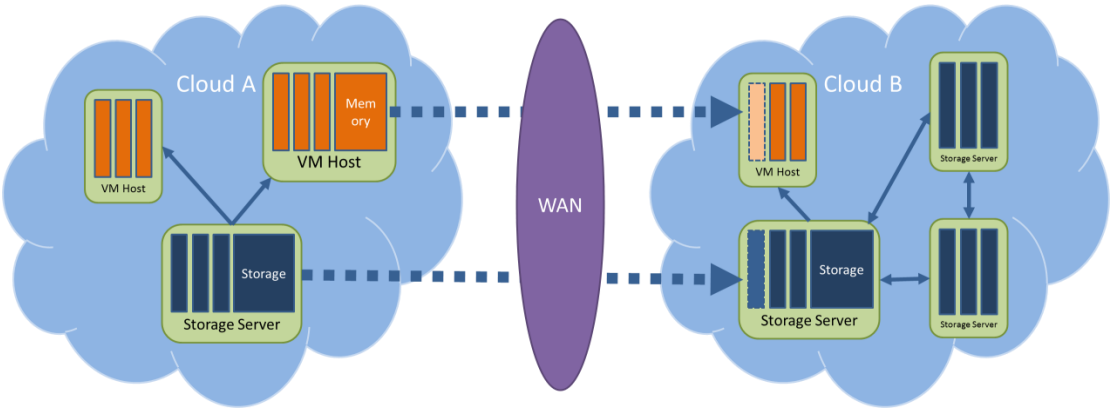


Figure 1: Overview of migration over WAN

Figure 1 shows the overview of our system. We build three key components dealing with the three problems to make the storage can be migrated from one storage server to another storage server via WAN. First we add the bitmap mechanism on storage server to detect changed blocks on the storage. For the second problem, we use the device mapping technique on VM host to create a virtual block device that can switch storage by remapping the physical block device. And finally we build an index mechanism on storage server to save migration time and reduce the amount of data transmission.

3.2 Indexing Virtual Machine Disk Storage States

Disk storage is relatively large compared to memory. It contains all the static data include operating system, programs and dynamic data like temp files, swap, data files, log files etc. If we use the same approach in migrating memory state to migrate disk storage, the migration process will take a lot of time to complete and the amount of data transmitted on the network will also be quite significant. To improve the efficiency of VM disk storage migration, we exploit data similarities among VM disk storages by building an index of the data blocks in the VM disk storages. The index is a hash table that stores the hash values of disk blocks as keys. We treat the key as the

fingerprint of a block. Every entry in the index represents a specific block and contains the reference information for the block within it. We call the reference information as block reference; it records of the device number and the storage block number associated with the block. Figure 2 shows an example index of disk storage with five blocks. The two tables on the right side are the prototypes of data structure and the arrows in the figure represent links in the data structure. When we index disk storage, first we allocate a block reference array with length equal to the number of blocks in the disk storage. The block reference array stores the links of block reference corresponding to the block in disk storage.

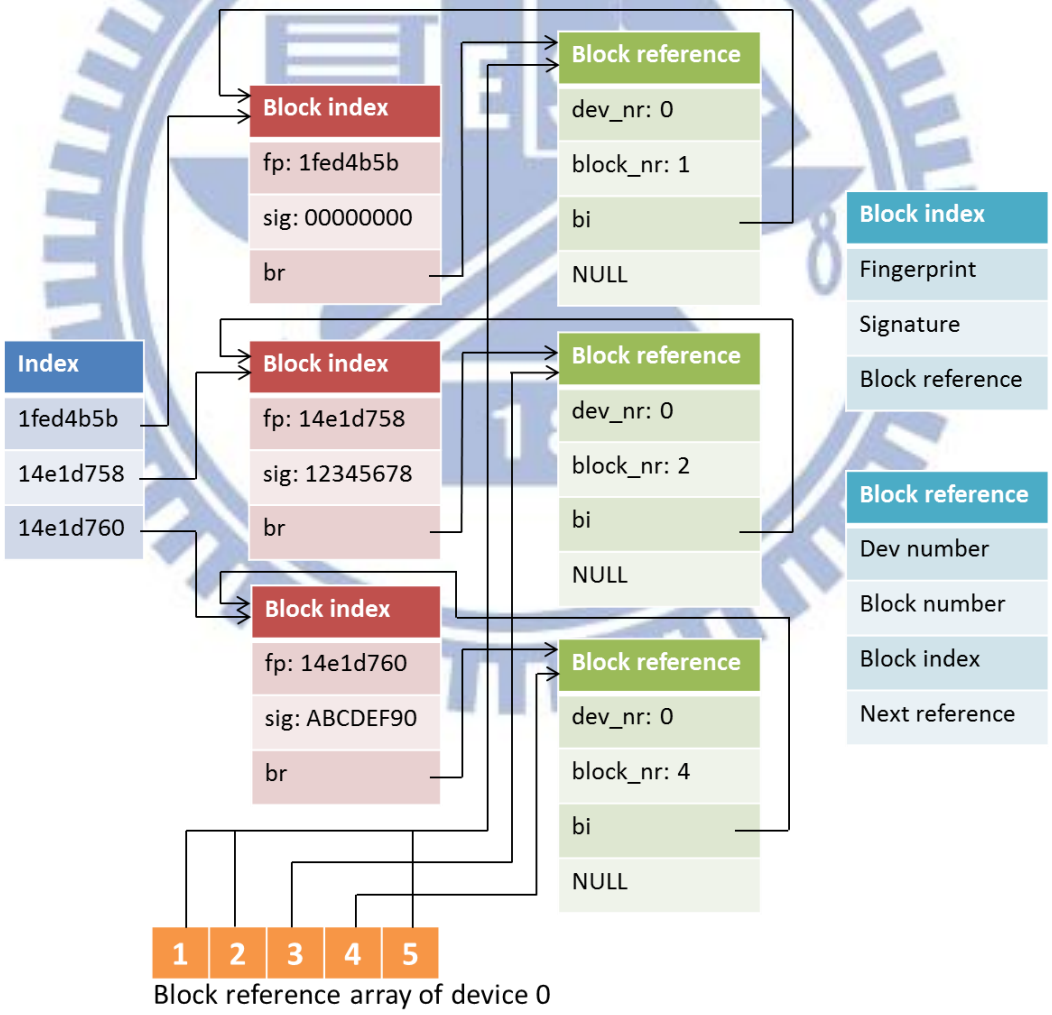


Figure 2: Example of index data structures

In this example, we can see that block 1, 2, and 5 have consistent data. So the links are point to the same block reference. The signature in the block index is a small piece of data sampled from the block. In our implementation, the block size is 512 bytes and the signature is the second 8 bytes of the block associated with the index. Since the hash value cannot guarantee the consistency of the block data, we use a two-step checking mechanism to reduce the probability of hash collision. Either we add index entry or search from the index; first we calculate the hash value of the block and match the hash value as usual. If the hash value gets a match, we then compare the signature of the block. The action gets valid if and only if it passes the two-steps checks. If the signatures do not match, we will treat the two blocks as different when creating their indexes or when querying the indexes during migration.

Given a key (or fingerprint), we can efficiently look up the index to determine if the corresponding block exists or not. If exists, we can use the information contained in the block reference of the key to retrieve the block data from the available sources. The index can also be queried by other storage servers on the same LAN to looking up blocks from neighboring servers or be used as local reference within a storage server.

3.3 Migrating Virtual Machine Disk Storage States

When moving storage from one storage server to another storage server, we want to reduce both the time and bandwidth consumed in the migration process. We use the index mechanism to achieve this purpose. In order to allow the migration process to take place while the VM is kept running, our system employs a bitmap to track dirty blocks on the VM storage (i.e. blocks that are modified by the running VM during the migration process).

Source storage server migration process flow

```
// Block interval (I, N) := the blocks start from the Ith block and end at the (I+N-1)th block
// Bit interval (I, N) := the bits start from the Ith bit and end at the (I+N-1)th bit
// D := the block device that would be migrated
// D[I] := the Ith block of D
// D.block_nr := the number of blocks of D
// D.bitmap := the bitmap of D
// D.bitmap[I] := the Ith bit of the bitmap of D
// D.br := the block references of D
// D.br[I] := the block reference of the Ith block of D
// D.br[I].index := the block index of the block reference D.br[I]
// BI := block index
// BI.fingerprint := the fingerprint of the block index
// BI.signature := the signature of the block index
// ZBI := the block index of zero block

// Clear bitmap of block device D
For (I = 0; I < D.block_nr; I++) {
    D.bitmap[I] = 0
}
PHASE1:
For (I = 0; I < D.block_nr; I++) {
    If (D.br[I] != NULL) {
        If (D.br[I].index != ZBI) {
            FP = D.br[I].index.fingerprint
            SIG = D.br[I].index.signature
            Send INDEX_TYPE = BLOCK to destination storage server
            Send index information (I, FP, SIG) to destination storage server
        } Else {
            N = 0
            Do {
                I++
                N++
            } While (D.br[I].index == ZBI)
            Send INDEX_TYPE = ZERO to destination storage server
            Send block interval (I, N) to destination storage server
        }
    }
}
// End of PHASE1
Send INDEX_TYPE = ZERO to destination storage server
Send block interval (0, 0) to destination storage server
PHASE2:
For each received block interval (I, N) from destination storage server {
    If (N != 0) {
        For (J = I; J < I+N; J++) {
            D.bitmap[J] = 1
        }
    } Else {
        Break   // block interval with length zero means the request is over
    }
}
PHASE3:
I = 0
While (number of dirty blocks on D > dirty threshold and I < iterate threshold) {
    For each dirty bit interval (I, N) in D.bitmap {
        For (J = I; J < I+N; J++) {
            Send OPCODE = TRANSFER_DATA to destination storage server
            Send (I, D[I]) to destination storage server
            D.bitmap[J] = 0
        }
    }
    I++
}
Send pause VM message to VM host machine
Send switch target message to VM host machine
PHASE4:
For each dirty bit interval (I, N) in D.bitmap {
    For (J = I; J < I+N; J++) {
        Send OPCODE = TRANSFER_DATA to destination storage server
        Send block information (I, D[I]) to destination storage server
    }
}
Send OPCODE = COMPLETE to destination storage server
Send resume VM message to VM host machine
```

Destination storage server migration process flow

```
 1 // Block interval (I, N) := the blocks start from the Ith block and end at the (I+N-1)th block
 2 // Bit interval (I, N) := the bits start from the Ith bit and end at the (I+N-1)th bit
 3 // D' := the new block device that would be synchronized with the migrated block device
 4 // D'[I] := the Ith block of D'
 5 // D'.block_nr := the number of blocks of D'
 6 // D'.bitmap := the bitmap of D'
 7 // D'.bitmap[I] := the Ith bit of the bitmap of D'
 8
 9 Create new block device D’
10 For (I = 0; I < D'.block_nr; I++) {
11     D'[I] = 0
12 }
13 PHASE1:
14 Do {
15     Receive INDEX_TYPE from source storage server
16     Switch (INDEX_TYPE) {
17         Case BLOCK:
18             Receive index information (I, FP, SIG) from source storage server
19             Find block index BI by FP on local index
20             If (BI != NULL) {
21                 Retrieve block data BD through the block reference of BI
22                 If (SIG match the signature part of BD) {
23                     D'[I] = BD
24                 }
25             }
26             Break
27         Case ZERO:
28             Receive block interval (I, N) from source storage server
29             If (N == 0) {   // block interval with length zero means the index transfer is over
30                 COMPLETE = True
31             }
32             For (J = I; J < I+N; J++) {
33                 D'.bitmap[J] = 1
34             }
35             Break
36     }
37 } While (COMPLETE == False)
38 PHASE2:
39 For each clean bit interval (I, N) in D'.bitmap {
40     Send block data request of block interval (I, N) to source storage server
41 }
42 // End of PHASE2
43 Send block data request of block interval (0, 0) to source storage server
44 PHASE3 and PHASE4:
45 Receive OPCODE from source storage server
46 While (OPCODE != COMPLETE) {
47     Receive block information (I, BD) from source storage server
48     D’[I] = BD
49     Receive OPCODE from source storage server
50 }
```

Figure 3: Pseudo code of migration process

Figure 3 shows the pseudo code of the storage migration process. We use the pre-copy mechanism in our storage migration process and the process has four phases. In phase one, the source storage server transmits the index information of all the blocks to the destination storage server (line 22~38 in the source side migration process pseudo code). The destination storage server will create a rebuilder to receive the index information of the storage to be migrated. The destination storage server will search the local index or nearby storage servers for the block

data corresponding to the index information (line 17~26 in the destination side pseudo code migration process flow). We have a zero block optimization in the index transfer phase (line 28~36 in the source side process flow and line 27~35 in the destination side process flow). Details about the zero block optimization will be presented in Chapter 4. The bitmap at the destination storage server is used to track which of the blocks have been located through the index information. If a bit in the bitmap is clean (with value 0), it means that the corresponding block cannot be found at the destination storage server or at any of its buddy nodes. In this case, the storage migration process will fall back to the conventional way of transferring block data directly.

In phase two, the destination storage server will send requests to the source storage server for those blocks that have not been properly migrated in phase one (line 39~41 in the destination side pseudo code). The source storage server receives the requests and sets the corresponding bits in the bitmap representing which blocks need to be transferred in phase three (line 43~51 in the source side pseudo code).

In phase three, the source storage server will transmit the blocks requested by the destination storage server in phase two together with dirty blocks to the destination storage server. The migration process will repeatedly transfer new dirty blocks to the destination storage server (Note that the VM is still running) until one of the following two conditions is satisfied (line 53~63 in source side pseudo code). The first condition is that the number of dirty blocks doesn't exceed the threshold, which is a customizable value or can be dynamic calculate during migration process. The second situation is when the number of the iterations on transmitting the dirty blocks has reached an upper limit. With

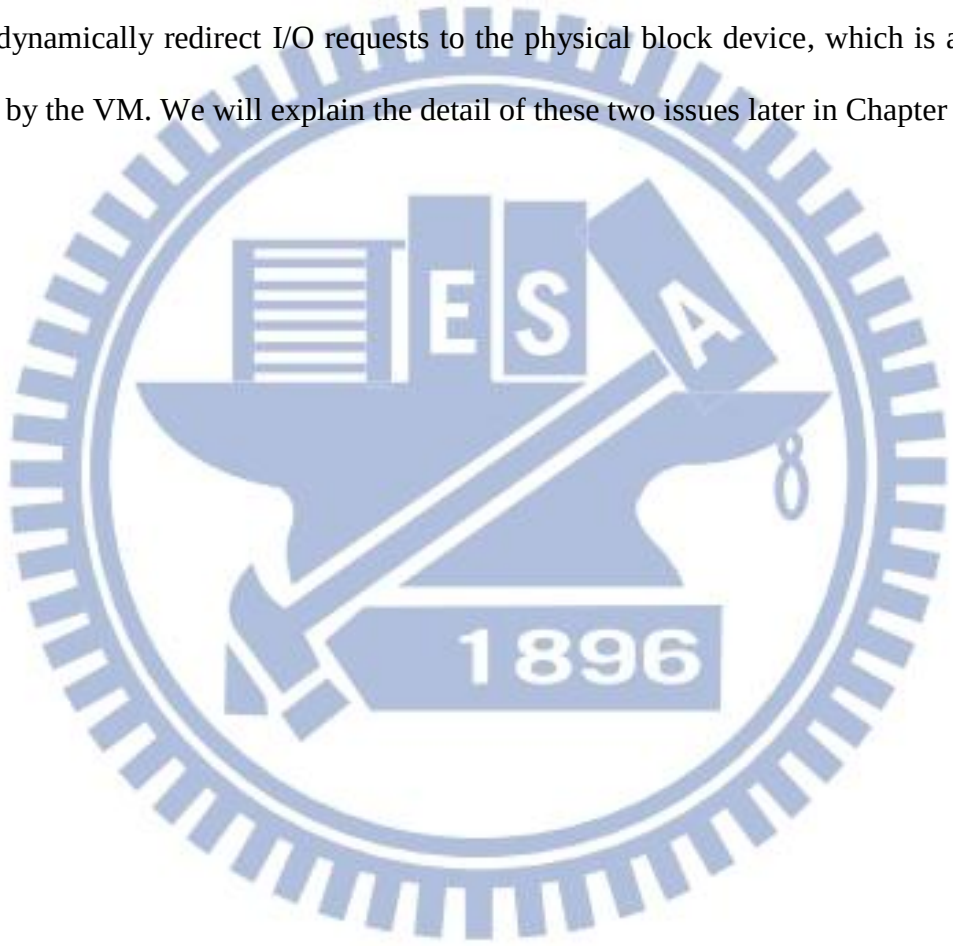
either one of these two conditions satisfied, the storage migration process will go into the final phase. In the final phase, we must ensure that there will be no more dirty blocks generating by the VM. So we have to suspend the virtual machine and switch the storage target for the VM (line 64~65 in source side pseudo code). After that, we can do the final round of dirty block transmission to the destination storage server. Finally, the VM storage at the destination storage server will be consistent with the original VM storage at the source storage server right before the VM is paused. The virtual machine will then be resumed (line 74 in source side pseudo code). The whole storage migration process is now complete.

3.4 Live Migration and VM State Consistency

When we do migration, we usually hope the services that VM provide can run continuously without interruption. If the migration process does not disrupt the user using the services, then the migration process is called live migration. That means the migration process must proceed with a running VM. However, it is necessary to suspend VM during transferring execution. Thus we have to shorten the downtime as much as possible so that the service would seem to be running continuously from the user's perspective.

To do live migration with storage, we must maintain data consistency of source and destination storage. Virtual machine continues running during migration process. There may be dirty blocks in the meantime of storage rebuilding. Thus we add the bitmap feature in storage server. A dirty bit (i.e. a bit with stat 1) in the bitmap represents the data of the corresponding block has been modified during the migration.

This change must also be updated again to the destination storage server. And we have to be able to clear the bit manually so that we know the update of the block is handled. In spite of dirty blocks, the switching action of storage may also be a problem. It is impossible to detach an active storage then re-attach it later when the system is still running. Thus we need another virtual layer to deal with this problem seamlessly. This virtual layer can create a virtual block device to let the VM attach and dynamically redirect I/O requests to the physical block device, which is actually used by the VM. We will explain the detail of these two issues later in Chapter 4.



Chapter 4 Implementation

Our system is implemented on x86_64 Fedora Linux environment. We use Xen 4.1.2 as the hypervisor for virtualization. Much of the implementations are user-level programs, which can be easily ported to other virtualization platform on Linux such as KVM. For storage, the implementation utilizes TCP/IP based iSCSI protocol, which is widely employed in datacenter environment and can be easily adapted to run in the WAN environment. iSCSI is a SAN protocol functions in the form of client-server architecture. iSCSI target is a storage resource exported by iSCSI storage server, and can be used through iSCSI initiator. iSCSI initiator acts as client to communicate with iSCSI storage server. The initiator can create a session with target and make the operating system use the storage resource as local SCSI block device.

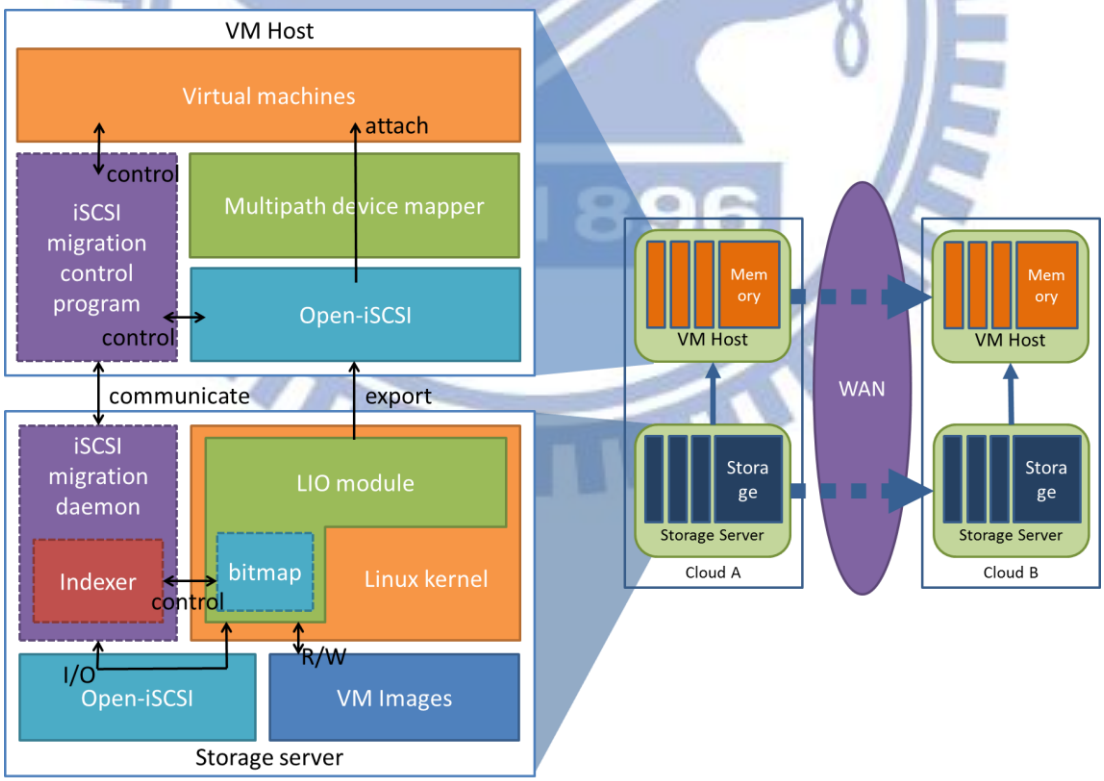


Figure 4: System architecture

Figure 4 shows the architecture of our system. The blocks with dotted frame are

the components we add to construct our system. iSCSI server is natively supported in Linux kernel 3.1 through the LIO module [15]. The LIO module is easy to configure and has good performance. The Open-iSCSI [16] is an initiator built on most Linux distributions. We build a migration daemon on the storage server. It can manage the LIO module by ConfigFS [17], create a new iSCSI target, and delete an iSCSI target on demand. The migration daemon comes with an indexer to index the block devices hosted on storage server. To ensure storage data consistency, the migration daemon's I/O are performed through the Open-iSCSI initiator on the storage server. The I/O path is same as one used by the virtual machines on the VM Host, so we only have to do sync to iSCSI target then other initiator would see the change we've done.

On the VM Host, we use our migration control program to send migration request to the migration daemon on the storage server to initiate a migration. When migration daemon receives the migration request, it would connect to the migration daemon on the destination storage server to start the migration process. And the migration daemon on the source storage server would send VM pause/resume or switch iSCSI target message back to migration control program corresponding to the progress of migration. When migration control program receive these messages, it would pause/resume VM through Xen control library or switch iSCSI target through Open-iSCSI library.

This chapter will present the implementation details of the proposed system. The specific details will be explained in Section 4.1~4.3.

4.1 iSCSI target block device dirty bitmap

During live migration of a VM, the VM may continuously write to the storage. It

is likely that a block that had been copied to the destination is written again in the migration process. The block is conventionally referred to as a dirty block, which should be copied again to the destination. We use a bitmap to track the dirty blocks. The bitmap is implemented in the LIO kernel module.

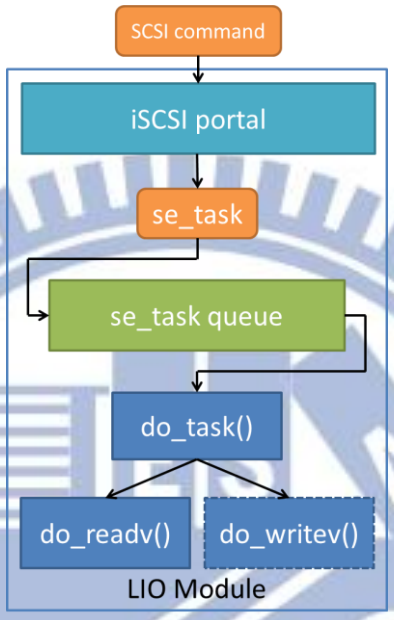


Figure 5: LIO Module

Figure 5 shows how LIO module handle iSCSI read/write command. The SCSI commands send via iSCSI protocol are received by the iSCSI portal of the LIO module. The iSCSI portal translates the SCSI command to se_task structure and adds it to the queue of the corresponding block device. Another kernel thread would fetch task from the queue using the do_task function to do the task. Then the read or write function would be called according to the task. When the write function is called to write some data blocks to storage, the corresponding bits of bitmap would be set representing the blocks are dirty. Since our migration daemon is in userspace, we create a file handle at /proc/target by using the proc filesystem [18][19] in Linux kernel. The migration daemon can access the content of the bitmap by memory mapping [20] through the file handle, and using ioctl [21][22] to manipulate the

bitmap including clearing the bitmap, setting or resetting bits in the bitmap.

4.2 Zero block optimization

Zero block is a block that all bits in it are zero. When we first do some analysis on VM image files, we find there are significant portion of a system image are zero blocks. This is because the VM image allocation tool would initialize the image with zero blocks when creating the image. This way, it is not necessary to transfer these zero blocks of storage during migration since all the zero blocks at the source storage are also zero blocks at the destination storage. Hence we add the zero block optimization in our migration daemon.

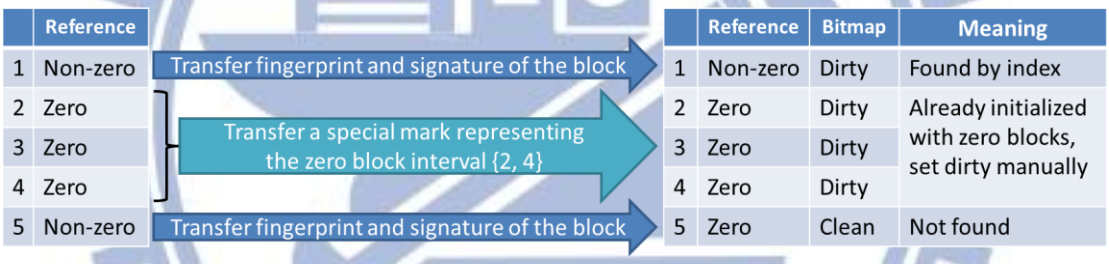


Figure 6: Example of zero block optimization

We manually create a zero block index entry before indexing any storage and save a link point to the index entry. This way we can tell which block is zero block simply by comparing the link of the block index entry in block reference and the link of the zero block index entry. Figure 6 shows an example on how zero block optimization works and presents the meaning of the bitmap on destination storage server. We transfer block indexes one by one. If we find the index of a block matches with the zero block index, we will keep looking for the index of the next block until a non-zero block is found. A special mark along with the offset of the starting zero block and the total number of zero blocks will be sent to the destination. The

destination storage server doesn't even write the content of zero blocks to the storage, it set the bits of bitmap instead to represent the contents of the blocks are found and written. This way we can greatly save the amount of data transmission between two storage servers and I/O overhead on destination storage server, and then save the migration time.

4.3 Seamless switching of iSCSI targets

Storage is attached to virtual machine when the virtual machine is running. We cannot directly detach it and attach another block device since the storage stat in operating system of the virtual machine would change and may cause some problem. Thus we need a virtual layer to create a virtual block device and dynamically map the virtual device to the block device we want to use. Linux kernel has device mapper [23] based on the foundation of LVM2 [24]. And we found a tool called multipath [25] can utilize device mapper to solve this problem.

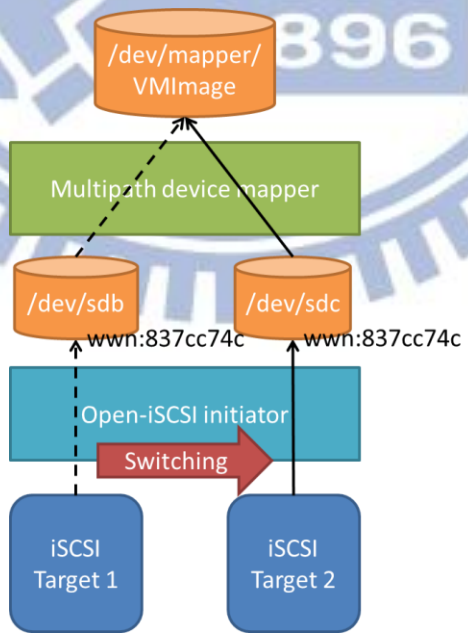


Figure 7: Example of switching iSCSI target

Figure 7 shows how multipath device mapper work. Originally multipath is used

to do fault tolerance of iSCSI storage on distinct network path. For example, if we have two different network paths, we can setup two sessions to the same iSCSI target storage. This way the system would add two block device, but multipath would recognize these two devices are from the same source by match their world wide name (wwn) and create a virtual block device to represent these two devices. It would choose one of two block devices mapping to the virtual block device for primary usage and automatically switch the mapping to the other when the connection of the primary storage has problem or just lost. We use multipath in a similar way. The virtual machine uses the block device that multipath created. At the beginning of migration, destination storage server would create storage with the same wwn of the source storage. When we need to switch target, we just logout from source iSCSI storage and login to the destination iSCSI storage. The multipath would be forced to switch the mapping to the block device of destination storage. It's unnecessary to detach the block device VM attached. Hence the iSCSI target is switched without disturbing execution of VM.

Chapter 5 Experiment Results

We evaluate our system design by several different experiments. Our storage servers in the testbed environment are equipped with Dual Intel Xeon E5520 processors, 16GB RAM and 1TB hard disk. In Section 5.1, we first analyze the similarity between the storages of multiple VM systems. And Section 5.2 gives the evaluation of the overhead for running our system. In Section 5.3, we evaluate the improvement on migration time by the proposed index mechanism. In Section 5.4, we use various benchmarks to measure the migration downtimes of the proposed system.

5.1 Similarity analysis of VM system storages

Before we test our system, we want to know whether there is duplicate data in the storages of representative VM systems. And if it exists, what is the percentage of the data that have duplicates. Higher similarity will bring higher index hit rate when we migrate VM system storages. It means that we can improve the migration performance significantly from the index mechanism. We build the representative VM systems with three Linux distributions including CentOS, Fedora and Ubuntu. For each distribution, the newest few versions were chosen to do this test. We do not use Windows in the comparison because it only accounts for 7.9% of share in the VMs deployed in IaaS clouds [26]. We allocate 32GB storage for each VM system and do a fresh installation of standard LAMP stack [27] on the system. The partition layout follows the default settings of each distribution. We also pick development tools including compilers, make utilities and develop libraries for the installation. Despite the fresh installed system storage, we also choose two production systems

Sense and Better. Sense runs CentOS5.8 and serves as the web portal of SENSE lab website. The storage size of Better is 300GB. Better runs Fedora14 and serves as the online judge system for undergraduate courses. In the end, we have a total of 10 VM system storages. The experiment is based on calculating the pairwise similarity of the storages. First we index one of the storage in the pair. Then we index the other. Finally, we count the number of blocks that hit in index. Since a sector is the smallest unit for accessing the disk, we use the size of a sector (512 bytes) as the unit for indexing in order to maximize the similarity.

Base\Target	CentOS 5.8	CentOS 6.3	Fedora 14	Fedora 15	Fedora 16	Ubuntu 11.04	Ubuntu 11.10	Ubuntu 12.04	Sense	Better	Average
CentOS5.8	n/a	5.36%	4.27%	3.61%	3.40%	3.85%	4.12%	3.96%	9.51%	0.39%	4.14%
CentOS6.3	7.26%	n/a	14.88%	11.57%	8.58%	7.18%	7.42%	7.35%	1.90%	1.21%	6.63%
Fedora14	7.37%	18.69%	n/a	28.03%	21.92%	9.08%	10.52%	7.64%	1.92%	4.01%	12.40%
Fedora15	6.68%	15.70%	29.45%	n/a	34.34%	13.25%	14.01%	11.39%	1.71%	2.04%	14.92%
Fedora16	6.25%	12.00%	21.94%	35.33%	n/a	11.23%	14.05%	14.95%	1.60%	1.65%	14.73%
Ubuntu11.04	2.65%	4.53%	4.79%	6.79%	5.64%	n/a	63.39%	21.09%	0.73%	1.13%	12.26%
Ubuntu11.10	2.09%	3.28%	3.66%	4.97%	4.83%	40.23%	n/a	23.04%	0.62%	1.02%	9.33%
Ubuntu12.04	1.79%	3.05%	2.57%	3.90%	4.86%	15.72%	22.70%	n/a	0.54%	0.80%	6.52%
Sense	41.75%	6.26%	5.41%	4.66%	4.40%	3.95%	4.26%	3.86%	n/a	1.22%	8.23%
Better	5.79%	14.20%	46.45%	24.16%	19.08%	13.65%	15.97%	11.33%	2.15%	n/a	13.31%

Table 1: Similarity between VM system storages

CentOS 5.8	CentOS 6.3	Fedora 14	Fedora 15	Fedora 16	Fedora 17	Ubuntu 11.04	Ubuntu 11.10	Ubuntu 12.04	Sense	Better
94.45%	92.92%	91.04%	87.53%	87.21%	85.30%	94.22%	96.96%	96.92%	65.66%	83.61%

Table 2: Zero block percentage in system storage

Table 1 shows our experiment result. The system on the left is the first storage we add into index and the system on the top is the second storage we add into index. The percentages in the table indicate how much data transmission we can save when we do migration on the storage of the system from the top and we have the index of the system from the left. We rule out the zero blocks since it's not used by the VM system and have a dominated rate in storage (see Table 2). CentOS has the lowest

average similarity compared with other distributions. We think this is because CentOS has a longer release cycle and different versions have much less overlapping. We can see in the table that the similarities between different versions of Fedora systems or different versions of Ubuntu systems are all above 15 percent. The similarities between any Fedora system and any Ubuntu system are all above 10 percent if we use Fedora as the index base. This indicates the amount of data that can be reduced by the index mechanism during migration.

We also carry out an experiment on the variation of the block size for indexing. We choose Ubuntu11.04 and Ubuntu11.10 for the experiment, as they have the highest average similarity according to Table 1. We increase the block size from 512 bytes to 16384 bytes and observe the influence on the average similarity.

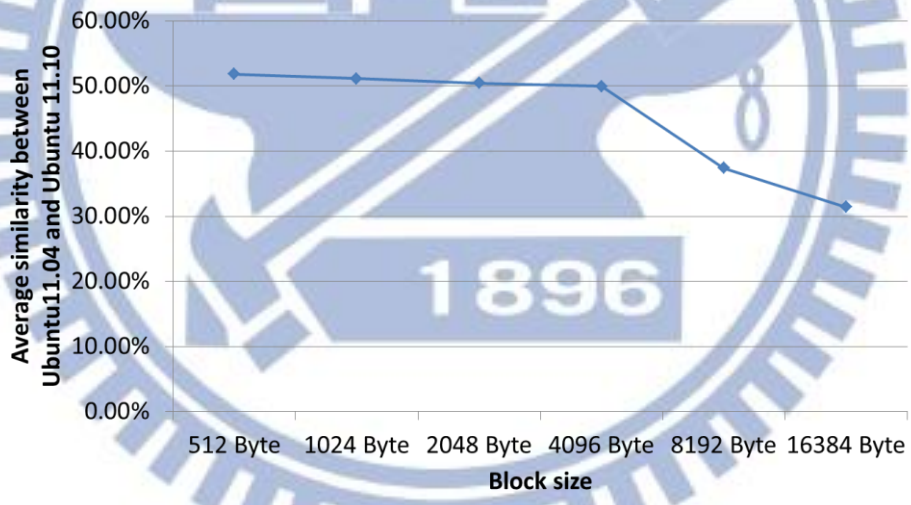


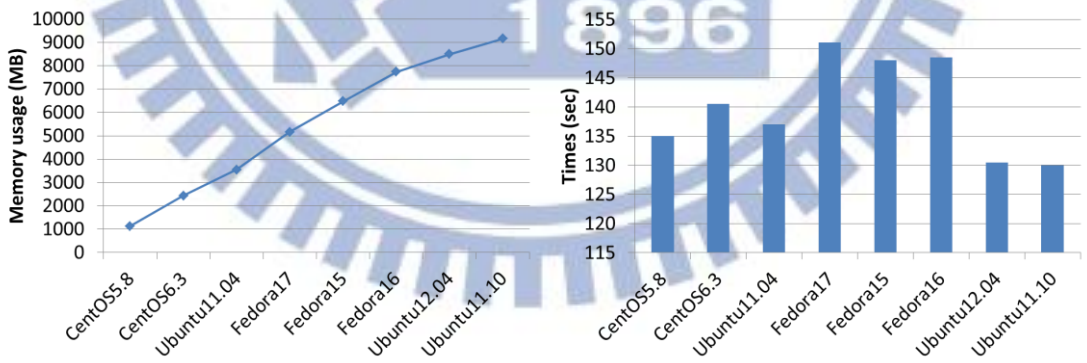
Figure 8: Similarity on different block sizes

Figure 8 shows the experiment result. Basically the similarity becomes lower when we use bigger block size since the probability of matching the data in a block becomes smaller. There is a big drop on similarity between block size of 4096 bytes and 8192 bytes. We think this is because the filesystem on the disk uses block size of 4096 bytes as the basic unit in order to improve I/O performance. But the partition or filesystem configuration may cause the blocks written by the filesystem do not align

with the block slice we use in the indexing process and data skew occurs. Thus for the following experiments, we still use the block size of 512 bytes for indexing to get the best similarity.

5.2 Indexing overhead

If we want to use the index mechanism to assist migration, we need to index the storages before migration. It would take time and memory space to build the index. We want to test how much time and space are required for building index. To index storage, it is necessary to read through the whole storage once. This is typically an I/O bound task limited by disk read speed. For every new block that cannot be found in the index, we must allocate some memory space to create new entry to store the index information for the block. We do the test by indexing the 8 sample system storage mentioned in Section 5.1. We record the time consumed and total memory usage every time after storage is indexed.



(a) (b)

Figure 9: Index overhead

Figure 9 shows the result of the test. The average time consumed by indexing is 140 seconds per storage. This means the indexing rate is about 234MB/s, which is far beyond the disk read speed. We think this is because the storage is stored in sparse

file format [28]. The read speed could be very fast when reading the holes in sparse file. Thus the average indexing time of Fedora is more than the time of Ubuntu since there are more data in the Fedora systems than the Ubuntu systems. The memory usage grows almost linearly when adding new storage to the index. We think the reason is that the similarities between any two system storages are on average 10%~20%, so an average 80~90% of new data will result in new entries in the index. In practice, if the size of the index is a concern, one can limit the size of the index at the expense of lower hit rates and potentially longer migration time.

5.3 Migration time and amount of data transmission

In this experiment, we look at the effect on migration time and amount of data transmission by employing our system for VM migration. We compare the results with the baseline system, which is emulated by running the proposed system with the indexing mechanism turned off. With the baseline system, the full disk states of a VM will have to be copied byte-by-byte from the source storage server to the destination storage server. We use the two production systems Sense and Better as in Section 5.1 for the experiment. We setup two iSCSI storage servers and limit the speed of network to 100Mbps to emulate the WAN environment. For migration with our system (with the indexing mechanism), we assume that a freshly installed system with the same OS used by the source VM is available on the destination storage server for building index (i.e. for Sense, a storage with freshly installed CentOS 5.8 is provided on the destination storage server for building index, and for Better, a storage with freshly installed Fedora 14 is provided instead). To avoid the influence of downtime, this experiment is done without VM running. We record the total time of migration

and the amount of data transmission over the network.

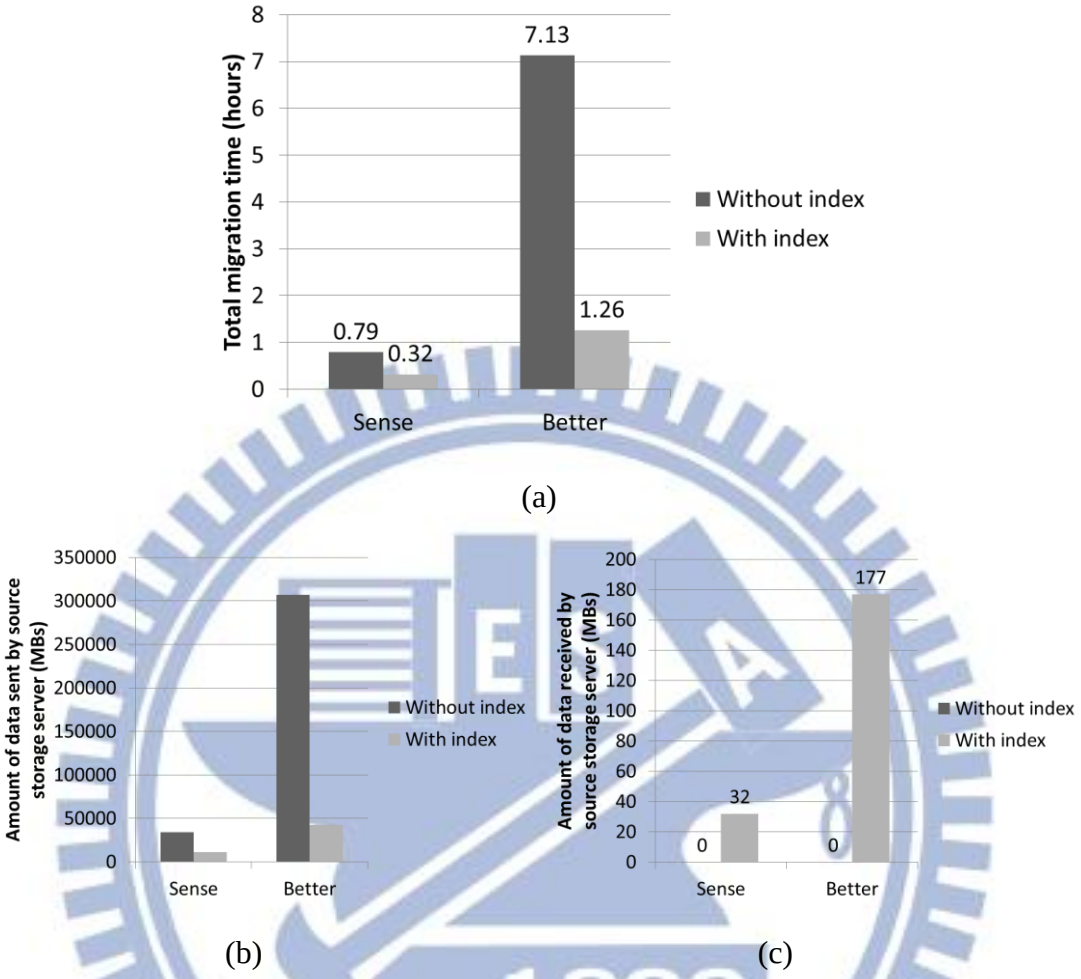


Figure 10: Comparison of migration time and network transmission with and without index mechanism

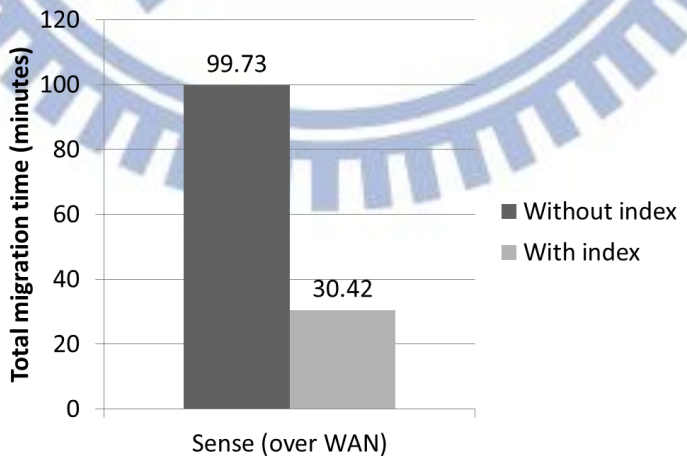


Figure 11: Comparison of migration time when migration over WAN

Figure 10 shows the result of the experiment. Our system reduces about 59% of

the migration time for Sense and about 82% of the migration time for Better. In terms of network data transmission, our system reduces about 66% of data transmission for Sense and about 86% of data transmission for Better. The rate of migration time saving is close to the rate of network data transmission saving. This is because most of the migration time is due to data transmission during migration. We also take our system to real world for testing. We migrate Sense from the educational network in NCTU, HsingChu to Yi-Lan. The bandwidth is 50Mbps. Figure 11 shows the result of the testing. Our system reduces about 69% of the migration time and takes only about half-hour to complete migration which is almost copy the storage at the speed of 19MB/s. This indicates our system is practical for WAN environment.

5.4 Downtime evaluation

Our system uses pre-copy mechanism so that the virtual machine must be paused in order to prevent VM from generating dirty memory pages or storage blocks. The services VM provide are temporarily unusable while VM is paused. This downtime can be varied depend on how much loading the VM is running on. If the VM is idle, only few dirty memory pages or storage blocks would be generated. Thus we can quickly synchronize these data and resume the VM. But if the VM is doing a lot of I/O tasks (like writing file, decompressing file) during storage migration, it would take more time to synchronize the data. Hence the downtime would also increase. We want to know that how much downtime will increase when there are I/O workloads running on VM during storage migration. We evaluate this by running several I/O intensive task including dbench [29] and kcbench [30] on VM while migrating. The execution time of I/O task is long enough to ensure the task is still running during the

downtime of the migration.

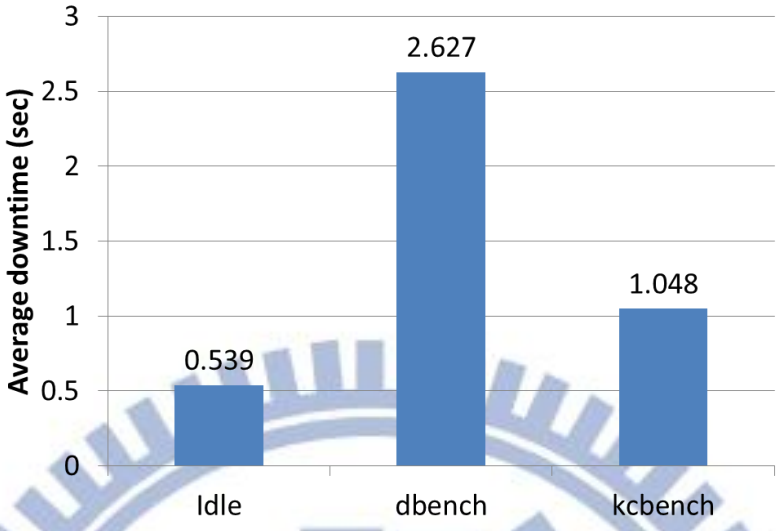


Figure 12: Average downtime of storage migration with different workloads

Figure 12 shows the result of the evaluation. The average downtime for migrating an idle VM is about 539 milliseconds. It's short enough for most services to operate continuously without interruption, which implies our system is able to do live storage migration. The downtime increases by 242% when there is workload on the VM. For instance, the downtime with dbench is 2.627 seconds and the downtime for kcbench is 1.048 seconds. This seems a little bit high but if we take a look on the absolute value of the downtime, we can see that the average downtime is less than 3 seconds. Those services which do not need persistent network connection (e.g. Web, Message) can tolerate the downtime with negligible impact on user experience. Overall, our system performs well with respect to the downtime evaluation.

Chapter 6 Conclusion and Future Work

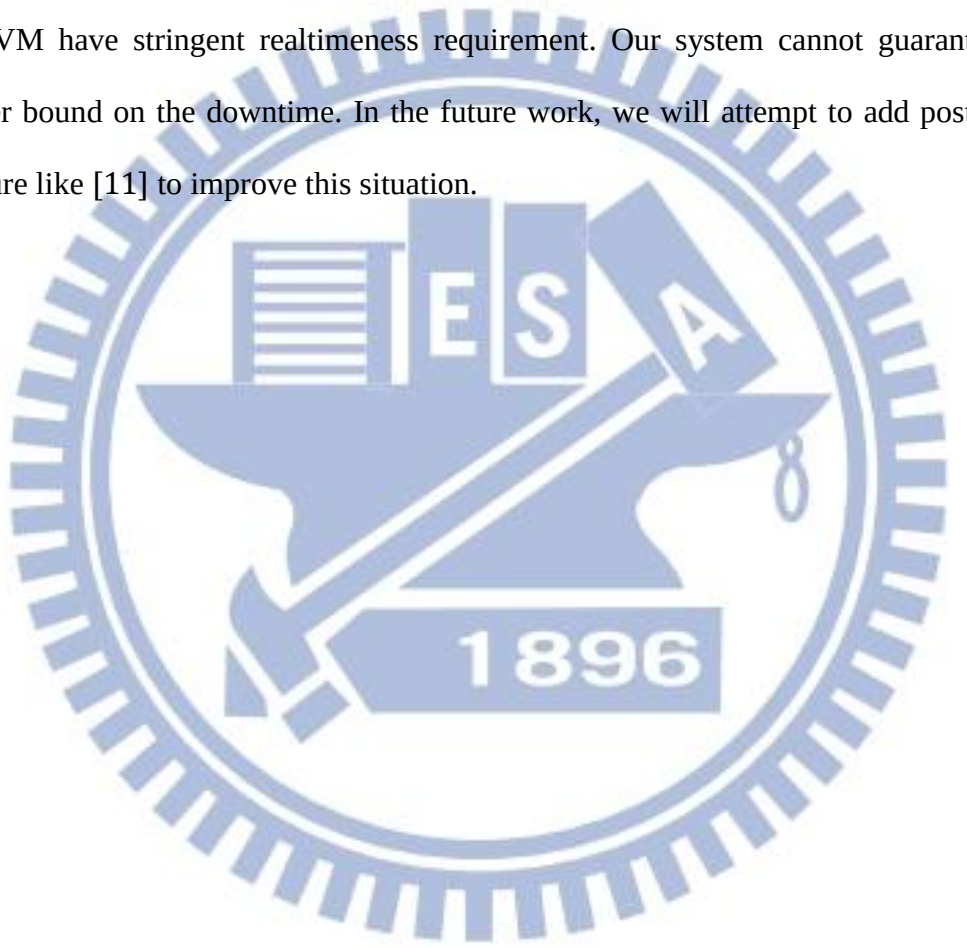
Conventional virtual machine migration is not suitable for WAN environment, because it lacks the ability of storage migration, and sharing storage WAN is impractical due to long transmission latency and limited bandwidth on the WAN. Even if storage migration across WAN is implemented, the migration process would take a lot of time and network bandwidth to complete since the storage typically has a lot of data which is typically in the range of at least hundreds of gigabytes.

We propose a system to facilitate storage migration on iSCSI storage across WAN, thereby making it possible to achieve VM migration over the WAN. Our system builds an index for the storage servers and uses the index to reduce the amount of data transmission in the migration process. We also design a bitmap mechanism in the iSCSI storage server and use the device mapper utility to help us achieve live storage migration.

We evaluate the system by several experiments. The results confirm that the system not only reduces the migration time significantly but also reduces the amount of network transmission greatly. Overall, our system realizes virtual machine migration over WAN.

The evaluation also brings up some deficiencies in the current implementation of the system. One deficiency is the memory usage by the indexing mechanism seems a little bit too high. The main reason causing this problem is that there are too many blocks to index. We can reduce the number of blocks by enlarging the size of a block. Ordinarily, the index block size is 512 bytes, but we can increase the size to 4096 bytes. This way the number of blocks would be decreased to one eighth of the current size, and the memory usage would also be reduced. However, there is the alignment

issue when we use bigger block sizes. A possible solution is to use filesystem information to assist the indexing mechanism instead of requiring the block device to use a larger block size throughout the whole storage system hierarchy. Filesystem contains high level information of the blocks in storage. We can use the information to aggregate smaller blocks into a huge block by ourselves, thereby not needing the paravirtualization driver. The downtime is also a deficiency if the services running on the VM have stringent realtimeness requirement. Our system cannot guarantee an upper bound on the downtime. In the future work, we will attempt to add post-copy feature like [11] to improve this situation.



References

[1] D. S. Milo'ji\vcić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process migration," *ACM Comput. Surv.*, vol. 32, no. 3, pp. 241–299, Sep. 2000.

[2] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, Berkeley, CA, USA, 2005, pp. 273–286.

[3] "Migration - KVM." [Online]. Available: http://www.linux-kvm.org/page/Migration. [Accessed: 10-Oct-2012].

[4] "VMware-VMotion-DS-EN.pdf." [Online]. Available: http://www.vmware.com/files/pdf/VMware-VMotion-DS-EN.pdf. [Accessed: 10-Oct-2012].

[5] Q. Li, J. Huai, J. Li, T. Wo, and M. Wen, "HyperMIP: Hypervisor Controlled Mobile IP for Virtual Machine Live Migration across Networks," in *11th IEEE High Assurance Systems Engineering Symposium, 2008. HASE 2008*, 2008, pp. 80 –88.

[6] M. Tsugawa, P. Riteau, A. Matsunaga, and J. Fortes, "User-level virtual networking mechanisms to support virtual machine migration over multiple clouds," in *2010 IEEE GLOBECOM Workshops (GC Wkshps)*, 2010, pp. 568 – 572.

[7] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan, "Live virtual machine migration with adaptive, memory compression," 2009, pp. 1–10.

[8] X. Zhang, Z. Huo, J. Ma, and D. Meng, "Exploiting Data Deduplication to Accelerate Live Virtual Machine Migration," 2010, pp. 88–96.

[9] F. F. Moghaddam and M. Cheriet, "Decreasing live virtual machine migration down-time using a memory page selection based on memory change PDF," 2010, pp. 355–359.

[10] K. Haselhorst, M. Schmidt, R. Schwarzkopf, N. Fallenbeck, and B. Freisleben, "Efficient Storage Synchronization for Live Migration in Cloud Infrastructures," in *Proceedings of the 2011 19th International Euromicro Conference on Parallel, Distributed and Network-Based Processing*, Washington, DC, USA, 2011, pp. 511–518.

[11] T. Hirofuchi, H. Ogawa, H. Nakada, S. Itoh, and S. Sekiguchi, "A Live Storage Migration Mechanism over WAN for Relocatable Virtual Machine Services on Clouds," in *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, Washington, DC, USA, 2009, pp. 460–465.

[12] S. Akoush, R. Sohan, B. Roman, A. Rice, and A. Hopper, “Activity Based Sector Synchronisation: Efficient Transfer of Disk-State for WAN Live Migration,” in *2011 IEEE 19th International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2011, pp. 22 –31.

[13] T. Wood, K. K. Ramakrishnan, P. Shenoy, and J. van der Merwe, “CloudNet: dynamic pooling of cloud resources by live WAN migration of virtual machines,” *SIGPLAN Not.*, vol. 46, no. 7, pp. 121–132, Mar. 2011.

[14] “VMware-Storage-VMotion-DS-EN.pdf.” [Online]. Available: http://www.vmware.com/files/pdf/VMware-Storage-VMotion-DS-EN.pdf. [Accessed: 26-Sep-2012].

[15] “Linux Unified Target - Main Page.” [Online]. Available: http://linux-iscsi.org/wiki/Main_Page. [Accessed: 10-Sep-2012].

[16] “Open-iSCSI project: Open-iSCSI – RFC3720 architecture and implementation.” [Online]. Available: http://www.open-iscsi.org/. [Accessed: 10-Oct-2012].

[17] “Linux Kernel Documentation :: filesystems : configfs.” [Online]. Available: http://www.mjmwired.net/kernel/Documentation/filesystems/configfs/. [Accessed: 14-Sep-2012].

[18]“proc(5): process info pseudo-file system - Linux man page.” [Online]. Available: http://linux.die.net/man/5/proc. [Accessed: 10-Oct-2012].

[19] “procfs - Wikipedia, the free encyclopedia.” [Online]. Available: http://en.wikipedia.org/wiki/Procfs. [Accessed: 10-Oct-2012].

[20] “mmap(2) - Linux manual page.” [Online]. Available: http://www.kernel.org/doc/man-pages/online/pages/man2/mmap.2.html. [Accessed: 10-Oct-2012].

[21] “ioctl(2) - Linux manual page.” [Online]. Available: http://www.kernel.org/doc/man-pages/online/pages/man2/ioctl.2.html. [Accessed: 10-Oct-2012].

[22] “ioctl - Wikipedia, the free encyclopedia.” [Online]. Available: http://en.wikipedia.org/wiki/Ioctl. [Accessed: 10-Oct-2012].

[23] “Device-mapper Resource Page.” [Online]. Available: http://sources.redhat.com/dm/. [Accessed: 08-Oct-2012].

[24] “LVM2 Resource Page.” [Online]. Available: http://sourceware.org/lvm2/. [Accessed: 08-Oct-2012].

[25] “multipath-tools:Home.” [Online]. Available: http://christophe.varoqui.free.fr/. [Accessed: 08-Oct-2012].

[26] “The Cloud Market: EC2 Statistics.” [Online]. Available: http://thecloudmarket.com/stats. [Accessed: 27-Oct-2012].

[27] “LAMP (software bundle) - Wikipedia, the free encyclopedia.” [Online]. Available: http://en.wikipedia.org/wiki/LAMP_(software_bundle). [Accessed: 10-Oct-2012].

[28] “Sparse file - Wikipedia, the free encyclopedia.” [Online]. Available: http://en.wikipedia.org/wiki/Sparse_file. [Accessed: 10-Oct-2012].

[29] “DBENCH.” [Online]. Available: http://dbench.samba.org/. [Accessed: 07-Oct-2012].

[30]“kcbench(1): Kernel compile benchmark - Linux man page.” [Online]. Available: http://linux.die.net/man/1/kcbench. [Accessed: 07-Oct-2012].