Shared Memory Based Communication Between Collocated Virtual Machines

MTP Stage 1 Report

Submitted in partial fulfillment of the requirements for the degree of

Master of Technology

by

Vaibhao Vikas Tatte
Roll No: 08305905

under the guidance of

Prof. Purushottam Kulkarni
&
Prof. Umesh Bellur

Department of Computer Science and Engineering
Indian Institute of Technology, Bombay
Mumbai
2010
# Contents

1 **Introduction** 1  
1.1 Types of Virtualization 1  
1.2 Xen 1  
1.3 Communication Between Collocated Virtual Machines in Xen 2  
1.4 Problem Definition 3  

2 **Related Work** 4  
2.1 Xensocket 4  
2.2 XWay 4  
2.3 XenLoop 5  
2.4 Optimization of Network File System (NFS) Read Procedure 5  
2.5 Fido 6  

3 **Solution Approach** 7  

4 **Background** 8  
4.1 Xen Support to Share Memory Among Domains 8  
4.2 Split Device Driver 8  
4.3 Network File System 9  

5 **Ongoing Work** 11  
5.1 NFS Benchmarking for Virtualized Environment 11  
5.1.1 Experiments 11  
5.1.2 Setup 11  
5.1.3 Results 12  
5.1.4 Experience During Experiments 15  
5.2 Porting NFS Read Procedure 16  

6 **Future Goals & Time-line** 17  

7 **Conclusions** 18  

A **Experience During Porting** 19
Abstract

Today, in the era of space & power crunch, Virtualization technologies are getting lot of research interest because of their capability to share the hardware resources among multiple operating systems and still maintains isolation between virtual machines. Isolation constraint between virtual machines limits maximum achievable communication throughput between two virtual machines running on same physical host. In this thesis we are concentrating on improving this communication throughput by bypassing TCP/IP network stack. We are exploiting use of shared memory based communication between two virtual machines. Initially we are concentrating on implementation of shared memory based communication in Network File System (NFS) by modifying read & write procedure of NFS. We are also presenting the benchmark for read & write procedure of NFS in virtualized environment for evaluation of implementation.
Chapter 1

Introduction

Virtualization is emerging as integral part of modern Data Center, mainly because of their capability to share the underlying hardware resources & still provides isolated environment to each application. Other benefits of virtualization include saving on power by consolidation of different virtual machines on a single physical machine, migration of virtual machine for load balancing etc. Virtualization provides full control of resource allocation to administrator, resulting in optimum use of resources.

1.1 Types of Virtualization

Core of any virtualization technology is Hypervisor or Virtual Machine Manager (VMM). Hypervisor is a piece of software which allows each virtual machine to access & schedule the task on resources like CPU, disk, memory, network etc. At the same time hypervisor maintains the isolation between different virtual machines.

Virtualization can be classified by the method in which hardware resources are emulated to the guest operating system. They are as follows

- **Full Virtualization** - Hypervisor controls the hardware resources & emulates it to guest operating system. In full virtualization guest do not require any modification. KVM[12] is an example of full virtualization technology.

- **Paravirtualization** - In paravirtualization hypervisor controls the hardware resources & provides API to guest operating system to access the hardware. In paravirtualization, guest OS requires modification to access the hardware resources. Xen[4] is an example of paravirtualization technology.

In this thesis, we are focusing on Xen as our target hypervisor.

1.2 Xen

Overall architecture of Xen is as shown in Figure 1.1. Xen architecture is a 3-layer architecture, Hardware layer, Xen hypervisor layer, Guest Operating System layer. Hardware layer contains all the hardware resources. Xen hypervisor is a virtualization layer. Guest operating system layer contains all the guest operating system installed. In Xen terminology, guest operating system is also called as domain. Among these domains, one domain have highest privileges and has direct access to all the hardware resources. This domain is called as domain-0 or dom0. When Xen is started on physical machine, it is booted in dom0 automatically. All other domains are unprivileged domain & do not have direct access to any of the resources directly. These domains are called as domain-U or domU, while U can be replaced with domain number.

domU has to access the hardware resources via dom0 using Xen API. More details are studied in Section 4.2.
1.3 Communication Between Collocated Virtual Machines in Xen

Two virtual machines which are running on same physical machine are called as collocated virtual machines. When two collocated virtual machines want to communicate with each other, they need to go through the dom0.

![Diagram of Xen Architecture](image)

**Figure 1.1: Xen Architecture**

To elaborate more on this communication cost, consider a process A executing on VM1, wants to communicate with process B executing on VM2 collocated on same physical machine. Its communication path will be as shown in figure 1.2. Communication will need to be go through TCP/IP stack on VM1, then it will be passed to dom0. dom0 will then forward packet to VM2. On VM2, first it will need processing at TCP/IP stack then it will be delivered two process B. Even though both the processes are executing on same machine, still it requires processing three three & has to travel through system bus two times. So this is a very high overhead.

![Diagram of Communication Between Two Virtual Machine Using TCP/IP Network Stack](image)

**Figure 1.2: Communication Between Two Virtual Machine Using TCP/IP Network Stack**

To validate this overhead, we did a small experiment. We compared the throughput of Native Unix Domain Socket with throughput of communication between two collocated VMs. In this experiment to calculate throughput of communication between two collocated VMs, we run iperf [14] client & server on two collocated virtual machines. To get the throughput of native unix domain socket, we executed a C program of native unix domain socket on a virtual machine. Its results are tabled in table 1.1. From the result it clear

<table>
<thead>
<tr>
<th>Inter-VM Communication</th>
<th>Native Unix Domain Socket</th>
</tr>
</thead>
<tbody>
<tr>
<td>2859 Mb/Sec</td>
<td>15010 Mb/Sec</td>
</tr>
</tbody>
</table>

**Table 1.1: Comparison of Throughput**

Two validate this overhead, we did a small experiment. We compared the throughput of Native Unix Domain Socket with throughput of communication between two collocated VMs. In this experiment to calculate throughput of communication between two collocated VMs, we run iperf [14] client & server on two collocated virtual machines. To get the throughput of native unix domain socket, we executed a C program of native unix domain socket on a virtual machine. Its results are tabled in table 1.1. From the result it clear
that throughput of native unix domain socket is little more than 4 times of throughput of collocated inter-VM communication. Even though, in both the scenario, communicating processes are running on same physical host, still there is large gap in throughput.

Questions that need to be ask is, can we improve upon throughput of collocated inter-VM communication?

In order to improve this performance of collocated inter-VM communication, we must bypass the TCP/IP newtork stack. So that we have to find the alternate solution for inter-VM communication. In Unix for inter process communication (IPC), we can use semaphore, message queues, pipes, shared memory [7]. But since Xen insured isolation between two Virtual Machines, these IPC mechanism of linux are not useful for inter-VM communication in Xen. But Xen has provided inherent support for sharing memory pages among two collocated virtual machines. These utility is called as Grant Table. This utility is discussed in detail in Section 4.1. Use of shared memory is depicted in figure 1.3.

Figure 1.3: Communication Between Two Virtual Machine Using Shared Memory

1.4 Problem Definition

Bypassing TCP/IP network stack by implementation of shared memory based communication between two virtual machines running on same physical host.

Issues that need to be addressed are during implementation are as follows

1. Who should share the memory pages. Client or Server.

2. Access level that should be given to be given to shared memory pages. This is very important from the security prospective. Since wrong access level may compromise the security of page sharing VM’s memory area.

3. How to identify, if the destination VM is collocated or not.

4. How to send event notification to destination VM.

5. How to support migration, still providing seamless communication using TCP/IP stack after migration.

6. Application transparent solution, so that legacy application should also support without need of recompilation

7. Minimum changes to guest OS.
Chapter 2

Related Work

In broader sense, shared memory based communication can be achieved in two ways.

1. **Application Transparent** - In this approach, shared memory based communication is implemented in Network stack. Positive point in this approach is that application do not require any changes & hence legacy application can continue to run without recompilation. Negative point here is, it require changes in GNU/Linux Kernel source code, which might be tedious to implement.

2. **Application Aware** - In this approach, shared memory based communication is achieved by means of API. Positive point here, it requires designing of dedicated library, hence no changes in the GNU/Linux Kernel source code. Negative point here is, application has to use this newly developed API instead of existing communication API. Hence application requires recompilation. So legacy application might not be easy to use with this approach, if its source code is not available.

We will now look at the various prior related work & will classify them according to above mentioned types.

2.1 Xensocket

Xensocket [18] is application aware solution to share memory across the domains in Xen. Xensocket provides its own socket API to use instead of TCP/UDP socket API. Xensocket bypass the TCP/IP stack. Xensocket provides one way communication between sender & receiver.

Sender creates Xen Event Channel to notify the events to receiver. Sender shares two types of pages, descriptor page of 4 KByte for intimating control information to receiver & buffer page of size multiple of 4 KByte as circular buffer to share the data with receiver. Xensocket is implemented in Xen 3.0.5 & GNU/Linux kernel version 2.6.16. Since this uses the Xen 3.0.5 APIs, it is not compatible with Xen 3.3, as there are lot of changes in Xen APIs. Researcher from different places have ported it to Xen 4.0. But some bugs are reported in this porting.

For message size upto 16 KBytes, Xensocket reports over 72 times improvement over TCP sockets, but this improvement is still less than Unix domain socket. While larger message size, Xensocket & Unix domain socket reports drop. But still Xensocket is comparable with Unix domain socket for larger file size.

One of the major drawback of Xensocket is that it do not support Migration. Also in Xensocket, administrator need to enter the information on collocation of VM manually while installing. In order to use Xensocket, application need to be changed & requires recompilation.

2.2 XWay

XWay [11] is an application transparent solution to share memory across the domains in Xen. XWay is implemented at socket layer of TCP/IP network stack. Although XWay is implemented at socket layer, it
bypass the rest of the stack. XWay creates a XWay channel for bidirectional communication. XWay channel is consist of Xen Event Channel to notify availability of data & data channel to transfer data. Data channel is having sender queue & receiver queue at both end of XWay channel. Positive point of XWay is that, it has full binary compatibility with existing Socket API. Since XWay channel is implemented at Socket Layer, it can easily switch from XWay channel to TCP channel to support Migration.

Drawback of XWay is that, execution time for socket interface; connect(), accept(), bind(), listen(), close() is longer for XWay as compared to TCP. Also XWay is developed in Xen 3.0.3 & GNU/Linux Kernel version 2.6.16.29. This implementation is not compatible with the Xen 3.3 & above.

2.3 XenLoop

XenLoop [17] is another implementation in category of application transparent solution. Difference between XWay & Xenloop is that, in XWay packet is traversed only upto Transpot Layer, after which it initiates shared memory based communication between sender & receiver. While in Xenloop, packet is traversed upto Network Layer of TCP/IP stack of guest domain, after which it initiates shared memory based communication between sender & receiver.

Xenloop is application as well as Kernel transparent. Xenloop is implemented as Kernel Modules. So neither application nor guest kernel need recompilation.

Xenloop provides automatic discovery of co-hosted VM’s. This is achieved by storing the [Domain ID,MAC Address] tuple in Xenstore at the time of Xenloop Kernel Module installation. Xenloop using GNU/Linux Netfilter hooks, captures every outgoing packet from Network layer. To determine the collocation, sender query the Xenstore for presence of [Domain ID, MAC Address] key. If query result is positive then sender initiate the bi-directional data channel with receiver, using shared memory. Xenloop facilitates domains to disconnect the data channel at any point of time. In such event Xenstore entry is deleted, domains can communicate with each other through normal network stack.

Xenloop supports migration. In event of migration from remote physical machine to local physical machine, dom0 intimates the collocation to domU & they can initiate the data channel for data transfer. In reverse scenario, when collocated machine migrates to remote physical machine, migrating VM can intimate the migration to other VM & remove entry from Xenstore & continue communication via regular network stack.

Xenloop is deployed in Xen 3.2 & GNU/Linux Kernel version 2.6.18.8. Limitation of Xenloop is that, it is implemented for IP V4 only.

2.4 Optimization of Network File System (NFS) Read Procedure

Avadhoot Punde, a 2008 batch alumni of IIT Bombay, worked on optimization of NFS as part of his thesis [15]. He was working on use of shared memory, for communication between NFS client & NFS server, where both, client & server are running on separate virtual machines, collocated on same physical host. [15] has contributed in NFS code in Kernel version 2.6.18. [15] has modified the NFS read procedure to use the shared memory for communication between client & server instead of using RPC. He successfully modified read procedure for communication between NFS server on dom0 & NFS client on domU. As per our communication over e-mail, he had some issue in shared memory based communication between both NFS client & server are running on 2 domU’s. Also Avadhoot has not modified the NFS write procedure.

In this implementation, client sends event notification by RPC call to server. This call traverses through TCP/IP network stack. Client send information of shared memory location (grant reference), file handle, amount of data to read. Once server receive this cal, it maps the corresponding memory pages & read the
data from corresponding file & copy it to shared memory. Returns the call to NFS client. Client read the data from shared memory pages & remove the corresponding entry from grant reference table.

[15] has implemented sharing of memory pages in three way.

1. **Dynamic Page Mapping** - Allocation of shared memory pages happens every time client want to read data from server. Server has to map the shared memory pages each time it receives the request.

2. **Client Side Static Page Mapping** - Client permanently allocates data & control pages. While at server side, server has to map the shared memory pages, each server receives the request.

3. **Client & Server Static Pages** - Client permanently allocates the shared memory pages. Server maps the shared memory when it first receives the request. After serving the request it do not unmap the shared memory pages. On subsequent request, it uses same pages to serve the read request.

[15] has reported the performance of above mentioned three methods of implementation. Among these three, third reports best performance for the obvious reason that, it saves time of mapping & unmapping of memory pages each time.

### 2.5 Fido

Fido [2] is in category of application transparent solution. Fundamental concept behind Fido is to decrease the degree of isolation between two collocated VMs. Fido is designed for the virtual environment in which all the domains on a physical machine are trusted. An example of trusted domains can be, all the application running on any of the domain on a virtual machine are designed & developed by same vendor. So according to Fido, in such trusted environment, a domain can open its entire memory area to other domains on same physical host. Since domains are trusted, they will not perform any malicious activity & will not cause any threat to security of domain.
Chapter 3

Solution Approach

We are planning to achieve solution to our problem in hierarchical order. First we are implementing the application specific shared memory based communication between two collocated virtual machine solution. Since [15] has already done some prior work on NFS, we have decided to choose NFS as our application. [15] has implemented NFS read procedure. We are extending it, to implement for NFS write procedure. Before implementation, we wanted to benchmark performance of NFS in virtualized environment. We will evaluate the performance of our implementation. We will compare performance of our solution with the benchmark of NFS. We aim to achieve the performance of at least equal to [18, 11, 17]. If we are not achieving the satisfactory performance, then we will analyze the possible bottleneck & try to improve upon them.

After implementing shared memory based communication in NFS, we will implement the same solution Linux Kernel.
Chapter 4

Background

In this section we will briefly study the Xen internals, which are of interest in our project.

We are using Paravirtualized Xen 3.3 in our project. As we have seen earlier, in Xen, dom0 is the privileged domain & guest are unprivileged domains. Calls made by any domain to any function/APIs of hypervisor are called as Hypercalls. These calls are equivalent to system calls in operationg syste. Other such equivalent term for xen hypervisor are listed in following Table 4.1

<table>
<thead>
<tr>
<th>Unix</th>
<th>Xen</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Calls</td>
<td>Hypercalls</td>
</tr>
<tr>
<td>Signals</td>
<td>Events</td>
</tr>
<tr>
<td>File System</td>
<td>XenStore</td>
</tr>
<tr>
<td>POSIX Shared Memory</td>
<td>Grant Table</td>
</tr>
</tbody>
</table>

Table 4.1: Unix Equivalent in Xen [3]

We will study Split Device Driver & Xen’s support to share memory among domains.

4.1 Xen Support to Share Memory Among Domains

Xen provides support for communication between two process running on two different domains by Shared Memory [3]. Xen support two basic operation on memory pages, sharing & transferring. This shared memory concept in Xen is called as Grant Table.

Since Xen deals with page table directly, grant table work at the page level. Whenever a domain shares a page using grant table, it uses grant reference API, which adds a tuple having domain id & pointer to location, into grant table. Grant table in turn returns an integer for reference of destination domain. Source domain needs to communicate this grant reference to destination domain. When the source domain wants to revoke the access to pages, it removes the corresponding entry from grant table. Xen provides various APIs to use grant table.

4.2 Split Device Driver

It is important to study split device driver in order to understand how communication between different domain happens in Xen.

In Xen, Linux device driver are at domain0. These device drivers are modified to handle the Xen events, which are equivalent of interrupts, from guest domains [3]. Henceforth we will call this modified device driver as Xen device driver. We will modify our Xen architecture digram of Figure 1.1 to Figure 4.1, in order to
understand the details of xen device driver.

Figure 4.1: Split Device Driver

Xen device driver are divided into four parts. They are as follows

1. Real Device Driver
2. Bottom Half of Split Device Driver (Back End)
3. Shared Ring Buffer
4. Upper Half of Split Device Driver (Front End)

Real device driver, which interact with hardware are available at dom0.

Bottom half of split device drivers, also called as back end drivers, are responsible for multiplexing of device among all the domains and provide generic interface to all the device like disk & NIC. These driver interact to with real device driver to do actual operation on hardware. Back end drivers notifies availability of data to destination domain by Xen Event.

Shared Ring Buffer, is a shared memory segment which is used for exporting the data from bottom half of split device driver to guest domains. Back end driver allocates shared memory pages & stores its grant reference in the Xenstore. All domains refers to Xenstore to get the reference to shared ring buffer.

At unprivileged domain, upper half of split device driver, also called as front end driver, on reception on Xen Event, queries the Xenstore for grant reference. Using this grant reference, it can retrieve the data from shared ring buffer.

All the I/O of guest domain has to follow split device driver. In case of inter-VM communication, domain0 does an optimization. Since domain0 is a privileged domain, it has access to memory pages of the domain which are communicating with each other, so domain0 directly copies the pages of source domain to destination domain, without actually sending the data to destination domain & just sends the Xenevent notification to destination domain.

4.3 Network File System

Network File System (NFS) [6] is file sharing protocol, in which files on a server can be allowed to access by authoritative client machines. NFS supports heterogeneous platforms. Clients imports the directory
shared by server on some mount point in its own local directory hierarchy. Client can perform normal file operation on this mounted directory.

NFS has four versions implemented. Version of our interest is NFS V3.

NFS at the backend uses Remote Procedure Call (RPC) for communication between server & client.

Details of NFS V3 specification are specified in RFC 1813 [16]. There are total 22 procedure that need to support specification. Out of this 22, only 2 procedure are of our interest, viz read & write. Read procedure executes the read operation on given file & returns the given number of bytes to client. Write procedure executes the write operation on given file & writes the given number of bytes on server.

There are two implementation of NFS, Kernel Level NFS & User Level NFS. Kernel level NFS is faster than user level NFS because of the fact that it require minimum context switching. So we will consider only Kernel level NFS,
Chapter 5

Ongoing Work

As discussed in Section 3, we are currently working on extending work in [15]. As part of this, we are working two threads, 1. Benchmarking of NFS Read & write procedure 2. Porting work in [15] to Xen 3.3 & GNU/Linux kernel version 2.6.27. We will look into details in following sub-section.

5.1 NFS Benchmarking for Virtualized Environment

In order to compare the results of our optimization in NFS, we are benchmarking the NFS Read & Write procedure without our modification.

Benchmark of NFS for non-virtualized environment can not be compared with the virtualized environment. Because NFS is a very high network & disk intensive protocol. In virtualized environment, network & disk activity has higher cost as compared to normal standalone physical machine scenario because of split driver, explained in Section 4.2. So performance of NFS in virtualized will be different from that of standalone physical machine scenario. So it is necessary to benchmark the performance of NFS in virtualized scenario.

5.1.1 Experiments

Our experiments has two goals.

1. We are interested in calculating throughput of the NFS, for read procedure & write procedure in virtualized environment.

2. We want to check how much NFS read procedure & write procedure affects the CPU usage.

In all experiments, NFS client & NFS server are running on virtual machines. Both the experiments are performed in two setup, first in which NFS Client & NFS Server are collocated on same physical machine, & second in which NFS client & server are located on different physical machines.

Our experimental setup has gone series of changes. First we used filesystem benchmarking tool, IOZone [5]. But this tool had issues with cache management of NFS. After IOZone, we used cp utility of GNU/Linux. Even cp was not able to handle the cache issue properly. Then we moved to custom C program which was using open() system call & O_DIRECT flag. Results that we have reported here are of C program. Experience with IOZone & cp is discussed in section 5.1.4.

NFS has an excellent caching mechanism. NFS has data cache at server side as well as client side. To disable effect of server side caching, we are using totally different, random file in each round of experiments, so that server should not find any current required data in its cache. Above it, at client side, NFS performs the Read-Ahead caching operation[13]. In Read-Ahead caching, NFS client prefetches the data from server. This is achieved by non-blocking, asynchronous read calls. When a process is reading from a file, the operating system performs read-ahead on the file and fills the buffer cache with blocks that the process will need in
future operations. So in future not all read may require to go to server, some read call may get the data from the local data cache. This optimization of NFS significantly improves the performance of NFS.

We are interested in disabling read-ahead optimization of NFS. Because this optimization is not going to give us the throughout of NFS in which NFS read operation are sequential & for each read procedure, client has to go to server to fetch data. One simple way to do this is, just disable the data cache at client side, so that client should not store the prefetched data in data cache. So to disable this cache, we are using open() system call of C with O_DIRECT flag. O_DIRECT flag disable the cache of filesystem. But in NFS, there is limitation with this flag. It can disable only client side cache & it can not disable server side cache. But as we already mentioned above, since we are using totally different, random files, NFS server side cache will not have any effect on our experiment results.

We have performed two types of experiments here. First experiment is a read experiment, second experiment is write experiment. In both experiments we are using a C program. For read experiment, our C program opens a file in read mode & read 32 KByte of data from a file using pread function of C. In write experiment, our C program opens a file in write mode & write to file from buffer of 32 KByte using pwrite function of C. In both experiment, we performing operation on record size of 32 KByte of data, because default block size of NFS is 32 KByte & for optimum performance from any file size, data record size should be equal to block size of the filesystem.

In each experiment we are varying the file size & we are measuring time required to copy & CPU utilization of all domains.

We are using time[9] utility of GNU/Linux to get the time required to copy a file, to measure the throughput. For measuring the CPU usage of the system, we are using xenmon.py[10] tool provided by xen-utils.

We are calculating throughput of read/write operation as

\[
Throughput = \frac{File\_Size}{Time}
\]  

(5.1)

5.1.2 Setup

We are using two physical machines. One machine physical is having Intel C2D E4500 2.2 GHz, 2 GB RAM. Second physical machine is having Intel C2Q Q6600 2.4 GHz, 2GB RAM.

NFS client & NFS server are running on separate virtual machines. All virtual machines are having configuration as 256 MB of memory, 8 GB of disk image. Each virtual machine is pinned to a particular CPU so that there should not be any interference among any of the domains. Disk image of all the virtual machines are on dom0 of respective physical machine.

5.1.3 Results

We will present our results in two subsection, in first subsection we will analyze the throughput, in second subsection we will analyze CPU usage.

Throughput

Graph of throughput of NFS read & write procedure for client & server running on collocated physical machine is as shown in Figure 5.1. Graph of throughput of NFS read & write procedure for client & server VMs running on different physical machines, is as shown in Figure 5.2.
Analysis of Graphs -

• In collocated NFS client & server VMs, throughput for write procedure is nearly two times higher than throughput of read procedure for file size upto 512 MBytes. Reason behind this better performance is Non-Blocking Write Calls. According to [6], NFS V3 implements the non-blocking write call, in which client continuously execute the sequential write call to write data on server, without waiting for acknowledgment of status of write from server. In case of read, after each read call, client has to wait for receiving data from server causing more latency in completion of read operation as compared to write.

• In collocated NFS client & server VMs, for read procedure, as the file size increases, throughput is decreasing gradually. This is because, as the file size increases, more number of page faults occurs, & system has to do more number of page replacement which penalize the throughput[1].

• In collocated NFS client & server VMs, for write procedure, after 512 Mbyte of file size, throughput drops by nearly 75%. Possible reason behind this drop is because of very large number of write call which causes bottleneck at the NFS server. According to [1], because of heavy load on NFS server, possible packet drop can be incurred at two queues, adaptive network driver queue & socket buffer queue.

• In NFS client & server VMs running on different physical machines, for both read & write procedure, graphs are straight linear curve.

• In NFS client & server VMs running on different physical machines, both read & write procedure has lower throughput than collocated NFS client & server VMs. Also for both read & write procedure, maximum throughput is limited upto 8.5 MBytes/Sec. Reason for this cap is backbone network. We have already studied in Section 4.2 that in collocated VMs, data is transferred from server to client via dom0 & no Network Interface Card (NIC) is involved. While in remotely located client & server, data is transported via NIC, which is having bandwidth of 100 Mbps. So maximum data that can be transferred is limited upto 100 MBits/Sec. So NIC become bottleneck here.

CPU Usage

1. Graph of CPU Usage for NFS read & NFS write procedure for client & server running on collocated physical machine, is as shown in Figure 5.3 & 5.4.
Analysis of Graphs -

- For read procedure, dom0 CPU usage is little higher than 2 time that of NFS client & server domU. This is because for each read call on server, server domU has to do disk read. And disk image of server domU is located on dom0. So dom0 has to do I/O processing each disk read. We have already discussed this I/O processing of domU in Section 4.2.

- For write procedure, CPU usage of server is much higher than client domU & dom0. Reason behind this increment is again asynchronous write call by client, which continuously sends data to server which has to queue all the request & process it. Here also dom0 CPU usage is higher that client domU CPU usage. This graph implies that, in NFS write operation server has to do more processing than dom0.

2. Graph of CPU Usage for NFS read & NFS write procedure for client & server running on different physical machines, is as shown in Figure 5.5 & 5.6. Here we are reporting the CPU usage for NFS client & NFS server domUs & their respective dom0s.
Analysis of Graphs -

- In both the graphs, CPU usage for client domU, server dom0 & client dom0 are nearly same. But for write procedure, CPU usage of server dom0 is higher than that of read procedure server dom0.
- In both the graphs, CPU usage of client0 & server dom0 is less than at least 10% as compared to corresponding graphs in collocated client & server scenario. This is because, in collocated client & server, dom0 has to do the processing of both client VM & server VM. But when client & server are on different physical machines, corresponding dom0 has to do processing of only one virtual machine.

5.1.4 Experience During Experiments

Initially we were trying to use the file system benchmarking tool IOZone [5]. According to IOZone remounting of NFS should clear the client cache. To verify this claim, we performed small experiment, in which, we were using same file & copying it from server to client repeatedly. And after each copy, we were unmounting & remounting the NFS directory. We observed that throughput for second copy was much higher than that of first round of copy. From this results we concluded that, this method do not clear cache. Graph of throughput using IOZone, for collocated client & server is as shown in Figure 5.7.

In the above graph, we can observe that throughput for NSF write procedure is decreasing to approximately

![Graph of Throughput for NFS Read & Write Procedure For Collocated Server & Client Using IOZone](image)

Figure 5.7: Graph of Throughput for NFS Read & Write Procedure For Collocated Server & Client Using IOZone

to level of what we observed in Figure 5.1. Here also reason for this drop is same, server is not able handle the larger file size due to buffer overflow. One important to note here is minimum

After IOZone we used cp utility of GNU/Linux. Initially we used to copy the file NFS mounted directory to local directory. So we observed that, this method not just performs the read operation from NFS directory, but also incurs the write operation on local filesystem. Which gives us incorrect results. To write on local disk, we copied the data to /dev/null, which did not incur any write & we can get only read throughput. But the problem with this overall procedure was that, still we were not able to control the caching at client side. So here we were not able disable Read-Ahead. So copying a file was not giving us correct throughput.

In order to tackle the problem of cache, we created random file of different size using /dev/random. /dev/random generates totally random numbers, but very slowly. It took around 13 hours to generate the 50 MByte of file. Using this 50 Mbytes of file we generate other various size files by using various utilities of GNU/Linux like split, cat & sed.

Copying a larger size file (16 GByte) was a problem. After copying 14 GByte of file, inode of the file used to get corrupt. We had to run fsck to correct the file system.
Capturing a CPU usage for very small file size is also a big issue. If latency of read or write is less than a second then none of the Xen tool reports the correct reading. Xentop reports the CPU usage at the granularity of 1 second. So it is impossible with xentop to record the CPU usage in millisecond interval. Xenmon reports the CPU usage at the interval of 2 millisecond, but then xenmon itself consumes a lot of CPU. For instance, say I am recording CPU usage at interval of 2 millisecond, then even without any other activity OS system, CPU usage shoots to 100%. For 500 millisecond interval, CPU usage is reported to be little more than 20% without any other activity in the system. So for the accurate snapshot of the CPU usage of the system, it is better to record CPU usage at the interval of 1 second.

5.2 Porting NFS Read Procedure

As discussed in section 2.4, [15] has implemented NFS optimization. But this implementation need porting to Xen 3.3 & GNU/Linux Kernel version 2.6.27 as there are few changes in the Grant Table APIs, also read procedure of NFS has many changes in it. So before implementing the Write procedure of NFS.

This porting is not completed yet. My initial attempt failed, and I am still debugging the errors. Lot of problems are being faced. Experience with porting is discussed in details in Appendix A.
Chapter 6

Future Goals & Time-line

• Completion of porting of NFS read procedure implemented in [15] to Xen 3.3 & GNU/Linux Kernel version 2.6.27. Since we have figured out the required changes in [15], it should require two weeks to port successfully.

• Implementation of shared memory based communication for NFS write procedure. This goal needs the in-depth study of NFS write procedure. This study should take two week. Actual implementation & testing of shared memory based communication for NFS write procedure should take two weeks.

• Perfor manace evaluation of optimized NFS read & write procedure. Since our method of evaluation is ready, this task should require one week.

• Comparison & analysis of performance evaluation with the benchmarks that we have presented in Section 5.1. This phase should require one week.

• If performance of our implementation is poor, then we will require to analyze the possible bottleneck in our implementation.

• Supporting migration in our optimization NFS. This phase will require design of architecture for migration. This will consist of design decision like how to identify migration, how to shift to TCP/IP stack etc. This phase will require This phase should take a month from design decision to implementation.

• Evaluation of migration support.

• Implementation & testing of shared memory based communication in Linux TCP/IP network stack. This phase should take one month to implement.

• Performance evaluation of implementation of shared memory based communication in Linux TCP/IP network stack.
Chapter 7

Conclusions

- In this report we have given introduction to Virtualization technologies & brief overview on Xen architecture. We have seen that performance of communication between collocated virtual machines is not optimum because of isolation constraint of Virtualization technologies. To improve upon this performance, we can use shared memory pages for communication to bypass the TCP/IP network stack.

- We are first working implementing shared memory based communication in NFS. We are porting work of [15] to Xen 3.3 & GNU/Linux kernel version 2.6.27. Also we will be extending [15], to implement the write procedure. We have setup the benchmark for NFS read & write procedure in virtualized environment. We empirically observed that caching is a big issue in benchmarking NFS. Caching at client side can be controlled using open system call & O_DIRECT flag. While at server side we should either use different file for each iteration of experiment or should use file size larger than physical memory.

- In virtualized environment if NFS server & client are Collocated, then it increases the CPU usage of dom0. This is because dom0 has to do the disk read/write as well as network communication activity. If we are using shared memory based communication, then we expect that CPU utilization should lower than our present because there will not be any network activity among between two virtual machines.
Appendix A

Experience During Porting

Since I was not familiar with the internals of GNU/Linux kernel, it took some time for me to figure out where to make changes & how to compile the changes.

Xen patched kernel requires different configuration files for dom0 & domU kernel compilation. Kernel compiled for dom0 do not work for domU. This is because this kernel has back end driver installed in it, while domU expects the front end driver, so at the time of booting, kernel panics.

To deploy my changes in NFS, I changed the NFS in kernel code & compiled it for domU. So we are using image generated by domU compilation for booting 2 virtual machine. If I have to use the NFS as client or server then I will have to compile the dom0 kernel with separately with changes in NFS.

For NFS to work, it require RPC & few other daemons [8]. RPC is installed as dependency while compiling & installing kernel. But daemons has to be installed separately. These daemons include portmap, lockd, mountd, statd. All these daemons are bundled in nfs-utils available on sourceforge [6].

While installing nfs-utils, initially even though compilation was successfully, still my mountd daemon was not running. Logs showed that there was error with respect to IP V6. I tried to change the code & removed the IPV6 implementation option from nfs-utils, but it failed. I was using latest version of nfs-utils, V1.2.3. According to a post on a mailing list on internet, there is a bug in nfs-utils V 1.2.3 & V 1.2.2 & V 1.2.1. So I tried with version V 1.2.0. It worked.
Bibliography


