**CREDIT SCHEDULING AND PREFETCHING IN HYPERVISORS USING
HIDDEN MARKOV MODELS**

A Thesis by

Vidya Rao Suryanarayana

Bachelor of Engineering, CMRIT, India 2006

Submitted to the Department of Electrical Engineering and Computer Science
and faculty of Graduate School of
Wichita State University
in partial fulfillment of
the requirements for the degree of
Master of Science

December 2010

© Copyright 2010 by Vidya Rao Suryanarayana
All Rights Reserved

CREDIT SCHEDULING AND PREFETCHING IN HYPERVISORS USING HIDDEN MARKOV MODELS

The following faculty members has examined the final copy of this thesis for form and content and recommended that it be accepted in partial fulfillment of the requirements for the degree of Master of Science with a major in Electrical Engineering.

Ravindra Pendse, Committee Chair

John Watkins, Committee Member

Krishna Krishnan, Committee Member

DEDICATION

In memories of my departed father, Suryanarayana Rao, who will always be remembered
and whose loss can never be filled

ACKNOWLEDGEMENTS

I would like to express my heartfelt thanks and gratitude to Dr. Ravi Pendse for giving me an opportunity to carry out the thesis under him. I would also like to thank Dr. John Watkins and Dr. Krishna Krishnan for their valuable time in reviewing the thesis.

I am thankful to Mr. Amarnath Jasti for his valuable input and his support throughout the thesis. I am especially thankful to Mr. Don Humlicek of the development team at LSI Logic for sharing his expertise in the area of current prefetching algorithms. I would also like to thank all my friends at LSI Logic for their valuable input in carrying out this thesis.

Most importantly, I would like to express my heartfelt gratitude to my family that has been very encouraging and patient throughout and helped me accomplish whatever I have. Without their encouragement and motivation, this work would not have been possible.

ABSTRACT

The advances in storage technologies like storage area networking, virtualization of servers and storage have revolutionized the storage of the explosive data of modern times. With such technologies, resource consolidation has become an increasingly easy task to accomplish which has in turn simplified the access of remote data. Recent researches in hardware has boosted the capacity of drives and the hard disks have become very inexpensive than before. However, with such an increase in the storage technologies, there come some bottlenecks in terms of performance and interoperability. When it comes to virtualization, especially server virtualization, there will be a lot of guest operating systems running on the same hardware. Hence, it is very important to ensure each guest is scheduled at the right time and decrease the latency of data access. There are various hardware advances that have made prefetching of data into the cache easy and efficient. But, however, interoperability between vendors must be assured and more efficient algorithms need to be developed for these purposes.

In virtualized environments where there can be hundreds of virtual machines running, very good scheduling algorithms need to be developed in order to reduce the latency and the wait time of the virtual machines in run queue. The current algorithms are more oriented in providing fair access to the virtual machines and are not very concerned about reducing the latency. This can be a major bottleneck in time critical applications like scientific applications that have now started deploying SAN technologies to store the explosive data. Also, when data needs to be extracted from these storage arrays to

analyze and process them, the latency of a read operation has to be reduced in order to improve the performance.

The research done in this thesis aims to reduce the scheduling delay in a XEN hypervisor and also to reduce the latency of reading data from the disk using Hidden Markov Models (HMM). The scheduling and prefetching scenarios are modeled using a Gaussian and a Discrete HMM and the latency involved is evaluated. The HMM is a statistical analysis technique used to classify and predict data that has a repetitive pattern over time. The results show that using a HMM decreases the scheduling and access latencies involved. The proposed technique is mainly intended for virtualization scenarios involving hypervisors and storage arrays.

Various patterns of data access involving different ratios of reads and writes are considered and a discrete HMM (DHMM) is used to prefetch the next most probable block of data that might be read by a guest. Also, a Gaussian HMM is used to classify the arrival time of the requests in a XEN hypervisor and the GHMM is incorporated with the credit scheduler used in order to reduce the scheduling latency. The results are numerically evaluated and found that scheduling the virtual machines (domains) at the correct time indeed decreases the waiting times of the domains in the run queue.

TABLE OF CONTENTS

Chapter	Page
1. INTRODUCTION	1
1.1 Literature Survey	2
1.2 Architecture of SAN	5
1.2.1 Importance of SAN	7
1.2.2 SAN in High End Scientific Computing	10
1.2.3 Benefits of SAN	11
1.2.4 Organization of Thesis	12
2. VIRTUALIZATION IN SAN	14
2.1 Storage Virtualization	14
2.2 Types of Virtualization	16
2.3 Shortcomings of Virtualization Techniques	19
2.3.1 Full Virtualization	19
2.3.2 Para Virtualization.....	19
2.3.3 Hardware Assist Virtualization.....	20
3. MOTIVATION FOR THE THESIS.....	21
4. INPUT/OUTPUT PERFORMANCE OF SAN.....	25
4.1 VMware ESXi Hypervisor: An Example Study.....	27
4.1.1 Architecture of ESXi.....	28
4.1.2 Storage Stack on ESX Server.....	30
4.1.3 Input/Output Pattern of ESX Server.....	32
4.2 Scheduling in a Hypervisor.....	33
4.2.1 XEN Hypervisor.....	34
4.2.2 Credit Scheduling in XEN Hypervisor.....	35
4.2.3 Event Channel Notification in XEN.....	39
5. GOAL OF THE THESIS.....	41
5.1 Current Data Prefetching Mechanisms in SAN.....	42
5.2 Mathematical Formulation of the Problem.....	44
6. HIDDEN MARKOV MODELS.....	48
6.1 Modeling a Hidden Markov Model.....	50
6.2 Hidden Markov Model in a SAN Environment.....	53
6.3 Proposed Hidden Markov Model.....	56
6.4 Training the Hidden Markov Model.....	57

TABLE OF CONTENTS (continued)

Chapter	Page
7. PATTERN CLASSIFICATION USING HMM.....	59
7.1 Prefetching in Hypervisor Versus Prefetching In Parallel Processors..	62
7.2 Hidden Markov Model with Gaussian Output.....	63
7.3 Gaussian HMM for Virtualization.....	65
7.4 Training the Gaussian HMM.....	68
7.5 Even Channel Notification and Scheduling in XEN Using GHMM...	70
7.6 Credit Scheduling Algorithm Without HMM	72
7.7 Credit Scheduling Algorithm With HMM	73
8. SIMULATIONS AND RESULTS.....	75
9. CONCLUSION AND FUTURE WORK.....	93
LIST OF REFERENCES.....	96

LIST OF FIGURES

Figure	Page
1. A typical storage area network.....	6
2. Levels of abstraction in a SAN.....	16
3. Binary translation approach to x86 virtualization.....	17
4. Paravirtualization approach to x86 virtualization.....	18
5. The Hardware assist approach to x86 virtualization.....	19
6. Mirroring in SAN between the primary and remote site.....	27
7. Architecture of ESXi.....	30
8. State transition diagram.....	55
9. Normal distribution plot for the arrival time of requests.....	67
10. Exponential distribution plot for the arrival time of requests.....	68
11. CPU fairness in a default XEN scheduler.....	77
12. Timeline of execution of domains with GHMM.....	81
13. Waiting times of the domains with and without GHMM.....	82
14. Classification time versus number of access patterns.....	84
15. Scatter plot of the log likelihoods of the access patterns.....	89
16. Time to fetch 32byte block with and without DHMM.....	89
17. Time to fetch 64 byte block with and without DHMM.....	90
18. Time to fetch 128 byte block with and without DHMM.....	90
19. Throughput of the system with and without DHMM.....	91
20. Response time with cache hits and cache misses.....	92
21. Cache miss time with increasing cache size.....	92

LIST OF TABLES

Table	Page
1. Arrival time pattern database for Gaussian HMM.....	78
2. Waiting time of the domains with and without GHMM.....	83
3. Read/write pattern database for discrete HMM.....	85

LIST OF ACRONYMS

CIM	Common Interface Model
CPU	Central Processing Unit
DCUI	Direct Control User Interface
DHMM	Discrete Hidden Markov Model
FC	Fiber Channel
HBA	Host Bus Adapter
HMM	Hidden Markov Model
GHMM	Gaussian Hidden Markov Model
I/O	Input / Output
IP	Internet Protocol
IT	Information Technology
iSCSI	Internet Small Computer System Interface
LBA	Logical Block Address
LUN	Logical Unit Number
OS	Operating System
RAM	Random Access Memory
SAN	Storage Area Network
SAS	Serial Attached SCSI
SATA	Serial Advanced Technology Attachment
SCSI	Small Computer System Interface
VM	Virtual Machine
vCPU	Virtual Central Processing Unit
VMFS	Virtual Monitor File System
VMM	Virtual Machine Monitor
WAN	Wide Area Network

LIST OF SYMBOLS

μ	Mu
π	PI
α	Alpha
Ω	Ohm
β	Beta
λ	Lambda
$\square$	Theta
Δ	Delta
Σ	Summation

CHAPTER 1

INTRODUCTION

Data prefetching has been a topic of research since the mid of 1960's and continues to gain interest from the research community, especially when high end technologies like virtualization, distributed computing, extensive data mining, remote accessing of data have become a necessity for the customers. Many of the research done in the past have indicated that the disk latency and access time can be reduced considerably by prefetching the data from disk in advance and storing it in a cache. With advances in storage technology like networked storage, hypervisor implementations, remote data access, mirroring, a need arises to efficiently prefetch the data, so that the data access time can be reduced considerably. The delay in accessing data can also be reduced by using efficient prefetching methods. When analyzed carefully, it can be observed that the total response time is not only a function of the read response time, but also a function of scheduling time. Hence, to reduce the total response time of a host, the scheduling time and the time for accessing the data need to be reduced. This is explained in more detail in Chapter 5.

There are many hardware and software prefetching methods used in the current implementations like using a hardware cache, using Markovian predictors with partial matching, Markovian predictors with some specialized hardware and the like. However, these techniques use expensive hardware and there is a lot of software overhead and computational complexity involved. Also, these techniques are mostly applicable to directly coupled systems where the hard disk is directly coupled to the host. A lot of research has also been done in prefetching sequential data where the data blocks requested by the hosts are sequential and near in space to each other on the disk. However, the research done focuses mainly on sequentiality and prefetching from the disk into the cache. But when the data access follows a random pattern, prefetching

becomes a complex task because the prefetched data may or may not be accessed by the hosts. Sometimes, when the data access is always random, it may not even be worthwhile prefetching the data into the cache.

In this thesis, the authors propose a data prefetching and scheduling mechanism using Hidden Markov model for server virtualized storage area networks wherein the disks are decoupled from the servers and are networked together in order to facilitate better storage management. Such an automatic classification and prediction of data access is very beneficial in many virtualized environments, where performance cannot be compromised. Such a prefetching mechanism can be applied in many scientific applications, wherein, even though the data access pattern is random, the access pattern is repeated over time.

The proposed method is mainly intended for use in server virtualized environments wherein multiple operating systems can be run on the same hardware and the prefetching and scheduling algorithms are proposed to be implemented on the server virtualization layer known as the hypervisor. It can also be extended to the regular SAN environment as well with suitable modifications, whenever the data access pattern has a repetitive nature as seen in many scientific applications [5]. The proposed method can be implemented on the hypervisor in a virtualized setting and on a SAN controller in a regular SAN setting.

1.1 Literature Survey

The work in this thesis is related to several previous work done in the area of pattern recognition and machine learning and is built on many of the techniques described below. The machine learning theory is exploited and is implemented in the area of storage area networks because of the data locality and block access pattern. The

hidden markov model is used to model the data access pattern in a Storage Area Network. The HMM is a statistical model used to model many complex processes and their probability distributions. HMMs can be either discrete or continuous depending on the output produced by the process. When the output of a process is discrete, it can be modeled as a discrete HMM and when it is continuous, it can be modeled by a continuous or a Gaussian HMM. HMMs are mainly used in the area of speech recognition where a continuous HMM is used because of the continuous nature of the sound produced.

In [1], Rabiner and Juang provide an extensive study of the theory behind the Hidden Markov Models and its application in speech processing. The authors of this thesis make use of the theory explained in this paper to build their models.

The algorithm proposed in [20] by P. Smyth can be used for more accurate pattern classification when several similar patterns belong to the same class. The algorithm exploits the proximity of each data point in a sequence and classifies the data more accurately. They make use of k-means clustering algorithms and Monte Carlo approximation techniques for the purpose of their research.

Another important application of the HMMs is in financial forecasting. HMMs are used to analyze the stock markets and predict the volatility of stocks and predict the closing price of the next day. An important contribution in this field is given by [10] proposed by Md. Hassan et al, where a Gaussian HMM is applied in the area of stock market analysis and the closing price of the next day is predicted based on the history of trained data.

In [2], Maria Fox et al study the HMM and its application in the area of robotics. They study the behavior of a robot by modeling its data parameters, which are acquired

through sensors. All the above mentioned work is related to areas such as speech recognition, stock market analysis and the like. However, very little research is done in applying the Hidden Markov Model in the area of virtualization in SAN.

D.Ongaro et al in [3] have studied the credit scheduler used in a XEN hypervisor and have proposed some changes used to improve the performance of the scheduler. They propose to sort the domains in the run queue based on their remaining credits and to place all the I/O intensive domains in the same domain to improve the performance. They also propose to reduce preemption of the even channel driver in order to reduce the latency and achieve a higher throughput.

T. Madhyastha et al in [5] have applied the Hidden Markov Model for automatic input/output access pattern classification which can help select the appropriate file system policies. They adopt the model for classifying the parallel and sequential file system I/Os. They compare the performance of the file system when Hidden Markov Model is used and when a neural network classification method is used.

In [4], H. Kim et al have developed an algorithm to improve the performance of a virtual machine monitor by incorporating inference techniques using gray box knowledge. The algorithm makes the VMM aware of the workload that each domain possesses using the knowledge gained by inference. The algorithm is developed to suit the needs of the scheduler which has to schedule unpredictable workloads seen in virtual desktop and cloud computing environments. They develop a partial boosting mechanism to boost the priority of a domain at the task level granularity.

In this thesis, the authors deploy a discrete Hidden Markov Model for classifying a read/write request by a guest domain in a server virtualized SAN environment and a Gaussian Hidden Markov Model for alleviating the scheduling delay and decrease the waiting times of the virtual machines in a XEN scheduler environment. The authors then use the HMM for predicting and prefetching the blocks of data residing on the disk into a cache so that the fetch time is reduced, thereby, enhancing the performance of the hypervisor. The authors then extend their work to optimize the performance of the credit scheduler used in XEN hypervisor by incorporating the Gaussian Hidden Markov Model. The proposed model not only helps in reducing the latency of scheduling but also provides a basic level of fairness of access of the domains in execution.

1.2 Architecture of Storage Area Network

Gone are the days when a lot of computers with directly attached disks were networked together to provide the users with convenience such as distributed computing, parallel processing, multi user support, multi thread support and the like. Even though such a technology is existent without getting completely obsolete, research in computer architecture has turned its direction to a relatively new and young technology which serves the above purposes in a much more resource efficient way. This emerging technology networks not only the processors, but even the hard disks hence increasing the capacity, reliability and consolidating the resources. The processors are interfaced with the disk array through a storage controller and physical layer such as fiber channel, iSCSI, SAS, SATA, Infiniband and the like with a switch as an intermediary, also known as a fabric. The networked hard drives are known as disk arrays. Such a network of servers and hard drives, together with the fabric and the physical interface is known as a

Storage Area Network, abbreviated as a SAN. A SAN can be represented as shown in figure1.

Figure 1. A typical Storage Area Network

The storage controller acts as an intermediary between the Host Bus Adapter (HBA) of the server and the disk array. It basically converts the signal from the HBA to a form understandable by the hard drive. The storage controller also has a firmware and NVSRAM on it that can speak the language of the hard drives. It also has device driver software on it which can interface with the hardware. The storage controller has other important responsibilities in addition to providing an interface. The most important function of a storage controller is to present the hard drives of the array as Logical Unit

Numbers (LUNS) to the application on the server. The hard drives are grouped together and presented to the hosts connected to the array in the form of LUNS, hence providing a layer of abstraction between the host and the array.

A LUN is actually a volume on the storage array that is presented to the operating system of the host as a device. A host can be provided with many LUNS, which it considers as seeing a lot of disks when actually the disk can be just one or a few, depending on how it is configured. In other words, many LUNS may represent the same target on the disk array. When the hosts are configured as a cluster, all the nodes in the cluster have access to the same storage volumes and hence the same LUNs. In such a case, the reads and writes must be well synchronized so that simultaneous reads/writes do not corrupt the data. When nodes are not configured as clusters, every host owns its set of volumes and those volumes cannot be accessed by any other host, even though they can have the same LUN number.

1.2.1 Importance of SAN

A number of large organizations like financial sectors, IT companies, hospitals, research organizations, banks have grown to a large number today when compared to 10-15 years back. Not only have the companies increased in number, but also in area. Almost all major IT companies have branches in various parts of the nation and some in various parts of the world. A single branch might span through a large geographic area with departments scattered. As an organization grows, the number of employees working for the organization also increases in number. This has not only triggered an increase in the resources made available but also in the efficient management of those resources.

The resources include anything that promotes the growth of an organization without which smooth management and running of an organization would not have been possible. The resources of an organization could include human resources, efficient management of their data, management of the long archived company documents, organization of e-mails, secured data access and protection of data, ensuring proper backup of the required and confidential data and the like. In a hospital, it is very important to manage the patient's data like the history of illness, MRI scans, CT scans, other images, appointment scheduling data, hospital administration related data and the like in addition to the personal data.

With advances in Science and Technology, research has taken a new turn. Research organizations like NASA, ISRO are finding new facts about the universe everyday that need to be stored for future references and correspondence. Most of these data show strong correlation that need to be processed and analyzed periodically. The artificial satellites that revolve round the planets generate such huge amount of data that a directly coupled system cannot store.

As seen above, it is very critical to protect and manage the exploding data efficiently. A server that has a hard drive attached to it directly is a poor manager of such large resources. Hence, researchers came up with the idea of decoupling the hard drives of all the systems used in an organization and consolidate them together and network the servers to the array of hard drives. This way, lots of data can be stored together and any server can access any of the hard drives based on the access policies set.

In a business sector, by deploying a SAN, an engineering department can access the required data of a manufacturing department even when the manufacturing department servers are not on, provided they are granted access to the data. Note however, that the engineering and manufacturing departments need not be in the same geographical location. Let us say, the engineering department is in California and the manufacturing department is in New York, the two departments can still access their data on the disk array by the utilization of protocols like fiber channel, iSCSI, SAS, SATA, infiniband and the like.

As we can observe, there are a lot of advantages of using a SAN in an organization. But, with every advantage comes some disadvantage. The advantages offered by a SAN needs to be actually traded off for the price and access time. There is no doubt that the capacity of the RAM and hard drives are growing tremendously. But, it is also equally important to ensure that the I/O operations on the hard disks are fast enough. If we consider the above company X, that has its engineering department in California and the manufacturing department in New York, the time required for the engineering department to access the manufacturing department information would definitely consume a lot of time.

Virtualization in SAN is the most researched topic of today in addition to other new technologies like cloud computing, security in SAN that have also gained momentum. The virtualization technology in SAN is explained in detail in Chapter 2. Hence, there is a need to improve the performance of a SAN in order to make it more reliable and robust. In this work, the authors propose algorithms to reduce the disk access

time and the scheduling latency involved in a virtualized environment, in particular the server virtualized setup.

1.2.2 SAN in high end scientific computing

Interestingly enough, SAN and science have become so inter dependent that they co-exist in every facet of modern day computing. Scientific applications have become so computing intensive and data explosive that all the modern day advances in science would have failed if there wasn't a way of storing the explosive data of the past, present and future. High tech innovations of software and hardware technologies have called for high performance computing environments involving the intervention of super computers and intelligent high performance algorithms [18]. High speed data storage, data mining and computational complexities are a necessity in these applications and a lot of research is going on in these multiple facets of science.

The high tech scientific computing companies like geophysical organizations, oil and gas companies, and chemical 3D-imaging companies are at stake if they do not invest the dollars necessary to store their terabytes of explosive data [18]. According to [18], seismic monitoring of even a small square feet of area generates about 20 terabytes of data. The generated data needs to be stored in order to further interpret and analyze. Many times, the same massive sets of data need to be processed again and again. This type of intensive data processing requires more intelligent algorithms that can provide a higher speed of access for workloads ranging from sequential to random without compromising the performance.

Is it not surprising to hear that the oil and gas companies now need a high performance and high computing platform? We can now imagine the extent to which data processing and data mining has grown. It is further interesting to note the connection between the oil and gas companies and the seismic data. Recent research has found that the structure and composition of the seismic data, certain patterns of fractures found in the seismic data affects the drilling of oil wells! Another important data intensive application is in the weather forecast. The temperatures throughout the globe are changing, ice capes are melting due to global warming. It is needed to forecast the weather for the next few days for safety purposes. All these applications demand intelligent storage because of the massive data and also need high speed algorithms for data access.

Such companies are exploring means and are beginning to use GPUs instead of CPUs. The CPUs are most suitable for sequential applications. But the above applications need a processor that can efficiently process data ranging from sequential to random. GPUs are the graphical processing units which are highly efficient in managing parallel and 3-D imaging data than a regular CPU. This demands the invention of more sophisticated algorithms that facilitates the data storage and data mining capabilities in such scientific applications.

1.2.3 Benefits of SAN

With the advance in research and development in a Storage Area Network, managing large volumes of data which was a challenge before has now become very simplified and easy to manage. Not only does SAN provide efficient data management, but it also provides other advantages as explained below:

- 1) Long distance SAN can provide access to important data always.
- 2) The connectivity in a SAN can provide backup for the data on the disks in case of application or hardware failures. This is provided by the multi path capability provided by SAN devices.
- 3) Reduces the heavy traffic in a network by providing separate channels for the data and the control traffic. The I/O traffic is traversed separately over a fiber channel, SAS or SATA network and the other control traffic is traversed through a regular IP network.
- 4) Enables cost saving by avoiding the use of extra disks and servers. Also enables cost saving by cutting down the resources (manpower, management) needed for migrating data from one server to the other.
- 5) Provides resource consolidation. By using clusters, virtualization techniques, application and resource failures do not affect the ongoing operation and still provide connectivity and data availability.

1.2.4 Organization of Thesis

Chapter 2 of the thesis discusses the virtualization technologies in SAN like block virtualization, server and network virtualization. It also discusses the full and para virtualization techniques deployed in today's SAN and their shortcomings. Chapter 3 is dedicated to the motivation of this thesis. Chapter 4 studies two hypervisors in today's market like the VMware and XEN and identifies the performance bottlenecks in them in terms of input/output and scheduling. Chapter 5 explains the current data prefetching algorithms used in SAN and develops a mathematical representation of the latency problem in SAN. It also discusses the goal of the thesis. Chapters 6 and 7 explain the

Hidden Markov Model and its incorporation into the SAN to reduce the performance problem in terms of latency. Chapter 8 explains the test setup and the simulation along with the results. Finally, chapter 9 discusses the future work that needs to be done in this area and also concludes the observations that were made during the research.

CHAPTER 2

VIRTUALIZATION IN SAN

Virtualization refers to the abstraction of the resources. Many kinds of virtualizations exist like memory virtualization, system virtualization, network virtualization and the like. In memory virtualization, the host sees more memory than the actual physical memory by making use of additional address translations. The host issues a virtual address during memory reads and writes, which is then converted into an actual physical address by an underlying hardware or software.

In a server virtualization, many operating systems are made to run on the same underlying physical host leading to better resource utilization. All the hosts running on the machine share the same underlying hardware like physical memory, printers, processors etc...without them being aware of the same.

In network virtualization, the network software and hardware like switches, routers are abstracted by providing a network software administrative entity residing on each host in a network or in the network itself [19]. This kind of virtualization separates the actual data from the control data, which makes management an easy task and also enhances the security of communications.

2.1 Storage Virtualization

A Storage Area Network is a network of storage devices and hosts where the storage disks are isolated from the hosts. Such a network is very useful for managing large volumes of data and is gaining momentum in today's world because of the convenience and advantages it provides to its customers. A storage area network allows

the hosts in the network to share the physical storage devices by providing a layer of abstraction between the hosts and the storage arrays. The hosts are unaware of the physical location where the data is stored and the physical path between the host and the storage.

Such a virtualization technology provides many advantages like better resource utilization, universal storage connectivity, remote storage and access of data, dynamic changing of data location depending on storage availability, implementation transparency and many more without the host being aware of the changes [19].

In a SAN, virtualization can take place at the host, at disk level (memory virtualization or block virtualization) and at the network level (network virtualization). In disk based virtualization, logical block addresses (LBAs) are used by the hosts to address the disk, which is then converted to a physical address on the disk by the disk array controller. Such a virtualization is implemented on the storage device.

In a host based virtualization, the logical volume managers reside on the hosts that perform functions like adding or deleting physical drives, concatenating and resizing the logical volumes shared by the various hosts in the network, providing snapshots of data for data recovery in case of failures and managing the logical volumes and physical drives on the whole [19].

In a network level of virtualization, abstraction takes place at the network layer, providing transparency of the interconnecting links and paths. A software entity resides on the network interfaces on the hosts or in the network to serve this functionality. Figure 2 provides an overview of the levels of abstraction in a SAN.

Figure 2. Levels of abstraction in a SAN

2.2 Types of Server Virtualization

Server virtualization in SANs can be of three main types:

- 1) **Full virtualization:** The full virtualization technique virtualizes the host and many hosts are made to run on the same underlying physical hardware without them being aware of the virtualization. This is done by using a technique known as binary translation. In this method, the kernel code of the OS replaces the non virtualizable instructions with new sequences of instructions that make virtualization possible. The user level code is directly executed on the processor. Every virtual machine running on the host has a virtual machine monitor that supplies all resources to the virtual machine like virtual BIOS, virtual memory management and other virtual devices.

Since there are many guest Oss running on the same machine, the virtualization layer abstracts the guests from the actual underlying hardware. The guest OS does not know that it is being virtualized and assumes that it is directly coupled to the physical hardware resources. In actuality, the resources are being shared by many such guest Oss. Examples of full virtualization include the VMware ESX server, Microsoft virtual server. Figure3 shows the full virtualization mechanism.

Figure 3. Binary translation approach to x86 virtualization

2) **Para virtualization:** This technique is also known as the operating system assisted virtualization. In this technique, the guest OS is aware that it is being virtualized and the guests communicate with the virtualization layer known as “hypervisor” in order to increase the performance and efficiency of the system. In this method, the non-virtualizable instructions are replaced with “hypercalls” that communicate directly with the hypervisor. The hypervisor provides interfaces for the guest Oss for memory management, interrupt handling and the like.

The difference between full virtualization and paravirtualization is that in a full virtualization, the non virtualizable instructions are converted into virtualizable

instructions without the guest OS knowing that it is virtualized. Whereas, in a paravirtualization, the OS kernel is modified and the guest is aware that it is virtualized. The XEN open project is an example of paravirtualization where the CPU and memory is virtualized. Figure 4 shows the implementation of paravirtualization.

Figure 4. Paravirtualization approach to x86 virtualization

3) **Hardware assisted virtualization:** The hardware assisted virtualization is a technique wherein the vendors develop hardware features that make the virtualization techniques more efficient. For example, Intel and AMD have developed enhancements wherein the CPU can run in a new CPU execution mode that allows the execution of virtualized instructions and allows the virtual machine monitor (VMM) to run in a root mode. Figure 5 shows the implementation of such a technique.

Figure 5. The hardware assist approach to x86 virtualization

2.3 Shortcomings of Virtualization Techniques

2.3.1 Full Virtualization

While the full virtualization technique is the best suited and secure technique, it has a main drawback of overhead. Every resource request from the guest machine needs to go through the VMM or hypervisor and hence, there is a lot of overhead on the hypervisor. Also, if there is a request for disk read/write operation, the request is routed through the VMM, which then validates the request and then transfers the requested data back to the guest OS. This could mean an additional delay along with the normal response latency and transfer time. Hence, data prefetching and efficient scheduling can help to alleviate the delay in such a case. The models developed in this thesis can be employed in such a real world setting to reduce the overall data fetch delay.

2.3.2 Paravirtualization

Unlike the full virtualization technique, where the hypervisor completely intermediates between the guest and the physical hardware, the para virtualization technique has a lower communication overhead. In that, the guests are already aware that they are virtualized and need modifications in the guest device drivers. Also, the

performance of the para virtualized setup is dependent on the workload of the guests. This leads to instability in performance swinging between the two ends of spectrum in terms of workload. Hence, a lot of research is still pending to improve the performance of such a technique for any kind of workload. A common platform still needs to be developed which would make portability and interoperability between vendors feasible and successful.

2.3.3 Hardware Assist Virtualization

The hardware assist techniques greatly help in simplifying virtualization at the hardware level. A lot of companies have already become successful in this yet to grow field. Intel and AMD are the two leaders in the hardware assist virtualization field. However, since this technique is very new, it has issues of portability since the older machines cannot use these techniques.

CHAPTER 3

MOTIVATION FOR THE THESIS

While there is a lot on ongoing research in improving each of these advancing SAN virtualization technologies, a common research platform that can enhance the performance of any of these techniques in general needs to be focused upon. For example, it would be very useful if there was some kind of an independent software that can be ported to any of the above virtualized platforms (fully virtualized, para virtualized) without compromising much of the important features but in turn will assist in enhancing the performance of the same. Such software can be incorporated into a fully virtualized or a para virtualized platform and can still provide the functionality that it is intended to provide.

Such a software program can be run as a separate thread on top of the existing setup or it can be incorporated into the existing code and can work together with the existing software to fulfill the responsibilities. The authors felt that this is an important entity that needs to be focused upon because there are lots and lots of programs that work for a specific kind of environment. But, very less research is done in developing techniques that is scalable and portable to any platform. However, this is not an easy task to accomplish because of the varying nature of programming languages used, different kinds of interfaces used, different functionalities involved and different tiers involved in the platform.

Especially in platforms such as virtualization, where the various drivers and kernels are modified, translation of instructions are involved, such a portability issue is predominant. The authors have researched on some of the common bottlenecks found in

the server virtualized platforms and have proposed mechanisms that are targeted in improving these bottlenecks. The advantage of this is in building software that can be run on any type of hypervisor like the ones developed by VMware or XEN and can achieve its goals.

During the research, the authors found that performance compromises are dealt with in any kind of hypervisor, especially when varying amounts and types of workloads are involved by the guests. The authors also found that, the number of guests running on the hypervisor adds to this compromise in terms of latency. This is because of the scheduling latency involved in scheduling the guests to run and the number of I/Os by the guests. The guests can be classified into two types depending on their workloads: they are either I/O intensive or processor intensive.

While the I/O intensive domains consume very less processor time, they are of a higher priority than the CPU intensive domains. This fact is indicative of the necessity for good scheduling and good algorithms for prefetching and caching. When observed and analyzed carefully, we can observe that these two problems are statistical in nature. Especially, when the nature of requests is repetitive over time, good statistical solutions are the best fit in improving these issues. The authors have mainly focused on improving the latency of access and scheduling in this research, which is why they use statistical approaches to reduce the latency involved.

Statistical analysis is not a new topic and is very frequently employed in research from good old times. However, the application of statistics to particular areas and developing new algorithms to suit the needs of the applications is what is gaining momentum. Statistical analysis very often involves the analysis of the data and its trend

in the future. The authors felt that the statistical approach very well suits their application because, if the requests by the domains are analyzed and extrapolated to view its trend in the future, then data can be prefetched probabilistically and domains can be scheduled well in advance leading to improvements in latency. Not only does it reduce the delay involved, but it also increases portability of code on platforms because, the analysis is statistical and general in nature and can be applied to anything, anywhere.

For example, wouldn't it be nice to know when the next customer might arrive and free a table at a busy restaurant? This would reduce the time that the customer needs to wait and will pave a way for the other customers. Similarly, it would be nice to know when the next car might arrive and have a ticket already ready at the toll booth. All these are examples of scheduling before hand in order to reduce the waiting time. Similarly, if we knew which domain and when the domain might request I/O next, we can schedule it accordingly to reduce the wait time of the domain. Also, if we knew which block of data, the domain might request next, we can prefetch and store the data in the cache.

If we observe carefully, the above examples involve at least two entities. One of the entities is not visible directly and needs to be determined, while the other entity/entities are observable. For example, we do not know which car will arrive next at the toll booth, but we know when a car might arrive. Or, we do not know which domain will request I/O next, but based on previous statistics, we know approximately when a request might arrive. Similarly, we do not know what block of data a domain might request next, but we know if the domain reads data from the disk or write data to the disk. Thus, the hidden entity is unknown and needs to be determined from the known entity.

The above problem is a very typical example of a Hidden Markov Model (HMM). HMM is a statistical and a doubly stochastic model that is used to determine the unknown entity probabilistically. Hence, the authors decided to use a HMM for their research. There are several other such statistical models used for classification and prediction namely neural networks, support vector machines, decision trees. However, those models are very complex and demand a lot of intricate details of the application. However, the authors found a HMM to best suit their application and is the most simple of all the other models. The details of HMM and its application is discussed in detail in the Chapter 6.

CHAPTER 4

INPUT/OUTPUT PERFORMANCE IN SAN

The key components of a good and efficient SAN are easy management of data, high throughput and providing efficient backup and failover. The SAN should also be scalable while its performance is not compromised. For the purpose of this thesis, the authors concentrate on the performance of a SAN, in particular, the I/O performance of a SAN.

As we have seen, a SAN consists of a large number of hosts connected to a large number of arrays through many fabrics. In a big enterprise with a lot of branches, the hosts, arrays and fabrics may be distributed throughout a large geographic area. The key operation in a SAN is the data read/write. Data access from/to a disk array should be very quick in order to reduce the latency of access and increase the throughput. Since a SAN is a scalable network, many more hosts and arrays can be added to the network. Even in such circumstances, it is very important not to compromise the performance. Delay in data read/write can become a major bottleneck of a SAN because it affects the transfer of data from the primary to the backup in disaster recovery circumstances. It also affects the scientific applications running on the hosts. It increases the round trip time between the hosts and the arrays and compromises application performance.

Hence, it is very important to devise methods and algorithms to reduce the latency of data access. In Fiber Channel SANs, separate data and management paths are provided for transferring the I/O data and the control traffic respectively. One of the well studied methods to reduce the I/O latency is prefetching. There are several prefetching

hardware and software mechanisms currently used in the CPU, caches of the processors that are used in the directly coupled architectures. Prefetching and caching are also enabled on the storage controllers. This thesis models a software prefetching mechanism that reduces the I/O latency in a SAN, particularly in the virtualization arena whenever the pattern of data access is repetitive in nature.

Let us consider a SAN scenario as shown in the below figure 6. The two ESX hosts are connected to an array through a SAN fabric at the primary site. There is a remote site consisting of two servers connected to two storage arrays and the two PCs are running some application. Both the setups are connected to each other over a WAN. Let us assume that one of the identical setup serves as a backup to the other setup. For reliability purposes, it is desirable that the contents of the primary site be copied to the remote site that serves as a backup. Hence, there is a mirroring relationship that is setup between the primary and the remote array. In the server virtualized setup, let us assume that the guest operating systems are reading/writing data from/to the array attached to the host. In this case, the guest domains should first send their request to the hypervisor which then grants access to the guest OSs to access the disks.

Figure 6. Mirroring in SAN between the primary and remote site

In all the above scenarios, we can see that reading from or writing to the array can be very time consuming because of the latency between the sites or because of the intervention of the hypervisor. This can be a major bottleneck in time critical applications like scientific applications. As we can see, even though the disk capacity has undergone a major and positive revolution, the slow access speeds can become an impediment to a robust application. Hence, there arises a need to increase the speed of the memory and disk accesses.

4.1 VMware ESXi Hypervisor: An Example Study

As discussed in the Chapter 2, a hypervisor is a thin layer of software that resides on top of the operating system of a virtualization host that acts as an intermediary

between the guest machines (virtual machines or VMs or domains) and the underlying hardware. Any VM that needs to access a physical resource such as memory, disk array, printers has to send its request to the hypervisor, which then grants access to the resource. Hypervisor is implemented in a fully virtualized environment wherein the VMs are not aware of the abstraction mechanism. The VMs are made to realize instead, that they are directly connected to the underlying hardware. Such an implementation is an example of a centralized architecture where the control lies in the hands of a hypervisor to grant access to the VMs.

In this section, the authors study the architecture of an ESXi hypervisor, which is a server virtualization software implemented by VMware, a leader in the storage virtualization software industry. We then explain how the virtualization software can be more efficient in the server virtualization scenario. The ESXi hypervisor is equivalent to the VMware's ESX3 hypervisor, except that it has enhanced security features and more simplified management. The ESXi hypervisor is an operating system with a kernel known as VMkernel.

4.1.1 Architecture of ESXi

Various kernel processes run on the hypervisor such as management processes that manage the application, virtual monitors (guest OSs) that run on the hypervisor. The hypervisor has full control of the hardware resources of the physical server and hence manages the resource allocation applications requested by the guest operating systems. The process that controls the virtual machines running on the hypervisor is called the virtual machine monitor. This process runs on the hypervisor kernel [12].

The Direct Control User Interface (DCUI) is an interface process that allows low level basic configuration and management of the virtual machines. The Common Interface Model (CIM) is another process running on the kernel that allows hardware level management from remote applications through APIs [12].

The VMkernel is the operating system that provides the same functionalities like any other operating system (OS). Some of the features provided by VMkernel are the creation of processes, resource scheduling, creation and management of file systems, device drivers for controlling the underlying hardware. The VMkernel supports file system known as the VMFS that stores the virtual machines.

The resource scheduler in the kernel is the software responsible for scheduling the request for hardware access by the various virtual machines. The resource scheduler of a XEN hypervisor is explained in detail in section 4.2. The architecture of an ESX server can be represented as shown in Figure 7.

Figure 7. Architecture of ESXi

4.1.2 Storage Stack on ESX server

The ESX server hosts many guest operating systems that run on top of it. The guests have the virtual adapters to which they send their input/output request. The virtual adapter also acts like a virtual device driver for the guest OS. The Virtual Machine Monitor (VMM) wraps these I/Os and passes it on to the virtual SCSI layer of the Virtual Machine Monitor kernel (VMkernel). These I/Os are very helpful in detecting the pattern of the application requests running on the guests. The I/O pattern forms a very important entity in implementing this thesis and will be discussed in Chapter 7. The I/Os reach the VMkernel directly through the SCSI layer or through the Virtual file system provided by the hypervisor known as the Virtual Machine File System. The I/Os are then queued at the SCSI layer. The above mentioned queue is a part of the VMkernel.

The I/Os issued by the guest are then fetched from the VMkernel queue and given to the device drivers. There are a number of queues present at various levels in a server virtualized setup. There are queues present at the guest driver level, then come the VMkernel queues, then there is the LUN queues and finally the device adapter queues. The I/O requests should pass through all these queues before it gets serviced. Because of this, there is a considerable overhead on the performance of the hypervisor in servicing these requests.

This thesis mainly focuses on the performance overhead due to I/O requests from the guest to access the disk array. It mainly focuses on reducing the latency involved in scheduling the virtual machines and accessing the disk array by classifying the I/O requests and then prefetching the next block of data that a guest might intend to read. This type of prefetching is also known as look ahead prefetching and utilizes various algorithms to effectively prefetch the data.

A lot of research has suggested that the queuing depth should be varied dynamically depending on the requests that can help in improving the performance. The queue statistics on an ESX server can be read by using the “esxtop” command [14]. The esxtop command helps in analyzing the performance and the QoS provided by the hypervisor. It generates such statistics such as disk statistics, memory statistics, and performance statistics. It provides a lot of information on the arrival time of a request, the time spent by the request waiting in the queue for it to get serviced, the response time taken, the total round trip time required for the system to respond to the application running on the guest, the scheduled requests, the guest IDs (also called “worlds” by VMware).

When there are I/O requests waiting in the queue from more than one guest, the scheduler tries to schedule the requests and grant access to the disk array based on various algorithms such that the requests are fairly serviced. The algorithm used by VMware for scheduling is not a topic of this thesis and hence is not explained. The thesis studies the credit scheduling algorithm used by XEN, which is another leader in server virtualization like VMware and tries to exploit the statistics generated at the scheduler to classify the I/O requests from the various guests and prefetch the data blocks from the disk. The statistical information is modeled using the Hidden Markov Model (HMM) and is discussed in chapter 5.

4.1.3 Input/Output Pattern of ESX Server

According to VMware, an ideal environment for virtualization is that which uses low CPU and low disk, a good candidate is that which uses low CPU but high disk [11]. According to their study and a lot of other research, an I/O stream will require “long disk seeks” quite rarely. This means to say that most of the seek operation depicts a nearness in space, wherein the requested blocks of data are present contiguously usually and also a nearness in time, wherein the application many times requests the adjacent blocks at time instants $t, t+1, t+2, \dots$

When two random I/O streams are mixed, the resulting stream is still random [11]. It is the same case when one random and one sequential stream is mixed and also when two sequential streams are mixed. But many server applications do not require sequential streams. The storage arrays use many caching algorithms and heuristics to mix arbitrary I/O streams. Since the I/Os from the various guests are queued, a need arises to properly order them for optimization purposes and schedule them properly. To increase

the performance of the I/O system, various algorithms and heuristics should be used to read ahead of the request coming in and cache the blocks read. The main idea of caching is to maximize the number of hits by increasing the likelihood of the requested block or page to be present in the cache. This functionality of reading ahead is not implemented in the ESX hypervisor and is taken care of by the storage array and the guests running on the ESX server.

The HMM model proposed in this thesis is intended to be implemented on the hypervisor in a server virtualized environment or on the controller in a regular SAN attached setup in order to reduce the overhead on the guests and on the array. By dedicating a separate module or a separate thread on the hypervisor or on a controller for the HMM, local and global requests can be classified based on the various heuristics collected at the hypervisor kernel or a storage controller. The idea proposed also includes a common cache for the guests to access and in order to store the prefetched blocks. We also need a dedicated space for each guest OS in order to store the history of accesses of the guest.

4.2 Scheduling in a Hypervisor

In a virtualized environment, the hypervisor is an important entity between the guests and the hardware. Any request by the guest to access the hardware is routed through the hypervisor and hence, it is very important for the hypervisor to ensure that all the guests are given fair access to the hardware. This way, the hypervisor provides a level of Quality of Service to the guests. Such a QoS mechanism is provided by scheduling the guests appropriately. A scheduling mechanism guarantees that a guest performs its transactions in its scheduled time and this way, all the guests get a fair chance to access

the resources. While this kind of a scheduling provides fairness in access, it trades off the latency of the access. Since the authors are mainly concerned with reducing the latency of access, they study the scheduling mechanism in a XEN hypervisor and propose an algorithm to improve the latency response in a hypervisor using HMM while still providing fairness of access.

4.2.1 XEN Hypervisor

XEN provides a paravirtualized platform for the guests to run. As described before, in a paravirtualized platform, the guests are aware that they are virtualized and the guest kernel is modified in order to achieve such a level of virtualization. The guests in XEN communicate with the Virtual Machine Monitor (VMM) through a set of interfaces known as “hypercalls” [7][13]. In addition to the guest domains that are hosted by XEN, an additional management domain called the domain0 is given charge of maintaining the other domains. XEN also hosts another domain called the “driver domain” that provides the guests with access to the actual physical hardware. Since the management functions are distributed to the various special domains running on XEN, the overhead on the hypervisor is considerably reduced.

Whenever a domain has an I/O request or any kind of a request, it generates the request in terms of ‘events’. The events are transferred from the backend driver to the front end driver also known as a “pending array” [3][4]. After the domain is scheduled by the hypervisor, the event is delivered to the target domain [4]. This time of delivery of the event to the target plays an important role in determining the latency of response. Hence,

the faster the domains are scheduled the lesser the time that they need to wait for the response.

Currently, XEN supports three types of schedulers namely, Simple Earliest Deadline First (SEDF), Borrowed Virtual Time (BVT) and the Credit Scheduler [17]. In an SEDF scheduler, the domains request a minimum time slice that require for communication. The request is a pair of (slice, period) where the domain requests for 't' slices every 'p' periods of CPU time [4]. The BVT scheduler is a proportional share scheduler that is suited for I/O intensive domains [17]. The scheduler when specified with the correct parameters adjusts itself dynamically with the varying I/O intensities. The credit based scheduler is recently incorporated into XEN and provides better load balancing and low latency mechanisms [17]. The authors propose an algorithm to improve the latency of a credit scheduler when there are a lot of I/O domains performing disk reads and writes. Their algorithm schedules the domains statistically using a Hidden Markov Model. The algorithm mainly relies on the arrival times of the requests from the domains and tries to classify them into one of the previously seen patterns. The details of the algorithm are discussed in the Chapter 8.

4.2.2 Credit Scheduling in XEN Hypervisor

The XEN hypervisor uses a credit scheduling mechanism to schedule the guests running on it. As the guests request access to the hardware, they are put in a run queue and depending on the type of scheduling algorithm used, the guests are run in a particular sequence. For example, in a round robin scheduler with time slicing, each guest is arranged in the run queue and run in a First Come First Serve (FCFS) basis and each

guest is allowed to perform I/O for a specified time by taking turns. This ensures fair access for every guest.

A credit scheduler works as follows: An administrator initially assigns credits to the guests and every time a guest runs, it utilizes its credits. If all the guests are assigned the same number of credits, then every guest receives an equal amount of CPU time as every other guest.

An accounting thread on the XEN hypervisor debits 100 credits from the domain at the head of the queue every 10ms. Every 30ms, the weights of the domains are recalculated and the domains are scheduled accordingly. If a domain has run lesser than 10ms, it does not have any credits debited. Any domain is in any one of the states, either 'UNDER' or 'OVER'. A guest in the 'UNDER' state still has credits in order to run and a guest in 'OVER' indicates that the guest has utilized all its credits and does not have any more credits to run. Such a guest has negative credits. The time on the processor that a guest obtains is determined by the number of credits that the guest possesses. The more credits a guest possesses, the greater time it is given to execute on the processor.

The guests are arranged in the run queue according to their states and not the credits. This means that a new guest enters from the rear of the queue and all the guests in the same state are grouped together. Specifically, the guests that are in the 'UNDER' state are at the front end of the queue and the guests in the 'OVER' state are pushed to the rear of the queue. The 'OVER' guests are not allowed to execute till they receive positive credits. It is important to note that the guests in the same state are not ordered in ascending or descending order of credits and are purely ordered based on the FCFS basis.

While this type of scheduling mechanism provides fair access to all the guests, it has some drawbacks. Applications running on the guest domains can be either processor intensive or I/O intensive. While the processor intensive domains need a lot of time and CPU resources, the I/O intensive domains can be given a higher priority as they are usually not CPU intensive. In the traditional XEN scheduler, an I/O domain is given the same priority as a processor intensive domain. Hence, if an I/O domain is at the end of the run queue, it has to wait for its time slice in order to execute. In order to overcome this drawback, XEN added another state called the “BOOST”. Any domain which enters the BOOST state tickles the scheduler and enters the head of the queue directly rather than entering from the rear end of the run queue. This prioritizes the BOOST domain over the other domains. Whenever a domain enters the BOOST state, the other currently running domain is pre-empted and the BOOST domain is allowed to execute.

The BOOST mechanism works pretty well in XEN but with a few challenges. Let us consider a scientific application scenario where most or all of the domains are I/O intensive. In this case, the BOOST state does not work very well and is as good as the XEN scheduler without the BOOST state. This is because, if all the domains are in BOOST state and only one domain can enter the head of the queue, all the other domains must wait for their time slot in order to execute. Another drawback is, if every domain is continuously in the BOOST state, it continuously tickles the scheduler and the scheduler in turn continuously interrupts the driver domain. Interrupting the driver domain continuously is not desirable as it might corrupt the demultiplexing of packets and might lead to other issues [3]. Hence, in such a scenario it is important to schedule the domains appropriately. The authors study the use of Hidden Markov Models in such scenarios

where the applications running on the domains have repetitive patterns over time and the I/O intensive domains can be scheduled using HMM.

The credit scheduling mechanism used in XEN has some drawbacks as discussed above. In addition to this, the domains who have negative credits and are in the “OVER” state cannot run till all the domains’ credits become negative. Once the credit of all the domains becomes negative, credits are again assigned to the domains in order for them to execute. This is definitely a bottleneck for I/O performance because it can increase the latency. Here, the authors propose a Hidden Markov Model based credit scheduler wherein the pattern of arrival of requests is exploited in order to reduce the latency of the I/O domains.

The proposed HMM classifies the arrival of requests in a 10ms window into one of the previously seen and closely resembling patterns and calculates the transition probability for the various domains. The pattern is classified using the log likelihood of the previously observed and the newly seen pattern. The log likelihood of the recently seen pattern is computed using the forward-backward algorithm and is compared to the log likelihoods of the previously recorded patterns. When the log likelihood of the newly seen sequence and the log likelihood of the previously seen sequence match or closely match, we can see that the sequence is following the trend of the previous sequence. The transition probability of the previous sequence is used as a basis and the newly observed sequence is trained based on the Baum Welch expectation maximization algorithm and a new transition probability is obtained for the newly seen sequence.

This new transition probability depicts the probability of seeing requests from the other domains given a current domain. Once the new sequence is classified this way, the scheduling takes place as described below.

4.2.3 Event Channel Notification in XEN

Whenever there is a request by a domain or a response packet waiting for the domain, event channels are used to communicate this request to the driver. The driver domain is also a domain like any other guest domains and is prioritized over the other domains while it still needs to be scheduled. There is a two layer hierarchy of ports that is shared by the hypervisor and the driver domain. This vector is known as the pending array and it lets the driver domain be aware of any pending requests from/to the domains [3]. So, whenever the domain has a request, it sets the corresponding bit in the backend array and when the driver domain scans the ports sequentially on the front end and finds a set port in the first layer, then it knows that there is a corresponding bit set in the second level and knows that there is a request pending. In the older XEN implementation, the next time the ports would be scanned starting from the port that was scanned last. But in paper [3], the authors propose scanning the port from the next port of the port scanned the last time. This way, a single domain will not consume many CPU cycles.

In this paper, the authors propose a mechanism that scans the port in the most probable sequence as predicted by the Hidden Markov Model. This is very important in reducing the delay between the events occurring and the time taken for scheduling the domain. The current scheduler implemented in XEN is mainly focused on fairness of the domains and does not concentrate on reducing the scheduling latency. This could be a major setback in domains running I/Os and hence, the authors exploit the Hidden Markov

Model to address the latency issue. This enhances the performance of the scheduler. The Hidden Markov Model for a XEN credit based scheduler is modeled and is found to improve the waiting times of each of the domains in the run queue. The importance of reducing the scheduling latency and the waiting times of the domains is explained and mathematically formulated in section 5.2.

CHAPTER 5

GOAL OF THE THESIS

The thesis is mainly focused on reducing the data read/write delay and the scheduling latency in a SAN environment. This problem is studied by developing a Hidden Markov Model (HMM). The Hidden Markov Model (HMM) is modeled for the purpose of study and applied to the disk read/write problem.

The HMM is a pattern classifier that classifies data depending on the patterns that the data access exhibit. Such models are found suitable for the thesis because of the following characteristics of I/O requests in a SAN environment:

- 1) **SPATIAL LOCALITY:** Data is stored in the form of contiguous blocks residing on the logical and physical disk [15]. Hence, the authors assume that the blocks on the disks are near in space, i.e., a concept called spatial locality.
- 2) **TEMPORAL LOCALITY:** Temporal locality deals with nearness in time, i.e., it is based on the concept that items that are referenced will be referenced again soon [15]. Since, we are dealing with the execution of a single process and a read/write operation on the logical and physical blocks, this property is well implied.

The study involved in this work is mainly intended for server virtualized environments, although it could be extended to a general SAN architecture as well. Although there are various protocols and algorithms proposed in the past to reduce the latency of a read/write, this paper is the first of its kind to study the Hidden

Markov Model to reduce the scheduling latency in a hypervisor and reducing the time for a read operation in a server virtualized setup.

Specifically, the goal of this thesis is two fold. First, a scheduling algorithm using a Gaussian HMM (GHMM) is developed which is proposed to reside as a part of the scheduler in a XEN hypervisor. The scheduling algorithm is proposed as an enhancement to the already existing credit scheduling algorithm in a XEN hypervisor. The algorithm is numerically evaluated and is found to improve the performance of the scheduler in terms of waiting times of the domains.

Secondly, the thesis extends the application of a discrete HMM (DHMM) to a server virtualized environment. The discrete HMM is modeled to prefetch the data from the disk into a cache in order to reduce the I/O delay. The modeling is done by storing and fetching the data from a database residing on the hard disk of the system incorporating a HMM and not incorporating a HMM. The results are then evaluated.

5.1 Current Data Prefetching Mechanisms in SAN

Some of the currently used data prefetching mechanisms in Storage Area Networks are discussed below. The current strategies trigger a prefetch when the host issues a read request. The most commonly used prefetching methods are the automatic prefetching and the adaptive prefetching. These algorithms work on the principle of reading some number of data blocks into the cache following the last read request issued by a host. The idea is, if a host requests for another data read sequentially, then the requested blocks of data can be found in the cache. This avoids a data fetch from the disk, thus reducing the latency of access.

The automatic prefetch mechanism uses an attribute called a “multiplier” that plays an active role in deciding how many block need to be prefetched. Whenever a value of zero is assigned to the multiplier, automatic prefetching is disabled and when the multiplier is assigned any other value, automatic prefetching is enabled. The value of the multiplier is set automatically by an API that decides if prefetching is required or not. The value of the multiplier is set after looking at the number of cache hits and misses when the host issues a sequential read request. The algorithm used in the automatic prefetching also recognizes the sequential read streams requested by the host and prefetches the data blocks in to the cache. The working of an automatic prefetch is as described below.

Two streams of host access history are maintained with the help of data structures and they are known as the active stream and potential stream. After a read request issued by a host is processed, the algorithm checks to see if the request is a part of the active stream. If it is, then the active stream is updated by removing that request from the stream and adding it to the front of the queue, hence making it the recently seen request. Depending on the cache hit/miss, the internal algorithm sets the value of the multiplier and decides if more data blocks need to be read into the cache. If yes, those many blocks of data will be read into the cache. If not, prefetching will not take place.

If the host read request is not a part of the active stream, the potential stream will be checked to see if the request is a part of it. If yes, then the number of such requests in the potential stream is searched. If the number is below some threshold, then the request is added to the front of the potential stream as a recently seen request. If it is above the threshold, then the request is removed from the potential stream and is made a part of the

active stream. If the request is not a part of the active or the potential stream, another potential stream is created and the request is added to the newer potential stream. The maximum size of the two streams is 64 and if the size of the stream exceeds 64, then the least recently used request is removed from the back end of the stream and the new request is fit in.

The automatic prefetching is never done for random host requests and is always done for the sequential read requests.

5.2 Mathematical Formulation of the Problem

In this thesis, the authors are mainly concerned with reducing the response time of a read request issued by a guest running on the hypervisor. Such a setup consists of the guest domains that are active, the hypervisor that hosts the guest domains, a physical interface that connects the hypervisor to the storage array and the storage array itself. In this work, the authors consider the physical interface to a fiber channel and a single storage array attached to a hypervisor that hosts 5 domains. The authors also consider a XEN hypervisor environment in particular, because it is open source and they mainly focus on improving the credit scheduler used in XEN. In such a case, the total response time consists of 5 components as shown below.

Let us denote the total response time by T_r , waiting time of the domain in the queue be W_d and the service time taken to service the domain be S_d . Let the latency at the sender be L_s and the latency at the receiver be L_r .

$$T_r = W_d + S_d + L_s + L_r \text{ ----- (1)}$$

$$T_r = L_s + L_r \text{-----} (2)$$

$$\text{But, } L_s = W_d + T_p \text{-----} (3)$$

$$L_r = S_d + R_d + T_p \text{-----} (4)$$

T_p is the transfer time on the fiber channel interface, S_d is the seek time for the request by domain 'd' and R_d is the rotational latency for the request by domain 'd'. Substituting equations (3) and (4) equation (2), we get,

$$T_r = W_d + T_p + S_d + R_d + T_p \text{-----} (5)$$

$$T_r = W_d + S_d + R_d + 2T_p \text{-----} (6)$$

Equation (6) is a general equation for any domain in SAN. Let us now consider a credit scheduler and see how to formulate the waiting time of the domain. The waiting time of the domains is directly proportional to the length of the run queue. If the length of the queue is more, the waiting times of the domains increases. This can be written as below.

$$W_d \propto L_q \text{-----} (7)$$

where L_q is the length of the run queue. But according to Little's law, the average number of customers in a stable system is equal to the average arrival rate multiplied by the average time the customer spends in the system. Let the average arrival rate of a domain be denoted by ' λ_d '. The average time spent by the domain in the run queue is W_d . Thus,

$$L_q = \lambda_d * W_d \text{-----} (8)$$

$$W_d = L_q / \lambda_d \text{-----} (9)$$

Substituting equation (9) in equation (6), we get,

$$T_r = L_q / \lambda_d + S_d + R_d + 2T_p \text{ ----- (10)}$$

The above equation 10 can be applied to a general SAN system. The authors add additional variables that are specific to the XEN scheduler and formulate what parameters need to be optimized in order to reduce the response time. The waiting time is split into two components. The first component is the time the domain has to wait after it has issued a request before it is scheduled and the second component is the time the domain has to wait in the run queue after it has been scheduled and before it is serviced. Let us denote the first component by ‘ μ ’ and can be defined by:

$$\mu = \text{sched}_d - A_d \text{ ----- (11)}$$

Equation 10 can be written as:

$$T_r = \mu + L_q / \lambda_d + S_d + R_d + 2T_p \text{ ----- (12)}$$

The authors introduce the first variable denoting the degree of ordering of the domains in the queue represented by ‘ Θ ’. Θ is a very important variable in time critical and I/O intensive applications. The authors define the degree of ordering of the domains as the ratio of the arrival time of the request from the domain to the time at which the domain was scheduled. The value of Θ determines the scheduling latency which in turn affects the waiting time of the domain. A higher value of Θ indicates less scheduling latency and a lower value of Θ indicates a higher scheduling latency. That is,

$$\Theta = A_d / \text{Sched}_d \text{ ----- (13)}$$

In the above equation (13), A_d cannot be changed as it is application dependent. So, equation (11) can be written in terms of scheduling latency. Thus,

$$\mu = \text{sched}_d - \Theta * \text{sched}_d \text{ ----- (14)}$$

$$\mu = \text{sched}_d (1 - \Theta) \text{ where } 0 < \Theta < 1 \text{ ----- (15)}$$

From the above equation, we can observe that when $\Theta = 0$, the scheduling delay is maximum and when $\Theta = 1$, scheduling delay is 0. Hence, Θ needs to be as large as possible. This means that if the domain is scheduled as soon as the request arrives, the domain does not have to wait for a long time after it has placed a request.

In equation 12, we cannot control the values of L_q / λ_d , R_d , $2T_p$. So, let us replace the sum of these variables by a constant 'C'. Equation 12 can now be written as:

$$T_r = \text{sched}_d (1 - \Theta) + S_d + C \text{ ----- (16)}$$

$$\text{Hence, } \min(T_r) = \min(\text{sched}_d, S_d) \text{ ----- (17)}$$

The goal of the thesis is intended to minimize the scheduling delay by maximizing Θ and by minimizing the seek time, S_d . This is achieved by deploying a Hidden Markov Model for prefetching and scheduling.

CHAPTER 6

HIDDEN MARKOV MODELS

Hidden Markov model holds its stand in the area of pattern recognition and classification that involves grouping several related patterns together. Many of the patterns seen in the real world are usually repetitive and complex in nature. When such patterns repeat over time, it is very useful to study the behavior of such patterns so that the future event can be predicted. For example, one could observe the weather pattern for about 3-4 months and study the probability of the weather change and model the same in order to predict the next day's or even the next year's weather.

In stock market analysis, one could model the stock market as a set of input and output parameters and study the behavior of the model. The predictive ability of the model can then be exploited to probabilistically determine what the state of the market will be the next day.

A hidden Markov model is a statistical model in which there are an underlying set of states that are unobservable and which emit some observable symbols with different probabilities. Transition takes place between the underlying states with different probabilities called the transition probability and the probability with which the observation symbols are emitted from the states is called the emission probability. In general, a hidden Markov model is a 5 tuple consisting of $(\Pi, A_{(i,j)}, B_{(i,j)}, Q, T)$ where,

Π = Initial probability

$A_{(i,j)}$ = Transition probability matrix

$B_{(i,j)}$ = Observation probability matrix

Q = Set of unobservable states

T = Length of observation

Hidden Markov Models can be classified into two types based on the type of observation symbol emitted from each hidden state. If the observation symbol emitted is discrete in nature, it is called a Discrete HMM (DHMM) and when the observation symbol emitted is continuous in nature, it is called a Gaussian HMM (GHMM). For example, in the weather forecast model, the observations could be walk, swim or skate depending on whether the state is sunny, rainy or snowy respectively. In this case, the observations are discrete and need to be modeled using a DHMM. Let us consider a speech recognition model. A speech recognition model is usually modeled with parameters such as energy characteristics of the uttered word, duration of silence as the observations and words as the states. The observations do not have any discrete values and vary with time continuously. Their value lies between a range of values $[x, x+\Delta x]$. In such models, a Gaussian HMM needs to be used. A DHMM is a set of 5 tuples as described above and a Gaussian HMM is described in detail in Section 7.2.

The observation symbol emitted by a state determines the next state. In a hidden Markov model, the observation sequence does not uniquely determine the state sequence. For example, in a stock market model, the day's closing price could be considered the observable and the examples of state could be low, high or constant. Depending on the day's closing price, it can be inferred if the market is in the state of high, low or constant. In a weather forecast scenario, the states could be assumed to be either sunny, rainy or snow and the observations could be walk, swim and skate. Hence, when there is a more

probability of a walk, the state can be guessed as being sunny. When people are skating more, the underlying state can be observed as being snowy and when people are swimming more, the hidden state can be determined to be rainy.

From the above examples, we can see that the states are not directly visible. The states need to be determined by looking at the observations. The transition between the various states, sunny, rainy and snowy, for example, can take place at different probabilities. This probability is expressed in the form a matrix called the “transition probability” matrix. When the state is, say sunny, there could be various probabilities of a walk, swim and skate. This probability is represented by a matrix known as the “emission probability” matrix.

Hidden Markov models are used in the area of pattern recognition wherein there is some kind of redundancy in the output observed over time. Hence, a hidden Markov model can be trained to learn the pattern generated by the application and can be used to predict the next event based on the previously learned information probabilistically. The states in a hidden markov model can be represented in the form of a finite automaton with the observables in each state. The change in states is represented by a transition from one state to another with some probability.

6.1 Modeling a Hidden Markov Model

A hidden markov model can be expressed in the form of finite automata as a set of states and corresponding observations wherein the next state depends only on the current state. The states in a HMM are the hidden states and the observations that are emitted from the states determine the next state and the underlying hidden states. The

transition between the various states is determined by the transition probability matrix, which represents the probability of transitioning from one state to another based on the observed output. There is another matrix, the emission probability matrix, which determines the most likely observation that can be emitted from a state. The process of determining the most likely sequence of states and observations and predicting the most likely next state involves the deployment of various pattern classification algorithms.

As explained in the previous section, a Hidden Markov Model can be expressed as a 5 tuple set. The model notations for a HMM are explained below [1]:

T = Length of the observation sequence. For a Gaussian HMM, it is the total number of clock times.

N = Number of states in the model

M = Number of observation symbols. It is the number of observable symbols emitted by each state in the model.

Q = Set of hidden states. It is a set represented by $\{q_1, q_2, q_3, \dots, q_n\}$.

V = Set of discrete observation symbols emitted by each state. It is a set represented by $\{v_1, v_2, v_3, \dots, v_m\}$.

$A = a_{(i, j)}$ is the probability matrix that describes the probability of transition to state 'j' when the current state is 'i'. It is also represented as $a_{(i, j)} = \Pr (q_j \text{ at time } t+1 \mid q_i \text{ at } t)$.

$B = \{b_j(k)\}$ is the emission probability distribution which describes the probability of observing the emitted symbols in each state. It is also represented as $b_j(k) = \Pr (v_k \text{ at } t \mid q_j \text{ at } t)$.

$\Pi = \{ \Pi \}$ describes the initial state distribution matrix and is represented as $\Pi = \Pr (q_i \text{ at } t=1)$.

Even before a pattern could be classified, it is very important to determine the right type of model for a given application in order to tune the predictive ability of the HMM and enhance its performance. For example, if we consider a stock market scenario, the states could represent the state of a market like whether it is a high, low or a constant. The observables could represent the day's closing price, the money flow ratio and the like. Another way of modeling the stock market could be to consider the states as profit or loss and the observations as the number of transactions and the money flow ratio. Hence, it is important to determine the right type of model for a given application, with the correct set of states and observations before deploying the model for usage.

Once the right kind of model is identified that can model an applications' input and output behavior accurately, the model needs to be trained well with the given training data set so that any future patterns can be identified correctly. The training data set is the large set of data that has been recorded previously as the output of the application. This kind of training is said to be offline training, because the training is not performed dynamically. Hence, Hidden Markov models are very handy in applications where the pattern generated by an application is repetitive in nature and follows some kind of a trend. Once the model is trained, the next step would be to classify the new data set that is fed to the application as belonging to one among the previously trained and observed data. This is a very critical step in pattern classification problems because any kind of misclassification would lead to very surprising results.

The classification of the new pattern implies that the new pattern has a similar probability distribution as that of the corresponding trained sample and it follows a

similar trend. Hence, the outcome would be the same and hence the next event can be predicted.

6.2 Discrete Hidden Markov Model in a SAN Environment

The thesis exploits the above mentioned capability of a HMM to prefetch the blocks of data in a SAN virtualized environment. In a virtualized server, the underlying physical devices like the memory and processor are abstracted and are presented to ‘n’ number of guests (virtual machines or domains) as if the processor was directly attached to each of them. The virtual machines are allowed to access a pool of memory unlike a directly coupled host accessing its own memory, there are I/O requests coming in from one or more hosts to either read a block in the array of disks or write to a block in the array of disks. In such scenarios, it would be very useful to have a cache wherein the application’s next read request can be pre determined from the training data and the pre-determined block of data can be pre-fetched into the cache. By doing this, the time required for the read operation can be reduced and when there are a lot of virtual machines running, scheduling can be done efficiently.

In this section, we look at how a DHMM can be exploited for prefetching blocks of data in SAN. Section 7.2 describes using a GHMM in a SAN environment. Before we look at the usage of HMM in SAN, it is worthwhile to mention about the type of access of data and the type of storage of data in SAN. RAID (Redundant Array of Inexpensive Disks) is frequently deployed in SAN setup in order to provide redundancy of data and higher capacity. Using various levels of RAID (RAID 0, RAID 1, RAID 5), a number of inexpensive disks are grouped together and data is distributed on those disks by a method known as striping. Volume groups are created from these disks of various RAID types

and are seen by the host that has access to the array. In SAN environments, the data is stored and accessed in the form of blocks. The size of each block varies and is dependent on the type of RAID used and the number of volume groups on the array.

Let us consider an organization that has some data stored on the disk arrays in different blocks. Let us also assume that there are 5 virtual machines running on a host that is connected to this array and the applications running on the VMs are accessing these data. By making various initial rounds of observations and recording the read/write observations, along with the block numbers read, the Hidden Markov Model can be efficiently deployed to prefetch the next block of data that could be read into a cache, so that the host can fetch the data from the cache instead of reading from the array which could be very time consuming. The size of the cache depends on the block size and the number of blocks present. The higher the size of the cache, the higher is the hit ratio. A graph of cache size versus hit time is shown in Figure 20 in Chapter 8.

The hidden markov model considered in this thesis is constructed with the block numbers as the hidden states and the observations emitted from each state (block number) are the reads/writes with different probabilities. As we can see, the observations are discrete in nature and hence, we deploy a DHMM for prefetching. The initial observations about the block numbers and the observations are trained using a Discrete Hidden Markov Model and the trained model is used to predict the next block that might be read depending on the current block. The state transition diagram for the HMM can be represented as shown in Figure 8.

Figure 8. State transition diagram

The disk I/Os are a series of reads/writes with different read/write request sizes. The application on the VM that starts the disk I/O operations can start all read operations or all write operations or a combination of reads and writes. These patterns of reads and writes can be categorized as sequential read/write, strided read/write and random read/write. In many block oriented applications, the I/Os are random in nature. The authors propose to use the model developed in this in applications where the access pattern is repeated over time. Many scientific applications have I/Os of this nature [5].

Because of the temporal and spatial locality discussed in the Chapter 5, the patterns generated by the applications can be exploited to prefetch the next block of data that is most likely going to be read, hence decreasing the read response time. In this

thesis, the authors mainly concentrate on reducing the time required for a read operation which determines the performance of the system. Decreasing the time required for a write is not addressed in the thesis. The critical step, however, would be to classify the pattern generated by the application on the VM correctly into one of the previously observed and recorded patterns. However, the training patterns need to be unique because HMMs do not determine the patterns uniquely based on the observations.

6.3 Proposed Hidden Markov Model

The HMM considered in the SAN environment for this thesis is considered to have 6 blocks of data. Hence, the number of hidden states is 6, i.e., block1, block2, block3, block4, block5, block6. Transitions take place among these blocks depending on the input given to the HMM module. The input consists of a sequence of read/write operations, the logical block address, the offset in the block and the block size to be read.

The observations from each hidden state are a sequence of reads and writes. The reads and writes from each of the block causes a transition to another block with some probability, known as the transition probability. The observation symbols like the reads and writes take place with some probability known as the emission probability. The emission probability is unique to each state and does not depend on any other state.

Hence, the transition probability is a 6X6 matrix as there are 6 states (blocks) and the emission probability is a 6X2 matrix as there are 2 emissions, either a read or a write from each state. An initial probability matrix is assumed randomly and is a 1X6 matrix. The initial probability matrix determines the starting state. If the initial probability of state 5 is the highest, then it is assumed that the state sequence starts from block5. The

training database consists of 30 patterns and each pattern consists of 48 observations of reads and writes. Hence, the length of the observation can be considered to be 48. After recording the 48 observations, the test pattern is classified as the one belonging to a specific training pattern and the next block is predicted using the respective algorithms. It is important to note that the length of the observations can be anything but here we consider 48 observations in order for better accuracy in classification. The greater the length of the observations, the better is the accuracy.

6.4 Training the Hidden Markov Model

Before deploying the HMM for classification and prediction, the training database needs to be established in order to train the recorded data. In a SAN scenario, the I/O consists of a sequence of reads and writes, the block size that needs to be read/written, the offset in the block that needs to be read, the logical block address that is read and so on. The logical block address is converted to a physical block address and a cache search is performed to see if the requested block is present in the cache. If so, the data is delivered to the VM from the cache. If not, it is read from the disk. The block number is found based on the input and is stored in the training database along with the requested operation, a read or a write. Data is recorded for all the different patterns of reads and writes found and is stored in the training database.

Since the previously observed data is used for training, the training is said to be done offline. 30 patterns are recorded where each pattern consists of a sequence of reads/writes. The block numbers that are read are recorded as well.

The HMM module is actually a software module that is proposed to reside on top of the hypervisor. Hence, the block number requested is made known to the HMM

module implicitly. It is expected to use the trained data to classify a new set of observations and predict the next block of data that might be read and prefetch it into the cache, so that the VMs will have easy access to it and will save the I/O time considerably. In this thesis, the authors propose to use a dedicated cache if the block size is large enough in terms of gigabytes. This cache will be a part of the hypervisor. If the block size and hence the cache size is not big enough, then the memory of the host that hosts the virtual machines can be used as a cache. In this thesis, the authors make use of a dynamic software cache to store the blocks of data depending on the request size. When a next read request comes in from any of the VMs, blocks are fetched from this cache if it is present. This is a very important observation to make in this thesis.

The training database is used to calculate the transition and emission probability matrices of the training data using the MATLAB function “hmmestimate”. The hmmestimate function accepts the observation sequence and the observed block sequence and calculates the transition and emission probabilities for that particular pattern. These matrices are used for testing a new pattern and for classification purposes. MATLAB uses another function for estimating the transition and emission probability matrices called the “hmmtrain”. Hmmtrain is used in scenarios where the state sequence is not known. But for the problem statement as described in the thesis, the state sequence (blocks) of the training pattern is already known to the HMM unit and hence does not need to use the hmmtrain function.

CHAPTER 7

PATTERN CLASSIFICATION USING HMM

HMM classifies the new pattern seen to the closest match among the previously seen patterns by calculating the probability of the observation sequence given the model. The new test pattern sequence depicts only the observation sequence and the underlying states that the observations went through are unknown. It is desirable to determine the state sequence that the test pattern has undergone in order to predict the next state. Since, the I/O patterns are produced by processes; it can be assumed that the patterns in disk access repeat over time. So, whenever a new pattern matches a previously observed pattern, it is necessary to classify it properly. Classification also helps to determine the probability distribution of the pattern as it closely resembles the probability distribution of a previously seen pattern.

The forward algorithm calculates the probability of the observation sequence given a model. From the training samples, the transition probability and the emission probability matrices are estimated using the `hmmestimate` function. The forward algorithm takes these matrices as inputs and calculates the log probability for the observation sequence given the model (initial probability, transition probability and emission probabilities). The higher the log probability, the higher is the likelihood that the model produced the new observation sequence.

The observation sequences exhibit 3 main kinds of patterns: sequential, strided or random. The sequential pattern is the best for predicting the future requests because of its sequential nature. So, if the current block is 'i', the next block would be 'i+1'. In a strided

access pattern, if the current block is 'i', the next two blocks would be 'i+2' and 'i+4' respectively. In a random access pattern, the accesses are not defined and any block can be accessed after any other block. Using a random access pattern for prediction is a poor indicator of future accesses and may not yield good results. However, using a HMM, a random pattern can be used for prediction as long as the patterns and the observations are unique. Hence, if there are two exactly similar observation sequences, HMM classifies them as belonging to the same class but does not indicate which pattern among the two are generated by the application. In such cases, we can use the algorithm proposed in [20] to classify the data sequence. However, for the purpose of this thesis, the authors follow the method proposed in [5] to differentiate between the two similar patterns based on the probability of occurrence. Hence, if pattern 1 has a higher probability of occurrence than pattern 2, then the newly seen pattern more closely resembles pattern 1.

The next step after calculating the log probability is to match the test sequence with the appropriate training sequence. As mentioned earlier, the model that has likely produced the test observation is the one that has the highest log likelihood. Once the model is known, the trained model is optimized in 2 steps.

In the first step, transitions to the same consecutive blocks are removed. This ensures that reading the same block 2 or more times is the same as reading the entire block once. In the second step, state transitions are scanned to find those transitions whose probability is less than a threshold of 0.05. In the event of finding such transitions, the transitions are removed. The presence of low probability transitions decreases the accuracy of prediction. All low probability transitions for all the transitions starting with

different starting states is eliminated in the second step of optimization to ensure that only the major transitions are present in order for effective prediction.

The resulting state sequence and observation sequence derived from the original classified and trained sequence is then used to find the new transition and emission probabilities using the `hmmestimate` function. In typical HMM applications, the next step after finding the new transition and probability matrix is to find the most likely state sequence using the Viterbi algorithm. The authors felt that this step is not necessary because of the following reasons:

- 1) The HMM predictor is proposed to run as a separate thread on the hypervisor. Hence, the HMM predictor acts as an intermediary between the application and the hypervisor.
- 2) The application knows the virtual address of the data which it supplies to the hypervisor. The hypervisor or the management domain that is responsible in turn looks up the physical page address and fetches the required data. During the physical page lookup, if the hypervisor supplies the block number info to the HMM module, then the HMM knows the current block that is being read and can prefetch the next block of data into the cache.

Now that the new transition and emission matrices are ready, a separate function written to prefetch the next block is called and the predicted next two consecutive blocks are read into the software cache depending on the size of the cache. In this thesis, a dynamic caching scheme is used by the authors. The software cache like a linked list or an array is used to store the prefetched blocks of data. Since the training

is done offline, it is assumed that the different data access patterns have already been observed, recorded and trained. However, the classification of the test data is done automatically using the forward algorithm and log likelihoods.

7.1 Prefetching in a Hypervisor versus Prefetching in Parallel Processors

The model described in [5] is closely related to the discrete HMM used in this paper. They model the HMM with hidden states representing the segments of a file and the observations are reads and writes. Although the Hidden Markov Model used in both the works is the same, the architecture and the application where the models are used are very different leading to different results. They use the HMM in a parallel processing environment where multiple processors running the same application are networked and the common files are accessed by the processors from a server serving the request by the processors. Each processor has its own memory and cache and depending on the access pattern of the files from the processors, the file access policies and cache policies are tuned. They classify the global access pattern generated by various parallel processors based on overlapping and provide hints in order to set the policies.

In this work, the authors use a Gaussian HMM to improve the performance of a XEN credit scheduler and a discrete HMM to prefetch blocks of data into the common cache shared by all the virtual machines running on a hypervisor. In the discrete HMM, the authors model the hidden states to represent the block numbers and the observations are sequences of disk I/Os generated by the virtual machines. The authors consider a single server hosting 5 virtual machines running I/O intensive

applications. For the sake of convenience, they consider each virtual machine with a single virtual CPU (vCPU). Hence, only one VM is allowed to execute at a time. Also, the fact that a common cache is shared by the VMs is very important in determining the likelihood of finding the required block in the cache. In this work, there is a greater hit ratio because the block prefetched by some other VM might be the block required by another VM next.

7.2 Hidden Markov Model with Gaussian Output

The HMM described in Section 7.1 consisted of discrete outputs of either a read or a write and a discrete set of hidden states describing the block numbers. The authors extend the thesis by modeling a Gaussian HMM for continuous observations and use the model to improve the performance of the XEN credit scheduler in terms of scheduling latency.

The Gaussian outputs follow a Gaussian distribution in which the data clusters around a mean or average. The central limit theorem states that any variable that is a sum of several independent variables is normally distributed. The Gaussian distribution is hence, used in statistical analysis and to model complex phenomena.

The Gaussian distribution was used in this thesis for the following reason. Let us assume that there is a server hosting several virtual machines. In this thesis, the authors consider a XEN hypervisor as it is open source. The VMs are running applications that are I/O intensive and let us assume that they are running scientific applications and requesting access to the storage array. When there are a large number of requests, the scheduler in the hypervisor schedules the requests and adds the domain to an I/O queue, also known as a run queue. A domain is a term used by XEN that is equivalent in

meaning to a virtual machine. If we observe the arrival of requests at the scheduler and record the arrival times, the distribution of these times follows a Gaussian distribution. The authors also assume that for scientific applications, the request patterns and the arrival times follow some pattern that recurs over time. Such a distribution is as shown in Figure 9.

Using an HMM with Gaussian outputs, such patterns can be trained and when it occurs the next time, the guest ID that will most probably arrive at the scheduler can be determined. Once the guest ID is determined, it can be immediately scheduled leading to a decrease in scheduling latency. The discrete HMM discussed in Section 7.1 can be used to process the request by the domain at the head of the queue and classify the access pattern.

A HMM with Gaussian output is represented in the same way as a discrete HMM with a few changes. The Gaussian HMM can be represented as:

$$\text{Lambda} = (A, B, \pi, \mu, \Sigma)$$

Where $A_{(i,j)}$ is the probability of transitioning from state i to state j and emitting a symbol 'O'. In the discrete HMM where we had 'B' as the probability of observing the discrete symbol 'O' from the states, the Gaussian density function emits an observation whose value lies between the range $[x, x+\Delta x]$. π is the initial probability matrix describing the starting state or the initial state. μ, Σ are the mean and covariance matrices respectively. The observation probability matrix, $b_j(x)$ is given by the following formula:

$$b_j(x) = \sum C_{ij} * \epsilon [x, \mu_{i,j}, \Sigma_{i,j}]$$

where C_{ij} is the mixture co-efficient. If all the observations are Gaussian, the co-efficient of mixture is 1. If the observations are a mixture of discrete and Gaussian observations, the co-efficient of mixture varies.

All the above parameters are initialized appropriately depending on some estimation based on the previous windows of observations. It is very crucial that the above parameters are initialized and estimated properly. When it is very difficult to determine these parameters for initialization, the Bayes method can be used wherein there is no prior knowledge about the state distributions and observation densities. The Bayes method calculates the above parameters by itself based on the description of the model given the number of states, number of observations, state names and so on. For this thesis, however, the authors assume an estimated value of the parameters based on the values assumed in the training database.

7.3 Gaussian HMM for Scheduling in a Hypervisor

The authors use the Gaussian HMM method of classification to classify the I/O request pattern coming from one of the 'n' number of guests in a server virtualized environment. If the guest that is going to request I/O next is known in advance, then it can be scheduled fast, thus reducing the waiting time of the domain and hence the total response time.

The HMM model for prefetching consists of a set of hidden states describing the domain numbers or the guest IDs, which are more frequently called “worlds” by VMware community and “domains” by XEN. These set of states are called “hidden”, because the guest ID is not directly visible to the HMM module. The goal is to determine these hidden states that lead to the observations.

The observations are made at the hypervisor level and hence, the authors consider the arrival times of the requests for classification. Since the arrival time has a lower bound and an upper bound, these variables are considered to be continuous variables and since they can be clustered around a mean value, they are assumed to follow a normal distribution. Figure 9 and Figure 10 shows that Gaussian distribution more closely fits the arrival time sequence.

- 1) Arrival time: The time at which a domain calls an interrupt requesting for disk I/O. This can be considered as a system clock time during the session in which the port of the domain is set. The arrival time is not the same as inter arrival time.
- 2) Response time: The time needed for the data retrieval/ write operation to successfully complete and the hypervisor receives the respective response. This response time is not the same as round trip time, which is the time required for the application running on the guest OS to receive the response.

For the purpose of this thesis, the authors assume that there are 5 guests running on the virtualized host and also assume that all the five guests are active within a session. This is because in many scientific applications, we see all the guests active and performing the read/write operations simultaneously. The arrival times are considered within the interval of $[0, 10]$ ms.

Figure 9. Normal distribution plot for the arrival time of requests

Figure 10. Exponential distribution plot for arrival time of requests

The transition matrix describes the probability of transition from one guest to another. During this transition, the observables such as the arrival time and response time of the guest are observed and the probability that the values of these observables are within some assumed range of values is calculated. This is done at each hidden state and forms the emission density matrix. For the purpose of this thesis, the emission density matrix is chosen randomly.

7.4 Training the Gaussian HMM

The Gaussian HMM is trained using the HMM toolbox for Matlab software. The training samples consist of 30 sequences of observations. Each sequence consists of 50 samples of arrival times within the interval $[0, 10]$ ms. The arrival times are normally distributed. As mentioned in the previous section, the authors consider 5 guest operating

systems running on the VM host with the IDs 1,2,3,4 and 5. Since, for training purposes we know the guest IDs initially, we can calculate the transition probability from the state sequence. The transition probability is calculated the same way as we had calculated in the discrete HMM. That is,

Transition probability from state A to state B

= Number of transitions from state A to state B / Total number of transitions from state A

It is also required to determine the emission density matrix. The emission density matrix is assumed randomly. It is the probability that the observed emissions (arrival time, response time) at the state 's' is between value 'x' and value 'x+Δx'.

For a given sequence, it is also required to determine the mean vector and variance. The mean vector and variance is determined for a given sequence by using the in-built function `mixgauss_init`. The prior probabilities for the states are assumed. The above method is done for all the training sequences. Once the appropriate model is determined, the initially estimated parameters like the transition matrix, emission density matrix are refined to better fit the data through the process of expectation maximization algorithm. The now refined parameters reflect the probability distribution of the data more clearly.

The next step is to determine the log likelihood that the assumed model for the training sequences indeed produced those sequences. The log likelihoods for all the training sequences are determined and recorded. The test sequence is the sequence that has newly arrived at the scheduler queue. The test sequence is hence, a sequence of arrival times. The underlying states (guest IDs) that produced those sequences are

unknown and needs to be determined. In scientific applications, the data access pattern is assumed to follow certain patterns that repeat over time [5].

. The HMM is capable of detecting the underlying pattern and classifying it as one of the previously seen pattern, provided the training samples are large enough. Hence, the authors have considered 50 observations of each of the observables. Even though a small number of samples could have been used for the training, it is recommended to provide a large data sample for better accuracy in classification. The log likelihood is calculated for the test data as well. Next, they compare the log likelihood of the test data with the log likelihoods of the training sequences. Whichever log likelihood is exactly the same or nearly the same is considered to be the probability distribution that the test data is going to follow.

7.5 Event Channel Notification and Scheduling in XEN using Gaussian HMM

Even channel notification and scheduling in a default XEN scheduler is explained in section 4.2.3. The domains are scheduled using HMM based on the time of occurrence of events. Most importantly, the authors propose not to scan the pending array sequentially in this paper. Instead, the authors scan the ports of the pending array in the order determined by the Gaussian Hidden Markov Model (GHMM). The thesis makes important observations on order of scanning of pending array and finds that sequentially scanning of ports leads to latency issues. The authors assume that the different domains issue I/O requests at particular time periods and no two domains issue requests at the same time. A $[0, 10]$ ms interval is used to note the arrival of requests of the domains.

Since time is continuous, the authors use a HMM with Gaussian output to model the arrival of requests as described in section 6.3. The pattern of request arrival is

classified and the transition probability matrix is calculated. Let us assume that there is a domain running at the head of the queue at the end of the pattern classification. The next most probable domain that can be executed is found and the ports are scanned corresponding to the predicted domain in the pending array. If the port is set, the authors push the domain into the run queue and schedule it to run after the currently executing domain. Similarly, the next most probable domain is found after the recently scheduled domain and that is put in the run queue as well. If the domain predicted does not have its port set, the authors find the next most probable domain from the transition probability matrix and see if its port is set. If so, the domain is scheduled.

The Hidden Markov Model provides a probabilistic framework for prediction of the future trend the data pattern follows. The accuracy of prediction is certainly a factor to consider in any probabilistic policies. Hence, we need to make the scheduler more fault tolerant while at the same time providing a level of fairness. Performance always trades off fairness and vice versa. In this paper, the authors propose an algorithm that increases the performance of the scheduler for I/O intensive domains. However, I/O intensive domains do not require a lot of processing power and usually run lesser than the 10ms period. Hence, providing a very high level of fairness to such domains is a point that is still looked into.

Currently, in the XEN credit scheduler, the weight of the domains is recalculated and the domains are rescheduled every 30ms. In this thesis, the authors propose to reclassify the arrival times every 30ms. The advantage of doing this is to reduce the classification error. The first 10 ms of arrival time of requests is recorded and used for classification. If the log likelihood of the newly seen data and the training data are not

anywhere near to each other, then the test sequence is considered to be new and not seen previously before. Such a sequence is added to the training database so that it can be used for classification in the future. Whenever this happens, the GHMM module is not used for prediction because the classification would be wrong leading to bad results and high latency.

At the end of the 30th ms, the next 10ms of data is classified again and this cycle can be repeated. Hence, by rescheduling every 30ms, the probability of an erroneous classification can be reduced leading to better results.

7.6 Credit Scheduling Algorithm without HMM

Let 'n' be the number of active domains on the hypervisor

Step 1a: Assign credits lesser than 65536 to the domains

Step 1b: Scan the pending array for any active domains

Step 2: Calculate the time each domain can execute on the processor and insert the domain into the run queue in the scanned order

Step 3: After every 10ms, debit 100 credits from the domain at the head of the run queue.

After every 30ms, recalculate the weights of the domains and reschedule them.

Step 4: If the credit of the domain becomes negative, go to step 5. Else go to step 6

Step 5: Push the domain to the end of the run queue and wait till the credits of all the domains becomes negative. If the credits of all the domains in the run queue become negative, go to step 10

Step 6: Add the domain to the run queue after the existing runnable domains in the run queue

Step 7: If the domain is at the head of the run queue, execute the domain

Step 8: If the domain has done executing completely, remove the domain from the run queue and execute the next domain in the run queue

Step 9: If not, go to step 3

Step 10: If number of domains in the run queue is lesser than the threshold, go to step 1a.

Else, go to step 3

7.7 Credit Scheduling Algorithm with HMM

Let 'n' be the number of active domains on the hypervisor

Step 1: Assign credits lesser than 65536 to the domains

Step 2: Calculate the time each domain can execute on the processor

Step 3: Record the arrival time of the requests from the domains and also the domain IDs

Step 4: After scanning the first 50 domains of the 10 ms window, run the HMM module to classify the arrival pattern into the previous closest sequence seen

Step 5: Find the most probable transition matrix that describes the most likely domain requesting a read/write next

Step 6: Scan the port on the pending array belonging to the domain predicted by the GHMM next. If that port is set, insert the domain to the run queue.

Step 6: Execute the domain at the head of the run queue for the time slice calculated

Step 7: After every 10 ms, debit 100 credits from the domain. After every 30ms, recalculate the weights of the domains and reschedule them. Reclassify after every 30ms.

Step 8: When the pending array is scanned the next time, scan the port of the next most probable domain after the current last executable domain in the run queue as explained by the transition probability matrix

Step 9: If the credit of a predicted domain becomes negative, assign the credit to the domain based on its priority as predicted by the HMM. That is, if the domain is near to the head of the queue, assign lesser credits when compared to the predicted domain that is far from the head of the queue

Step 10: If a domain has completed execution, remove it from the run queue. Else, go to step 6.

CHAPTER 8

SIMULATIONS AND RESULTS

The simulations were conducted by modeling classification and prefetching scenarios using MATLAB. The scheduling algorithm is numerically evaluated after classification by the MATLAB. MATLAB provides several useful functions and algorithms that are used in statistics that enable the application to be programmed easily and is very user friendly. The authors used the statistics toolbox in MATLAB that provides functions like `hmmtrain`, `hmmestimate`, `hmmdecode` to model and analyze the discrete HMM used in block prefetching. However, MATLAB does not support any special functions for the Gaussian HMM. Hence, the authors used the HMM toolbox written by Kevin Murphy to model and analyze the Gaussian HMM for scheduling purposes.

We first describe the simulations and results for the scheduler using Gaussian HMM and then we describe the discrete HMM used in block prefetching. As described in the previous section, the HMM toolbox helps in classification of the newly observed sequence into one of the previously seen sequences. For scheduling purposes, the data was classified based on the arrival time of the requests.

For the purpose of the experiments, the authors have considered 15 sequences of arrival time data, each sequence consisting of 50 arrival time information. The interval of the arrival times observed were taken between $[0, 10]$ ms. These sequences are considered to be the training data and are stored in an Oracle database residing on the hard disk of the system. The interval or the number of observations can be of any length. However, the authors used a long observation sequence for accuracy of prediction. The

domain IDs of the domains are considered to be the hidden states in their model as it is desirable to predict them and they are not directly observable. The observations are the arrival time of the requests from the domains. The authors assume there are 5 domains running on the hypervisor other than the driver domain for the sake of their experiments for convenience.

The credit scheduler is modeled as follows. The arrival time of the requests from domains is recorded for the first 50 requests. This is the new test data that needs to be classified. Next the log likelihood for all the training data residing on the disk as well as the newly seen test data is calculated using the forward-backward algorithm. The log likelihood of the training data that closely matches the log likelihood of the test data is recorded. We can then assume that the test data follows a similar pattern as that of the training data recorded. This completes the first stage, pattern classification, of their experiment.

Then, the transition matrix needs to be determined. This is the matrix that describes which domain is most likely to place the next request. To determine this, we first need to train the test sequence using expectation maximization algorithms using the transition matrix of the recorded training sequence as the base transition matrix. The toolbox uses the Baum-Welch algorithm to train the test data. It takes the transition matrix of the recorded training data, observed test sequence as the arguments and calculates the transition probability for the test sequence. From this the authors determine the probability of transition from one state to the other. For testing purposes, the authors considered each of the sequences from the training database and classified it. All the

sequences were correctly classified into the respective classes. This transition matrix serves very important purposes for the scheduling algorithm.

Figure 11. CPU fairness in a default XEN scheduler

The authors considered an arrival time sequence and the respective domains that placed the requests as shown in Table 1. The authors found that this sequence of arrival times closely matches training sequence 10 in the database. From Figure 14, we can see that the classification time varies linearly with the number of access patterns. Hence, storing a history of 30 access patterns as the metadata will be optimal in terms of classification time. The disk space needed to store these 30 patterns for GHMM is at most 30KB in their experiment. The last domain that is in the test sequence is 3. It is now necessary to determine which domain might place the next request so that it can be scheduled before hand and the latency be reduced. The authors find from the transition matrix that whenever the current state is 3, the most probable

transition is going to be to state 4. Hence, they schedule domain 4 to run next. They initially assign equal credits to all the domains and calculate the waiting time of each domain in the run queue when HMM is used and when HMM is not used. When HMM is not used, the default XEN credit scheduler provides fairness of access to all the domains as shown in Figure 11.

TABLE 1
ARRIVAL TIME PATTERN DATABASE FOR GAUSSIAN HMM

SEQUENCE NUMBER	ARRIVAL TIME	GUEST ID
10	0.8	1
10	1.1	2
10	1.4	1
10	1.5	3
10	1.8	4
10	1.9	1
10	2.5	1
10	2.7	4
10	2.9	5
10	3.0	3
10	3.2	2
10	3.4	1
10	3.6	1
10	3.9	2

TABLE 1 (contd)

10	4.1	3
10	4.2	3
10	4.5	5
10	4.7	4
10	4.8	5
10	5.0	2
10	5.2	1
10	5.3	1
10	5.4	3
10	5.5	4
10	5.7	5
10	5.9	2
10	6.0	3
10	6.1	4
10	6.3	5
10	6.4	1
10	6.7	2
10	6.8	1
10	7.0	3
10	7.1	1
10	7.2	5
10	7.4	4

TABLE 1 (contd)

10	7.6	4
10	7.8	3
10	7.9	1
10	8.0	5
10	8.2	2
10	8.4	3
10	8.5	1
10	8.7	2
10	8.8	4
10	8.9	3
10	9.3	2
10	9.4	5
10	9.5	5
10	9.6	3

To calculate the latency, the authors assume from the transition matrix that the next 30 domains that will place requests after the first 50 domains are:

1, 2, 5, 3, 5, 4, 5, 3, 1, 3, 1, 2, 5, 3, 1, 2, 5, 3, 1, 3, 5, 3, 1, 2, 5, 4, 5, 3, 1, 2

Since they have assumed initially that all the domains get equal credits to run, they assign 300 credits to each of the 5 domains. Thus, every domain gets 20% of the CPU time to execute till the next 30ms, after which the weights of the domains and the time to execute on CPU are recalculated. The percentage of time on CPU that each domain gets is calculated as:

$\% \text{ of time} = (\text{weight of the domain} / \text{sum of weights of all domains}) * 100$

Figure 11 shows the default configuration of XEN when every domain is given equal percentage of CPU time to execute. It is assumed that most of the scientific applications run at least for 40-50ms and the arrival time is reclassified after every 30ms in order to reduce the error of classification. For the time being, let us assume that the predictions about the domains using HMM is correct. If they had not used HMM to schedule the domains, the order of scanning the next 30 domains by the driver domain would have been as follows as determined by the default scanning by the XEN driver domain.

4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3, 4, 5, 1, 2, 3

Figure 12. Timeline of execution of domains with GHMM

The timeline of execution of domains while using HMM is as shown in Figure 12. From this timeline, the authors determine the waiting times for each of the active domain. When HMM is not used, the waiting times for each of the domains is calculated and is

found to be as shown in Table 3. The calculations for the waiting time are done as follows: In the first sequence, the first instance of domain 5 has to wait for the execution of domains 1 and 2 to complete. Each of these domains takes 0.2 ms to execute. So, domain 5 has to wait for $0.2 + 0.2 = 0.4$ ms. Similarly, the waiting times for all the instances of all the domains in the run queue is calculated.

Figure 13. Waiting times of the domains with and without GHMM

Figure 13 shows the comparison of the waiting times for the domains when HMM is used and when HMM is not used. We can see that using HMM reduces the waiting time of the domain by about 30%.

TABLE 2
WAITING TIME OF THE DOMAINS WITH AND WITHOUT GHMM

DOMAIN NUMBER	WAITING TIME WITHOUT HMM (ms)	WAITING TIME WITH HMM (ms)
1	4.2	3.3
2	3.2	2.8
3	4.4	3.5
4	3.5	0.9
5	4.4	3.3

We now describe the prefetching of blocks into the cache using HMM. The authors use a discrete HMM to model a prefetching scenario in SAN. It is desirable to determine what block of data will be requested by the domain next in order to prefetch the data. Hence, they model a discrete HMM with block number as the hidden state and each state has observations of reads and writes. The training database consists of 30 patterns and each pattern is a sequence of 48 reads and writes issued by a host. Data blocks are stored in an Oracle database residing on the hard disk. Hence, accessing these blocks is same as accessing the hard disk. From Figure 14, the authors found that storing about 30 access patterns of history in the training database is found to be optimal in terms of classification time and like in the GHMM, the 30 patterns occupy at most 30KB of disk space. Every domain running on the hypervisor has a dedicated metadata space to store the history of pattern access. For example, the access history of domain 1 is stored

separately from the access history of domain 2 and the history of every domain has 30 patterns. The pattern access database is shown in Table 2.

Figure 14. Classification time versus number of access patterns

Classification is done at a local level within the history of access of each domain. This is because the authors have considered a single server in their thesis allowing only one domain to execute at any time. Hence, if domain 1 is executing, its current data access is matched to one of its own previously seen patterns. Whenever, more patterns are stored, it is found to increase the classification time linearly. This is shown in the above Figure 14. When new patterns are added and when 30 patterns are exceeded, the least recently used pattern can be evicted from the database to store the new pattern. Thus, if 5 domains are executing on a hypervisor and if each domain needs 30KB to store its history, then 150KB of disk space is needed to store the history of the 5 domains. But advances in hardware have made large disk spaces and lots of memory available and

space is not a concern. Also, the data compression algorithms can be used to minimize the disk space used by this metadata.

In the discrete HMM, the patterns are based on the sequence of reads and writes unlike the Gaussian HMM where the patterns are classified based on arrival times. They also record the block numbers requested by the hosts in that pattern in the database. One of the sequences among the training sequences is considered to be a test sequence and the log likelihood of the test sequence is found using the forward-backward algorithm. The test sequence follows the same trend of that training sequence whose log likelihood closely matches that of the test sequence.

TABLE 3
READ/WRITE PATTERN DATABASE FOR DISCRETE HMM

BLOCK NUMBER	OBSERVATION	SEQUENCE NUMBER	REQUEST SIZE
1	1	1	32
3	1	1	32
5	1	1	32
4	1	1	32
2	1	1	32
5	1	1	32
4	1	1	32
4	1	1	32
3	1	1	32
3	1	1	32

TABLE 3 (contd)

4	1	1	32
6	1	1	32
5	1	1	32
1	1	1	32
3	1	1	32
1	1	1	32
2	2	1	32
2	2	1	32
4	2	1	32
6	2	1	32
6	2	1	32
3	2	1	32
1	2	1	32
4	2	1	32
3	2	1	32
3	2	1	32
4	2	1	32
4	2	1	32
5	2	1	32
6	2	1	32
1	2	1	32
1	2	1	32

TABLE 3 (contd)

1	1	1	32
6	1	1	32
6	1	1	32
2	1	1	32
2	1	1	32
4	1	1	32
6	1	1	32
4	1	1	32
1	1	1	32
3	1	1	32
2	1	1	32
5	1	1	32
4	1	1	32
2	1	1	32
6	1	1	32
3	1	1	32

They then use the ‘hmmestimate’ function to find the transition probability matrix for each of these training sequences. ‘hmmestimate’ function is used in place of ‘hmmtrain’ whenever the transition probability matrix is known. The transition probability matrix of the resembling training sequence is then optimized in two stages. First, whenever there is a transition from one state to itself, the transition is removed. Next, whenever the probability of transition is lesser than 0.05, such a transition is

removed. The optimization is performed in order to increase the accuracy of prediction. The transition matrix obtained after optimization is now functional and is ready to predict the next block of data that the guest might access next. The block of data that is predicted is looked up in the cache to see if it is already present. If so, the next prediction is made and looked up in the cache. If the block is not present in the cache, it is fetched from the disk into the cache. The response time that the guest incurs when HMM is used and when HMM is not used is recorded. They found that using HMM reduces the read response latency by 38.06% on an average.

Figure 15 shows the classification of the read/write patterns into different classes. We can see that all similar patterns have the same color and are present close to each other. When the patterns have the same color but are not close to each other, there is a classification error. In the experiments described in this thesis, the authors found the accuracy of classification to be 87% with DHMM. The latency involved during the disk reads is studied by varying the block sizes. The latency incurred when HMM is used and when HMM is not used is as shown in Figures 16, 17 and 18.

Figure 15. Scatter plot of the log likelihoods of the access patterns

Figure 16. Time to fetch 32 byte block with and without DHMM

Figure 17. Time to fetch 64 byte block with and without DHMM

Figure 18. Time to fetch 128 byte block with and without DHMM

As seen from the above graphs, when a HMM is used for prediction and caching and when the prediction is accurate, a lot of time can be saved which thereby reduces the time taken for the read operation. This in turn increases the throughput of the system. A plot of the throughput of the system when HMM is used and when a HMM is not used is shown in Figure 19.

Figure 19. Throughput of the system with and without DHMM

Figure 20 shows the response time for a read operation when the requested block of data is present in the cache and when it is not present in the cache when a Hidden Markov Model is used. As we can see, when the prediction made by the HMM module is correct, it saves a lot of time because of a cache hit. But, when the HMM has made wrong predictions or when some other block of data that is not present in the cache is requested, it takes much more time. However, the time taken for a miss is still lesser than the time taken when HMM is not used.

Figure 20. Response time with cache hits and cache misses

The authors found that when the cache size is increased to hold 3 blocks instead of two, it helped in alleviating the response delay by reducing the number of cache misses. Figure 21 shows the response time when the cache size is increased.

Figure 21. Cache miss time with increasing cache size

CHAPTER 9

CONCLUSION AND FUTURE WORK

The thesis mainly focuses on reducing the latency involved in a server virtualized environment in terms of scheduling and the read response time. An algorithm that uses Gaussian Hidden Markov Model is used to model the credit scheduler in XEN and a discrete HMM is used to model the read and write requests by the guests. Although the algorithm is proposed to be deployed in the credit scheduler, the HMM itself can be used as a part of any other scheduler or virtualization platform with some modifications. The GHMM considers the arrival times of the requests from domains and classifies it into one of the previously seen sequences. The transition matrix obtained helps determine the most probable domain that might run next probabilistically. The GHMM correctly classifies the observations with 100% accuracy provided the correct parameters are specified. The authors have found that using a GHMM considerably reduces the latency of scheduling the domains.

Similarly, a discrete HMM (DHMM) is used to model the block requests of each domain individually. The DHMM considers the block number as the hidden state and the reads and writes by the domains as the observations. The DHMM classifies the newly seen sequence as one of the previously seen sequences and the transition matrix is used to determine the next most probable block of data that the domain might request next. The classification by the DHMM toolbox is 87% accurate in classification.

The above two models are based on the fact that the nature of data seen is repetitive in nature. That is, the sequences and the corresponding states display a trend over time that can be exploited for predicting the future behavior of the application.

However, if the data behavior is highly random and does not display any trend over time, then the two models are found to worsen the latency rather than improving it. In such cases, other mechanisms and different statistical methods need to be used. The thesis however does not look into any additional methods and is left as a future work in their research. Most of the scientific applications are found to display repetitive nature in requesting the data. Hence, their model will work best in such applications even though it can be used in other applications with similar nature.

Statistical analysis techniques demand a lot of data and hence the space to store that data increases. The training sequences need to be stored somewhere on the disk or the disk array and if there are a large number of domains running on the host, then a lot of space is utilized for storing the metadata. Even though the amount of disk space has increased over the years, storing metadata is expensive. As a solution to this, compression techniques can be deployed to compress and store the data. But again, the number of active domains plays an active role. This motivates the need to develop mechanisms for inexpensive ways of storing metadata.

Another important observation about statistical analysis is that the predictions made are probabilistic and not 100% accurate. The transition matrix obtained in the two models developed in this thesis provides the most ‘probable’ next transition and not the ‘exact’ next transition. This could mean the need to develop algorithms that can improve the accuracy of prediction, hence lowering the error rate and increasing the fault tolerance. But overall, the authors found the statistical analysis using Hidden Markov Models simpler than the other techniques but yet a powerful tool that can be used to exploit complex data pattern.

Pattern classification using HMM classifies the test data into one of several classes. The test data could resemble several patterns that follow the same trend but yet have different state transitions. Once trained, HMM could be insensitive to small changes in the transition between states. But, this amount of noise and distortion is the bottleneck of any pattern classification algorithm and is a risk that needs to be taken. In this thesis, we use the probability of occurrence of the similar patterns to select the pattern that the test data resembles. Hence, other techniques need to be developed on top of the HMM that can classify the test data into the correct sequence. The paper proposed by [20] gives the details of such algorithms. The incorporation of such algorithms into the algorithm discussed in this thesis is left as a future work by the authors.

The thesis considers a scheduler in a single server, single vCPU environment hosting many virtual machines and numerically evaluates the performance. The work done in this thesis can be extended to a multi server, multi vCPU scenario with varying workloads and a real time software that can be incorporated into the existing hypervisor needs to be developed.

LIST OF REFERENCES

LIST OF REFERENCES

- [1] L.R.Rabiner, B.H.Juang, "An Introduction to Hidden Markov Models," IEEE ASSP Magazine, 3(1, Part 1), pp. 4, 16.
- [2] Maria Fox, Malik Ghallab, Guillaume Infantes, Derek Long, "Robot Introspection through Learned Hidden Markov Models," Artificial Intelligence, Volume 170, Issue 2 (February 2006), pp. 59, 113.
- [3] Diego Ongaro, Alan L. Cox, Scott Rixner, "Scheduling I/O in Virtual Machine Monitors," in VEE '08, Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, 2008, pp. 1,10.
- [4] Hwanju Kim, Hyeontaek Lim, Jinkyu Jeong, Heeseung Jo, Joonwon Lee, "Task Aware Virtual Machine Scheduling for I/O Performance," in VEE '09, Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, 2009, pp. 101, 110.
- [5] Tara M. Madhyastha, Daniel A. Reed, "Input/Output Access Pattern Classification using Hidden Markov Models," in Workshop on I/O in parallel and distributed systems, 1997, pp. 57, 67.
- [6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, Andrew Warfield, "XEN and the Art of Virtualization," in Proceedings of the nineteenth ACM symposium on Operating systems principles, 2003, pp. 164, 177.
- [7] Jian Wang, "Survey of State-of-the-art in Inter-VM Communication Mechanisms", September 2009.
- [8] Jinwoo Kim, Krishna V. Palem, Weng-Fai Wong, "A Framework for Data Prefetching using Off-line Training of Markovian Predictors," in Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '02), 2002, pp. 340, 347.
- [9] Lin Qiao, Balakrishna R. Iyer, Divyakant Agrawal, Amr El Abbadi, "SVL: Storage Virtualization Engine Leveraging DBMS technology," in ICDE '05: Proceedings of the 21st International Conference on Data Engineering, 2005, pp. 1048, 1059.
- [10] Md. Rafiul Hassan, Baikunth Nath, "StockMarket Forecasting using Hidden Markov Models: A New Approach," in Proceedings of the 5th International Conference on Intelligent Systems Design and Applications, 2005, pp. 192, 196.

LIST OF REFERENCES (continued)

- [11] Thorbjorn Donbaek, “ESX Storage Performance Tips”, <http://www.ntpro.nl/blog/archives/342-ESX-Storage-Performance-Tips.html>, Cited January 16, 2008
- [12] VMware, “The Architecture of VMware ESXi”, www.vmware.com, Cited October 14, 2008
- [13] Chris Takemura, Luke S. Crawford, “The Book of XEN: A Practical Guide for the System Administrator”, published by William Pollock, October 2009
- [14] VMware, “Using esxtop to Troubleshoot Performance Problems”, http://www.vmware.com/pdf/esx2_using_esxtop.pdf, Cited December 23, 2004
- [15] David Loshin, “Efficient Memory Programming”, published by McGraw-Hill, November 1998
- [16] Richard O. Duda, Peter E. Hart, David G. Stork, “Pattern Classification”, published by John Wiley and Sons, November 2000
- [17] Ludmila Cherkasova, Diwaker Gupta, Amin Vahdat, “Comparison of the three CPU schedulers in XEN,” in ACM SIGMETRICS Performance Evaluation Review, 2007, pp. 42, 51.
- [18] Kari Johnson, “Hardware, Software Innovations Ushering HPC Capabilities Into Oil and Gas Mainstream”, The American Oil and Gas Reporter, 2009,
- [19] Storage Networking Industry Association, www.snia.org, Cited December 2000
- [20] Padhraic Smyth, “Clustering Sequences with Hidden Markov Models”, Advances in Neural Information Processing Systems, 1997, pp. 648, 654.