Prefetching and **C**lustering **T**echniques for **N**etwork Based Storage

A thesis submitted to Middlesex University
in partial fulfilment of the requirements for the degree of
Doctor of Philosophy

Dhawal Navinbhai Thakker

School of Engineering and Information Sciences
Middlesex University
November 2009

Dedicated to Lord Shri Krishna (Almighty)

Abstract

The usage of network-based applications is increasing, as network speeds increase, and the use of streaming applications, e.g BBC iPlayer, YouTube etc., running over network infrastructure is becoming commonplace. These applications access data sequentially. However, as processor speeds and the amount of memory available increase, the rate at which streaming applications access data is now faster than the rate at which the blocks can be fetched consecutively from network storage. In addition to sequential access, the system also needs to promptly satisfy demand misses in order for applications to continue their execution.

This thesis proposes a design to provide Quality-Of-Service (QoS) for streaming applications (sequential accesses) and demand misses, such that, streaming applications can run without jitter (once they are started) and demand misses can be satisfied in reasonable time using network storage. To implement the proposed design in real time, the thesis presents an analytical model to estimate the average time taken to service a demand miss.

Further, it defines and explores the operational space where the proposed QoS could be provided. Using database techniques, this region is then encapsulated into an autonomous algorithm which is verified using simulation. Finally, a prototype Experimental File System (EFS) is designed and implemented to test the algorithm on a real test-bed.

Acknowledgements

I would like to express my gratitude to all those who gave me the possibility to complete this thesis. I want to thank the School of Engineering and Information Sciences (EIS), Middlesex University for giving me the opportunity to commence this thesis in the first instance, to do the necessary research work and to use departmental resources. I have furthermore to thank the former and current Associate Deans of Research, Prof. Colin Tully and Prof. Richard Comley respectively.

I would like to express my deep and sincere gratitude to my supervisors Dr. Glenford Mapp and Dr. Orhan Gemikonakli from the Middlesex University whose help, stimulating suggestions and encouragement motivated me to complete this thesis. I would also like to thank Prof. Richard Bornat for his constructive suggestions.

My colleagues in the School of EIS supported me in my research work. I want to thank them for all their help, support, interest and valuable hints. I am especially obligated to David Silcott, Amala Rajan, Enver Ever, Yoney Kirsal, Anjum Sheikh and Yonal Kirsal. I also want to thank Michael Robinson and Louise Slabbert for all their assistance in the computer labs.

I am deeply indebted to my parents, brother, Badrinarayan Thakker (uncle) and Chandrika Thakker (aunt), for much of what I have become. I thank them for their patience, love and support. They kept me away from family responsibilities and encouraged me to concentrate on my studies.

I would like to give my special thanks to my family for sponsoring me and giving me an opportunity to come to United Kingdom to study. I thank them for their all support and love which enabled me to complete this work.

List of Publications

The results presented in this thesis have also appeared in the following publications:

- Mapp, G.; Thakker, D.; & Gemikonakli, O, Exploring Gate-Limited Analytical Models for High Performance Network Storage Servers, PMECT, 2009.
- Thakker, D.; Mapp, G. & Gemikonakli, O., Clustering and Prefetching techniques for Network-based Storage Systems, Multi-Service Networks (MSN), 2009.
- Thakker, D.; Mapp, G. & Gemikonakli, O. Modelling mixed access-patterns in Network-Based Systems UKSIM '09: 11th International Conference on Computer Modelling and Simulation, IEEE Computer Society, 2009.
- Thakker, D. & Mapp, G. Clustering and Prefetching techniques for Network-based Storage Systems - II, WITS 2008, 2008.
- Thakker, D. & Mapp, G. Clustering and Prefetching techniques for Network-based Storage Systems - I, IEEE ANTS '08: 2nd IEEE International Symposium on Advanced Networks and Telecommunication Systems (ANTS), IEEE Computer Society, Dec 15-17, 2008, Poster Presentation.
- Thakker, D. & Mapp, G. Balancing streaming and demand accesses in a Network Based Storage Environment CISSE 2008, Online conference.
- Mapp, G.; Thakker, D. & Silcott, D. The Design of a Storage Architecture for Mobile Heterogeneous Devices icns, IEEE Computer Society, 2007.
- Gemikonakli, O.; Mapp, G.; Ever, E. & Thakker, D. Modelling Network Memory Servers with Parallel Processors, Break-downs and Repairs ANSS '07: Proceedings of the 40th Annual Simulation Symposium, IEEE Computer Society, 2007, 11-20.

- Gemikonakli, O.; Mapp, G.; Thakker, D. & Ever, E. Modelling and Performability Analysis of Network Memory Servers ANSS '06: Proceedings of the 39th annual Symposium on Simulation, IEEE Computer Society, 2006, 127-134.
- Mapp, G.; Thakker, D. & Silcott, D. Network Memory Servers: An idea whose time has come Multi-Service Networks (MSN), 2004.

Contents

List of Figures

List of Tables

Glossary

CTIP CTIP is an implementation of TIP over the network. 20

Ext2 Second Extended Filesystem is a file system for the Linux kernel. 1

Ext3 Third Extended Filesystem is a journaled file system that is commonly used by the Linux kernel. 1

Network Memory Server Network Memory Server stores all the data of its clients in the memory of the server. 23

Prefetch Horizon At a certain depth the benefit of prefetching further blocks will be zero, this point is referred to as the Prefetch Horizon. 14, 35

TIP An implementation of Informed Prefetching and Caching was referred to as TIP. 19

Acronyms

EFS Experimental File System. 4

GFS Google File System. 22

HD High Definition. 1

JIT Just-In-Time. 42

LRU Least Recently Used. 14

MMC Mobile Memory Cache. 27

NFS Network File System. 4

NMS Network Memory Server. 3, 4

NSS Network Storage Server. 33

PBM Partial Batch Model. 49

PonD Prefetch-On-Demand Strategy. 5

PSS Persistant Storage Server. 27

RAID Redundant Array of Inexpensive Disks. 2, 3

SAP Systems applications and Products. 3

xFS Serverless Network File System. 23

List of Symbols

C is the time taken to search for the block and copy it into the network buffer. For lightly loaded systems C is assumed to be a constant value. 28

D represents the number of demand blocks being fetched in a network operation. 38, 39

K represents the maximum number of requests that can be served during the current service period. 49

L is the overhead time in going up and down the protocol stack on client and server side and the transmission time over the network. 28

N_s represents the mean queue length of the system. 51

P represents the number of prefetch blocks being fetched in a network operation. 38, 39

$P(T_{CPU})$ is the time taken to fetch a block from the disk divided by the time to access and use the block once the block is in memory. 15

P_{max} is the maximum number of prefetch blocks that could be fetched in an operation. 81

P_{opt} is the optimal number of prefetch blocks that should be fetched in an operation. 81

T_{CPU} represents the time to process/consume a block. 14

T_{D-NMS} represents the average waiting time experienced in satisfying a demand miss using the network memory server. 34

T_{disk} is the latency experienced by the request while using the disk. 14

T_{driver} is the time experienced while servicing a request from the disk due to computational overhead of allocating a buffer and queuing the request at the drive. 14

T_{hit} is the time taken to read a block from the cache. 14

T_{miss} represents the time taken to fetch the block from the disk. 14

$T_{net}(y)$ is the time taken to fetch y blocks from the network memory server. 29

T_{oct} represents the optimal consume time of the streaming applications that could be supported for the given value of $D\,P$ and λ_d. 81

$T_{process}(y)$ is the time taken to consume y blocks by the streaming applications. 34

T_{stall} is the time the application has to wait when the block is not in the cache. 15

$T_{storage}$ represents the average time taken to satisfy a demand request on the commonly used storage device. 40

T_{wait} is the average waiting time experienced by a demand request in the demand queue. 40

n represents the total number of requests in the system. 50

s represents the number of requests being served. 50

Chapter 1

Introduction

1.1 Introduction

Due to the increase in CPU speeds and the amount of main memory available, there has been an increase in the use of multimedia streaming applications in today's computing environment. These applications access files sequentially from the storage device (e.g. disk) and need to be served at constantly high data rates while they are executing, e.g. High Definition (HD) video data rate is 5 MB/Sec, MPEG-4 data rate is 2.5 MB/Sec etc. Also, due to the increase in network speeds, most of these applications are accessed over the network rather than from local storage.

Since these applications access data sequentially, their future access patterns are known, and hence prefetching can be used. Prefetching enables the file system to bring in blocks of data before they are requested. This allows applications to run without waiting for the blocks to be fetched from the storage device i.e. without stalling, thus reducing the latency experienced by the running application. For example, Ext2 and Ext3 file systems, used in the Linux environment, perform static prefetching whereby a maximum depth of 128KB is used to prefetch when sequential access is detected.

Prefetching can only work if it is economical. That is, if more blocks are fetched than requested, the time to fetch additional and requested blocks should be comparable to the time it would take to fetch only the requested blocks. The additional time incurred by prefetching will result in increased latency on waiting requests which might also need to be promptly serviced.

Clustering should be used to minimise the latency experienced. It has the ability to fetch multiple blocks simultaneously thereby reducing the latency when compared to fetching them one block at a time. However, the number of blocks clustered in a request and the reduction in latency depends on the storage mechanism being used. This can be exemplified using disk hardware where the major component of latency is the "seek time", the time to move to the correct track. Once the correct track can be accessed, all the data on that track can be obtained without further seek time. This means that a disk can optimally cluster if multiple requests are laid on the same track. Clustering on a disk can obtain up to 23 MBps compared to 200 KBps[1] when blocks are fetched one at a time.

This observation demonstrates that clustering can provide high data rates, but it is storage dependent. Also it could be exploited by prefetching, as it allows the system to prefetch blocks economically.

In addition to clustering techniques on disk systems, the elevator algorithm, Redundant Array of Inexpensive Disks (RAID) systems and caching can also be used. These techniques also minimise the latency and can achieve high data rates for disk systems. The elevator algorithm rearranges the requests in such a way that the disk will experience a minimal amount of seek time. This is done by arranging requests in increasing or decreasing track numbers, hence minimising the movement of the head. RAID systems can satisfy multiple requests in parallel and so achieve high data rates.

In most operating systems, recent accesses to the disk are stored in a memory cache such as the UNIX buffer cache. Therefore, when blocks are

[1]Assuming 5 millisecond revolution time (latency) to fetch one block

needed by an application, the cache is initially searched in order to satisfy a request. If the block is found in the cache, minimal latency is experienced by the running application[2], if it is not found, it generates a demand miss which must be promptly satisfied.

Using selective or all of the above techniques discussed, file systems can fetch blocks for streaming applications (for sequential accesses), and demand misses from the disk storage system, such that the overall latency experienced by the running applications is minimised.

However, due to the increase in the speed and availability of networks (e.g. 1 Gigabit networks are readily available and the availability of 10 Gigabit network speed is not very far in the future), most of the multimedia and database applications such as BBC iPlayer, YouTube, Systems applications and Products (SAP) etc., can now run over the network. Some of the discussed techniques to obtain high data rates and minimise latency are not relevant in networking environments. Similar techniques, like a RAID system to satisfy multiple requests in parallel, are not readily available for network storage. Caching techniques in network storage systems could remain similar to disk-based systems, as the accessed data could be cached regardless of being fetched from the disk or network storage. In addition, prefetching and clustering techniques could also be used in network storage. As pointed out earlier, the clustering effect depends on the storage mechanism, therefore clustering on different network storage devices needs to be explored.

In general, a network storage server manages a huge amount of memory blocks in the server. These blocks are used by the other computers over the high-speed network. The network storage server only provides blocks of storage to its clients. The blocks could be used by the client to store anything, for example, it could be used to store file system meta-data, to store data from an actual file or to store media data etc. Network storage servers can either work at the file level or at the block level. One such network storage server is the Network Memory Server (NMS) [Mapp et al., 2004], [Mapp et al.,

[2]Obviously, it will take time to search for a block in the cache

2007]. It is being developed by Glenford Mapp, Dhawal Thakker and David Silcott from Middlesex University. The NMS server stores all the data of its clients in the main memory of the server. Hence, the latency experienced in fetching a block of data will be dominated by the characteristics of the network. Unlike the Network File System (NFS), the NMS works at the block level whereas NFS works at the file level.

In this thesis, we propose to explore the network characteristics and clustering effects over the network using the NMS and investigate the operational constraints involved in this set-up. It also analyses different prefetching strategies to come up with a strategy which can exploit the explored clustering effects in order to use the network and buffering systems effectively. The clustering techniques will be applied at the block level while a prefetching strategy will be executed at the file system level. In order to demonstrate and analyse the working of the proposed algorithm, we also propose to develop an Experimental File System (EFS).

The research question to be addressed is:
"Can a file system be developed using clustering and prefetching techniques over the network which can be used in the working environment, to allow streaming applications (once they are started) to run with no jitter[3] while satisfying demand requests in reasonable time?".

1.2 Structure of the Thesis

This thesis is structured in the following way:

- Chapter 2 reviews the literature of most relevance to the research question. This includes the following areas:

[3]Jitter is an unwanted variation observed in video (streaming application)

1. A description of the important prefetching strategies, though most of them are disk-oriented.
2. A review of several major distributed file systems, in order to investigate how these systems implement mechanisms to decrease the latency experienced in fetching data over the network.
3. Finally, it presents the design of the Network Memory Server.

- Chapter 3 introduces the fundamental approach to address the research question and shows how to treat streaming applications and demand misses. It also describes the different prefetching strategies that could be used.

- Chapter 4 reviews different analytical models and then proposes a new analytical model, to estimate the average time to satisfy a demand miss, using a Prefetch-On-Demand Strategy (PonD) over the network.

- Chapter 5 puts together the results achieved from Chapters 3 and 4 and explores the operational space where the required QoS for streaming applications and demand misses could be provided.

- Chapter 6 shows how to use the explored operational space in a real system in order to provide the required QoS. It also demonstrates the design and implementation of an algorithm which satisfies the required constraints.

- Chapter 7 summarises the findings and contributions of this effort and discusses directions for future research.

Chapter 2

Literature Review

INTRODUCTION

Prefetching has proven to be a very effective way of reducing the latency for accessing data on slow devices such as hard drives. Its basic objective is to speed up system calls by prefetching some information into memory before it is required, rather than reading it from disk or over the network when it is required. One of the most important aspects of memory management is its prefetching policies, which have a great impact on file system performance and play a central role in file system research.

2.1 A Study of Integrated Prefetching and Caching Strategies

Prefetching and caching are effective techniques for improving the performance of a file system, but for a long time, they had not been studied in an integrated fashion. Cao et al. [1995], first proposed four rules that optimal integrated strategies for prefetching and caching must satisfy.

The four rules for optimal prefetching and caching are given below:

1. Optimal Prefetching: Every prefetch should bring into the cache the next block in the reference stream that is not (already) in the cache.

2. Optimal Replacement: Every prefetch should discard the block whose next reference is furthest in the future.

   The first two rules uniquely determine what to do, once the decision to prefetch has been made. However, they say nothing about when to fetch: the next two rules address that question.

3. Do No Harm: Never discard block A to prefetch block B when A will be referenced before B. A prefetch that disobeys this rule does more harm than good, as it can only increase the program's running time. Unfortunately, existing prefetching algorithms do not always satisfy this requirement, because they separate caching from prefetching, and separate cache replacement decisions from prefetching decisions.

4. First Opportunity: Never perform a prefetch and replace operation when the same operations (fetching the same block and replacing the same block) could have been performed previously.

   The algorithm must perform each operation at the first opportunity. A new opportunity may arise when either a) a fetch completes or b) the block that would be discarded was just referenced in the previous unit.

Taken together, the four rules provide some guidance on when to prefetch and what to discard. These were integrated into two strategies:

The Conservative Strategy: The conservative prefetching strategy tries to minimise the elapsed time while performing the minimum number of fetches. This means that prefetching is only done when a cache miss occurs.

The Aggressive Strategy: The aggressive prefetching strategy always prefetches the next missing block at the earliest opportunity consistent with the four rules. In order to bring in the next missing block, it replaces the block whose next reference is furthest in the future.

Results showed that both strategies are close to optimal in theory and that these strategies can reduce the running time of applications by up to 60%. This was explored by using an experimental file system, but only static prefetching in which a fixed number of blocks were prefetched was implemented. Our research will explore more dynamic prefetching techniques to improve the performance of the overall system.

2.2 Aggressive Prefetching: An Idea Whose Time has Come

Papathanasiou and Scott [2005] argued that technological trends and emerging system design goals have dramatically reduced the potential costs and increased the potential benefits of highly aggressive prefetching policies i.e., prefetching blocks whenever it is possible, in which prefetching is done at every opportunity. The authors proposed that the memory management of operating systems should to be redesigned to embrace such policies.

Having understood the changes in technological trends, the authors explored different research challenges for computer systems in relation to prefetching. They are listed below:

- Device-Centric Prefetching: Traditionally, prefetching has been application-centric i.e. previous work only explored the control prefetching strategy to minimise the latency of disk accesses experienced by running applications. Such an assumption is not applicable in modern systems. In today's systems, the performance, power, availability and reliability characteristics of devices must be exposed to prefetching algorithms. This point was further explored by Papathanasiou and Scott [2004]. This work is discussed in this literature review.

- Characterisation of I/O demands: Revealing device characteristics is

not enough. To make an informed decision the prefetching and memory management system will also require high level information on access patterns and other application characteristics. Using access pattern information was further explored by Li et al. [2007].

- Coordination: Non-operational low-power modes depend on long idle periods in order to save energy. Uncoordinated I/O activity generated by multitasking workloads reduces periods of inactivity and frustrates the goal of power efficiency.

- Prefetching and caching metrics: Traditionally, cache miss ratios have been used to evaluate the efficiency of prefetching and caching algorithms. The usefulness of this metric, however, depends on the assumption that all cache misses are equal which is not the case. Power efficiency, availability and varying performance characteristics lead to different costs for each miss.

To summarise, Papathanasiou and Scott explored the traditional methods of handling prefetching techniques which were centred on the use of the buffer cache, I/O bandwidth and device congestion. They argued the case that there is a need to change this approach as available resources have increased in volume and speed. For example, 1 GB of Memory and 1 Gbps network speed are readily available on laptops and desktops. Therefore there will be a resultant change in the cost of I/O service time experienced due to each subsystem, for example, prefetching more data into memory would be less costly than disturbing disk idle time or using a network.

This observation motivated us to explore prefetching as it highlighted that resources have improved and hence the need to review how to use them effectively, for example, network and memory. Also, the authors' observation about the increase in the usage of multimedia applications due to improved resources motivated us to explore how we can satisfy the demands for multimedia streaming which require high data rates to allow the user to watch videos without jitter.

2.3 Energy Efficient Prefetching and Caching

Papathanasiou and Scott [2004], proposed new rules for prefetching and caching that maximised power-down opportunities in disk systems without performance loss by creating an access pattern characterized by intense bursts of activity separated by long idle times.

This is done by re-examining the four rules described by Cao et al. [1995]. However, Papathanasiou and Scott replaced the 4^{th} rule and added the 5^{th} rule, to accommodate the requirements of energy efficiency. These rules are stated below:

4' Maximize Disk Utilization: Always initiate a prefetch operation after the completion of a fetch, if there are blocks available for replacement (with respect to rule 3).

5' Respect Idle Time: Never interrupt a period of inactivity with a prefetch operation unless the prefetch has to be performed immediately in order to maintain optimal performance.

Rule 4' guarantees that a soon-to-be idle disk will not be allowed to become inactive if there are blocks in the cache that may be replaced by blocks that will shortly be accessed. This way, disk utilization is maximized and short intervals of idle time that cannot be exploited for energy efficiency are avoided.

Rule 5' attempts to maximise the length of a period of inactivity without degrading performance. Note that the rule implies that the prefetching algorithm should take into account additional delays due to disk activation or congestion as well as the time required for a fetch to be completed.

An algorithm that follows rules 4' and 5' will lead to the same hit ratio and execution time as an algorithm following the rules stated by Cao et al. [1995], but will exhibit fewer and longer periods of disk inactivity whenever possible.

Experimental results showed that, for the slower application, with even a small amount of memory dedicated to prefetching, significant energy savings can be achieved. Just 5 MB of memory prefetching leads to over 50% energy savings, even for "false-positive" (prefetched blocks that are not needed after all) to "true-positive" (prefetched blocks that turn out to indeed be needed)ratio as high as 20 to 1 i.e., even if they prefetch more than 20 times as much data as is actually used. For faster applications, a 50% saving in disk energy can be achieved with a 25 MB prefetch buffer for "false-positive" to "true-positive" ratios of up to 5 to 1. Larger ratios require significantly more prefetch memory.

Papathanasiou and Scott concentrated on saving energy used by the storage device, using prefetching techniques which respect the disk idle time. Our research will explore prefetching techniques for the NMS so that the network is used effectively, so as not to keep the network busy.

2.4 Competitive Prefetching for Concurrent Sequential I/O

During concurrent I/O workloads, sequential access to one I/O stream can be interrupted by accesses to other streams in the system. Frequent switching between multiple sequential I/O streams may severely affect I/O efficiency due to long disk seeks and rotational delays of disk-based storage devices. To overcome these, Li et al. [2007] proposed a competitive prefetching strategy that controls the prefetching depth so that the overhead of disk I/O switching and unnecessary prefetching are balanced. The proposed work does not require a priori information of the data access pattern, and achieves at least half the performance (in terms of I/O throughput) of the optimal off-line policy.

Li et al. presented an analytical model which showed that when the

prefetching depth is equal to the amount of data that can be sequentially transferred within the average time of a single I/O switch, the total disk resource consumption of an I/O workload is at most twice that of the optimal off-line strategy.

In order to accommodate random accesses without penalising the overall performance of the system by prefetching excessively, the competitive prefetching policy employs a slow-start phase i.e., prefetching takes place with a relatively small initial prefetching depth. On the detection of a sequential access pattern, the depth of each additional prefetching operation is increased until it reaches the desired competitive prefetching depth.

Overall evaluation demonstrated that competitive prefetching can improve the throughput of real applications by up to 53%. It did not incur noticeable performance degradation on a variety of workloads. However, the competitiveness of the proposed prefetching strategy is not applicable in the presence of high memory contention, because in the case of extreme contention, the previously cached or prefetched blocks may be evicted before they are accessed.

Li et al. proposed to improve the performance for sequential accesses which is normally affected by frequent I/O switches. Analysing optimal prefetching depth i.e. how much to prefetch, for sequentially accessed data is important. It can increase the throughput of the storage mechanism, allowing applications to run smoothly and efficiently use the available memory without loading up excessive blocks.

Their work focused on storage devices which have disk-like characteristics. Similarly, we would like to explore the prefetching depth for the streaming applications which access data sequentially at high data rates over the network; therefore, prefetching a minimum number of blocks from the server to achieve the same features presented in this work, but for network-based storage.

2.5 Informed Prefetching and Caching

Patterson et al. [1995] described application-disclosed access patterns (hints) to expose and exploit I/O parallelism, and how to dynamically allocate file buffers among four competing demands: prefetching hinted blocks, caching hinted blocks[1] for reuse i.e. the hinted blocks which were prefetched and are already referenced and then cached for future references, caching recently used data for unhinted accesses i.e., cache references which were generated due to demand access, and satisfying demand misses. The approach estimates the impact of alternative buffer allocation strategies on the application execution time and applies cost-benefit analysis to allocate buffers where they will have the greatest impact.

Cost-benefit analysis for I/O management

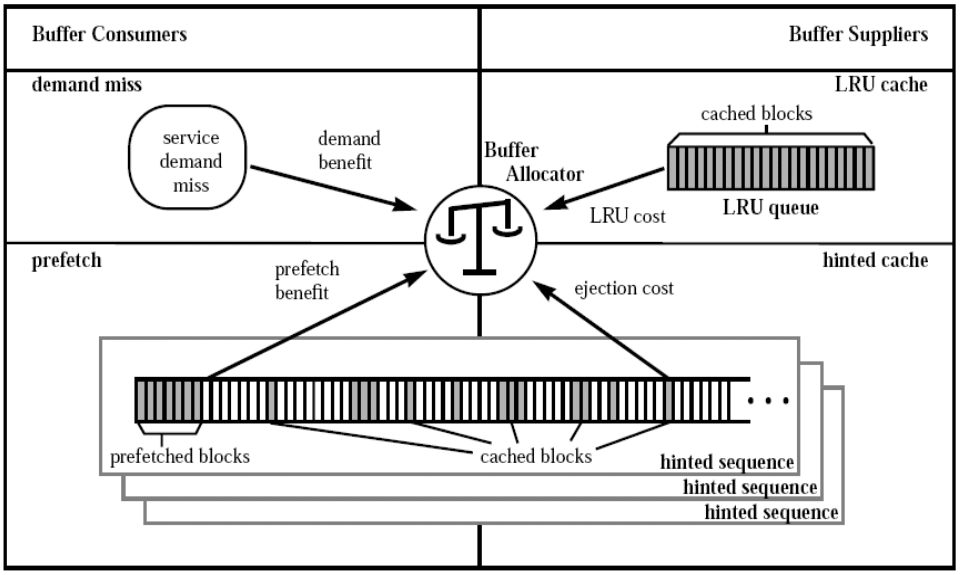

Figure 2.1: Design of Informed Prefetching and Caching

As shown in the Figure 2.1, to acquire cache buffers there are two consumers: demand accesses that are misses in the cache, and the prefetching of hinted blocks. Holding out are two buffer suppliers: the Least Recently

[1]blocks that are prefetched based on access hints and will be referenced in near future

Used (LRU) [2] cache, and the cache of hinted blocks. The I/O manger must resolve this tension between buffer consumers and suppliers, using a concept of prefetch horizon. The Prefetch Horizon is defined as a point at which the prefetching should be initiated for a block, such that it is available for use, just before it is required. Each potential buffer consumer and supplier has an estimator which independently computes the value of its use of a buffer.

The authors analysed each of the estimators, starting with the notion of the prefetch horizon. In their model, a cache hit experiences time T_{hit} to read the block from the cache and the computation time T_{CPU}, i.e., the time to consume/process a block. In the case of a cache miss, T_{miss}, the block needs to be fetched from the disk before it can be delivered to the application. In addition to the latency of the fetch, T_{disk}, these requests suffer the computational overhead, T_{driver}, of allocating a buffer, queuing the request at the drive, and servicing the interrupt when the disk operation completes.

- Prefetch Horizon: The benefit of adding prefetching buffers starts out high, because if a hinted block is not prefetched in time for its consumption, the application will stall for more than T_{disk}[3].

  As the prefetching depth increases, the benefit of increasing it further diminishes quickly. However, after a certain depth the benefit of prefetching further blocks will be zero, this point is referred to as the *Prefetch Horizon*. Prefetching beyond the prefetch horizon will not increase the benefit, because the application will not be able to consume blocks immediately and also the blocks will be in the cache for a longer period of time, occupying cache memory which could be used for caching some other data.

[2]The LRU algorithm keeps an ordered list of cached pages with the most recently used at the top of the list. It replaces pages at the bottom of the list i.e. the least recently used.

[3] T_{disk} is the latency to fetch the block from the disk; this suffers from the computational overhead T_{driver} of allocating a buffer, queuing the request at the drive, and servicing the interrupt when the disk operation completes

Finally, since the authors modelled disk accesses as constant-time operations (where that constant is T_{disk}), and it takes at least T_{hit} to read a block i.e., when it is in the cache, a prefetched block will always arrive in time if it is prefetched at $P(T_{CPU})$:

$$P(T_{CPU}) = \frac{T_{disk}}{T_{CPU} + T_{hit} + T_{driver}} \tag{2.1}$$

$P(T_{CPU})$, is the time taken to fetch a block from the disk divided by the time to access and use the block once the block is in memory. When prefetching is initiated at $P(T_{CPU})$ accesses beforehand, then the stall time experienced by the application in accessing that prefetched block is zero.

- Allocating a buffer for prefetching: The prefetching estimator estimates the benefit of allocating an additional buffer so that the system can prefetch one buffer further ahead of the application's current position. The benefit of using an additional buffer to prefetch one access deeper is the change in the service time:

$$\triangle T_{pf}(x) = T_{stall}(x+1) - T_{stall}(x) \tag{2.2}$$

where T_{stall} is the time the application has to wait, if the prefetch is initiated before x and $x+1$ accesses. To solve the above equation T_{stall} must be calculated.

A key observation is that the application's data consumption rate is finite. Typically, the application reads a block from the cache in time T_{hit}, does some computation, T_{CPU}, and pays an overhead, T_{driver}, for future accesses currently being prefetched. Thus, even if all intervening accesses hit in the cache, the soonest we might expect a block, x accesses into the future, to be requested is $x(T_{CPU} + T_{hit} + T_{driver})$. Under the assumption of no disk congestion, a prefetch of this xth future block would complete in T_{disk} time. Thus the stall time when requesting this

block is at most:

$$T_{stall}(x) \leq T_{disk} - x(T_{CPU} + T_{hit} + T_{driver}) \tag{2.3}$$

The above equation is an upper bound of the stall time experienced by the x^{th} future access assuming that the intervening accesses are cache hits and do not stall. Unfortunately, it overestimates the stall time in practice. In steady state, multiple prefetches are in progress and a stall for one access masks latency for another so that, on average, only one in x accesses experience the stall i.e

$$T_{stall}(x) \leq \frac{T_{disk} - x(T_{CPU} + T_{hit} + T_{driver})}{x} \tag{2.4}$$

When $T_{stall}(x)$ is zero, the distance at which prefetching was initiated for that block is called the prefetch horizon. Now putting Equation 2.4 into Equation 2.2, we have:

$$\triangle T_{pf}(x) = \begin{cases} x = 0 & -(T_{CPU} + T_{hit} + T_{driver}) \\ x < P(T_{CPU}) & -T_{disk}/x(x+1) \\ x \geq P(T_{CPU}) & 0 \end{cases} \tag{2.5}$$

The above equation showed the change in the service time when initiating prefetching for a block at x and $x+1$ distances. When $x = 0$, the change in service time $((x+1) - x)$ is equal to $-(T_{CPU} + T_{hit} + T_{driver})$ (the minus sign indicates that the stall time for $x+1$ is less than x, because the block is being fetched one access earlier). If $x < P(T_{CPU})$, then the increase in the service time is equal to $\frac{-Tdisk}{x(x+1)}$. Finally once $x > P(T_{CPU})$, the change in the stall time is always equal to zero, because the application will not stall once $x > P(T_{CPU})$, so prefetching one access before in this region will have no effect. This is why the authors suggested not to prefetch beyond the prefetch horizon as it does affect the service time.

- Allocating cache blocks for demand reads: An unhinted read or a hinted read whose data has not yet been prefetched is a *demand read*. Because the application will stall indefinitely unless it receives resources to satisfy a demand read, there is no estimation involved with demand reads - it always makes sense to allocate blocks for them.

- Shrinking or growing the LRU cache: The framework also maintains a traditional LRU cache to satisfy many unhinted accesses without fetching from storage. The LRU estimator keeps track of what the hit rate in the LRU cache would be for several LRU cache sizes, and uses this information, coupled with the disk-model cost of an additional cache miss which is:

$$\triangle T_{LRU}(n) = T_{LRU}(n-1) - T_{LRU}(n) \tag{2.6}$$

  where n is the number of buffers in the cache. The LRU cache estimator dynamically estimates the number of hits, when n and $(n-1)$ number of buffers are available. The Equation 2.6 estimates the benefit or cost of making the LRU cache larger or smaller, respectively.

- Shrinking or growing the hinted cache: Finally, the framework tracks cache blocks for which future accesses are hinted. If the system allows such a cached and hinted block to be ejected, it will later have to stall the application for at least $T_{pf}(x) - T_{hit}$ to read the block back into memory i.e., the same block is prefetched again, x accesses before it is required and minus the T_{hit}, as it is known the block is evicted and there is no need to search the cache, which is calculated as:

$$\begin{aligned} \triangle T_{eject}(x) &= T_{pf}(x) - T_{hit} \\ &= T_{driver} + T_{stall}(x) \end{aligned} \tag{2.7}$$

  where $T_{pf}(x) = T_{hit} + T_{driver} + T_{disk}$. Remember, $T_{stall}(x) = 0$, when x is greater than prefetch horizon.

The cost of ejecting a block, $\triangle T_{eject}(x)$, does not affect every access; it only affects the next access to the ejected block. Thus, to express this cost in terms of the common currency, the authors averaged this change in I/O service over the accesses that a buffer is freed.

$$\triangle T_{eject}(x, y) = \frac{T_{driver} + T_{stall}(x)}{y - x} \tag{2.8}$$

where y indicates the block will be read in y accesses and the prefetch happens x accesses in advance.

Since the benefit of avoiding this driver work is amortized over all accesses until the block is fetched, the cost of allowing such a block to be ejected decreases from T_{driver} as the number of accesses before the block will be fetched back increases.

The above framework is for informed prefetching and caching based on a cost-benefit model of the value of a buffer. The authors have showed how to make independent local estimates of the value for caching a block in the LRU queue, prefetching a block, and caching a block for hinted. This framework was further simplified.

To reduce estimation overhead and increase tolerance of both variations in application inter-access computation, T_{CPU}, and the need to prefetch other blocks, the authors assume $T_{CPU} = 0$ and discount the overhead of prefetching other blocks, T_{driver} to arrive at T_{hit}, which is constant. They also used the concepts of ghost buffers and marginal hit ratio [Patterson et al., 1995] to reduce overhead and estimation variations in calculating cost of losing an LRU buffer.

Finally, to eliminate the overhead of determining the value of x dynamically for Equation 2.8, Patterson et al. simplified this expression by assuming that the prefetch will occur at the prefetch horizon and if the block is already within the prefetch horizon, the authors assume that the prefetch will occur at the next access. This simplified version was then implemented

to find out the real time performance of the framework and was referred to as TIP.

Informed prefetching with an least four disks reduces the elapsed time of the applications which include text search, data visualization, database join etc., by between 20% and 85%. For the computational physics application, which repeatedly reads a large file sequentially, OSF/1's aggressive read-ahead does as well as informed prefetching. However, informed caching's adaptive policy devalues recently used blocks more than older blocks and so "discovers" an MRU-like policy, where the most recently used block is first evicted from the cache. This improves the performance by up to 42%.

The idea of having a prefetch horizon i.e., when to initiate prefetching was very interesting, as it allows prefetched blocks to be brought in just before they are needed. This guarantees that the cache will not be overloaded with prefetch blocks, hence will not hurt caching performance and will also allow applications to run without waiting for the request to be fetched from the disk. However, the derived concept of prefetch horizon is disk-oriented.

This model assumes that there are enough disks available to fetch data at any given point of time i.e., no constraints on disk. Based on this assumption, there is no benefit in prefetching beyond the prefetch horizon and it only prefetches one block at a time. However, this may not be true in a real working environment, as the storage device or disk may have a number of requests waiting to be served and we might need to prefetch beyond the prefetch horizon and prefetch more than one block, to allow applications to run without stalling while the storage device is busy serving other requests.

The above work only considered the time to fetch a block from the disk, it did not consider the time taken to consume, T_{CPU}, which is equally important. It assumed T_{CPU} is constant for all applications, which is not true. For example, T_{CPU} of HD stream is $200\mu sec$ whereas T_{CPU} of MPEG-4 video is $400\mu sec$. Our research will explore the relationship between the time to fetch the block and the time to consume that block so that the required

number of blocks are fetched at a time, hence benefiting from clustering rather than just fetching one block at a time.

2.6 Prefetching Over a Network: Early Experience With CTIP

CTIP, by Rochberg and Gibson [1997], is an implementation of a network filesystem extension of the successful TIP informed prefetching and cache management system. CTIP uses hints to aggressively prefetch file data from a NFS file server and to make better local cache replacement decisions.

CTIP is a minimal extension of the TIP model to a network storage system. It treats the network file server as a disk with longer latency. Two modifications to the local TIP system are: the first is the addition of a set of routines for prefetching from an NFS server and the second is a new set of estimates for the time required to retrieve data from storage.

In order to serve multiple requests at a time i.e., to emulate the parallel disk approach, the authors substantially increased the number of NFS I/O daemon processes and threads on both the client (from the suggested 7 to 64), and the server (from the suggested 16 to 70), enough to handle all the asynchronous threads on the client plus a few additional synchronous requests.

Experiments done with this model showed that for remote storage, the hinted versions of the applications experience reductions in elapsed execution time of $17\% - 62\%$. The magnitude of these savings suggests that informed prefetching and caching are worthwhile, even if application data must be accessed over a network. CTIP provides a speed-up of 2.0 compared with a speed-up of 2.2 for TIP, which suggests that hints benefit from local storage about 10% more than they do from remote storage. The point to be noted

Parameter	Local(TIP)	Remote(CTIP)
T_{disk}	13.6ms	15ms
T_{driver}	$580\mu sec$	$877\mu sec$
T_{hit}	$190\mu sec$	$190\mu sec$

Table 2.1: Storage model parameters for TIP and CTIP. Because of a fast network, T_{disk} for the remote case is surprisingly close to T_{disk} for the local case. CTIP must send its request through an expensive set of protocol stacks while TIP uses a faster SCSI stack. Because of this difference, T_{driver}, the client CPU time cost to retrieve an 8 KB block, is substantially higher for CTIP. This increased cost translates into longer run times for remote applications.

is that unlike xFS or NFS, CTIP has the ability to implement prefetching techniques by using daemon processes.

However, the principal limitations of CTIP are increased CPU cost of going through the longer code path required to access network storage as shown in Table 2.6, increased latency of unhinted network accesses, and the standard NFS interface, which makes it difficult to express information about hints, I/O priorities, clustering, and load balancing.

The work showed that by using prefetching techniques the performance of accessing data over the network could be improved. However, the performance result was not good enough compared to TIP due to its inability to do clustering, as it uses NFS which does not have the ability to do clustering because it is a file-based server. In the local system, the I/O system coalesces, or clusters, requests for contiguous blocks into larger requests. This clustering has two effects: it allows the disk to transfer data more efficiently and it allows the client to amortize some of the driver cost over more data. Therefore, our research will explore the characteristics of network, using block-based network storage which will allow us to exploit the effects of clustering.

2.7 Other Distributed File Systems

In this section, the literature will be reviewed in brief covering distributed file systems (DFS):

2.7.1 The Google File System

The Google File System (GFS) [Ghemawat et al., 2003], is a scalable distributed file system for large distributed data-intensive applications including fast web searching. It provides fault tolerance while running on inexpensive commodity hardware, and it delivers high aggregate performance to a large number of clients.

A GFS cluster consists of a single master and multiple chunk servers and is accessed by multiple clients, as shown in the Figure 2.2.

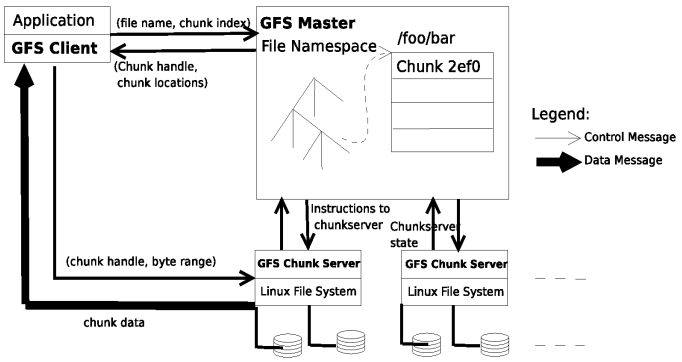

Figure 2.2: GFS Architecture.

Clients interact with the master for meta-data operations, but all data-bearing communication goes directly to the chunk-servers. Most of the applications access huge files sequentially thus neither the client nor the chunk-server caches file data. This simplifies the client and the overall system by eliminating cache coherence issues (clients do cache meta-data, however). By not caching file data, the GFS showed a performance gain for huge files that

are mostly appended to and read sequentially. This work showed that sequential access is becoming important and it should be treated differently. However, it did not look at the prefetching techniques for sequential access.

2.7.2 OceanStore: An Architecture for Global-Scale Persistent Storage

Kubiatowicz et al. [2000] have published a concept for global scale persistent storage by using untrusted infrastructure and by caching data anywhere and at any time. The OceanStore authors have also discussed basic issues such as naming and access control (Restricting Readers and Restricting Writers), data location and routing strategies. Cache coherence is the biggest issue, as the data can be cached anywhere. Basically, the authors of OceanStore have proposed access to data throughout the globe and to cache data anywhere, at anytime to improve the response time and availability of the data.

In general, this idea is similar to our Network Memory Server. However, promiscuous caching, complex cache coherency algorithms and strong encryption must be used in OceanStore, as any machine can cache and serve data. In our NMS, only authorised servers can cache and serve blocks of data to particular clients, hence simpler caching techniques can be used. OceanStore showed that extensive caching is needed to build large geographically distributed file systems. Similar to GFS, OceanStore did not look at prefetching techniques for distributed file systems which is important to improve the performance of sequential access.

2.7.3 Serverless Network File System

The Serverless Network File System (xFS) [Anderson et al., 1995] distributes storage, cache, and control over cooperating workstations. This approach

contrasts with traditional file systems such as Netware [Major et al., 1994], NFS [Sandberg et al., 1985], Andrew [Howard et al., 1987], and Sprite [Nelson et al., 1988] where a central server machine provides all file system services. Such a central server is both a performance and reliability bottleneck. A serverless system, on the other hand, distributes control processing and data storage to achieve scalable high performance, migrates the responsibilities of failed components to the remaining machines to provide high availability, and scales gracefully to system management.

The xFS design attempted to make extensive use of both memory and local on-disk caches at client nodes and used sophisticated cache coherency algorithms to eliminate the need for a central server at the core of the system. BitTorrent, which allows peer-to-peer file sharing, uses similar techniques [Legout et al., 2007]. The xFS work showed that one xFS client can significantly outperform one NFS client by benefiting from the bandwidth (high data rate), of multiple disks and from co-operative caching, therefore removing the need for a central server.

Similar to NFS, xFS communicates at the file level i.e blocks of a particular file at a time. This restricts the implementation of prefetching and clustering techniques which can fetch multiple blocks from different files to achieve high data rates and use network bandwidth effectively.

2.7.4 Caching in the Sprite Network File System

The Sprite network operating system [Nelson et al., 1988] uses large main-memory disk block caches to achieve high performance with its file system. It provides no-write-through file caching on both client and server machines. A simple cache consistency mechanism permits files to be shared by multiple clients without the danger of stale data. In order to allow the file cache to occupy as much memory as possible, the file system of each machine negotiates with the virtual memory system over physical memory usage and

changes the size of file cache dynamically.

To provide good performance under a wide variety of workloads, physical memory on a Sprite workstation is dynamically partitioned between the virtual memory subsystem and the file cache.

Similar to NFS, this work tried to exploit the available network bandwidth by transferring data in bulk. They used large packet sizes, typically 8 KB. Having a large packet size can bring in unnecessary data which may not be required, for example, for random access and could occupy the network for longer periods of time. Therefore, large block sizes can have adverse effects on caching and network performance. Also, this only fetches more data for the same file at a time and not multiple blocks of different files at a time. Therefore, there is a need to explore clustering effects where multiple blocks of different files can be brought into the client, using the caching and networking subsystems effectively.

2.7.5 Recent Research Efforts

CRISP

The Caching and Replication for Internet Service Performance (CRISP) [Gadde et al., 1997] project looked at a new Internet caching paradigm to serve the needs of ISPs with thousands or tens of thousands of users. It used shared caching proxy servers to store read-only objects. The mapping of these objects to the proxy servers where they were stored was done by a mapping service, using central servers.

Results showed that caches for sharing should be large for such communities as data is frequently evicted from a small cache due to its size. For large caches, the more users were allowed to access the caches the higher the degree of sharing among the users.

A trace from the workstations of the Digital Equipment Corporation (DEC), revealed that there was a 37% hit ratio exclusively due to sharing and hits on shared objects accounted for 60% of all hits in the trace. These results showed that caching is extremely important for the future growth of the Internet. However, it did not look at prefetching techniques to further improve the performance.

Parallel NFS

Parallel NFS (pNFS) [Gibson, 2008] is a part of the emerging NFS 4.1 [Shepler et al., 2003] standard which allows NFS clients to directly obtain data from storage servers. In effect, pNFS attempts to separate the meta-data or file management part of NFS from the reading and writing of data. Therefore, the storage servers do not need file-based interfaces but can also have block-based or object-based interfaces which could be used to exploit clustering. This is an emerging standard and therefore it will take some time before it is commonly used but throws a favourable light on this research.

2.7.6 Memory Mobile Memory Cache and the Persistent Storage Server

Mobile computing devices such as smart PDAs and ultra-light laptops with several networking interfaces are becoming commonplace. Users of these devices will expect to be always connected, with seamless switching between available systems. This is being made possible by the development of an architectural framework for heterogeneous networking with support for vertical hand-overs [Mapp et al., August 2006].

The proposed work is also justified by the development and deployment of high-speed networks. Network interfaces of 1-10 Gbps are now commonplace and are fairly inexpensive. In addition, wireless technology has

moved on to 802.11n. This delivers 540 Mbps over-the air speeds and Media Access Control, Service Access Point speeds of 200 Mbps [802]. It should be noted that at these speeds, it is commonly faster to get data from the memory of a remote machine than from a local hard disk [Felton and Zahorjan, March 1991].

As a part of the Network Memory Server project, a design was proposed for a Storage Architecture for Mobile Heterogeneous Devices [Mapp et al., 2007] based on a two-component approach. The first component is the Mobile Memory Cache (MMC) which is a memory server similar to the NMS and the second is a Persistant Storage Server (PSS) which provides persistent storage.

The Design of the Mobile Memory Cache, and the Persistent Storage Server

The MMC has been designed with the following core properties: Firstly, the mobile node viewpoint, all actions performed by its MMC are atomic. This means that calls to the MMC either succeed completely or fail completely. Secondly, the MMC operates in a stateless manner. It does not keep track of former requests from the mobile node. It is assumed that the MMC works over a reliable transport protocol. At present TCP/IP is used.

All references returned by the MMC should be regarded as immutable and must not be changed. The security tag in the BlockID structure will be used to detect when a BlockID has been modified. The core operations supported by the MMC are creating and deleting ClientIDs, creating and deleting data blocks, and reading from and writing to data blocks using BlockIDs.

The design of the Persistent Storage Server is similar to the MMC. However, MMCs are the only clients of a PSS; its services are not directly available to mobile nodes. In addition, while the MMC uses the memory on a

network server, the PSS uses disk blocks on a hard disk to provide persistent storage and so must run a file system in order to achieve this. Therefore, a simple file server known as the Block File System (BFS) was built. The BFS takes a BlockID, a PSS ClientID and maps it to disk blocks on the hard disk. It maintains these mappings in the meta-data part of the system.

In this work, the design of a storage system for mobile devices was proposed and a test-bed was built to evaluate the performance of the MMC. It showed that the performance achieved for reading and writing over the network was comparable with using a local disk. It also showed that by using write buffers, the performance of the write operations could be improved significantly.

In our research, the data for the streaming applications and demand misses should be fetched from the network storage device. To implement and to investigate prefetching / clustering techniques over the network, we require a network based storage system. We will use the MMC or NMS for memory storage. The PSS, which is also referred to as the Network Storage System, will provide persistent storage. In the next section, we present the network characteristics obtained for the Network Memory Server.

Exploring Network Characteristic using NMS

We explored the performance of the NMS by creating large partitions and reading from / writing to these partitions. The cost of using the NMS is composed of a latency cost (L) and a constant cost (C). The latency cost is the overhead time, (going up and down the protocol stack on client and server sides), and the transmission time, (sending a network buffer between client and server). It varies depending on network load. The constant cost[4] is the time taken to search for the block and copy it into the network buffer

[4]Constant cost will vary depending on the number of requests waiting to get served on the server, it is referred as constant because we assume that there is no queue at the server-end

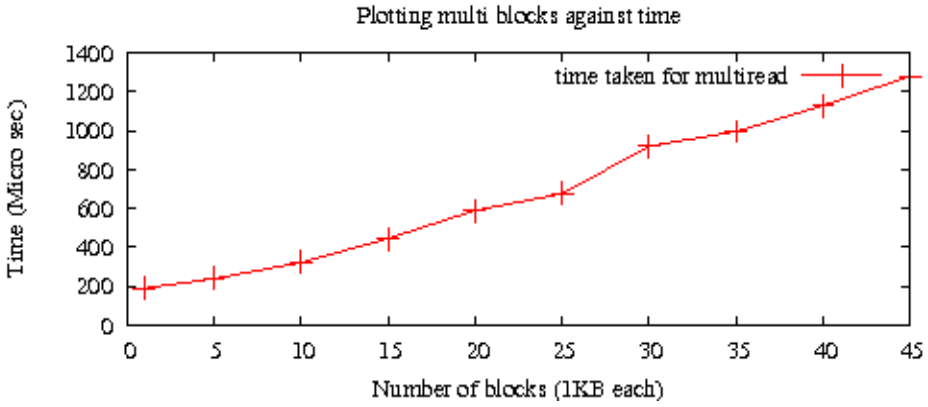

Figure 2.3: Multiple Block Latency

on the server, and to copy the block from the network buffer into memory on the client. These two variables sum to the time $T_{net}(y)$ which is the time taken to fetch y blocks requested in a network buffer, as demonstrated by the formula below:

$$T_{net}(y) = L + Cy \tag{2.9}$$

It has been observed from experiments using the Linux 2.4 kernel platform, that the time to read one block from NMS takes $200\mu sec$ in which 170 $\mu sec(L)$ is the latency and $30\mu sec(C)$ is the constant i.e. 85% overhead to give a data rate of 5 MB/Sec, when fetching one block. The size of each block is 1024 bytes. In this system, if 5 blocks are requested at a time then the transfer rate rises to 16 MB/sec with a latency of $320\mu sec$. Moreover, if 10 blocks are requested at a time then the transfer rate obtained will be 33 MB/Sec with a latency of $470\mu sec$. This analysis and experimental results are shown in Figure 2.3. They indicate that there is a huge clustering effect which can provide high data rates that could be exploited by prefetching. The above experiments were undertaken when there was no other traffic on the network. However, the data rates obtained through clustering could provide sustainable transfer rates for varying network loads.

We will use the above obtained results to provide the proposed QoS for streaming applications and demand misses over the network.

2.8 Summary

The section summarises the reviewed literature.

- Cao et al. [1995], proposed four rules that optimal integrated strategies for prefetching and caching must satisfy. However, the prefetching work was more theoretical and the implemented prefetching strategy was static prefetching i.e., where only a fixed number of blocks are prefetched.

- Papathanasiou and Scott [2005], argued that technological trends and emerging system design goals have dramatically reduced the potential costs and increased the potential benefits of highly aggressive prefetching policies. The authors proposed that memory management needs to be redesigned to embrace such policies.

  These authors also came up with efficient prefetching and caching techniques [Papathanasiou and Scott, 2004] to maximise power-down opportunities, (without performance loss), by creating an access pattern characterized by intense bursts of activity separated by long idle times, thus resulting in saving energy used by disk systems.

- Li et al. [2007] used the knowledge of I/O switch time, to decide how much to prefetch, to improve performance of sequential access.

- Patterson et al. [1995] proposed the notion of prefetch horizon i.e., when to initiate prefetching for a known reference, to use the cache effectively and to minimise the execution time of applications.

  However, the work done by Li et al. and Patterson et al. was disk-centric and both these efforts did not consider the time taken to con-

sume blocks by applications. We will explore some of these ideas for network-based storage and will propose similar techniques which will consider the network characteristics of network storage devices.

- Rochberg and Gibson [1997] extended the work of Patterson et al. by implementing the TIP, (CTIP), framework over the network, but due to the limitation on clustering similar to NFS, this work did not perform as well as TIP.

Distributed file systems such as the Google File System [Ghemawat et al., 2003], OceanStore [Kubiatowicz et al., 2000], the Serverless Network File System [Anderson et al., 1995], and the Sprite network operating system [Nelson et al., 1988] implemented caching techniques to improve response time. They also used large block sizes instead of prefetching or clustering techniques, to utilise the available network bandwidth. However, the previous research work did not look at clustering requests or fetching multiple blocks from different files at the same time.

This research will build upon these efforts by looking at the network characteristics. Most importantly, the Network Memory Server [Mapp et al., 2007], showed that clustering can provide high data rates and hence, will be used to investigate prefetching techniques over the network.

Chapter 3

Approach and Constraints

This chapter analyses the approach and constraints needed to address the research question. The first section of this chapter recalls the research question. The second section presents the environment which will be required to answer the research question. The third section discusses the approach and analyses the constraints for streaming applications. The fourth section discusses the concept of spare-time. The fifth section discusses the approach and investigates the constraints for demand misses. Lastly, the chapter describes different prefetching strategies and their pros and cons. The chapter concludes by describing the need for an analytical model.

3.1 Proposed Work

We propose to investigate prefetching and clustering techniques over the network, to develop a mechanism/algorithm that can guarantee the quality of service for the streaming requests and the demand requests i.e. it should prefetch/cluster enough blocks, so that it can allow streaming applications, (once they are started), to run without jitter [1], while satisfying demand re-

[1] Jitter is an unwanted variation of the inter-arrival time of data observed in streaming applications

quests in reasonable time [2], using the networking and buffering sub-systems effectively.

3.2 Required Environment

In this research, the data for streaming applications and demand misses should be fetched from the network storage device. Hence, the Network Memory Server will be used.

In the NMS, all the data of the client is brought into the memory of the NMS from Network Storage Server (NSS), when the client connects to the NMS. This is similar to Video-On-Demand (VOD) services, where the entire requested video is loaded into the memory of the server before it begins to serve to clients. Hence, the NMS represents the commercial environment. Also, having all the data of the client in the memory of the server before it is requested by the client means that the time taken in fetching the blocks from NMS to the client will largely incur network transaction time, (going to and fro between the client and the NMS), and will be only affected by the network load [3]. Hence, this makes it easier to analyse the parameters involved in fetching blocks from the NMS, as the latency experienced to fetch any block from the memory is same.

In order, to start to analyse the approach / constraint for the streaming application, there should be a way to represent the time taken to fetch y blocks over the NMS. As shown in Equation 2.9, the time to fetch y blocks over the NMS is equal to $L + Cy$, where L is equal to network protocol processing time and transmission time between the client and the NMS and C is the constant time involved in fetching a block, (copying and searching), and could be used to analyse the approach for streaming applications. Similarly,

[2] As long as the time taken to satisfy the demand request over the network is less than or equal to the time taken by disk

[3] Assuming the number of clients connected to the server is reasonable.

for demand misses, let T_{D-NMS} represent the average waiting time experienced in satisfying a demand miss over the NMS and will be used to analyse the approach / constraint for demand misses. Having these parameters for the NMS, we can now begin to look at the approach and constraints involved in providing required QoS. Further details of the NMS can be found in the papers [Mapp et al., 2004], [Mapp et al., 2007].

3.3 Streaming Access

Streaming applications access blocks sequentially at a constant rate. Knowing that streaming applications access blocks constantly over a period of time, we will first derive an equation to analyse how many blocks should be prefetched so that streaming applications can run without jitter once they are started. Streaming applications will not start until enough resources i.e., network and prefetch buffers, are available. We refer to this wait time as *start stall-time*.
As pointed out in the literature review, T_{cpu}, (time to consume one block), is different for each application, therefore we will start with T_{cpu} to find out how long an application will take to consume y blocks.

Let T_{cpu} be the time to consume a block for a streaming application, then the rate at which it will consume y blocks will be $T_{process}(y)$ i.e.,

$$T_{process}(y) = T_{cpu} * y \tag{3.1}$$

From our experiments, we have observed that the time to fetch y blocks over the network is equal to $T_{net}(y)$ which is

$$T_{net}(y) = L + Cy \tag{3.2}$$

where L is the Latency i.e., the time to go up and down the protocol stack

on the client and server, plus the packet transmission time between client and server, y, is the number of block requests in a network buffer and C is the constant time taken to fetch a block from the NMS server. The variable L depends on the network bandwidth available and the load on the network at a given time. Now, for an application to run without any delay or jitter, the time taken to fetch blocks should be less than or equal to the time taken to consume them i.e.,

$$\begin{aligned} T_{net}(y) &\leq T_{process}(y) \\ L + Cy &\leq T_{cpu} * y \\ L &\leq (T_{cpu} - C) * y \\ L/(T_{cpu} - C) &\leq y \end{aligned} \tag{3.3}$$

The equation shows that the number of blocks prefetched for streaming applications should be equal to or greater than $L/(T_{cpu} - C)$. It should be fetched at an interval of $T_{process} * (y)$, to allow them to run without jitter[4].

If $L/(T_{cpu} - C) = y$, only double buffering is needed to satisfy requests for streaming accesses as shown in Figure 3.1, that is to use only two sets of buffers for prefetching and to initiate the next prefetch as soon as the first buffer is available, (at an interval of ($T_{process} * y$)), and this will not cause any jitter/delay in streaming applications, as required blocks will be prefetched just before it is needed, this is similar to the notion of the Prefetch Horizon which was discussed in Informed Prefetching and Caching [Patterson et al., 1995]. Note, having more than two network buffers will be a waste of memory for the above condition, as the time taken to fetch a network buffer is equal to the time taken to process blocks. However, this situation will not give any additional time or slack to satisfy any demand misses or to allow new streaming applications to join, as asking for more blocks than y will increase the fetch time and will incur stall/jitter in the running applications and will not achieve the quality of service needed.

[4]It will experience start stall time to prefetch the first y blocks

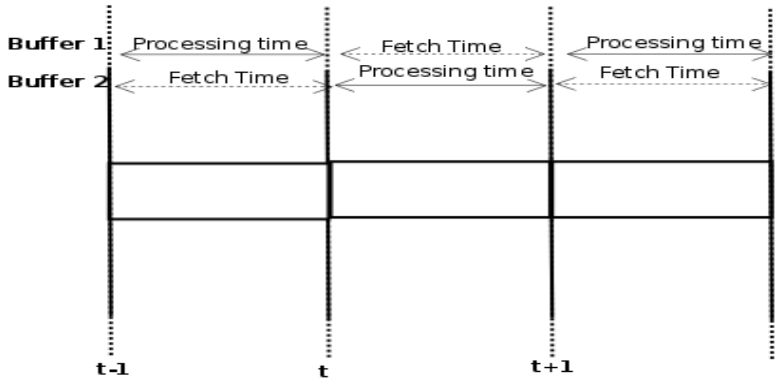

Figure 3.1: Double Buffering in steady state: Processing time is the time taken to consume the block and fetch time is the time taken to fetch the block from the NMS. There is no spare-time as the time to fetch a given number of blocks is equal to the time to process the same number of blocks.

In Informed Prefetching and Caching there was no need to prefetch beyond the prefetch horizon as their work used parallel disks; this allowed them to satisfy demand misses as they occurred, therefore it made sense in their work to not prefetch beyond the prefetch horizon.

However in our case, networks cannot benefit from such parallelism. Hence they are constrained. Therefore prefetching beyond the prefetch horizon would make sense whenever it is possible, to allow streaming applications to run while the demand misses are being satisfied.

The only way to minimise this constraint in the network is to make sure that time taken to process prefetched blocks should be greater than the time taken to fetch prefetched blocks. Hence, in the Equation 3.4, there will not be an equal sign. Also, as the service time over the network varies for each operation, operating at the maximum level would not be an ideal strategy.

$$L/(T_{cpu} - C) \quad < \quad y \tag{3.4}$$

The above condition will leave spare time as shown in Figure 3.2, but having

y much greater than $L/(T_{cpu}-C)$ will increase the start stall-time. Therefore, the value of y should be such that it leaves enough spare-time and it minimises the start stall-time. In short, Equation 3.4 guarantees the quality of service

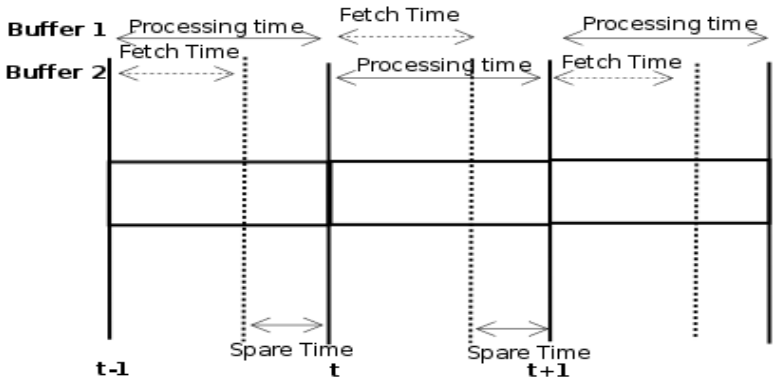

Figure 3.2: Beyond Double Buffering in steady state: Processing time is the time taken to consume the block and fetch time is the time taken to fetch the block from the NMS. Spare time is the time difference between the processing and fetch times, which can allow more than double buffering if needed.

for streaming applications and there will be no overload on the network and buffering sub-systems, as it only prefetches the required number of blocks that are needed for streaming applications and only prefetches more blocks according to requirements of spare-time to satisfy demand misses or to allow new streaming applications to join, thus using buffers effectively.

3.4 Spare-time

The spare-time is very important to provide the required level of quality-of-service. Spare time can satisfy demand misses or allow new streaming applications to join the ongoing prefetching without incurring any jitter in the running applications.

Spare-time can be used to fetch, in the same operation, additional blocks with ongoing prefetch, for demand misses or for new streaming ap-

plications: The number of additional blocks ($N_{AB}(P + D, j)$) (Number of Additional Blocks with Prefetching, can be demand blocks D or for a new stream, (j)) joining the ongoing prefetching P without affecting prefetching for streaming applications, is equal to:

$$N_{AB}(P + D, j) = SP/C \tag{3.5}$$

where C is the constant cost for fetching a block. SP is the spare-time between fetches. The more the spare-time, the more additional blocks could be fetched without penalising streaming applications. $N_{AB}(P+D, j)$ is the maximum number of blocks that could be clustered within an ongoing prefetch cycle without affecting the running streaming applications.
The above equation shows the number of block requests that can be fetched with the ongoing prefetch cycle using spare-time.

The spare-time in the current cycle can be increased by doing more aggressive prefetching i.e., prefetching more blocks than the required number of blocks for the running streaming applications in previous cycles as shown in Figure 3.3. But this is only possible, if the spare-time is available in the first place i.e. if there is no spare time to start with, then additional spare time cannot be created. For example, suppose we have $N_{AB}(P + D, j)$ is

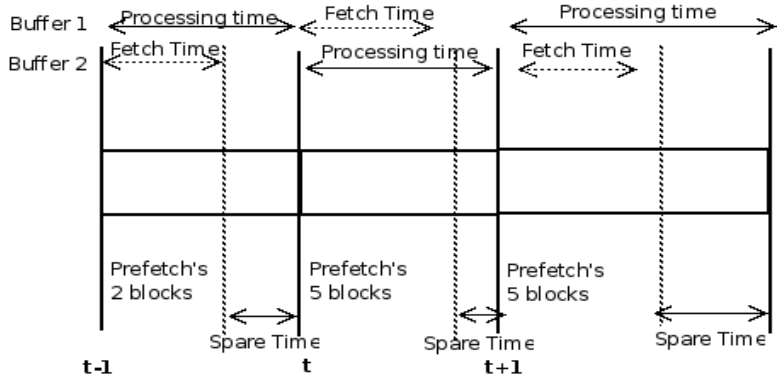

Figure 3.3: Prefetching Beyond Double Buffering: More prefetching increases the fetch time for the cycle t and increases the processing time for cycle $t+1$. Hence, it increases the spare-time from cycle $t + 1$ onwards.

equal to 6 and the ongoing prefetching require 2 blocks at constant interval of ($T_{process} * y$). Now as shown in the Figure 3.3, in cycle $t-1$, it will prefetch 2 blocks. If we prefetch 5 blocks rather than 2 blocks for streaming applications, as in cycle t, then the fetch time will increase for the cycle t. However, fetching more blocks at cycle t will increase the processing time for the cycle $t+1$, as 3 extra blocks are available to consume.

In this example, if $T_{CPU} = 400/mu$sec and $C = 30/mu$sec, then the increase in the spare-time= $3(400-30) = 1110\mu$sec. Analysis showed that the spare-time can also be used to do more prefetching i.e buffering, hence, the Equation 3.5 notation will change to:

$$N_{AB}(P+D, j, b) = SP/C \tag{3.6}$$

where b refers to additional blocks that could be used for buffering i.e more prefetching.

However, the question comes down to, how much spare-time is required for a system to provide the required QoS for a given rate of demand misses. In other words, how many blocks, given by P, should be prefetched for streaming applications and how many blocks, given by D, should be fetched for demand misses in a network operation, for a given demand arrival rate.

3.4.1 Demand Access

Demand accesses are generated by applications without giving any prior notice and will need to be satisfied as soon as possible, as the application may be waiting on that block to continue its execution.

Clustering a demand miss with ongoing fetches will add an additional $C\mu$secs to the fetch operation. To bring in an additional D demand blocks

with ongoing P prefetch blocks will be $T_{demand}(P+D)$:

$$\begin{aligned} T_{demand}(P+D) &= L + C_P + C_D \\ T_{demand}(P+D) &= L + C_{P+D} \end{aligned} \tag{3.7}$$

where $C_P(C*P)$ is the time to prefetch p number of prefetch blocks. The $C_D(C*D)$ is the time to bring in additional demand blocks with ongoing prefetch. The number of demand misses (C_D) fetched should be less than or equal to $N_{AB}(P+D,j,b)$.

Note that the number of requests clustered into a network buffer increases the time to fetch the network buffer, therefore adding more block requests to the network buffer will add more delay to the demand requests. This shows that there is a need to analyse how many requests should be clustered, so that demand misses are not penalised by clustering too many requests.

Let $T_{storage}$ represent the average time taken to satisfy a demand request on the commonly used storage device. T_{wait}, is the average waiting time experienced by a demand request in the demand queue. Now, as long as we can guarantee that the time taken to satisfy each demand request over the network, in this case NMS, T_{D-NMS}, is less than or equal to the average time taken to satisfy a demand request on the commonly used storage device, the Quality-Of-Service will be better than or equal to the commonly used storage device. From this we have,

$$\begin{aligned} T_{D-NMS} &\leq T_{storage} \\ T_{demand}(P+D) + T_{wait} &\leq T_{storage} \end{aligned} \tag{3.8}$$

Equation 3.8 shows that in fetching the network buffer, the sum of the time taken to fetch a demand request, ($T_{demand}(D)$), and the average waiting time (T_{wait}) of that request should be less than or equal to $T_{storage}$ i.e. $T_{D-NMS} \leq T_{storage}$

Substituting the Equation 3.7 in Equation 3.8 for clustering demand misses:

$$L + C_{P+D} + T_{wait} \quad < \quad T_{storage} \tag{3.9}$$

The total number of blocks that could be fetched for a given demand rate and prefetch rate is equal to the number of blocks being prefetched plus the number of blocks that could be fetched using spare time should be less than $T_{storage}$, which is,

$$L + C_{P+D} + T_{wait} + N_{AB}(P + D, j, b) \quad < \quad T_{storage} \tag{3.10}$$

Equation 3.4 and Equation 3.10, will guarantee the quality-of-service for demand requests and for running streaming applications respectively. The analysis indicates how many requests should be clustered into a network buffer so that demand requests do not get penalised too heavily for specific demand miss and prefetch rates.

3.5 Using Different Prefetching Strategies

In the previous section, we analysed the constraints that need to be satisfied by clustering and prefetching techniques in order to provide the required QoS for demand requests and streaming applications. Demand requests cannot be controlled as they happen randomly. Streaming applications consume blocks at a constant rate and therefore the prefetch rate can be controlled. Hence the prefetching could be achieved in a controlled way while making use of resources i.e. network, memory etc., effectively. In order to study the effects of prefetching on the resources, we need to look at well-known prefetching

strategies and they are:

- Aggressive Prefetching: In this strategy, prefetching will be done at the earliest opportunity, i.e. whenever the network and memory are available. Papathanasiou and Scott [2005], proposed that aggressive prefetching is the way forward for today's modern computers, because huge amounts of memory and network bandwidth are now available, and processors are faster.

  However, though network bandwidth and memory have increased, the aggressive prefetching is still not an optimal prefetching strategy, as it will keep the network busy most of the time and could overload the memory. Also, using aggressive prefetching for streaming applications can penalise demand misses heavily. Using simulation, we have studied the effect of aggressive prefetching on the demand misses. This will be discussed later in Section 3.6.

- Just-In-Time (JIT): In this strategy, prefetching will be done such that the block is available in the system just before it is required by the application, as discussed in Informed Prefetching and Caching [Patterson et al., 1995], hence, using the memory and network effectively. This work used the concept of parallel disks in order to satisfy requests. Using parallel disks, a demand block request could be processed instantaneously as there will always be a free disk available to satisfy a request.

  In our work, the data is located over the network in memory servers and once the requests are being processed no other request could be sent until the first set of requests are satisfied. Hence, the JIT prefetching strategy would not be able to guarantee the required QoS, as the network might not be available all the time. However, we will study and compare the effects of JIT strategy on the demand misses using simulation.

- Prefetch-on-Demand miss (PonD): In this strategy, prefetching for the streaming applications will only be done if there is a demand miss

to be serviced. This is also known as conservative prefetching, which was explored in Cao et al. [1995], where Cao tried to minimise the number of fetch operations while reducing the application execution time. The results showed that the conservative strategy performs close to the theoretical optimum. Also, Papathanasiou and Scott [2004], proposed that minimising the fetch operation can result in a reduction of the energy used.

Using the PonD strategy results in the least number of fetches over the network. Thus, it will therefore have minimal effect on the average waiting time experienced by demand misses compared to Aggressive and JIT prefetching strategies. This makes the PonD strategy environmentally friendly and ideal for the network environment.

In order to maintain the QoS for streaming applications using PonD strategy, buffering will be used. Buffered blocks can be used by streaming applications when there are no demand misses. The buffering strategy will be analysed based on the demand rates i.e. the number of prefetch blocks that need to be fetched during an operation will depend on the rate at which demand misses occur.

3.6 Evaluating Prefetching Strategy

In order to study the effects of the three different prefetching strategies above on demand misses, we developed a simulation which can simulate the Network Memory Server environment. Tests were carried out using simulation for different arrival rates of demand misses with each prefetching strategy. The results are shown in Figure 3.4, where *PonD*, represents the Prefetch-On-Demand prefetching strategy, *Aggressive*, represents the Aggressive prefetching strategy and *JIT* represents the Just-In-Time prefetching strategy.

It showed that aggressive prefetching always keeps the network very busy and therefore the average time experienced to service a demand miss,

(T_{D-NMS}), is much higher compared to the PonD and JIT strategies. There is not much difference in the average time experienced to serve a demand miss when using the JIT and PonD prefetching strategies. However, the JIT prefetching strategy will not be able to provide the required level of QoS. The results from the simulation and the work of Pei Cao and Papathanasiou and Scott showed that the conservative prefetching is the way forward, and therefore, we believe that PonD prefetching strategy could be an ideal prefetching strategy in the network environment. Using the PonD strategy

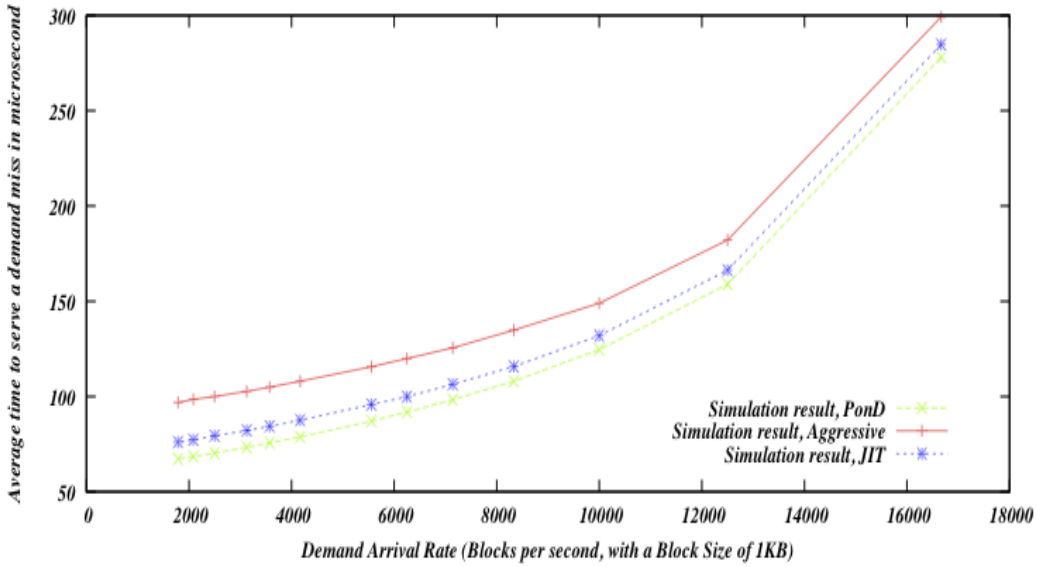

Figure 3.4: PonD vs Aggressive Prefetching: Comparison of average waiting time experienced by demand misses using PonD

shows that networks will be used effectively. However, the effect on memory is still unknown, as more buffering will be required for the lower rates of demand misses and less buffering will be done for higher rates of demand misses.

3.7 Towards an Analytical Model

Prefetching for streaming applications could be controlled, once the total consumption rate of running streaming applications is known. Demand misses are random and need to be satisfied in a reasonable time. The average time experienced to satisfy a demand miss, T_{D-NMS}, needs to be analysed in order to provide QoS for demand misses .

In order to estimate the average time to satisfy a demand miss for a given scenario we propose to develop an analytical model using queueing theory, which can estimate the average time to satisfy a demand miss using the PonD strategy. This will allow us to balance the number of prefetch and demand blocks being fetched in one operation at runtime and will also allow us to analyse boundaries, i.e., how many ongoing streaming applications can be executed and number of demand misses should be satisfied in one operation. Beyond the analysed boundaries, the proposed work will not be able to provide the required quality of service.

3.8 Conclusion

This chapter outlined the required environment and approach that will be used to address the research question. It discussed different prefetching strategies for streaming applications and proposed that the Prefetch-On-Demand strategy is a very good prefetching strategy for a network-based service. Finally, it also showed that there is a need to develop an analytical model for the PonD strategy, to calculate the average time to service demand misses, T_{D-NMS}, and to explore boundaries based on established constraints.

Chapter 4

Analytical Model for Prefetching and Clustering

This chapter describes in detail the analytical model for the Prefetch-On-Demand (PonD) strategy, to estimate the average time to serve a demand miss. Firstly, it starts by describing the standard models. Secondly, it then describes a model which can represent clustering over the network and presents two solutions. Finally, it presents preliminary results of the solved models and concludes by describing the need to explore an operational space where QoS for streaming applications and demand misses could be satisfied.

4.1 Analysis

From the previous analysis, we can represent the system by two queues: the demand queue and the prefetch queue, as shown in Figure 4.1. Let λ_d be the rate at which demand requests are arriving at the demand queue and let λ_p be the rate at which prefetch requests are arriving at the prefetch queue. While serving, more than one request could be taken from both queues, clustered into a network buffer which is then sent off to the server. This can be viewed as a type of *bulk service*.

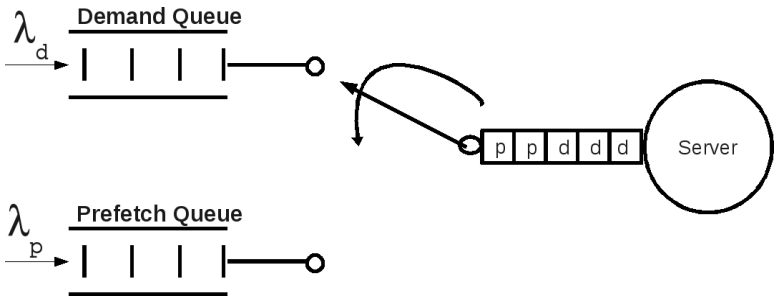

Figure 4.1: A model with a server serving two queues: Prefetch and Demand.

This analysis attempts to answer the question: Given the arrival rates of the two queues, can we find a way to calculate the average service time experienced by demand misses?

4.2 Literature

In order to answer the question, we looked at several polling models. A polling model is a system of multiple queues accessed in a cyclic order by a single server. In recent decades, polling models have been used to analyse the performance of a variety of systems. According to [Takagi, 1988], in the late 1950s, a polling model with a single buffer for each queue was used in an investigation of a problem in the British cotton industry involving a patrolling machine repairman [Mack, 1957a,b]. In the 1960s, polling models with two queues were used to analyse traffic signal control, (see a survey by Stidham, 1969). There were also some early studies from the viewpoint of queueing theory that were apparently independent of traffic analysis (e.g Avi-Itzhak et al., 1965). In the 1970s, with the advent of computer communication networks, an extensive study was carried out on a polling scheme for data transfer from terminals on multidrop lines to a central computer. Since the early 1980s, the same model has been revived by Bux [1981] and others to study token passing schemes, (e.g., the token ring and token bus), in local-area networks (LANs). It has also been used for resource arbitration and load

sharing for multiprocessor computers [Wang and Morris, 1985]. A polling model was used in a non-technical article in Scientific American [Leisowitz and Konheim, 1980], as an example of an interesting and important queuing system.

The usual objective in analysing polling models is to find the *message waiting time*, defined as the time from the arrival of a randomly chosen message to the beginning of its service. The mean waiting time plus the mean service time is the mean message response time, which is the single most important performance measure in the most computer communication systems [Kleinrock, 1976].

Polling models are referred to in many survey articles and books on data communication systems such as Chu and Konheim [June 1972.], Dimitri and Robert [1992], Hayes and Sherman [November 1971], Kaye and Richardson, Kobayashi [Jan 1977], Leisowitz and Konheim [1980], Penny and Baghoadi [1979], Reiser [1982], Simon LAM January and Simon S. LAM [1983].

The vast majority of the literature is concerned with the two traditional service disciplines, the *exhaustive* and *gated* policies. Exhaustive service means that a queue must be empty before the server moves on, whereas in the case of gated service only those customers in the queue at the start of polling are served. The main drawback of these traditional policies [Takagi, 1990, 1988, 2000] is the inability to prioritise among the different queues for improving total system performance. A more sophisticated service strategy offering this possibility is the *K-limited* service strategy. Under this strategy the server continues working at a queue until either a predefined number of K customers is served or until the queue becomes empty, whichever occurs first. Note that the case $K \longrightarrow \infty$ is equivalent to the exhaustive service strategy. In many applications of polling systems, the objective function typically depends not only the mean queue lengths, but on the complete marginal queue length distributions. Therefore, Vuuren and Winands, 2006, proposed to study the marginal queue length distributions in a continuous-

time polling systems with K-limited service under the assumption of general arrival, service and set-up distributions.

A feasible approximate approach for the queue length distribution in a K-limited polling system is the decomposition method, in which the polling system is decomposed into vacation systems, for which the vacation distributions are computed in an iterative approximate manner. At each step in the iteration, the mathematical analysis focuses on one single queue, whereas the other queues in the system determine the length of the vacation period. We have to remark that these decompositions methods seem to be applicable to a wide variety of queueing systems (e.g., [Dallery et al., 1989, Gershwin and Burman, 2000, Van Vuuren and IJBF, 2006, Van Vuuren et al., 2005]). However, the main disadvantage of this method is that time and memory requirements on computational resources are exponential functions of the number of queues.

In our work, we believe that the streaming applications access blocks at a constant rate. The number of blocks needed to be fetched for the prefetch queue to provide the required QoS can be controlled. Hence, we can reduce the model to a single-queue system based on demand requests but the service time for the demand blocks will include the cost of fetching prefetch blocks. We can further simplify the analysis by only prefetching when there are demand requests in the demand queue i.e. conservative prefetching as shown by Cao et al. [1995]. This strategy is also justified as it uses the network more effectively.

4.3 Standard Approach (Partial Batch Model)

As a first step in analysing the average time to satisfy a demand request in the demand queue, we will use the Partial Batch Model (Partial Batch Model (PBM)) described in [Gross and Harris, 1998]. In this model a server can serve up to a maximum of K requests. If there are less than K requests

in the system, the server begins service these requests. Furthermore, when there are less than K requests being serviced, new arrivals immediately enter service. The amount of time required to service requests, is an exponentially distributed random variable with mean $\frac{1}{\mu}$.

This model is represented in Figure 4.2. Each state of the model is represented in terms of n and s. n is the total number of requests in the system and s is the number of requests currently being served. It can be seen that any new arrival enters the service immediately as long as there are less than K number of requests being served and time taken to service those requests is exponentially distributed to a mean value of $\frac{1}{\mu}$. A stochastic

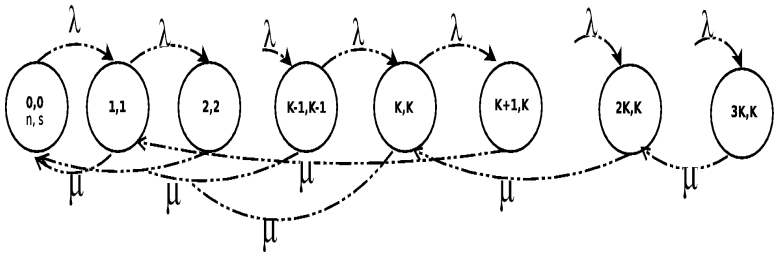

Figure 4.2: Partial Bulk Service model.

balance equation for the model can be written as:

$$\begin{aligned} 0 &= -(\lambda+\mu)p_n + \mu p_{n+K} + \lambda p_{n-1} \quad (n \geqslant 1) \\ 0 &= -\lambda p_0 + \mu p_1 + \mu p_2 + ... + \mu p_{K-1} + \mu p_K \end{aligned} \tag{4.1}$$

For $K = 1$;

$$\begin{aligned} 0 &= -(\lambda+\mu)p_n + \mu p_{n+1} + \lambda p_{n-1} \\ 0 &= -\lambda p_0 + \mu p_1 \end{aligned} \tag{4.2}$$

The above equations are the basic equations for the M/M/1 queue. Hence,

we can say the solution for the Partial Batch Model is same as the M/M/1:

$$p_n = p_0 r^n$$

By finding the root (r) of this equation that is between 0 and 1, one can work out the mean queue length, N_s and the average waiting time (W) for the queue, using the equations below.

$$N_s = \frac{r}{1-r} \quad \text{and } W = \frac{r}{\lambda(1-r)} \tag{4.3}$$

The results presented in Figure 4.6, showed that this approach is extremely accurate for very heavy traffic, since on these occasions the server will always be serving the maximum batch size. However, for lighter traffic loads the model is inaccurate because according to this approach new requests will immediately enter service when the server is serving less than the maximum batch size which is not the case in our scenario. Here, the server only serves the number of requests in the queue at its arrival, requests arriving after this point must be serviced in the next cycle regardless of whether or not the maximum batch size is being served in the current cycle. Hence, the scenario is gate-limited and not exhaustive-limited as seen in the Partial Batch Model.

4.4 Proposed Gated-Limited Model

In this section we attempt to develop a more accurate model which could be used under operational loads. As shown in Figure 4.3, the state of the model is defined by two variables i.e. n and s. n is the total number of requests in the system including the requests being served and s is the number of requests being served at any given time. Therefore, for the maximum batch size $s =$

K, s goes from 0 to K, so when $s = 0$, the system is empty and when $s = K$ up to K requests are being served at a time. For reasons of tractability, the network buffer is assumed to be of infinite length. Also, excluding $(0, 0)$, this will give rise to the K different stages as shown in Figure 4.3 with each stage having a service rate depending on the number of blocks being served.

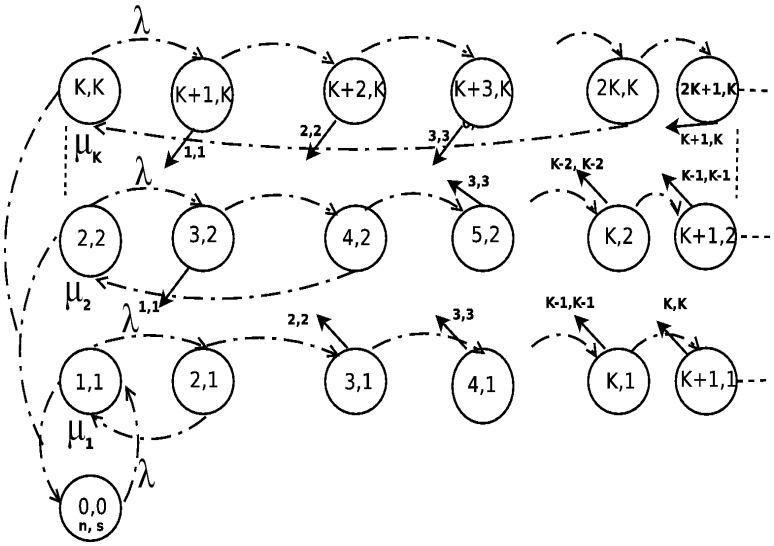

Figure 4.3: A model with a server which can serve up to K demand requests in batch mode, n = the total number of requests in the system and s = the number of requests being served.

4.4.1 Simple Scenario

We start by looking at a simple scenario by restricting K to 2, i.e. $s = 2$, as shown in Figure 4.4. Having K equal to 2 there can be only three stages: either the server is serving 1 request or it is serving 2 requests or the queue is empty. This means that with the exception of the transition, $(2, 2)$ to $(0, 0)$, each transition can only jump one stage at a time (i.e. 1 to 2 or 2 to 1) for e.g. $(3, 1)$ goes to $(2, 2)$ or $(3, 2)$ goes to $(1, 1)$. We will analyse each series individually starting with Series 1.

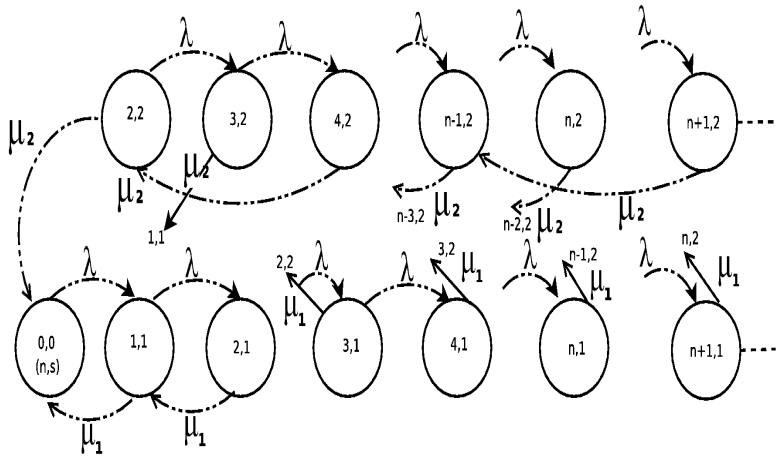

Figure 4.4: Two Stage Model, $K = 2$.

Considering Series 1 **i.e. when** $s = 1$

Let us consider Series 1 of the Figure 4.4. In Series 1, $s = 1$ and for $n > s$ i.e. $n > 1$, we will have:

$$\lambda p_{n-1,1} = (\lambda + \mu_1) p_{n,1} \tag{4.4}$$

This implies that for any $n > 1$, in Series 1:

$$\begin{aligned} p_{n,1} &= \frac{\lambda}{(\lambda + \mu_1)} (p_{n-1,1}) \\ p_{n,1} &= (\frac{\lambda}{(\lambda + \mu_1)})^{n-1} (p_{1,1}) \end{aligned} \tag{4.5}$$

And for $n = s$, i.e., $n = 1$, we have:

$$(\lambda + \mu_1) p_{1,1} = \lambda p_{0,0} + \mu_1 p_{2,1} + \mu_2 p_{3,2} \tag{4.6}$$

Finally, for $n = s = 0$, i.e., $p_{0,0}$ will be:

$$\lambda p_{0,0} = \mu_1 p_{1,1} + \mu_2 p_{2,2} \tag{4.7}$$

Considering Series 2 i.e. when $s = K = 2$

Similarly, for $s = 2$, we will derive equations for $n > s$ and $n = s$, using Figure 4.4. when $n > s$, we have:

$$(\lambda + \mu_2) p_{n,2} = \lambda p_{n-1,2} + \mu_2 p_{n+2,2} + \mu_1 p_{n+1,1} \tag{4.8}$$

And for $n = s$, we have:

$$(\lambda + \mu_2) p_{2,2} = \mu_2 p_{4,2} + \mu_1 p_{3,1} \tag{4.9}$$

Now using the derived equations for Series 1 and Series 2, we will try to obtain an equation for Series 2 at point $p_{3,2}$[1]. We can find out the roots of these equations as in the Partial Batch Model and thus will be able to find out the probability of being at each point in Series 2.

$$(\lambda + \mu_2) p_{3,2} = \lambda p_{2,2} + \mu_2 p_{5,2} + \mu_1 p_{4,1} \tag{4.10}$$

From Equation (4.5), $p_{4,1}$ can be expressed as $(\frac{\lambda}{\lambda+\mu_1})^3(p_{1,1})$. Further, from Equation (4.7), $\mu_1 p_{1,1}$ can be expressed as $\lambda p_{0,0} - \mu_2 p_{2,2}$. Therefore,

$$\mu_1 p_{4,1} = (\lambda p_{0,0} - \mu_2 p_{2,2})(\frac{\lambda}{\lambda + \mu_1})^3 \tag{4.11}$$

[1]Similar techniques can be used for different points of Series 2, e.g. $p_{2,2}$, $p_{4,2}$, $p_{5,2}$

Substituting the value of $p_{4,1}$ into the Equation 4.10, we get:

$$\begin{aligned}(\lambda+\mu_2)p_{3,2} &= \lambda p_{2,2} + \mu_2 p_{5,2} \\ &\quad + (\lambda p_{0,0} - \mu_2 p_{2,2})(\frac{\lambda}{\lambda+\mu_1})^3 \\ 0 &= -(\lambda+\mu_2)p_{3,2} + \lambda p_{2,2} + \mu_2 p_{5,2} \\ &\quad + (\lambda p_{0,0} - \mu_2 p_{2,2})(\frac{\lambda}{\lambda+\mu_1})^3\end{aligned} \tag{4.12}$$

Now, we need to find the root r which is between 0 and 1 such that it solves the above equation, as in the Partial Batch Model.

4.5 First Attempt to Solve Series 2

In this first attempt, we primarily wanted to determine whether our proposed model was better than the PBM model. The approach taken was to regard our model as two separate and independent chains/series both emanating from $p_{0,0}$, so the idea is to be able to express all the values of all the states in terms of $p_{0,0}$.

By finding the root (r) which will be between 0 and 1, we can find the probability of being at each point in Series 2 in terms of $p_{0,0}$. We will use the same approach as in the M/M/1 queueing as well as the Partial Batch Model and so we will express $p_{n,2}$ in terms of $p_{0,0}$ as follows:

$$p_{n,2} = r^n p_{0,0} \tag{4.13}$$

Similarly, using Equation (4.7), we can find out the probability of being at each point in Series 1 in terms of $p_{0,0}$.

$$
\begin{aligned}
\lambda p_{0,0} &= \mu_1 p_{1,1} + \mu_2 p_{2,2} \\
p_{1,1} &= \frac{(\lambda - \mu_2 r^2)}{\mu_1} p_{0,0} \quad \text{(substituting } p_{2,2} = r^2 p_{0,0}\text{)} \\
p_{1,1} &= C_{1,1} * p_{0,0} \quad \text{where } C_{1,1} = \tfrac{(\lambda - \mu_2 r^2)}{\mu_1} \\
&\textit{and} \\
p_{n,1} &= (\frac{\lambda}{(\lambda + \mu_1)})^{n-1} * C_{1,1} * p_{0,0} \\
&\quad \text{(substituting value of } p_{1,1} \text{ in Equation 4.5)}
\end{aligned} \tag{4.14}
$$

From Equations (4.13) and (4.14), we can see that the probability at any point in the model can be known if $p_{0,0}$ is known. Also, the sum of all the probabilities should be equal to 1. Hence, we sum the two independent chains as follows:

$$\sum_{n=0}^{\infty} p_{n,1} + \sum_{n=0}^{\infty} p_{n,2} = 1 \tag{4.15}$$

Let S_1 be equal to $\sum_{n=0}^{\infty} p_{n,1}$. From Equation 4.14, we have:

$$
\begin{aligned}
S_1 &= \sum_{n=0}^{\infty} (\frac{\lambda}{(\lambda + \mu_1)})^{n-1} * C_{1,1} * p_{0,0} \\
&= \frac{C_{1,1} p_{0,0} (\lambda + \mu_1)^2}{\lambda \mu_1}
\end{aligned} \tag{4.16}
$$

Similarly, let S_2 be equal to $\sum_{n=0}^{\infty} p_{n,2}$. From Equation 4.13, we have:

$$\begin{aligned} S_2 &= \sum_{n=0}^{\infty} r^n p_{0,0} \\ &= \frac{1}{1-r} p_{0,0} \end{aligned} \tag{4.17}$$

$$\begin{aligned} S_1 + S_2 &= 1 \\ p_{0,0}(\frac{C_{1,1}(\lambda+\mu_1)^2}{\lambda\mu_1} + \frac{1}{1-r}) &= 1 \\ p_{0,0} &= \frac{\lambda\mu_1(1-r)}{(\lambda+\mu_1)^2 * C_{1,1}(1-r) + \lambda * \mu_1} \end{aligned} \tag{4.18}$$

where r is the root between 0 and 1 of Equation 4.12 expressed in terms of 4.13.

Once we know $p_{0,0}$, the total number of requests, (N_s), in the system can be calculated using:

$$N_s = \sum_{n=0}^{\infty} n * p_{n,1} + \sum_{n=0}^{\infty} n * p_{n,2} \tag{4.19}$$

and the average time to serve a demand miss will be equal to W:

$$W = \frac{L}{\lambda} \tag{4.20}$$

4.5.1 Simulation

A simulation was developed to verify the analytical model results. It is written in C++ and is a discrete event based simulation. There are two types of events supported and they are *service* and *arrival* events.

On an arrival event, the next arrival event is generated and the current arrival event is placed in the demand queue and the time of arrival is also stored. A service event indicates that the request at the head of the queue has been served and its waiting time is calculated. Further, if the queue is empty the server is set to free or the next service event is generated (depending on the values of P, D, L, C). This procedure continues until the required number of jobs have been served. The flow chart of the simulation is shown in the Figure 4.5

In the simulation the values of P, $D(MaxD), L$ and the iteration ($JOBS$) can be set. The full code for the simulation can be found in APPENDIX C.

4.5.2 Results of the First Attempt

In the experiment, the number of prefetch blocks (P) was kept constant, ($P = 1$), and the arrival rate of the demand queue was varied. The analytical results were calculated using different points at Series 2, however, $p_{4,2}$ appeared to give the best results. The results estimate the average time to serve a demand miss, (T_{D-NMS}), by the Partial Batch Model and the Proposed Model, are shown in Figure 4.6. The simulation points in the graph have 95% of confidence level with confidence interval of $\pm 5\%$. It shows that the results from the analytical model are significantly better than the Partial Batch Model.

The results from the first attempt show that the model appears to be very accurate at medium and high loads but inaccurate at lower loads.

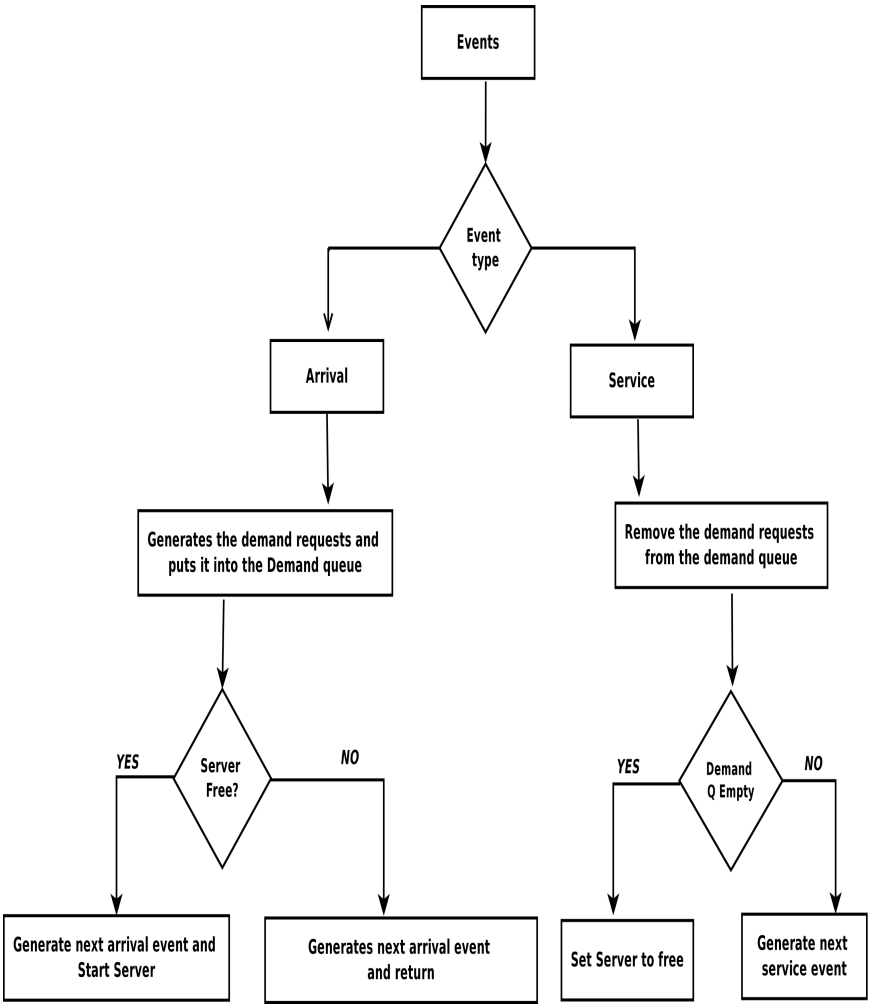

Figure 4.5: Working Of Simulation

This is inadequate as it is unable to estimate the average time to serve a demand miss over the network when operating over a large operational range of λ_d. In addition, at very high non-operational loads the model overshoots the PBM and the simulation (not shown above but in the UKSIM paper [Thakker et al., 2009]). We therefore need to revisit the approach of treating the model as two independent series based around $p_{0,0}$. Instead, we base each series on its first element, i.e., $p_{1,1}$ and $p_{2,2}$, and we use Equation 4.7 to include $p_{0,0}$. This allows us to come up with another solution based on the state of $p_{2,2}$ instead of $p_{0,0}$.

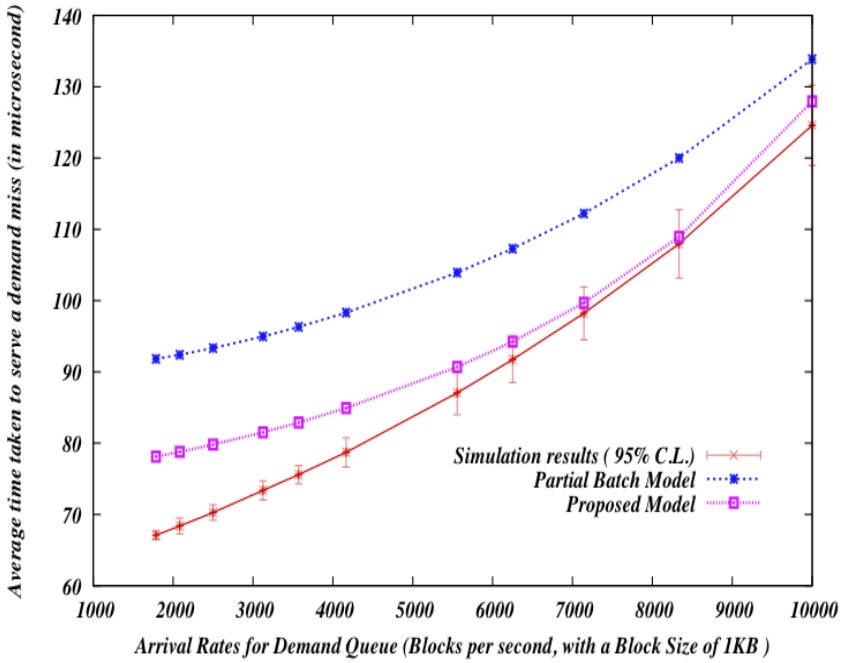

Figure 4.6: Estimates the average time to serve a demand miss (T_{D-NMS}) using Simulation, Partial Batch Model and First Attempt (Our Model).

4.6 Second Attempt to Solve Series 2

In this section, we attempt to solve the Equation 4.12 based on the state $p_{2,2}$. First, we find out the roots of this equation using the same technique that was used in the Partial Batch Model. Thus the state probabilities of Series 2 for $n >= 2$ can be given by:

$$p_{n,2} = r^{n-2} p_{2,2} \tag{4.21}$$

In order to solve the above equation, we assume the second Series to be identical to a Partial Batch Model represented by Equation 4.12. This is shown in Figure 4.7. However, for states where $n > 2$, there is no real difference between the real or imaginary Series as Equation (4.21) is valid in

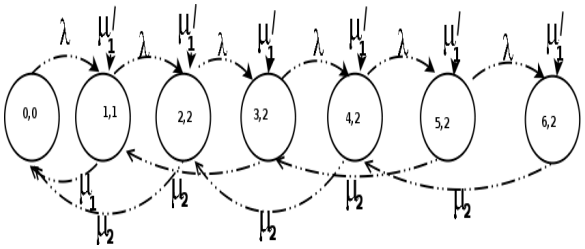

Figure 4.7: Imaginary Partial Batch Model for Series 2

both scenarios. This means we can use the same approach as was taken in the Partial Batch Model to calculate r. Once this is done we can represent any state in the second Series by the Equation 4.21. In addition, using the previous equations, it will also be possible to represent $p_{0,0}$ and Series 1 in terms of $p_{2,2}$.

Using Equation 4.7, we substitute for $\lambda p_{0,0}$ in Equation 4.6. In addition, we note that according to Equation (4.21): $p_{3,2} = rp_{2,2}$. Rearranging, we get: $p_{1,1} = C_{1,1}\ p_{2,2}$ where $C_{1,1}$ is given by the equation:

$$C_{1,1} = \mu_2(1+r)(\frac{\lambda+\mu_1}{\lambda^2}) \tag{4.22}$$

By substituting for $p_{1,1}$ in Equation 4.7, we can get an equation for $p_{0,0}$ in terms of $p_{2,2}$; i.e., $p_{0,0} = C_{0,0}\ p_{2,2}$ where $C_{0,0}$ is given by:

$$C_{0,0} = \frac{\mu_1 C_{1,1} + \mu_2}{\lambda} \tag{4.23}$$

4.6.1 Solving for $p_{2,2}$

The sum of all the state probabilties must be equal to 1. Let S_1 be the sum of the state probabilities for Series 1 and S_2 be the sum of the state probabilities in Series 2. So we can write:

$$p_{0,0} + S_1 + S_2 = 1 \tag{4.24}$$

where:

$$S_1 = \sum_{n=1}^{\infty} (\frac{\lambda}{(\lambda + \mu_1)})^{n-1} p_{1,1} \tag{4.25}$$

$$S_2 = \sum_{n=2}^{\infty} r^{n-2}\ p_{2,2} \tag{4.26}$$

For S_1, let $m = n - 1$ and substituting for $p_{1,1}$

$$S_1 = \frac{\lambda + \mu_1}{\mu_1} C_{1,1} p_{2,2} \tag{4.27}$$

Similarly for S_2, let $m = n - 2$

$$S_2 = \frac{1}{1 - r} p_{2,2} \tag{4.28}$$

Summing to one we get:

$$p_{2,2} = \frac{1}{C_{0,0} + \frac{\lambda+\mu_1}{\mu_1} C_{1,1} + \frac{1}{1-r}} \tag{4.29}$$

Using the value of $p_{2,2}$ in Equation 4.22 and 4.23 , we can find values for $p_{1,1}$ and $p_{0,0}$.

The average number of requests in the queue can be expressed as:

$$N_s = \sum_{n=1}^{\infty} n(\frac{\lambda}{(\lambda+\mu_1)})^{n-1} p_{1,1} + \sum_{n=2}^{\infty} n r^{n-2} p_{2,2} \tag{4.30}$$

We can further obtain an exact formula for N_s, as shown in the section below.

4.6.2 Further Solving for N_s

From Equation 4.30, we first solve for the first term on the Right Hand Side of the equation and then second term.

$$\sum_{n=1}^{\infty} n(\frac{\lambda}{(\lambda+\mu_1)})^{n-1} p_{1,1} \tag{4.31}$$

Let $q = \frac{\lambda}{\lambda+\mu_1}$

$$= \sum_{n=1}^{\infty} n q^{n-1} p_{1,1} \tag{4.32}$$

Now, $n * q^{n-1} = \frac{d}{dq}q^n$

$$\begin{aligned} &= \sum_{n=1}^{\infty} \frac{d}{dq} q^n p_{1,1} \\ &= \frac{d}{dq} \sum_{n=0}^{\infty} q^n p_{1,1} \\ &= \frac{d}{dq}(\frac{1}{1-q})p_{1,1} \quad (\text{substituting} \sum_{n=0}^{\infty} q^n = \frac{1}{1-q}) \end{aligned}$$

$$\sum_{n=1}^{\infty} n(\frac{\lambda}{(\lambda+\mu_1)})^{n-1} p_{1,1} = \frac{1}{(1-q)^2} p_{1,1} \tag{4.33}$$

Now solving the second term on the Right Hand Side of the Equation 4.30.

$$\sum_{n=2}^{\infty} n r^{n-2} p_{2,2} = \sum_{n=2}^{\infty} (n-1) r^{n-2} p_{2,2} + \sum_{n=2}^{\infty} r^{n-2} p_{2,2} \tag{4.34}$$

In order to present the solution of Equation 4.34 in simple form, we will again solve the terms in the Right Hand Side one by one, starting with the second term on the Right Hand Side of Equation 4.34.

$$\begin{aligned} &\sum_{n=2}^{\infty} r^{n-2} p_{2,2} \\ &= \sum_{q=0}^{\infty} r^q p_{2,2} \quad (\text{ substituting } q = n-2 \text{ }) \\ &= \frac{1}{1-r} p_{2,2} \quad (\text{substituting} \sum_{q=0}^{\infty} r^q = \frac{1}{1-r}) \end{aligned} \tag{4.35}$$

Now, solving the first term on the Right Hand Side of Equation 4.34:

$$\sum_{n=2}^{\infty}(n-1)r^{n-2}p_{2,2}$$

$$\begin{aligned}
&= \sum_{n=2}^{\infty}\frac{d}{dr}r^{n-1}p_{2,2} \quad (\text{substituting } (n-1)r^{n-2} = \tfrac{d}{dr}r^{n-1}) \\
&= \frac{d}{dr}\sum_{n=1}^{\infty}r^{n-1}p_{2,2} \\
&= \frac{d}{dr}\sum_{q=0}^{\infty}r^{q}p_{2,2} \quad (\text{substituting } q = n-1\ ) \\
&= \frac{d}{dr} * \frac{1}{1-r}p_{2,2} \quad (\text{substituting } \sum_{n=0}^{\infty}r^{q} = \frac{1}{1-r}) \\
&= (\frac{1}{(1-r)^2})p_{2,2} \quad (\text{substituting } \tfrac{d}{dr}\tfrac{1}{1-r} = \tfrac{1}{(1-r)^2}) \quad (4.36)
\end{aligned}$$

The results expressed in the Equations 4.35 and 4.36 showed that Equation 4.34 can be expressed as:

$$\begin{aligned}
\sum_{n=2}^{\infty}nr^{n-2}p_{2,2} &= ((\frac{1}{1-r}) + (\frac{1}{(1-r)^2})) * p_{2,2} \\
&= (\frac{1-r+1}{(1-r)^2}) * p_{2,2} \\
\sum_{n=2}^{\infty}nr^{n-2}p_{2,2} &= (\frac{2-r}{(1-r)^2}) * p_{2,2} \quad (4.37)
\end{aligned}$$

Equation 4.30 can be expressed as:

$$N_s = \frac{1}{(1-q)^2}p_{1,1} + (\frac{2-r}{(1-r)^2}) * p_{2,2} \quad (\text{ From Equations } 4.33 \text{ and } 4.37\ ) \quad (4.38)$$

where $q = \frac{\lambda}{\lambda+\mu_1}$ and r is the root between 0 and 1 of Equation 4.12 expressed in terms of Equation 4.21.

The average waiting time in the demand queue, $W_d = \frac{N_s}{\lambda_d}$

4.6.3 Results of Second Attempt

We have used values measured from the NMS simulation to investigate the analytical model presented. Simulation results for $p = 1$ and $d = 2$ were obtained for different demand miss rates. The simulation results are then compared with results from analytical model. This is shown in Figure 4.8 and also in Mapp et al. [2009]. In addition, the results are also shown in terms of utilization rather than arrival rates in Figure 4.9.

The two results are quite close in value over a wide operational range. This indicates that the model will be useful in developing practical algorithms for high-performance network-based servers. It should be noted that the model is approximate as it depends on which state of the imaginary Chain/Series is used to calculate r. This is because the solution for r varies slightly depending on which state is used. The best results were obtained using the state $2, 2$ which, in this case, is equal to K, the maximum batch size.

This is shown by referring back to Equation 4.8. If $n = 2$, we get the following Equation:

$$(\lambda + \mu_2)p_{2,2} = \lambda p_{1,1} + \mu_2 p_{4,2} + \mu_1 p_{3,1} \tag{4.39}$$

where $p_{1,1}$comes from the imaginary PBM chain and not from the gate-limited service model.

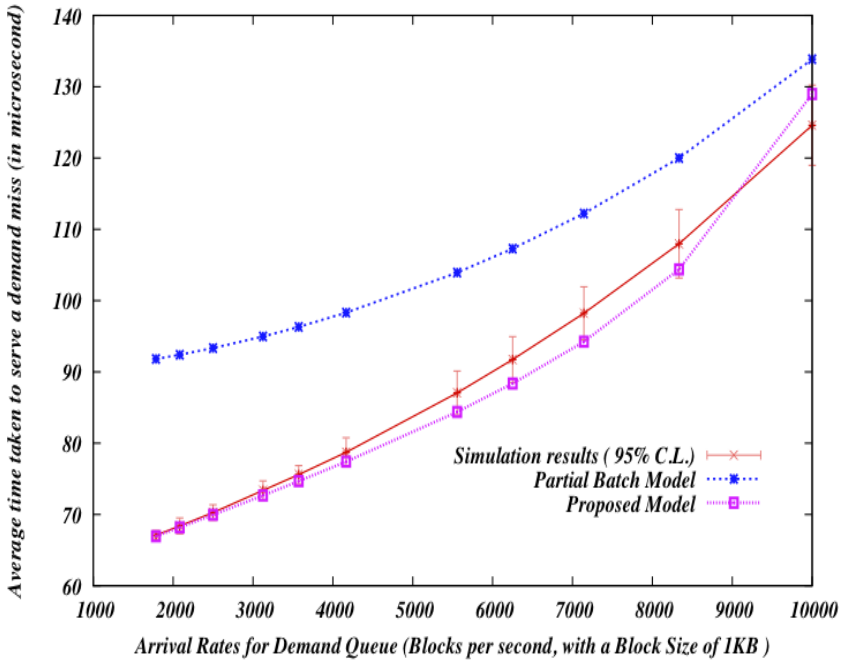

Figure 4.8: Average time to serve a demand miss (T_{D-NMS}) using Simulation, Partial Batch Model and Second Attempt (Our Model).

4.7 Towards a General Solution

In this section, we seek to extend the method used for $K = 2$ to a general value of K. So a gate-limited model, where K is equal to the maximum number of requests that can be served at any moment, can be represented by a gated-limited model of K series or chains. Furthermore, if we represent a given chain by m, we can express the average number of requests in that chain, N_m, in terms of the first element of that chain, $p_{m,m}$. For $m < K$, this sum for that chain is given by:

$$N_m = \sum_{n=m}^{\infty} n(\frac{\lambda}{(\lambda + \mu_m)})^{n-m} p_{m,m} \tag{4.40}$$

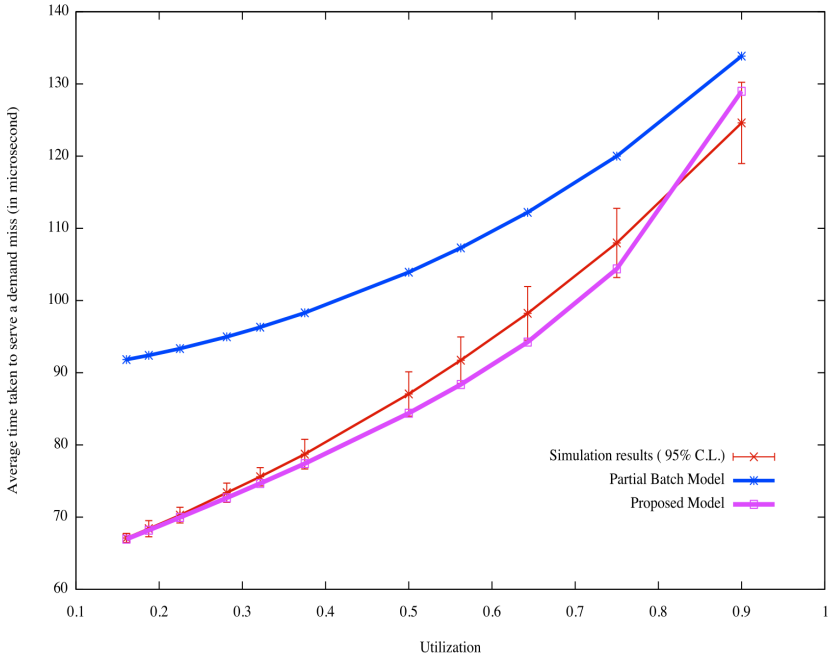

Figure 4.9: Utilization, ρ

Expanding:

$$N_m = \sum_{n=m}^{\infty}(n-(m-1))(\frac{\lambda}{(\lambda+\mu_m)})^{n-m}p_{m,m} + (m-1)\sum_{n=m}^{\infty}(\frac{\lambda}{(\lambda+\mu_m)})^{n-m}p_{m,m} \tag{4.41}$$

Using the same technique as above and by letting $r_m = \frac{\lambda}{\lambda+\mu_m}$, the first term can be expressed as:

$$\sum_{n=m}^{\infty}(n-(m-1))r_m^{n-m}p_{m,m} = \sum_{n=m}^{\infty}\frac{d}{dr}r_m^{n-(m-1)}p_{m,m} \tag{4.42}$$

Rearranging:

$$\sum_{n=m}^{\infty} \frac{d}{dr} r_m^{n-(m-1)} p_{m,m} = \frac{d}{dr} \sum_{n=m-1}^{\infty} r_m^{n-(m-1)} p_{m,m} \tag{4.43}$$

Let p = n-m+1

$$\begin{aligned} &= \frac{d}{dr} \sum_{p=0}^{\infty} r_m^p p_{m,m} \\ &= \frac{d}{dr} (\frac{1}{1-r_m}) p_{m,m} \\ &= \frac{1}{(1-r_m)^2} p_{m,m} \end{aligned} \tag{4.44}$$

The second term:

$$\begin{aligned} &(m-1) \sum_{n=m}^{\infty} (\frac{\lambda}{(\lambda+\mu_m)})^{n-m} p_{m,m} \\ &= (m-1) \sum_{n=m}^{\infty} r_m^{n-m} p_{m,m} \end{aligned} \tag{4.45}$$

Let $q = n - m$;

$$\begin{aligned} &= (m-1) \sum_{q=0}^{\infty} r_m^q p_{m,m} \\ &= (m-1) \frac{1}{1-r_m} p_{m,m} \end{aligned} \tag{4.46}$$

and thus we get the sum:

$$N_m = \frac{m - (m-1) * r_m}{(1-r_m)^2} p_{m,m} \tag{4.47}$$

$$N_s = \sum_{m=1}^{K} N_m = \sum_{m=1}^{K} \frac{m - (m-1) * r_m}{(1-r_m)^2} p_{m,m} \tag{4.48}$$

For $m < K$,

$$r_m = \frac{\lambda}{\lambda + \mu_m} \tag{4.49}$$

For $m = K$, we use the imaginary PBM technique to solve for r_K, which is given in its general form:

$$p_{n,K} = r^{n-K} p_{K,K} \tag{4.50}$$

Furthermore, we can sum the probabilities in each chain, for $m < K$

$$S_m = \sum_{n=m}^{\infty} (\frac{\lambda}{(\lambda + \mu_m)})^{n-m} p_{m,m} \tag{4.51}$$

Let $q = n - m$:

$$S_m = \sum_{q=0}^{\infty} (\frac{\lambda}{(\lambda + \mu_m)})^{q} p_{m,m}$$

$$S_m = \frac{\lambda + \mu_m}{\mu_m} p_{m,m} \tag{4.52}$$

If we let $p_{m,m} = C_{m,m} p_{K,K}$, we can express $p_{K,K}$ as:

$$p_{K,K} = \frac{1}{C_{0,0} + \sum_{m=1}^{m=K-1} \frac{\lambda + \mu_m}{\mu_m} C_{m,m} + \frac{1}{1 - r_k}} \tag{4.53}$$

For a general technique, we need to find the value of $C_{m,m}$ and we can do so using the equations for the states of $p_{m,m}$ in our model. For $K = 2$,

these equations are Equations 4.6, 4.7 and 4.8. This can be done by solving a series of simultaneous equations. This is not further pursued in this thesis because we are primarily interested in getting an algorithm for prefetching and caching based on the constraints highlighted in the previous chapter. Hence, simulation results from the simulation platform can also be used for this purpose. However, this effort shows that it is possible to get fairly accurate waiting time results over a wide operational range based on this analytical model.

4.8 Conclusion

This chapter presented an analytical model which could be used to estimate the average time to serve a demand miss, T_{D-NMS}, for a given demand arrival rate and prefetch rate. Comparison of the results from the analytical model and simulation results showed that the results estimated by the analytical model are within the confidence range of the simulation. Hence, the model can be used at run time to estimate the average time to serve the demand misses for a given scenario. We will now explore the operational space where QoS could be provided for streaming applications and demand misses.

Chapter 5

Exploring the Boundaries

This chapter brings together the results achieved from Chapters 3 and 4 to define the operational space that needs to be explored. Using the analysed equations / constraints, it explores the operational space where the QoS for streaming applications and demand misses could be provided and obtains optimal operational points. Finally, it describes how to put all the optimal operational points into a database in order to develop an autonomic system.

5.1 Putting It All Together

This section brings together the results achieved from Chapter 3 and Chapter 4. In Chapter 3, Section 3.3 and Section 3.4.1 obtained the equations for streaming applications and demand accesses individually, i.e., without considering each other, to provide QoS to each of them. Also, in Chapter 3, we proposed to use the Prefetch-On-Demand strategy as it uses network and memory resources efficiently. Further, we developed a simulation to simulate the Prefetching and Clustering strategy over network, using the Network Memory Server.

In Chapter 4, we derived an equation which could be used to estimate

the average time, (T_{D-NMS}), to serve a request on the demand queue when using the PonD strategy, for a given value of C (constant cost), P (number of prefetched blocks) and D (number of demand blocks).

Now, using the PonD strategy and the equations derived in Chapters 3 and 4, we need to explore the space where the required QoS for streaming applications and demand misses could be provided over the network. As a first step in exploring the space, we need to define the space itself.

5.2 Defining the Operational Space

This space will consist of three variables. The first variable is the number of prefetched blocks being fetched. The second variable is the rate at which demand misses are occurring. The third variable is the average time experienced in fetching both prefetch and demand blocks in one operation. The average time experienced to serve requests in the prefetch and demand queues will depend on the number of D and P blocks being fetched in a single operation. This region could be viewed as a 3-dimensional object. The three axes of the space consist of the prefetch rate (λ_p), represented as the number of blocks p, on the x-axis, the demand miss rate (λ_d), (y-axis) and the average time experienced to serve a demand miss T_{D-NMS}, (z-axis). This is represented in Figure 5.1. Since the rate at which demand arrival occurs cannot be controlled, we need to analyse the other two variables i.e., the number of prefetch blocks that could be fetched and the average time experienced to serve a demand miss for a given demand miss rate. This demand arrival rate should be measured at a given time. Given the demand arrival rate, we should analyse the number of prefetch blocks that could be fetched and the average time experienced to serve a demand miss, keeping the fixed value of D and varying the value of P.

Hence, we can represent the above 3D space in 2D space for each demand arrival rate. In 2D space, the two axes will be the number of prefetch

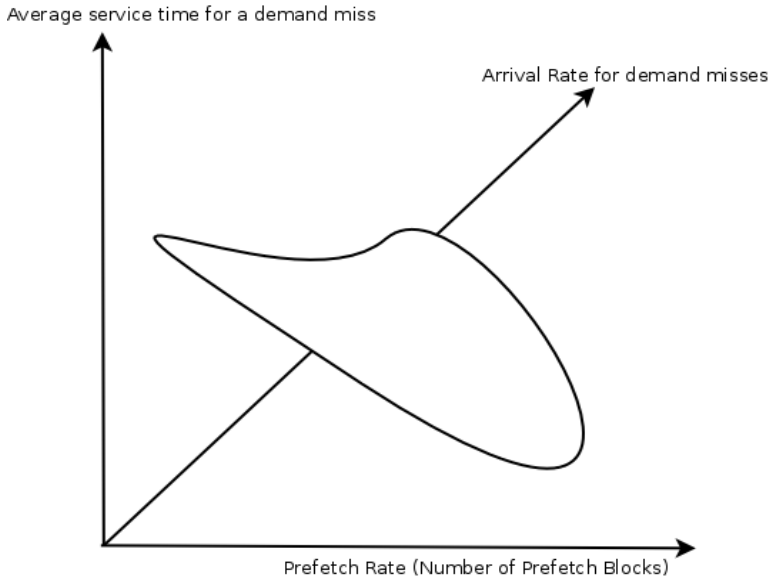

Figure 5.1: Visualisation of the Space in 3D.

blocks P and the average time experienced to serve a demand miss for a given demand arrival rate and for a given D. Further, we then apply the constraints to obtain the *Working Space*. The *Working Space* is the space where QoS for streaming applications and demand misses could be provided for a given arrival rate of demand miss and prefetch rate. For example, it can be seen from Figure 5.2 that for a given demand arrival rate is 0.002857 blocks per microsecond (mean arrival time 350μsec) and having $D = 2$ blocks, we can find the average time experienced to serve a demand miss on the demand queue for different values of P. We can obtain these results either by using simulation or by using an analytical model. Since we have built a simulation platform and the simulation results are readily available for different values of D, we will use the results from the simulation to analyse the average time experienced on a demand miss, for a given value of demand arrival rate and values of P and D.

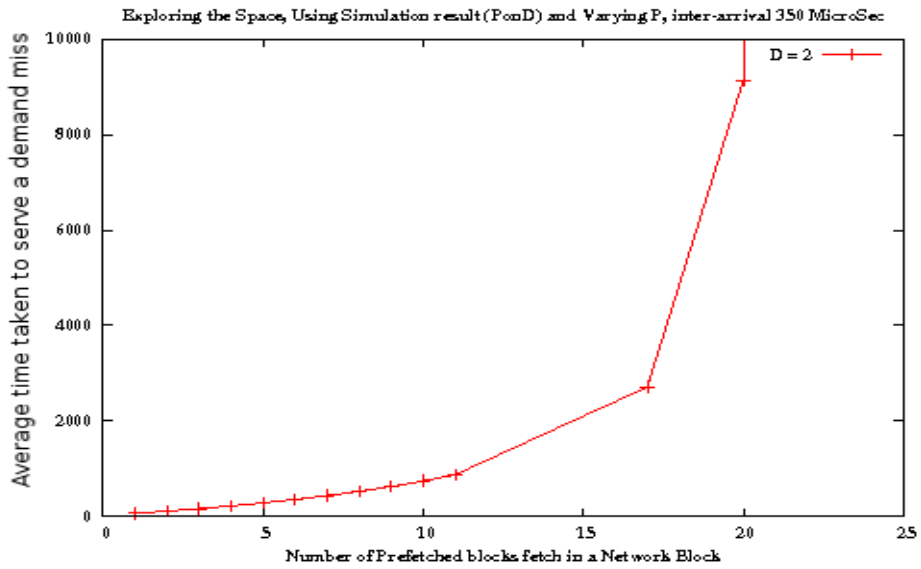

Figure 5.2: Exploring space for demand arrival rate of 0.002857 blocks per μsec (2857 blocks per second, where 1 block is 1024 bytes)

5.3 Fundamental Constraints

Once the space is derived by our simulation results, we will apply constraints on the space so that the *Working / Operational Space* could be derived.

The fundamental constraints consist of the time taken to serve a demand request on the demand queue and the service rate experienced at the prefetch queue.

Demand Requests

For the demand queue, the average time taken to serve a demand miss, (T_{D-NMS}), over the NMS should be less than the average time experienced on a commonly used storage device, for example, a local hard-disk. This is referred to as the *Storage Constraint*.

$$T_{D-NMS} < T\_storage \tag{5.1}$$

where, $T\_storage$ is the average time experienced to serve a request on com-

monly used storage for a demand miss.

Prefetch Requests

For given values of D and P, the average time experienced to serve a demand miss on the demand queue over the NMS can be estimated using simulation. As we are using PonD prefetching techniques, the average time experienced in fetching P prefetch blocks will be the same as the average time experienced in satisfying a demand miss. Therefore, the prefetch rate would be equal to $\frac{P}{T_{D-NMS}}$ blocks per μsec. For the prefetch queue, the service rate experienced at prefetch queue ($\frac{P}{T_{D-NMS}}$) should be greater than the rate at which blocks are consumed ($\frac{1}{T_{cpu}}$) by streaming applications. This is referred to as the *Prefetch Constraint*, as shown in the below equation:

$$\begin{aligned} Prefetchrate &\geqslant Consumerate \\ \frac{P}{T_{D-NMS}} &\geqslant \frac{1}{T_{cpu}} \\ \frac{T_{D-NMS}}{P} &\leqslant T_{cpu} \end{aligned} \tag{5.2}$$

In other words, for the prefetch constraint, the time to fetch P blocks should be less than the time to consume P blocks, or the average time experienced to satisfy a demand miss should be less than or equal to the time to consume P prefetched blocks in one operation i.e.

$$T_{D-NMS} \leqslant T_{cpu} * P \tag{5.3}$$

Once these constraints are applied to the explored space, the *Working Space* can be obtained.

5.4 Applying the Constraint to Explore Space

In this section, we will apply the discussed constraints on the results mapped out by the simulation. The two constraints are: the storage constraint and the prefetch constraint.

- **Storage Constraint**: The Storage Constraint would really depend on the QoS that we would like to provide to demand misses using the NMS compared to other storage devices. The question is what type of storage device should be considered in order to carry out the comparative study. However, the concept of a storage constraint would be valid regardless of the storage device being used to do the comparison with the NMS.

  There are various types of storage devices used to store data over the network or on the local disk. Some of the available techniques to store data over the network are Network File System (NFS), Storage Area Network (SAN), etc. However, the results from previous research by Rochberg and Gibson [1997], showed that the performance of NFS is slower than the local disk and for a SAN to be implemented, it needs special hardware and therefore cannot be readily implemented. Hence, it is not commonplace.

  The most commonly used storage device is the local hard-disk. There are different types of hard-disks available, for example, IDE, SATA, etc., each having a different cache size on it. In our research, for a comparative study for demand misses, we have used a SATA hard drive, as it is much faster than IDE. The SATA hard drive has 32MB cache size and 7200rpm and it is now the de-facto industrial standard. Hence, the storage constraint will be referred to as the *Disk Constraint*, (T_{disk}). The average time experienced to serve a demand miss for those hard-disks is 7.8 milliseconds, i.e., T_{disk} is 7.8 milliseconds. Although, the popularity of Solid State Drives, (SSD) is increasing due to its high read/write speeds and fault tolerance, they are very expensive and still not commonly used.

Applying the first constraint, i.e., the disk constraint in Figure 5.2, we obtain Figure 5.3, where, T_{disk}, is the average time experienced to

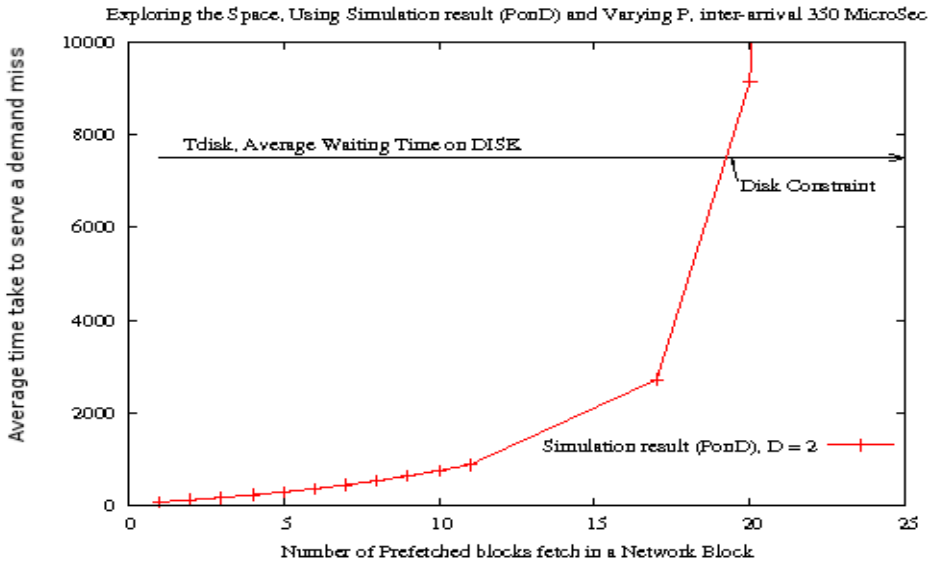

Figure 5.3: Applying the Disk Constraint on the explored space for demand arrival rate of 0.002857 blocks per μsec (2857 blocks per second, where 1 block is 1024 bytes), having $D = 2$ blocks and T_{disk} is 7.8 milliseconds.

serve a demand miss on the SATA hard-drive and *PonD*, is the time experienced to serve a demand miss (T_{D-NMS}) for a given demand arrival rate of 0.002857 blocks per μsec, having $D = 2$ blocks and prefetching different number of P prefetch blocks.

From Figure 5.3 the maximum number of prefetch blocks P that can be fetched at demand arrival rate of 0.002857 blocks per μsec and having $D = 2$ blocks, is equal to 17 prefetch blocks. Fetching less than or equal to 17 prefetch blocks for a given scenario, will be able to provide the required QoS for the demand misses using the NMS, as shown by Equation 5.1. Fetching more than 17 prefetch blocks will increase the time experienced to serve a demand miss on the demand queue and will not be able to satisfy the QoS for demand misses.

- **Prefetch Constraint**: Similar to the disk constraint, the prefetch constraint also depends on the type of video being executed, as differ-

ent types of video will consume blocks at different rates and hence will require different prefetching rates. For example, the time to consume a block in HD video is $200\mu sec$ (5000 blocks per second, where 1 block is 1024 bytes) and the time to consume a block of MPEG-4 video is $400\mu sec$ (2500 blocks per second, where 1 block is 1024 bytes). Similarly, the consumption rates for MPEG-2 and MPEG-3 are different.

However, MPEG-4 video is the most commonly used type of video. Therefore, in our example, we assume that the running streaming application is using MPEG-4 video, and hence the time to consume a block is 400μsec. Also, the HD videos are becoming quite popular, but to provide QoS to HD video, the network infrastructure needs to be improved in terms of speed and so a large amount of storage space will be required.

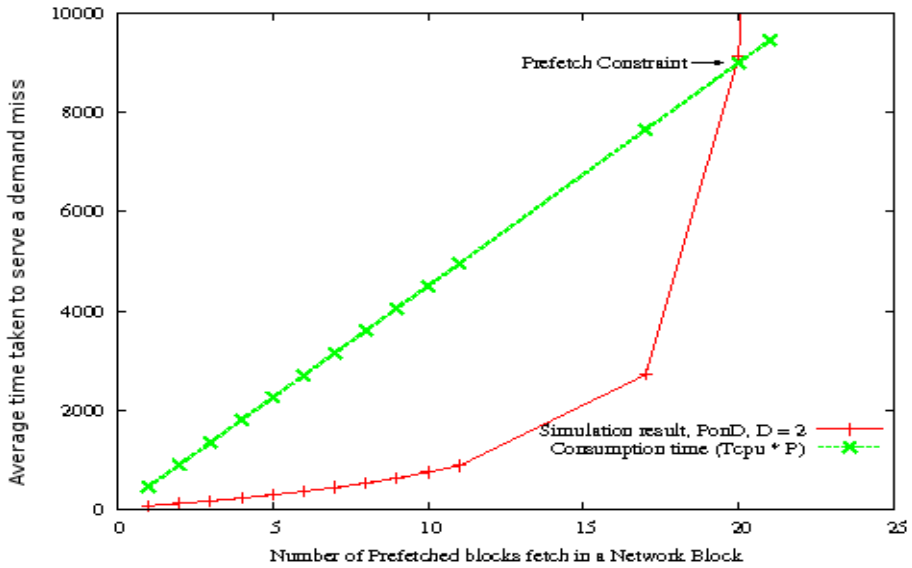

Figure 5.4: Applying the Prefetch Constraint on the explored space for demand arrival rate of 0.002857 blocks per μsec (2857 blocks per second, where 1 block is 1024 bytes), having $D = 2$ blocks and $T_{cpu} = 400\mu$sec.

Applying the second constraint i.e. the prefetch constraint, in Figure 5.2, we obtain Figure 5.4, where, the *consumption time* is the time taken by the streaming applications to consume P blocks and the time

taken to fetch P blocks is given by the PonD simulation results.

Figure 5.4 shows that when the number of prefetch blocks being fetched is less than 19 blocks then the time to fetch those P prefetch blocks, for a given scenario, is much less than the time taken to consume those blocks by a given streaming application and hence, the required QoS for streaming applications could be provided, as in Equation 5.3.

However, once the number of prefetch blocks being fetched is more than 19 blocks, the time to fetch P ($P > 19$) blocks would be greater than the time to consume P blocks and hence, the streaming application will experience jitter or will stall and therefore will not be able to obtain the required QoS.

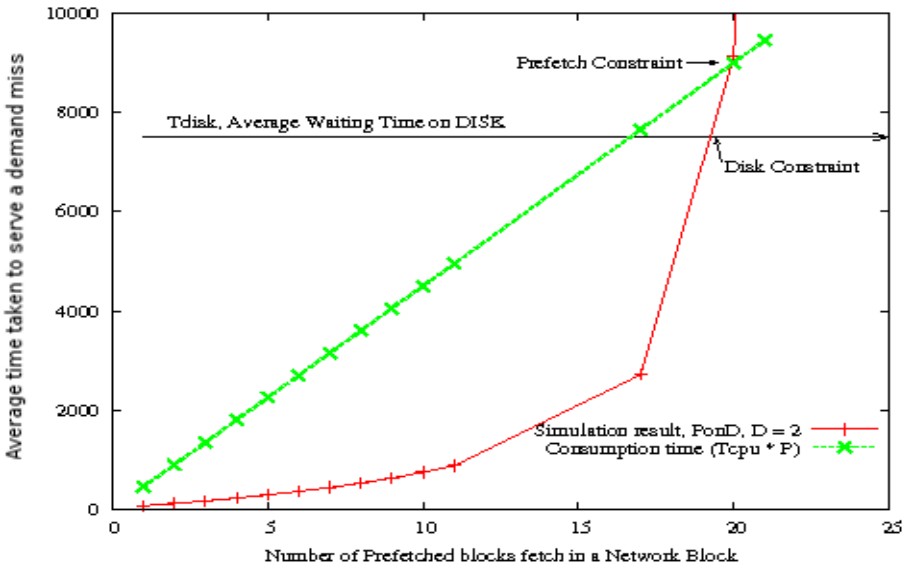

Figure 5.5: Applying the Disk and Prefetch Constraints on the explored space for demand arrival rate of 0.002857 blocks per μsec (2857 blocks per second, where 1 block is 1024 bytes) and having $D = 2$ blocks where $T_{disk} = 7.8$ milliseconds and $T_{cpu} = 400\mu$sec.

- **Applying both the constraints**: Applying both constraints from Figure 5.2, we obtain Figure 5.5. From Figure 5.5, it can be seen that the prefetch constraint intersects with the average service time, when

$P = 19$ blocks and the disk constraint intersect with the average service time, when $P = 17$ blocks. However, fetching more than 17 prefetch blocks will not guarantee the QoS for demand misses. Therefore, our maximum number of prefetch blocks, P_{max}, that could be fetched in a single operation for this scenario would be 17 prefetch blocks, which would satisfy both the constraints and is referred to as the working space. In this example the disk constraint dominates the prefetch constraint.

In contrast, in Figure 5.6, it can be seen that the prefetch constraint dominates the disk constraint, where the demand arrival rate is 0.004 μsec (4000 blocks per second, where 1 block is 1024 bytes) and $D = 2$ blocks. In this case, the number of prefetch blocks should be less than or equal to 6 blocks in an operation and is referred to as the explored working space.

This section explores the space consisting of different values of P (prefetch blocks) with demand misses for a given scenario. However, there is still a need to analyse the optimal operational points i.e., optimal value of P_{opt} for a given value of demand arrival rate and the given value of D.

5.5 Optimal Operational Points

This section first analyses the optimal number of prefetch blocks, P_{opt}, that should be fetched for a given value of demand arrival rate and the value of D. Secondly, it analyses the optimal consumption time, T_{oct}, for the streaming applications that could be supported for given values of D, P and demand arrival rate.

1. **Optimal number of prefetch blocks**: In the previous section, we explored the working space for a given demand arrival rate and for a given value of D. The space consists of different values of P and is

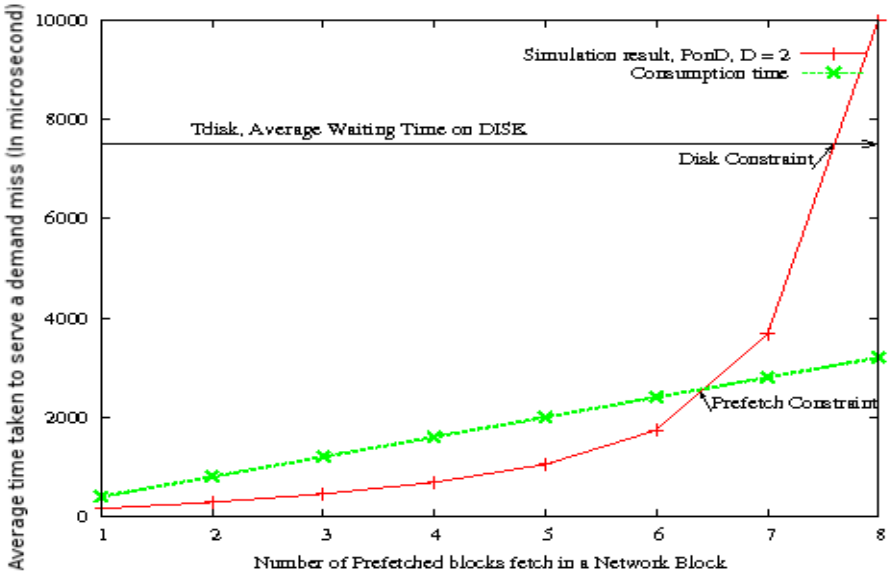

Figure 5.6: Applying the Disk and Prefetch Constraints on the explored space for demand arrival rate of 0.004 blocks per μsec (4000 blocks per second, where 1 block is 1024 bytes), having $D = 2$ blocks, $T_{disk} = 7.8$ milliseconds and $T_{cpu} = 400\mu$sec.

bounded by the maximum number of P, i.e., P_{max}. This means that the QoS could be provided for a system for P less than or equal P_{max}. However, in our work, we would always like to operate at P_{max} for a given D, because we are using the PonD strategy where prefetching is only done when demand misses occur. Hence, the system should attempt to prefetch the maximum number of prefetch blocks when there is a demand miss, as we would not know when the next demand miss will occur. Also, fetching the maximum number of prefetch blocks will allow network bandwidth to be used effectively. Hence, for a given demand arrival rate and value of D, the optimal point for prefetch blocks (P_{opt}) is always the maximum number of prefetch blocks P_{max} that can satisfy both constraints, as shown in Figure 5.7.

If P_{max} is set by the dominant disk constraint, as in Figure 5.7, then it will always buffer some prefetch blocks, as the time to consume blocks is more than the time to fetch those blocks. As the number of fetch cycle

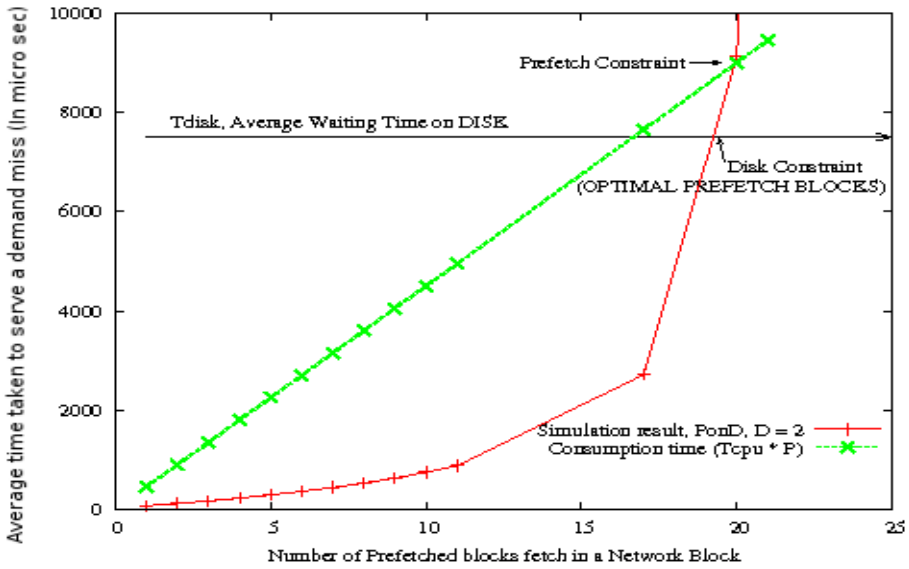

Figure 5.7: Showing the optimal prefetch number of blocks for demand arrival rate of 0.002857 blocks per μsec (2857 blocks per second, where 1 block is 1024 bytes) at $D = 2$ blocks, where $T_{disk} = 7.8$ milliseconds and $T_{cpu} = 400\mu$sec.

increases, the number of buffered blocks will increase, hence increasing the buffering.

If P_{max} is set by the dominant prefetch constraint, as in Figure 5.6, then the time to consume blocks will be equal to the time to fetch blocks and hence, the system will operate like a JIT prefetching strategy. However, this leads to very small values of P and hence the network bandwidth will not be used effectively for prefetching. The only way to increase the value of P_{max} is to increase the value of D blocks being fetched.

In order to use the maximum strength of prefetching on a demand miss and hence, using the network bandwidth and buffer effectively, we would ideally like to operate at a point where the average time to serve a demand miss over the NMS, (T_{D-NMS}), is equal to the average time experienced on commonly used storage, (T_{disk}). In mathematical form, the optimal number of prefetch blocks, P_{opt}, for a given demand arrival rate and given value of D, should be set when $T_{D-NMS} = T_{disk}$.

For example, in the Figure 5.6, we would ideally like to use $P_{opt} = 7$

blocks for the given value of demand arrival rate (0.004 μsec) and value of $D = 2$ blocks. However, using $P = P_{opt} = 7$ blocks, would not be able to provide the QoS for streaming applications, because the time to fetch 7 prefetch blocks is greater than the time to consume those 7 prefetched blocks.

2. **Optimal Consume Time**: The optimal consume time for a given value of D is the point, i.e. the number of prefetch blocks, where the two constraints are satisfied such that $T_{D-NMS} = T_{disk}$ and also $P * T_{cpu} = T_{disk}$. At this point P number of blocks should be equal to P_{max} blocks i.e. $P = P_{max} = P_{opt}$, and T_{cpu} becomes the ideal consume time for the streaming applications, T_{oct}, which also means $P_{opt} * T_{oct} = T_{disk}$ and therefore, the optimal consume time for given demand arrival rate, D and P_{opt} can be calculated by the following equation:

$$T_{oct} = \frac{T_{disk}}{P_{opt}} \tag{5.4}$$

From Figure 5.8 that the optimal consumption time (T_{oct}) of the streaming applications for a given scenario ($D = 2$ blocks, $P_{opt} = 7$ blocks and demand arrival rate (0.002857 μsec) should be equal to 350μsec and not 400μsec. Similarly, in the Figure 5.9, the optimal consumption time per block (T_{oct}) of the streaming application should be equal to 980μsec and not 400μsec.

However, the rate at which the prefetch blocks are consumed depends on the applications being executed and it cannot be controlled once the applications have been started. Therefore, we would like to analyse the optimal number of prefetch blocks (P_{opt}) and optimal consume time (T_{oct}) that could be supported at different values of D for a given demand arrival rate. Given this data, we can always set the value of $P = P_{opt}$, where the rate at which blocks are consumed by streaming applications is close to the rate at which they can be fetched from the storage device (NMS) i.e., $T_{cpu} \approxeq T_{oct}$. Also, using T_{oct} for the streaming applications can also minimise the buffer requirements, as it

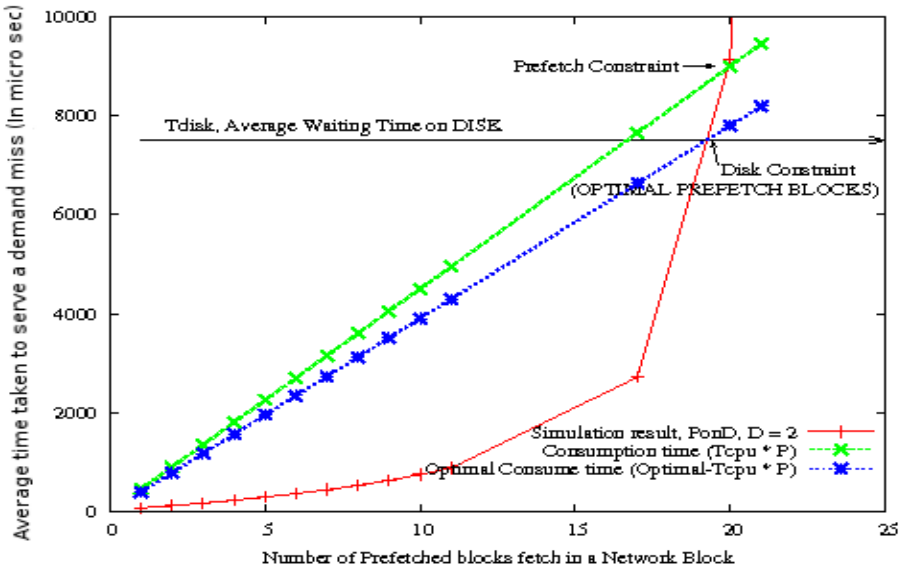

Figure 5.8: Shows the optimal prefetch number of blocks and optimal consume rate of streaming applications for given demand arrival rate of 0.002857 blocks per μsec (2857 blocks per second, where 1 block is 1024 bytes) at $D = 2$ blocks, where $T_{disk} = 7.8$ milliseconds, $T_{cpu} = 400\mu$sec and optimal-Tcpu $= 350\mu$sec.

operates in a similar manner to the JIT prefetching strategy.

It can be seen from Figures 5.6 and 5.10, that having only data for $D = 2$ blocks, the system would only prefetch 6 prefetch blocks in one operation. Having the results of the average time taken to serve a demand miss, for D equal to 1 to 4 blocks, as in Figure 5.10, we can set the value of D, where the rate at which blocks are prefetched is approximately equal to the rate at which the blocks are consumed by the streaming applications and $P = P_{opt}$. In this case $D = 3$ blocks, $P = P_{opt} = 19$ blocks and $T_{cpu} \approxeq T_{oct}$, hence prefetching at maximum rate and using bandwidth and buffers effectively.

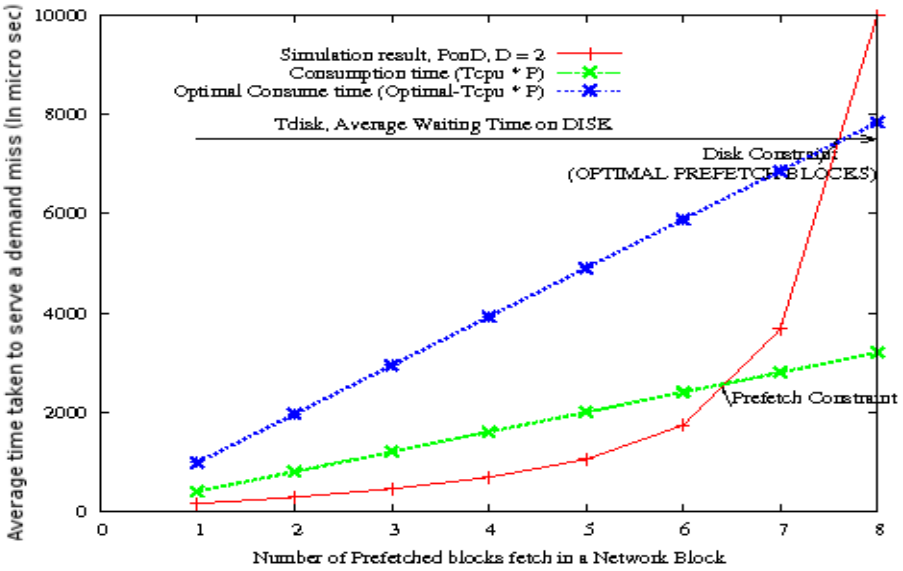

Figure 5.9: Shows the optimal prefetch number of blocks and optimal consume rate of streaming applications for given demand arrival rate of 0.004 blocks per μsec (4000 blocks per second, where 1 block is 1024 bytes) at $D = 2$ blocks, where $T_{disk} = 7.8$ milliseconds, $T_{cpu} = 400\mu$sec and optimal-Tcpu $= 980\mu$sec.

5.6 More Detailed Results

This section determines the optimal number of prefetch blocks, P_{opt} and T_{oct}, the optimal consume time that could be supported, for the various demand arrival rates and for different values of D. Having seen the preliminary results, we believe that the values of D will vary from 1 to 7 blocks [1] and values of P will vary from 1 to 40 blocks. The arrival rates of the demand misses was set, in decreasing order, to 0.0125 (12500 blocks per second (blkps)), 0.01 (10000 blkps), 0.0066 (6600 blkps), 0.005 (5000 blkps), 0.004 (4000 blkps), 0.0033 (3300 blkps), 0.00285 (2850 blkps), 0.0025 (2500 blkps), 0.00222 (2220 blkps) blocks per microsecond.

The following are the results for demand arrival rates 0.01 and 0.0022

[1]For the graphs to look clear, we have only represented the values of D from 1 to 5 blocks in the graph.

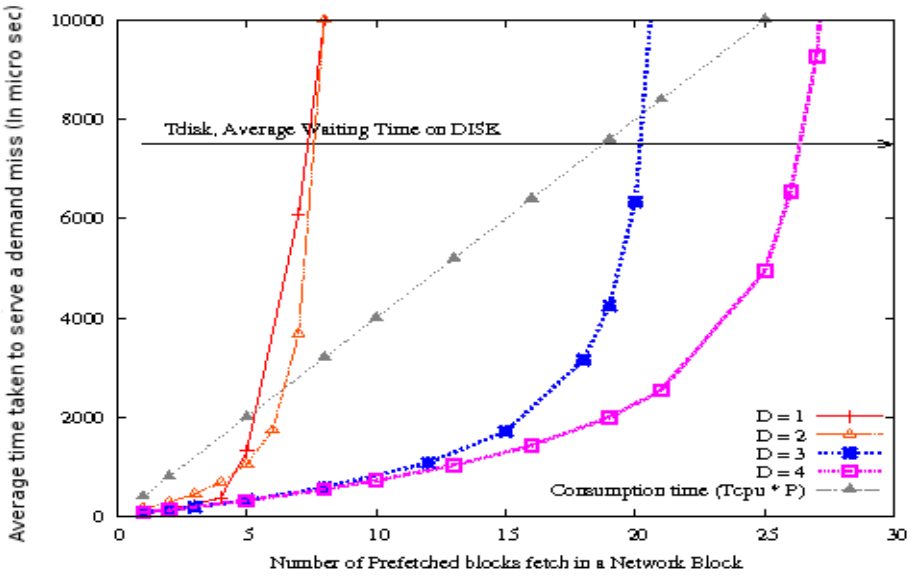

Figure 5.10: 2D space: demand arrival rate of 0.004 blocks per μsec (4000 blocks per second, where 1 block is 1024 bytes).

blocks per microsecond with the disk constraint applied on it. It shows the P_{opt} and T_{oct} that should be used for the given demand arrival rate and the value of D. Here, we present the observations made from these results:

- It can be seen from Figures 5.11 and 5.12 that as the number of D demand blocks fetched in an operation increases for a given demand arrival rate, the number of prefetch blocks (P_{opt}) that could be fetched increases. For example, in Figure 5.11, as the number of D increases from the 1 to 7 blocks, the value of P_{opt} increases from 2 to 15 blocks. Similar effects can be seen in Figure 5.12. Table 5.1 and Table 5.2 show the optimal consume time (T_{oct}) that can be supported for the each value of D and P_{opt}, for a given demand arrival rate.

- Being able to increase or decrease the number of prefetch blocks i.e. increase or decrease the prefetch rate, that could be fetched in an operation for a given value of demand arrival rate, allows us to set the value of D such that the prefetching rate is approximately equal to the rate at which blocks are consumed by streaming applications, hence,

The Optimal value of P_{opt} for a given value of demand arrival rate and D			
λ_d	No. D blocks	$P_{opt}blocks$	$T_{oct}\mu$sec
0.01 blocks per μsec (10000 blocks per second)	1	2	2700
	2	4	1600
	3	6	1140
	4	8	860
	5	11	690
	6	13	600
	7	15	520

Table 5.1: This table shows the values of P_{opt} and T_{oct} for different values of D, for a given demand arrival rate of 0.01 blocks per microsecond.

The Optimal value of P_{opt} for a given value of demand arrival rate and D			
λ_d	No. D blocks	$P_{opt}blocks$	$T_{oct}\mu$sec
0.0022 blocks per μsec (2200 blocks per second)	1	13	550
	2	25	300
	3	35	210
	4	47	159
	5	56	135
	6	64	121
	7	71	110

Table 5.2: This table shows the values of P_{opt} and T_{oct} for different values of D, for a given demand arrival rate of 0.0022 blocks per microsecond, (2200 blocks per second).

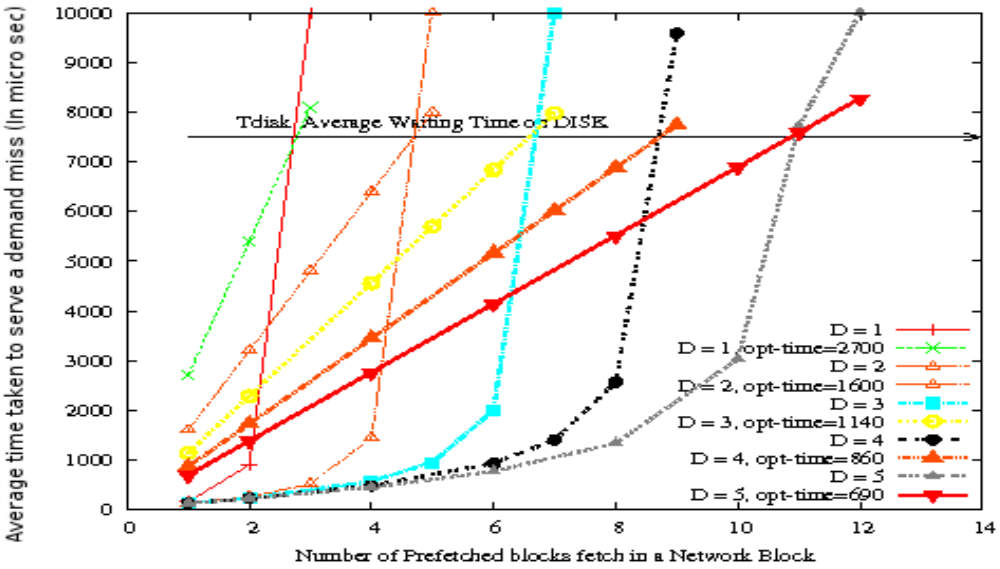

Figure 5.11: Shows optimal prefetch rate that should be for a demand rate of 0.01 blocks per microsecond (10000 blocks per second) and given value of D.

using network bandwidth and buffering effectively.

For example, for the demand arrival rate equal to 0.0022 blocks per microsecond, if the streaming applications consume prefetched blocks at the rate approximately equal to $\frac{1}{550}(\frac{1}{T_{oct}})$ blocks per μsec, then the value of D should be set to 1 block and the value of P equals to 14 blocks, as shown in Figure 5.12. Similarly, if the rate at which blocks are consumed by the streaming applications is approximately equal to $\frac{1}{159}(\frac{1}{T_{oct}})$ blocks per μsec, then the value of D and P_{opt} will be equal to 4 and 48 blocks respectively. These shows that the prefetch rate could be adjusted based on the value of D.

- These experimental results also verified Equation 5.4, i.e. in Figure 5.12, for the given value of demand arrival rate of 0.0022 blocks per μsec and value of $D = 4$ blocks, $P_{opt} = 48$ blocks and $T_{disk} = 7800\mu sec$. Now from Equation 5.4, we have T_{oct} is equal to $\frac{T_{disk}}{P_{opt}}$ and therefore, T_{oct} will be 162.5μsec. The results from the experiment showed that the value $T_{oct} = 159\mu$sec. This indicates that having the value of P_{opt}

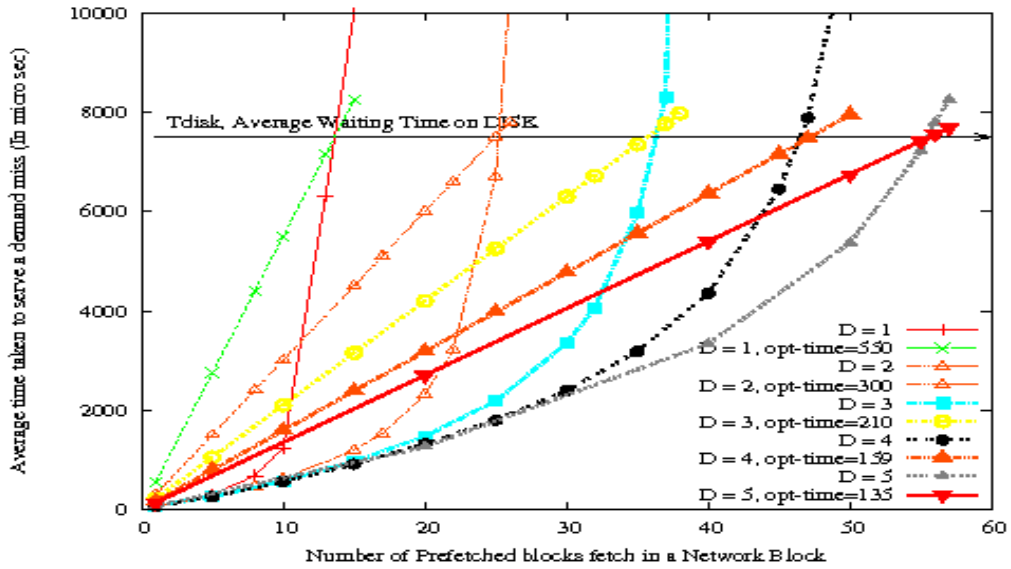

Figure 5.12: Shows optimal prefetch rate that should be for a demand rate of 0.0022 blocks per microsecond (2200 blocks per second) and given value of D.

for the each value of D, for a given demand arrival rate, the value of T_{oct} can be calculated on "the fly".

- It can be also seen from Figures 5.11 and 5.12, as the arrival rate of the demand miss decreases, the amount of buffering required increases i.e., number of prefetch P_{opt} blocks that need to be fetched increases. This is because when using the PonD strategy, prefetch is only done on a demand miss, as demand miss rates decrease the amount of time needed to serve streaming applications without a fetch operation increases, hence more blocks must be fetched in each operation.

5.7 Using the Explored Space

This section describes how to use the analysed data, (P_{opt} and T_{oct}), to develop an autonomous system which dynamically adjusts to changing values

of the demand miss rate. It first proposes to incorporate the data into a database such that it can be retrieved when required.

In order to do this, the database should be designed such that for a given value of demand arrival rate, it should be possible to obtain the optimal consumption time (T_{oct}) that could be supported for each value of D, where D is equal 1 to 7 blocks. Depending on the running streaming applications, the consumption rate of these applications should be known. By comparing the consumption rate of those running streaming applications with the consumption rate that can be supported for a given demand arrival rate, the optimal value of D and P_{opt} could be found such that the QoS for the streaming applications and demand misses could be provided.

5.8 Conclusion

This chapter defined the operational space and represented it as a 3-D figure. It showed how the 3-D space could be explored as 2-D spaces for discrete values of the demand arrival rate. It showed how the working space could be obtained by applying storage and prefetch constraints. However, the values of the constraints depend on the storage device and streaming applications. It analysed the optimal prefetch point (P_{opt}) and optimal consumption time (T_{oct}), for the streaming applications for a given demand arrival rate and given values of D so that the QoS for streaming applications and demand misses could be satisfied. Furthermore, it explored the optimal operational points for different values of demand arrival rates and for values of D from to 1 to 7 blocks. Finally, it proposed to develop an autonomous algorithm to use the analysed data, (P_{oct} and T_{oct}), in a real system.

Chapter 6

Database and Implementation Design

In the previous chapter, we analysed the optimal operational points, in terms of P_{opt} and T_{oct}, for given demand arrival rates and for different values of D. The next challenge is to use these operational points in a real system in order to provide the required QoS. In order to do this, we propose to store the derived operational points in a database such that data can be retrieved efficiently. Furthermore, the database will be used in the simulation to develop and to verify the working of the developed algorithm. Once the algorithm is developed and verified, the design and implementation of the proposed work, using the Network Memory Server (NMS) and the Experimental File System (EFS) are presented and used to show that the algorithm can be implemented on normal computer systems.

6.1 Database Design

In this section, we propose the design of a database such that the data can be retrieved efficiently and could be used in a real system. The design of the database could be implemented using any database system such as MY-SQL,

MS-SQL, MS-ACCESS etc, or as a hash table.

As an initial step in designing the database, we need to analyse the number of fields that will be involved and which of them need to be stored. Once all the fields are analysed then they can be stored in tables such that the database structure is suitable for general-purpose querying and free of certain undesirable characteristics-insertion, update, and deletion anomalies - that could lead to a loss of data integrity. This is prevented by using normalization techniques in which each record of the data is uniquely identifiable using the primay key, as defined by E.F.Codd [Codd, June 1970], [Codd, August 31st, 1971], [Codd, April 23rd, 1974].

In this research, the fields that need to be stored are the demand arrival rates, (λ_d), and the values of the prefetch rates, (λ_p), for a given value of demand arrival rate, (λ_d), and for each value of D, where D ranges from 1 to 7. Since the results explored in the previous chapter are in units of time, (microseconds and milliseconds), rather than rates, the database will also have the values in units of time, i.e., the values of mean arrival time of demand misses ($\frac{1}{\lambda_d}$) and the values of optimal consume time per block (T_{oct}) for the streaming applications that could be supported.

The optimal consume time, (T_{oct}), for the streaming applications that could be supported for the given mean arrival time depends on the number of prefetch blocks, (P_{opt}), being fetched in a network operation. Further, the number of prefetch blocks, (P_{opt}), that could be fetched in a network operation depends on the number of demand misses being satisfied in a network operation for a given demand arrival rate. Hence, the fields that we are interested in storing are the mean arrival time for the demand misses, ($\frac{1}{\lambda_d}$), T_{oct} and values of P_{opt} and D. However, once the value of P_{opt} is stored, for the corresponding value of $\frac{1}{\lambda_d}$ and D then the value of T_{oct} can be calculated by using equation 5.4. Hence, the value of T_{oct} will not be stored in the database.

Once the fields that need to be stored are analysed, they need to be

stored in tables such that the design of the tables adheres to the rules of normalisation. We would like to store the value of P_{opt} for each value of a demand arrival rate and D. In order to satisfy these requirements and to make sure that the database is normalised, we have designed two tables: *ArrivalRates* and *OptimalPs*.

The fields for the ArrivalRates table are the *A_id* and the *arrivalrate*, where *A_id* is the primary key for the table. In the ArrivalRates table, we have all the arrival rates listed for which we have analysed the corresponding values of P_{opt} and D. In the OptimalPs table, we have the values of P_{opt}, D and *A_id*. The value of D and *A_id* together correspond to the primary key of the OptimalPs table, and *A_id* is the foreign key for the table OPtimalPs. The structure of the tables with the fields and the data types are shown in Figure 6.1. It can be seen from Figure 6.1 that all the fields could be stored

```
mysql>
mysql> explain ArrivalRates;
+-------------+---------+------+-----+---------+-------+
| Field       | Type    | Null | Key | Default | Extra |
+-------------+---------+------+-----+---------+-------+
| A_id        | int(11) | NO   | PRI | NULL    |       |
| arrivalrate | int(11) | NO   |     | NULL    |       |
+-------------+---------+------+-----+---------+-------+
2 rows in set (0.00 sec)

mysql> explain OptimalPs;
+-------+---------+------+-----+---------+-------+
| Field | Type    | Null | Key | Default | Extra |
+-------+---------+------+-----+---------+-------+
| Pno   | int(11) | NO   |     | NULL    |       |
| Dno   | int(11) | NO   | PRI | NULL    |       |
| A_id  | int(11) | NO   | PRI | NULL    |       |
+-------+---------+------+-----+---------+-------+
3 rows in set (0.00 sec)
```

Figure 6.1: Diagram of Tables ArrivalRates and OptimalPs.

in the OptimalPs table, (by replacing the field *A_id* in the table OptimalPs by *Arrivalrate*), then the question is why does the database design have a separate ArrivalRates table? The reason is that in future we might want to store more information like the network bandwidth, load on the client system

and load on the server, available memory, etc., along with the demand arrival rates, to select the value of P_{opt} and D. Hence, the database is designed such that the ArrivalRates table can describe environments with the demand arrival rate value, as shown in Figure 6.2.

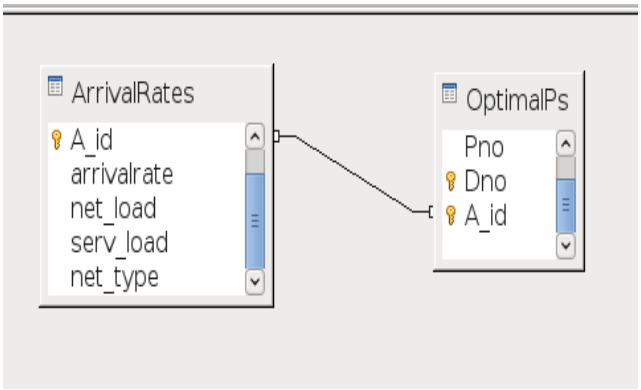

Figure 6.2: Further fields that may need to be stored.

6.2 Algorithm

This section proposes an algorithm, to fetch the appropriate values of P_{opt} and D from the database designed in the previous section, for a given value of mean arrival time of demand misses and the streaming application per block consume time.

The code represented in Listing 6.1, adjusts the measured mean demand arrival time to the boundaries of multiples of 25μsecs. This is because we have only obtained the values of P_{opt} and D for the mean arrival time where mean arrival time is a multiple of 25μsec, as shown in APPENDIX A.

For example, for the measured mean arrival time of 229μsec (4366 blocks per second), the code in the Listing 6.1 will adjust the mean arrival time to 225μsec (4444 blocks per second). Because this is the lower value of the mean arrival rate that is closest to 229μsec and hence will satisfy the

stability of the demand queue.

Listing 6.1: adjusting arrival rate to boundaries of multiple of 25

```
int Quo = Darrivaltime / 25;
CalArrivaltime = (Quo * 25);
```

Once the mean arrival time of demand misses is a multiple of 25μsec then the data from the database can be fetched such that all the values of P_{opt} and D are known for each value of D and for a given value of demand arrival rate. For example, for an adjusted mean arrival time of demand miss as 225μsec, the values of P_{opt} are fetched for D is equal to 1 to 7 blocks, as shown in the Figure 6.3. Knowing the values of P_{opt} and D for a given mean arrival time of

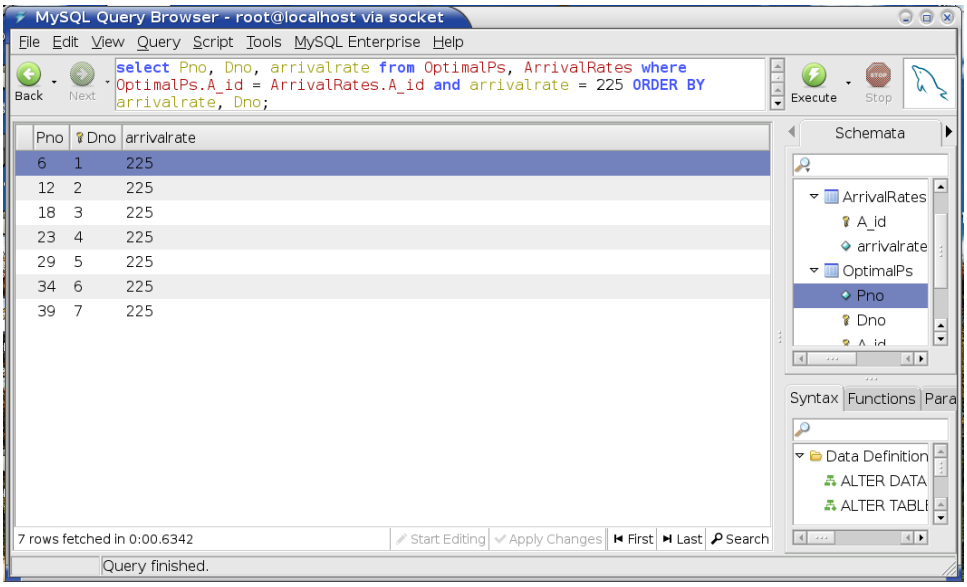

Figure 6.3: Diagram of Tables ArrivalRates and OPtimalPs

demand misses, we can then pick the values of D and P_{opt} such that $\frac{Tdisk}{P_{opt}}$ is less than or equal to the per block consume time of streaming applications, in ascending order of D. As the value of D increases the higher consume rate for the streaming applications could be supported i.e. more prefetch blocks are fetched in a read operation. We would like to prefetch close to the required consumption rate and hence we will start with the value of P_{opt} where $D = 1$

and then increment as required. The algorithm below represents the same logic.

Algorithm 1 Picking up the optimal value of D and P_{opt}

```
{/* The row object is an array, having all fields (P_opt and D) for a given
demand arrival rate */}
while row ≠ NULL do
  SET Pmax = row[0]
  SET Dmax = row[1]
  if Tdisk/Pmax ≤ Consumetime then
    SET P = Pmax
    SET D = Dmax
    BREAK
  end if
  row++
end while
```

6.3 Results

The proposed database and the algorithm were integrated into the simulation, to verify if the required QoS could be provided by the proposed work.

6.3.1 Streaming Applications

In the experimental set-up, the demand arrival rate was regularly changed and the value of the streaming applications' per block consume time was kept constant. Given this, the value of P_{opt} and D were dynamically picked by the simulation from the database based on the streaming applications' per block consume time and the measured demand mean arrival time. Below we have presented the results from two such experiments. In this set-up, the required prefetch rate of the streaming application to execute without jitter was set to 0.004 blocks per microsecond (where 1 block is 1024 bytes,

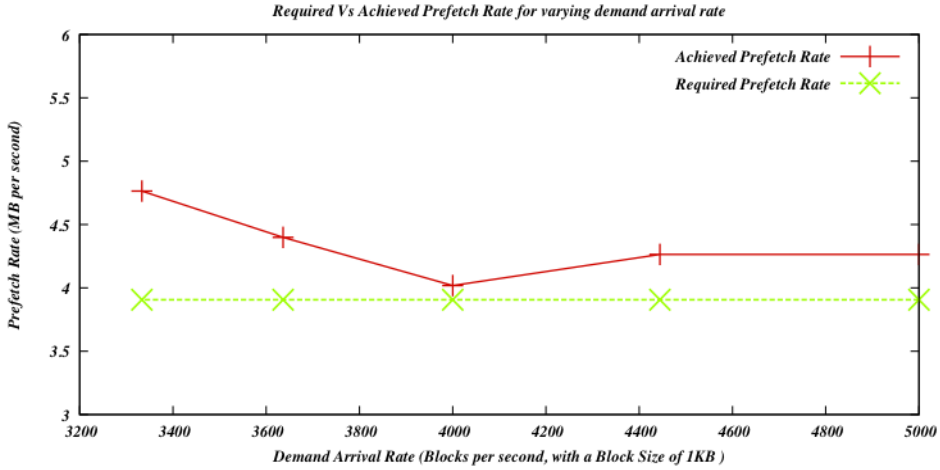

Figure 6.4: Required Prefetch Rate (4 MB per second) vs Achieved Prefetch Rate having different demand arrival rate.

4 MB per second). It can be seen from Figure 6.4 that the prefetch rate experienced by the system was just above 0.004 blocks per microsecond for different mean arrival time of demand misses. This means that the requested prefetch rate and the achieved prefetch rate were very close, hence, using the memory and the network bandwidth effectively, (as it will only prefetch on a demand miss). Similar results are shown in Figure 6.5, for the streaming applications which need the prefetch rate of 0.006 blocks per microsecond (where 1 block is 1024 bytes, 6 MB per second) to run without jitter.

6.3.2 Demand Misses

In the experimental set-up, the prefetching rate was varied, for a given demand mean arrival time. As above, the simulation dynamically picked the values of D and P_{opt} for each scenario. The results show that the average time taken to serve a demand miss while varying the prefetch rate is always less than T_{disk}. Below we have presented the results from two such experiments.

In this set-up, the demand arrival rate was set to 0.0025 blocks per

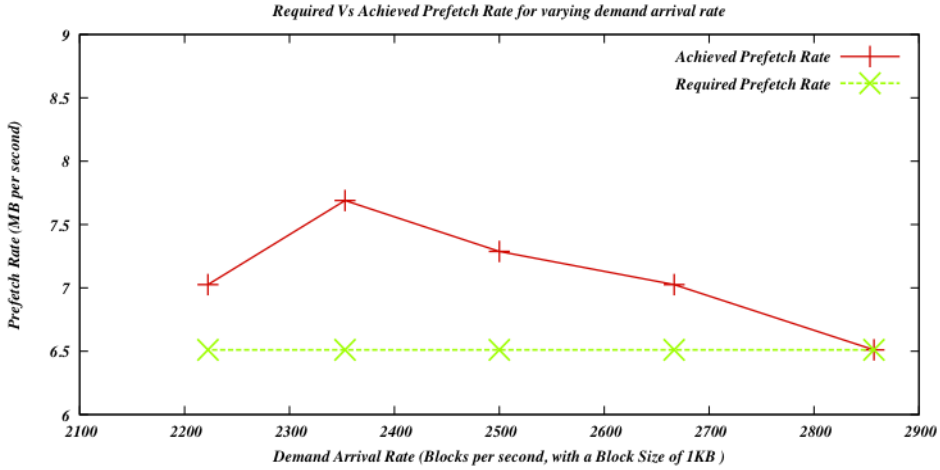

Figure 6.5: Required Prefetch Rate (6 MB per second) vs Achieved Prefetch Rate having different demand arrival rate.

microsecond (2500 blocks per second) and the results are shown in Figure 6.6. The demand arrival rate was reset to 0.004 blocks per microsecond (4000 blocks per second) and the results are shown in Figure 6.7. The results show that the QoS for streaming applications and demand misses could be provided by using the proposed design. In the next section, we will look at implementing the proposed algorithm in a real system, to provide the same level of QoS.

6.4 Implementation Design

The previous section showed promising results using the proposed design in the simulation. We would like to take this further and explore the working of the proposed design in a real system. In order to implement the proposed prefetching and clustering strategies on network-based storage, a network-based storage system and a file system are used. A network-based storage system such as the NMS will allow us to implement clustering over the network. The file system will be required in order to implement prefetching

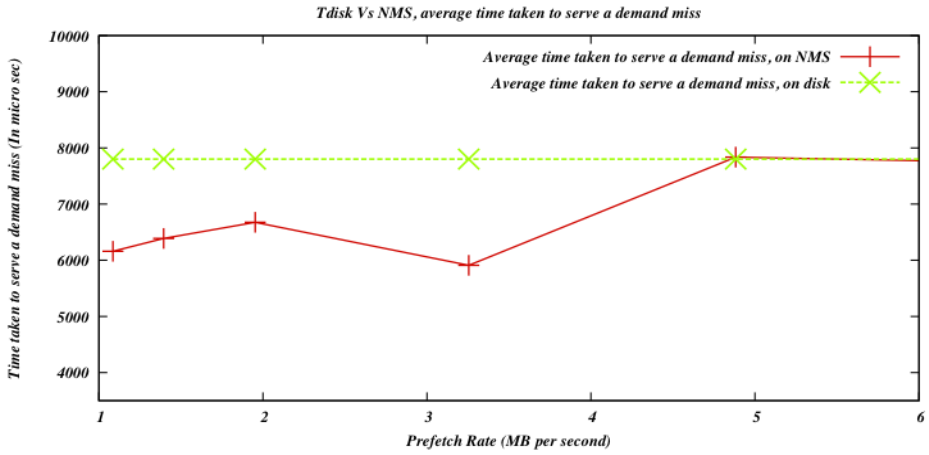

Figure 6.6: The average time to service a demand miss, NMS vs Disk, having demand arrival rate = 0.0025 blocks per μsec (2500 blocks per second)

techniques, as the prefetch requests are generated at the file layer. To achieve this, we could have modified any existing file system to implement prefetching strategy, but doing so would require that we understood all parts of the file system such as journalling, meta-data caching etc. Hence, we propose to develop an Experimental File System (EFS) using prototype UXFS [Pate, January, 2003].

6.4.1 Design of Experimental File System (EFS)

This section looks at the design of the Experimental File System (EFS). The EFS file system creates two major directories on the device/partition, as shown in Figure 6.8. The first is called the *SEQUENTIAL* directory; the files stored in this directory will have prefetching enabled i.e. the file system will prefetch blocks for those files. The rate of prefetching will depend on the analysed consumption rate of running streaming applications. The second is called the *DEMAND* directory, the files in this directory will not be prefetched.

Now the two key questions are: where to implement the prefetching

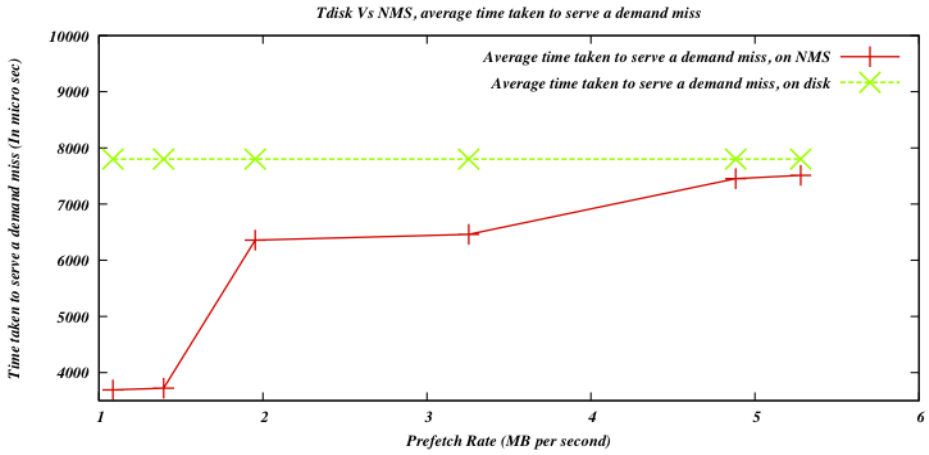

Figure 6.7: The average time to service a demand miss, NMS vs Disk, having demand arrival rate = 0.004 blocks per μsec (4000 blocks per second)

and clustering techniques, and how to store and return the prefetch blocks to the system when they are required. In order to answer these questions, it is necessary to understand how read calls are made to the file system using the page cache, which is explained in the next section.

6.4.2 Read Calls to the File System via the Page Cache

The page cache is - as the name suggests - a cache of physical pages. In the UNIX world, the concept of a page cache became popular with the introduction of SVR4 UNIX, where it replaced the buffer cache for data IO operations.

While the SVR4 Page Cache is only used for filesystem data cache and thus uses the *struct vnode* object and an offset into the file as hash parameters, the Linux Page Cache is designed to be more generic, and therefore uses a *struct address_space*, (explained below), as the first parameter. Because the Linux Page Cache is tightly coupled to the notation of address spaces, we need at least a basic understanding of address_spaces to understand the way the page cache works. An address_space is a structure that allows memory

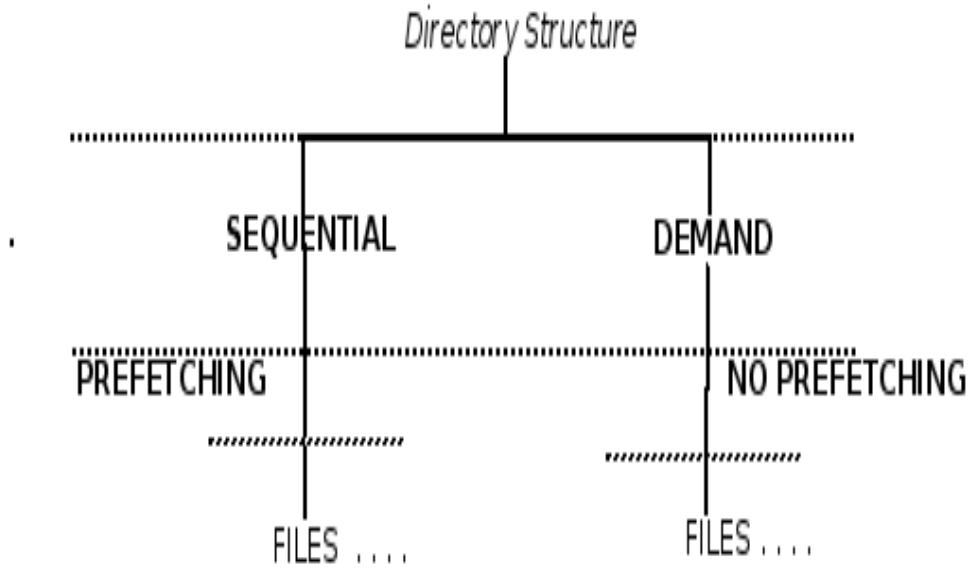

Figure 6.8: Design of Experimental File System.

to be effectively assigned to processes. Hence, it maps all pages of one object (e.g. inode) to another currency (typically physical disk blocks). The struct address_space is defined in include/linux/fs.h as:

Listing 6.2: Definition of address space structure

```
struct address_space {
        struct list_head         clean_pages;
        struct list_head         dirty_pages;
        struct list_head         locked_pages;
        unsigned long            nrpages;
        struct address_space_operations *a_ops;
        struct inode             *host;
        struct vm_area_struct    *i_mmap;
        struct vm_area_struct    *i_mmap_shared;
        spinlock_t               i_shared_lock;
                };
```

To understand the way address_spaces work, we only need to look at a

few of its fields: *clean_pages*, *dirty_pages* and *locked_pages* are double linked lists of all clean, dirty and locked pages that belong to this address_space, *nrpages* is the total number of pages in this address_space. *a_ops* defines the methods supported by the address space structure and *host* is a pointer to the inode which is used to map data into the address_space. A NULL host pointer is associated with the swapper address_space (mm/swap_state.c,). The address space operations are defined by the file system and they are called by the system whenever the data belonging to a page needs to be written to or read from the storage device.

The usage of *clean_pages*, *dirty_pages*, *locked_pages* and *nrpages* is obvious, so we will take a closer look at the address_space_operations structure, defined in the same structure:

Listing 6.3: Definition of address space operations structure

```
struct address_space_operations {
   int (*writepage)(struct page *);
   int (*readpage)(struct file *, struct page *);
   int (*sync_page)(struct page *);
   int (*prepare_write)(struct file *, struct page *,
    unsigned, unsigned);
   int (*commit_write)(struct file *, struct page *,
    unsigned, unsigned);
   int (*bmap)(struct address_space *, long);
   };
```

For a basic view of the workings of address_spaces and the page cache we need to understand the *readpage* function.

It may be understood what the address_space_operations methods do by virtue of their names alone; nevertheless, they do require some explanation. Their use in the course of filesystem data I/O, by far the most common path through the page cache, provides a good way of understanding them. Unlike most other UNIX-like operating systems, Linux has generic file op-

erations, (a subset of the SYSVish vnode operations), for data I/O through the page cache. This means that the data will not directly interact with the file-system for read/write/mmap calls, but it will be read from / written to the page cache whenever possible. The page cache has to get the data from the actual low-level file-system in case the user wants to read from a page not yet in memory, or write data to disk when memory gets low.

In the read path, the generic methods will first try to find a page that matches the wanted inode/index tuple, then it tests whether the page actually exists, as shown in listing 6.4.

Listing 6.4: Searching for the page in the Page Cache

```
hash = page_hash(inode->i_mapping, index);
page =__find_page_nolock(inode->i_mapping,index,*hash);
```

If it does not exist, it allocates a new free page, and adds it to the page cache hash, as shown in listing 6.5.

Listing 6.5: Allocating a Page and adding it to the Page Cache

```
page = page_cache_alloc();
__add_to_page_cache(page, mapping, index, hash);
```

After the page is hashed, it then uses the *readpage* function from the address_space operation to actually fill the page with data, as shown in listing 6.6.

Listing 6.6: adjusting arrival rate to boundaries of multiple of 25

```
error = mapping->a_ops->readpage(file, page);
```

Finally, we can copy the data to user space. Note that the *readpage* function is defined by the file system and it will be called when the data is not found in the cache but is required. Similarly, the *writepage* function from the *address_space_operations* will be called when the system needs to write pages to the storage device.

Having this knowledge, we can now start looking at how to insert the proposed designed into a real system.

6.4.3 Insertion

In a common file system, *readpage* function will call *block_read_full_page* function which allocates the buffer to the newly allocated page and then calls *get_block function*, (defined by the file system), which returns the sector number and maps the buffer to the sector number. Our intention is to modify the *get_block function* so that it will not only map the buffer to the block number but it also copies the data into the buffer. If the file block belongs to a sequential file then it will also initiate prefetching for that file, i.e. in the get_block_function, it will generate a demand request for the requested block and then it generates prefetching requests for that file.

6.4.4 Working of the EFS and the NMS

Figure 6.9 shows the way the request is processed. It can be seen from the figure that there are two queues: the Demand queue (Dq) and the Prefetch queue (Pq). When the *get_block* function is called, it will add the requested block to the Demand queue. If the block requested is from the sequential file then it will generate another P_{opt} number of prefetch requests and then will add it to the prefetch queue, depending on the value of P_{opt} analysed. If the block request is not from the sequential file then it will only add the requested block to the demand queue. The entries in these queues are made up of *fetch_blk_info* structures. The *fetch_blk_info* structure is defined below:

Listing 6.7: Allocating a Page and adding it to Page Cache

```
struct fetch_blk_info{
 int status; /* AVAILABLE or IN PROCESS */
```

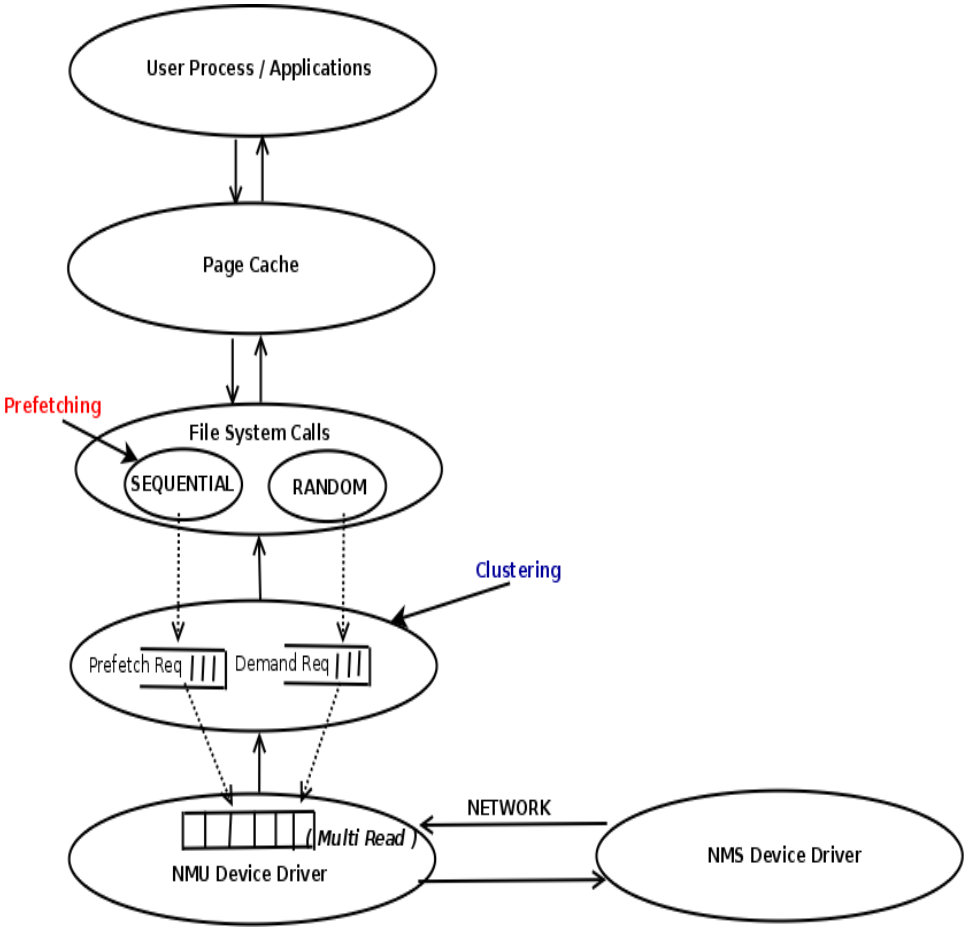

Figure 6.9: Implementation of Prefetching and Clustering.

```
int sector;
int len;
int offset;
char *data; /* will be either pointing to the
system buffer or to our block in circular list */
int Type;
/* is it a Prefetch Request or Demand Request */
struct fetch_blk_data *ptr_data;
/* data block in the circular list */
wait_queue_head_t wait_on_fetch;
/* will wait on this event */
```

```
}
```

Most of the variables in the Listing 6.7 are self-explanatory. If the block request is a demand miss, the data pointer will point to the system buffer. However, if the request is a prefetch request then the data pointer will point to a buffer allocated by our file system and is managed by *struct fetch_blk_data*, i.e. *ptr_data* pointer in the *fetch_blk_info* structure.

Once the requests are added to the respective queues, the *get_block* function will give a signal, i.e. a wakeup call, to the multiread thread to wake up and start processing the requests in the queue. The application thread will sleep until the requested block is fetched.

There is a small routine that calculates the current demand arrival rate and then the proposed algorithm dynamically calculates P_{opt} and D based on the current demand arrival rate and the average per block consumption time of streaming applications.

The multiread thread will cluster requests into a buffer based on the calculated values of P_{opt} and D. The buffer is then passed to the multiread function defined and exported by the NMS device driver. This function will satisfy all the clustered requests in one network operation. Once the requested blocks are returned, the required data will be in the system and the thread will give the necessary wakeup calls to the application threads waiting on the those blocks.

Hence, the multiread will read the demand blocks but will also prefetch blocks using clustering. The demand block will be copied into the system buffer directly, while the prefetch blocks will be copied into the buffers allocated by the file system. The next time the *readpage* function is called, and a sequential block is required, the file system looks into the hash table to see if the block has already been prefetched. If the block is available, the file system can copy the data directly into the user space buffer rather then calling multiread operation. Once data is copied from the prefetch buffer to

the user space buffer the prefetched buffer is freed, as the block belongs to a sequential file and will not be used again. This shows that the clustering will be implemented by the NMS device driver whereas prefetching techniques will be implemented by the file system. We have implemented this design and have a working prototype. Refer to the Appendix B for further details on using the NMS and the EFS.

6.4.5 Evaluation of PonD strategy on the NMS and the EFS

In order to use the proposed algorithm in the working environment we need to measure the current rate of demand misses (λ_d). This is done when a demand miss occurs and the *ux_get_block* function is called to read in the block. A function called *calculatelambda*, which uses *time_val* structures along with a *diff_time* function and the *do_gettimeofday* call, is then used to calculate λ_d by measuring the number of demand misses over a given period. The time period is reset, if there have been over 20 demand misses or a 2 millisecond period has expired.

The function that implements the PonD strategy, (*ux_get_Popt_D*), is then called to calculate the optimal values of P and D based on the measured λ_d and the consume time of the streaming applications. This uses the values of the parameters that were obtained from the simulation which are placed in a hash table, (instead of a database), indexed by the current demand miss rate λ_d. Hence, from this function we get P and D which is used to assemble the request packets that are then sent to the multiread function.

6.4.6 Testing the Prototype System

In order to test the prototype system, an MPEG-4 video player called *totem* was imported onto the EFS and placed in the DEMAND directory, because it is an application program. However, two MPEG streams were placed in the SEQUENTIAL directory and the T_{oct} was set to 400μsec which is the consume time for MPEG-4 videos. Then the *totem* program was started and instructed to show the videos in the SEQUENTIAL directory. As predicted, the current demand miss rate was measured as the *totem* program executed and this was used to invoke the *ux_get_Popt_D* function which calculated P_{opt} and D dynamically and so the required blocks were prefetched accordingly, hence showing that the algorithm works on a normal computer system, as shown in Figure 6.10.

This figure shows the *totem* program playing one of the videos, the EFS measures the demand arrival rate using the *calculatelambda* function. The consume time is set to 400μsec since MPEG video is being viewed. The function *ux_get_Popt_D* calculates the values of P_{opt} and D based on the measured demand arrival rate, which is then used by the *multiread* thread to cluster the prefetch and demand blocks into a network buffer. This clearly demonstrates that the algorithms developed in this thesis can be easily implemented in real networking environments.

6.5 Conclusion

This chapter presented the database design to store operational points and proposed an algorithm to select the values of P_{opt} and D dynamically for a given demand arrival rate. Most importantly, it showed that the proposed QoS for the streaming applications and the demand misses could be provided. The chapter concluded by showing a mechanism to implement the proposed work on a real system.

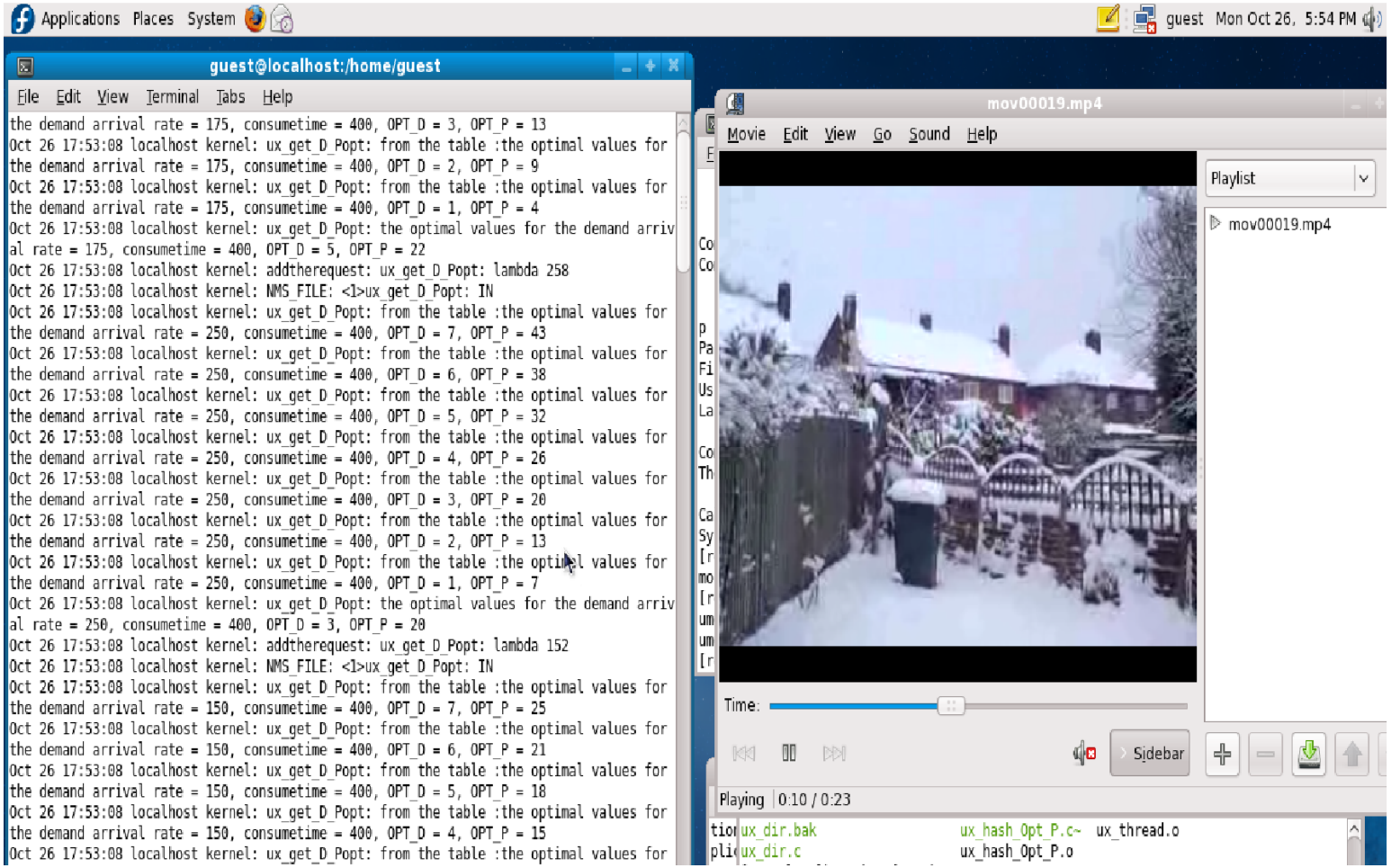

Figure 6.10: Output: Test of a Prototype System.

Chapter 7

Conclusion and Future work

In this chapter, a summary of the work done in this thesis is given where the major contributions are highlighted. This is followed by a conclusion resulting from this work and a discussion on the directions for future research is presented.

7.1 Summary of the Work Done

As the network speed increases and memory becomes cheaper, the usage of network-based applications increases. In addition, the trend of using mobile devices such as Net-Books and Smart Phones, also increases the usage of network-based applications. In this thesis, we have looked at clustering and prefetching techniques over the network so that streaming applications can run without jitter and demand accesses can be satisfied in reasonable time. Experimental evaluation demonstrated that the proposed design can provide the required QoS. It also showed that it can be easily integrated into existing infrastructure, at the file system level and at the block level.

7.2 Contribution to Knowledge

The main contribution of this research is the idea of using clustering and prefetching techniques, in order to provide quality-of-service to today's network-based applications. The contributions of the thesis can be summarised as follows:

1. In Chapter 1, we began by motivating the need to look at prefetching and clustering techniques for network-based applications. Chapter 2, discussed and reviewed the existing prefetching and clustering techniques and it showed that the previous work did not focus on prefetching and clustering techniques over the network. Most of the work was based on disk systems.

2. In Chapter 3, the approach to providing quality-of-service was analysed and simple but very important equations were derived, in order to provide required QoS. The chapter showed that there was a need for an analytical model which can estimate the average time experienced to serve a demand miss at run time.

3. In Chapter 4, the chapter presented an important breakthrough for the research by presenting an analytical model which can represent clustering and prefetching over the network. The model was able to estimate the average time taken to serve a demand miss, given the time to bring a block over the network and the arrival rates of prefetching and demand misses. The results from the analytical model were close to the results obtained by simulation. It was also shown to be effective over a wide operational range. In addition, the model itself, which is gate-limited, can be applied to several other areas including transport where gate-limited service, for e.g. in buses or trains, is common. In addition, a general solution was outlined.

4. Chapter 5 described and explored the operational space using the fundamental constraints discussed in Chapter 3. Using the results derived

from the simulation and by applying the fundamental constraints, it showed how to derive the optimal operational points. These were derived for different values of demand arrival rates and prefetch rates. Finally, it proposed the development an autonomic algorithm, which could be used in a real system.

5. In Chapter 6, we stored the optimal points derived in Chapter 5, in a database and showed that they can be used dynamically to satisfy different demand arrival rates. Furthermore, it demonstrated the design and implementation of an Experimental File System in which the algorithm was implemented. This showed that this work can be directly applied to current networking environments.

7.3 Conclusion

In conclusion, the thesis showed that based on the networking environment it is possible to provide QoS for streaming applications to run without jitter and demand misses to be satisfied in reasonable time over modern computer networks which answers the research question in this thesis.

7.4 Future Work

This dissertation has raised various issues that need to be addressed. Several interesting problems are discussed below as well as potential avenues for further research.

7.4.1 Exploring the Effects of Network Loads

We have looked at prefetching and clustering techniques over the network by using the NMS and the EFS. However, the performance of the NMS and, prefetching and clustering techniques relied on the underlying network. In our work we assumed that the network was lightly loaded, i.e. that there was no other traffic other than NMS-related traffic on the network.

Researchers, at Middlesex University, have studied the workings of the NMS using analytical models, to analyse the effect on the performance and availability of the NMS server due to the other traffic on the network (non-NMS related).

They also varied the service time experienced using the NMS server which consists of: the service time experienced on the network, (μ_1), and the actual time to serve the request on the NMS server $s(\mu_2)$ [Gemikonakli et al., 2006]. The study showed that effective client caching strategy should be used in order to reduce the traffic generated by reading over the network and to improve the performance of the NMS. In this thesis, caching and buffering on the client were not examined and so more work needs to be done in this area.

Furthermore, this work was extended to look at parallel processors with break-downs and repairs, [Gemikonakli et al., 2007]. This method makes extensive use of spectral expansion techniques, [Chakka and Ru, 1995], which assumes that server availability increases or decreases in a monotonic fashion. However, this is not true for the analytical model for gate-limited service developed in this thesis, where the number of servers available is dependent on the number of requests at the end of the previous service time. Therefore, more work is needed to integrate these models to get an accurate picture of the overall effects of network loads on prefetching and clustering.

Another way of improving the overall performance is to reduce the latency experienced over the network. The work being carried out by Silcott

[2005], at Middlesex University, is looking at decreasing the latency experienced due to protocol processing, hence enabling higher throughputs to be achieved. Furthermore, the increase in the usage of iPhones, Blackberrys etc., also increases the use of network-based applications over wireless networks. The issue of network bandwidth for wireless networks could be resolved by using Next Generation technology such as 802.11n technology and by providing QoS to different applications as explored in the work done by Shaikh et al. [2007] .

7.4.2 Managing Multiple Streams

Our work proposed and evaluated a design to provide QoS to streaming applications while satisfying demand misses. However, there is still work to be done to support multiple streams especially the starting and stopping of streams which must be smoothly handled. The transition mechanism should allow the new streaming application to be added to existing streams without compromising the QoS for already running applications and at the same time providing the QoS to the new application. Similarly, when a streaming application terminates the number of blocks being fetched should be adjusted accordingly.

7.4.3 Effects of Prefetching on Caching on the Client Machine

In our work, we assumed that there is a huge amount of memory available on the client side and hence, we proposed to prefetch as many blocks as we can for streaming applications on a demand miss, i.e., prefetching enough number of blocks so that streaming applications can run without jitter. However, the reduced availability of memory buffers due to prefetching might affect the performance of the UNIX buffer cache. Also, mobile devices will not have a

huge amount of memory, so prefetching too many blocks into mobile devices will not be an ideal strategy. We believe that this needs to be looked at urgently.

7.4.4 Towards an Explicitly Caching File System

One of the ways of improving the overall performance is to allow applications to indicate the kind of access they require to files. Our Experimental File System can be used to allow an application to indicate which files it will access sequentially by placing them in the sequential directory allowing these files to be prefetched and not cached. We can extend this idea to look at other access patterns such as looping.

7.5 Final Statement

BBC-iPlayer / You-Tube are rapidly becoming the main method of personal interaction in society. Also, as the speed of the network increases, there is an increasing demand for storing data over the network. In order to allow streaming applications and to satisfy demand misses over the network, we have proposed a framework where streaming applications can run without jitter and demand misses can be satisfied in reasonable time. However, this contribution represents a small step towards the fundamental goal of providing QoS support for network applications.

Bibliography

A definition of 802.11 technologies. Technical report, http://en.wikipedia.org/wiki/IEEE 802.11.

Thomas Anderson, Michael Dahlin, Jeanna Neefe, David Patterson, Drew Roselli, and Randolph Wang. Serverless network file systems. In *Proceedings of the 15th Symposium on Operating System Principles. ACM*, pages 109–126, Copper Mountain Resort, Colorado, December 1995. URL citeseer.ist.psu.edu/anderson95serverless.html.

B. Avi-Itzhak, W. L. Maxwell, and L. W. Miller. Queuing with Alternating Priorities. volume 13, pages 306–318, 1965. doi: 10.1287/opre.13.2.306. URL http://or.journal.informs.org/cgi/content/abstract/13/2/306.

W. Bux. Local-Area Subnetworks: A Performance Comparison. volume 29, pages 1465–1473, Oct 1981.

Pei Cao, Edward W. Felten, Anna R. Karlin, and Kai Li. A study of integrated prefetching and caching strategies. In *SIGMETRICS '95/PERFORMANCE '95*, pages 188–197, New York, NY, USA, 1995. ACM Press. ISBN 0-89791-695-6. doi: http://doi.acm.org/10.1145/223587.223608.

Ram Chakka and Newcastle Upon Tyne Ne Ru. Spectral Expansion Solution for a Finite Capacity Multiserver System in a Markovian Environment. In *Proceedings of the 3rd International Workshop on Queueing Networks with Finite Capacity*, pages 6–1, 1995.

W. W. Chu and A. G. Konheim. On the analysis and modeling of a class of computer-communications system. volume vol. COM-20, pages 645–660, June 1972.

Edgar F. Codd. Recent Investigations into Relational Data Base Systems. In *IBM Research Report RJ1385*, April 23rd, 1974.

Edgar F. Codd. Further Normalization of the Data Base Relational Model. In *IBM Research Report RJ909*. Courant Computer Science Symposia Series 6,, August 31st, 1971.

Edgar F. Codd. A Relational Model of Data for Large Shared Data Banks. In *Communications of the ACM*, volume 13 of *doi:10.1145/362384.362685*, pages 377–387. ACM Press, June 1970.

Y. Dallery, R. David, and X.-L. Xie. Approximate analysis of transfer lines with unreliable machines and finite buffers. volume 34, pages 943–953, Sep 1989. doi: 10.1109/9.35807.

Bertsekas Dimitri and Gallager Robert. *Data Networks*. Prentice Hall, second edition, 1992.

E. Felton and J. Zahorjan. Issues in the implementation of a remote memory paging system. Technical report, University of Washington, March 1991.

S. Gadde, M. Rabinovich, and J. Chase. Reduce, Reuse, Recycle: An Approach to Building Large Internet Caches. volume 0, page 93, Los Alamitos, CA, USA, 1997. IEEE Computer Society. ISBN 0-8186-7834-8. doi: http://doi.ieeecomputersociety.org/10.1109/HOTOS.1997.595189.

O Gemikonakli, G Mapp, D Thakker, and E Ever. Modelling and Performability Analysis of Network Memory Servers. In *ANSS '06*, pages 127–134, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2559-8. doi: http://dx.doi.org/10.1109/ANSS.2006.30.

O. Gemikonakli, G. Mapp, E. Ever, and D. Thakker. Modelling Network Memory Servers with Parallel Processors, Break-downs and Repairs. In

ANSS '07: Proceedings of the 40th Annual Simulation Symposium, pages 11–20, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2814-7. doi: http://dx.doi.org/10.1109/ANSS.2007.29.

S. B. Gershwin and M.H. Burman. A decomposition method for analyzing inhomogeneous assembly/disassembly systems. volume 93, pages 91–115. Annals of Operation Research, 2000.

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-757-5. doi: http://doi.acm.org/10.1145/945445.945450.

Garth Gibson. Parallel NFS, 2008. URL http://www.pnfs.com/.

Donald Gross and Carl M. Harris. *Fundamentals of Queueing Theory (Wiley Series in Probability and Statistics)*. Wiley-Interscience, February 1998. ISBN 0471170836. URL http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0471170836.

J. F. Hayes and D. N. Sherman. A Study of Data Multiplexing Techniques and Delay Performance. volume 51, pages 1983–2011, November 1971. doi: 10.1287/opre.13.2.306.

J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and M. West. Scale and performance in a distributed file system. volume 21, pages 1–2, New York, NY, USA, 1987. ACM Press. doi: http://doi.acm.org/10.1145/37499.37500.

A. R. Kaye and T. G. Richardson. A performance criterion and traffic analysis for polling systems. volume 11, pages 93–112.

L. Kleinrock. *Queueing Systems, Volume 2: Computer Applications*. Wiley, 1976.

A. Kobayashi, H.; Konheim. Queueing Models for Computer Communications System Analysis . volume 25, pages 2 – 29, Jan 1977.

John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinskiand Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Chris Wells, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 190–201, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-317-0. doi: http://doi.acm.org/10.1145/378993.379239.

Arnaud Legout, Nikitas Liogkas, Eddie Kohler, and Lixia Zhang. Clustering and sharing incentives in BitTorrent systems. In *SIGMETRICS '07: Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 301–312, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-639-4. doi: http://doi.acm.org/10.1145/1254882.1254919.

M. A. Leisowitz and A. G. Konheim. Mathematical models for computer data communication. In Case Studies in Mathematical Modeling. pages 256–334, 1980.

Chuanpeng Li, Kai Shen, and Athanasios E. Papathanasiou. Competitive prefetching for concurrent sequential I/O. volume 41, pages 189–202, New York, NY, USA, 2007. ACM. doi: http://doi.acm.org/10.1145/1272998.1273017.

C. Mack. The efficiency of N machines uni-directionally patrolled by one operative when walking time is constant and repair times are variable. pages 173–8, 1957a.

C. Mack. The efficiency of N machines uni-directionally patrolled by one operative when walking time is constant and repair times are constants. pages 166–172, 1957b.

Drew Major, Greg Minshall, and Kyle Powell. An overview of the NetWare operating system. In *WTEC'94: Proceedings of the USENIX Winter*

1994 Technical Conference on USENIX Winter 1994 Technical Conference, pages 27–27, Berkeley, CA, USA, 1994. USENIX Association.

G. Mapp, D. N. Cottingham, and F. Shaikha. An Architectural Framework For Heterogeneous Networking. In *the International Conference on Wireless Information Networks and Systems.*, pages 5–12, August 2006.

Glenford Mapp, Dhawal Thakkar, and David Silcott. Network Memory Servers: An idea whose time has come. In *Multi-Service Networks (MSN)*, Coseners House, UK, July 2004.

Glenford Mapp, Dhawal Thakker, and David Silcott. The Design of a Storage Architecture for Mobile Heterogeneous Devices. volume 0, page 41, Los Alamitos, CA, USA, 2007. IEEE Computer Society. ISBN 0-7695-2858-9. doi: http://doi.ieeecomputersociety.org/10.1109/ICNS.2007.113.

Glenford Mapp, Dhawal Thakker, and Orhan Gemikonakli. Exploring Gate-Limited Analytical Models for High Performance Network Storage Servers. volume 0, pages 1–5, Los Alamitos, CA, USA, 2009. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/ICCCN.2009.5235246.

Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. volume 6, pages 134–154, New York, NY, USA, 1988. ACM Press. doi: http://doi.acm.org/10.1145/35037.42183.

Athanasios E. Papathanasiou and Michael L. Scott. Energy efficient prefetching and caching. In *ATEC '04: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 22–22, Berkeley, CA, USA, 2004. USENIX Association.

Athanasios E. Papathanasiou and Michael L. Scott. Aggressive prefetching: an idea whose time has come. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 6–6, Berkeley, CA, USA, 2005. USENIX Association.

Steve D. Pate. UNIX Filesystems: Evolution, Design, and Implementation, January, 2003. URL http://code.google.com/p/uxfs/wiki/layout.

R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 79–95, New York, NY, USA, 1995. ACM. ISBN 0-89791-715-4. doi: http://doi.acm.org/10.1145/224056.224064.

B. K. Penny and A. A. Baghoadi. Survey of computer communications loop networks. volume 2, pages 165–180 and 224–241, 1979.

M. Reiser. Performance evaluation of data communications systems. pages 171–196, 1982.

David Rochberg and Garth Gibson. Prefetching over a network: early experience with CTIP. volume 25, pages 29–36, New York, NY, USA, 1997. ACM. doi: http://doi.acm.org/10.1145/270900.270906.

Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and Implementation of the Sun Network Filesystem. In *Proc. Summer 1985 USENIX Conf.*, pages 119–130, Portland OR (USA), 1985. URL citeseer.ist.psu.edu/sandberg85design.html.

Fatema Shaikh, Glenford Mapp, and Aboubaker Lasebae. Proactive Policy Management using TBVH Mechanism in Heterogeneous Networks. In *NGMAST '07: Proceedings of the The 2007 International Conference on Next Generation Mobile Applications, Services and Technologies*, pages 151–157, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2878-3.

S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network File System (NFS) version 4 Protocol. 2003. URL http://www.ietf.org/rfc/rfc3530.txt.

David Silcott. Distributed Operating System for High Performance Servers. 2005.

Simon LAM January and Simon S. LAM. Tr-88 Multiple Access Protocols*. In *Computer Communications, Volume I: Principles, Englewood Cliffs, NJ.* Prentice Hall, 1983.

S. Stidham. Optimal control of a signalized intersection. Technical report, Cornell Univ, Ithaca, NY, 1969.

H Takagi. Queueing analysis of polling models: An update, Stochastic Analysis of Computer and Communication Systems. Technical report, North-Holland, Amsterdam, 1990.

Hideaki Takagi. Queuing analysis of polling models. volume 20, pages 5–28, New York, NY, USA, 1988. ACM. doi: http://doi.acm.org/10.1145/62058.62059.

Hideaki Takagi. Analysis and Application of Polling Models. In *Performance Evaluation: Origins and Directions*, pages 423–442, London, UK, 2000. Springer-Verlag. ISBN 3-540-67193-5.

Dhawal N. Thakker, Glenford E. Mapp, and Orhan Gemikonakli. Modelling Mixed Access-Patterns in Network-Based Systems. In *UKSIM '09: Proceedings of the UKSim 2009: 11th International Conference on Computer Modelling and Simulation*, pages 514–519, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3593-7. doi: http://dx.doi.org/10.1109/UKSIM.2009.67.

M Van Vuuren and Adan IJBF. Performance analysis of assembly systems. pages 89–100, Charleston, 2006. Proceedings of the Markov anniversary meeting.

M Van Vuuren, Adan IJBF, and Resing-Sassen SA. Performance analysis of multi-server tandem queues with finite buffers and blocking. volume 27, pages 315–338, Charleston, 2005. OR Spectrum.

M Van Vuuren and E Winands. Iterative approximation of k-limited polling systems. Technical report, Technische Universiteit Eindhoven, May 2006.

Yung-Terng Wang and R. J. T. Morris. Load Sharing in Distributed Systems. volume 34, pages 204–217, Washington, DC, USA, 1985. IEEE Computer Society. doi: http://dx.doi.org/10.1109/TC.1985.1676564.

Appendices

Appendix A

Case Study - Database

The data below shows the number of prefetch requests and demand requests that could be clustered into a network buffer for a given network condition and demand arrival rate.

It could be seen from the Figure A.1 that for a given demand arrival rate of $0.005(\frac{1}{200})$ blocks per μsec (5000 blocks per second) and the value of $D = 4$, the number of prefetch blocks that should be fetch is equal to 20 blocks, supporting $T_{cpu} = \frac{T_{disk}}{P_{opt}} = \frac{7800}{20} = 390\mu sec$ (supporting prefetch rate of 2.5 MB per second).

	Value of P_max for each value of D and mean Demand arrival time						
Demand Arrival Rate	D = 1	D = 2	D = 3	D = 4	D = 5	D =6	D = 7
100	2	4	6	8	11	13	15
125	3	6	9	12	14	17	20
150	3	7	11	15	18	21	25
175	4	9	13	18	22	26	29
200	5	10	14	20	25	30	34
225	6	12	18	23	29	34	39
250	7	13	20	26	32	38	43
275	7	15	22	29	35	41	47
300	8	15	24	31	38	45	51
325	9	18	25	32	42	48	55
350	10	18	28	33	44	52	58
375	10	21	30	33	48	56	62
400	10	21	30	40	50	58	65
425	12	22	30	40	50	61	68
450	13	25	35	47	56	64	71
475	13	25	37	45	57	66	74
500	14	28	40	50	60	69	77

Figure A.1: Shows the analysed values of P_{opt} and D for a demand mean arrival time, where demand mean arrival time is a multiple of 25μsec and greater than 100μsec.

Appendix B

Using the NMS and the EFS

The following are the steps to use the NMS and the EFS. The development is distributed in three source packages and they are the NMS server, NMS client and EFS file system modules. The kernel version required for these packages to work is 2.6.27, any other kernel version will require porting of the above packages to the corresponding kernel version.

1. **Compiling**: First step is to compile the above packages. Each package has its own Makefile file. If you have the right kernel version, then *make* command will compile the source package to get the corresponding module.

2. **Installing**: In order to install the above compiled modules, you will need to use the *insmod* command. For example, for NMS server, it would be: *insmod nms.ko*.
   NMS SERVER
   The server module can be installed without passing any parameters, if it is installed without passing any parameters, then it will use the default values for the TCP port number and the memory that needs to be allocated. However, if there is a need to specify these parameters

i.e. the TCP port number and memory to be allocated, you can pass parameters while installing the module. The information on passing parameters for each module could be known by using command *modinfo modulename.ko* e.g. modinfo nms.ko.

NMS CLIENT

While installing the NMS client module, one has to pass the IP-address of the NMS server as a parameter. Remember, if the NMS server's IP address is not specified then it will assume that the NMS server is running on the same system. The NMS server should be installed before a client is installed, as client will communicate with the NMS server on its installation. For security reasons, each client have a user_id and password which is stored in a file. This file is stored on a TFTP server. Hence, you will also need to pass IP address of the TFTP server as a parameter, while loading a NMS client. A common example of installing NMS client: insmod nmssneh.ko servip=192.168.10.1 tftp_ip=192.168.10.4

EFS

Installing the EFS file system is very simple. Once the EFS source package is compiled the module will be available to install. Use command *insmod uxfs.ko*, to install. For it to install successfully, you need to make sure that NMS client is installed, as the EFS code is referencing to the multiread function, which is exported by the NMU client.

3. **Creating Partition**: Once all the above modules are installed. The /dev/nmu device will be created in your system to communicate with the NMS server. If it is not created, then you will need to create it manually by finding out the major and minor number of the NMS server module and using the *mknod* command. Once /dev/nmu is available, you can treat /dev/nmu as any other block device such as /dev/sda or /dev/hda. Creating partitions on this device, would be simple as using command *fdisk /dev/nmu*.

4. **Creating File System**: The partition created above could have any file systems by using command *mkfs -t fs /dev/nmu1*. Note that the

device will be */dev/nmu1* and not /dev/nmu. We will the use the EFS file system on the partition created.

In order to create the EFS file system, you will need to change to the cmd directory under the EFS source directory. Compile the mkfs.c file found in the cmd directory by using command, *gcc -o mkfs mkfs.c* and then create the EFS file system on the device, using command: *./mkfs /dev/nmu1*.

5. **Mounting File system**: The device can be mounted as any other storage devices, using command: *mount -t uxfs /dev/nmu1 /media/nmu1*, where /media/nmu1 is the mount point, that you will need to create using command *mkdir /media/nmu1*. Once the device is mounted, anything read/written to the mount point will be read/stored from/to the NMS server.

Appendix C

Simulation Code

In this APPENDIX, we have presented the detail of the simulation code which represents Network Memory Server.

Listing C.1: Simulation Code Representing NMS: simulation_PonD.cpp

```
1  #include "Request.h"
2  using namespace std;
3
4  list<Request> QDemand;
5  list<Request> QServed;
6  list<Event> QEvent;
7  list<Event> QFreeEvent;
8
9  /*  Variable defined, which are need in the function
10       to avoid the stack over flow
11 */
```

```
12
13 double timetoserve = 0;
14 int noreq = 0;
15 list<Request>::reverse_iterator rev_i;
16 int reqno =0, count = 0;
17 int reqsize = 0;
18
19 int main()
20 {
21   int i = 0, ret = 0;
22   CalculateD(); /* this will set the maximum number of D ↩
       that can be served given P and Latency (Delay) and ↩
       Tprocess */
23 #ifdef Database
24   Connect2Database();
25   SearchValueofPD(ARR_T, P_CONSUMET, MaxD, P); /* we need↩
        to set value of P and MaxD */
26   printf("Value assing from database for MaxD = %d, and P↩
       = %d", MaxD, P);
27 #endif
28
29 #ifndef Database
30   MaxD = 7;
31   P = 1;
32 #endif
33   if(P == 0){
```

```
34      cout<<"when P = 0, there will be no vacation event"<<↩
            endl;
35      cout<<"This simulation yet does not simulate this ↩
            scenairo"<<endl;
36      cout<<"We need to set MaxD Manually"<<endl;
37      return -1;
38    }
39    lemda = (1.0 / ARR_T);
40    Mu = (1.0) / (Delay + (P + MaxD) * C );
41
42    if(lemda >= (Mu * MaxD) ){
43      cout<<"Arrival rate is greater than Service Rate ! ! ↩
            ! !, I am going to exit" <<endl;
44      cout<<"Lemda = "<<lemda<<", Mu * MaxD = "<<(Mu * MaxD↩
            )<<endl;
45
46      exit(0);
47    }
48
49    now_time = 0;
50    srand( (unsigned) time(NULL));/* seed */
51    GenerateEvent(ARRIVAL, ARR_T, 0);
52
53    while(NumberOfJobServed<JOBS){
54
55    switch(phase())
```

```
56    {
57      case ARRIVAL:
58      //  cout<< "ARRIVAL";
59        ret = DealWithArrival();
60        if (ret < 0)
61          break;
62        break;
63
64      case SERVICE:
65      //  cout<< "SERVICE";
66        DealWithService();
67        break;
68
69      case -1:
70        cout<<"Switich Error"<<endl;
71        break;
72
73    }
74    i++;
75    }
76    printout();
77    #ifdef Database
78    CloseDatabase();
79    #endif
80    return 0;
81  }
```

```
82
83
84  int phase()
85  {
86    /* Take the first event and
87     * return what time of event
88     * we are going to deal with
89     */
90    if(QEvent.empty()){
91      cout<<"The event queue is empty, Somthing has gone ↩
          wrong"<<endl;
92      return -1;
93    }
94    Event Etmp = QEvent.front();
95    if(Etmp.Type == ARRIVAL)
96      return ARRIVAL;
97    else if(Etmp.Type == SERVICE)
98      return SERVICE;
99
100   return -1;
101 }
102
103 int DealWithArrival(){
104   Event Etmp = QEvent.front();
105   timetoserve = noreq = 0;
106   if(Etmp.Type == ARRIVAL){
```

```
107     /* put the request in to the demand queue */
108     /* and also create next event for the arrival */
109     now_time = Etmp.Event_time;
110     Request Dnew(ARRIVAL, now_time);
111     QDemand.push_front(Dnew);
112     QEvent.pop_front();
113     GenerateEvent(ARRIVAL, ARR_T, 0);
114     if(SERVER_STATE == SERVER_BUSY)
115       return 1;
116     else{
117       //cout<<"Server is free and will serve probably 1 ↩
            request "<<endl;
118       CountNoServerIdeal++;
119       SERVER_STATE = SERVER_BUSY;
120       timetoserve = Service(noreq);
121       GenerateEvent(SERVICE, timetoserve, noreq);
122     }
123   }
124   else{
125     cout<<"DealWithArrival: Wrong Call to DealWithArrival↩
          "<<endl;
126     return -1;
127   }
128   return 1;
129
130 }
```

```
131
132 int DealWithService(){
133
134   /* Take out the jobs from the Demand Q, as service ↩
        Completion */
135   /* also create next event for the service completion*/
136   timetoserve = noreq = 0;
137   Event Etmp = QEvent.front();
138   now_time = Etmp.Event_time;
139   if(Etmp.Type == SERVICE){
140     /* On completion of Service; We now remove requests ↩
          from DemandQ*/
141     if(Etmp.NoReq != 0) RemoveReqDemandQ(Etmp.NoReq);
142     else cout<<"DealWIthService: Error with NoReq"<<endl;
143     QEvent.pop_front();
144
145     /* Going Forward, As Glenford Says
146      * If QDemand is not empty then create another ↩
           service
147      * completion event, else assign SERVER_STATE == FREE
148      */
149
150     if(QDemand.size()>0){
151     timetoserve = Service(noreq);
152     GenerateEvent(SERVICE, timetoserve, noreq);
153     }
```

```
154       else {
155         SERVER_STATE = SERVER_FREE;
156       }
157     }
158     else{
159       cout<<"DealWithService: Wrong Call to DealWithService↩
            "<<endl;
160       return -1;
161     }
162
163     return 0;
164
165 }
166
167 int GenerateEvent(int type, double interval, int noreq){
168     if(type == ARRIVAL || type == VACATION){
169             double exp = exponential(1.0/interval);
170       interval = exp;
171     }
172     interval += now_time;
173     Event Enew (type, interval, noreq);
174     /* we cannot just stick the request at the end
175      * or front of the queue
176      * it has to be order based on the start time */
177     QEvent.push_back(Enew);
178     QEvent.sort();
```

```
179 //  printoutEvent();
180   return 0;
181 }
182
183 double Service(int& noreq)
184 {
185   reqno = count = 0;
186   timetoserve = 0;
187   if(QDemand.empty()){
188     cout<< "There is no demand request in the queue"<<↩
          endl;
189     return -1;
190   }
191   /* check the queue size if queue size > D */
192   reqno = QDemand.size()>MaxD?MaxD:QDemand.size();
193   timetoserve = Tnet(P + reqno);
194 //  cout<<"Timetoserve"<<timetoserve<<endl;
195   timetoserve = exponential(1.0/timetoserve);
196   /* take the last d requests from the Queue */
197   count = 0;
198   for(rev_i=QDemand.rbegin(); count < reqno; ++rev_i, ↩
       count++) {
199     (*rev_i).ReqWaitingT = (now_time - (*rev_i).↩
          ReqArrived);
200     (*rev_i).ReqServiceT = (timetoserve);
201   }
```

```
202   servicecount++;
203   noreq = reqno;
204   return (timetoserve);
205 }
206
207 int RemoveReqDemandQ(int Number)
208 {
209   MeanD += Number;
210   NumberOfJobServed += Number;
211   reqsize = QDemand.size()-Number;
212   /*delete the tmp entry from the QDemand and add it to ↩
        the QServed */
213   for(int i = 0; i< Number; i++){
214     Request tmp = QDemand.back();
215     tmp.served = true;
216     tmp.ReqTotalWaitingT = tmp.ReqWaitingT + tmp.↩
          ReqServiceT;
217     tmp.WaitingReq = reqsize;
218     QServed.push_back(tmp);
219     QDemand.pop_back();
220   }
221   return 0;
222 }
223 int CalculateD()
224 {
225
```

```
226   MaxD= ( (Tprocess * P) - Tnet(P  * NoApp) ) / C;
227   cout<<"Maximum Number of demand request that can be ↩
          served() = " << MaxD << endl;
228   return 0;
229 }
230
231 double Tnet(int y)
232 {
233   return ((Delay + C * y));
234 }
235
236
237 int printoutEvent(){
238
239   list<Event>::iterator i;
240   int count = 1;
241   cout <<"Number of Events " << QEvent.size() <<endl ;
242   /* once the time is over we need to print out the info ↩
          */
243   cout << "Please note all the time measurements are in ↩
          micro seconds" << endl;
244   for(i=QEvent.begin(); i != QEvent.end(); ++i, count++) ↩
          {
245     cout << count <<"\t\t" <<*i << " "; // print all
246   }
247   return 0;
```

```
248 }
249
250
251 int printout(){
252
253   list<Request>::iterator i;
254   int count = 1;
255   cout <<"Number of Requests Served " << QServed.size() ↩
        <<endl ;
256   total_wait_S_T = 0;
257   total_wait_T = 0;
258   total_len = 0;
259
260   /* once the time is over we need to print out the info ↩
        */
261   cout << "Please note all the time measurements are in ↩
        micro seconds" << endl;
262   cout<<"Job Number\t" << "Arrival time\t" << "Waiting ↩
        Time\t\t" << "Service Time" << "No. Waiting Request ↩
           " << endl;
263   for(i=QServed.begin(); i != QServed.end(); ++i, count↩
        ++) {
264   // cout << count <<"\t\t" <<*i << " "; // print all
265   total_wait_S_T += (*i).ReqTotalWaitingT;
266   total_wait_T += (*i).ReqWaitingT;
267   total_len += (*i).WaitingReq;
```

```
268   total_serv += (*i).ReqServiceT;
269   }
270   /* We are substracting ARRIVAL from count to neglify the effect of zeros when the simulation starts */
271
272   cout<<"Job Number\t" << "Arrival time\t" << "Waiting Time\t\t" << "Service Time" << "No. Waiting Request " << endl;
273   cout<< "Tprocess = " << Tprocess << ", Number of Prefetch = " << P << ", C = " << C << ", MaxD = "<< MaxD << ", Arrival Rate: " << (ARRIVAL * 1000000 ) / ARR_T << " blocks/sec" << " , DELAY = " << Delay <<endl;
274   cout<< "Time to serve "<< P << "p blocks and " << MaxD << "d blocks is equal to " << (Delay + C * (P+MaxD)) <<"musec" << endl;
275   cout<<"Number of service generated: "<< servicecount <<", Total NUmber of Jobs Served: "<< NumberOfJobServed << ", Total wait "<< total_wait_T <<endl;
276   MeanWT = (total_wait_T / NumberOfJobServed);
277   cout<< "Mean Waiting time (Excluding Service Time)" << MeanWT << endl;
278   TotalWT =  (total_wait_S_T / NumberOfJobServed);
279   cout<< "Mean Waiting time (Including Service Time)" <<TotalWT << endl;
```

```
280
281
282     cout<< "Mean Length: " << (total_len / ↩
            NumberOfJobServed) << endl;
283     MeanServiceTime = (total_serv / NumberOfJobServed);
284     cout<< "Mean Service Time : "<< MeanServiceTime <<endl;
285     MeanD = MeanD/servicecount;
286     cout<< "Mean D: " <<MeanD<<endl;
287     formula();
288     D = MaxD;
289     cout<<"Calculated Using Bulk Service Model"<<endl;
290     BulkService();
291     return 0;
292 }
293
294 double formulaR(double r1)
295 {
296     ret =    ( (Mu * (pow(r1, D+1))) - (lemda + Mu) * r1 + ↩
            lemda );
297
298     return ret;
299 }
300
301 void formulaL()
302 {
303     L = r / (1.0 - r);
```

```
304 }
305
306 void formulaW()
307 {
308   W = ( L / lemda );
309 }
310
311 void formulaWq()
312 {
313   Wq = W - (1.0 / Mu);
314 }
315 void formulaLq()
316 {
317   Lq = L - (lemda / Mu);
318 }
319
320
321 void BulkService(){
322   double i, ans;
323   for(i = 0 ; i < 1; ){
324   ans = formulaR(i);
325   if(ans < 0 ){
326   //  cout<<"ans is less than zero:" << ans<<endl;
327     i  = i +  0.000001;
328     continue;
329   }
```

```
330     else if(ans < least){
331       least = ans;
332       r = i;
333     }
334     i  = i +  0.000001;
335     }
336
337     cout<<"R = "<<r<<", Least = "<<least<< endl;
338
339     formulaL();
340     formulaW();
341     formulaWq();
342     formulaLq();
343     cout<<"Lemda = "<<lemda<<", Mu = "<<Mu<<" Using D = "<<↩
          D<<", MU*D = "<<(Mu*D)<<endl;
344     cout <<"L = "<<L<<", W = "<<W<<", Wq = "<<Wq<<", Lq = "↩
          <<Lq<<endl;
345     double per = ( (W - TotalWT) /TotalWT) * 100;
346     cout<<"Error Percentage: "<<per<<"%"<<endl;
347     cout <<"W + (Mean Service Time / 2 ) : " << (W + (↩
          MeanServiceTime / 2 ))<<endl;
348     cout <<"Wq + (Mean Service Time / 2 ) : " << (Wq + (↩
          MeanServiceTime / 2 ))<<endl;
349 }
```

Listing C.2: Simulation Code Representing NMS: Request.h

```
1  #include <iostream>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <list>
5  #include <string>
6
7  #include <math.h>
8
9  //#define Database  /* UN COMMENT this line; if database ↩
       is used;
10 /* Type of request */
11 #define ARRIVAL 1
12 int ARRIVAL1 = 1;   /* 1 means Demand (ARRIVAL is DEMAND)↩
       */
13 #define PREFETCH 2 /* 2 means Prefetch */
14 #define SERVICE 3
15 #define VACATION 4 /* vacation finished */
16 #define FREE_F  5 /* server free finished */
17
18 #define ARR_T 250 /* demand miss: mean arrival time. In ↩
       microsec, in microsecond */
19 #define P_CONSUMET  350 /* Prefetch per block consume ↩
       time */
20
21 unsigned int P = 0;
```

```
22 unsigned int MaxD = 0;
23 unsigned int CountNoServerIdeal = 0;
24 #define Tprocess 200
25 #define NoApp 1
26 #define Delay 0
27 #define C 30
28 #define JOBS 10000000
29
30 int CalculateD();
31 int printout();
32 double Tnet(int y);
33 double Service(int& noreq);
34 int GenerateEvent(int type, double interval, int noreq);
35 int phase();
36 int DealWithArrival();
37 int DealWithService();
38 int DealWithVacation();
39 int DealWithServerFree();
40 int deleteanEvent(int Event_type);
41 int printoutEvent();
42 int RemoveReqDemandQ(int Number);
43 #ifdef Database
44 #include "Database.h"
45 int Connect2Database();
46 int CloseDatabase();
```

```
47 int SearchValueofPD(int Darrivaltime, int Consumetime, ↩
      unsigned int &D, unsigned int &P);
48 #endif
49 double TotalWT = 0.0;
50 double total_wait_S_T = 0;
51 double total_wait_T = 0;
52 double total_len = 0;
53 double now_time = 0;
54 double MeanD = 0;
55 double total_serv = 0;
56 double MeanWT = 0;
57 double D = 0.0;
58
59 int NumberOfJobServed = 0;
60 int SERVER_FREE = 1;
61 int SERVER_BUSY = 2;
62 /* for vacation */
63 int SERVER_ON_VACATION = 3;
64 int SERVER_STATE = SERVER_ON_VACATION;
65 int vac_time = Delay + (P * C); /* it has to be set after↩
      P */
66
67 double servicecount = 0;
68 /* For Bulk Service */
69 void BulkService();
70 double L = 0, Lq = 0, W = 0, Wq = 0;
```

```
71 double lemda, Mu, least = 1.0, ret, r;
72 double MeanServiceTime = 0;
73
74 using namespace std;
75 class Event
76 {
77
78 friend ostream &operator<<(ostream &, const Event &);
79 public:
80
81   int Type; /* 1 = Arrival
82           2 = Service
83           3 =
84         */
85   double Event_time;
86   int NoReq;
87   Event(int& type, double& Etime, int& No);
88   Event(int& type, double& Etime);
89   int operator<(const Event &rhs) const;
90
91 };
92
93 Event::Event(int& type, double& Etime, int& No){
94   Type = type;
95   Event_time = Etime;
96   NoReq = No;
```

```
97    }
98
99  Event::Event(int& type, double& Etime){
100   Type = type;
101   Event_time = Etime;
102   NoReq = 0;
103   }
104
105 // This function is required for built-in STL list ↩
      functions like sort
106 int Event::operator<(const Event &rhs) const
107   {
108 //      if( this->Start_time == rhs.Start_time ) return 1;
109       if( this->Event_time  < rhs.Event_time ) return 1;
110       return 0;
111   }
112
113 ostream &operator<<(ostream &output, const Event &E)
114   {
115       output << "Event time " << E.Event_time << "\t \t \t↩
            " <<"\t \t Type: " << (E.Type)<<endl;
116       return output;
117   }
118
119 /*********************************/
120
```

```
121 /*********************************/
122 class Request
123 {
124     friend ostream &operator<<(ostream &, const Request ↩
        &);
125 public:
126
127
128   int type; /* 1 = Demand, 2 = Prefetch */
129   bool served;
130   double ReqArrived;
131   double ReqWaitingT;
132   double ReqServiceT;
133   double ReqTotalWaitingT;
134   double WaitingReq;
135
136   Request();
137   Request(int t, double ReqA);
138   Request(const Request &);
139   Request &operator=(const Request &copyin);
140 };
141
142 Request::Request()
143 {
144   type = 0;
145   ReqArrived = 0;
```

```
146   ReqWaitingT = 0;
147   ReqServiceT = 0;
148   served = false;
149   ReqTotalWaitingT = 0;
150   WaitingReq = 0;
151 }
152
153 Request::Request(int t, double ReqA)
154 {// cout<< "Request Constructor " << ReqA<< endl;
155   type = t;
156   ReqArrived = ReqA;
157   ReqWaitingT = 0;
158   ReqServiceT = 0;
159   served = false;
160   ReqTotalWaitingT = 0;
161   WaitingReq = 0;
162 }
163
164 Request::Request(const Request &copyin)    // Copy ↩
    constructor to handle pass by value.
165 {
166 //  cout<<"Copy Constructor is called "<< endl;
167   type = copyin.type;
168   ReqArrived = copyin.ReqArrived;
169   ReqWaitingT = copyin.ReqWaitingT;
170   ReqServiceT = copyin.ReqServiceT;
```

```
171   served = copyin.served;
172   ReqTotalWaitingT = copyin.ReqTotalWaitingT;
173   WaitingReq = copyin.WaitingReq;
174 }
175
176
177 Request& Request::operator=(const Request &copyin)
178 {
179   this->type = copyin.type;
180   this->ReqArrived = copyin.ReqArrived;
181   this->ReqWaitingT = copyin.ReqWaitingT;
182   this->ReqServiceT = copyin.ReqServiceT;
183   this->served = copyin.served;
184   this->ReqTotalWaitingT = copyin.ReqTotalWaitingT;
185   this->WaitingReq = copyin.WaitingReq;
186     return *this;
187 }
188
189 ostream &operator<<(ostream &output, const Request &R)
190 {
191   output << R.ReqArrived << "\t \t \t" << (R.ReqWaitingT)↩
          << "\t \t"<< R.ReqServiceT <<"\t \t "<< R.↩
         WaitingReq <<endl;
192   return output;
193 }
194
```

```
195 double uniform()
196 {
197   int high = 10;
198   int low = 1;
199   return ((double) rand() / (double) RAND_MAX);
200 }
201
202
203
204 double exponential(double mean)
205 {
206   double u;
207   double ret;
208   do{
209     u = uniform();
210   }while (u == 0.0 || u == 1.0);
211   ret =  (-1.0/mean*log(u));
212   if(ret < 0)
213     ret = -(ret);
214   return (ret);
215 }
216
217 void formula()
218 {
219   cout<<"Calculated Using simple M/M/1 queue Model"<<endl↩
        ;
```

```
220   double WaitT = 0, num, denom;
221   num =  (Delay + ( (P +MaxD) * C) );
222     cout <<"Numerator: " << num << endl;
223     denom = (MaxD - ( ( num ) * (double)(1.0/ARR_T) ) ) ;
224   cout << "Denomenator: "<<denom<<endl;
225     WaitT = (num / denom);
226     cout<<"Waiting Time calculate by formula:"<<WaitT<<↩
        endl;
227 }
```

Listing C.3: Simulation Code Representing NMS: Database.h

```
1  #include <sys/time.h>
2  #include <stdio.h>
3  #include <iostream>
4  #include <stdlib.h>
5  #include <mysql.h>
6  #include <string>
7  using namespace std;
8
9  MYSQL_RES *result;
10 MYSQL_ROW row;
11 MYSQL *connection, mysql;
12 int state;
13 int Quo, rem;
14 int Tdisk  = 7800;
```

```
15
16 int Connect2Database(){
17   mysql_init(&mysql);
18   connection = mysql_real_connect(&mysql,"localhost","↩
       root","computer","Research",0,0,0);
19   if (connection == NULL)
20   {
21   printf(mysql_error(&mysql));
22   return -1;
23   }
24   return 0;
25 }
26
27 int CloseDatabase(){
28   mysql_free_result(result);
29   mysql_close(connection);
30 }
31
32 int SearchValueofPD(int Darrivaltime, int Consumetime, ↩
     unsigned int &D, unsigned int &P){
33   int CalArrivaltime = 0;
34   int Quo = Darrivaltime / 25;
35   char qstring[255];
36   P = D = 0;
37     unsigned int Pmax = 0;
38   unsigned int Dmax = 0;
```

```
39  unsigned int arrivalr = 0;
40  CalArrivaltime = (Quo * 25);
41
42  printf("CalArrivaltime = %d, Darrivaltime = %d \n", ↩
       CalArrivaltime, Darrivaltime);
43  sprintf(qstring, "SELECT Pno, Dno, arrivalrate FROM ↩
       OptimalPs, ArrivalRates where OptimalPs.A_id = ↩
       ArrivalRates.A_id and arrivalrate = %d ORDER BY ↩
       arrivalrate, Dno ASC", CalArrivaltime);
44  printf("%s \n", qstring);
45  state = mysql_query(connection,qstring);
46  if (state !=0)
47  {
48    printf(mysql_error(connection));
49    return -1;
50  }
51  result = mysql_store_result(connection);
52
53  printf("Rows:%d\n",mysql_num_rows(result));
54
55  while ( ( row=mysql_fetch_row(result)) != NULL )
56  {
57      Pmax = atoi(row[0]);
58    Dmax = atoi(row[1]);
59    arrivalr = atoi(row[2]);
60    printf(" %s, %s\n", (row[0]), (row[1]));
```

```
61      printf("Pmax = %d, Dmax = %d, arrivalrate = %d \n", ↩
            Pmax, Dmax, arrivalr);
62      if((Tdisk/Pmax) <= Consumetime ){
63       P = Pmax;
64       D = Dmax;
65      break;
66      }
67    }
68    if (P == 0){
69      printf("Error: Could not support, hence could not ↩
            find the value of P and D for arrivaltime = %d and↩
             consume time = %d\n", Darrivaltime, Consumetime);
70      return -1;
71    }
72    cout<<"Calculated Consume time = "<<(Tdisk/P)<<", ↩
        Consume time="<<Consumetime<<endl;
73    cout<<"The value of Max P =  "<<P<<" The value of ↩
        Corresponding D = "<<D<<", Pmax * Consumetime = "<<(↩
        P * Consumetime)<<endl;
74    return 0;
75  }
```

Listing C.4: Simulation Code Representing NMS: Makefile

```
1  BINDIR=/usr/local/bin
2  OBJS = Request.o
```

```
3  CC = g++
4  CFLAGS = -Wall -O -g
5  DFLAGS1 = -I/usr/include/mysql
6  DFLAGS2 = /usr/lib/mysql/libmysqlclient.so
7
8  Request.o : Request.h simulation_PonD.cpp \$(DFLAGS2)
9    clear
10   \$(CC) -o myprog \$(CFLAGS) \$(DFLAGS1) \$(DFLAGS2) ↩
       simulation_PonD.cpp
11 clean:
12   rm *.o myprog
```