A Major Project Report on

A Novel Approach of Asynchronous Cache Prefetching for Storage Volume Controller

Submitted in partial fulfillment of the

Requirements for the award of the degree

of

MASTER OF ENGINEERING

in

COMPUTER TECHNOLOGY & APPLICATIONS

By

Vaibhav Verma

(16/CTA/09)

(University roll no - 8555)

Under the Guidance of

Dr. Rajni Jindal

DEPARTMENT OF COMPUTER ENGINEERING
DELHI COLLEGE OF ENGINEERING
BAWANA ROAD, DELHI-110042
DELHI UNIVERSITY

Certificate

This is to certify that the major project entitled **"A Novel Approach of Asynchronous Cache Prefetching for a Storage Volume Controller "** is the work of Vaibhav Verma(university roll no - 8555) , a student of Delhi College of Engineering. This work was completed under my direct supervision and guidance and forms a part of Master of Engineering (Computer Technology & Applications) course and curriculum. He has completed his work with utmost sincerity and diligence.

(Dr. Rajni Jindal)
Associate Professor and Project Guide
Department of Computer Engineering
Delhi College of Engineering

Acknowledgement

It gives me a great pleasure to express my profound gratitude to my project guide **Dr. Rajni Jindal,** Associate professor**,** Department of Computer Engineering, Delhi College of Engineering, for her valuable and inspiring guidance throughout the progress of this project. At the same time, I would like to extend my heartfelt thanks to **Dr. Daya Gupta,** Head of the department, Department of Computer Engineering, Delhi College of Engineering, for keeping the spirits high and clearing the visions to work on the project.

Also I would like to thank **Mr. Praveen Padia,** Tech lead**,** India Storage Labs, IBM, for his constant support and encouragement. I would like to thank **Mr**. **Anuj Chandra,** Peoples manager, India Storage Labs, IBM, for making all the resources available and providing healthy environment for the successful completion of the project.

Vaibhav Verma.
(16/CTA/09)

TABLE OF CONTENTS

Certificate	ii
Acknowledgement	iii
Table Of Contents	iv
List Of Figures	v
List Of Tables	vi
Abstract	1
1. Introduction	3
1.1. Objective	3
1.2. Problem Statement	4
1.3. Motivating Factor	4
1.4. Organization Of The Dissertation	6
2. Storage Virtualization	8
2.1. The Need For Storage Virtualization	9
2.2. IBM's San Volume Controller	11
3. Prestaging Strategies	27
3.1. Where Is Prefetching Applied	28
3.2. When Is Prefetching Useful	29
3.3. What To Prefetch	30
3.4. The Problem Of Cache Pollution	32
3.5. The Problem Of Wasted Prefetches	33
3.6. Similar Work	34
4. Adaptive Asynchronous Algorithm (AAA)	39
4.1. Prestaging Strategy	39
4.2. A Basic Prestaging Algorithm	40
4.3. Asynchronous Prefetching Algorithm	41
4.4. Challenges	42
4.5. Proposed AAA(Adaptive Asynchronous Algorithm)	43
5. Simulation Of AMP And AAA(Adaptive Asynchronous Algorithm)	50
5.1. Cache Simulator	52
6. Comparative Study	55
6.1. Comparison Between IBM's SVC Algorithm And AAA (Proposed Algorithm)	56
6.2. Comparison Between AMP And AAA(Proposed Algorithm)	57
7. Conclusion And Future Work	61
7.1. Conclusion	62
7.2. Future Work	62
8. References	63

List Of Figures

Figure 2.1	Extents being used to create a virtual disk [IBM's SVC hand book]	19
Figure 2.1	The relationship between physical and virtual disks [IBM's SVC hand book]	20
Figure 2.3	Vdisk and Vdisk copy [IBM's SVC hand book]	22
Figure 2.4	SAN Volume Controller logical view [IBM's SVC hand book]	24
Figure 4.1	A general asynch Prestage algorithm	39
Figure 4.2	Two stages of AAA.	43
Figure 4.3	Data prestage pattern in ramp up stage	44
Figure 5.1	Basic BUI for cache simulator	52
Figure 5.2	GUI for I/O patter selection	53
Figure 5.3	GUI for log section	53
Figure 6.1	Plot showing number of miss for sequential I/O patterns	60
Figure 6.2	Plot showing number of miss for semi-random I/O patterns	61

List Of Tables

Table 6.1	Results of Sequential pattern for AMP and AAA	59
Table 6.2	Results of Semi Random pattern for AMP and AAA	60

Abstract

Cache prefetching is a commonly used technique in which the data is prefetched from the disk to cache in advance, before host actually requests it. These days this technique is used in almost all modern storage volume controllers. One of the widely popular classes of prefetching algorithms is sequential prefetching. But there are two problems with these state-of-the-art sequential prefetching algorithms: (i) cache pollution, which occurs when a prefetched data replaces some more useful prefetched or demand-paged data, and (ii) prefetch wastage, which occurs when prefetched data is evicted by LRU (maintained in the cache) from the cache before it can be used.

A sequential prefetching algorithm [BINNY S. GILL at al 2007] can have a fixed (static) or adaptive (dynamic) degree of prefetch and can either have synchronous (when it can prefetch only on amiss) or asynchronous (when it can also prefetch on a hit) way of prefetching. To capture these distinctions there are four classes of prefetching algorithms [BINNY S. GILL at al 2007]: fixed synchronous (FS), fixed asynchronous (FA), adaptive synchronous (AS), and adaptive asynchronous (AA). After exploring all these sets algorithms, their advantages and disadvantages we found that the relatively unexplored class of AA algorithms is in fact the most promising type of algorithms for sequential prefetching.

We studied the cache prefetch algorithm presently used in IBM's SAN volume controller and AMP algorithm [BINNY S. GILL at al 2007]. We also analyzed the basic problems and deficiencies of these two algorithms. In this thesis we will discuss the basic aspect of cache prefetching, and the challenges for a prefetch algorithm in a volume controller. On the basis of knowledge gained by the analysis of above two algorithms we proposed a new algorithm that has capabilities to tackle most of the problems in the discussed

algorithm. Also we implemented AMP algorithm and our proposed algorithm to compare the effectiveness of the two algorithms. We prepared a cache simulator which simulates a cache and all its basic features like staging and destaging of data from disk. At last we compared the two algorithms on the basis of results we got from the simulation of two algorithms.

Chapter: 1 Introduction

1.1 Objective

The objective of this thesis is to design a new prefetching algorithm that can work effectively in a storage volume controller and have the capability to handle random as well as semi random I/O patters in multi threaded environment. It should also have the capability to reduce the average life time of a data page in cache.

Currently there are many synchronous types of prefetching algorithms being used in most of the volume controllers. These algorithms prefetches some extra amount of data at every miss occurs on user's request. They are simple to implement but have some disadvantages. The relatively unexplored types of algorithms are adaptive asynchronous prefetching algorithms. These algorithms prefetches the data on cache hits also (defining some event for prefetching).

We studied two very efficient algorithms, one used in IBM's SAN volume controller which is an adaptive synchronous type algorithm and AMP algorithm which is adaptive asynchronous type of algorithm.

The objective of this research work is to find out the advantages and disadvantages of both class of prefetching algorithms and to come up with more effective way of prefetching. This research work is carried in IBM, India Storage labs and the conclusions of the work will be used in further research at IBM.

1.2 Problem Statement

We want to design an efficient Prefetching algorithm that works well in multi-threaded, sequential, semi-random as well as random request patterns. The algorithm should be able to reduce the average life time of a data page in cache, it should have high hit rate, and it should also works well in changing host I/O size and rates.

The desired algorithm must be generic and can be customized for any storage virtualization layer of volume controller. It must also be adaptable with the changing needs and requirements of host.

1.3 Motivating Factor

In computer engineering, cache is a component that transparently stores the data so that future requests for that data can be served faster. The data that is stored within a cache might be values that have been computed earlier or duplicates of original values that are stored elsewhere.

If requested data is contained in the cache (cache hit), this request can be served by simply reading the cache, which is comparatively faster. Otherwise (cache miss), the data has to be recomputed or fetched from its original storage location, which is comparatively slower. Hence, the more amounts of requests served directly from cache make the system performance better.

But only demand paging cannot solves the purpose. There is always a need to understand user’s request patterns and prefetch the data accordingly in advance. The technique of prefetching was first used in mid-60’s when multiple words were prefetched in processors in the form of a cache line. Soon it was realized that increasing the size of the cache line can decrease performance due to false sharing.

So, numerous hardware-initiated prefetching techniques were introduced in both uniprocessor and multiprocessor architectures. Subsequently, software-initiated methods for prefetching were introduced where applications disclosed access patterns to the hardware, or controlled prefetching directly.

To reduce read latency, there must be an efficient prefetching algorithm. These days prefetching is used everywhere in almost all the applications where data is being accessed from a relatively slower media. This makes the field of prefetching techniques a very hot and potential area for research and to develop new ways to prefetch that works in current demanding environment.

So in this thesis we tried to propose a relatively unexplored class of prefetching (AA type) and we find it very challenging to design an algorithm that can tackle all the

challenges for a good prefetch algorithm.

1.4 Organization of the Dissertation

This thesis work is organized as follows

Chapter 1 deals with providing the objective, problem statement, motivation of undertaking this research work as well as organization of this dissertation.

Chapter 2 deals with the concept of storage virtualization. It also provides a basic knowledge of “compass architecture “of IBM’s SAN volume controller. This helps us in better understanding of needs and requirement of a volume controller.

Chapter 3 provides introduction to prefetching and various classes of prefetching strategies. It also provides introduction to prefetching algorithm used in IBM’s SVC and AMP algorithm.

Chapter 4 begins with description of our research work. It describes the basic design of an asynchronous prefetching algorithm. It also gives list of challenges for a good prefetching algorithm. Based on that it describes our proposed algorithm and it gives the detailed view on how the prefetch is done in ramp up stage and asynch stage.

Chapter 5 gives the basic overview of the cache simulator that we implemented for the simulation of AMP algorithm and our proposed algorithm. It also gives the basic over of the user interface offered by the simulator.

Chapter 6 provides the comparison among the two reference algorithm with our proposed algorithm discussed in chapter 4. It tells about the basic problems with the two referenced algorithms and how those problems are tackled in the proposed algorithm.

Chapter 7 gives the final findings and outcomes of the research. It lists the problems that we solved and those that still remain to be tackled. It also lays the ground to the future work in this direction.

Chapter: 2

Storage Virtualization

Storage virtualization is a concept and term used within computer science. Specifically, storage systems may use virtualization concepts as a tool to enable better functionality and more advanced features within the storage system [STORAGE VIRTUALIZATION Wikipedia].

Broadly speaking, a 'storage system' is also known as a storage array or Disk array or a filer. Storage systems typically utilize specialized hardware and software along with disk drives in order to provide very fast and reliable storage for computing and data processing.

Storage systems are complex, and may be thought of as a special purpose computer designed to provide storage capacity along with advanced data protection features. Disk drives are only one element within a storage system, along with hardware and special purpose embedded software within the system.

Storage systems can provide either block accessed storage, or file accessed storage. Block access is typically delivered over Fiber Channel, iSCSI, SAS, FICON or other protocols. File access is often provided using NFS or CIFS protocols.

2.1 The need for storage virtualization

At the business level, clients are faced with three major storage challenges [BINNY S. GILL at al 2007]:

- Managing storage growth: Storage needs continue to grow at a rate that is normally higher than what has been planned for each year. As an example, storage subsystems can be purchased to last for 3 to 5 years; however, organizations are finding that they are filling to capacity much earlier than that.

To fill the growth, customers are then either extending their current storage subsystems in chunks, or buying different types of storage subsystems to match their storage needs and budget.

- Increasing complexity: As storage needs grow, this need can be filled by more than one disk subsystem, which might not even be from the same vendor. Together with the variety of server platforms and operating systems in a customer's environment, customers can have storage area networks (SAN) with multiple and diverse storage subsystems and host platforms. Combining this with the shortage of skilled storage administrators, the cost and risk of storage increases as the environment becomes more complex.

- Maintaining availability: With the increased range of storage options available, the storage growth rate, and no similar increase in storage budget, customers have to manage more storage with minimal or no additional staff. Thus, with the complexity highlighted above, and with business requirements on IT resources demanding higher business system availability, the room for errors increases as each new storage subsystem is added to the infrastructure. Additionally, making changes to the storage infrastructure to accommodate storage growth traditionally leads to outages that might not be acceptable by the business.

Storage needs are rising, and the challenge of managing disparate storage systems is growing. The IBM System Storage SAN Volume Controller brings storage devices together in a *virtual pool* to make all storage appear as:

- One “logical” device to centrally manage and to allocate capacity as needed.

- One solution to help achieve the most effective use of key storage resources on demand.

Virtualization solutions can be implemented in the storage network, in the server, or in the storage device itself. The IBM storage virtualization solution is SAN-based, which helps allow for a more open virtualization implementation. Locating virtualization in the SAN, and therefore in the path of input/output (I/O) activity, helps provide a solid basis for policy-based management. The focus of IBM on open standards means its virtualization solution supports freedom of choice in storage-device vendor selection.

The IBM System Storage SAN Volume Controller solution is designed to:

- Simplify storage management
- Reduce IT data storage complexity and costs while enhancing scalability
- Extend on demand flexibility and resiliency to the IT infrastructure
- Increase application availability by making changes in the infrastructure without having to shut down hosts.

2.2 IBM SAN Volume Controller

In this part, we describe the major concepts behind the IBM System Storage SAN Volume Controller to provide the framework for the discussion.

2.2.1Virtualization overview

The SVC nodes are the hardware elements of the IBM System Storage SAN Volume Controller, a member of the IBM System Storage virtualization family of solutions. The SAN Volume Controller combines servers into a high availability cluster. Each of the servers in the cluster is populated with 8 GB of high-speed memory, which serves as the cluster cache. A management card is installed in each server to monitor various parameters that the cluster uses to determine the optimum and continuous data path. The cluster is protected against data loss by uninterruptible power supplies.

The SAN Volume Controller nodes can only be installed in pairs to avoid a single point of failure. Storage virtualization addresses the increasing cost and complexity in data storage management. It addresses this increased complexity by shifting storage

management intelligence from individual SAN disk subsystem controllers into the network through a virtualization cluster of nodes.

The SAN Volume Controller solution is designed to reduce both the complexity and costs of managing your SAN-based storage. With the SAN Volume Controller, you can:

- Simplify management and increase administrator productivity by consolidating storage management intelligence from disparate disk subsystem controllers into a single view.

- Improve application availability by enabling data migration between disparate disk storage devices non-disruptively.

- Improve disaster recovery and business continuance needs by applying and managing copy services across disparate disk storage devices within the Storage Area Network (SAN). These solutions include a Common Information Model (CIM) Agent, enabling unified storage management based on open standards for units that comply with CIM Agent standards.

- Provide advanced features and functions to the entire SAN, such as:
    - Large scalable cache
    - Copy Services
    - Space management (later releases to include Policy Based Management)
    - Mapping based on desired performance characteristics
    - Quality of Service (QoS) metering and reporting

- Simplify device driver configuration on hosts, so all hosts within your network use the same IBM device driver to access all storage subsystems through the SAN Volume Controller.

2.2.2Compass architecture

The IBM System Storage SAN Volume Controller is based on the *Com*modity *Pa*rts *S*torage *S*ystem (Compass) architecture developed at the IBM Almaden Research Center. The overall goal of the Compass architecture is to create storage subsystem software applications that require minimal porting effort to leverage a new hardware platform. To meet this goal:

- Compass, although currently deployed on the Intel® hardware platform, can be ported to other hardware platforms.

- Compass, although currently deployed on a Linux kernel, can be ported to other Portable Operating System Interface (POSIX)-compliant operating systems.

- Compass uses commodity adapters and parts wherever possible. To the highest extent possible, it only uses functions in the commodity hardware that are commonly exercised by the other users of the parts. This is not to say that Compass software could not be ported to a platform with specialized adapters.

- However, the advantage in specialized function must be weighed against the disadvantage of future difficulty in porting and in linking special hardware development

plans to the release plans for applications based on the Compass architecture.

- Compass is developed in such a way that it is as easy as possible to troubleshoot and correct software defects.

- Compass is designed as a scalable, distributed software application that can run in increasing sets of Compass nodes with near linear gain in performance while using a shared data model that provides a single pool of storage for all nodes.

- Compass is designed so that there is a single configuration and management view of the entire environment regardless of the number of Compass nodes in use.

The approach is to minimize the dependency on unique hardware, and to allow exploitation of or migration to new SAN interfaces simply by plugging in new commodity adapters. Performance growth over time is ensured by the ability to port Compass to just about any platform and remain current with the latest processor and chipset technologies on each.

The SAN Volume Controller implementation of the Compass architecture has exploited Linux as a convenient development platform to deploy this function. This has enhanced and will continue to enhance the ability of IBM to deploy robust function in a timely way.

SVC relies on the Compass architecture to provide high levels of fault tolerance and high availability. Extensive dump capabilities are provided to enable first failure capture of software defects.

Fault tolerance and high levels of availability are achieved by:

- The RAID capabilities of the underlying disk subsystems
- SVC clustering using the Compass architecture
- Auto-restart of hung nodes
- UPS units to provide memory protection in the event of a site power failure
- Host System Failover capabilities

High levels of serviceability are achieved by providing:

- Cluster error logging
- Asynchronous error notification
- Dump capabilities to capture software detected failures
- Concurrent diagnostics
- Directed maintenance procedures
- Concurrent log analysis and dump data recovery tools
- Concurrent maintenance of all SVC components
- Concurrent upgrade of SVC software and microcode
- Concurrent addition or deletion of SVC nodes in a cluster
- Software recovery through a service panel push button
- Automatic software version correction when replacing a node
- Detailed status and error conditions displayed on the service panel
- Error and event notification through SNMP and e-mail

Support is provided for the end-to-end SAN problem determination through detailed fabric status reporting capabilities.

SAN Volume Controller clustering

In simple terms, a cluster is a collection of servers that, together, provide a set of resources to a client. The key point is that the client has no knowledge of the underlying physical hardware of the cluster.

This means that the client is isolated and protected from changes to the physical hardware, which brings a number of benefits. Perhaps the most important of these benefits is high availability.

Resources on clustered servers act as highly available versions of unclustered resources. If a node (an individual computer) in the cluster is unavailable, or too busy to respond to a request for a resource, the request is transparently passed to another node capable of processing it, so clients are unaware of the exact locations of the resources they are using.

For example, a client can request the use of an application without being concerned about either where the application resides or which physical server is processing the request. The user simply gains access to the application in a timely and reliable manner. Another benefit is scalability: If you need to add users or applications to your system and want performance to be maintained at existing levels, additional systems can be incorporated into the cluster.

The IBM System Storage SAN Volume Controller is a collection of up to eight cluster nodes, added in pairs. In future releases, the cluster size will be increased to permit further performance scalability. These nodes are managed as a set (cluster) and

present a single point of control to the administrator for configuration and service activity.

Note: Although the SAN Volume Controller code is based on a Linux kernel, the clustering feature is not based on Linux clustering code. The clustering failover and failback feature is part of the SAN Volume Controller application software.

Within each cluster, one node is defined as the configuration node. This node is assigned the cluster IP address and is responsible for transitioning additional nodes into the cluster.

During normal operation of the cluster, the nodes communicate with each other. If a node is idle for a few seconds, then a heartbeat signal is sent to ensure connectivity with the cluster. Should a node fail for any reason, the workload intended for it is taken over by another node until the failed node has been restarted and re-admitted to the cluster (which happens automatically). In the event that the microcode on a node becomes corrupted, resulting in a failure, the workload is transferred to another node. The code on the failed node is repaired, and the node is re-admitted to the cluster (again, all automatically).

For I/O purposes, SAN Volume Controller nodes within the cluster are grouped into pairs, called *I/O groups*, with a single pair being responsible for serving I/O on a given VDisk. One node within the I/O group represents the preferred path for I/O to a given VDisk. The other node represents the non-preferred path. This preference alternates between nodes as each VDisk is created within an I/O group to balance the workload evenly between the two nodes.

Note: The preferred node by no means signifies absolute ownership. The data can still be accessed by the partner node in the I/O group in the event of a failure.

Beyond automatic configuration and cluster administration, the data transmitted from attached application servers is also treated in the most reliable manner. When data is written by the host, the preferred node within the I/O group stores a write in its own write cache and the write cache of its partner (non-preferred) node before sending an “I/O complete” status back to the host application. The write cache is automatically destaged to disk after two minutes of no writes to a VDisk. To ensure that data is written in the event of a node failure, the surviving node empties all of its remaining write cache and proceeds in write-through mode until the cluster is returned to a fully operational state.

Note: Write-through mode is where the data is not cached in the nodes, but written directly to the disk subsystem instead. While operating in this mode, performance can be degraded. More importantly, it ensures that the data makes it to its destination without the risk of data loss. A single copy of data in cache would constitute exposure to data loss.

Another data protection feature that the SAN Volume Controller has is uninterruptible power supply units. In addition to voltage regulation to protect valuable electronic components within the SAN Volume Controller configuration, in the event of a main power outage, the uninterruptible power supply provides enough power to destage data to the SAN Volume Controller internal disk and shut down the nodes within the SAN Volume Controller cluster gracefully. This is a feature found in most high-end disk subsystems.

SAN Volume Controller virtualization

The SAN Volume Controller provides block aggregation and volume management for disk storage within the SAN. In simpler terms, this means that the SAN Volume Controller manages a number of back-end disk subsystem controllers and maps the physical storage within those controllers to logical disk images that can be seen by application servers and workstations in the SAN.

The SAN must be zoned in such a way that the application servers cannot see the same back-end LUNs seen by the SAN Volume Controller, preventing any possible conflict between the SAN Volume Controller and the application servers both trying to manage the same back-end LUNs.

As described earlier, when an application server performs I/O to a VDisk assigned to it by the SAN Volume Controller, it can access that VDisk through either of the nodes in the I/O group. Each node can only be in one I/O group, and since each I/O group only has two nodes, the distributed redundant cache design in the SAN Volume Controller only needs to be two-way.

The SAN Volume Controller I/O groups are connected to the SAN in such a way that all back-end storage and all application servers are visible to all of the I/O groups. The SAN Volume Controller I/O groups see the storage presented to the SAN by the back-end controllers as a number of disks, known as *managed disks*.

Because the SAN Volume Controller does not attempt to provide recovery from physical disk failures within the back-end controllers, MDisks are recommended, but not

necessarily required, for a RAID array. The application servers must not see the MDisks at all. Instead, they should see a number of logical disks, known as virtual disks or VDisks, which are presented to the SAN by the SAN Volume Controller.

MDisks are collected into groups, known as *managed disk groups (MDGs)*. The MDisks that are used in the creation of a particular VDisk must all come from the same MDG. Each MDisk is divided into a number of extents. The minimum extent size is 16 MB, and the maximum extent size is 2048 MB, based on the definition of its MDG. These extents are numbered sequentially from the start to the end of each MDisk. Conceptually, this is represented as shown in Figure 2-1.

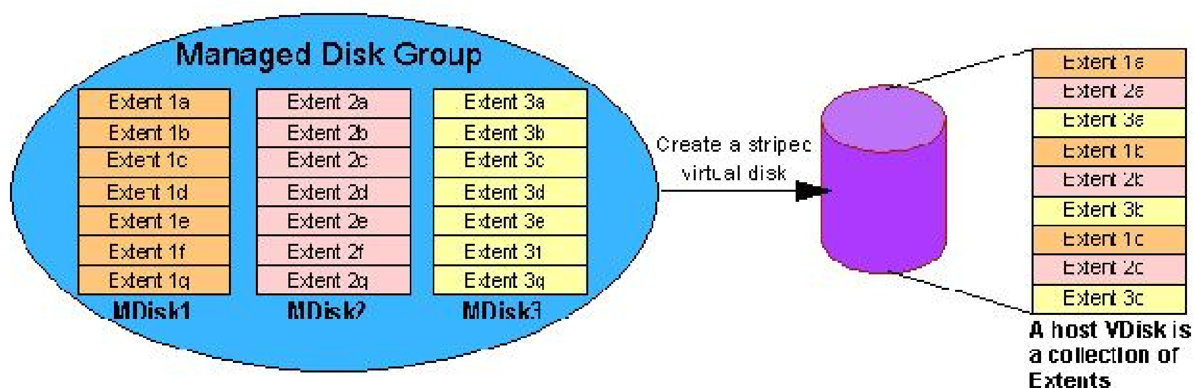

Figure 2-1 Extents being used to create a virtual disk [IBM's SVC hand book]

The virtualization function in the SAN Volume Controller maps the VDisks seen by the application servers to the MDisks presented by the back-end controllers. I/O traffic for a particular VDisk is, at any time, handled exclusively by the nodes of a single I/O group.

Although a cluster can have several pairs of nodes, the nodes handle I/O in independent pairs. This means that the I/O capability of the SAN Volume Controller scales well (almost linearly), since additional throughput can be obtained by simply adding additional I/O groups.

Figure 2-2 summarizes the various relationships that bridge the physical disks through to the virtual disks within the SAN Volume Controller architecture.

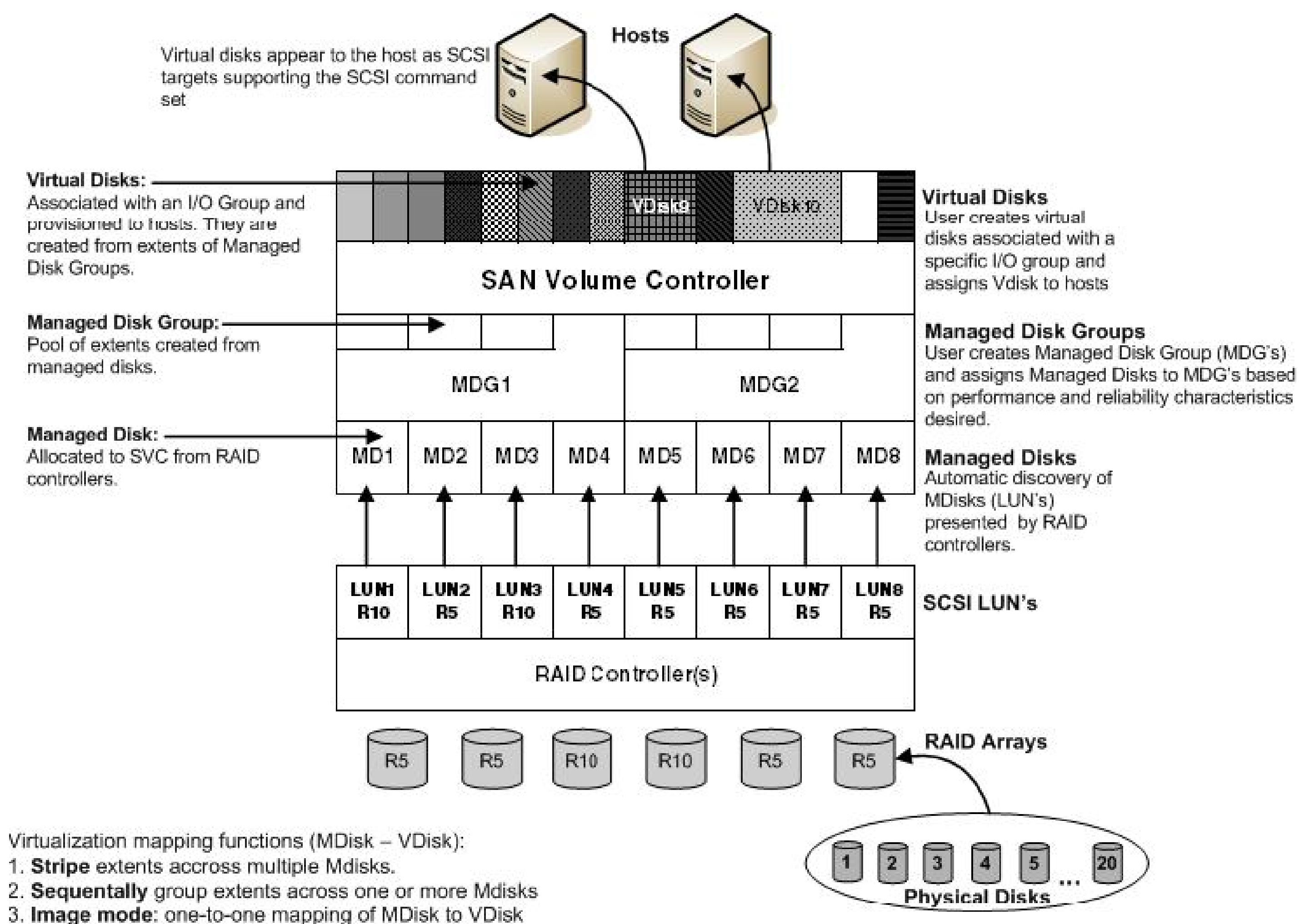

Figure 2-2 the relationship between physical and virtual disks [IBM's SVC hand book]

Virtualization mappings

Several different mapping functions are provided by the SAN Volume Controller:

- Striped: Here a VDisk is mapped to a number of MDisks in an MDG. The extents on the VDisk are striped over the MDisks. Therefore, if the VDisk is mapped to five MDisks, then the first, sixth, eleventh (and so on) extents come from the first MDisk, the second, seventh, twelfth (and so on) extents come from the second MDisk, and so on. This is the default mapping.

- Sequential: Here a VDisk is mapped to a single MDisk in an MDG. There is no guarantee that sequential extents on the MDisk map to sequential extents on the VDisk, although this might be the case when the VDisk is created.

- Image: Image mode sets up a one-to-one mapping of extents on an MDisk to the extents on the VDisk. Because the VDisk has exactly the same extent mapping as the underlying MDisk, any data already on the disk is still accessible when migrated to a SAN Volume Controller environment.

- Within the SAN Volume Controller environment, the data can (optionally) be seamlessly migrated off the image mode VDisk to a striped or sequential VDisk within the same or another MDG.

Virtual Disk Copy

Every VDisk is associated to at least one VDisk Copy. The VDisk itself is a *logical* entity whereas the VDisk Copy is a *physical* entity. The VDisk Copy represents the physical capacity occupied by the VDisk on the MDisks.

A second copy can be created as a VDisk Copy Mirror, as shown in Figure 2-3.

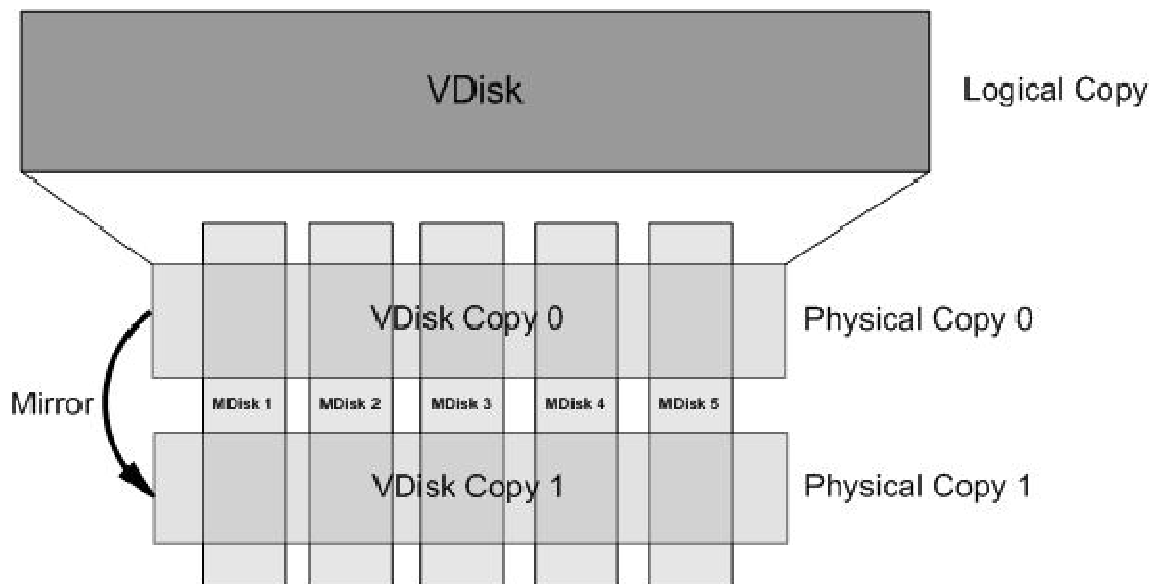

Figure 2-3 VDisk and VDisk copy [IBM's SVC hand book]

SAN Volume Controller multipathing

Each SAN Volume Controller node presents a VDisk to the SAN through multiple paths. We recommend that a VDisk be seen in the SAN by *four* paths. In normal operation, two nodes provide redundant paths to the same storage. This means that, depending on zoning and SAN architecture, a single host might see *eight* paths, to each LUN presented by the SAN Volume Controller. Because most operating systems cannot resolve multiple paths back to a single physical device, IBM provides a multipathing device driver.

The multipathing driver supported by the SAN Volume Controller is the IBM Subsystem Device Driver (SDD).

SDD groups all available paths to a virtual disk device and presents it to the operating system. SDD performs all the path handling and selects the active I/O path(s).

Note: There are no ordering requirements in the MDisk to VDisk extent mapping function for either striped or sequential VDisks. This means that if you examine the extents on an MDisk, it is quite possible for adjacent extents to be mapped to different VDisks.

It is also quite possible for contiguous extents on the MDisk to be mapped to widely separated extents on the same VDisk, or to close-by extents on the VDisk. In addition, the position of the extents on the MDisks is not fixed by the initial mapping, and can be varied by the user performing data migration operations.

2.2.3 SAN Volume Controller logical configuration

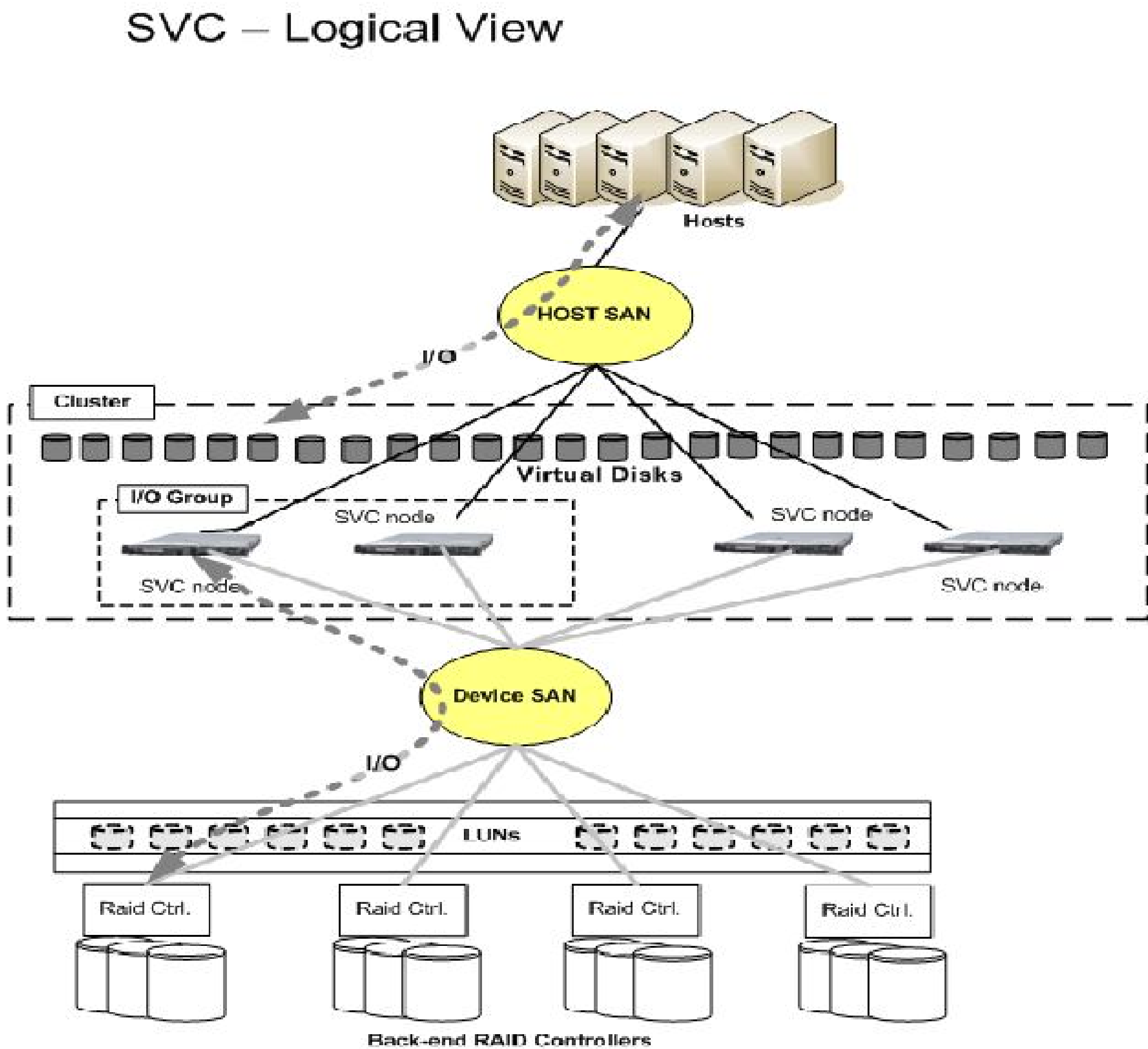

Figure 2-4 SAN Volume Controller logical view [IBM's SVC hand book]

Cluster

- There is a *maximum of four I/O groups* (node pairs) per cluster.
- The performance of a cluster increases almost linearly with additional I/O groups.

I/O Group

- Because there is a *minimum of two nodes per I/O group*, there is no single point failure. Even if one node breaks away, data in the cache is not lost because the cache is mirrored between the two nodes.
- Every virtual disk is assigned to a *single* I/O group.
- For load balancing, a virtual disk is owned by *alternating* nodes in an I/O Group.

Chapter: 3

Pre-staging strategies

Over the last several decades, we have witnessed remarkable improvements in the information processing capabilities of computing systems. A large number of data storage technologies have also been developed with diverse speeds, capacities, reliability, and affordability characteristics.

We often find that cost considerations force us to design systems with a data storage component which runs significantly slower than the processing unit. To bridge this gap between data supplier and data consumer, faster data caches are placed between the two.

Since caches are expensive, they can typically keep only a subset of the entire dataset. Consequently, it is extremely important to manage the cache wisely in order to maximize its performance. The cornerstone of read cache management is to keep recently requested data in the cache in the hope that such data will be requested again in the near future.

Data is placed in the cache only when requested by the consumer (demand paging). Another, and rather competing, method is to fetch into the cache data that is predicted to be requested in the near future (prefetching).

3.1 Where is Prefetching Applied?

The technique of prefetching dates as far back as the mid-60's when multiple words were prefetched in processors in the form of a cache line. It was soon realized that increasing the size of the cache line can decrease performance due to false sharing. So, numerous hardware-initiated prefetching techniques were introduced in both uniprocessor and multiprocessor architectures [Rogers and Li 1992; David Callahan and Porterfield 1991; Metcalf 1993].

Subsequently, software-initiated methods for prefetching were introduced where applications disclosed access patterns to the hardware, or controlled prefetching directly [Patterson et al. 1995; Mowry and Gupta 1991]. For other applications, compiler techniques were used to predict access patterns and insert fetch requests in the compiled executables [Gornish et al. 1990; Chen and Baer 1992].

ACM Transactions on Storage, Vol. 3, No. 3, Article 10, Publication date: October 2007. Optimal Multistream Sequential Prefetching in a Shared Cache • 10:3 Compiler-assisted prefetching was also extended for pointer-based accesses [Luk and Mowry 1996; Roth et al. 1998; Lipasti et al. 1995].

Today prefetching is ubiquitously applied in web servers and clients [Kroeger et al. 1997], databases [Curewitz et al. 1993], file servers [Griffioen and Appleton 1994; Kotz and Ellis 1991], on-disk caches, and multimedia servers.

3.2 When is Prefetching Useful?

The goal of prefetching is to make data available in the cache before the data consumer places its request, thereby masking the latency of the slower data source below the cache. However, prefetching is not without cost. It requires:

(i) cache space to keep the prefetched data;
(ii) network bandwidth to transfer the data to the cache;
(iii) data source bandwidth to read the data; and
(iv) Processing power to carry out the prefetch.

If the prefetched data is not subsequently used by the data consumer, the extra cost of prefetching normally reduces performance. Only in over provisioned systems can prefetching with low predictive accuracy improve performance. However, the data cache is obviously under provisioned, as it can keep only a subset of the dataset. The prefetched data typically shares the cache space with demand-paged data.

Therefore, the utility of the prefetched data should not be lower than that of the demand-paged data it replaces. To maximize performance, the marginal utility of both kinds of data should be equalized [Gill and Modha 2005a]. Since the utility of prefetched data that is not subsequently used is zero, it is extremely important to prefetch judiciously,

keeping the number of wasted prefetches to a minimum. Furthermore, any prefetching algorithm needs to be able to predict accesses sufficiently in advance to allow for the time it takes to prefetch the data. As a rule of thumb, prefetching is useful when the long-term prediction accuracy of access patterns is high.

3.3 What to Prefetch?

The most common prefetching approach is to perform sequential readahead. The simplest form is one block lookahead (OBL), where we prefetch one block beyond the requested block. OBL can be of three types:

(i) always prefetch—prefetch the next block on each reference;
(ii) prefetch on miss—prefetch the next block only on a miss; and
(iii) Tagged prefetch—prefetch the next block only if the referenced block is accessed for the first time.

P-block lookahead extends the idea ofOBLby prefetching P blocks instead of one, where P is also referred to as the degree of prefetch. Dahlgren et al. [1993] proposed a version of the P-block lookahead algorithm which dynamically adapts the degree of prefetch for the workload. Tcheun et al. [1997] suggested a per-stream scheme which selects the appropriate degree of prefetch on each miss based on a prefetch degree selector (PDS) table. For the case where cache is abundant, infinite-block lookahead has also been studied.

Stride-based prefetching has also been studied, mainly for processor caches where strides are detected based on information provided by the application [Fu and Patel

1991], a lookahead into the instruction stream [Lee et al. 1987], or a reference prediction table indexed by the program counter [Chen and Baer 1995]. Dahlgren and Stenstr¨om [1996] found that sequential prefetching is a better choice because most strides lie within the block size and it can also exploit locality.

History-based prefetching has been proposed in various forms. Grimsrud et al. [1993] used a history-based table to predict the next pages to prefetch. Prefetching using Markov predictors has been studied in Joseph and Grunwald [1999], wherein multiple memory predictions are prefetched at the same time. Data compression techniques have also been applied to predict future access patterns [Curewitz et al. 1993]. Vitter and Krishnan [1996] provided an optimal (in terms of miss ratio) prefetching technique based on the Lempel-Ziv algorithm. Lei and Duchamp [1997] suggested a file prefetching technique based on historical access correlations maintained in the form of access trees.

Offline algorithms, although primarily of theoretical interest, have also been studied. Cao et al. [1995] were able to show that a simple and natural offline algorithm called aggressive, which prefetches as early as is reasonable, has near-optimal performance for a single disk. The interaction between caching and prefetching is significantly more complicated in a system with multiple disks because a set of blocks can then be prefetched in parallel if they reside on different disks. Kimbrel and Karlin [1996] provided a near-optimal offline algorithm in the multiple disk case and compared various offline algorithms [Kimbrel et al. 1996].

They used a fixed cost for prefetching and assumed that disks can handle only one prefetch request at-a-time. Furthermore, they did not consider the case of sequential prefetching, where multiple pages in the same sequence can be prefetched with

relatively small added cost. The fact is that most commercial data storage systems use very simple online prefetching schemes like sequential prefetching. This is because only sequential prefetching can achieve high long-term predictive accuracy in data servers.

Strides that cross-page or track boundaries are uncommon in workloads and therefore not worth implementing. History-based prefetching suffers from low predictive accuracy and the associated cost of the extra reads on an already bottlenecked I/O system. The data storage system cannot use most hardware- or software-initiated prefetching techniques, as the applications typically run on external hardware. Furthermore, offline algorithms [Cao et al. 1995; Kallahalla and Varman 2002; Kimbrel and Karlin 1996; Kimbrel et al. 1996] are not applicable, as they require knowledge of future data accesses.

3.4 The Problem of Cache Pollution

In the context of prefetching, cache pollution is said to occur when prefetching replaces or keeps out more useful data from the cache. This can happen when prefetched data replaces demand, paged data with high temporal locality, or when prefetching is done too aggressively for some workloads while others continue to incur misses.

There have been attempts to reduce cache pollution by restricting the amount of cache that the prefetched data can occupy [Reungsang et al. 2001], or via software hints [Jain et al. 2001].

The SARC algorithm [Gill and Modha 2005a] provides an adaptive and autonomous solution to limit this problem by allocating cache space so as to equalize the marginal ACM Transactions on Storage, Vol. 3, No. 3, Article 10, Publication date: October 2007.

Optimal Multistream Sequential Prefetching in a Shared Cache • 10:5 utility of demand-paged and prefetched data. However, we are not aware of any prior online solution for minimizing cache pollution that occurs when less useful data is prefetched in favor of the more useful.

3.5 The Problem of Wasted Prefetches

In data storage systems, the disks are typically the bottleneck. If pages are prefetched speculatively and not subsequently used, then not only does this cause cache pollution and an increase in the backend bandwidth usage, but, importantly, it causes additional I/O load on the disks.

This additional load can lead to degradation in performance, defeating the purpose of prefetching. This is the reason why most history-based prefetching schemes which do not have high prediction accuracy are not used in commercial systems.

3.6 Similar Work

In this section we will first discuses about cache prefetching in SVC and then we discuses about AMP algorithm [BINNY S. GILL at al 2007].

3.6.1 Prefetching in SVC

Prefetching algorithm used in SVC is a synchronous type of algorithm which tries to evaluate prefetching event at every I/O from the user. Let us first find out how it works.

In SVC prefetching algorithm they uses multiple decision making variables, The concepts of heat, effectiveness, and weight, as defined below, are used to make judgments on what is frequent and what is sufficient.

• On a read miss, Cache always stages data to the end of the segment that includes the last logical block requested by the read command. The read ahead algorithms below are independent of this staging policy.

• Cache will only read ahead following an I/O client read command and will read blocks that immediately follow those requested by the prompting I/O client read command.

• Read commands that are serviced from the cache (i.e. a read hit), do not prompt cache to read ahead unless the read command reads data that was prefetched because of the read ahead strategy.

• Read ahead is based on locality of reference so each virtual disk is divided into regions --currently 128Mb-- and statistics are kept for each extent. These statistics are calculated at the beginning of any read ahead operation.

• Heat, which is defined as delta read count for the extent / delta read count for all extents. The delta in both the numerator and the denominator refers to the difference between the current heat calculation and the last heat calculation.

• Effectiveness, which is defined as delta read hits on 4K pages in the extent / delta 4K pages read ahead for the extent. The delta in both the numerator and the denominator refers to the difference between the current effectiveness calculation and the last effectiveness calculation.

• Weight, is defined to be one of three values: low, neutral, and high. Weight is high whenever the effectiveness is over 0.5. Weight is low whenever the heat is below 0.125 and the effectiveness is under 0.25. Otherwise the weight is neutral.

• Cache reads ahead based on the weight of the target extent and the size of the prompting read command.

• When the prompting read command requested less than 4 Kilobytes, cache will read ahead 4 Kilobytes if the weight is high or neutral.

• When the prompting read command requested between 4 Kilobytes and 32 Kilobytes, cache will read ahead the remainder of the track if the weight is not negative and it will also read ahead the next track if the weight is positive.

• When the prompting read command requested between 32 Kilobytes and 256 Kilobytes inclusive, cache will read ahead the remainder of the track if the weight is not negative and it will also read ahead the size of the prompting read if the weight is positive.

3.6.2 AMP algorithm

AMP is also a per-stream prefetching algorithm which works an asynchronous type of prefetching. It also defines adaptive P and G for prefetching. It works like as follows.

- P and G are adaptive components. P shows degree of prefetch and G shows when to prefetch same as what we have discussed before.

- Initially the algorithm works as fixed synchronous prefetch.

- Once P increases to some threshold (A.P.T) they switch the mode to adaptive asynchronous.

- A page at a distance of *APT/2* from the last prefetched page is chosen as the prefetch trigger page and the *tag* is set (G)

- If *p* is more than this optimal value, the last page in a prefetched set will reach the LRU end unaccesed . We give such a page another chance by moving it to the MRU position and setting the *old* flag.

- Whenever an unaccesed page is moved to the MRU position in this way, it is an indication that the current value of *p* is too high, and therefore we reduce the value of *p*.P is incremented by the *readsize* (the size of read request is pages) whenever there is a hit on the *last* page of a read set (pages read in the same I/O) which is also not marked *old*.

- The main idea is to increment *g* if on the completion of a prefetch, we find that a read is already waiting for the first page within the prefetch set (readWaiting()).

- If *g* was larger, the prefetch would have been issued earlier, reducing or eliminating the stall time for the read.

- Thus we increment *g*. However, we also need to decrement *g* when *p* itself is being decremented.

This is how SVC cache algorithm and AMP algorithm works. There are certain disadvantages with this way of prefetching. We will study the comparison study in the last chapter. In the next chapter we will propose our algorithm that tries to overcome these problems.

Chapter: 4 Adaptive Asynchronous Algorithm (AAA)

4.1 PRE-STAGING STRATEGY

Pre-staging strategy or Read ahead strategy attempts to reduce read / write latency by analyzing what blocks will be referred in future I/Os. The basic idea is to copy those hot blocks in the cache before they are referred during an I/O. But in order to achieve better read/write latency a strategy should be designed to suggest What , When , and How much to be pre-stage.

A pre-fetching algorithm can have a fixed or adaptive degree of pre-fetch and can be either asynchronous (when it can pre-fetch on a hit) or synchronous (when it can pre-fetch only on a miss). This naturally leads to four classes which we call fixed synchronous (FS), fixed asynchronous (FA), adaptive synchronous (AS), and adaptive asynchronous (AA).

In this thesis the algorithm that we are going to propose is an AA type algorithm. It defines a trigger page as a trigger event. Whenever there will be a hit encountered on that page the system will do some pre-fetch.

4.2 A BASIC PRESTAGE ALGORITHM

In a basic pre-fetching algorithm there are two important decisions that we have to take, which are, WHEN to pre-fetch and HOW MUCH to pre-fetch. For these to questions we are defining two decision making parameters P and G.

P is symbolized as degree of pre-fetch, or in other words the amount of data to be pre-fetched. And G defined as the distance to trigger page (which when gets hit, causing a pre-fetch) from the last page of pre-fetched data. To illustrate it more clearly let us take a look at the diagram below.

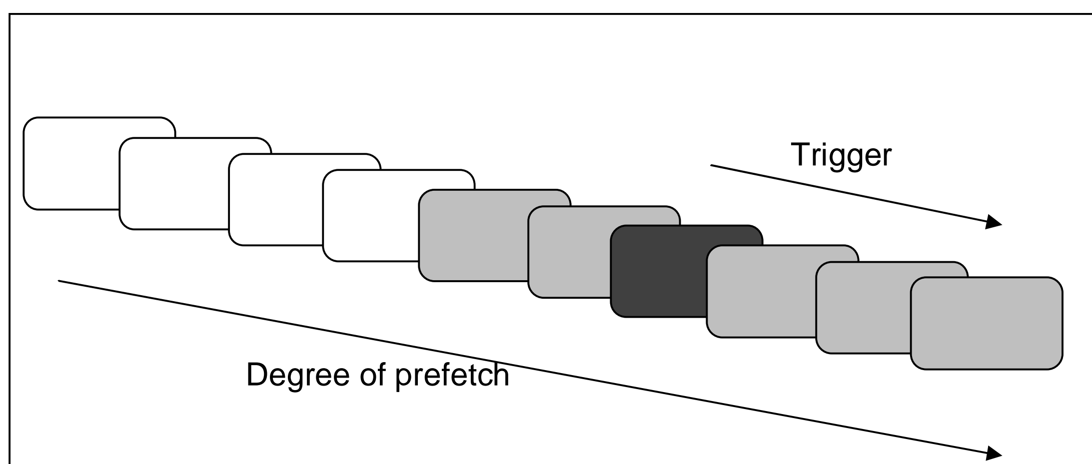

Fig 4.1: A general asynch Prestage algorithm

In this diagram empty page segments are represented as boxes. White color box are representing as accessed pages. And dark color box are represented as un-accessed pages and a black box represents trigger page.

We have to make both P and G to be adaptive in nature so that the algorithm can perform well in diverse nature of I/O patterns. For that we have to understand a thick relationship between P and G.

So through above section as we have discussed a general asynchronous pre-stage algorithm. Now in rest of the document we will consider the various sub divided design aspect of our designed algorithm.

In the next section we will discuss closely about our asynchronous algorithm. We will first see the challenges that the algorithm should defeat. And then its advantages and disadvantages.

4.3 ASYNCHRONOUS PREFETCHING ALGORITHM

The purpose of prestaging is to read the blocks from disk to main memory prior to the request. This is done by analyzing request pattern. But it is limited to sequential IO patterns. So a proper intelligence is required to analyze the patterns as well to check “WHEN “ and “HOW MUCH” to prefetch.

Prefetching in access is also harmful to system performance. The reason is that even is intelligent prefetch is done access will pollute the cache which is a very vital resource .Prefetch comes with a cost of cache pollution. Too much prefetch would limit normal read-write cache adversely effecting performance.

We need some intelligent way to prefetch data into the cache. Ideally it should be the case that the moment data is being requested by host, we prefetch it i.e. average life time for prefetched data is 0.

This is very hard to impossible, so a good algorithm should be close to that. Let us see various challenges that a good algorithm need to face.

4.4 CHALLENGES

- **Prefetch only when chance of hit is very high**

Means that to avoid cache pollution we need to prefetch only in the case when probability of getting hits is very high. In other words it is the case when we are having sequential I/O. In random I/O we the algorithm should not prefetch.

- **Should be able to handle multiple I/O size requests in a stream.**

As Host's I/O request size is variable and changes time to time. So our algorithm should be able to consider this case in to account and performed well in such variable I/O size case.

- **Should be able to handle out of order sequential requests**

In multithreading environment even sequential patterns (when request is generated by multiple threads) appears like random request patterns. So our algorithm should have such intelligence so that these types of cases can be taken care of.

- **Should be able to cope with change in host I/O rate**

It should be able to cop up with variable I/O rates of host. Ant perform well is such case also.

Our proposed AAA is a very simple algorithm which has a capability to define all these challenges in a very efficient and simple manner. In the next part of this chapter we will discuss our algorithm more deeply.

4.5 PROPOSED AAA (ADAPTIVE ASYNCHRONOUS ALGORITHM)

The AAA algorithm tries to adapt the value of P and G on per extent bases. We now draw attention to the important portions of the algorithm and the logic behind the choices we have made.

AAA algorithm works in 2 stages: RAMP UP and ASYNCH stage.

Fig 4.2 two stages of AAA.

We will see the elaboration of both the states as follows.

4.5.1RAMP UP STAGE

Ramp up stage is a first step stage in which a large and faster amount of pre-stage is done, so that a reasonable amount of buffer (of pre-fetched data) can be created in the cache and hence we can switch our self to asynchronous mode of pre-fetching.

Ramp up is a kind of boot strap problem in which u don't have any previous history available for pre-stage but still you need to initiate pre-stage.

If we see the graph of the expected prestaging of data in a ramp up stage, it would be like as given below.

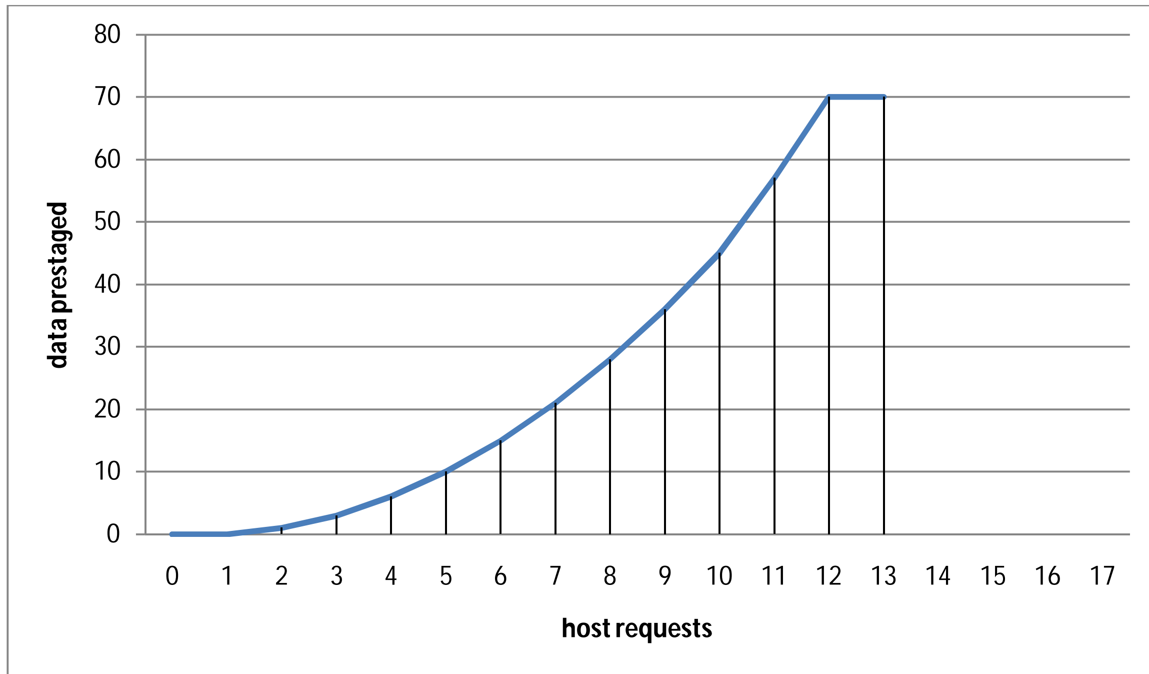

Fig: 4.3 data prestage pattern in ramp up stage

In the above graph as we have seen that in the ramp up stage the algorithm needs to prefetch exponentially so that it can reach to a desires level (in terms of multiple of avg I/O size). Once the desired level is reached we can switch to the above discussed asynchronous mode of prestaging.

So as of now this thing is clear that ramp up is important for initialization of pre-stage plus also we need to create a sufficient buffer to start asynchronous pre-stage. While making a ramp up strategy we have to consider and answer following points.

- When to start doing pre-stage?
- How much to pre-stage?
- What is the frequency of pre-stage?
- When to stop ramp-up stage and start normal prestage?

So let us take first question. In out algorithm we are calculating heat of an extent. And we maintain a heat_state counter which is initialized to 0 for an extent and incremented by one whenever the heat of that extent crossed a threshold of 0.78. So we are entering in ramp up stage at the even when our heat_state >=2.

Now the question comes that how much we need to pre-stage every time when we are in a ramp up stage. It is very much clear that we need to pre-stage a little in large amount so that we can create a sufficient buffer of available data. So currently in the initial stage of development we will pre-stage 3 times the last io-size considering that host cannot demand more than disk speed limits, otherwise ramp up cannot be possible.

In the ramp up stage the frequency of pre-stage should definitely be much higher than that of in normal asynchronous stage and should be of the order of host prestage. So in a ramp up stage we will add prestage request to lower layer at every host read.

Last question comes is that when should we come out of ramp up stage. The answer is, we have to create a sufficient buffer. So now we have to define a buffer that can't be acted as sufficient for starting of asynchronous pre-stage. At this stage we are defining that buffer should be as sufficient so that it can fulfill 20 user's I/O requests. So now the buffer size will be 20*average I/O size (that we are maintaining in some data structure with valid scope).

These specifications of ramp up stage can become more adaptive as the development goes ahead and we reach at much finer solution.

4.5.2ASYNCH STAGE

In Asynch Stage our main concern is to maintain the minimum buffer (we define it as equal to 20*(avg I/O size). We may note that– this size of this buffer is dependent on system state. So in this stage we synchronize our parameters P and G so that we can attain this buffer amount.

In this phase we required to do small number (but of large) of prestage so we allow only one prestage I/O at a time and if there is more than one I/O, then we will merge them into one big I/O.

Apart from this we also take a track on the max buffer level in cache and minimum buffer as well. To define both of them we map them to cache prefetching events as follows.

Min. Buffer level - just before prestage completes.
Max. Buffer level – just after prestage completes.

At the time of completion of our prestage we know, “how much host access?”And “how much we prefetched?” . Through which we can find the change in host request rate and we can adapt P and G.

4.5.3 ADAPTIVENESS OF P AND G

In a pre-fetching algorithm there are two important decisions that we have to take, which are, WHEN to pre-fetch and HOW MUCH to pre-fetch. For these to questions we are defining two decision making parameters P and G.

P is symbolized as degree of pre-fetch, or in other words the amount of data to be pre-fetched. And G defined as the distance to trigger page (which when gets hit, causing a pre-fetch) from the last page of pre-fetched data. To illustrate it more clearly let us take a look at the diagram below.

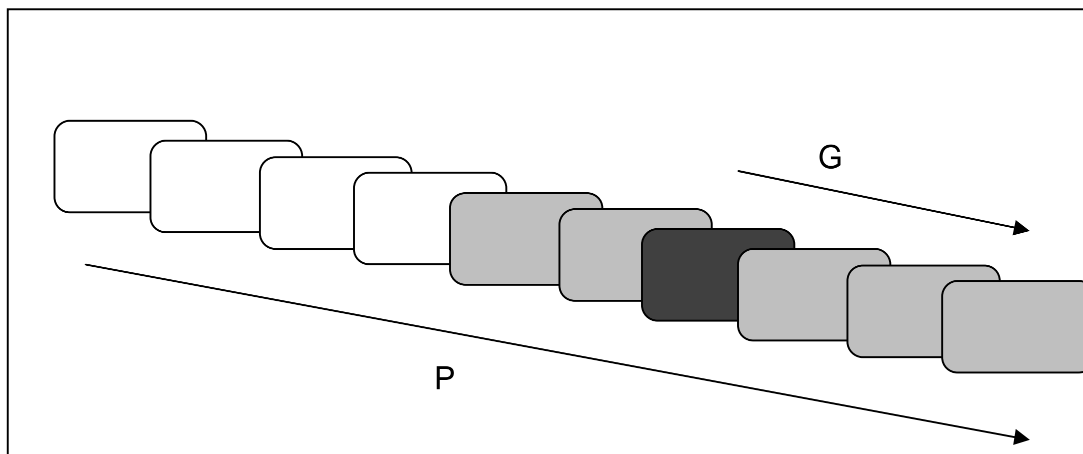

Fig 4.4: asynch Prestage algorithm (AAA)

In this diagram empty page segments are represented as boxes. White color box are representing as accessed pages. And dark color box are represented as un-accessed pages and a black box represents trigger page.

We have to make both P and G to be adaptive in nature so that the algorithm can perform well in diverse nature of I/O patterns. For that we have to understand a thick relationship between P and G.

One of the most important aspect of our algorithm is that all the time our algorithm tries to maintain a minimum buffer data in to the cache (20 * I/O size).When the host I/O rate changes, we have to adapt either of P and G to adapt the new situation such we have to maintain minimum buffer levels.

Now let's understand that what is the impact of change of host rate on buffer and how we adapt it by changing the values of p and g. The level of buffer in our cache is minimum just before our last prefetch request data fills the data in cache. Now host's request rate can be increased or decreased

1. When host increase its rate of request

- In this case we can maintain the buffer level by either increasing our amount of prefetch (increasing P) or by requesting for prefetch earlier (increasing G).

2. When host decrease its rate of request

- In this case we can maintain the buffer level by either decreasing our amount of prefetch (decreasing P) or by delaying the prefetch request (decreasing G).

So we can make the equilibrium with both P and G but we have to choose one of them.

In our algorithm we always first try to handle the uncertain situation by adjusting G. we increase G if host request increases and vice versa. But G has limitations (g can be

adjusted from value 0 to P). To summarize it in 2 lines we can say

- We define a range of G (as favourable rage which will be determined by system state).

- If host rate increases (means our buffer comes less the minimum buffer threshold) we will decrease G (or increase P is G is exceeding the favourable range) and vice-verse

For that we are defining a range of g (p/4 to 3p/4). And when G reaches close to its end limits we will alter P to make G again bake to its normal range.

Chapter: 5 Simulation of AAA and AMP algorithms

As we have studied both AMP and AAA (our proposed algorithm) in previous chapters. Now in this chapter we will discuss about the simulation of both the algorithms in c#.net (using visual studio). Let us first give a basic introduction to Microsoft Visual Studio.

Microsoft Visual Studio is an integrated development environment (IDE) by Microsoft. It is used to develop console and GUI applications along with Windows Forms applications, web sites, web apps, and web services in both native code together with managed code for all platforms supported by Microsoft Windows, Windows Mobile, Windows CE, .NET Framework, .NET Compact Framework and Silverlight.

Visual Studio includes a code editor supporting IntelliSense as well as code refactoring. The integrated debugger works both as source-level debugger and machine-level debugger. Other built-in tools include a forms designer for building GUI apps, web designer, class designer, and database schema designer. It accepts plug-ins that further enhance the functionality at almost every level—which includes adding support for source-control systems (like Subversion and Visual SourceSafe) and adding new tools like visual editors and visual designers for domain-specific languages or tools for

other aspects of the software development lifecycle (like the Team Foundation Server client: Team Explorer).

Visual Studio supports multiple programming languages by means of language services, which allow the code editor and debugger to support (to varying degrees) nearly any programming language, provided a language-specific service exists. Built-in languages include C/C++ (via Visual C++), VB.NET (via Visual Basic .NET), C# (via Visual C#), and F# (as of Visual Studio 2010[2]). Support for some other languages such as M, Python, and Ruby among others is available through language services installed separately. It also supports XML/XSLT, HTML/XHTML, JavaScript and CSS etc. Individual language-specific versions of Visual Studio also exist which provide more limited language services to the user: Microsoft Visual Basic, Visual J#, Visual C#, and Visual C++.

5.1. **Cache Simulator**

We developed a cache simulator that simulates a cache in form of an array. The data pages are numbered with unique page_id (interger starting from 0). Every time when the page is being staged to the cache, one entry is added to cache (array). We also simulated the latency of staging and destaging of page by adding sleeps at multiple points to make the simulation closer to real.

For I/O generation we created a multi-threaded environment of host threads that creates multiple threads for I/O to simulate the situation similar to most of the modern volume controllers.

We also created a log component that puts a log in the log section at every I/O, which shows how the I/O is being executed in the algorithm. We created a GUI for the simulator to make is intuitive and easy to give inputs and get outputs. The basic GUI and its components are given below.

- **BASIC GUI**

The figure below shows the basic GUI for the simulator

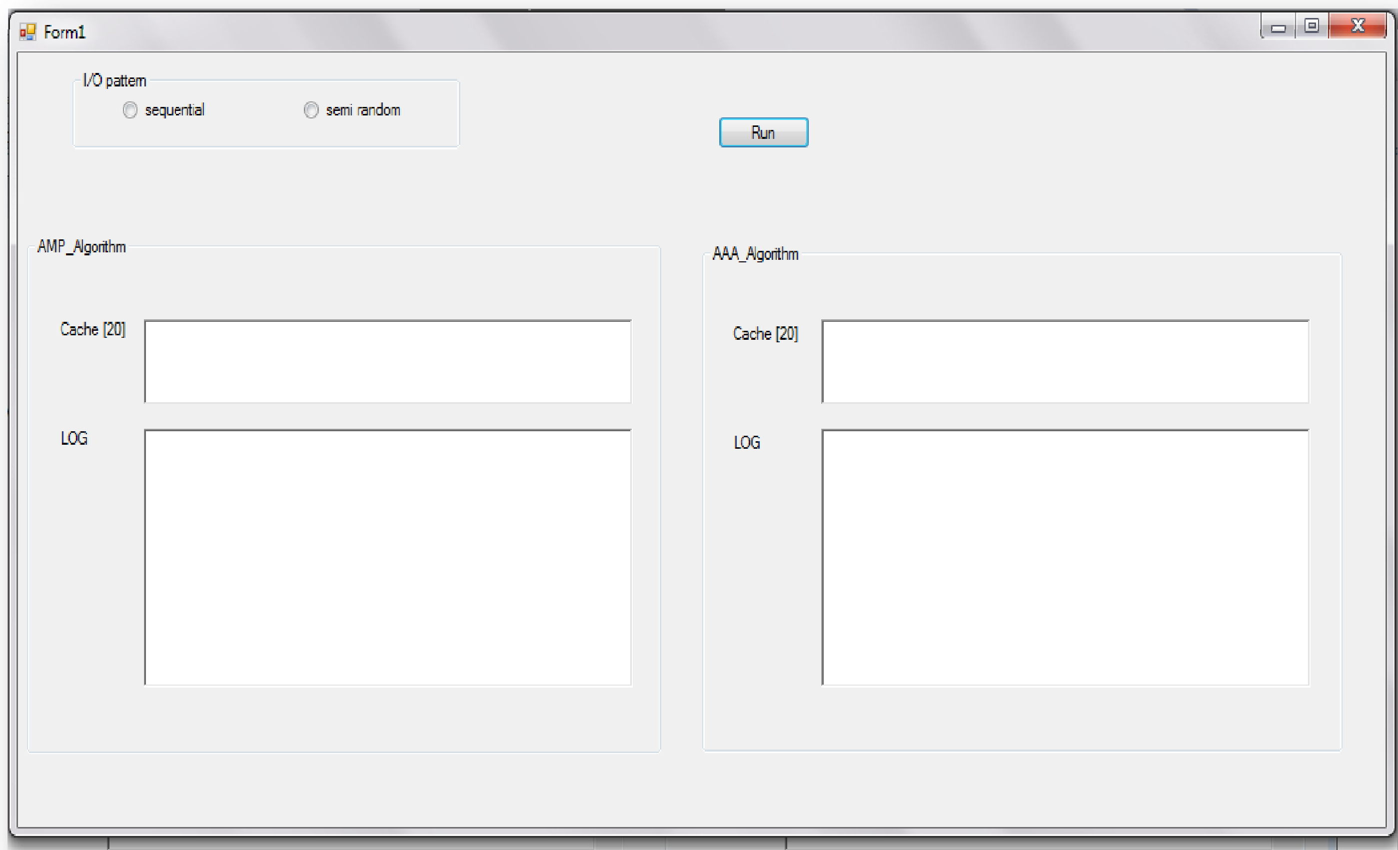

Fig5.1 basic BUI for cache simulator

The description of GUI and its basic components are in the section as follows.

- **I/O pattern type Section**

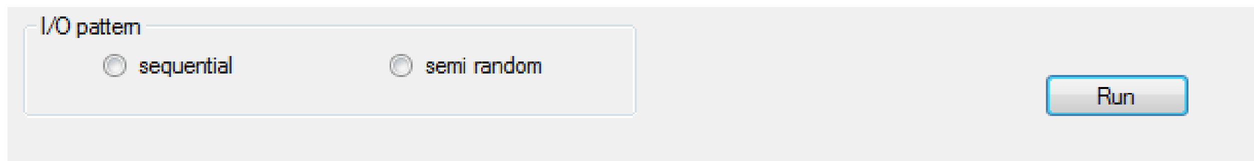

Fig 5.2 GUI for I/O pattern selection

In this part of GUI you can select the type of I/O pattern. We included two types of I/O, Sequential and Semi random. We did not include the random type of I/O because it is not very common case to have a random type of I/O in a volume controller usually.

- **Log Section**

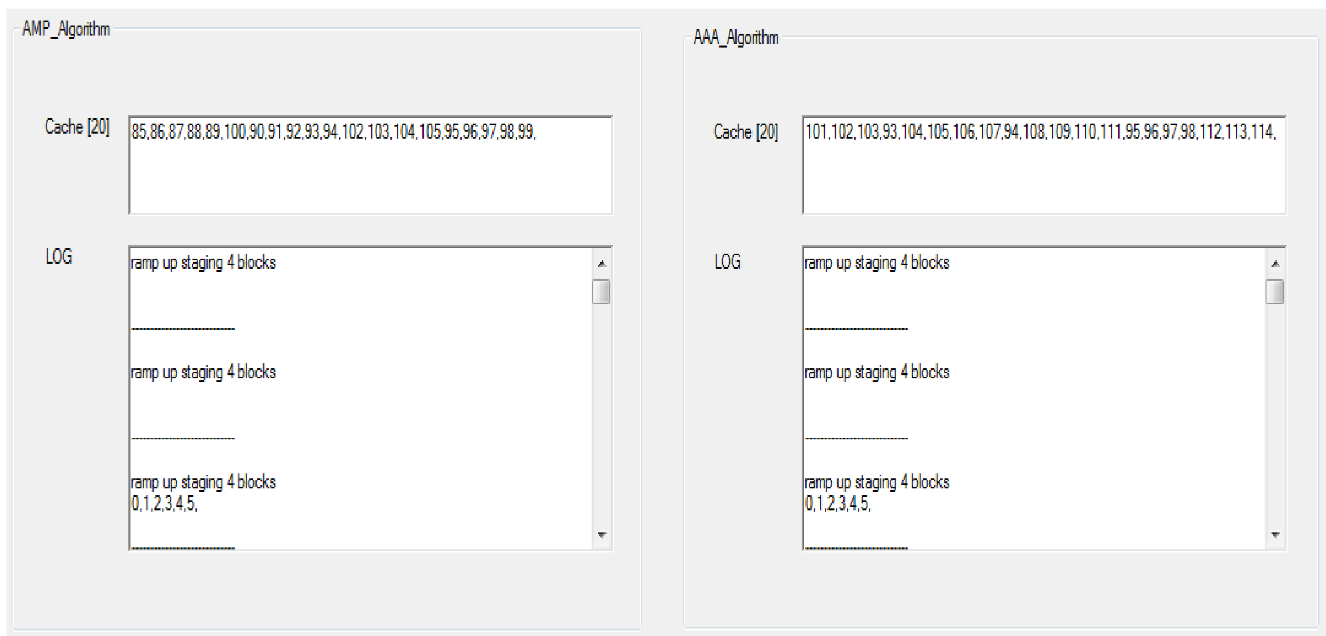

Fig 5.3 GUI for log section

In this section the generated LOG is shown. This log gives all the details of how the algorithm reacts in different I/O patterns. This also gives a better way to compare both the algorithms.

In the next chapter we will compare the two algorithms on various factors and on the basis of results we got from this simulation.

Chapter: 6
Comparative Study

In this chapter we will compare the two reference algorithms and our proposed algorithm. Here we will see the basic problems with AMP algorithm and reference algorithm used in IBM storage volume controller.

Due to privacy reasons we cannot discuses IBM's SVC algorithm in much details but we try to consolidate the comparison in affordable reasons.

6.1. Comparison between IBM's SCV algorithm and AAA (proposed)

IBM's SVC prefetching algorithms is a synchronous type prefetching algorithms. In the algorithm the cache always stages the data at every miss. Similarly it stages the data at prestage hits also. This synchronous behavior of algorithm causes multiple problems.

- No consideration of the amount of data prestaged and available in cache

- This current algorithm does not consider the amount of data present in the cache, but on every read miss or prefetch hit it stages the data blindly.

- Also it do not consider any thing about the data which is there in stage request but not available in cache.

- On the other hand the current algorithm due to continuous prestaging more and more data is being accumulated which causes cache wastage and average lifetime of data in cache increases.

- Also the current algorithm does not perform well in close random I/O. The reason is a bit confidential and cannot be discussed.

- Prefetching decision of current algorithm is based on calculated effectiveness which fluctuates and effects the decision and a uniform pattern of staging cannot be made.

6.2. Comparison between AMP algorithm and AAA (proposed)

Both the AMP algorithm and our proposed algorithm are very much similar in terms of decision parameters P and G. Both are adaptive asynchronous type algorithm and works well in multi-threading environment.

But the basic difference among both the algorithm is the way of adapting P and G. let us discuses the basic advantages of out proposed algorithm over AMP algorithm.

- Adapting the value of p

This thing is really clear that the more accurate your decision parameters are the better efficiency your system can achieve. In AMP algorithm it tries to optimize the value of P only when either an unaccessed page comes to LRU position or when a last page of

prefetch sequence is unaccessed and got hit.

Because of this there is a delay in updating the value of P to its optimal (or close to optimal) value. In between these updating events we make have multiple prefetch requests that use non optimal value of P.

- Adapting the value of G

Similar is the case with adapting G also, the AMP algorithm updates G every time when a prefetch completes and read request is pending. It can cause multiple subsequent read requests into miss.

So AMP algorithm lags a logic to map the value of P and G. both the parameter are dependent of each other and have a dependent relationship among each other. We can manipulate P (and / or) G to cope up with change in user rate and patter of I/O request.

This is the reason why AAA algorithm is more efficient then AMP algorithm. It tracks the buffer amount in cache and dynamically changes the value of P (and / or) G according to the needs.

- No mechanism to detect the close but random I/O

In AMP algorithm it treats a close but random I/O as a part of sequential I/O which can cause unintelligent prefetching. Where as in our proposed algorithm in random I/O which is can be detected by its distance from last prefetch and will be taken care of.

- No mechanism to detect a semi- random I/O

In the AMP algorithm have no mechanism to distinguish between a random and semi random I/O as such. And on the other hand our AAA algorithm distinguishes between semi random and random I/O. by considering the distance of I/O from last prefetch.

All these factors make our AAA algorithm a better algorithm and it works well in all type of I/O patterns with all the challenges like dynamic I/O size and multi threading etc.

We tested both the algorithms in out simulator on both sequential as well as semi random I/Os. We got the following results.

Test runs on sequential I/O

Test runs	Number of Miss	
	AMP	AAA(our proposed algorithm)
1	23	12
2	21	18
3	21	12

Table 6.1 Results for sequential pattern of I/O

If we plot the above result in form of a graph, it looks like this.

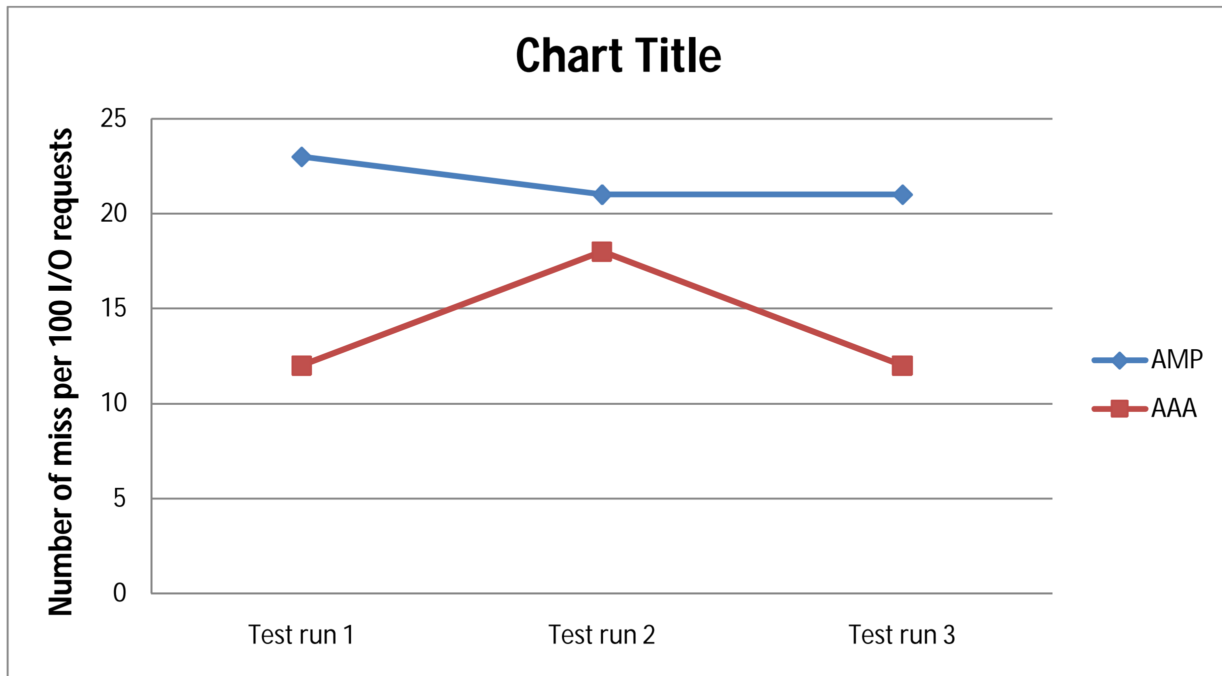

Fig 6.1 plot showing number of miss for sequential I/O patterns

Through the graph it can easily be shown that AAA works well in all the 3 test runs. Similar are the result for Semi-Random I/O patterns.

Test runs of Semi-Random I/O

Test runs	Number of Miss	
	AMP	AAA(our proposed algorithm)
1	85	44
2	90	41
3	85	36

Table 6.2 Results for Random pattern of I/O

If we plot the above result in form of a graph, it looks like this.

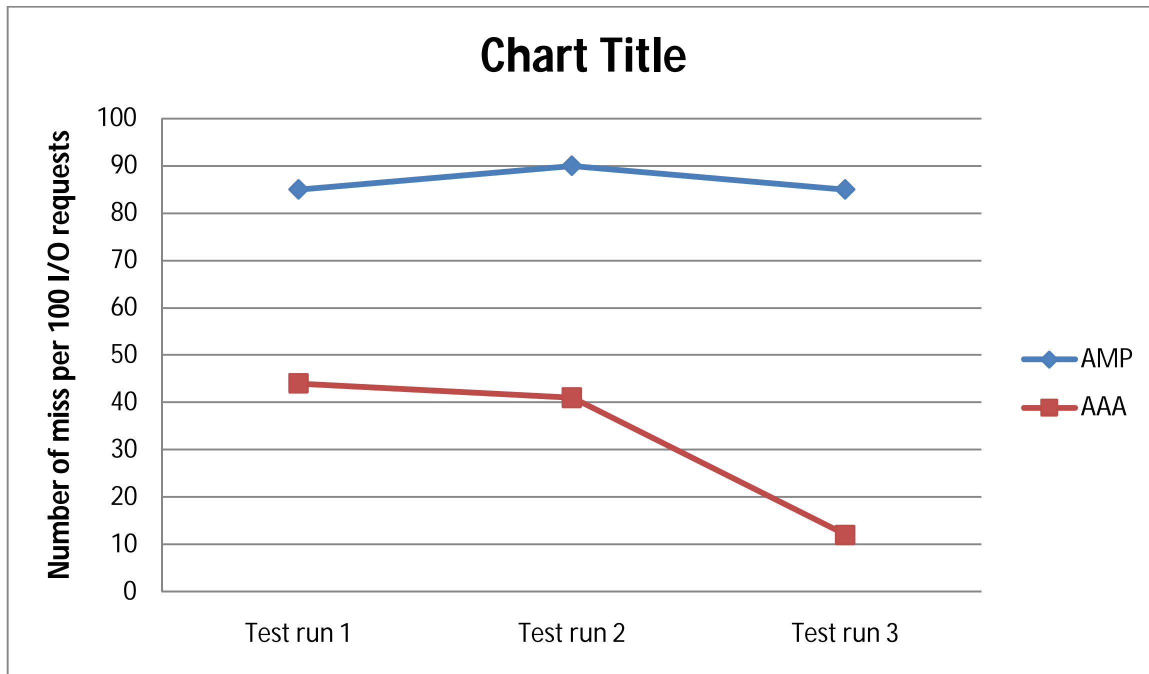

Fig 6.2 plot showing number of miss for semi-random I/O patterns

These test results also shows that our proposed algorithm works significantly better in semi–random I/O pattern. And because of this our algorithm seems more suitable for modern volume controllers in which sequential I/O patterns appear semi-random because of Multi-threaded host I/Os.

Chapter: 7 Conclusion and Future Work

7.1 Conclusion

Sequential prefetching is one of the widely used prefetching techniques in storage volume controllers .We found that there is a need for a way of prefetching that can adapt both the prefetch degree P and the trigger distance G on a per-stream basis in response to changing user's I/O patterns and workloads.

We studied two very efficient prefetching algorithms as discussed and on the basis of knowledge gained we proposed a novel, simple, adaptive, prefetching algorithm called AAA (adaptive asynchronous algorithm).

We compared all the three algorithms on various challenges and conclude that our algorithm works better in many cases and has the ability to overcome the deficiencies in the other two algorithms. Our proposed algorithm is general algorithm and can be applied in any storage virtualization layer.

7.2 Future Work

For future work we are planning to enhance our algorithm further, such as interaction with the host application and use this knowledge to derive a better decision of “when” and “how much” to prefetch.

We will also try to find out more efficient way to adapt the value of P and G is incrementing and decrementing P and G by an adaptive and dynamic parameter based on current state of cache.

Also we are trying to simulate a general volume controller on which we will prototype AMP algorithm and our proposed algorithm. We can use the result from this simulation for better comparison among the algorithms and it will give us more points for further enhancement of our algorithm.

References

1) Implementing the IBM System Storage SAN Volume Controller V4.3 (SVC hand book) JON TATE, SAMEER DHULEKAR, JUERG HOSSLI, DAN KOECK, SUAD MUSOVICH.

2) BINNY S. GILL ACM Transactions on Storage, Vol. 3, No. 3, Article 10, Publication date: October 2007 Optimal Multistream Sequential Prefetching in a Shared Cache IBM Almaden Research Center and LUIS ANGEL D. BATHEN University of California, Irvine

3) CAO, P., FELTEN, E. W., KARLIN, A. R., AND LI, K. 1995. A study of integrated prefetching and caching strategies. In Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems. 188–197.

4) CHEN, T. -F. AND BAER, J. -L. 1992. Reducing memory latency via non-blocking and prefetching caches. In Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS). SIGPLAN Not. 27, 9, 51–61.

5) CHEN, T. -F. AND BAER, J. -L. 1995. Effective hardware based data prefetching for high-performance processors. IEEE Trans. Compute. 44, 5, 609–623.

6) CUREWITZ, K. M., KRISHNAN, P., AND VITTER, J.S. 1993. Practical prefetching via data compression. ACM SIGMOD Rec. 22, 2, 257–266.

7) DAHLGREN, F., DUBOIS, M., AND STENSTR¨OM, P. 1993. Fixed and adaptive sequential prefetching in shared memory multiprocessors. In Proceedings of the International Conference on Parallel Processing (ICPP), 56–63.

8) DAHLGREN, F. AND STENSTR¨OM, P. 1996. Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors. IEEE Trans. Parallel Distrib. Syst. 7, 4, 385– 398.

9) DAVID CALLAHAN, K. K. AND PORTERFIELD, A. 1991. Software prefetching. In ACM SIGARCH Comput. Archit. News. 19, 2, ACM Press, New York, 40–52.

10) FU, J. W. C. AND PATEL, J. H. 1991. Data prefetching in multiprocessor vector cache memories.In Proceedings of the 18th Annual International Symposium on Computer Architecture, Toronto, Intario, Canada, 54–63.

11) GILL, B. S. ANDMODHA, D.S. 2005a. SARC: Sequential prefetching in adaptive replacement cache. In Proceedings of the USENIX Annual Technical Conference. 293–308.

12) GORNISH, E. H., GRANSTON, E. D., AND VEIDENBAUM, A. V. 1990. Compiler-Directed data prefetching in multiprocessors with memory hierarchies. In Proceedings International Conference on Supercomputing ACM SIGARCH Comput. Archite. News. 18, 3, 354–368.

13) GRIFFIOEN, J. AND APPLETON, R. 1994. Reducing file system latency using a predictive approach. In Proceedings of the Conference, USENIX Summer. 197–

207.

14) GRIMSRUD, K. S., ARCHIBALD, J. K., ANDNELSON, B. E. 1993. Multiple prefetch adaptive disk caching.IEEE Trans. Knowl. Data Eng. 5, 1, 88–103.

15) JAIN, P., DEVADAS, S., AND RUDOLPH, L. 2001. Controlling cache pollution in prefetching with software-assisted cache replacement. Tech. Rep. CSG-462. Massachusetts Institute of Technology.

16) JOSEPH, D. AND GRUNWALD, D. 1999. Prefetching using Markov predictors. IEEE Trans. Comput. 48, 2, 121–133. KALLAHALLA, M. AND VARMAN, P. J. 2002. Pc-Opt: Optimal offline prefetching and caching for parallel I/O systems. IEEE Trans. Comput. 51, 11, 1333–1344.

17) KIMBREL, T. ANDKARLIN,A.R. 1996. Near-Optimal parallel prefetching and caching. In Proceedings of the (FOCS), IEEE Symposium on Foundations of Computer Science. 540–549.

18) KIMBREL, T., TOMKINS, A., PATTERSON, R. H., BERSHAD, B., CAO, P., FELTEN, E., GIBSON, G., KARLIN, A. R., AND LI, K. 1996. A trace-driven comparison of algorithms for parallel prefetching and caching. In Proceedings of the Symposium on Operating Systems Design and Implementation, USENIX Association, 19–34.

19) KOTZ, D. AND ELLIS, C. S. 1991. Practical prefetching techniques for parallel file systems. In Proceedings of the 1st International Conference on Parallel and Distributed Information Systems.IEEE Computer Society, 182–189.

20) KROEGER, T. M., LONG, D. D. E., AND MOGUL, J. C. 1997. Exploring the bounds of web latency reduction from caching and prefetching. In USENIX Symposium on Internet Technologies and Systems.

21) LEE, R. L., YEW, P. -C., AND LAWRIE, D. H. 1987. Data prefetching in shared memory multiprocessors.

22) LEI, H. AND DUCHAMP, D. 1997. An analytical approach to file prefetching. In *the USENIX Annual Technical Conference*, Anaheim, CA.

23) LIPASTI, M. H., SCHMIDT,W. J., KUNKEL, S. R., AND ROEDIGER, R. R. 1995. SPAID: Software prefetching in pointer- and call-intensive environments. In *Proceedings of the 28th Annual IEEE/ACM International Symposium on Microarchitecture*, 231–236.

24) LUK, C. -K. AND MOWRY, T. C. 1996. Compiler-Based prefetching for recursive data structures. In *Architectural Support for Programming Languages and perating Systems*. 222–233.

25) METCALF, C. 1993. Data prefetching: A cost/performance analysis. Area Exam, MIT, October.

26) MOWRY, T. AND GUPTA, A. 1991. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *J. Parallel Distrib. Comput. 12,* 2, 87–106.

27) PATTERSON,R. H.,GIBSON,G. A.,GINTING, E., STODOLSKY,D., AND ZELENKA, J. 1995. Informed prefetching and caching. In *Proceedings of the ACMSymposium on Operating Systems Principles (SOSP)*. 79–95.

28) REUNGSANG, P., PARK, S. K., JEONG, S. -W., ROH, H. -L., AND LEE, G. 2001. Reducing cache pollution of prefetching in a small data cache. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*. 530–533.

29) ROGERS, A. AND LI, K. 1992. Software support for speculative loads. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, *SIGPLAN Not. 27*, 9, 38–50.

30) ROTH, A.,MOSHOVOS, A., AND SOHI, G. S. 1998. Dependence based prefetching for linked data structures. *ACM SIGPLAN Not. 33,* 11, 115–126.

31) STORAGE VIRTUALIZATION , Wikipedia http://en.wikipedia.org/wiki/Storage_virtualization