Reducing Disk I/O Performance Sensitivity for Large Numbers of Sequential Streams

by

Georgios Panagiotakis

Thesis

Presented to the Faculty of the Graduate School of

University of Crete

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Computer Science

University of Crete

May 2006

Reducing Disk I/O Performance Sensitivity for Large Numbers of Sequential Streams

Georgios Panagiotakis

Master's Thesis

Department of Computer Science
University of Crete

Abstract

Over the last few years I/O subsystem performance issues have attracted significant attention due to the increasing needs for storing and retrieving information. Cost-effective, large-scale storage systems can enable novel online content delivery services. In particular, rich media content is becoming increasingly popular due its importance for entertainment, education, and other application domains.

However, retrieving sequential rich media content from modern commodity disks is a challenging task. As disk capacity increases, there is a need to increase the number of streams that are allocated to each disk. However, when multiple streams are accessing a single disk, throughput is dramatically reduced because of disk head seek overhead, resulting in higher requirements for disk numbers. Thus, there is a tradeoff between how many streams should be allowed to access a disk and the total throughput that can be achieved.

In this work we first examine this tradeoff using simulation. We use Disksim, a detailed architectural simulator, to examine several aspects of an I/O subsystem and we show the effect of various disk parameters on system performance under multiple sequential streams. Then, we propose a solution that adjusts I/O request

streams, based on host and I/O subsystem parameters. We implement our approach in a real system and perform experiments with a small and a large disk configuration. Our approach improves disk throughput up to a factor of 8 with a workload of 100 sequential streams, without requiring large amounts of memory on the storage node. Moreover, it is able to adjust (statically) to different storage node configurations, essentially making the I/O subsystem insensitive to the number of I/O streams used.
Thesis Supervisor: Angelos Bilas

Μειώνοντας την ευαισθησία των δίσκων του υποσυστήματος Εισόδου/Εξόδου για μεγάλο αριθμό διαδοχικών ροών δεδομένων

Γεώργιος Παναγιωτάκης

Μεταπτυχιακή Εργασία

Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

Περίληψη

Τα τελευταία χρόνια οι επιδόσεις του υποσυστήματος Εισόδου/Εξόδου έχουν συγκεντρώσει σημαντική προσοχή, εξαιτίας κυρίως των ολοένα και αυξανόμενων απαιτήσεων για αποθήκευση και ανάκτηση δεδομένων. Οικονομικά, μεγάλης κλίμακας αποθηκευτικά συστήματα μπορούν να παρέχουν καινοτόμες διαδικτυακές υπηρεσίες παροχής υλικού. Συγκεκριμένα το περιεχόμενο των εμπλουτισμένων μέσων γίνεται ολοένα και πιο δημοφιλές εξαιτίας της σπουδαιότητας του στη διασκέδαση, την εκπαίδευση και τους άλλους τομείς εφαρμογών.

Όμως η ανάκτηση περιεχομένου εμπλουτισμένων μέσων απο σύγχρονους δίσκους είναι ένα εξαιρετικά απαιτητικό εγχείρημα. Όσο η χωρητικότητα του δίσκου αυξάνει, τόσο μεγαλώνει η ανάγκη να αυξηθεί ο αριθμός των ροών δεδομένων που προσπελαύνουν ένα δίσκο. Όμως όταν πολλαπλές ροές δεδομένων προσπελαύνουν ένα δίσκο τότε η απόδοση του μειώνεται δραματικά, εξαιτίας των επιβαρύνσεων που προκύπτουν από τη μετακίνηση της κεφαλής του δίσκου, με αποτέλεσμα να οδηγούμαστε σε συνεχώς αυξανόμενες απαιτήσεις για μεγαλύτερο αριθμό δίσκων. Συνεπώς θα πρέπει να υπάρξει κάποιος συμβιβασμός μεταξύ του αριθμού των ροών στις οποίες θα πρέπει να επιτραπεί να προσπελαύνουν ένα δίσκο, και στη συνολική απόδοση η οποία μπορεί

να επιτευχθεί.

Σε αυτήν την δουλειά εξετάζουμε αυτό τον συμβιβασμό χρησιμοποιώντας προσομοίωση. Χρησιμοποιούμε τον Disksim έναν λεπτομερή προσομοιωτή αρχιτεκτονικών, για να εξετάσουμε τις διαφορετικές πλευρές ενός υποσυστήματος Εισόδου/Εξόδου, και δείχνουμε τις επιδράσεις που προκαλούν οι διάφορες παράμετροι του δίσκου στις επιδόσεις του συστήματος, υπό την παρουσία πολλαπλών ροών δεδομένων. Έπειτα προτείνουμε μια λύση που ρυθμίζει τον αριθμό των ροών δεδομένων Εισόδου/Εξόδου βασιζόμενη σε υλοποίηση στο χώρο του χρήστη και στις παραμέτρους του υποσυστήματος Εισόδου/Εξόδου. Εφαρμόζουμε την προσέγγιση μας σε ένα πραγματικό σύστημα και πραγματοποιούμε πειράματα με μικρές και μεγαλύτερες διατάξεις δίσκων. Η προσέγγιση μας επιτυγχάνει βελτίωση εώς και 8 φορές, με ένα φόρτο εργασίας αποτελούμενο από 100 σειριακές ροές, χωρίς να απαιτεί μεγάλη ποσότητα μνήμης. Επιπλέον είναι ικανή να προσαρμόζεται (στατικ·α) σε διαφορετικές αρχιτεκτονικές αποθηκευτικών κομβών, κάνοντας με αυτό τον τρόπο το σύστημα ανθεκτικό στον αριθμό των ροών δεδομένων που χρησιμοποιούνται.

Επόπτης Μεταπτυχιακής εργασίας: Άγγελος Μπίλας

Acknowledgments

First of all I would like to thank my supervisor, Angelos Bilas who kept an eye on the progress of my work and was always available when I needed his advice. The extensive discussions about my work and his wide knowledge and logical way of thinking have been of a great value for me.

I also wish to thank Michail D. Flouris for providing a lot of assistance especially with the equipment I used in my work. I have also benefited from many discussions and constructive comments that helped me with the development of my thesis.

I am also very grateful to my colleagues, who helped me with my work, through discussion and the routine mutual aid, which was indispensable. Special thanks to G. Kotsis and K. Xinidis for their encouragement and assistance and of course to G. Passas, K. Kapelonis, S. Passas, and D. Xinidis.

Last and most important I would like to thank my family for supporting my dreams and aspirations, while being a constant source of support and encouragement for me.

Georgios Panagiotakis

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the years technology constantly advances to keep pace with the enduringly increasing user requirements more efficiently, so that its benefits will be available to all sectors of society and individuals who may wish to use it. Nowadays, a new world of virtually limitless data is craving for higher performance computer systems and storage platforms. In particular, in recent times rich media content is becoming increasingly popular and it is deployed in a wide spectrum of application domains, varying from educational, medical, informational purposes to just plain entertainment.

Faster processors, memories, and even networks all contribute on boosting the performance of large-scale storage systems used to store and retrieve media content. However the fact remains that disk drive performance is improving at a slower rate than the other computer components. For instance, processors performance grows at a rate of 50% per year while disks drive performance achieves a 10% improvement per year. This increasing speed mismatch between system components has a great impact on both computers performance and at the end user experience. Also, given this heavy reliance on disk drives that exists nowadays, I/O subsystems can become even more restrictive on overall systems performance, unless disk drives

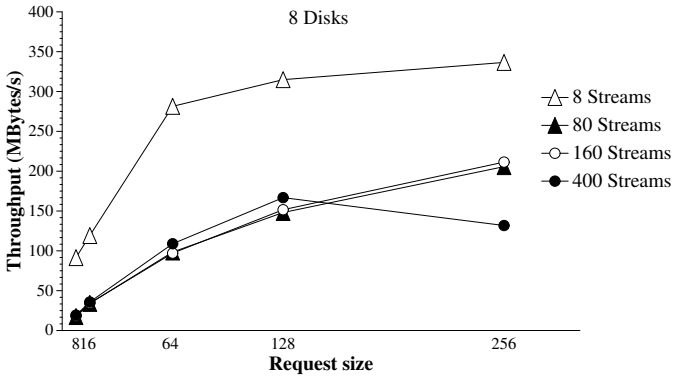
may be used intelligently.

The increasing requirements of many application domains in I/O capacity and performance has resulted in new tradeoffs and challenges in the design of I/O subsystems. In particular, with the increasing popularity of media content applications and services, there is a need to design cost-effective disk I/O subsystems that will be able to scale to large numbers of I/O streams.

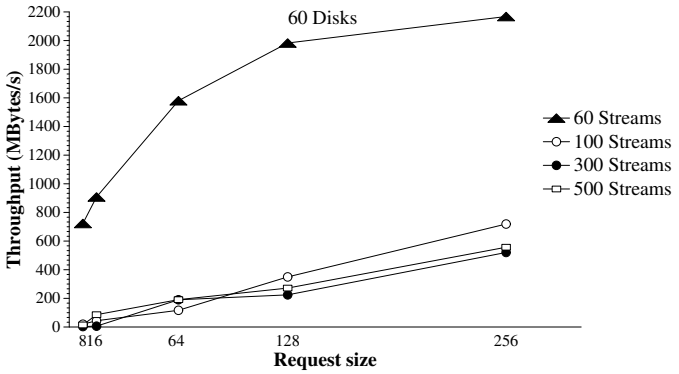
One of the main types of access that media content requires is sequential access for storing and retrieving (large) streams. Although media applications and services may require other, more demanding types of accesses as well, sequential access is almost always important, especially for read-only and write-once type applications.

Sequential access is considered the simplest and most efficient type of access for traditional disk subsystems, due to the fact that it results in minimum head movement and thus, seek overhead. Disks, usually achieve their maximum throughput with request sizes of 64-128 KBytes and above. However, modern disk I/O subsystems that are required to scale to large numbers of I/O streams present new challenges.

As disk capacities increase, building cost-effective disk subsystems requires servicing multiple sequential streams from the same disk. For instance, disk capacities are currently approaching 1 TByte for a single spindle. Such a disk could store hundreds or thousands of media streams. Moreover, disk throughput has not been increasing at the same rate, limiting maximum disk throughput to less than 100 MBytes/s. While building a storage system that is required to service large numbers of I/O streams, there is a tradeoff on how many streams should be serviced by each disk to sustain the required stream throughput and the number of disks in the subsystem to limit system cost. For instance, if an application requires streams of 1 MByte/s then, although a disk of let's say maximum throughput of 50



(a) 2 Ctrls - 8 Disks



(b) 15 Ctrls - 60 Disks

Figure 1.1: Throughput of multiple sequential streams for configurations with 8 and 60 disks. (Disksim simulations)

MBytes/s could sustain 50 streams, in practice a much smaller number of streams can be serviced by a disk, requiring more disks to service the workload. Figure 1.1 shows how disk throughput degrades in a system when multiple sequential streams are serviced by the same disk. Figure 1.1 shows that throughput scales with the number of disks. However, as the number of streams increases, throughput drops dramatically, by a factor of 2-5.

Moreover, this problem is exacerbated by the fact that it is difficult to address the problem through static data placement. Streams are generally short-lived compared to the time it takes to reorganize data on a disk. Thus, optimizing data place-

ment for a given workload to reduce head movement is not practical. Finally, given the diversity of commodity disks and I/O subsystems, it usually is difficult to tune an application or service for different subsystems, requiring a system-independent approach to addressing this problem.

Current trends in technology present new opportunities for addressing this problem in a transparent manner. Storage is becoming increasingly networked for scalability, performance, and management reasons. In such setups, disks are attached to network controllers that are similar to today's PCs. Each storage controller is equipped with a number of disks (usually 4 to 32), a controller and memory similar to the host CPU and memory, and one or more network interfaces. All application I/O requests are received and completed through the network interfaces. Such systems, are (or can be) equipped with fairly large amounts of volatile memory (regular DRAM). Thus, they present the opportunity to using the available CPU cycles and memory capacity for improving disk accesses.

1.1 Thesis Layout

In this work we first examine the various I/O parameters that affect system behavior for multiple sequential workloads. We use DiskSim [3], a detailed architectural simulator, to examine the impact of each parameter on disk throughput for various disk and workload configurations. We find that, although some parameters are able to address this issue, these are usually out of the control of the host operating system and the application. Moreover, due to the diversity required at the storage subsystem, it is unlikely that these parameters will be tuned solely for sequential workloads.

Next, we propose an extension to the I/O protocol stack, that sees all I/O requests at the host level and adjusts request size and order, to improve disk throughput and thus, disk utilization. Our approach relies on (1) identifying and separating

sequential stream access to disks, (2) using host memory as intermediate buffers for effectively coalescing small user requests to larger disk requests, and (3) using a notion similar to the working set when servicing multiple I/O streams to mitigate the impact on I/O request latency.

We implement our solution at the user-level and experiment with both low- and high- disk throughput configurations. We use a single host attached to commodity SATA disks through two SATA controllers. We show that our approach makes the system insensitive to the number of sequential streams used, achieving an improvement of up to 8 times with 100 sequential streams. We also examine the impact of the available host memory and we find that even small amounts of host-level buffering can be very effective in improving disk utilization significantly. Finally, we find that the number of streams employed has the most significant impact on response time, with read-ahead size being of secondary importance.

1.2 Organization of Thesis

The rest of this thesis is organized as follows. First, Chapter 2 presents the necessary background. Then Chapter 3 presents a detailed analysis of the problem showing our simulation results. Next, Chapter 4 discusses our proposed solution and our implementation and Chapter 5 shows our experimental results. Finally, Chapter 6 draws our conclusions.

Chapter 2

Background

This chapter discuss the foundational material that is necessary for the reader to understand the concepts related on storage systems focusing on the stages of I/O path that are important for our work.

2.1 I/O Path Overview

Nowadays a whole spectrum of applications demand timely delivery of data to end users. A media server offers access to large quantities of audio, video, and other time-based data elements. Media servers retrieve data from large storage systems and deliver them to the end users, via a high speed network (Figure 2.1).

Data retrieval is accomplished via the I/O path. Figure 2.2 shows the main stages of the I/O request path. An I/O operation starts when an application decides to issue a request to the disk subsystem. This I/O request is passed to the system kernel, in which device driver are executed. The device driver is the part of an operating system that is used to handle device-specific interactions. The device driver knows what the device hardware is capable of, and thus, it is the part of the host that interacts with the controller of the device, exchanging information,

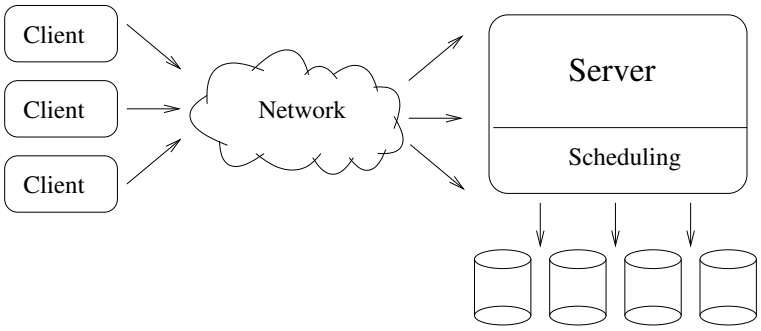


Figure 2.1: A high level view of how a media server works.

sending commands, and even receiving data.

The controller usually resides on the system bus and its job is to act as the gateway between the host and the storage devices it manages. In particular, it handles the communication protocol, through which it sends and responds to commands and controls the activity of several storage devices. The next step is to forward the request over the interface cable (such as SCSI,IDE,SATA) to the internal disk controller, which will in turn handle the retrieval of the requested data from the hard disks platters.

2.1.1 Queues and Caches on I/O path

In this work we are mainly interested in stages that involve queuing and caching. In most storage systems today, an I/O request is generated by an application, enters the kernel, is issued to the disk controller, and finally is sent to the disk itself.

I/O Queues

Queues are present in multiple stages of the I/O path, and are used to contain the list of pending requests for a particular device. Their main target is to decrease the mean response time for the requests issued, by exploiting spatial locality on disk platters. In order to achieve this, queues employ several scheduling algorithms

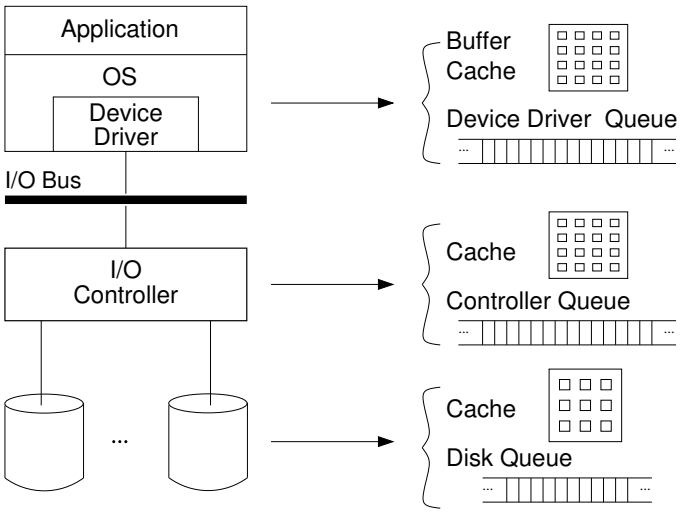


Figure 2.2: This Figure describes the overall path of an I/O request, and the components involved for the completion of a request.

that attempt to improve response time by reordering the requests. Queues can also group requests for adjacent blocks into larger ones or even merge multiple requests for the same data to a single request. The longer the queues are, the greater the possibility for throughput optimization. However as the maximum queue depth length increases, the average response time may rise.

Depending on the location of the queues different scheduling algorithms are usually employed, since every component of the I/O system has a partial knowledge of the disk drive mechanisms. For instance, the host only knows roughly the location of the last I/O it issued, while the drive, on the other hand, knows the actual geometry of the data, and the actual position of the heads.

Currently Linux 2.6 supports four different types of I/O schedulers for its queues. The default I/O scheduler is the "Anticipatory" but there are also "CFQ" (Complete Fairness Queuing), "Deadline"(for time critical applications), and Noop (stands for no operation).

The disk drives may implement a number of scheduling algorithms, based

on various criteria. The most common are FCFS , SCAN [9] and its variations (CSCAN [12], VSCAN [23]), LOOK [13] (CLOOK [13]) ,SSTF [13] based on seek time, SPTF [16, 24] and its variations, GSTF (Grouped Shortest Time First) [24] ,WSTF (Weighted Shortest Time First) [24] based on rotational latency, and algorithms that target real time applications such as EDF (Earlier Deadline First) or P-SCAN (Priority SCAN).

I/O Caches

I/O caches are divided into cache lines where data are stored. A cache segment is a group of cache lines used to store sequential data. The number of segments as well as the cache size are crucial for system overall performance. Usually, sequential I/O requests will be serviced by a single segment. Essentially, each segment functions as a small, private cache for a sequential stream. Therefore the number of segments is important for I/O performance.

Of equal importance is the size of segments. Segments need to be large enough to accommodate large portions of frequently used data. But since there is a fixed cache size, increasing the number of segments results in smaller size for each segment. To mitigate this side effect modern disk drives use adaptive segmented caches, namely caches that adapt the size and number of their segments in proportion with the number of I/O streams.

Data are fetched into segments when they are requested, or as a result of prefetching (readahead). Prefetching of data is based on the spatial and temporal locality of requests and may be very effective, if future requests are predicted correctly. On the contrary, if prefetching fails, it will induce several significant overheads. Prefetched data could congest the I/O system and at the same time pollute the cache with unused data.

Since readahead has the potential to increase I/O efficiency, several caches

implement nowadays prefetching based on their I/O access pattern. However, prefetching strategies differ, depending on caches location over the I/O path. Most caches avoid aggressive prefetching and they usually function based on two predefined values about readahead; minimum readahead and maximum readahead. Minimum readahead is usually employed when caches perform readahead on cache misses. On the other hand maximum readahead is employed when sequentiality has been detected. Maximum readahead ahead value is usually slightly smaller than the segment size used.

I/O path stages

The I/O stack in the operating system kernel usually queues I/O requests in device queues and is also able to cache I/O requests in a system-wide cache, called the buffer cache in the Linux kernel. Disk controllers and disks have their own queues and caches. The system kernel can have practically unlimited queue sizes and a fairly large system cache[1]. This is especially true today, with dropping memory cost and the ability to attach large amounts of memory to commodity systems.

Multi-device disk controllers also employ queuing of requests, but unlike device drivers, where the queue length can be considered unbounded, the queues in the devices controllers have a finite maximum size. However, controllers have a better knowledge about the disk mechanisms they host and therefore they may provide better scheduling decisions. Also, modern controllers are establishing the use of cache which is about one order of magnitude larger than the corresponding caches on disks, and is usually shared among all disks.

Commodity disk controllers have small amounts of caches, in the order of 8-16 MBytes. The main components of a disk are the magnetic medium and a controller attached to it. The controller usually includes a queue for scheduling purposes. The

[1]The OS kernel usually employees multiple I/O queues, however, for the purpose of our work we can view these as a single queue.

disk's internal controller has detailed information about the internals of the device, such as zone boundaries, LBN to PBN translation, defective sectors, that allows more sophisticated and accurate policies on queuing. Commodity disks have small queue sizes in the order of a few entries. Disk caches usually consist of a single memory chip to reduce system components and thus, mean time to failure of the whole disk. Thus, it has limited size, which does not exceed a few MBytes.

The disk cache is used for three purposes: as a speed-matching buffer for staging data and balancing speeds between the mechanical components (platters/heads) and the digital disk interface, as a readahead buffer for prefetching data to increase disk efficiency by reducing head seek time, and (possibly) as a cache to reduce accesses to disk. The disk cache is usually divided in a number of cache segments, which are memory chunks used to hold contiguous data, similar to cache lines in traditional caches. Nowadays disk cache capacity is usually a small fraction (0.1% - 0.3%) of the overall disk drive capacity.

2.2 Related Work

Previous work for improving the performance of disks can be categorized in (a) modeling disk systems and (b) disk optimizations.

Activity in (a) has focused on taking into account detailed features of modern disk drives and producing models and simulators for workload driven evaluation [16, 22, 26, 11]. The authors in [21] provide detailed information about disk access patterns on Unix-based systems. Finally, [28] discusses how disk parameters can be extracted using high-level experiments.

In (b), research efforts include either new scheduling algorithms and methods or new techniques devised to improve several aspects of I/O performance [15, 2, 25]. The authors in [14] provide an evaluation of I/O optimization techniques, such as caching, prefetching, write buffering, and scheduling. They use various workloads

and system configurations and quantify the impact of each optimization. In our work we focus on large numbers of sequential streams and provide a solution based on prefetching that is able to adapt to the amount of memory available in storage nodes. The authors in [4] propose a disk-level optimization to improving system throughput. The proposed method requires modifications to the actual disks.

Closer to our research is [27], where the authors describe how I/O performance is affected by sequential read requests. Similarly, [31] uses Disksim to explore if large disk caches can improve system performance. Both papers also explore the effect of read-ahead and segmented caches on I/O performance. However, our work also proposes and examines through implementation on a real system the effectiveness of host-level solutions that are motivated by changing storage system architectures.

The authors in [10, 18] examine various parameters that affect sequential I/O performance in the Windows operating system. They find that achieving high performance requires using large requests, asynchronous and direct I/O, and multiple outstanding requests. In our work, we use similar parameters in our base system and propose a technique that further improves performance in the presence of large number of sequential streams.

Work in storage caches [29, 6, 30, 17] aims at improving the performance of workloads that do not have sequential access patterns by minimizing disk accesses. In our work we reduce disk seek overheads for a specific, but important type of workloads.

Previous work has examined issues related to the performance of I/O subsystems for multimedia applications [20, 5, 8, 19]. Unlike our work, the focus is on directly attache storage systems and on techniques related to the storage disks and controllers themselves. The authors in [7] examine how caching and buffering in a Video server can improve response time and system cost for video streams. They

also propose an inter-stream caching method. In contrast, our work we examine how prefetching and buffering can provide adaptive solution to improving disk utilization and system throughput with dedicated CPU and memory resources (in the form of storage nodes).

Chapter 3

Disksim Analysis

In this section we present a detailed analysis of the problem using architectural simulation exploring possible solutions.

Figures 1.1 show how I/O throughput degrades when multiple sequential streams are serviced by a single disk head, due to the increased seek overheads. Traditionally, scheduling and caching have been used in the various stages of the I/O hierarchy to improve data access patterns and thus, reduce seek overhead. Next we examine how various parameters of the I/O hierarchy affect I/O throughput for large numbers of sequential streams.

3.1 Methodology

This section describes the methods and the tools used to conduct our experiments and to collect the results. The approach we adopt is to use a detailed simulator such as Disksim. Disksim is a highly parameterized disk simulator which is capable of modelling a large number of storage system configurations, including buses, intermediate controllers, queues and caches. But despite the fact that Disksim comes with many disk models, unfortunately these disk models are obsolete. For this reason,

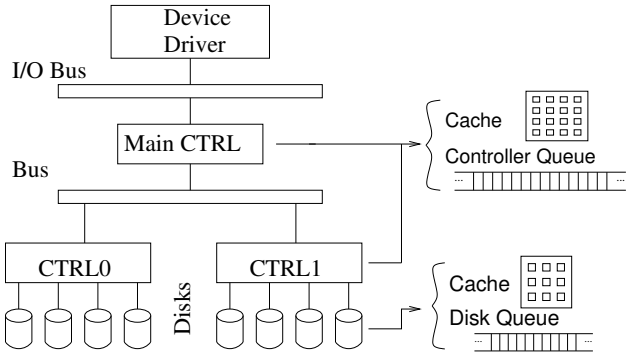


Figure 3.1: A sample configuration consisting of a main controller hosting two more controllers with four disks attached to each of them.

we make several changes to the parameters used for modelling the disks, to better model modern disks.

We examine three I/O hierarchies, a base configuration with a single controller and disk drive, a medium one with two controllers and eight disks, and a larger-scale configuration with sixteen controllers, hosting up to four disks controller, all attached to a single host. Figure 3.1 shows one of these configurations and the queues and caches we assume at each system component.

To generate the required workloads we use a set of request generators. Each generator issues I/O requests for a single, sequential stream with the parameters shown in Table 3.1. Each generator essentially issues synchronous, sequential reads and can be considered as a different stream originating from a different client (thread) that executes sequential reads on different disk areas. In our experiments we vary the number of generators (streams) and the request size.

3.2 System Parameters

In this work we are interested in examining system parameters that when tuned in commodity disks they can have a significant impact on I/O performance. Given that we are concerned with multiple sequential streams system throughput may be

Parameter	Value
Storage capacity per device	156301488
devices	target_disk
Blocking factor	8
Probability of sequential access	1.0
Probability of local access	0.0
Probability of read access	1.0
Probability of time-critical request	1.0
Probability of time-limited request	0.0
Time-limited think times	[normal, 0.0, 0.0]
General inter-arrival times	[normal , 0.0, 0.0]
Sequential inter-arrival times	[normal , 0.0, 0.0]
Local inter-arrival times	[normal, 0.0, 0.0]
Local distances	[normal, 0.0, 40000.0]
Sizes	[uniform, req_size, req_size]

Table 3.1: Parameters of a Generator.

affected by caching and prefetching. The system parameters that are associated with caching and prefetching are:

- Controller cache size, cache line size, and maximum queue size
- Disk number and size of segments and maximum queue size

The maximum queue length on the controller has little effect on I/O throughput for the workloads under consideration. To determine this we examine two scenarios, based on the number of segments of the disk cache.

First, we examine the effects of the maximum controller queue length when the number of segments in the disk cache is larger than the number of streams. Figure 3.2 shows that when maximum queue length is smaller than the number of streams, the throughput is slightly affected. This is because not all streams issue the same number of requests, and as the maximum queue length increases, it creates a slight unbalanced. On the other hand when the maximum queue size

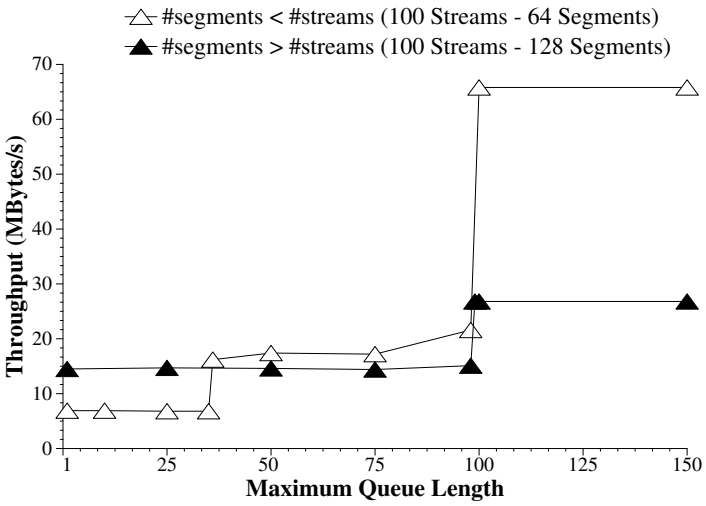


Figure 3.2: Effect of controller maximum queue size.

exceeds the number of streams then we have a sharp increase in throughput. This is due to scheduling decisions, that give preference to requests of a single sequential stream, while other streams possibly suffer from starvation; Immediately after the disk scheduler has served a request from a particular stream, the same stream is able to issue its next request, which due to the sequential access pattern in each stream, gets serviced by the scheduling algorithm (CYCLE-LBN) before other requests from different streams.

Second, similar conclusions can be reached for the effect of the maximum queue length in the controller when the number of streams is greater than the number of segments (Figure 3.2). When the maximum queue length is greater than the number of (#streams - #segments) no major fluctuations of throughput are noticed. But this not the case when the maximum queue length starts to exceed the number of (#streams - #segments). This does occur, because a number of streams equal to the number of segments issues a disproportional number of requests compared to other streams, and since these requests are handled by the disk cache, throughput increases. When maximum queue length exceeds the number of streams,

results are similar as above, due to disk scheduling effects.

Thus, the only effect of the controller queue size is at border conditions, when it results in giving preference to requests from a single stream and there is no effect on system throughput when all streams are serviced concurrently.

Similarly, we have examined the impact of disk queue size on system throughput and we have found that it has no effect on system throughput for large number of streams. In the rest of our experiments we set the controller maximum queue length to the number of streams. Next we examine the impact of the rest of the parameters under consideration, all related to disks, to I/O throughput.

3.2.1 Effect of prefetching

As shown in Figure 1.1 system throughput reduces significantly when the number of sequential streams increases, due to increased seek overheads. Increasing workload request size can address this issue. Figure 3.3 shows that throughput generally improves, as workload request size increases and there is enough buffer space (cache) to service all available streams. We use a disk cache size of 8 MBytes and tune segment size and readahead for the disk cache to be equal to the request size. This ensures that no prefetching takes place and our measurements depend only on request size. To keep disk cache size fixed to 8 MBytes we adjust (reduce) the number of segments as segment size increases. However, when the number of streams and the request size increase and the total available disk cache is not able to service concurrent requests from all streams, throughput degrades dramatically.

Although in many applications request size is fixed, employing prefetching and caching in the storage subsystem can have a similar effect and have been used traditionally in storage subsystems to reduce such overheads. When using sequential access patterns, prefetching can effectively increase the transfer size and thus, reduce seek overheads. On the other hand, caching has no real impact, since each block

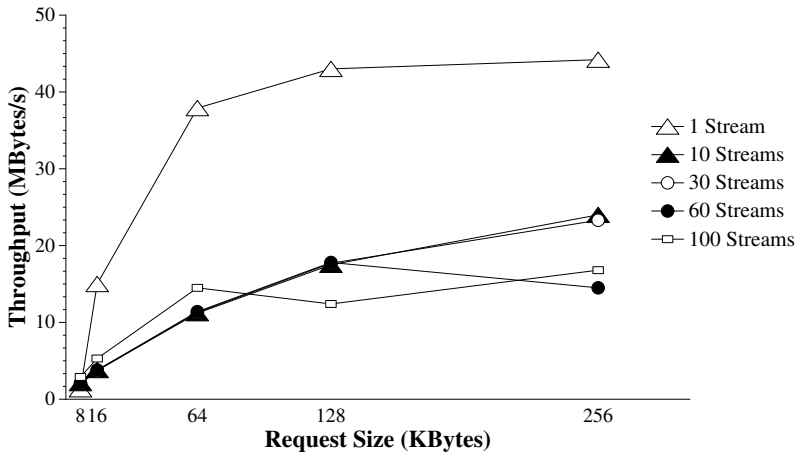


Figure 3.3: Impact of request size on throughput.

of data is used only once. However, prefetching in the disk subsystem is performed using memory available for caching (as opposed to separate buffers) and thus, the ability to prefetch is related to disk and controller cache configuration. In this section we examine the impact of the related parameters.

Figure 3.4 shows the effect of disk prefetching on overall throughput, in a configuration with 30 sequential streams on a single disk. In this experiment we vary the size of the disk segment keeping the total number of segments fixed to 32. As a result, disk cache size increases as well. We see that as segment size increases, disk throughput improves dramatically from about 8 MBytes/s for 32 KBytes segment size to about 40 MBytes/s for 2 MBytes segment size.

Next we examine the effect of prefetching when keeping the disk cache size fixed. We increase the size of disk segments, reducing at the same time the number of segments to maintain the total cache size fixed to 8 MBytes. Figure 3.5 shows that as disk segment size increases, throughput improves when the number of segments is greater than the number of streams. However, when the number of streams is greater than the number of segments, throughput drops dramatically. Moreover, in this case, large prefetch size has an undesirable effect; The disk starts reclaiming

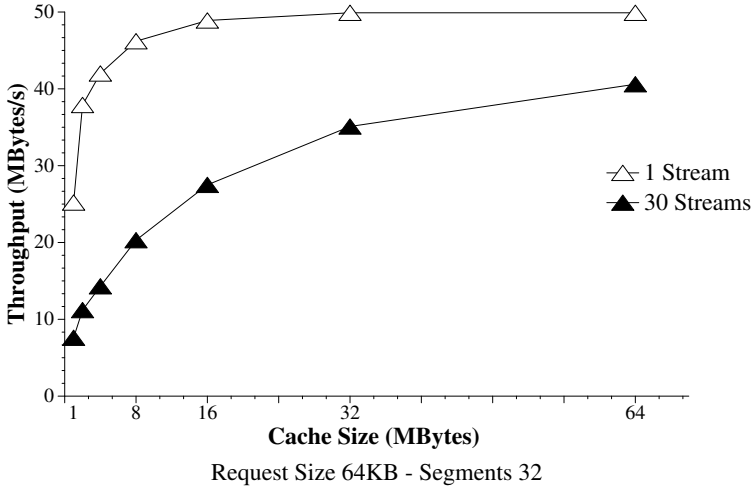


Figure 3.4: Effect of prefetching on throughput with increasing disk segment size. The number of segments is fixed to 32 (cache size increases as segment size increases).

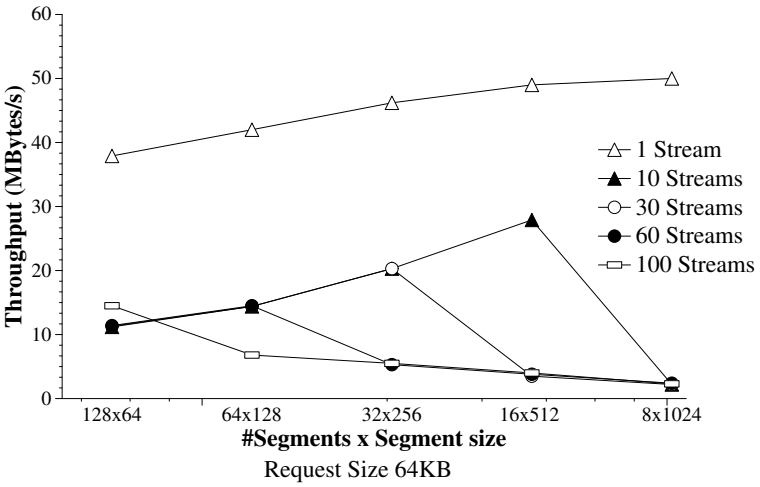


Figure 3.5: Effect of readahead on throughput. Disk cache size is 8 MBytes.

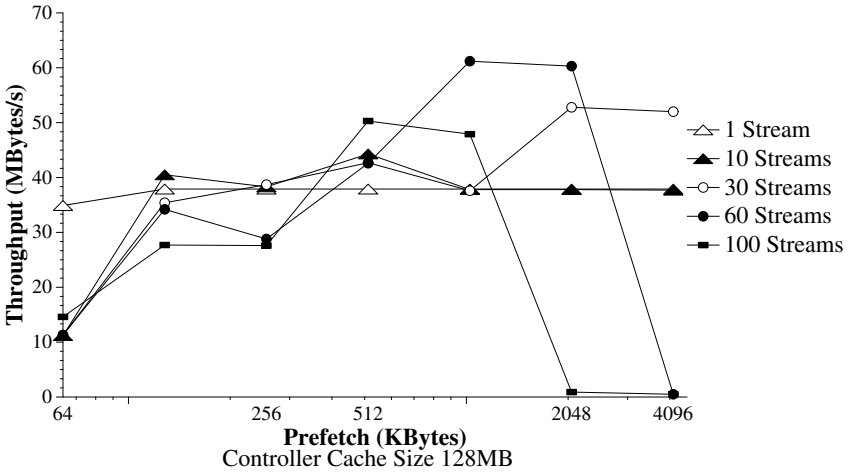


Figure 3.6: Prefetching on controller.

segments that were allocated to other streams, even though the full amount of data that have been prefetched into each segments has not been used by the corresponding stream. This results in worse throughput, even compared to when no (or little) prefetching is used.

Finally, we examine the impact of prefetching at the controller level. Figure 3.6 shows that prefetching at the controller helps improve throughput significantly. First, with one stream throughput is constant at about 35-40 MBytes/s regardless of controller readahead size. At 10 streams, even small readahead sizes increase throughput from about 10 to about 40 MBytes/s. Similarly at higher stream numbers, readahead has a significant impact, and for 60 streams a readahead of 1 MByte results in almost maximum disk throughput. However, when readahead increases and the controller does not have enough memory for all streams performance drops dramatically. For instance, at 4 MBytes readahead, throughput for 60 and 100 streams drops almost to 0 MBytes/s. Thus, prefetching at the controller level can help, however, would require large amounts of memory for large numbers of streams. This is particularly problematic if we consider that a commodity controller is usually able to host 4-16 disks, thus, requiring readahead buffers for thousands of streams.

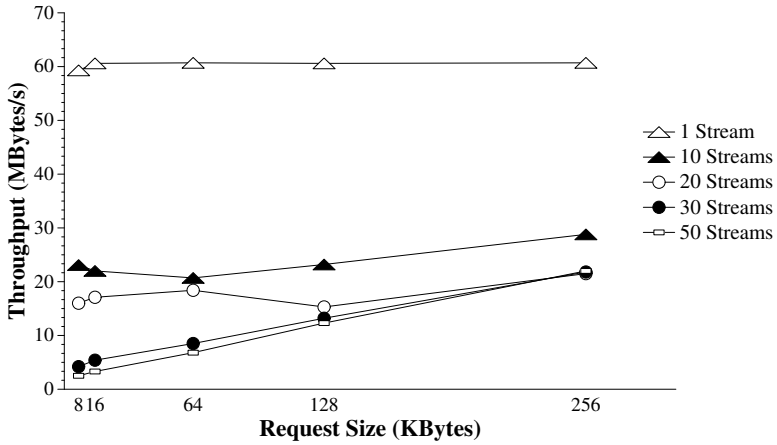


Figure 3.7: Xdd throughput with a single disk.

3.3 Xdd Measurements - Validation

To verify that our simulation results are close to realistic systems, we conduct an experiment on a real setup. We use the `xdd` microbenchmark [1] on a system equipped with one SATA controller and one disk, and running Linux Fedora core 3, with kernel version 2.6.11. We examine throughput for different request sizes (8 - 256KBytes) for a variable number of streams (1 - 50)[1]. Streams access the disk at 1 GByte intervals. We see (Figure 3.7) that throughput deteriorates as the number of sequential streams increases, similar to our simulation results in Figure 3.3. Note that for small requests with xdd we achieve a relatively high throughput, compared to Figure 3.3. The explanation for this fact may reside on the disks cache readahead policy. This effect seems to degenerate as the number of streams increases and the disk cache (and segments) cannot accommodate data from all streams.

[1]We perform these measurements with direct I/O to bypass the system buffer cache.

3.4 Summary

In summary, we see that increasing prefetch size at the disk or controller level and ensuring that all streams can be serviced by the available memory, results in significant improvements in system throughput. However, this approach is not practical, because disks subsystems are designed for multiple workloads and not tailored to the specific workload we examine in this work. Thus, we examine next how we can address this issue at the storage node level, and independent of the types of disks and controllers used.

Chapter 4

System Design and Implementation

In this section we present a more practical host-level solution that supports efficiently large numbers of sequential streams. Our design automatically detects sequential streams, separates them from other I/O operations, and schedules them appropriately to the disks to maximize disk utilization without significantly hurting I/O response time. We also examine how our scheduler can automatically adjust to disk and storage node characteristics making it possible to support storage nodes of varying technologies.

Figure 4.1 shows the overall system architecture. Each storage node runs a server that receives requests from multiple streams and manages all requests issued to local disks. When a request arrives at the storage node, it is forwarded to the classifier module, which is responsible for detecting sequential streams. Sequential streams are handled by our scheduler, which is responsible for issuing I/O requests to the disks. Sequential streams that have been handled by the scheduler and have prefetched data in memory, are placed in a buffered set, until their buffers are empty and can be deallocated. Next, we describe in detail the following three aspects of

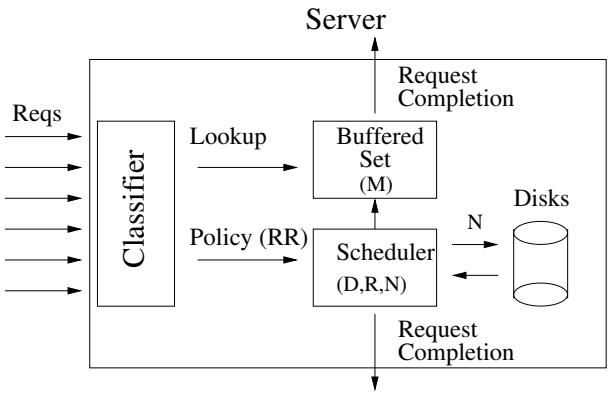


Figure 4.1: Path for processing I/O requests.

our design:

- Classification of requests to sequential streams
- Dispatching requests to disks
- Staging prefetched data to memory

4.1 Request classification

The classifier categorizes requests in two categories: Requests that belong in sequential streams and are kept in separate queues (one per stream) and non-sequential requests that are sent directly to the disk. To classify requests to sequential streams we use a bitmap to identify requests to neighboring blocks. Bitmap represents a consecutive set of blocks on the disk, with each bit referring to a single block.

However, given increasing disk capacities a single bitmap may require a large amount of memory, when each bit refers to a single physical block. There are two options to reducing the size of the bitmap. Either allow each bit to represent blocks of larger sizes or allocate smaller bitmaps for disk regions dynamically, as requests arrive at the storage node. Since representing large blocks with a single bit may

have an impact on the precision of detecting sequential accesses we choose the second approach.

When a request arrives, we examine its disk device number and block address. We allocate a small bitmap that represents the blocks around the block accessed by the request. For instance, if the request block number is B, the bitmap will represent consecutive blocks in the range $[B - offset, B + offset]$. The value of $offset$ affects the size of the bitmaps and the memory they require. In our experiments we find that a small value for $offset$, in the order of a few tens, is adequate for the experiments we perform. This may become more important when trying to cope with near-sequential streams, which however, is beyond the scope of this work. For each subsequent request that arrives in the range represented by a bitmap we set the corresponding bit. If the request size spans more than one blocks then all bits will be set.

On arrival of a request, we examine its corresponding bitmap for other requests that may have been issued to the the same range during the last time interval. If the number of set bits exceeds a threshold, we conclude that we have detected a sequential stream and we enable read-ahead allocating for this stream a private request queue. If the number of distinct requests in the range under consideration is not high enough, requests are issued directly to the disk. This mechanism ignores out of order requests, multiple requests to the same block, and only takes into account proximity in time.

4.2 Dispatching requests to disks

Once the classifier has created a queue for the requests that belong to a single stream, this stream may be passed to the scheduler. The number of streams that are handled by the scheduler at each point in time is called the *dispatch* set D, and is a parameter that can be set explicitly. The maximum number of streams in the

dispatch set is limited by the amount of memory M that is available for buffering in the storage node (M can also be set explicitly).

When one stream is placed in the dispatch set, the system allocates I/O buffers for this stream and uses it to generate disk requests. Each request generated is of size *read-ahead* R, which is independent of the actual request size in the sequential stream. Thus, the size and maximum number of outstanding requests to disks depends on the size of the dispatch set (values R and D respectively).

Since the number of sequential streams is usually higher than the desirable size of the dispatch set, we use a policy to replace streams in the dispatch set. Each stream remains in the dispatch set until it has issued N requests to the disk. Then it is replaced from the next (round-robin) sequential stream. Although, other more involved policies are possible, their benefits are not clear, given the fact that issued requests usually have large sizes. However, in cases, where storage nodes do not support large amounts of memory dedicated to buffers, and thus I/O buffers (and readahead) may be smaller, different scheduling policies may result in increased throughput. For instance, depending on the available buffering capacity of the storage node, it may improve overall throughput if streams are placed in the dispatch set based on their initial offset, trying to keep streams that access nearby areas of the disk in the dispatch set.

The scheduler issues asynchronous requests to the disks, using direct I/O to avoid (a) blocking on a single request when multiple disks are present on the storage node and (b) additional I/O buffer management in the OS kernel. Direct I/O bypasses the kernel buffering of data and reduces CPU overhead by moving data directly between user space buffer and the device performing I/O without copying through kernel space. When a disk request completes, the completion path of the scheduler is invoked asynchronously. The scheduler completes all requests associated with the I/O buffer and responds to the client.

When the completion path of the scheduler examines which requests may be completed by a full I/O buffer, it may potentially need to complete a large number of requests. This is especially true if in between more disk requests complete, causing completions of more client requests. However, during this process no new requests are issued to the disks resulting possibly in disk under-utilization. For this reason, the completion path of the scheduler gives priority to the issue path starting with the classifier. Before completing a pending client request, the scheduler calls the classifier that examines if new requests have arrived, if they are sequential, and if they can be placed in a stream queue and issued to the disk.

4.3 Staging prefetched data to memory

When a stream is removed from the dispatch set, it is staged in memory (in the buffered set) until its prefetched data are either used by subsequent requests or a timeout expires. This staging aims at reducing impact of the scheduler on individual request response time. The size of the buffered set defines the overall system requirements in memory (M). Thus, in all configurations and during system operation, $M \geq D * R * N$.

When a new request arrives, the classifier examines if it can be serviced immediately by a stream in the buffered set. If this is the *last* request that corresponds to an I/O buffer in the buffered set, the stream is removed from the buffered set.

In essence, the dispatch and buffered sets are used to coalesce stream requests to longer disk requests, so that disk seek overhead is reduced. However, there is no guarantee that all requests mapped to a single I/O buffer will actually be part of a stream. Thus, a buffer may remain allocated to the buffered set, waiting for certain requests to arrive and read the data that it contains. To avoid such cases, we use a periodic thread that garbage collects I/O buffers allocated to streams that are inactive as well as hash entries and stream queues that although were classified as

sequential, they have not received a large number of sequential requests and thus, remain allocated.

4.4 Implementation issues

In our implementation we choose to use asynchronous I/O as opposed to multiple threads, since asynchronous I/O is a more lightweight mechanism, especially as the number of disks increases. Disk request completion is detected using a kernel signal that causes the storage server process to exit a select system call. After exiting the select system call, the server detects if it should execute the issue or completion path, based on the select return value. This mechanism allows the server to check all conditions with a single select system call.

Chapter 5

Experimental Results

In our experiments we use a storage node equipped with two CPUs (AMD Opteron 242) and 1 GByte of memory. The system has two PCI-X buses. We attach to each one of the PCI-X buses a SATA controller. The controller (Broadcom's BC4810 RAID) is an eight channel entry level Serial ATA (SATA) RAID controller that can achieve high performance up to 450 MBytes/sec, and therefore can host up to eight disks. Table 5.1 shows the characteristics of the disks we use. The operating system we use is Linux Fedora core 3 with Kernel version 2.6.11.

WD Caviar SE WD800JD Serial ATA	
Capacity	80 GB
Rotational Speed	7,200 RPM
Seek Time (Average)	8.90 ms
Buffer Cache	8 MB
Buffer to Host(SATA)	150 MBytes/s(Max)
Buffer to Disk	748 Mbits/sec (Max)

Table 5.1: Disk characteristics.

We emulate streams in our system by using multiple, separate client systems. Each client can emulate a large number of sequential streams issuing requests to a single storage node. The parameters of each stream are the destination disk and

offset, the number of requests, the size of requests, and the number of outstanding requests. Each client issues requests from all streams it emulates as soon as it receives a response, never exceeding the maximum number of outstanding I/Os. For each issued request, the client maintains a handle in a pending list. When the request completes all related structures are deallocated.

Finally, we measure throughput and response time for each stream of requests. To be as close as possible to the throughput seen by real applications, we calculate the throughput delivered from a disk by summing the throughput of all individual streams serviced by the disk. In our setup, clients communicate with storage nodes over 1 GBit/s Ethernet using TCP/IP. To ensure that the network we use is not a bottleneck, responses to and from storage nodes do not include the data of read/write requests.

To simplify exploration of parameters, we first examine the case where all streams staged in memory are also used to dispatch requests to disk (M = D * R * N) and then we allow the system to use a smaller set of streams for dispatching requests. In addition to M,R,D,N, we use S to denote the number of input streams. In our experiments we set the number of outstanding requests to 1 for each stream and we distribute the available streams equally on the full disk, thus each stream is placed $disksize/\#streams$ blocks away from the previous one.

5.1 Read-ahead (R)

First we examine the effect of readahead on disk throughput when the system has adequate memory to stage all input streams, thus in this experiment $M = S * R * N$ and $S = D$. When increasing readahead, depending on the number of streams that need to be serviced, the amount of memory required on the storage node may be substantial. For instance, if we use 100 sequential streams and the readahead is 800 MBytes, then the storage node needs 800 MBytes of buffer space to service all

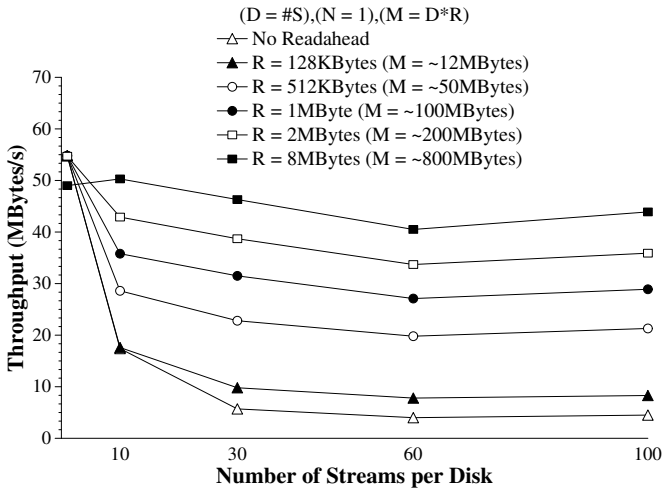


Figure 5.1: Effect of readahead. The storage node is configured with adequate memory to place all streams in the active set.

streams.

Figure 5.1 summarizes our results. We notice that readahead has a significant impact. Throughput increases significantly (about 20%) even when readahead increases from 2 to 8 MBytes for all numbers of streams. Moreover, with a readahead of about 8 MBytes, the low-cost SATA disks reach almost maximum utilization with a throughput of about 50 MBytes/s out of the maximum 55 MBytes/s we have measured in our experiments.

5.2 Memory size (M)

Next, we examine the effect of memory size on disk throughput. We are especially interested in the case where the number of streams is large and the available memory is not enough to accommodate the readahead data for all the streams. In this case only a subset of the streams will be staged in memory and will be used for dispatching at the same time, as limited by the available memory (M). Thus, we vary S, R, M whereas D is limited by (and always set to) $M/(R * N)$.

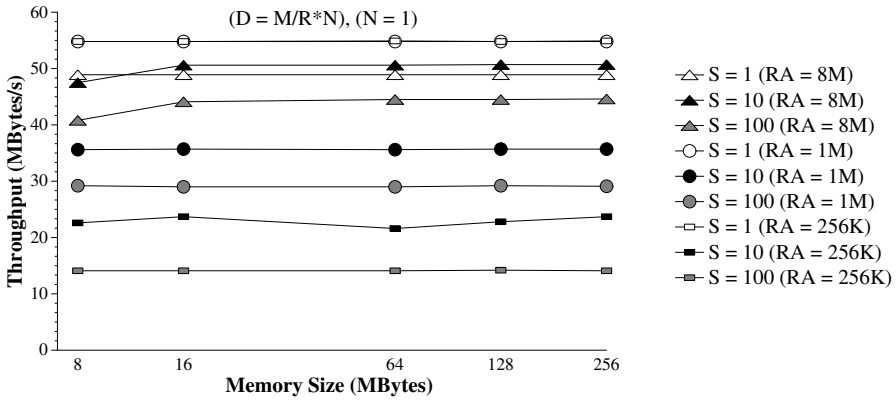


Figure 5.2: Effect of storage memory size on throughput. In each case, readahead is set to the indicated values (in MBytes).

Figure 5.2 shows our results. First, we notice that readahead does not have an effect when a single stream is used. Next, if we examine the curves for a fixed readahead, we see that increasing the number of streams results in lower throughput, almost independently of the amount of memory, when not all streams can be placed in the active set. For instance, with readahead of 8 MBytes, when the storage node has 16 MBytes of memory devoted to I/O buffering for sequential streams, only 2 streams can be placed in the active set. However, when increasing the number of streams from 10 to 100, system throughput reduces from about 50 MBytes/s to about 45 MBytes/s, a reduction of about 10%. This is due to the fact that now our streams cover a larger percent of the disk surface, and seek times are aggravated.

Next, we notice that increasing readahead is more important than increasing the number of streams that can be placed in the active set. Even when available memory is able to accommodate 1 stream with 8-MByte readahead, system throughput is higher (about 40 MBytes/s) than if we place all 100 streams streams in the active set with a readahead of 256 KBytes (about 14 MBytes/s).

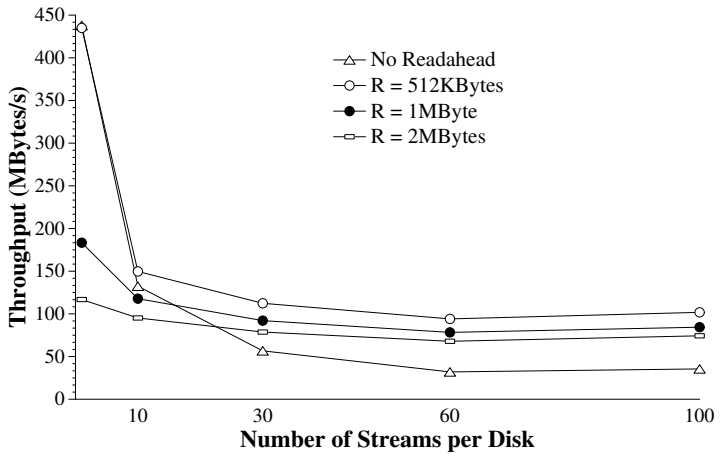


Figure 5.3: Throughput for an eight disk configuration.

5.3 Multiple Disks

Figure 5.3 shows how throughput scales as the number of disks in the system increases to eight. We note that throughput reduces significantly regardless of the readahead value. The reason is that with a large number of streams used for dispatching requests to disks, a single controller that hosts all eight disks needs to handle a large number of large I/O requests. Thus, although disk seek time is reduced due to the increased readahead, I/O throughput is significantly lower than the maximum possible of about 450 MBytes/s. Next, we examine how this can be addressed by disassociating the dispatched from the staged streams.

5.4 Dissacociating dispatching and staging

Reducing the number of streams that are used for dispatching requests to disks has the potential of significantly lowering memory overhead and buffer management in the I/O path.

Figure 5.4 shows system throughput with eight disks when the dispatch set is smaller than the number of staged streams. This results in much better behavior and

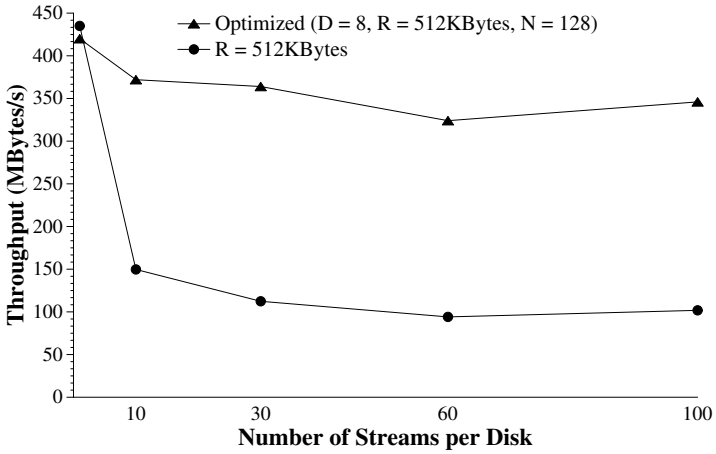


Figure 5.4: Figure compares throughput of our new implementation with the best case scenario (8MBytes readahead) of our previous implementation for a configuration of eight disks.

in an overall throughput of about 80% of the maximum available I/O throughput. Examining the amount of memory used dynamically in this experiment, we find that, although the number of staged streams is larger than the dispatched streams, in practice this difference is small, and the overall memory used is in the order of $D * R * N$.

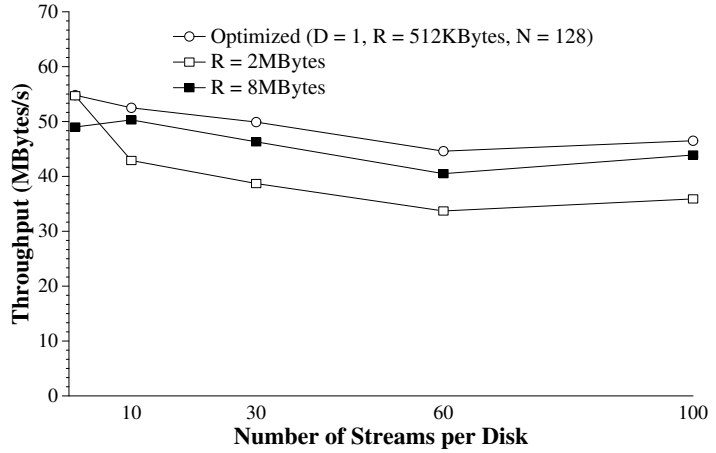


Figure 5.5: Single-disk throughput when dispatching a small number of streams.

Figure 5.5 shows throughput for a single disk configuration when using a small number of streams for dispatching. For this set of experiments we tune $R = 512$ KBytes, $D = \#disks = 1$, and $N = 128$. We see that compared to our previous results (where all streams staged are used for dispatching as well), there is a small improvement in performance, due to the lower buffer management overhead in the I/O path. Thus, we are able to achieve high utilization in fixed I/O system configurations by appropriately setting system parameters.

5.5 Impact on response time

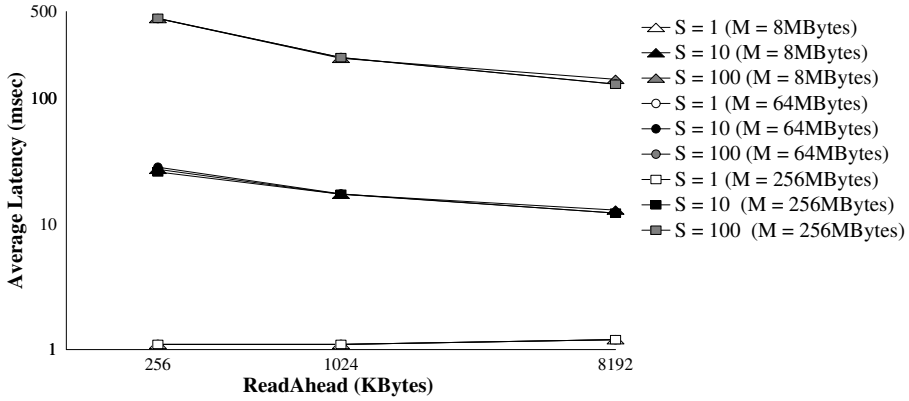


Figure 5.6: Average stream response time. Stream requests are 64 KBytes and each stream has 1 outstanding request.

Finally, although we are primarily interested in improving I/O subsystem utilization (throughput), we briefly examine the impact of various parameters on I/O response time. Figure 5.6 shows the average response time for different numbers of streams and readahead sizes, when we vary storage node memory. We should note that response time is measured on the client side and thus, includes all costs associated with the servicing a request and not only disk overhead. We see that increasing the number of streams has a significant impact on response time. Next, we observe that at a given number of streams, increasing readahead improves average

response time.

By examining the response times of individual streams we also find that average request response time for each stream does not differ significantly among streams. This is mainly due to the round-robin policy we use in placing streams in the active set. Within each stream, request response times can be divided in two broad categories: Requests that require disk I/O and requests that may be serviced directly from memory. When a large readahead is employed, most requests belong to the second category, reducing the average response time.

Chapter 6

Conclusions

In this work we discuss the impact of multiple sequential I/O streams on disk performance. We examine the effect or related I/O subsystem parameters through disk simulation and find that although certain parameters can significantly improve disk throughput, they are not under OS or application control in commodity subsystems. Next, we propose a solution at the host level that effectively coalesces sequential request patterns to large disk accesses and schedules them appropriately to maximize disk utilization. We implement our approach on a real system and we perform experiments with small- and medium- disk configurations. We find that our approach is able to improve disk throughput up to 8 times with 100 sequential streams and that it effectively makes the I/O subsystem insensitive to the number of I/O streams used. We also find that small amounts of storage node memory are adequate for our approach to work well and that request time is affected primarily by the number of streams being serviced and secondarily by readahead size.

Finally we believe that as more and more content will be stored and accessed online by an increasing number of applications and services, it is important to examine how the techniques we propose as well as similar techniques can be combined with techniques that aim at providing quality of service guarantees in the I/O sub-

system. Combining such techniques can result in significantly more cost-effective I/O subsystems for broad application domains.

Appendix A

Appendix

A.1 Modern Disks Drive

The disk mechanism consists of a number of platters that rotate around a spindle mechanism. Each platter can be described as a flat disk, with its two surfaces covered with a special media material used to store the data in the form of magnetic patterns. Each surface of the platter has an attached head used for read or write data. All hard disk read/write heads are attached to a single arm assembly (actuator) that moves the heads together.

In order to enable the organized storage and retrieval of the data, platter surfaces are divided into tracks, which are concentric circles with different diameters, that extend from the inner to the outer part of the platters surface. In turn tracks are divided into sectors. In most disk devices the size of sectors is 512 bytes, although there are devices that use larger sectors. The sector is the smallest individually addressable unit of information stored on a disk.

A cylinder comprises all the tracks that are located in the same position on each platter. That means that when a head for one surface is on a track, the heads for the other surfaces are also on the corresponding tracks that altogether

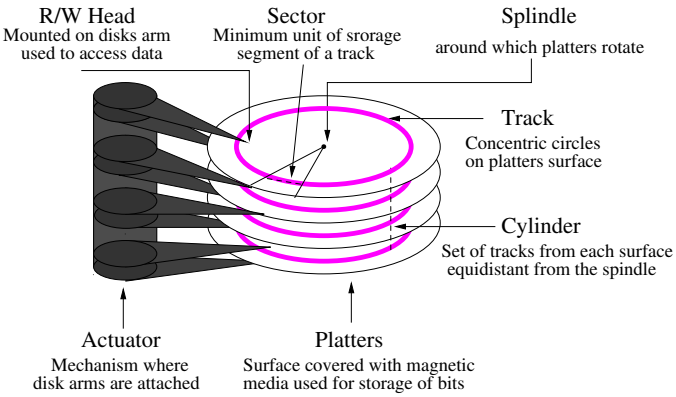


Figure A.1: Disk Overview

form the cylinder. The number of surfaces (or heads), cylinders, and sectors on a disk constitute the so called geometry of a hard disk.

In past the number of sectors was the same for all tracks, but nowadays modern hard disks put more sectors in outer tracks. This is called zoned recording. The reason is, that because the lengths of the tracks increase as you move from the inside tracks to the outside tracks, there would be a huge amount of wasted space between them if the longer tracks were limited to the same number of sectors as the shortest (innermost) tracks. Consequently the disk surface is partitioned into zones that spread over several cylinders and have a different number of sectors per track. Note that within each zone, all tracks have the same number of sectors, and that since the number of sectors increases as the radius of the cylinder increases, the outermost cylinder will have the most sectors (data).

A.1.1 Disk performance

This section describes the most common performance characteristics of modern disk drives, while providing some insight into how mechanical and electronic functions, affect hard drives performance.

When a disk services a request, in order to read or write data on the disk

surface it must position the read or write head to the proper place. The time needed to position the head to the proper track of data is called the seek time, while the further time needed to line up the right sector under the appropriate head is called the rotational latency. Both of them plus the command overhead are combined to produce the access time. Once the head is in the position to read or write, request is executed and the data are transferred to their destination. The time needed for this operation is called the transfer time. Access time and tranfer time are combined for the service time. The latter combined with the queueing delays presented, produce response time.

$$ServiceTime = AccessTime + TransferTime$$

$$ResponseTime = ServiceTime + QueueDelays$$

where:

$$AccessTime = SeekTime + RotationalLatency + CommandOverhead$$

Seek Time

Seek time is normally expressed in milliseconds, and compromise most of the disk access time, as it is dangled with mechanical limitations. Ruemmler and Wilkes describe a seek as being composed of four phases.

"A speed up", where the head is accelerated until it reaches half of the seek distance or a fixed velocity.

"A coast for long seeks", where the arm moves at its maximum velocity

"A slowdown", where the arm is brought to a rest close to the desired track

"A settle" where the disk controller adjusts the head to access the desired location

Since a significant portion of request service time may be consumed on seek time, the target is to minimize seek times by avoiding long seeks. Several techniques have been proposed based on that idea. Most of them introduce seek-reducing scheduling algorithms, while others focus on data placement techniques that favor sequential access over random. The average seek times for most modern drives today, is in the range of 8 to 10 ms, while a seek from the outermost cylinder to the innermost (known as full stroke) typically requires a time around 20 ms.

Rotational Latency

The other mechanical delay is rotational latency. On average, rotational latency will be half the time needed for a disk platter to complete a full rotation, and as a corollary rotational latency is dependent on drive's RPM. As before, several algorithms and techniques have been proposed to alleviate this problem, and many modern disk drives nowadays employ techniques such as head or cylinder skewing in order to minimize rotational latency when switching between consecutive heads or cylinders. The average rotational latency for modern drives, is in the range of 2 ms (for disks that are spinning with 15000RPM) to 4.2 ms (disks spinning with 7200RPM).

Command overhead

The command overhead refers to the time that elapses from when a command is issued to the hard disk until the disk starts to fulfill the command. Command overhead is generally very small and not highly variable between drive designs, it is generally around 0.5 ms for pretty much all modern drives.

Transfer Time

The transfer time is determined by the transfer rate.In turn the transfer rate is specified by the internal transfer rate and the external transfer rate.

The internal transfer rate refers to the speed that the disk can read data from its surface to its speed-matching buffer. Due to the zone recording the internal transfer is not constant across the entire surface of the disk, and can be specified as the $amountofdatapertrack/timeperrotation$. On the other hand the external transfer rate is specified from the speed at which those data are moved between the hard disk and the rest of the system. Nowadays the internal transfer rate is in the range of 80 Mbytes/sec, while the external transfer rate is in the range of 150 Mbytes/sec.

Bibliography

[1] Xdd manual. Technical report, I/O Performance Inc, 2005.

[2] E. Bachmat and J. Schindler. Analysis of methods for scheduling low priority disk drive tasks. In *SIGMETRICS '02: Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 55–65, New York, NY, USA, 2002. ACM Press.

[3] J. S. Bucy and G. R. Ganger. The disksim simulation environment version 3.0 reference manual. Technical Report CMU-CS-03-102, Carnegie Mellon University, School of Computer Science, January 2003.

[4] E. Carrera and R. Bianchini. Improving disk throughput in data-intensive servers, 2002.

[5] M.-S. Chen, D. Kandlur, and P. Yu. Support for fully interactive playout in disk-array-based video server. In *MULTIMEDIA '94: Proceedings of the second ACM international conference on Multimedia*, pages 391–398, New York, NY, USA, 1994. ACM Press.

[6] M. Dahlin, C. Mather, R. Wang, T. E. Anderson, and D. A. Patterson. A quantitative analysis of cache policies for scalable network file systems. In *Measurement and Modeling of Computer Systems*, pages 150–160, 1994.

[7] A. Dan, D. M. Dias, R. Mukherjee, D. Sitaram, and R. Tewari. Buffering and caching in large-scale video servers. In *COMPCON '95: Proceedings of the 40th IEEE Computer Society International Conference*, page 217, Washington, DC, USA, 1995. IEEE Computer Society.

[8] J. K. Dey-Sircar, J. D. Salehi, J. F. Kurose, and D. Towsley. Providing vcr capabilities in large-scale video servers. In *MULTIMEDIA '94: Proceedings of the second ACM international conference on Multimedia*, pages 25–32, New York, NY, USA, 1994. ACM Press.

[9] L. K. E. Coffman and B. Ryan. Analysis of scanning policies for reducing disk seek times. *SIAM Journal of Computing*, 1(3):269–279, 1972.

[10] C. v. I. Erik Riedel and J. Gray. A performance study of sequential i/o on windows nt(tm) 4. In *Proceedings of the 2nd USENIX Windows NT Symposium*, pages 1–10, 1998.

[11] G. R. Ganger. *System-Oriented Evaluation of I/O Subsystem Performance*. PhD thesis, University of Michigan, June 1995.

[12] R. Geist and S. Daniel. A continuum of disk scheduling algorithms. *ACM Transactions on Computer Scheduling*, 5(1):77–92, 1987.

[13] C. Gotlieb and G. MacEwen. Performance of movable-head disk storage devices. *Journal of the Association for Computing Machinery*, 20(4):604–623, 1973.

[14] W. Hsu and A. J. Smith. The performance impact of i/o optimizations and disk improvements. *IBM J. Res. Dev.*, 48(2):255–289, 2004.

[15] S. Iyer and P. Druschel. Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous i/o. In *Symposium on Operating Systems Principles*, pages 117–130, 2001.

[16] Jacobson and J. Wilkes. Disk scheduling algorithms based on rotational position. Technical Report HPL-CSP-91-7rev1, 2 1991.

[17] T. Johnson and D. Shasha. 2Q: a low overhead high performance buffer management replacement algorithm. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 439–450, Santiago, Chile, 1994.

[18] R. H. Leonard Chung, Jim Gray and B. Worthington. Windows 2000 disk io performance. Technical Report MSR-TR-2000-55, 7 2000.

[19] P. V. Rangan and H. M. Vin. Designing file systems for digital video and audio. In *SOSP '91: Proceedings of the thirteenth ACM symposium on Operating systems principles*, pages 81–94, New York, NY, USA, 1991. ACM Press.

[20] A. L. N. Reddy and J. C. Wyllie. I/o issues in a multimedia system. *Computer*, 27(3):69–74, 1994.

[21] C. Ruemmler and J. Wilkes. UNIX disk access patterns. In *Usenix Conference*, pages 405–420, Winter 1993.

[22] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[23] R. G. S. Daniel. V-scan: An adaptive disk scheduling algorithm. In *Proceedings of International Workshop on Computer Systems Organization*, pages 96–103, New Orleans, LA, 1983. IEEE.

[24] M. Seltzer, P. Chen, and J. Ousterhout. Disk scheduling revisited. In *Proceedings of the USENIX Winter 1990 Technical Conference*, pages 313–324, Berkeley, CA, 1990. USENIX Association.

[25] P. J. Shenoy and H. M. Vin. Cello: A disk scheduling framework for next generation operating systems. Technical Report CS-TR-97-27, 1, 1998.

[26] E. Shriver. *Performance modeling for realistic storage devices.* PhD thesis, New York University, May 1997.

[27] E. Shriver, C. Small, and K. A. Smith. Why does file system prefetching work? In *Proceedings of Usenix Annual Technical Conference*, pages 71–84, June 1999.

[28] B. L. Worthington, G. R. Ganger, Y. N. Patt, and J. Wilkes. On-line extraction of SCSI disk drive parameters. Technical Report CSE-TR-323-96, 19 1996.

[29] Y. Z. Zhifeng Chen and K. Li. Eviction based cache placement for storage caches. In *Proceedings of Usenix Technical Conference*, pages 269–282, 2003.

[30] Y. Zhou, J. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the General Track: 2002 USENIX Annual Technical Conference*, pages 91–104, Berkeley, CA, USA, 2001. USENIX Association.

[31] Y. Zhu and Y. Hu. Can large disk built-in caches really improve system performance? *SIGMETRICS Perform. Eval. Rev.*, 30(1):284–285, 2002.