

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

George Panagiotakis<sup>†</sup>, Michail D. Flouris, Angelos Bilas<sup>†</sup>  
Institute of Computer Science (ICS)  
Foundation for Research and Technology - Hellas (FORTH)  
N. Plastira 100, Vassilika Vouton, Heraklion, GR-70013, Greece  
{gpanagio, flouris, bilas}@ics.forth.gr

## Abstract

*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 requirements for more disks. 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 examine this tradeoff and provide an understanding of issues along with a practical solution. We use DiskSim, a detailed architectural simulator, to examine several aspects of a modern 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 dynamically 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 4 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.*

## 1. Introduction

Rich media content has become increasingly popular for entertainment, education, or business purposes. One of the main types of data access for media content is sequential access for storing and retrieving (large) I/O streams. Although media applications 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, besides being important, is also considered the simplest type of access for traditional disk subsystems, resulting in minimum head movement and thus, seek overhead.

However, disk subsystems that are able to scale to large numbers of sequential data streams present a challenge. As disk capacities increase, improving cost-effectiveness of disk subsystems requires servicing multiple sequential streams from as few disks as possible. For instance, disk capacities are currently more than 1 TByte for a single spindle. Such a disk could store hundreds or thousands of media streams. On the other hand, disk throughput has not been increasing at the same rate as capacity, limiting maximum (commodity) disk throughput to about 100 MBytes/s.

Thus, building a storage system that is required to service large numbers of I/O streams results in a tradeoff on how many streams should be serviced by each disk to sustain the required stream throughput without increasing the number of disks in the subsystem. For instance, if an application requires streams of 1 MByte/s, then a disk with a maximum throughput of 50 MBytes/s could sustain 50 streams. In practice, a much smaller number of streams can be serviced per disk by the system, requiring more disks to service the workload. Figure 1 shows how disk throughput degrades in a system when multiple sequential streams are serviced by the same disk. We see that as the number of streams increases, throughput drops dramatically, by a factor of 2-5.

This problem is exacerbated by the fact that it is difficult to address it through static data placement. Stream playout is generally short-lived compared to the time it takes to reorganize data on a disk. Thus, optimizing data placement for a given workload to reduce head movement is not practical. Finally, given the diversity of commodity disks and I/O subsystems, it is usually difficult to tune an application or service for different subsystems, requiring a system-independent approach to addressing this problem.

Mechanisms commonly used by modern operating systems and filesystems to deal with these issues are: (i) caching and (aggressive) prefetching of data in the system's buffer cache in order to sustain disk throughput, amortize the cost of an I/O operation over a large amount of data, as well as

<sup>†</sup> Also with the Department of Computer Science, University of Crete, P.O. Box 2208, Heraklion, GR 71409, Greece.

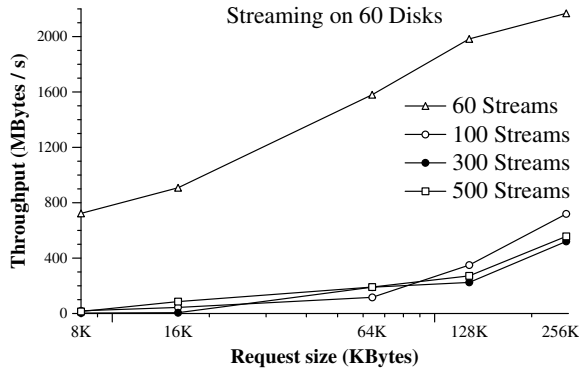


Figure 1. Throughput collapse for multiple sequential streams for a 60 disk setup.

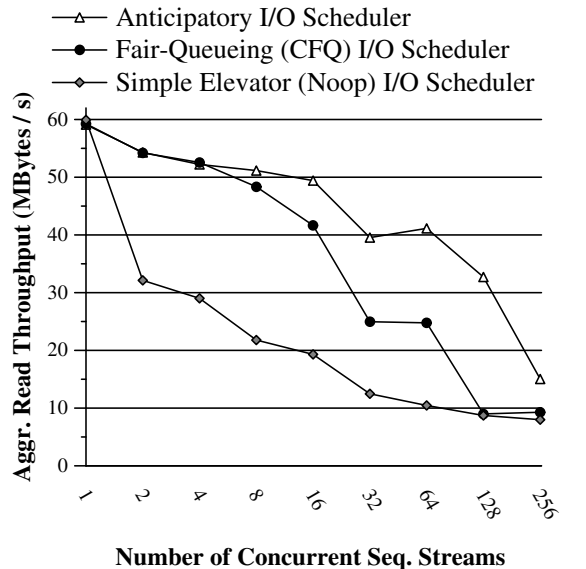
minimize I/O wait time in the application [4], [19], [20], [22], [30], [34] and (ii) using smart I/O schedulers, such as the “anticipatory” scheduler [15], which detects sequential accesses within a window of time, servicing them as a single large read. These mechanisms solve the issues of I/O stalls and concurrent accesses for a small number of streams, however they are not efficient in a large number of sequential streams, especially when the amount of data being read exceeds the size of the buffer cache.

Figure 2 demonstrates this behavior in a Linux system, comparing the performance of Linux I/O schedulers over a single disk. We have used the *xdd* benchmark on the standard Ext3 filesystem. The buffer cache is enabled, and the machine used is identical to the one described in section 5. As shown, when the number of streams exceeds 16, all schedulers perform significantly slower. In the case of 256 streams, even the performance of the anticipatory scheduler is reduced by a factor of 4.

In this paper we provide an analysis of the problem and a solution that takes advantage of modern storage node trends. We use both DiskSim [3], a detailed architectural simulator, and real system implementation. First, we examine the impact of I/O parameters on disk throughput for various configurations. We show that statically configuring certain disk and controller parameters could address the problem. However, these parameters are out of the control of the host operating system and the application and, due to the diversity required at the storage subsystem, it is unlikely that they will be tuned solely for sequential workloads.

Instead, we propose a host-level solution that sees all I/O requests and adjusts certain I/O parameters to improve disk throughput and thus, disk utilization. Our approach relies on (1) transparently 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.

We implement our solution at the user-level and experi-



Iozone on Ext3: Reading Sequential Files, 4KB Block

Figure 2. Linux I/O scheduler performance.

ment with both low- and high- disk throughput configurations, using eight commodity SATA disks and 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 4 times with 100 sequential streams per disk (800 total 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 response time is affected mostly by the number of streams, with read-ahead size having only a small negative impact.

The rest of this paper is organized as follows. First, Section 2 presents the necessary background. Section 3 presents a detailed analysis of the problem showing our simulation results. Section 4 discusses our proposed solution and our implementation and Section 5 shows our experimental results. Finally, Section 6 presents related work and Section 7 draws our conclusions.

## 2. Background

### 2.1. I/O path queuing and caching

Figure 3 shows the main stages of the I/O request path. In this work we are mostly 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. 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 data in a system-wide cache (buffer cache). Disk controllers and disks have their own queues and caches.

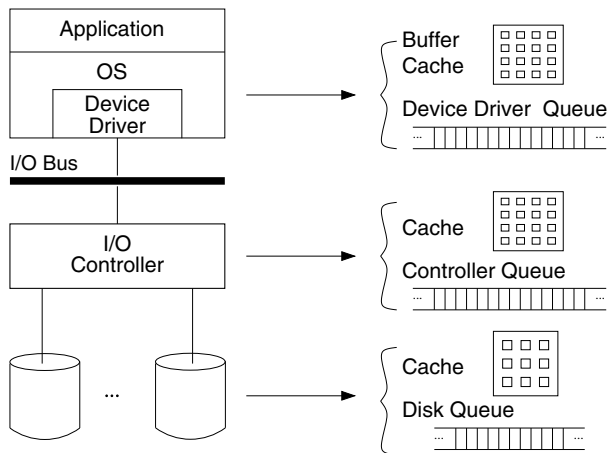


Figure 3. Components of the I/O path in storage nodes that perform queuing and caching.

The *system kernel* can have practically unlimited queue sizes<sup>1</sup> and a fairly large system cache. This is especially true today with dropping memory cost and the ability to attach large amounts of memory to commodity systems.

*Disk controllers* have smaller I/O queues in the order of 32-128 requests and caches that vary depending on the complexity and cost of the controller. For instance, mid-range controllers may have 64 MBytes of cache and may host 4 SATA disks. Commodity disk controllers have smaller amounts of cache, in the order of 4-16 MBytes and small I/O queues in the order of a few requests.

*Disks* have a queue themselves for scheduling purposes. The disk's controller has detailed information about the internals of the device, such as knowledge of zone boundaries, logical to physical block number translation, and defective sectors, which allows more sophisticated and accurate policies on queuing. However, commodity disks have small queue sizes in the order of a few entries and small caches.

The disk cache is used for three purposes: (i) As a speed-matching buffer for staging data and balancing speeds between the mechanical components (platters/heads) and the digital disk interface. (ii) As a read-ahead buffer for prefetching data to increase disk efficiency by reducing head seek time. (iii) Possibly, as a cache to reduce disk accesses.

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. The disk cache usually consists of a single memory chip to reduce system components and thus, mean time to failure of the whole disk. Thus, it has limited size that does not exceed a few MBytes. Nowadays disk cache capacity is usually a small fraction (0.05% - 0.1%) of the overall disk drive capacity.

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

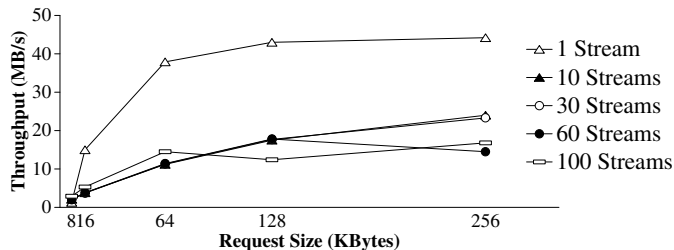


Figure 4. Impact of request size on throughput.

### 3. Problem Analysis

Traditionally, scheduling and caching have been used in various stages of the I/O hierarchy to improve data access patterns and thus, reduce seek overhead. In this section, we examine how various parameters of the I/O subsystem affect I/O throughput for large numbers of sequential streams. We use Disksim [3] and three I/O hierarchies: a base configuration with a single controller and disk drive, a medium-size configuration with two controllers and eight disks, and a large-size configuration with sixteen controllers hosting up to four disks each, all attached to a single host.

To generate the required workloads we use a set of request generators, each generating a single sequential stream of synchronous reads with a fixed size of 64 KBytes. Each generator represents 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.

We are interested in examining system parameters that when tuned in commodity disks they can have a significant impact on sequential I/O performance. Given that we are concerned with multiple, sequential streams there is no notion of temporal locality. However there is significant spatial locality making prefetching important. Prefetching in disks and controllers can happen by: (i) explicitly or implicitly increasing application request size, or (ii) increasing disk segment size and allowing the disk to prefetch full segments.

#### 3.1. Request size

Increasing workload request size can improve overall disk utilization. Figure 4 shows how throughput improves, as workload request size increases, assuming there is enough disk buffer space (cache) to service all available streams. We use a disk cache size of 8 MBytes and tune the segment size and read-ahead of 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. We see that when the number of streams and the request size increase beyond the available cache capacity, throughput degrades dramatically.

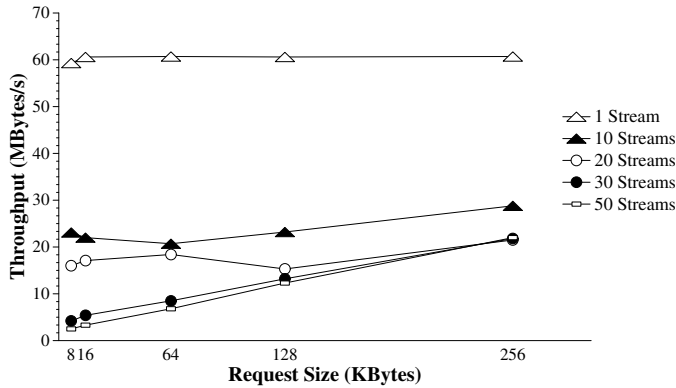


Figure 5. Xdd throughput with a single disk.

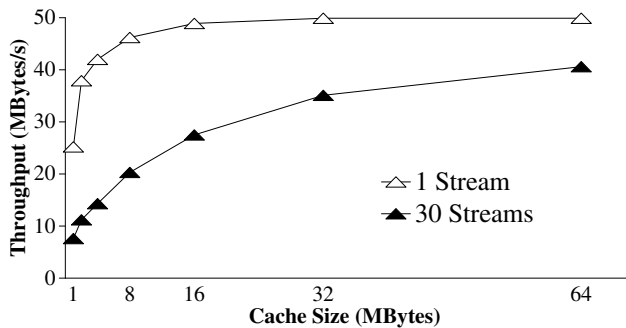


Figure 6. Effect of prefetching on throughput when increasing disk segment size.

To verify that our simulation results are close to realistic systems, we also conduct a similar experiment on a real setup. We use the `xdd` micro-benchmark [1] on a system equipped with one SATA controller and one disk, running Linux Fedora core 3 (kernel version 2.6.11)<sup>2</sup>. We examine throughput for different request sizes for a variable number of streams (1 - 50) that access the disk at 1 GByte intervals.

Figure 5 shows that throughput degrades as the number of sequential streams increases, similar to our simulation results in Figure 4. Note that for small requests with `xdd` we achieve a relatively high throughput, compared to Figure 4. The explanation for this fact lies on the disk's cache segment size, which in this experiment is fixed and cannot change (due to the fact that we use a real disk), unlike Figure 4 where we tune the segment size equal to the request size.

Figure 6 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 and using a request size of 64 KBytes. As a result, disk cache size increases as the segment size increases. We see that as segment size increases, disk throughput improves dramatically from about 8 MBytes/s for 32KBytes segment

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

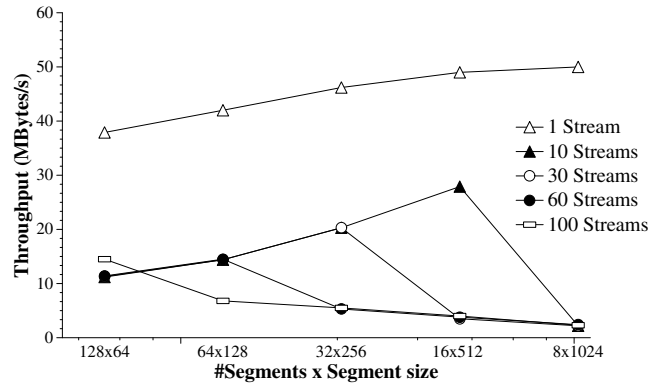


Figure 7. Effect of read-ahead on throughput.

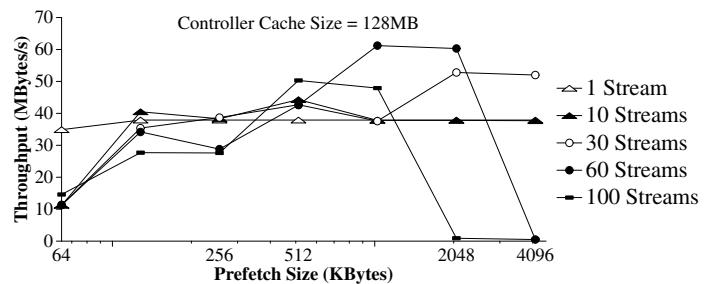


Figure 8. Prefetching at the controller level.

size to about 40 MBytes/s for 2MBytes 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 7 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 segments that were allocated to other streams, even though the full amount of data that have been prefetched into each segment has not been used by the corresponding stream. This results in worse throughput, even compared to using little or no prefetching.

Finally, we examine the impact of prefetching at the controller level. Figure 8 shows that prefetching at the controller level helps improve throughput significantly. First, with one stream, disk throughput is constant at about 35-40 MBytes/s regardless of controller read-ahead size. At 10 streams, even small read-ahead sizes increase throughput from about 10 to about 40 MBytes/s. Similarly at higher stream numbers, read-ahead has a significant impact and for 60 streams a read-ahead of 1 MByte results in almost maximum disk throughput. However, when read-ahead increases and the controller does not have enough memory for all streams

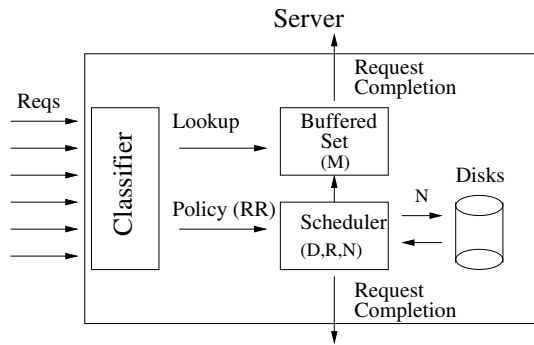


Figure 9. Path for processing I/O requests.

performance drops dramatically. For instance, at 4 MBytes read-ahead, throughput for 60 and 100 streams drops almost to 0 MBytes/s.

Thus, increasing request size or prefetching at the controller or disk level aggressively 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 large prefetch buffers for thousands of streams.

Moreover, 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.

## 4. System Design and Implementation

In this section we present a more practical host-level solution that supports efficiently large numbers of sequential streams. The basic idea is to automatically detect sequential streams, separate them from other I/O operations, and schedule them appropriately to disks to maximize disk utilization without significantly hurting I/O response time. We also examine how our system can automatically adjust to disk and storage node characteristics making it possible to support storage nodes of varying technologies and configurations.

Figure 9 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, 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 staged in a *buffered* set, until their buffers are empty and can be deallocated. Next, we describe in detail the following three aspects of our design: (i) Classification of requests to

sequential streams, (ii) Dispatching requests to disks, and (iii) 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. Each bitmap represents a consecutive set of blocks on the disk, with each bit referring to a single block.

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 dynamically allocate smaller bitmaps for disk regions, 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 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 efficiently detecting sequential streams. 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 block 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 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*

$D$ .  $D$  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 I/O buffering in the storage node.  $M$  may 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.

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 sequential stream, based on some policy. Involved policies are possible. 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. However, their benefits are not clear, given the fact that issued requests usually have large sizes, and for this reason we currently use a simple round-robin policy.

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

The scheduler issues asynchronous requests to disks using direct I/O. We choose to use asynchronous I/O as opposed to multiple threads to avoid blocking on a single request when multiple disks are present on the storage node. Asynchronous I/O is a more lightweight mechanism, especially as the number of disks increases. We use direct I/O to avoid 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 a user space buffer and the device performing I/O without copying through kernel space.

Finally, 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.

### 5. Experimental Results

In our experiments we use a storage node equipped with two CPUs (AMD Opteron 242) and 1 GByte of memory, running Fedora Core 3 Linux (kernel version 2.6.11). The system has two PCI-X buses, each hosting a SATA disk controller. The controller (Broadcom's BC4810 RAID) is an 8-channel, entry level Serial ATA (SATA) RAID controller that can achieve up to 450 MBytes/s and can host up to eight disks. The disks we use are WD Caviar SE WD800JD, with 80GByte capacity, rotating at 7,200 RPM, and with an average seek time of 8.9 ms as given by their manufacturer. The SATA interface maximum transfer rate is 150 MBytes/s. We measure the maximum disk throughput at the application level to be about 55-60 MBytes/s, thus, 8 disks are able to saturate the disk controllers.

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 ( $M = D * R * N$ ). 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. We set the number of outstanding requests to 1 for each stream and we distribute the available streams uniformly on the disks: Each stream is placed  $disksize/\#streams$  blocks away from the previous one. We use ‘\_’ to denote a value that is varied in a range. Finally, since we do not set explicitly the number of staged streams (this is rather limited by the amount of available memory) we do not use a specific parameter, but rather use the word *staged* in our figures.

### 5.1. Read-ahead (R)

First we examine the effect of  $R$  on disk throughput when the system has adequate memory to stage all input streams. In this experiment  $M = S * R * N$  and  $S = D$ . When increasing read-ahead, 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 read-ahead is 8 MBytes, then the storage node needs 800 MBytes of buffer space to service all streams.

Figure 10 summarizes our results. We notice that read-ahead has a significant impact. Throughput increases significantly (about 20%) even when read-ahead increases from 2 to 8 MBytes for all numbers of streams. Moreover, with a read-ahead 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.

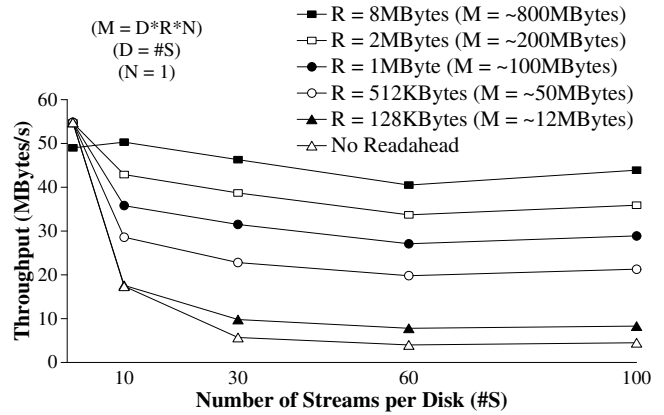


Figure 10. Effect of read-ahead. The storage node is configured with adequate memory to place all streams in the dispatch set.

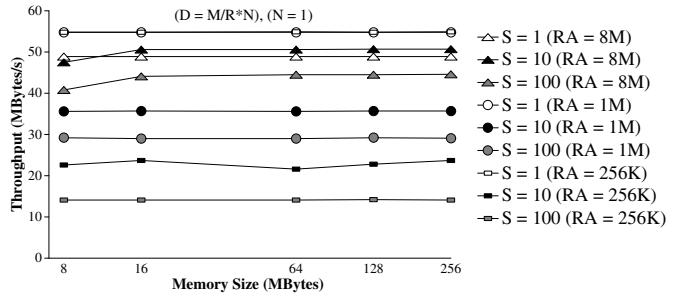


Figure 11. Effect of storage memory size on throughput. In each case, read-ahead is set to the indicated values (in MBytes).

### 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 read-ahead data for all 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 11 shows our results. First, we notice that read-ahead does not have an effect when a single stream is used. Next, if we examine the curves for a fixed value of  $R$ , 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 dispatch set. For instance, with read-ahead 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 dispatch set. However, when increasing the number of streams from 10 to 100, system throughput reduces from

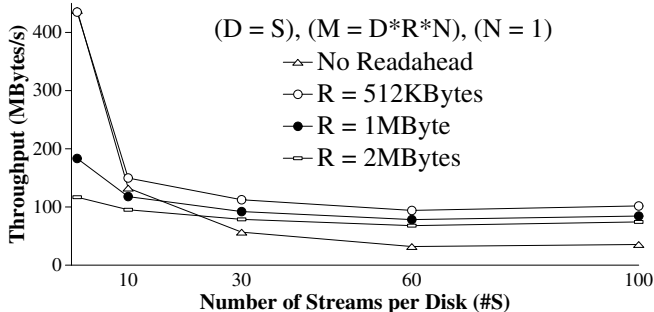


Figure 12. Throughput for an 8-disk setup.

about 50 MBytes/s to about 45 MBytes/s, a reduction of about 10%. This is due to the fact that in this case streams cover a larger percent of the disk surface, and seek times are aggravated.

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

### 5.3. Multiple disks

Figure 12 shows how throughput scales as the number of disks in the system increases to eight. We note that throughput reduces significantly regardless of the read-ahead 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 read-ahead, I/O throughput is significantly lower than the maximum possible of about 450 MBytes/s due to buffer management. Next, we examine how this can be addressed by disassociating the dispatched from the staged streams.

### 5.4. Disassociating 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 13 shows system throughput with eight disks when the dispatch set ( $D$ ) is smaller than the number of staged streams. This results in much better behavior and in an overall throughput of about 80% of the maximum available I/O throughput of about 450MBytes/s. 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

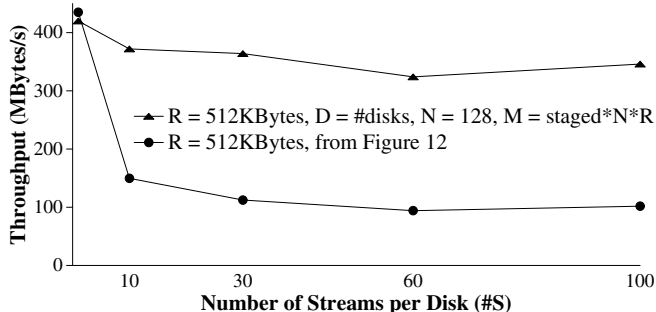


Figure 13. Throughput when fewer streams are dispatched than staged (8-disk setup).

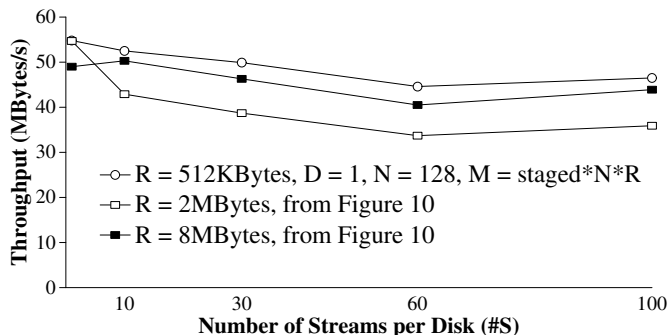


Figure 14. Single-disk throughput.

is small, and the overall memory used is in the order of  $D * R * N$ .

Figure 14 shows throughput for a single disk configuration when using a small number of streams for dispatching. We see that compared to our previous results (Figure 10), where all staged streams are used for dispatching as well, there is a small improvement in performance, due to the lower buffer management overhead. Thus, it is possible to achieve high utilization in different I/O subsystem configurations by appropriately setting parameters  $R$ ,  $D$ ,  $N$ , and  $M$ .

### 5.5. Impact on response time

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 15 shows the average response time for different numbers of streams and read-ahead sizes, when we vary storage node memory. Stream requests are 64 KBytes, memory size is shown in MBytes, and each stream has 1 outstanding request. 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 read-ahead improves average response time.

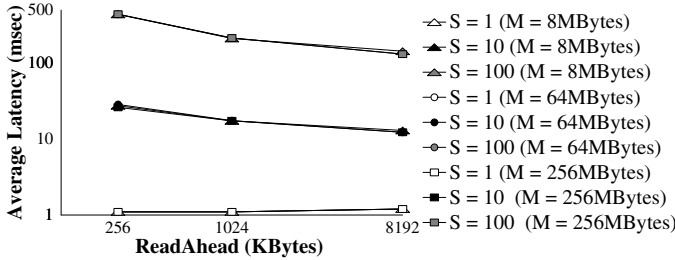


Figure 15. Average stream response time.

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 dispatch 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 read-ahead is employed and there is adequate memory for buffering, most requests belong to the second category, reducing the average response time.

## 6. Related Work

Related work for improving the performance of disks can be categorized in (a) modeling disk systems and (b) disk optimizations. Previous work has examined in detail features of modern disk drives and has produced models and simulators for workload driven evaluation [16], [27], [29], [13], [31]. Ruemmler et al. [26] provide detailed information about disk access patterns on Unix-based systems, whereas Worthington et al. [33] discuss how disk parameters can be extracted using high-level experiments.

Research efforts in I/O optimizations include both new scheduling algorithms and new techniques for improving several aspects of I/O performance [21], [15], [2], [28], [32], [18]. Hsu et al. [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. Carrera et al. [5] propose a disk-level optimization to improving system throughput, which however, requires modifications to the actual disks. Also, recently there has been renewed interest in performance and optimizations for parallel and I/O intensive applications [35], [9].

Closer to our research is Shriver et al. [30], which discusses how I/O performance is affected by sequential read requests and how prefetching amortizes the cost of an I/O operation over a large amount of data. However, we consider a much larger number of streams, where prefetching alone

is not effective. Similarly, Zhu et al. [37] use DiskSim to explore if large disk caches can improve system performance.

Prefetching and caching has also been studied in [4], [19], [20], [22], [34]. Cao et al. [4] and Kimbrel et al. [19] study integrated prefetching and caching policies for a single or multiple disks. However, such policies are focused on optimizing a small number of non-sequential streams in order to speed up applications. [22] presents a framework that exploits hints from the application for its future I/O accesses in order to adapt caching policies and perform informed read-ahead and prefetching, using a cost-benefit model. The proposed system aims at minimizing application runtime and not at maximizing disk throughput.

The authors in [25], [8] 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 improves performance in the presence of large number of sequential streams.

Work in storage caches [7], [10], [36], [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 [24], [6], [12], [23]. Unlike our work, the focus is on directly attached storage systems and on techniques related to the disks and controllers themselves. The authors in [11] 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, in 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.

## 7. 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. Our approach identifies the structures that are needed for this purpose, parameterizes each structure, and allows for setting these parameters (D, R, N, M) independently.

We implement our approach on a real system and we perform experiments with small- and medium- disk config-

urations. We find that our approach is able to improve disk throughput up to 4 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 read-ahead size.

## Acknowledgment

We would like to thank the members of the CARV laboratory at ICS-FORTH for the useful discussions. We thankfully acknowledge the support of the European Commission under the 7th Framework Program through the STREAM project (Contract No. FP7 216181).

## References

- [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*, pages 55–65, 2002.
- [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, Jan. 2003.
- [4] P. Cao, E. W. Felten, A. R. Karlin, and K. Li. A Study of Integrated Prefetching and Caching Strategies. In *Measurement and Modeling of Computer Systems*, pages 188–197, 1995.
- [5] E. Carrera and R. Bianchini. Improving disk throughput in data-intensive servers. Technical Report DCS-TR500, Rutgers University, Sept. 2002.
- [6] M.-S. Chen, D. Kandlur, and P. Yu. Support for fully interactive playback in disk-array-based video server. In *MULTIMEDIA '94*, pages 391–398, 1994.
- [7] Z. Chen, Y. Zhou, and K. Li. Eviction based cache placement for storage caches. In *Proc. of 2003 Usenix Annual Technical Conference*, pages 269–282, 2003.
- [8] L. Chung, J. Gray, R. Horst, and B. Worthington. Windows 2000 Disk IO Performance. Technical Report MSR-TR-2000-55, July 2000.
- [9] K. Coloma, A. Choudhary, A. Ching, W. K. Liao, S. W. Son, M. Kandemir, and L. Ward. Power and Performance in I/O for Scientific Applications. In *Proc. of IPDPS '05*, 2005.
- [10] 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.
- [11] A. Dan, D. M. Dias, R. Mukherjee, D. Sitaram, and R. Tewari. Buffering and caching in large-scale video servers. In *COMPCON '95*, page 217, 1995.
- [12] J. K. Dey-Sircar, J. D. Salehi, J. F. Kurose, and D. Towsley. Providing vcr capabilities in large-scale video servers. In *MULTIMEDIA '94*, pages 25–32, 1994.
- [13] G. R. Ganger. *System-Oriented Evaluation of I/O Subsystem Performance*. PhD thesis, University of Michigan, June 1995.
- [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 *18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Oct. 2001.
- [16] Jacobson and J. Wilkes. Disk scheduling algorithms based on rotational position. Technical Report HPL-CSP-91-7rev1, Feb. 1991.
- [17] T. Johnson and D. Shasha. 2Q: a low overhead high performance buffer management replacement algorithm. In *Proc. of the 20th VLDB Conference*, pages 439–450, 1994.
- [18] Y. Kim, S. Gurumurthi, and A. Sivasubramaniam. Understanding the performance-temperature interactions in disk I/O of server workloads. In *HPCA '06*, Feb. 2006.
- [19] T. Kimbrel and A. R. Karlin. Near-Optimal Parallel Prefetching and Caching. *SIAM J. Comput.*, 29(4):1051–1082, 2000.
- [20] C. Li, K. Shen, and A. E. Papathanasiou. Competitive prefetching for concurrent sequential I/O. *SIGOPS Oper. Syst. Rev.*, 41(3):189–202, 2007.
- [21] M. F. Mokbel, W. G. Aref, K. Elbassioni, and I. Kamel. Scalable Multimedia Disk Scheduling. In *Proc. of the 20th International Conf. on Data Engineering*, page 498, 2004.
- [22] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed Prefetching and Caching. In *SOSP '95*, pages 79–95, 1995.
- [23] P. V. Rangan and H. M. Vin. Designing file systems for digital video and audio. In *SOSP '91*, pages 81–94, 1991.
- [24] A. L. N. Reddy and J. C. Wyllie. I/O Issues in a Multimedia System. *Computer*, 27(3):69–74, 1994.
- [25] E. Riedel, C. van Ingen, and J. Gray. A Performance Study of Sequential I/O on Windows NT(TM) 4. In *Proc. of the 2nd USENIX Windows NT Symposium*, pages 1–10, 1998.
- [26] C. Ruemmler and J. Wilkes. UNIX Disk Access Patterns. In *Usenix Winter '93 Conference*, pages 405–420, 1993.
- [27] C. Ruemmler and J. Wilkes. An Introduction to Disk Drive Modeling. *IEEE Computer*, 27(3):17–28, 1994.
- [28] 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.
- [29] E. Shriver. *Performance modeling for realistic storage devices*. PhD thesis, New York University, May 1997.
- [30] E. Shriver, C. Small, and K. A. Smith. Why does file system prefetching work? In *Proc. of 1999 Usenix Annual Technical Conference*, pages 71–84, 1999.
- [31] E. Varki, A. Merchant, J. Xu, and X. Qiu. Issues and challenges in the performance analysis of real disk arrays. In *IEEE TPDS*, volume 15, June 2004.
- [32] R. Wijayarathne and A. L. N. Reddy. System support for providing integrated services from networked multimedia storage servers. In *MULTIMEDIA '01*, 2001.
- [33] 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.
- [34] F. C. K. D. Xiaoning Ding, Song Jiang and X. Zhang. DiskSeen: exploiting disk layout and access history to enhance I/O prefetch. In *Proc. of the 2007 USENIX Annual Technical Conference*, June 2007.
- [35] H. Yu, R. Sahoo, C. Howson, G. Almasi, J. Castanos, M. Gupta, J. Moreira, J. Parker, T. Engelsiepen, R. Ross, R. Thakur, and R. L. W. Gropp. High performance file I/O for the Blue Gene/L supercomputer. In *HPCA '06*, Feb. 2006.
- [36] Y. Zhou, J. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proc. of the 2002 USENIX Technical Conf.*, pages 91–104, 2002.
- [37] Y. Zhu and Y. Hu. Can large disk built-in caches really improve system performance? *SIGMETRICS Perform. Eval. Rev.*, 30(1):284–285, 2002.