Accounting for Memory Use, Cost, Throughput, and Latency in the Design of a Media Server

Edward Chang and Hector Garcia-Molina
Department of Computer Science
Stanford University
{echang, hector}@cs.stanford.edu

Abstract

Conventional wisdom holds that reducing disk latency leads to higher disk utilization, maximizing disk utilization leads to higher throughput, employing a faster disk leads to better performance. All of this is true when building a conventional file or database system. In this paper we show that these principles can be misleading when applied to a media server. We examine a number of techniques that have been developed for maximizing disk bandwidth utilization in media servers, including disk arm scheduling and data placement ones. We show that, some disk latency reduction techniques can be counterproductive since memory, rather than disk, is the bottleneck in improving throughput. We present techniques for best memory use under different disk policies, and derive precise formulas for computing memory use. In addition, to precisely account for the design tradeoffs between disk bandwidth and memory use, we propose a cost-based approach that focuses on minimizing the per-stream costs, including the disk and memory ones. Lastly, we study the worst-case initial latency of presentations, a factor that may be important in interactive systems. We present a novel technique for significantly reducing this initial latency under some disk scheduling schemes.
Keywords: multimedia, disk scheduling, disk utilization, memory utilization, initial latency.

Note to referees: This archival journal submission combines and integrates work presented at VLDB 97 [4] and RIDE 98 [5]. In addition to the prior published material, this paper contains new experimental results using data for the latest disks, as well as theorem proofs that were not included in the original papers.

1 Introduction

The design of a media server faces more challenges than that for a conventional file or database system. First, media data must be retrieved from the storage system at a specific rate — if not, the system exhibits "jitter" or "hiccups" [11]. This timely data retrieval requirement is also referred to as the continuous requirement or the real-time constraint [20]. Second, the required data rate is very high. For example, an MPEG-1 compressed video requires an average data rate of 1.5 Mbps and MPEG-2 4 Mbps. Guaranteeing real-time supply at this high data rate for concurrent streams is a major challenge for multimedia storage systems.

Most media server research has focused on optimizing disk bandwidth via scheduling policies and data placement schemes. However, there is a second critical resource that has not received as much attention: the main memory that holds data coming off the disk. In this paper we carefully analyze how memory is used and *shared* among concurrent media requests. Our analysis provides more accurate results than prior analysis, and suggests novel ways in which memory should be shared for maximum performance. Our evaluation also contrasts the gains achievable by disk latency techniques and those achievable by efficient memory use, and shows that with effective memory use, techniques that have higher disk overhead may actually achieve better throughput!

Traditionally, maximum throughput (number of concurrent media streams supported) has been the principal evaluation metric for multimedia storage systems. In this paper we also consider two other important metrics: the per-stream costs and the worst-case initial latency before a new media request can be satisfied. We notice that achieving high throughput often comes at a huge cost in memory. Most research in the area has tended to ignore this, focusing on how to improve disk utilization. Instead, we propose to limit throughput to less than what is feasible in order to minimize per stream costs. Finally, the initial latency metric can be very important in an interactive system where relatively short video clips are shown. In this paper we present a new resource management policy, *BubbleUp*, that can reduce initial latencies by about an order of magnitude, without impacting throughput or memory requirements.

The rest of this paper is organized into ten sections. Section 2 presents our evaluation model and analyzes a traditional system with an elevator disk scheduling algorithm, which we call Sweep. In Section 3 we present the principles behind effective memory sharing, formally proving that memory use can be minimized by spacing out IOs. (This had been hypothesised earlier, but not proven.) We also show how scheme Sweep can best use memory and derive precise formulas for its throughput and memory use.

Next (Section 4) we consider a disk scheduling scheme that generates IOs in a fixed order, independent of the location of the data on disk. (In each period, the disk services requests in a fixed order.) If one implements this scheme in a straightforward way, the performance is terrible because in the worst case each IO may require moving the disk arm across the disk. However, because the order of IO requests is fixed, one can enhance this scheme so that data arrives in memory in a more regular fashion, and this, together with effective memory menagement, can lead to better performance than Sweep's. We call our modified scheme Fixed-Stretch, and we again present performance formulas that precisely account for memory sharing.

The third scheme we consider is the Group Sweeping Scheme (GSS) [22], which can be considered a hybrid between Sweep and Fixed-Stretch. We discuss how memory can be effectively used by this scheme (something that was not clearly spelled out in the original GSS paper). Section 6 examines analytically all schemes presented and shows that reducing disk latency and increasing disk utilization may have adverse impact on throughput. In Section 7 we compare the schemes in a realistic case study, highlighting the basic disk bandwidth and memory use tradeoff. To formally quantify this tradeoff, Section 8.1 proposes a cost model and derives formulas to obtain the minimum per-stream cost.

Sections 9 and 10 analyize initial latencies and data placement policies that are not considered in the first seven sections. We describe our novel resource management technique, BubbleUp, that reduces initial latencies to tens of milliseconds. We also consider the impact of partitioning the disk into regions, and of using multiple disks. Partitioning and multiple disks can impact performance, but they do not alter the conclusions of our study. Finally, Section 11 offers our conclusions.

2 Scheme Sweep

In this section we briefly describe a well known media data delivery scheme, which we call *Sweep*. Scheme Sweep uses an elevator policy for disk scheduling in order to amortize disk seek overhead. It is representative of a class of schemes [9, 11, 15, 16, 18] that optimize throughput by reducing disk seek overhead. Study [16] shows that an elevator policy is superior for retrieving continuous media data in comparison to a policy in which requests with the earliest deadlines are serviced first. We will use scheme Sweep as a benchmark for comparing with other schemes.

For now, let us assume a single disk and let us make no assumptions as to the data placement policies. (We discuss data placement and multiple disks in Section 10.) We first present Sweep under the assumption that each request is allocated its own *private* memory buffer with no memory sharing among the requests. (Buffer sharing among requests is discussed in Section 3.)

During a sweep of the disk, Sweep reads one *segment* of data for each of the requested *streams*. The data for a stream is read into a memory buffer for that stream, which must be adequate to sustain that stream until its next segment is read during the following disk elevator sweep. To analyze the performance of Sweep, we typically are given the following parameters:

- TR: the disk's data transfer rate.
- $\gamma(d)$: a concave function that computes the rotational and seek overhead given a seek distance d. For convenience, we will refer to the combined seek and rotational overhead as the seek overhead.
- Mem_{Avail}: the storage system's available memory.

Parameter	*Description*
Mem_{Avail}	Total available memory, MBytes
DR	Data display rate, Mbps
TR	Disk transfer rate, MBytes/s
CYL	Number of cylinders on disk
$\gamma(d)$	Concave function for rotational and seek overhead given distance d
P	Number of disk partitions
T	Service time for a round of N requests
T_{Seek}	Total worst case seek (and rotational) time in period
$T_{Transfer}$	Total worst case data transfer time in a period
T_{Delay}	Total delay in a period T
T_{Rot}	Worst case rotational delay for one segment read
N	Number of requests being serviced
M	Number of disks
N_{Limit}	System enforced limit on throughput per disk
N_{All}	System-wide throughput
R_i	i_{th} request
cyl_i	Seek distance of the ith request
N_{Max}	Throughput, or maximum number of simultaneous streams
Mem_{Min}	Minimum memory requirement, MBytes
$T_{Latency}$	Initial Latency, seconds
D_{util}	Disk utilization

Table 1: Parameters

- N: the number of stream requests. Each request is denoted as R_1, R_2, ..., R_N. Each stream requires a display rate of DR ($DR < TR$). (For simplicity we assume that the display rates are equal.[1]) The value of N must be less than N_{Limit} as explained below.

Scheme Sweep has the following tunable parameters. They can be adjusted, within certain bounds, to optimize system throughput.

- T: the period for servicing a round of requests. We assume that T is constant, i.e., it does not vary depending on N, the number of streams being serviced at a given time. As discussed below, T must be made large enough to accommodate the maximum number streams we expect to handle. (Although we do not discuss it here, allowing T to vary from cycle to cycle does not improve throughput and may actually hurt latency.)

[1]The techniques we discuss in this paper can be adapted to work with differing display rates in some cases. One alternative is to design for the maximal rate, which is safe and does not hurt performance if the differences between rates are small. Another option is to use the greatest common divisor of the display rates as the unit display rate, and to treat each display rate as a multiple of the unit one. For example, if the display rates are 6 and 4 Mbps, we can treat a 6 Mbps request logically as 3 requests of 2 Mbps each, and we can treat a 4 Mbps request as 2 of 2 Mbps. Thus, all our base requests are of the same rate.

- S: the segment size, i.e., the number of bytes read for a stream with one contiguous disk IO. Since T is constant, S must also be constant over time.

- N_{Limit}: the maximum number of concurrent requests the media server allows. The media server implements an admission control policy that turns away requests when the system is already handling N_{Limit} requests.

To assist the reader, Table 1 summarizes these parameters, together with other parameters that will be introduced later. The first portion of Table 1 lists the basic fixed and tunable parameters. The second portion describes subscripted parameters that are used for characteristics of individual requests. The third portion gives three optimization parameters.

2.1 Analysis

We assume that the media server services the requests in rounds. During a round of service (time T), the media server reads one *segment* of data (sized S) for each of the N_{Limit} requested *streams*. We can run Sweep with many possible values for T, S, and N_{Limit}. However, some values will make it impossible to deliver data for each stream at the appropriate rate due to the violation of certain constraints. Other values will lead to suboptimal performance. For example, we wish to set N_{Limit} as high as possible. For optimal feasible performance, parameters T, S, and N_{Limit} need to satisfy the equations we derive next.

In a feasible system, the amount of data retrieved in a period, S, must be at least as large as the amount of data displayed. That is, $S \geq DR \times T$. However, if we want a stable system, the input rate should equal the output rate, else every period we would accumulate more and more data in memory. Thus, we have the equation

$$S = DR \times T. \tag{1}$$

In a feasible system, the period T must be large enough so that all necessary IOs can be performed. Since T is fixed and cannot vary depending on the number of concurrent requests in the system at a particular moment, we must make T large enough to accommodate N_{Limit} seeks and transfer N_{Limit} segments. Furthermore, in computing these seek times, we have to assume a worst case situation, so that no matter where the segments are located on disk, we will have enough time to read them.

The total seek overhead for N_{Limit} requests is $\sum_{i=1}^{N_{Limit}} \gamma(cyl_i)$, where γ gives the seek delay for the i^{th} request. Since γ is a concave function [13, 17, 19], the largest value of the total seek overhead for Sweep occurs when the segments are equally spaced on the disk, or $cyl_i = CYL/N_{Limit}$. Thus, the worst case seek (and rotational) time is:

$$T_{Seek} = N_{Limit} \times \gamma(CYL/N_{Limit}). \tag{2}$$

The total transfer time for N_{Limit} requests, each of size S, at a transfer rate TR, is

$$T_{Transfer} = N_{Limit} \times S/TR. \tag{3}$$

As we stated above, the period T must be larger or equal than the worst case seek and transfer times, i.e., $T \geq T_{Seek} + T_{Transfer}$. For optimal performance, however, we take the smallest feasible T value, since otherwise we would be wasting both disk bandwidth and memory resources. That is,

$$T = T_{Seek} + T_{Transfer} = N_{Limit} \times (\gamma(CYL/N_{Limit}) + S/TR). \tag{4}$$

The third equation that must be satisfied by scheme Sweep is obtained from our physical memory limit. Although in a period we only read S bytes for each stream, it turns out we need a buffer of twice that size for each stream to cope with the variability in read times. To see this, consider a particular stream in progress, where we call the next three disk arm sweeps A, B, and C. Assume that the segments needed by our stream are a, b, and c. It so happens that because of its location on disk segment a is read at the beginning of sweep A, while b is read at the end of B, $2 \times T$ time units after a is read. At the point when a is read we need to have in memory $2 \times S$ data, to sustain the stream for $2 \times T$ time.

When segment b is read, we will only have S bytes in memory, which is only enough to sustain us for T seconds. Fortunately, because b was at the end of its sweep, the next segment c can be at most T seconds away, so we are safe. Actually, c could happen to be the very first segment read in sweep C, in which case we would again fill up the buffer with roughly $2 \times S$ data (minus whatever data was played back in the time it takes to do both reads).

Intuitively, what happens is that half of the $2 \times S$ buffer is being used as a *cushion* to handle variability of reads within a sweep. Before we actually start playing a stream we must ensure that this cushion is filled up. In our example, if this stream were just starting up, we could not start playback when a was read. We would have to wait until the end of sweep A (the sweep where first segment a was read) before playback started. Then, no matter when b and c and the rest of the segments were read within their period, we could sustain the DR playback rate. (This startup delay will be important when we analyze initial latencies in Section 9.)

Adding a cushion buffer for each request doubles the memory required. So, to support N_{Limit} requests, we must have $Mem_{Avail} \geq 2 \times N_{Limit} \times S$. For optimal performance, however, we should use all available memory. (By using all available memory we make segments larger. This lets us increase T (Eq. 1), which then lets us increase N_{Limit} in Equation 4, since $TR > DR$.) Thus, we have that for optimal feasible performance,

$$Mem_{Avail} = 2 \times N_{Limit} \times S. \tag{5}$$

In summary, scheme Sweep has three tunable parameters, and we have derived three equations they must satisfy (Equations 1, 4, 5) for optimal performance. From these equations we can solve for T, S, and N_{Limit}.

2.2 Minimizing Memory

To obtain the minimum required memory, we assume that N_{Limit} is given and that Mem_{Avail} is unknown, and we solve for it. Substituting $T = S/DR$ (Eq. 1) into Equation 4, we can solve for S, the segment size needed to support the N_{Limit} requests:

$$S = \frac{N_{Limit} \times \gamma(CYL/N_{Limit}) \times TR \times DR}{TR - (DR \times N_{Limit})}. \tag{6}$$

(We assume that $TR - (DR \times N_{Limit}) > 0$, else no segment size is sufficient. Some literature refers to this as disk bandwidth constraint.) Multiplying this value by $2 \times N_{Limit}$ (Eq. 5), we obtain the minimum amount of memory,

$$Mem_{Min} = \frac{2 \times N_{Limit}^2 \times \gamma(CYL/N_{Limit}) \times TR \times DR}{TR - (DR \times N_{Limit})}. \tag{7}$$

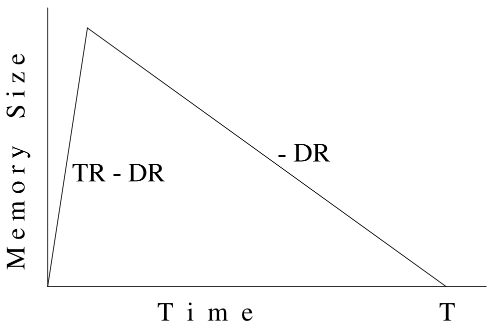
2.3 Maximizing Throughput

Given Mem_{Avail}, we can compute the maximum throughput, denoted as N_{Max}, using Equation 7. We try different values of N_{Limit}, each gives us a value of Mem_{Min}. The N_{Limit} that produces the maximum Mem_{Min} satisfying $Mem_{Min} \leq Mem_{Avail}$ is the maximum throughput, N_{Max}, the server can support. Since having the formula to compute Mem_{Min} one can obtain N_{Max}, we do not repeat how to obtain N_{Max} for other schemes in the rest of this paper.

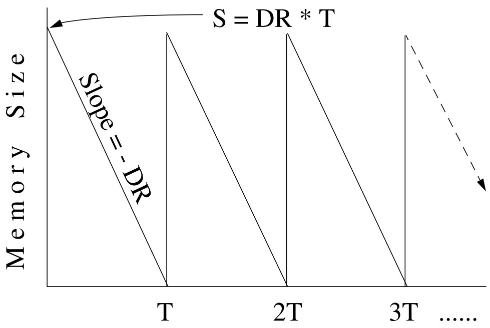
3 Reducing Required Memory

In scheme Sweep each request is allocated a fixed private buffer of size $2 \times S$. One way to reduce the memory requirements is to have requests share their buffer space [14, 19]. That is, we create a shared memory pool, and as the space used by one request frees up, it can be used to hold data from other requests. Various papers have estimated that sharing can cut the memory requirements by "roughly half." However, these estimates are obtained with very strong assumptions, in particular that all seeks times must be zero. In this section we revisit how exactly memory sharing works (without strong assumptions), and in doing so *prove* under what conditions maximal sharing can be obtained, and what the savings actually are.

Figure 1(a) depicts the amount of memory used by a request in a period T. An IO starts shortly before the data staged into memory in the previous period is used up. The data accumulates in memory at the rate of $TR - DR$ until the IO completes.



(a) Memory Required in A Period



(b) Memory Use Function $p_i(t - \tau_i)$

Figure 1: Memory Requirement Function

For our analysis we make two simplifying assumptions. First, we assume that memory can be freed in a continuous fashion. In other words, Figure 1(a) shows the actual memory used by a request. In practice, of course, memory is released in pages, so Figure 1(a) would have a sequence of small decreasing steps, each one page in size. This implies that our estimates for memory use may be up to one memory page off for each request. Thus, our continuous release assumption is an optimistic one for buffer sharing schemes. However, if as expected the page size is small compared to the segment size, the difference will be negligible.

Our second assumption is to approximate the memory use function by a right triangle. Our assumption causes us to overestimate memory use: we will assume that the peak in Figure 1(a) is S (at time 0 in the figure), while in reality it is $S \times (1 - DR/TR)$. This is a pessimistic assumption, but since typically the data transfer rate TR is much larger than the display rate DR, the difference is very small.

Notice that the small differences caused by our two assumptions tend to cancel out each other. In particular, if the page size is $S \times DR/TR$, the effects will cancel. If the page size is less than this value, as is probably the case, then overall our results will be slightly pessimistic for memory sharing.[2]

[2]When an IO is initiated, the physical memory pages for the data it reads may not be contiguous due to the way buffers are shared. There are several ways to handle these IOs. One idea is to map the physical pages to a contiguous virtual address, and then initiate the transfer to the virtual space (if the disk supports this). Another idea is to break up the segment IO into multiple IOs, each the size of a physical page. The transfers are then chained together and handed to an IO processor or intelligent DMA unit that executes the entire sequence of transfers with the same performance as a larger IO. Other ideas are discussed in [14].

3.1 Optimal Delays

Before discussing memory sharing under Sweep, it is instructive to analyze an ideal case where IOs for a given stream occur in a regular fashion, as shown in Figure 1(b). In this scenario the data for a request is fully played back just as the next IO completes, so there is not need for cushion buffers.

Let us denote the periodic function in Figure 1(b) as $p_i(t - \tau_i)$, where t represents time and τ_i is the displacement from the beginning of the period (e.g., the example shown in Figure 1(b) has a displacement of 0). The memory use function $p(t)$ for N_{Limit} concurrent requests is a superposition of N_{Limit} such periodic functions, or

$$p(t) = \sum_{i=1}^{N_{Limit}} p_i(t - \tau_i).$$

Notice that each function $p_i(t - \tau_i)$ has a different displacement. To minimize the memory requirement of a system, one has to minimize the largest value of $p(t)$. The only parameters that can be adjusted in $p(t)$ are the $\tau_i's$. The following results tell us what these displacements should be for optimal memory sharing among requests.

Theorem 1: We are given a multimedia storage system that supports N_{Limit} continuous streams with equal display rate DR. Minimizing memory usage requires the IO start times to be spaced equally in T. The proof is provided in Appendix A.1.

Corollary 1: The minimum memory space required to support N_{Limit} streams with equal display rate DR is $\frac{S\times(N_{Limit}+1)}{2}$. (Keep in mind that this does not include cushion buffer requirements.) The proof is provided in Appendix A.2.

These results suggest that even if one cannot perfectly separate in time the IO sequences, it is desirable to space out requests as much as possible. This is precisely what we do in order to optimize the memory use of Sweep.

3.2 Scheme Sweep*

We refer to scheme Sweep with memory sharing as Sweep*. With a sweeping scheme we cannot control when IOs occur within a period, since they are done in the order found as the head sweeps the disk. In the worst case, all the IOs in a period will be bunched together (if all the segments needed in a period happen to be nearby on the disk.) This means that the memory peaks are summed, leading to poor memory sharing. However, the IOs need to be separated by at least the time it takes to read a segment, so the peaks are not fully added. In addition, the last IO of a period can be delayed and separated from a cluster of IOs, further improving memory sharing. Finally, it is also possible to share the cushion buffers used by each stream to account for IO variability, leading to even better memory utilization. All these effects are carefully analyzed in Appendix A.3, where we also show the following result. Incidentally, note that various papers had earlier "guessed" how much memory a scheme like Sweep* uses, but these estimates were not accurate.

Theorem 2: The minimum memory space required to support N_{Limit} streams under scheme Sweep* is $(N_{Limit} - 1) \times S + N_{Limit} \times DR \times (T - \frac{(N_{Limit}-2) \times S}{TR})$.

4 Scheme Fixed-Stretch*

In order to reduce the variability between the IOs of a request, in this section we consider a scheme where IOs are performed in a fixed order from period to period. We call this scheme Fixed-Stretch* because in addition the IOs are spaced out as described next. For completeness, a version of this scheme that does not use memory sharing, Fixed-Stretch, is discussed at the end of this section.

To eliminate the need for cushion buffers entirely and maximize memory sharing (Theorem 1), we must separate the IOs of a request by a constant time T. However, since the data on disk for the requests are not necessarily separated by equal distance, we must add time delays between IOs to space them equally in time.

For instance, if the seek distances in a disk sweep are cyl_1, cyl_2,..., and $cyl_{N_{Limit}}$ cylinders, and cyl_i is the maximum of these, then we must separate each IO by at least the time it takes to seek to and transfer this maximum i^{th} request. One can choose a different separator for each period, depending on the maximum seek distance for the requests of that period. However, as we have argued earlier, there is no benefit allowing T to vary from cycle to cycle. To have a constant T and simplify the algorithms, scheme Fixed-Stretch* uses the worst possible seek distance (CYL) and rotational delay, together with a segment transfer time, as the universal IO separator, Δ, between any two IOs. The length of a period, T, will be N_{Limit} times Δ.

Figure 2 shows an example with three requests that thusly separated. The time on the horizontal axis is divided into service cycles each lasting T units. Each service cycle T (the shaded area) is equally divided into three service slots, each lasting Δ units (delimited by two up-arrows). The vertical axis in Figure 2 represents the amount of memory utilized by an individual stream.

Fixed-Stretch* executes according to the following steps:

1. At the start of a service slot (indicated by the up-arrow in Figure 2), it sets the *end of slot timer* to expire in Δ.

2. If there is no request to be serviced in the service slot, it skips to Step 6.

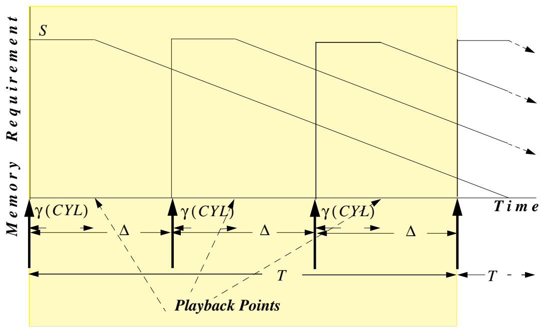


Figure 2: Service Slots of Fixed-Stretch*

3. It allocates S amount of memory for the request serviced in this time slot.[3]

4. It sets the *IO timer* to expire in $\gamma(CYL)$, the worst possible seek overhead, and starts the disk IO. Since the actual seek overhead cannot exceed $\gamma(CYL)$, when the *IO timer* expires the data transfer must have begun.

5. When the *IO timer* expires, the playback starts consuming the data in the buffer (indicated by the "playback point" pointers in Figure 2), and the memory pages are released as the data is consumed.

6. When the *end of slot timer* expires, the data transfer (if issued in Step 4) must have completed.[4] Fixed-Stretch goes to Step 1 to start the next service slot.

As its name suggests, the basic Fixed-Stretch* scheme has two distinguishing features:

- Fixed-order scheduling: A request is scheduled in a fixed service slot from cycle to cycle after it is admitted into the server. For instance, if a request is serviced in the k^{th} slot when it first arrives, it will be serviced in the same k^{th} slot in its entire playback duration, regardless of whether other requests depart or join the system. (As we will see in Section 9, the "fixed" scheduling may be changed by the BubbleUp scheme.)
- Stretched out IOs: The allocated service slot assumes the worst possible disk latency $\gamma(CYL)$ so that the disk arm can move freely to any disk cylinder to service any request. This property ensures that the fixed-order scheduling is feasible no matter where the data segments are located on the disk.

At first glance, Fixed-Stretch* appears to be inefficient since it assumes the worst seek

[3]When an IO is initiated, the physical memory pages for the data it reads may not be contiguous due to the way buffers are shared. There are several ways to handle these IOs. One idea is to map the physical pages to a contiguous virtual address, and then initiate the transfer to the virtual space (if the disk supports this). Another idea is to break up the segment IO into multiple IOs, each the size of a physical page. The transfers are then chained together and handed to an IO processor or intelligent DMA unit that executes the entire sequence of transfers with the same performance as a larger IO. Other ideas are discussed in [14].

[4]The accuracy of the timers used by Fixed-Stretch* can be tuned periodically by cross-checking the amount of data in the stream buffers.

overhead between IOs. However, it actually saves memory in two ways. First, because IOs in a period are spaced out, memory sharing is at its best. Second, because there is almost no time variability between the IOs of a given request, we need tiny cushion buffers (discussed shortly). Fixed-Stretch* does require larger segments (T is artificially enlarged, and $S = DR \times T$), but in our analysis we will see that overall Fixed-Stretch* does save substantial amounts of memory and actually leads to improved throughput over scheme Sweep! This result is surprising since Fixed-Stretch* is underutilizing bandwidth by slowing down the disk, doing just the opposite of what previous scheduling schemes do.

4.1 Analysis

In scheme Fixed-Stretch*, a period T is composed of three elements: seek overhead, padding, and transfer time for the maximum possible N_{Limit} requests. The transfer time, $T_{Transfer} = N_{Limit} \times S/TR$, is the same as for scheme Sweep. As we have discussed, each individual seek plus padding delay must be as large as the worst case seek of CYL cylinders. The total seek overhead and delay for N_{Limit} requests is N_{Limit} times this value, so $T_{Seek} + T_{Delay} = N_{Limit} \times \gamma(CYL)$. Thus T can be written as

$$T = N_{Limit} \times \Delta = N_{Limit} \times (\gamma(CYL) + \frac{S}{TR}), \tag{8}$$

which replaces Equation 4 we had obtained earlier for scheme Sweep.

According to Corollary 1, scheme Fixed-Stretch* requires $\frac{S \times (N_{Limit}+1)}{2}$ memory. In addition, since the release of the memory allocated by each request at the beginning of a service slot does not start until at the playback point of the slot (see Figure 2 and execution step 5), each request needs to retain the data for extra $\gamma(CYL)$ time. This delay of data consumption (and hence memory deallocation) requires that each request has an additional $\gamma(CYL) \times DR$ amount of buffer space. (This extra space requirement is very small; for the parameters values of Section 7, the extra buffer requirement is about 0.5% of the segment size.) The required memory must be smaller than or equal to the available memory Mem_{Avail}. For optimal performance, we try to give each request as much memory as possible to maximize N_{Limit}. Thus, we have the memory resource constraint

$$Mem_{Avail} = \frac{S \times (N_{Limit} + 1)}{2} + (N_{Limit} \times \gamma(CYL) \times DR), \tag{9}$$

which replaces our old equation 5.

In summary, scheme Fixed-Stretch* also has three tunable parameters: T, S, and N_{Limit}; and we have derived three equations they must satisfy (Equations 1, 8, 9) for optimal performance. From these equations we can solve for T, S, and N_{Limit}.

4.2 Minimizing Memory

Substituting $T = S/DR$ (Eq. 1) into Equation 8, we can solve for S, the segment size needed to support the N_{Limit} requests:

$$S = \frac{N_{Limit} \times \gamma(CYL) \times TR \times DR}{TR - (DR \times N_{Limit})}. \tag{10}$$

From equation 9 we can then solve for the required minimum memory for scheme Fixed-Stretch:

$$\begin{aligned} Mem_{Min} &= \frac{S \times (N_{Limit}+1)}{2} + (N_{Limit} \times T_{Rot} \times DR) \approx \frac{S \times (N_{Limit}+1)}{2} \\ &= \frac{(N_{Limit}+1)N_{Limit} \times \gamma(CYL) \times TR \times DR}{2(TR - (DR \times N_{Limit}))}. \end{aligned} \tag{11}$$

If memory is *not* shared among the requests, we simply replace Equation 9 with

$$Mem_{Avail} = S \times N_{Limit} + (N_{Limit} \times \gamma(CYL) \times DR).$$

We call this scheme Fixed-Stretch (without the *), and the steps to derive Mem_{Min} are the same.

5 Group Sweeping Scheme* (GSS*)

So far we have presented two extreme schemes: Sweep* minimizes seek overhead with high memory requirement, and Fixed-Stretch* maximizes memory sharing and minimizes cushion buffer requirement with the worst seek overhead. In this section we consider a hybrid scheme that lies between Sweep* and Fixed-Stretch*.

The Group Sweeping Scheme (GSS) proposed in [22] divides N_{Limit} streams into G groups, with N_{Limit}/G streams serviced in each group by a disk sweep. (For simplicity, we assume that N_{Limit} is divisible by G.) The groups are serviced in a round-robin fashion. A request is assigned to a single group from the start to the end of its playback.

In the published descriptions of GSS it is not clear to us how memory sharing is handled nor how much time transpires between reading the last request in a group and reading the first one of the next group. Here we clarify these issues, and improve GSS with the techniques we developed for Fixed-Stretch*. We call the resulting scheme GSS*, to differentiate it from other possible interpretations of the scheme in [22]. (Our GSS* also uses a variant of BubbleUp that reduces initial latency; see Section 9.) In this paper we use GSS to refer to the scheme with *no* memory sharing.

In GSS* we assume that a period T is divided into G *epochs*, each of duration T/G *exactly*. During an epoch i, we perform a single disk sweep, reading N_{Limit}/G segments. (Epochs are long enough so that this can always be accomplished.) After an epoch starts, we start performing the first $(N_{Limit}/G)-1$ IOs as we sweep the disk. Before we perform the last of the IOs for this epoch, however, we *wait* until the epoch is about to finish, and then we perform the last IO, just as the epoch finishes. This process of delaying the last IO is the same we used in Fixed-Stretch*.

Scheme GSS* is like Fixed-Stretch*, in that it introduces artificial delays to space out IOs. In fact, if $G = N_{Limit}$, GSS* is identical to Fixed-Stretch*: each GSS* epoch corresponds to one of the IOs of Fixed-Stretch*. If $G = 1$, GSS* is like Sweep*. Hence, GSS* is a parameterized hybrid between Fixed-Stretch* and Sweep*.

5.1 Analysis

The worst total seek overhead for an epoch occurs when its segments are equally spaced on the disk (see Section 2.1). Since each epoch under GSS* services N_{Limit}/G requests, the worst case seek distance cyl_i is $CYL \times G/N_{Limit}$. Thus, the worst case seek (and rotational) time is: $T_{Seek} = N_{Limit} \times \gamma(CYL \times G/N_{Limit})$. Following the same steps in Section 2.2, we obtain the segment size:

$$S = \frac{N_{Limit} \times \gamma(CYL \times G/N_{Limit}) \times TR \times DR}{TR - (DR \times N_{Limit})}. \tag{12}$$

The following theorem gives the memory requirements for GSS*, taking into account sharing of all memory (including any cushions needed to cope with IO variability).

Theorem 3: The minimum space required to support N_{Limit} streams under scheme GSS* is

$$(N_{Limit}/G) \times S \times \frac{G+1}{2} - S + N_{Limit} \times DR \times (T/G - (N_{Limit}/G - 2)\frac{S}{TR}). \tag{13}$$

Please refer to Appendix A.4 for the proof.

6 Disk Latency

Having studied several scheduling and memory management policies, it is useful to step back a little and observe some common features in all schemes. In particular, it turns out that some simple relationships hold between disk latency, disk utilization, and the required memory, independent of the scheme used. From these relationships we will be able to see that contrary to the conventional wisdom, reducing disk latency or increasing disk utilization in media servers may be detrimental to throughput.

6.1 Disk Latency vs. Disk Utilization

One definition of disk utilization is the percentage of time the disk is busy (not idle). However, this definition does not reflect how efficiently a disk is used. For example, a disk can spend all its time seeking without transferring data to achieve 100% utilization. To measure the effectiveness of disk scheduling or data placement, we define instead disk utilization as the fraction of time the disk performs useful work, i.e., transfers data. In the following we show that given N_{Limit} to support, disk utilization is not affected by disk latency.

Theorem 4 The disk utilization, or the fraction of time the disk performs useful work, of a media server is independently to the disk latency $\gamma(cyl_i)$, given the same number of users (i.e., N_{Limit}) to support.

[Proof] Let D_{util} denote disk utilization. Given that the media server services N_{Limit} requests, disk utilization can be written as

$$D_{util} = \frac{N_{Limit} \times S/TR}{T}.$$

Replacing T with S/DR (Eq. 1) yields

$$D_{util} = \frac{N_{Limit} \times DR}{TR}. \tag{14}$$

This tells us that, given N_{Limit} and the DR/TR ratio, disk utilization is constant, regardless of the disk latency $\gamma(cyl_i)$! Thus, reducing disk latency does not lead to better disk utilization. That is, since a disk scheduling policy cannot change DR nor TR of a given disk, a smart disk scheduling policy that reduces disk latency does not help improve disk utilization. In other words, smart scheduling policies do not increase the amount of useful work done by the disk. $\Box$

The reason why reduced latency does not imply better utilization is that media servers must deliver data to meet deadlines. To prepare data in time for the clients, the server must read the proper amount of data to compensate for disk latency. If disk latency is high, the server must read more data for a request to sustain a longer data consumption period before the request's next read can be scheduled. Conversely, if the latency is low, the server reads less data. Reading more data than necessary wastes memory, and reading less data than required violates the real-time constraint. Thus, the media server always adjust the amount of data it must transfer according to the latency of the disk, and disk utilization remains constant.

6.2 Disk Utilization vs. Memory Requirement

Equation 14 shows that the disk utilization, D_{util}, is directly proportional to N_{Limit}. In a traditional file system, increasing disk utilization is good, as long as response time does not suffer too much. Since in our multimedia server after the initial delay to start the playback the continuous data supply is "guaranteed," one might be tempted to make utilization very high, as this leads to a higher throughput N_{Limit}. Unfortunately, memory use is adversely affected by high throughput.

The minimum memory requirements of the disk scheduling policies that we have discussed can be written in the following form:

$$Mem_{Min} = K_1 \times N_{Limit} \times S + K_2, \tag{15}$$

where K_1 and K_2 are policy dependent constants. To analyze the relationship between Mem_{Min} and N_{Limit} we can ignore the constant terms and rewrite the relationship as

$$Mem_{Min} \propto N_{Limit} \times S.$$

Substituting S by $(N_{Limit} \times \gamma(cyl_i) \times TR \times DR)/(TR - (DR \times N_{Limit}))$, we obtain

$$Mem_{Min} \propto \frac{N_{Limit}^2 \times \gamma(cyl_i) \times TR \times DR}{TR - (DR \times N_{Limit})}. \tag{16}$$

It is important to note in Equation 16 that Mem_{Min} grows more than linearly with the desired N_{Limit}. First, the numerator grows quadratically with N_{Limit}. Second, as N_{Limit} grows the denominator of Equation 16 approaches zero, causing Mem_{Min} to grow without bound. As the denominator gets close to zero, we are driving the system to its physical limits: N_{Limit} streams at a DR rate require $N_{Limit} \times DR$ bytes per second, and the disk can only read at most TR per second. When $N_{Limit} \times DR$ approaches TR, the system needs a huge amount of memory to support an additional request. As we will discuss in Sections 7 and 8, maximizing disk utilization is not cost effective.

7 Evaluation

To evaluate and compare the performance of the various schemes discussed in this paper, we use the Seagate Barracuda 9LP disk [2]; its parameters are listed in Table 2. We also assume a display rate DR of 1.5 Mbps, which is sufficient to sustain typical video playback. For the seek overhead we follow closely the model developed in [13, 17] that is proven to be asymptotically close to the real disks. The seek overhead function is a concave function as following:

$$\gamma(d) = \alpha 1 + (\beta 1 \times \sqrt{d}) + 8.33 \text{ if } d < 400$$

$$\gamma(d) = \alpha 2 + (\beta 2 \times d) + 8.33 \text{ if } d \geq 400$$

Note that the seek time is proportional to the square root of the seek distance when the distance is small, and is linear to the seek distance when the distance is large. We derive

Parameter Name	Value
Disk Capacity	*9.19 GBytes*
Number of cylinders, CYL	*6,000 (est)*
Min. Transfer Rate TR	*120 Mbps*
RPM	*7,200*
Max. Rotational Latency Time	*8.33 milliseconds*
Min. Seek Time (read)	*0.8 milliseconds*
Max. Seek Time (read)	*13.4 milliseconds*
$\alpha 1$	*0.54 milliseconds*
$\beta 1$	*0.26 milliseconds*
$\alpha 2$	*5 milliseconds*
$\beta 2$	*0.0014 milliseconds*

Table 2: Seagate Barracuda 9LP Family Disk Parameters

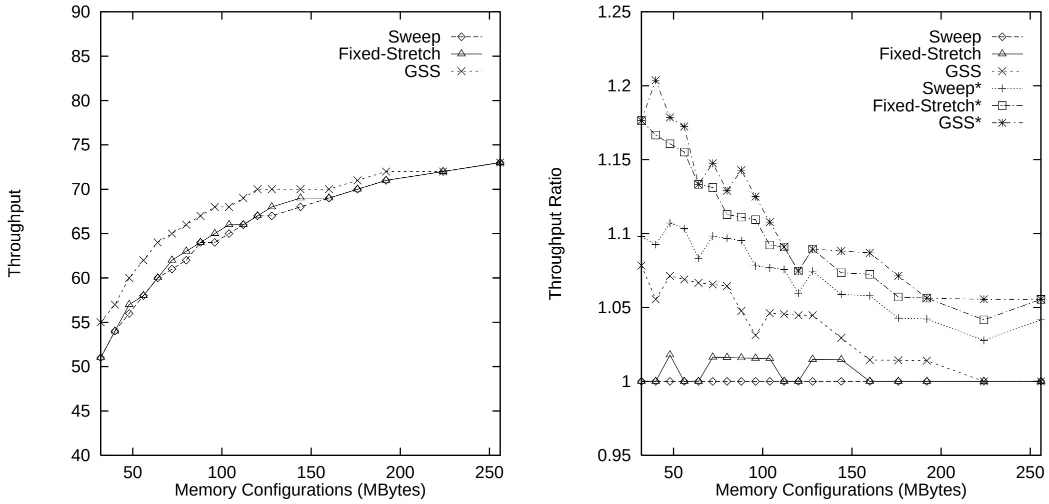
the parameters for this function as follows. We first allocate 2/3 of the minimum seek time provided by the vendor (0.8 ms) as the disk arm's fixed overhead $\alpha 1$ (which includes the speedup, slowdown, and settle phases). Parameter $\beta 1$ then is the remaining portion of the minimum seek time. We then select $\alpha 2$ and $\beta 2$ so that the maximum seek time matches the manufacture's time (15 ms), and so that function γ is continuous at $d = 400$. The values obtained are given in Table 2.[5]

In each seek overhead we have included a full disk rotational delay T_{Rot} of 8.33 ms. The rotational delay depends on a number of factors, but we believe that one rotation is a representative value. One could argue that rotational delay could be eliminated entirely if a segment is an exact multiple of the track size. (In that case we could start reading at any position of the disk.) However, the optimal segment size depends on the scenario under consideration, so it is unlikely it will divide exactly into tracks. If we assume that the first track containing part of a segment is not full, then in the worst case we need a full rotation to read that first portion, even with an on-disk cache. If we assume that the last track could also be partially empty, then we could need a second rotational delay, and our 8.33ms value may be conservative! Note incidentally that we use a *full* rotational delay (not average) since we are estimating a worst case scenario.

7.1 Throughput

Figure 3 presents the throughput of our schemes for various memory sizes. First, Figure 3(a) shows performance when memory is not shared. For GSS, we show only the best throughput achieved by the optimal G value. With no memory sharing, the throughput of Sweep and

[5] Incidentally, [17] suggests using between 200 to 600 cylinders to separate short and long seeks. Although we do not show it here, our results are not very sensitive to the exact value used in this range.



(a) Memory Not Shared

(b) Throughput Ratio

Figure 3: Throughput Comparison

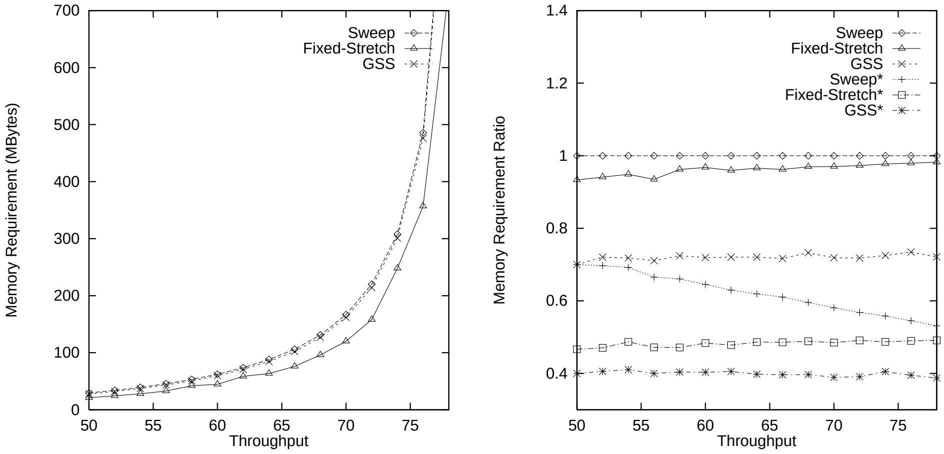
Fixed are almost identical. This is because Sweep's benefit from the reduced seek overhead is canceled out by its large cushion buffer requirement. As expected, GSS is able to achieve better throughput by balancing the seek time and cushion buffer requirements. However, Figure 3(a) shows that the gap between the best and worst schemes is not very significant: three to four streams at best under most memory configurations.

Figure 3(b) shows the throughput improvement that each scheme offers over the basic Sweep with no memory sharing. The ratios shown are the performance of each scheme divided by the Sweep throughput. Thus, Sweep has a constant ratio of 1. A ratio greater than 1 means that the scheme performs better than Sweep.

In the figure we can easily see that as memory increases, the throughput of all schemes converges. It is also clear that for limited memory, memory sharing pays off in terms of improved throughput. However, even with limited memory, the differences among the memory sharing schemes are not very significant. That is, as long as memory is shared efficiently, disk scheduling policies do not have a great impact on throughput. Earlier studies had predicted greater differences, partly because memory sharing had not been carefully analyzed or considered.

7.2 Memory Requirements

We next set a desired throughput, and study how much memory each scheme needs to support it. Figure 4 shows the minimum amount of memory required, Mem_{Min}, to support a given number of requests (N_{Limit}). Again, the (a) part of the figure shows only the no-memory-



(a) Memory Requirement

(b) Ratio

Figure 4: Memory Requirement

sharing schemes, while the (b) part shows for all schemes the ratio of its memory requirement to that of Sweep. For GSS and GSS* we again use an optimal G value.

With no memory sharing, scheme GSS requires about 75% of the memory of the other schemes, so it is clearly superior. With memory sharing, GSS* is still the best, but the gap with Fixed-Stretch* is reduced. Interestingly, Sweep* performs quite poorly even compared to Fixed-Stretch*, unless we require a very high throughput. As we argue next, we probably do not wish to operate at a very high throughput level, so Sweep* does not seem attractive. Thus, Sweep*, in its attempt to optimize disk movement, uses memory less effectively, and ends up being not desirable.

As expected, Figure 4(a) shows that as N_{Limit} increases, the required memory grows rapidly. For example, say we are running scheme Sweep without sharing memory with 300 MB, supporting up to 74 concurrent streams. If we wish to add memory to bump our limit to 76 concurrent requests, we need to add about 200 MB of memory! Even an efficient scheme like GSS* would require a huge amount of memory to increase throughput by one.

For all schemes, the marginal memory requirement starts dramatically increasing around $N_{Limit} = 65$. From $N_{Limit} = 53$ to the maximum achievable throughput 79, the memory requirement grows almost 30 fold. This suggests that although it is theoretically possible to use memory to reduce seek overhead and improve throughput, it may be economically unwise. This leads to the evaluation of the next section.

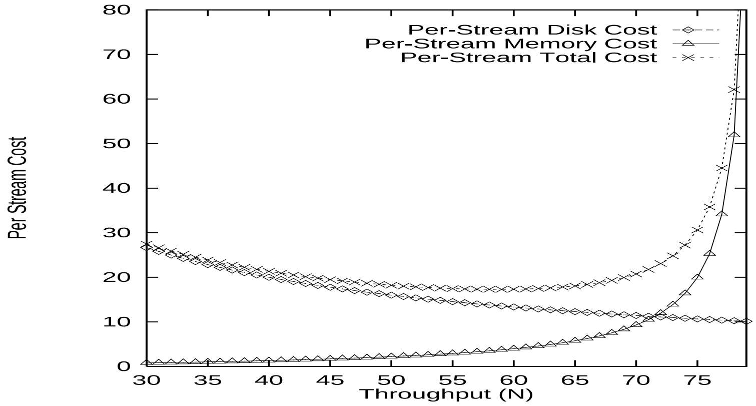


Figure 5: Per Stream Cost by Throughput

8 Per Stream Cost

One important measure for a media server's performance is the per stream cost, including the cost of memory, disk, CPUs, and peripherals. Since the per-stream CPU and peripheral costs are fixed, independent of the number of streams, for simplicity, our discussion focuses only on memory and disk costs.

To illustrate the importance of using the per-stream cost to determine the load of a disk (i.e., disk utilization), Figure 5 plots the per stream disk, memory, and total cost for Scheme Sweep, as a function of N_{Limit}. We assume that each disk costs \$800 and the memory cost is \$5 per MByte (including other associated cost such as for the memory boards). Notice that when N_{Limit} is between 55 and 63, the per-stream total cost is at the lowest, If we try to increase throughput beyond that, our costs will start increasing. If we continue to push performance past say 70 concurrent streams, we must pay a high premium. (Supporting $N_{Limit} = 70, 73, 76$ incurs additional per-stream cost of $20\%, 50\%$, and 100% respectively.) Thus, we argue that given a fixed budget, a server that supports the minimum per-stream cost (instead of maximum disk utilization or minimum memory use) achieves the maximum throughput.

In the remainder of this section, we first derive a single disk's optimal load (i.e., N^*_{Limit}) which achieves the minimum per-stream cost. We then show how to decide the optimal amount of memory and number of disks to buy to optimize a media server's throughput. Finally, we present steps for selecting a disk among many models by considering the cost/performance factor. We show that employing a faster disk does not necessarily achieve better throughput than using a slower model.

8.1 Minimum Per-Stream Cost

Let C_d denote the cost of a disk, and C_m the cost of one MByte of memory. The per-stream cost function, denoted as $C_s(N_{Limit})$, can be written as

$$C_s(N_{Limit}) = C_d/N_{Limit} + C_m \times Mem_{Min}/N_{Limit}.$$

Substituting Mem_{min} by $K_1 \times N_{Limit} \times S + K_2$ (Eq. 15) we obtain

$$C_s(N_{Limit}) = C_d/N_{Limit} + C_m \times K_1 \times S + K_2/N_{Limit}.$$

In the following theorem, we derive the disk load N^*_{Limit} that minimizes the per-stream total cost for scheme Sweep. Since each stream of the Sweep scheme requires $2 \times S$ amount of memory, or $K_1 = 2$ and $K_2 = 0$, the per-stream cost function can be simplified as

$$C_s(N_{Limit}) = C_d/N_{Limit} + 2 \times C_m \times S. \tag{17}$$

The derivation steps for other disk scheduling policies are similar: they only differ in the values of K_1 and K_2.

Theorem 5 Let C_d is the cost of a disk and C_m the cost of one MByte memory. The N^*_{Limit} that achieves the minimum per-stream cost for scheme Sweep is

$$N^*_{Limit} = \frac{TR}{DR} \times \frac{TR \times \sqrt{2 \times C_d \times C_m \times DR \times \gamma(cyl_i)} - C_d \times DR}{2 \times C_m \times TR^2 \times \gamma(cyl_i) - C_d \times DR}.$$

The proof of Theorem 5 is provided in Appendix A.5.

8.2 Storage Cost

Theorem 5 gives us the optimal load a disk should operate on to minimize the per stream cost. Given a required throughput level N_{All}, we can now determine how many disks d and how much memory to minimize the total cost. That is, we select d so that N_{All}/d is as close as possible to N^*_{Limit} on each disk. For instance, if N_{All} is 38 and N^*_{Limit} is 10, we buy 4 disks, and enough memory to support the resulting segment size S. This gives us the lowest per-stream costs and hence the lowest overall cost. However, we still need to consider the storage of the media, and the associated costs.

The storage space required depends on the number, the length, and the average bitrates of the presentations stored by the media server. For example, a movie-on-demand server may require much larger storage space than a news-on-demand one since a typical movie plays longer and may require higher display quality (and hence more bits to encode a frame) than a typical news clip. Therefore, the number of disks that leads to minimum per-stream cost may be insufficient (e.g., for a movie-on-demand server) or more than enough (e.g., for a news-on-demand server) for storing data. We discuss these two cases separately.

1. *Less disk space is needed than what the minimum per-stream cost suggests.* In this case, we have space left unused on the disks. One alternative is to buy fewer disks, i.e., just enough to satisfy the storage requirement. This, as we have shown in Section 6.2, increases disk utilization on each disk and actually increases the per-stream cost. It is hence undesirable. Therefore, one is better off maintaining the number of disks at the minimum per-stream cost level by "wasting" some disk space.

2. *More disk space is needed than what the minimum per-stream cost suggests.* If the optimal number of disks is not adequate for storing all presentations, some of the "colder" presentation can be stored on tertiary storage (i.e., optical disks and tapes). Most likely we will anyway have backup copies on tertiary storage (for failure recovery), so the storage space does not cost us more. Of course, cold presentations would suffer much grater initial latencies.

   An alternative is adding more disks. This makes the server operate at a higher per-stream cost. As we will see in Section8.3, under-utilized disks are not as expensive as over-utilized disks, so the cost of storing the presentations may not increase cost too much beyond what is required for simply playing them back.

8.3 Choosing Disks

Symbol	*Disk Model*	*Avg/Max Seek*	*RPM*	*Worst Latency*	*Min/Max TR*	*Price*
2A	*Seagate ST-32107N*	*8.5/18*	*7200*	*26.33*	*47/88*	$349
2B	*Seagate ST-31051N*	*9/22*	*5400*	*33.11*	*44/66*	$349
4C	*IBM DCAS 34300*	*8.5/15*	*5400*	*26.11*	*40/NA*	$379
4D	*Micropolis 3242AV*	*8.9/16*	*7200*	*24.33*	*46/80*	$469
4E	*Micropolis 4345NS*	*7.9/15*	*7200*	*23.33*	*76/125*	$529
4F	*Seagate ST-14209N*	*9/18*	*7200*	*26.33*	*47/88*	$499
4G	*Seagate ST-34371N*	*9/17.5*	*7200*	*25.83*	*80/122*	$559
4H	*Seagate ST-23501N*	*7.7/18.3*	*10000*	*24.3*	*122/177*	$625
9I	*Seagate ST-410800*	*11/23*	*5400*	*24.11*	*44/65*	$659
9J	*Seagate ST-19171N*	*9.7/19.0*	*7200*	*27.32*	*80/124*	$959

Table 3: Disk Parameters and Prices

In this section we show how to select a disk model when a variety of choices are available. Table 3 lists the specifications and prices of some disks that we found on the Web [1] (April 1998). Although there are many more models available, and prices for the same disk often vary

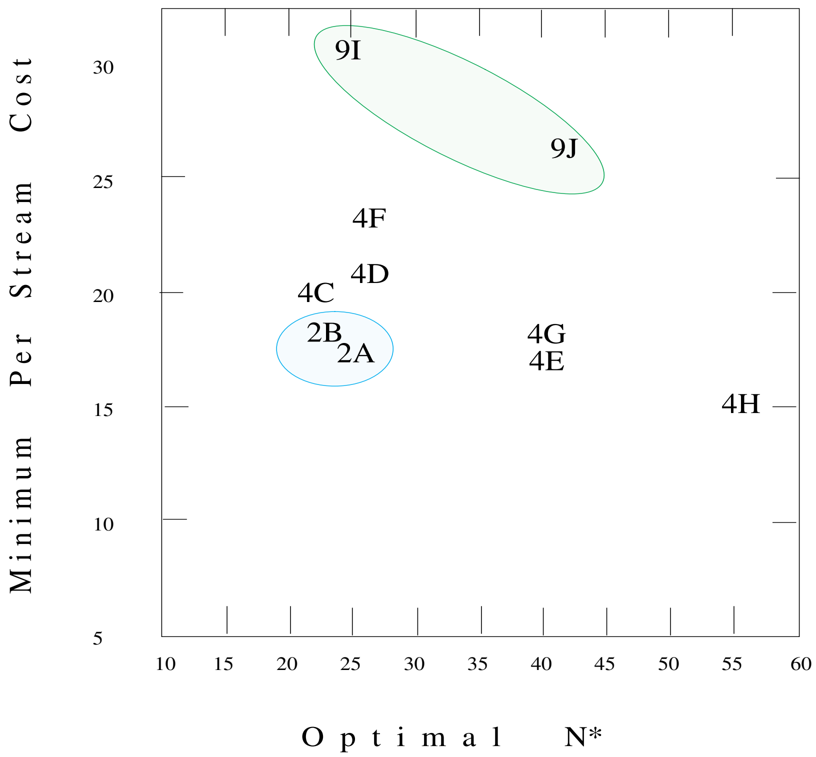


Figure 6: Per Stream Cost vs. N^*_{Limit} Samples

a lot, we believe that the ones in our table form a representative sample. Table 3 divides the disks into three groups based on their storage capacity: 2 GBytes, 4 GBytes, and 9 GBytes. In each group, the table sorts the disks by vendors and then by their prices. We use a two character symbol to represent each disk model: the first character indicates the storage capacity (e.g., 2 means 2 GBytes) and the second character the sequence of the disk in the table. For instance, the 4 GByte Seagate ST-14209 disk in the 6^{th} row of the table is represented by symbol $4F$. We continue to assume that the memory cost is \$5 per MByte. Figure 6 plots the minimum per-stream cost versus N^*_{Limit} for all disks listed in the table, using the minimum data transfer rate and the worst disk latency.

Figure 6 can guide the selection of the best disk, depending on whether total storage capacity is an issue:

- *Minimizing cost given a throughput requirement only.* If the storage capacity is not a concern, the best disk to buy is simply the one that yields the minimum per-stream cost, i.e., disk $4H$. Disks $2A$, $4E$, and $4G$ are close runners-up. All these disks share one common characteristic: they have both lower latency/price and higher TR/price ratios than their peers (shown in Table 4).

Disk	*Latency/Price* (10^{-6})	*TR/Price* (10^{-2})
2A	*75*	*13.5*
2B	*95*	*12.6*
4C	*69*	*10.5*
4D	*52*	*9.8*
4E	*44*	*14.3*
4F	*53*	*9.4*
4G	*46*	*14.3*
4H	*39*	*19.5*
9I	*52*	*6.6*
9J	*29*	*8.3*

Table 4: Latency/Price and TR/Price Ratios

- *Minimizing cost given both throughput and capacity requirements.* In this case, we first find the best candidates in each capacity group (i.e., 2, 4, and 9 GByte groups), then we find the globally optimal disk.

  Given a group of disks with the same storage capacity, we want to select a disk that not only has small per-stream cost, but also has a small N^*_{Limit}. The reason why a small N^*_{Limit} is preferable is because, if the per-stream cost of two disks are the same, the disk that has a smaller N^*_{Limit} can be bought in a larger quantity, and subsequently provides more storage space. Conversely, if two disks have the same N^*_{Limit}, the one with smaller per-stream cost is the choice. Using these two criteria, we can first eliminate some bad choices from each group. That is, we can eliminate a given disk if another disk in its category appears in the area below it or to its left in Figure 6. This is because these other disks either have a lower per-stream cost (with the same N^*_{Limit}), or a smaller N^*_{Limit} (with the same per-stream cost), or both. In Figure 6 we can eliminate from the 4 MByte category disks 4D, 4F, and 4G.

  To select from among the remaining candidates in the same capacity group, we need to know the total storage requirement. For example, comparing between disks 4C and 4H, we should select 4H because of its lower per-stream cost, as long as when bought in the optimal quantity, it yields sufficient storage space for our presentations. If it does not, disk 4C may be preferable. It may have a higher cost per stream, but since we get more of them at the optimal configuration (smaller N^*_{Limit}), we will have more total storage space.

  Finally, we need to select from which group we want to buy disks. In addition to the per-stream cost and N^*_{Limit}, we have to also consider the capacity of the disks. In Figure 6 we can eliminate disks 2A and 2B since they are close to disk 4C in both the per-stream cost and N^*_{Limit} but provide only a half of storage capacity. Also, disk 9J may be eliminated since 4C has a much smaller N^*_{Limit} and smaller per-stream cost. The choice between 4C and 9I, however, is not as obvious and requires knowledge of the exact disk space requirement (as was the case with the 4C, 4H choice).

In summary, a faster (e.g., disk 4G) or a larger disk (e.g., disk 9J) may not be the best choice for a server. To make a decision, one must take into account both the cost of the improving the storage or speed of a disk, and any additional cost for the memory needed to support the higher performance disk. The overall storage capacity of the disks, relative to the size of the presentations, is also an important factor. In the full version of the paper we will provide additional examples and a more detailed analysis of the disk selection process.

9 Reducing Initial Latency

In this section we consider a third important performance metric for multimedia storage systems. We define *initial latency* to be the time between the arrival of a *single* new request (when the system is unsaturated) and the time when its first data segment becomes available in the server's memory. In computing the initial latency we do not take into account any time spent by a request waiting because the system is saturated (with N_{Limit} streams), as this time could be unbounded no matter what scheduling policy is in place. In other words, our focus is on evaluating the worst initial delay when both disk bandwidth and memory resources are available to service a newly arrived request.

For scheme Sweep (and Sweep*), the worst initial latency happens when a request arrives just after the disk head has passed over the first segment of the media. The request must wait for a cycle (T) until its first segment can be retrieved. As discussed in Section 2.1, playback cannot start right away, since this first segment just fills up the playback cushion. Actual playback can start at the end of the first cycle, which in the worst case can be another T seconds away. The worst initial latency is therefore $T_{Latency} = 2 \times T$. Since $T = DR \times S$ and we have shown that S can grow without bound as N_{Limit} increases, $T_{Latency}$ can also grow without bound.

The initial latencies of scheme Sweep (and Sweep*) can be fairly significant, ranging from five to ten seconds according to the system configuration. This may not be a problem for multimedia application such as "movies on demand" where a delay of a few seconds before a new multi-hour movie starts may be acceptable. However, we believe that there are important applications where high latencies are not acceptable. For example, consider a video game where at each step the player's actions determine what short video to play next. Here we clearly do not want the player to wait a significant amount of time before each video scene starts. We could try to pre-fetch all the possible videos that might be selected, but this would significantly increase the server load and memory requirements. Another application that requires low latency is hypermedia documents, as found in the World Wide Web or a Digital Library. Here a user may examine a "page" that contains links to a variety of other pages, some of which may be multimedia presentations. Again, it is inefficient to pre-fetch all linked options, and the user does not want a significant delay between the time he clicks on a link and the time the presentation starts. Even with "movies on demand," high initial latency may imply poor response time for interactive features such as fast forward and rewind. Thus, we believe that initial latency is an important performance parameter, and our next scheme will reduce it significantly.

9.1 Policy BubbleUp

Scheme Fixed-Stretch* divides up a period into N_{Limit} equal duration *slots*. When a request arrives, it must be serviced in one of these slots, or the minimum memory requirement is violated. This means that in the worst case a new request arriving when the system is not saturated (i.e., when there is an unused slot) may have to wait roughly T seconds for the one unused slot to occur.

It is possible to modify scheme Fixed-Stretch* so that this initial latency is almost eliminated. We call this modification *policy BubbleUp* for reasons that will become apparent. In this section we simply illustrate the intuition behind it via an example. In [3, 6] we present scheme BubbleUp in much greater detail, including its data placement policy, its support of the fast-scan operations, and its scalability into managing parallel disks.

We use an example to illustrate how BubbleUp reduces initial latency. Assume that a media server supports up to three requests ($N_{Limit} = 3$) in a period T, as depicted in Figure 2. To service three requests in T, BubbleUp divides the period into three service slots, s_1, s_2, and s_3, each lasting time Δ.

Table 5 shows a sample execution for three requests R_1, R_2, R_3. Each row of the table represents the execution for one service slot. The first two columns identify the time involved. Each slot *instance* is labeled as Δ_i to remind us that its duration is Δ time units. Note that the first period or cycle has three slot instances (Δ_1, Δ_2, Δ_3). The first slot in a cycle, s_1, occurs in instances Δ_1, Δ_4, $\Delta_7, \ldots$. The last three columns in Table 5 show the requests that are *currently scheduled* for each slot in a cycle. For our example, let us assume that request R_1 arrives before Δ_1, R_2 arrives during Δ_2, and R_3 during Δ_4. When R_1 arrives the system is empty and it is scheduled into the first slot s_1. The following steps then occur under policy BubbleUp:

Period	*Time*	s_1	s_2	s_3
1	Δ_1	$*$ R_1		
1	Δ_2		$*$ R_1	
1	Δ_3		R_1	$*$ R_2
2	Δ_4	$*$ R_1		R_2
2	Δ_5	R_1	$*$ R_3	R_2

Table 5: BubbleUp Example

- Δ_1: During the first instance of s_1, Δ_1, request R_1 is serviced; this is indicated by the asterisk next to R_1. (Note that the asterisk moves from slot to slot as time progresses.) The amount of data retrieved for R_1 is S.
- Δ_2: The current slot is s_2, and no new request has arrived before the start of this time unit. Instead of staying idle, we service R_1 *again* in s_2. The amount of data retrieved for R_1 is $S/3$, since only $S/3$ has been consumed from the last "playback point" of R_1 to the playback point of this slot instance. Note that in Row 2 of Table 5, R_1 now has moved to s_2 and s_1 is empty. Empty slots s_3 and then s_1 are (roughly) Δ and 2Δ time units away, ready to accept any new requests that might arrive. This is the goal of the BubbleUp policy: trying to keep open slots as close time-wise as possible.
- Δ_3: The current slot is s_3, and request R_2 has just arrived (during Δ_2). We service R_2 immediately in s_3.
- Δ_4: Slot s_1 is the current slot, no new request has arrived before the start of this time unit. Policy BubbleUp selects the next future request that would be due for service, R_1 in this case, and services it early. Enough data is read in ($2S/3$) to fill up the R_1 buffer, and now R_1 has moved to slot s_1. Again, we have freed the following slot, s_2, to be able to service some new request that might arrive.

  tem Δ_5: The current slot is s_2, and request R_3 has arrived (during Δ_4). We service R_3 immediately. If we had not swapped R_1 out of this slot in Δ_4, R_3 would have had to wait two slots before receiving service.

Policy BubbleUp "bubbles up" empty slots in the future so they occur in the next slot. Thus, if there is a free slot, it always occurs in the very next time slot, and the maximum delay a new request faces is limited. In particular, the worst delay occurs if the new request arrives just after a slot has started. It then has to wait for the next slot ($\gamma(CYL) + S/TR$), plus the time for its own seek to complete ($\gamma(CYL)$). Thus, the worst case latency:

$$T_{Latency} = 2\gamma(CYL) + S/TR, \tag{18}$$

is independent of N_{Limit}. Also, notice that policy BubbleUp works only with scheme Fixed-Stretch* (and Fixed-Stretch), since only Fixed-Stretch* allows the disk arm to freely travel to

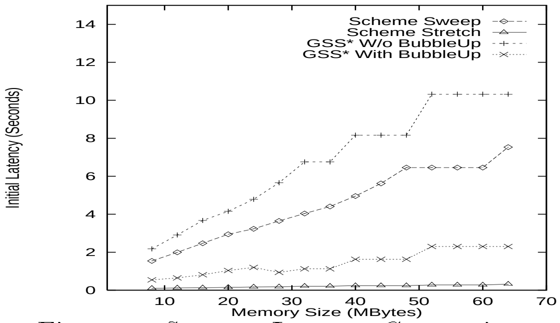


Figure 7: Startup Latency Comparison

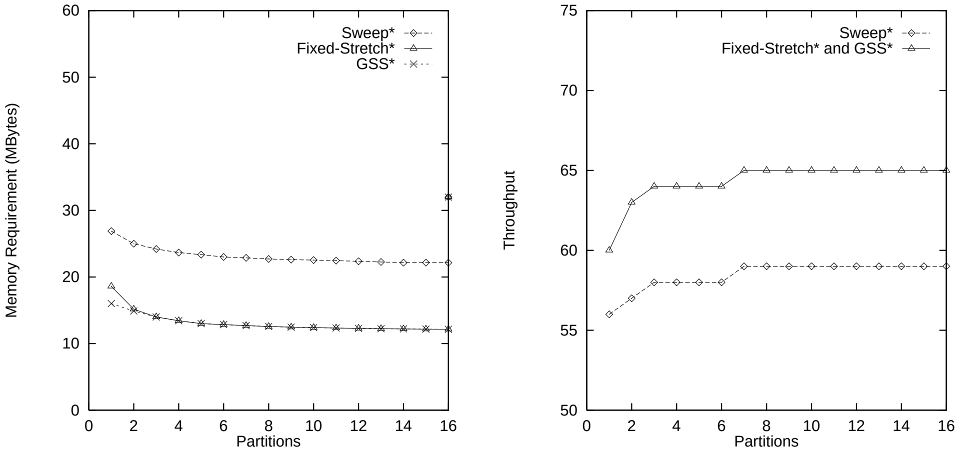
any desirable cylinder without causing "jitter".

Policy BubbleUp can also work with GSS*. If BubbleUp were not used, the worst initial latency of GSS* would happen when there is only one epoch with fewer than N_{Limit}/G requests, the sweep for that epoch has just started, and we just missed the opportunity to read the first segment of the newly arrived request. In this case we must wait a fully period T for that epoch to start again, and then up to T/G to retrieve the first segment of the new request. The worst initial latency would thus be $T_{Latency} = T + T/G$.

However, GSS* uses policy BubbleUp to reduce initial latency. Instead of moving the requests forward in slots as described for Fixed-Stretch*, the requests can swap their groups. Policy BubbleUp in this case keeps available slots in the upcoming group(s), available for servicing new requests. The worst latency then is the time to complete the current sweep plus another sweep to service the new request. The worst case initial latency is cut down to $T_{Latency} = 2T/G$.

Figure 7 presents the worst-case initial latency of four schemes under different memory configurations: Sweep* (equal to Sweep), Fixed-Stretch* (equal to Fixed-Stretch), GSS* without BubbleUp, and GSS* with BubbleUp. For both GSS* schemes, we use the G value that yields best throughput.

The initial latency of schemes Sweep and GSS* without BubbleUp is much larger because in both schemes the latency expression depends on N_{Limit}. On the other hand, the latencies for Fixed-Stretch* and GSS* with BubbleUp are minimal and almost constant. Notice the initial latency of GSS* without BubbleUp is worse than Sweep* due to its larger segment size. In a moderate work load (e.g., $N_{Limit} = 55\ to\ 63$), even though Fixed-Stretch* supports one less stream then GSS*, its initial latency is at least 2 seconds shorter than GSS* with BubbleUp. For interactive applications that are sensitive to delays, Fixed-Stretch* may be the choice.



(a) Mem_{Min} vs P,$N_{Limit} = 56$ (b) N_{Limit} vs P,$Mem_{Avail} = 32MBytes$

Figure 8: Disk Partition

10 Data Placement Policies

We have studied disk scheduling and memory policies and their impact on throughput, startup latency, and cost. To complete our study, this section discusses some data placement policies. We evaluate the impact of a placement policy, called disk partitions, on memory use and startup latency. We also discuss the layout of data across multiple disks.

10.1 Disk Partitions

Reference [12] proposes a partition scheme that divides a disk into P concentric regions. The idea is that in each period T the disk arm services only one region. Partitioning the disk into P regions reduces the worst seek distance by a factor of P.

For scheme Sweep*, the worst seek distance can be reduced from CYL/N_{Limit} to $\frac{CYL}{N_{Limit} \times P}$. For scheme Fixed-Stretch*, the worst seek distance is bound by $\frac{CYL}{P}$ rather than CYL. Seek times are also reduced by P for GSS*. The rest of the analysis for schemes Sweep*, Fixed-Stretch*, and GSS* is identical to what we already have, except that the reduced seek times are used. Notice that since the worst seek distance for scheme Sweep* is much shorter to start with, we expect the partition scheme to benefit Fixed-Stretch* (and GSS* with large G) more than it does Sweep*.

To illustrate the effect of partitions, we return to the case study of Section 7. Figure 8(a) shows the amount of memory required for up to 16 partitions at $N_{Limit} = 54$ (near the minimum per-stream cost) for schemes Sweep*, Fixed-Stretch*, and GSS*. Disk partitioning does save memory under each scheme. For instance, at $P = 2$, the memory savings are 22% for scheme Fixed-Stretch*, and about 7% for both GSS* and Sweep*. As expected, Fixed-Stretch* benefits more dramatically from partitions since it depends directly on the maximum seek distance. Notice that the gains for all schemes flatten out when $P > 6$.

Figure 8(b) plots the throughput achievable with 32 MBytes of available memory and up to 16 partitions. In terms of throughput, Fixed-Stretch* performs the same as GSS*. Again, Fixed-Stretch* (and GSS*) benefits more from partitions than Sweep* because it is more sensitive to the maximum seek overhead. For all schemes, however, disk partitions do not help too much in improving throughput. This is because even though disk partitions help save memory, the memory required to support additional requests is huge at the tail of the memory requirement curve (see Figure 4). Note that since initial latency grows with P [21], a disk partition scheme may not be suitable for interactive applications.

10.2 Multiple Disks

There are two common ways to allocate data when multiple disks are available in a system. With the first, which we call *independent disks*, a segment of a presentation is always stored within a single disk. (Although different segments of a presentation can be stored on multiple disks for the purpose of balancing workload.) Thus, when a segment of a presentation is retrieved, only one disk is involved in the transfer. If we playback presentations from different disks, their IOs can take place concurrently.

The second way to use disks, called *stripped disks* [10], treats a group of disks as one storage unit, with each segment broken into several subsegments, each stored on a separate disk. The time to transfer one segment into memory is reduced since the subsegments can be fetched in parallel. With striping, a group of disks services one request at a time.

Several factors must be considered in choosing between independent and stripped disks. For example, if we have a display rate that cannot be supported by a single disk, then stripping is a must. Also, independent disks may not work well if we cannot balance the load across them well, e.g., because presentations in one disk are much more popular than others [8, 21, 7]. (In [6] we survey schemes for managing parallel disks and propose an efficient policy which not only achieves balanced disk load, minimizes costs, but also minimizes the initial latency. Please refer to the reference for details.) However, from the point of view of memory utilization, which is the focus of our paper, independent disks are much superior under normal circumstances. The following theorem shows that with M disks stripping requires M times as much memory as independent disks for equivalent throughput.

Theorem 6: Say we are given M disks with equal transfer rate TR and we wish to support N_{Limit} requests. Assuming that we can balance the load with independent disks, stripping requires M times as much memory as independent disks do. Please refer to Appendix A.6 for the proof.

Notice that this result is independent of the scheduling scheme used. It shows that at least as far a memory is used, stripping is not desirable.

11 Conclusion

In this paper we have shown that disk latency reduction is secondary to optimizing memory use in video delivery schemes. Stretching out IOs with "artificial" delays for the disk surprisingly leads to much more effective memory use, and subsequently better throughput. This is because stretching out IOs minimizes the cushion buffer requirement and maximizes memory sharing among streams. In an analogous way, stop lights at freeway entrance ramps can slow down input traffic, and lead to better throughput. Of course, the reason why traffic lights work in a freeway is different from why a scheme like Stretch works, but intuitively the result is the same: pacing inputs can improve throughput.

We also propose policy BubbleUp which can reduce startup latency to the time of servicing one request, independent of the number of requests in the system. Low startup latency enables interactive multimedia applications, making scheme that stretching out IOs even more desirable.

As a part of our evaluation, we have noted that achieving high throughput on a disk often comes at a huge cost in memory. Most research in the area has tended to ignore this, focusing on how to reduce seek overheads. Instead, we have proposed to limit throughput per disk to less than what is feasible to minimize the per-stream cost. As a consequence, given a fixed budget, the server can be configured to accomplish the maximum throughput.

References

[1] Bason computer, inc. *URL: http://www.baconcomputer.com*, November 1997.

[2] Seagate barracuda 9lp family product specification. *URL: http://www.seagate.com*, April 1998.

[3] E. Chang and H. Garcia-Molina. Bubbleup - low latency fast-scan for media servers. *Proceedings of the 5th ACM Multimedia Conference*, pages 87–98, November 1997.

[4] E. Chang and H. Garcia-Molina. Effective memory use in a media server. *Proceedings of the 23rd VLDB Conference*, pages 496–505, August 1997.

[5] E. Chang and H. Garcia-Molina. Cost-based media server design. *Proceedings of the 8th Research Issues in Data Engineering*, Feburary 1998.

[6] E. Chang, H. Garcia-Molina, and C. Li. 2d bubbleup - managing parallel disks for media servers. *Stanford Technical Report SIDL-WP-1998-0090*, February 1998.

[7] M.-S. Chen, H.-I. Hsiao, C.-S. Li, and P. Yu. Using rotational mirrored declustering for replica placement in a disk-array-based video server. *ACM Multimedia Systems*, December 1997.

[8] S. Christodoulakis and F. Zioga. Data base design principles for striping and placement of delay-sensitive data on disks. *Proc. of PODS (to appear)*, 1998.

[9] T. Chua, J. Li, B. Ooi, and K.-L. Tan. Disk striping strategies for large video-on-demand servers. *ACM Multimedia*, pages 297–306, November 1996.
[10] H. Garcia-Molina and K. Salem. Disk striping. *ICDE*, pages 336–342, Feburary 1986.
[11] S. Ghandeharizadeh, S. Kim, and C. Shahabi. On configuring a single disk continuous media server. *Sigmetrics Performance Evaluation*, 23(1):37–46, May 1995.
[12] S. Ghandeharizadeh, S. Kim, and C. Shahabi. On disk scheduling and data placement for video servers. *USC Technical Report USC-CS-TR97-650*, December 1995.
[13] D. Kotz, S. B. Toh, and S. Radhakrishnan. A detailed simulation model of the hp 97560 disk drive. *Dartmouth College Technical Report PCS-TR94-220*, 1994.
[14] D. Makaroff and R. Ng. Schemes for implementing buffer sharing in continuous-media systems. *Information Systems*, 20(6):445–464, 1995.
[15] B. Ozden, A. Biliris, R. Rastogi, and A. Silberschatz. A low-cost storage server for movie on demand databases. *Proc. VLDB*, September 1994.
[16] A. Reddy and J. Wyllie. I/o issues in a multimedia system. *Computer*, 2:69–74, March 1994.
[17] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *Computer*, 2:17–28, 1994.
[18] R. Steinmetz. Multimedia file systems survey: approaches for continuous media disk scheduling. *Computer Communications*, pages 133–44, March 1995.
[19] F. Tobagi, J. Pang, R. Baird, and M. Gang. Streaming raid-a disk array management system for video files. *First ACM Conference on Multimedia*, August 1993.
[20] H. M. Vin. and P. V. Rangan. Designing a multi-user hdtv storage server. *IEEE Journal on Selected Areas in Communication*, 11(1), January 1993.
[21] J. Wolf, , P. Yu, and H. Shachnai. Disk load balancing for video-on-demand systems. *ACM Multimedia Systems*, December 1997.
[22] P. Yu, M.-S. Chen, and D. Kandlur. Grouped sweeping scheduling for DASD-based multimedia storage management. *Multimedia Systems*, 1(1):99–109, January 1993.

Appendix A: Proof of Theorems

A.1: Proof of Theorem 1

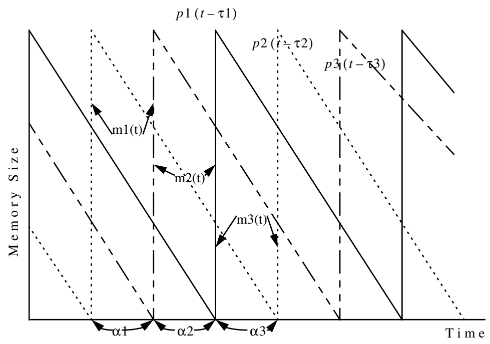


Figure 9: $p(t)$

Consider the periodic function $p(t)$ shown in Figure 9 that is composed of three $p_i(t-\tau_i)'s$, $i = 1\ to\ 3$. Since $p(t)$ is periodic, we can just examine the first period of it from time 0 to T. The time between IOs are denoted as α_1, α_2, and α_3, where the sum of the $\alpha's$ is T. Because $p(t)$ is discontinuous, each period of $p(t)$ is an aggregate of three continuous function $m1(1)$, $m2(t)$, and $m3(t)$ that represent three regions $[0, \alpha_1)$, $[\alpha_1, \alpha_1+\alpha_2)$, and $[\alpha_1+\alpha_2, \alpha_1+\alpha_2+\alpha_3)$ respectively. Thus, $p(t)$ is rewritten as

$$t \in [0, \alpha_1) \quad p(t) = m1(t) = DR \times (T-t) + DR \times (T-t-\alpha_2-\alpha_3) + DR \times (T-t-\alpha_3)$$

$$t \in [\alpha_1, \alpha_1+\alpha_2) \quad p(t) = m2(t) = DR \times (T-t) + DR \times (T-t+\alpha_1) + DR \times (T-t-\alpha_3))$$

$$t \in [\alpha_1+\alpha_2, \alpha_1+\alpha_2+\alpha_3) \quad p(t) = m3(t) = +DR\times(T-t)+DR\times(T-t+\alpha_1)+DR\times(T-t+\alpha_1+\alpha_2)$$

Simplifying above expression, we get

$$t \in [0, \alpha_1) \quad m1(t) = 3DR(T-t) - \alpha_2 DR - 2\alpha_3 DR$$

$$t \in [\alpha_1, \alpha_1+\alpha_2) \quad m2(t) = 3DR(T-t) - \alpha_3 DR + \alpha_1 DR$$

$$t \in [\alpha_1+\alpha_2, \alpha_1+\alpha_2+\alpha_3) \quad m3(t) = 3DR(T-t) + 2\alpha_1 DR + \alpha_2 DR$$

Because m1(t), m2(t), and m3(t) are monotonically decreasing, the maximum values are at the beginning of the intervals, or

$$Max\ m1(t) = DR(3T - \alpha_2 - 2\alpha_3)$$

$$Max\ m2(t) = DR(3T - \alpha_3 - 2\alpha_1)$$

$$Max\ m3(t) = DR(3T - \alpha_1 - 2\alpha_2)$$

Since DR is a positive constant, dividing both sides of the equations by DR does not affect the maximization. In addition, to simplify the computation we subtract $3T$ from all equations, and flip their signs. Due to the sign change, we must then replace Max with Min, and hence the objective becomes maximizing the minimum values of m'(1), m'2(t), and m'3(t). This can be written as:

$$Maximize\ (Min\ m'1(t),\ Min\ m'2(t),\ Min\ m'3(t)),\ where$$

$$Min\ m'1(t) = \alpha_2 + 2\alpha_3$$

$$Min\ m'2(t) = \alpha_3 + 2\alpha_1$$

$$Min\ m'3(t) = \alpha_1 + 2\alpha_2,\ and$$

$$\alpha_1 + \alpha_2 + \alpha_3 = T.$$

Replacing α_3 with $T - \alpha_1 - \alpha_2$, we have

$$Maximize\ (Min\ m'1(t),\ Min\ m'2(t),\ Min\ m'3(t)),\ where$$

$$Min\ m'1(t) = 2T - 2\alpha_1 - \alpha_2$$

$$Min\ m'2(t) = T + \alpha_1 - \alpha_2$$

$$Min\ m'3(t) = \alpha_1 + 2\alpha_2$$

$$0 \le \alpha_1, \alpha_2 \le T.$$

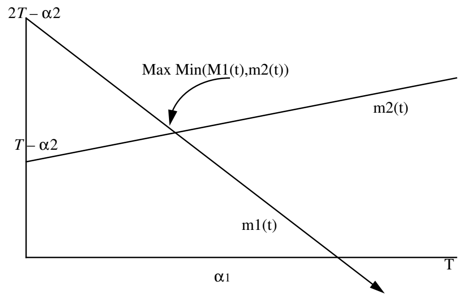


Figure 10: Maximize the Minimum Values of m'1(t) and m'2(t)

Assume α_2 is known. To find the optimal α_1 one must find the α_1 that makes $Min\ m'1(t)$ equal to $Min\ m'2(t)$, as illustrated in Figure 10. When $Min\ m'1(t) = Min\ m'2(t)$, $3\alpha_1$ is equal to T. Therefore, $\alpha_1 = T/3$. Substitute α_1 back into $Min\ m'1(t)$ and $Min\ m'2(t)$ gets $\alpha_2 = T/3$, and subsequently $\alpha_3 = T/3$.

We have therefore proven for $N = 3$ that the condition that minimizes the memory requirement is to separate IOs equally in each period. This procedure of proof is applicable for any given $N's$. $\square$

A.2: Proof of Corollary 1

With N_{Limit} requests, $p(t)$ is

$$p(t) = \sum_{i=1}^{N_{Limit}} p_i(t - \tau_i),$$
$$where \quad p_i(t-\tau_i) = DR \times T - DR \times (t - \tau_i).$$

The minimum memory requirement is the maximum value of $p(t)$. Since Theorem 1 shows that spreading out IOs evenly in T minimizes the total memory requirement, we substitute the proper $\tau_i's$ into function $p(t)$ to get

$$p(t) = \sum_{i=1}^{N_{Limit}} p_i(t - \frac{i \times T}{N_{Limit}}),$$
$$\text{where} \quad p_i(t - \frac{i \times T}{N_{Limit}}) = DR \times T - DR \times (t - \frac{i \times T}{N_{Limit}}). \tag{19}$$

Because $p_i(t-\tau_i)'s$ are monotonically decreasing, the start time of IOs gives the maximum value of $p(t)$. In addition, since all $p_i(t-\tau_i)'s$ have the same shape due to the same display rate, all IO start times give the same maximum value. Without loss of generality, let us pick $t = T$ and substitute $t = T$ back into Equation 19, obtaining

$$Max \quad p(t) = \frac{DR \times T}{N_{Limit}} \times \sum_{i=1}^{N_{Limit}} i$$
$$\text{or} \quad Max \quad p(t) = DR \times T \times (N_{Limit} + 1)/2.$$

Since $DR \times T = S$, the above expression is equivalent to $S \times (N_{Limit} + 1)/2$. This gives us our minimum memory requirement. $\Box$

A.3: Proof of Theorem 2

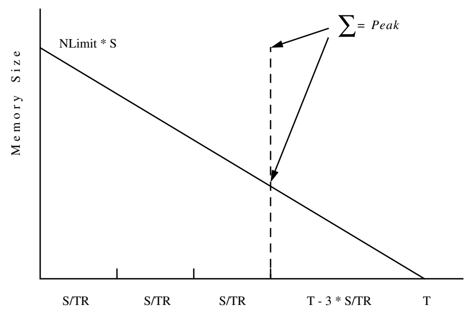


Figure 11: Memory Requirement for Sweep*

Figure 11 shows the memory requirement in a period. The horizontal axis shows the time in a period, while the vertical axis shows the memory required by Sweep*. At the beginning of each period, for each stream we need to have enough data in memory to sustain it for a full period, i.e., we need $S = T \times DR$ data per stream, or a total of $S \times N_{Limit}$. Call this the *old* data. During the period, all this old data will be played back, but we will also read in, at various points within the period, new data. All the new data is simply accumulated during the period, and not played back. At the end of the period, we have played back all of the old data, and the new data amounts to $S \times N_{Limit}$, and we start over the process.

The memory required at any given time in a period is the old data that has not been played back, plus any new data that has been read in. In the worst case, the new data arrives as early as possible, without giving us a chance to drain out much of the old data. This occurs if the IOs for the period occur at the beginning of the period, only separated by their transfer times (and no seek overhead), i.e., separated by (S/TR) time.

If the first of the IOs takes place at the very beginning of the period, then the last one of the IOs will occur, in this worst case scenario, at time $(N_{Limit} - 1) \times S/TR$ (see Figure 11). This leaves $T - ((N_{Limit} - 1) \times S/TR)$ time remaining in the period. Since the data consumption rate (or memory release rate) for N_{Limit} streams is $N_{Limit} \times DR$, the unreleased old memory at that point is

$$N_{Limit} \times DR \times (T - ((N_{Limit} - 1) \times S/TR)).$$

To this we add the new memory that has been read at that point, to obtain the minimum memory requirement of scheme Sweep*:

$$N_{Limit} \times S + N_{Limit} \times DR \times (T - \frac{(N_{Limit} - 1) \times S}{TR}). \tag{20}$$

By delaying the last IO to the end of the period, we can reduce the memory requirement. Putting off the last IO at the end of the period does not violate the continuity constraint, and is always feasible to do. This means, in the worst case, we only have $N_{Limit} - 1$ IOs pack together in the beginning of the period. This means, at worst, $(N_{Limit} - 1) \times S$ amount data are read in when there is $T - ((N_{Limit} - 2) \times S/TR)$ time remaining in the period. Since the data consumption rate (or memory release rate) for N_{Limit} streams is $N_{Limit} \times DR$, the unreleased old memory at that point is

$$N_{Limit} \times DR \times (T - ((N_{Limit} - 2) \times S/TR)).$$

Adding this to $(N_{Limit} - 1) \times S$, we have the reduced memory requirement as

$$(N_{Limit} - 1) \times S + N_{Limit} \times DR \times (T - \frac{(N_{Limit} - 2) \times S}{TR}). \tag{21}$$

□

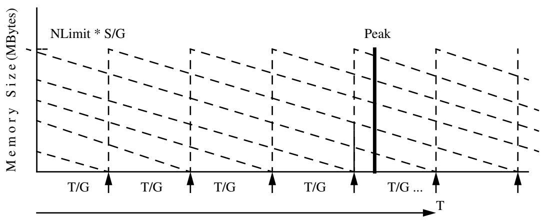


Figure 12: Memory Requirement of GSS*

A.4: Proof of Theorem 3

Here we derive the minimum memory requirement for GSS*. Let us assume that $G \geq 2$. (For $G = 1$ we can use the analysis of Fixed-Stretch.) Figure 12 illustrates memory use in GSS*, where each small up arrow on the horizontal time axis indicates the end of an epoch. At the end of each epoch, we need to have in memory enough data to sustain that group of N_{Limit}/G requests until the next epoch. Since we do not know at what time during the next epoch the IOs will occur, we need to plan for the worst case, so we need a total of $S \times N_{Limit}/G$ data at the end of each epoch. The dotted lines in Figure 12 show this peak memory for each epoch, and show how it is consumed over the next T time units.

Of course, memory use is not this regular, because the data read in an epoch does not show up in memory exactly at the end of the epoch, as the figure suggests. However, let us first compute the memory required as if new data arrives at the end of an epoch, and we will then compensate for the early arrival of some data. Given the regular memory pattern of Figure 12, we can use Corollary 1 to compute the memory requirement at the end of each epoch as

$$(N_{Limit}/G) \times S \times \frac{G+1}{2}.$$

To compensate for early data arrival, let us focus on the epoch in Figure 12 containing the heavy vertical line. In the worst case, the IOs for this epoch will not occur at the very end of the epoch, but will occur at the beginning of the epoch, separated by the IO transfer time only, S/TR. (For now, let us ignore the fact that the last IO of an epoch does occur at the end of the epoch.) This means that the peak memory use for the requests of this epoch will occur where the heavy vertical line is in Figure 12, when the time remaining for the end of the epoch is $T/G - ((N_{Limit}/G - 1) \times S/TR)$. Since no IOs take place between this point and the end of the epoch, the difference in the memory used at this point and the memory in use at the end of the epoch is simply the amount of data that N_{Limit} streams can consume, or $N_{Limit} \times DR \times (T/G - ((N_{Limit}/G - 1) \times S/TR))$. This extra memory needs to be added to the peak we had computed at the end of the epoch. Thus, the peak memory use at the heavy line in Figure 12 is

$$(N_{Limit}/G) \times S \times \frac{G+1}{2} + N_{Limit} \times DR \times (T/G - (N_{Limit}/G - 1)\frac{S}{TR}). \qquad (22)$$

This would be the total memory requirement, except that we have not taken into account that the last IO for an epoch takes place at the end of the epoch in GSS*. (Actually, it may not be at the very end of the epoch because of the rotational delay uncertainty. Here we ignore this small factor, although it could be taken into account as we did for scheme Fixed-Stretch.) This means that the time between the heavy line and the end of the epoch is $T/G - ((N_{Limit}/G - 2) \times S/TR)$. This causes us to have more unconsumed data at the heavy line. However, we can subtract S from that peak because the last IO does not take place until the end of the epoch. Thus, the total memory requirement can be obtained by replacing the $(N_{Limit}/G - 1)$ term in Equation 22 by $(N_{Limit}/G - 2)$ and by subtracting S from the overall memory use. Thus, we obtain:

$$(N_{Limit}/G) \times S \times \frac{G+1}{2} - S + N_{Limit} \times DR \times (T/G - (N_{Limit}/G - 2)\frac{S}{TR}). \qquad (23)$$

Note that because $TR \geq N_{Limit} \times DR$, the amount in Equation 23 will always be smaller than that in Equation 22.

A.5: Proof of Theorem 5

Let C_d denote the cost of a disk, and C_m the cost of one MByte of memory. Since each stream of the Sweep scheme requires $2 \times S$ amount of memory. The per-stream cost function, denoted as $C_s(N_{Limit})$, can be written as

$$C_s(N_{Limit}) = C_d/N_{Limit} + 2 \times C_m \times S. \qquad (24)$$

Substituting for S the right hand side of Equation 6 we get

$$C_s(N_{Limit}) = C_d/N_{Limit} + 2 \times C_m \times \frac{N_{Limit} \times \gamma(cyl_i) \times TR \times DR}{TR - (DR \times N_{Limit})}. \qquad (25)$$

To minimize $C_s(N_{Limit})$, we take the first derivative of Equation 24 and set it to zero; we get

$$-C_d N_{Limit}^{-2} + 2 \times C_m \times DR \times TR \times \gamma(cyl_i) \times (TR - DR \times N_{Limit})^{-1}$$
$$+2 \times C_m \times DR^2 \times TR \times \gamma(cyl_i) \times N_{Limit} \times (TR - DR \times N_{Limit})^{-2} = 0.$$

Multiplying both sides by $N_{Limit}^2 \times (TR - DR \times N_{Limit})^2$ we have

$$-C_d \times (TR - DR \times N_{Limit})^2$$

$$+2 \times C_m \times DR \times TR \times \gamma(cyl_i) \times N_{Limit}^2 \times (TR - DR \times N_{Limit})$$

$$+2 \times C_m \times DR^2 \times TR \times \gamma(cyl_i) \times N_{Limit}^3 = 0.$$

Rearranging terms we obtain

$$(2 \times C_m \times DR \times TR^2 \times \gamma(cyl_i) - C_d \times DR^2) \times N_{Limit}^2$$

$$+2 \times C_d \times DR \times TR \times N_{Limit} - C_d \times TR^2 = 0.$$

Since $C_s(N_{Limit})$ is convex, by solving for N_{Limit} and retaining the positive root, we obtain the optimal N_{Limit}^* that minimizes $C_s(N_{Limit}^*)$ as

$$N_{Limit}^* = \frac{TR}{DR} \times \frac{TR \times \sqrt{2 \times C_d \times C_m \times DR \times \gamma(cyl_i)} - C_d \times DR}{2 \times C_m \times TR^2 \times \gamma(cyl_i) - C_d \times DR}. \tag{26}$$

□

A.6: Proof of Theorem 6

With independent disks, each disk must support N_{Limit}/M requests. In this case, the segment size is given by Equations 6, 10 or 12 (substituting N_{Limit} by N_{Limit}/M):

$$S = \frac{\frac{N_{Limit}}{M} \times \gamma(cyl_i) \times TR \times DR}{TR - (DR \times \frac{N_{Limit}}{M})}, \tag{27}$$

where cyl_i is the scheme dependent worst seek distance. On the other hand, with stripping we have a "single" disk with transfer rate $TR \times M$ that must handle the N_{Limit} requests. Thus, the required segment size is

$$S' = \frac{N_{Limit} \times \gamma(cyl_i) \times TR \times M \times DR}{TR \times M - (DR \times N_{Limit})}.$$

If we multiply Equation 27 by M we see that $S' = S \times M$. □