Effective Memory Use in a Media Server

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

February 12, 1997

Abstract

A number of techniques have been developed for maximizing disk bandwidth utilization in media servers, including disk arm scheduling and data placement ones. Instead, in this paper we focus on how to efficiently utilize the available memory. We present techniques for best memory use under different disk policies, and derive precise formulas for computing throughput and memory use. We show that with proper memory use, maximizing disk bandwidth utilization does not necessarily lead to optimal throughput. In addition, we study the worst-case initial latency of presentations, a factor that may be important in interactive systems. We present a technique for significantly reducing this initial latency under some disk scheduling schemes.

Keywords: multimedia, disk scheduling, memory utilization.

1 Introduction

The storage system of a multimedia system faces more challenges than a conventional one. First, media data must be retrieved from the storage system at a specific rate — if not, the system exhibits "jitter" or "hiccups" [6]. This timely data retrieval requirement is also referred to as the continuous requirement or the real-time constraint [16]. 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 multimedia storage 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 worst-case initial latency before a new media request can be satisfied, and the per stream costs. 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 memory management policy, *BubbleUp*, that can reduce initial latencies by about an order of magnitude, without impacting throughput or memory requirements. Finally, as a part of our evaluation, we have noted 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 reduce seek overheads. Instead, we have proposed to limit throughput to less than what is feasible in order to minimize per stream costs.

The rest of this paper is organized into eight 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 a Group Sweeping Scheme (GSS) [17], 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 paper). In Section 6 we compare the schemes in a realistic case study, highlighting the basic disk bandwidth and memory use tradeoff.

Finally, Sections 7 and 8 analyize initial latencies and data placement policies that are not considered in the first six sections. We describes our novel memory 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.

2 Scheme Sweep

In this section we briefly describe a well known multimedia 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 [4, 6, 10, 12, 14] that optimize throughput by reducing disk seek overhead. Study [12] 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 8.) 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.
- 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.

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 number of requests
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

Table 1: Parameters

- 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, 15], 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 7.)

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}.

2.1.1 Maximizing Throughput

From our optimal performance equations we can derive a closed form for N_{Limit} as follows. We call the result N_{Max} to clarify that this is the maximum throughput for a given system. Substituting $T = S/DR$ (Eq. 1) and $S = Mem_{Avail}/(2 \times N_{Limit})$ (Eq. 5) into Equation 4, we obtain

$$N_{Limit} = \frac{Mem_{Avail} \times TR}{(2 \times N_{Limit} \times \gamma(CYL/N_{Limit}) \times TR + Mem_{Avail}) \times DR}. \tag{6}$$

Dividing both sides by $TR \times DR$, and moving all terms to one side, we have

$$2 \times \gamma(CYL/N_{Limit}) \times N_{Limit}^2 + \frac{Mem_{Avail}}{TR} \times N_{Limit} - \frac{Mem_{Avail}}{DR} = 0.$$

Selecting the only positive root for this second order polynomial, and selecting the closest smaller integer value we get an expression for the maximum throughput:

$$N_{Max} = \left\lfloor \frac{\sqrt{(\frac{Mem_{Avail}}{TR})^2 + \frac{8 \times \gamma(CYL/N_{Limit}) \times Mem_{Avail}}{DR}} - \frac{Mem_{Avail}}{TR}}{4 \times \gamma(CYL/N_{Limit})} \right\rfloor. \tag{7}$$

2.2 Minimizing Memory

Instead of maximizing throughput for a given amount of memory (Mem_{Avail}), we can also try to determine the minimum amount of memory, Mem_{Min}, required to support some desired

N_{Limit}. To obtain this, 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:

$$S = \frac{N_{Limit} \times \gamma(CYL/N_{Limit}) \times TR \times DR}{TR - (DR \times N_{Limit})}. \tag{8}$$

(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,

$$Mem_{Min} = \frac{2 \times N_{Limit}^2 \times \gamma(CYL/N_{Limit}) \times TR \times DR}{TR - (DR \times N_{Limit})}. \tag{9}$$

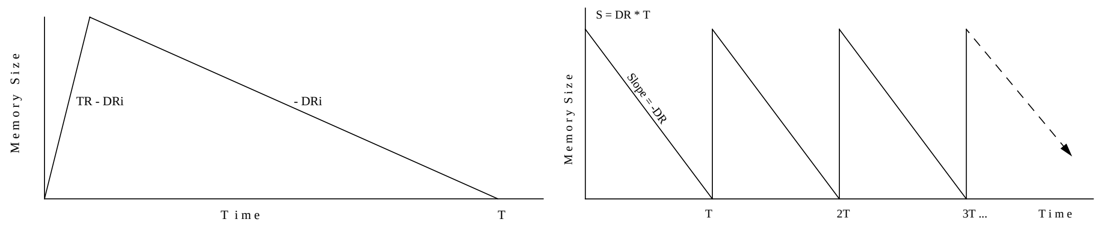
It is important to note in Equation 9 that Mem_{Min} does not grow linearly with the desired N_{Limit}. First, the numerator grows quadratically with N_{Limit}. Second, as N_{Limit} grows the denominator of Equation 9 approaches zero, causing Mem_{Min} to grow without bound. As the denominator gets close 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 Section 6.3, even if the system can support N_{Limit} concurrent streams, doing so is not cost effective.

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 [9, 15]. 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.

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



(a) Memory Required in A Period (b) Memory Use Function $p_i(t - \tau_i)$

Figure 1: Memory Requirement Function

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]

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

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

[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 [9].

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.

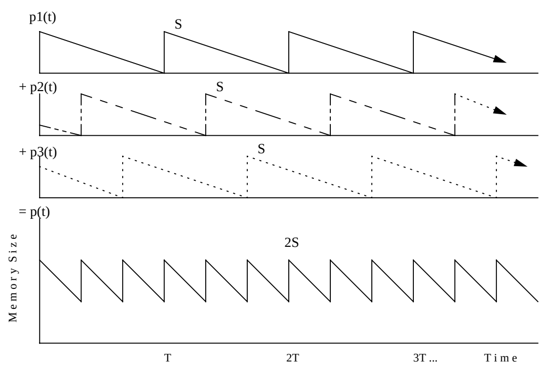
Corollary 1: The minimum memory space required to support N_{Limit} streams with equal display rate DR is $\frac{S \times (N_{Limit}+1)}{2}$. (Keep in mind that this does not include cushion buffer requirements.)

We provide in Appendix A.1 a proof of Theorem 1 for the special case $N_{Limit} = 3$. That proof can be easily generalized to more streams. (In [2] we prove a stronger version of the theorem that involved different display rates.) The proof of Corollary 1 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.

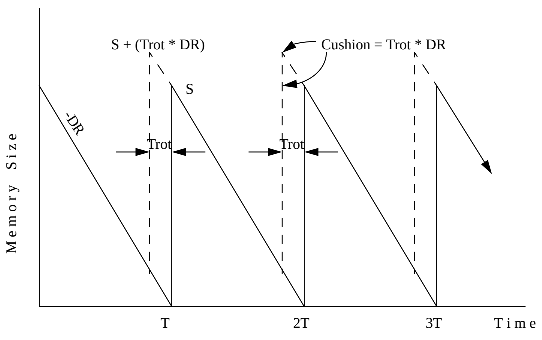
3.2 Scheme Sweep*

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 $(N_{Limit} - 1) \times S + N_{Limit} \times DR \times (T - \frac{(N_{Limit}-2) \times S}{TR})$. If we replace the right hand side of Equation 5 by this expression, we can derive formulas for the throughput and memory requirement of Sweep*.



(a) Minimum Memory Requirement



(b) Rotational Delay Cushion

Figure 2: Fixed-Stretch* Memory Requirement

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. Figure 2(a) shows an example with three requests that thusly separated. 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 $i^{\rm th}$ 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.

The goal for scheme Fixed-Stretch* is that IOs for a particular request are separated by exactly T time, as illustrated by the solid curve of Figure 2(b). The next request (not shown) will be shifted by Δ from this one, yielding minimum memory usage. Unfortunately, it is not possible separate the IOs *exactly* by the universal separator. In particular, say we wish to perform an IO at time T shown in the figure. At time $T - \Delta$ the previous IO has completed, so we then seek to the position of the next segment, without actually starting the transfer. Since

Δ can accommodate any possible seek distance, we will complete the seek in time. However, we may also get to the destination early, so we wait until time T approaches. Since we still have an unpredictable rotational delay before the data starts arriving, we need to initiate the IO at time $T - T_{Rot}$, where T_{Rot} is the maximum rotational delay. The data may arrive at time T as desired, but it may also arrive up to T_{Rot} seconds early, as shown by the dotted line in Figure 2(b). In the worst case this premature arrival will occur with each transfer. This requires a cushion of size $T_{Rot} \times DR$ for each request, (instead of one of size S for Sweep) to deal with IO time variability. This cushion is quite small; for instance, for the parameters values of Section 6, the cushion is 0.16% of the segment size.[3]

Scheme Fixed-Stretch* 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. 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. Furthermore, as we will see in Section 7, the very regular access pattern of Fixed-Stretch* also leads to a very small initial latency, much better than that of the seek reduction schemes.

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

$$T = N_{Limit} \times \Delta = T_{Seek} + T_{Delay} + T_{Transfer} = N_{Limit} \times (\gamma(CYL) + \frac{S}{TR}), \qquad (10)$$

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, plus any cushion buffer space. In this case, the cushion space is $T_{Rot} \times DR$ per request. 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,

[3]Another way to handle rotational delay variability is by allocating one dedicated system buffer (size S) to hold data as it arrives from disk. After a segment is read into this buffer, the system waits until the full Δ seconds have expired, and then copies the data to the playback area. This eliminates the need for the cushions we described, but does require a fixed S overhead, which is amortized over all N_{Limit} requests.

we have the memory resource constraint

$$Mem_{Avail} = \frac{S \times (N_{Limit} + 1)}{2} + (N_{Limit} \times T_{Rot} \times DR), \tag{11}$$

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, 10, 11) for optimal performance. From these equations we can solve for T, S, and N_{Limit}. Following the same derivation steps in Sections 2.1.1 and 2.2, we get the maximum throughput and minimum memory requirement as follows:

$$N_{Max} = \left\lfloor \frac{\sqrt{(\frac{Mem_{Avail}}{TR})^2 + \frac{2 \times \gamma(CYL) \times Mem_{Avail}}{DR}} - \frac{Mem_{Avail}}{TR}}{\gamma(CYL)} \right\rfloor \tag{12}$$

$$Mem_{Min} = \frac{S \times (N_{Limit} + 1)}{2}, \tag{13}$$

$$\text{where} \quad S = \frac{N_{Limit} \times \gamma(CYL) \times TR \times DR}{TR - (DR \times N_{Limit})}.$$

Please see Appendix B for the detailed derivations. If memory is *not* shard among the requests, we simply replace Equation 11 with

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

We call this scheme Fixed-Stretch (without the *), and the steps to derive N_{Max} and 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 [17] 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 [17]. (Our GSS* also uses a variant of BubbleUp that reduces initial latency; see Section 7.) 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:

$$S = \frac{N_{Limit} \times \gamma(CYL \times G/N_{Limit}) \times TR \times DR}{TR - (DR \times N_{Limit})}. \qquad (14)$$

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

$$(N_{Limit}/G) \times S \times \frac{G+1}{2} - S + N_{Limit} \times DR \times (T/G - (N_{Limit}/G - 2)\frac{S}{TR}). \qquad (15)$$

Please refer to Appendix A.4 for the proof.

6 Evaluation

To evaluate and compare the performance about various schemes discussed in this paper, we use the Seagate Barracuda 4LP disk [1]; 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] that is proven to be asymptotically close to the real disks. The seek overhead function is a concave function as following:

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

Parameter Name	*Value*
Disk Capacity	*2.25 GBytes*
Number of cylinders, CYL	*5,288*
Min. Transfer Rate TR	*75 Mbps*
Max. Rotational Latency Time	*8.33 milliseconds*
Min. Seek Time	*0.9 milliseconds*
Max. Seek Time	*17.0 milliseconds*
$\alpha 1$	*0.6 milliseconds*
$\beta 1$	*0.3 milliseconds*
$\alpha 2$	*5.75 milliseconds*
$\beta 2$	*0.0021 milliseconds*

Table 2: Seagate Barracuda 4LP Family Disk Parameters

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

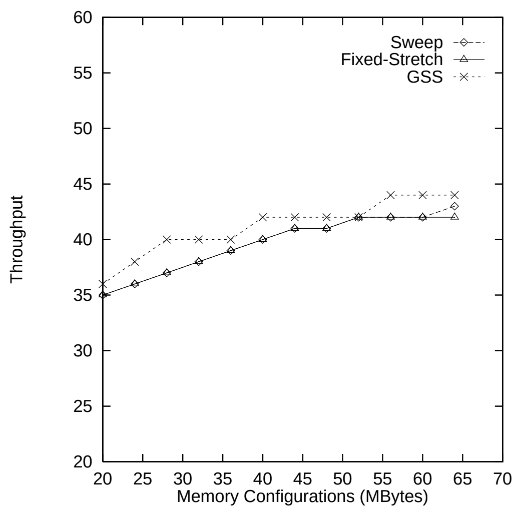
Note that the seek time is proportional to the square root of the seek distance when the distance is small, and is linear to the seek distance when the distance is large. We derive the parameters for this function as follows. We first allocate 2/3 of the minimum seek time provided by the vendor (0.9 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 (17 ms), and so that function γ is continuous at $d = 400$. The values obtained are given in Table 2.[4]

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.

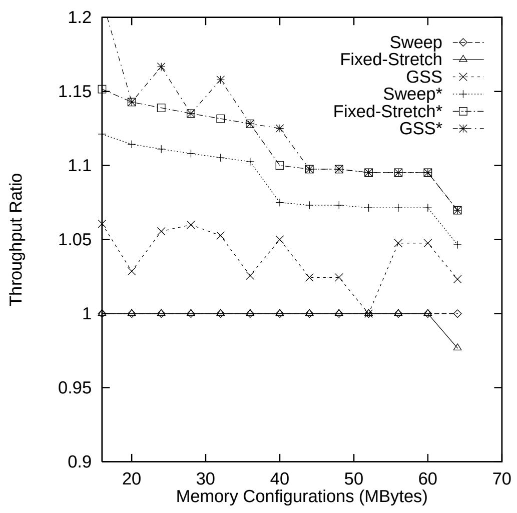
6.1 Throughput

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

[4]Incidentally, [13] 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

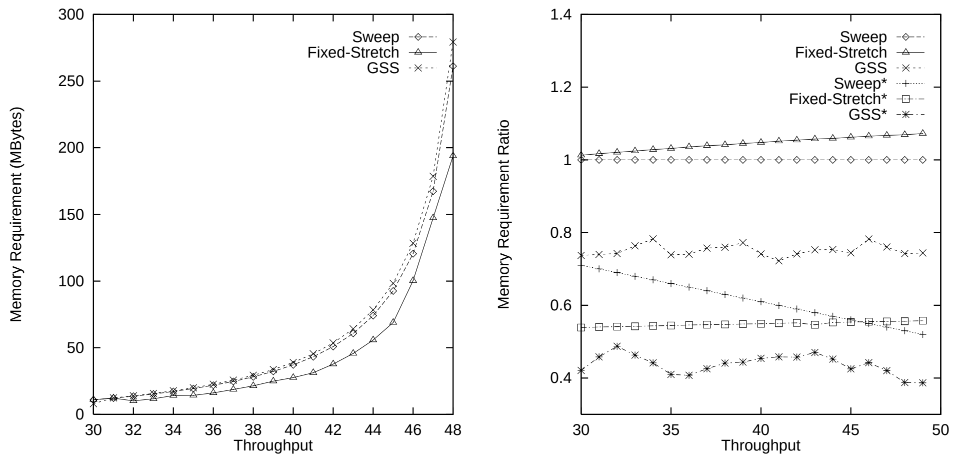
achieved by the optimal G value. With no memory sharing, the throughput of Sweep and Fixed are almost identical. This is because Sweep's benefit from the reduced seek overhead is canceled out by its large cushion buffer requirement. As expected, GSS is able to achieve better throughput by balancing the seek time and cushion buffer requirements. However, Figure 3(a) shows that the gap between the best and worst schemes is not very significant: two to three 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.

6.2 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-sharing schemes, while the (b) part shows for all schemes the ratio of its memory requirement



(a) Memory Requirement (b) Ratio

Figure 4: 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 160 MB, supporting up to 47 concurrent streams. If we wish to add memory to bump our limit to 48 concurrent requests, we need to add about 100 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 $N_{Limit} = 38$ to 40. From $N_{Limit} = 38$ to the maximum achievable throughput 49, the memory requirement grows almost 20 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.

6.3 Minimize Per Stream Cost

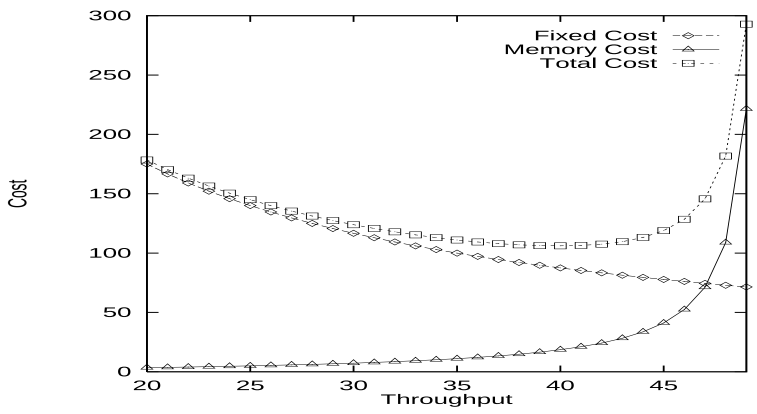


Figure 5: Per Stream Cost by Throughput

One important measure for a multimedia system storage system is the per stream cost. This is composed of the hardware cost, including CPU, buses, disks, and memory. Assuming common retail prices, a low-end computer with a two-gigabyte disk drive is about \$3,500, and the memory cost is \$20 per MByte (including other associated cost such as memory board). We refer to the non-memory cost as the fixed cost. The fixed cost is amortized by N_{Limit}: The larger N_{Limit} is, the lower the per stream fixed cost. On the other hand, the per stream memory cost grows rapidly with N_{Limit} as illustrated in Figure 4. Figure 5 plots the per stream fixed, memory, and total cost for scheme Sweep, as a function of N_{Limit}.

Since we are using the same fixed cost in all cases, we see that the per-stream fixed cost decreases as the number of streams grows. However, as the number of supported streams grows, we need to purchase additional memory, so the per-stream memory cost grows. Notice that when $N_{Limit} = 40$, the per-stream total cost is at its lowest for scheme Sweep. If we try to increase throughput beyond that, our costs will start increasing. If we continue to push performance past say 45 concurrent streams, we must pay a high premium.

Of course, the actual numbers we give here are just examples for our current scenario. If we use a different cost factor, then the values will be different. However, the shape of the curves and the overall conclusions will be similar. Although we do not show cost results for our other schemes, they display the same pattern.

In closing this section we make two important points. First, our cost analysis considered a single disk. Clearly, if we are considering how much money to spend to increase throughput, we should also consider buying more disks, as this may be a better investment than buying more memory. However, as we argue in Section 8, a multi-disk system can be analyzed as a collection of single disk systems. Thus, for each disk we purchase we need to consider how much memory to purchase to support that one disk. This means that a single disk graph like Figure 5 can still be useful in making our decision.

Second, the results of this section are for a specific hardware scenario. However, we believe that our general conclusions hold even under different disk parameters. Unfortunately, due to space limitations we cannot present here the evidence to support this claim. Nevertheless, just to illustrate, in Appendix D we briefly present some results that show the impact of expected future disk trends.

7 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.

7.1 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.

However, if the data for each request is contiguous on disk, 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. Let us first illustrate policy BubbleUp via an example. Assume that a Fixed-Stretch* storage system supports up to three requests ($N_{Limit} = 3$) with a three time unit period ($T = 3$). This means we have three one-unit slots each period, s_1, s_2, and s_3.

Table 3 shows a sample execution for three requests R_1, R_2, R_3, where R_1 arrives at time 1, R_2 arrives at time 3, and R_3 at time 5. Each row of the table represents the execution for one time unit. The columns show what requests are assigned to what slots. For example, when request R_1 arrives at time 1, it is assigned to slot s_1 and serviced right away. The asterisk in row 1 shown that R_1 is being serviced. Notice that the asterisk moves from slot to slot as time progresses. The following steps occur under policy BubbleUp:

- Time Unit 1: Slot s_1 is the current slot, and request R_1 has arrived. Service R_1 in s_1. The amount of data retrieved for R_1 is S.
- Time Unit 2: The current slot is s_2, no new request has arrived. 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. Notice that in row 2 of Table 3, R_1 now has moved to s_2 and s_1 becomes empty. Empty slots, s_3 and then s_1 are 1 and 2 time units away, ready to accept any new requests that might arrive. This is the goal of policy BubbleUp: trying to keep open slots as close time wise as possible. Incidentally, notice that we have left 2/3 of a segment unread. Next time we service R_1 we may have to read a full segment, which means reading the left over 2/3 plus a third of the next one. This is why we need to assume that all data for R_1 is contiguous, so that all reads take a single IO.
- Time Unit 3: The current slot is s_3, and request R_2 has just arrived. We service R_2 immediately in s_3.
- Time Unit 4: Slot s_1 is the current slot, no new request has arrived. 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

Time	s_1	s_2	s_3
1	* R_1		
2		* R_1	
3		R_1	* R_2
4	* R_1		R_2
5	R_1	* R_3	R_2

Table 3: BubbleUp Example

slot s_1. Again, we have freed the following slot, s_2, to be able to service some new request that might arrive.

- Time Unit 5: The current slot is s_2, and request R_3 has arrived. We service R_3 immediately. If we had not swapped R_1 out of this slot in step 4, R_3 would have had to wait for 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:

$$T_{Latency} = 2\gamma(CYL) + S/TR, \tag{16}$$

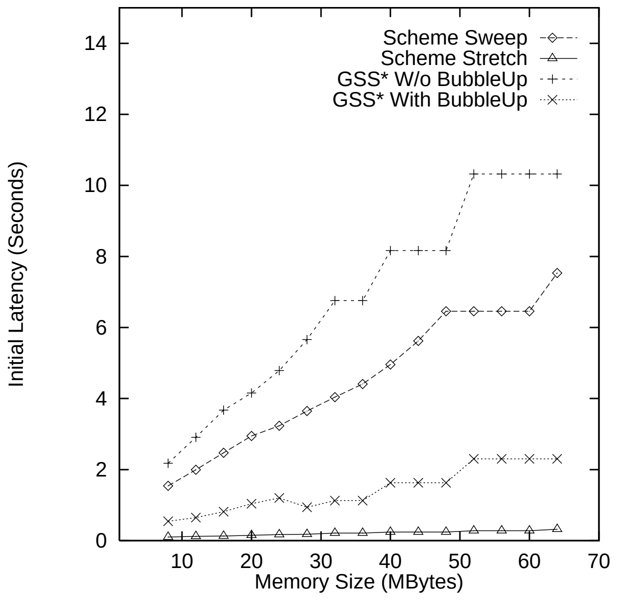
is independent of N_{Limit}. Also, notice that policy BubbleUp works only with scheme Fixed-Stretch* (and Fixed-Stretch), since only Fixed-Stretch* allows the disk arm to freely travel to any desirable cylinder without causing "jitter". Appendix C provides a formal description of policy BubbleUp.

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

$$T_{Latency} = 2T/G.$$



(a) $T_{Latency}$ vs Mem_{Avail}

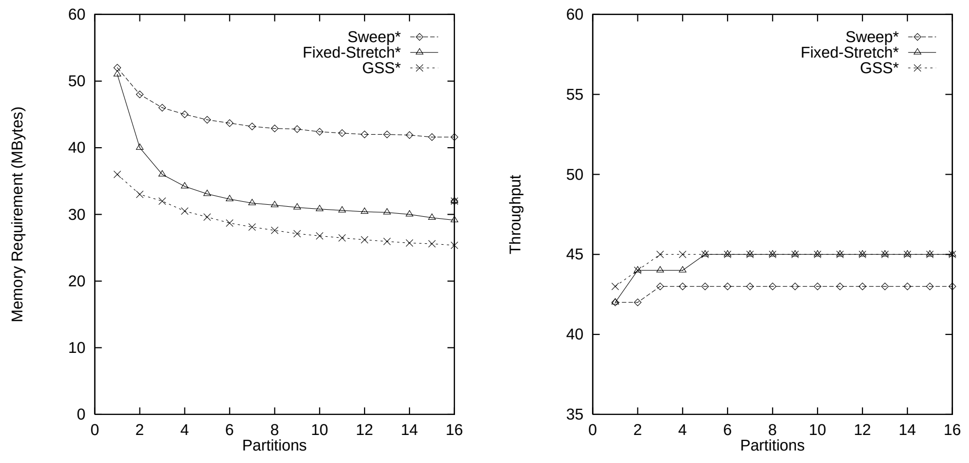
Figure 6: Startup Latency Comparison

Figure 6 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. We assume that data for each request is contiguous on disk.

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., $N_{Limit} = 40 \ to \ 45$), even though Fixed-Stretch* supports one less stream then GSS*, its initial latency is at least 1 second shorter than GSS* with BubbleUp. For interactive applications that are sensitive to delays, Fixed-Stretch* may be the choice.

8 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.



(a) Mem_{Min} vs P,$N_{Limit} = 45$ (b) N_{Limit} vs P,$Mem_{Avail} = 32MBytes$

Figure 7: Disk Partition

8.1 Disk Partitions

Reference [7] 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 6. Figure 7(a) shows the amount of memory required for up to 16 partitions at $N_{Limit} = 45$ 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 7(b) plots the throughput achievable with 32 MBytes of available memory and up to 16 partitions. In terms of throughput, using 5 or more partitions makes Fixed-Stretch* perform the same as GSS*. Again, Fixed-Stretch* benefits more from partitions than GSS* 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 [3], a disk partition scheme may not be suitable for interactive applications.

8.2 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.

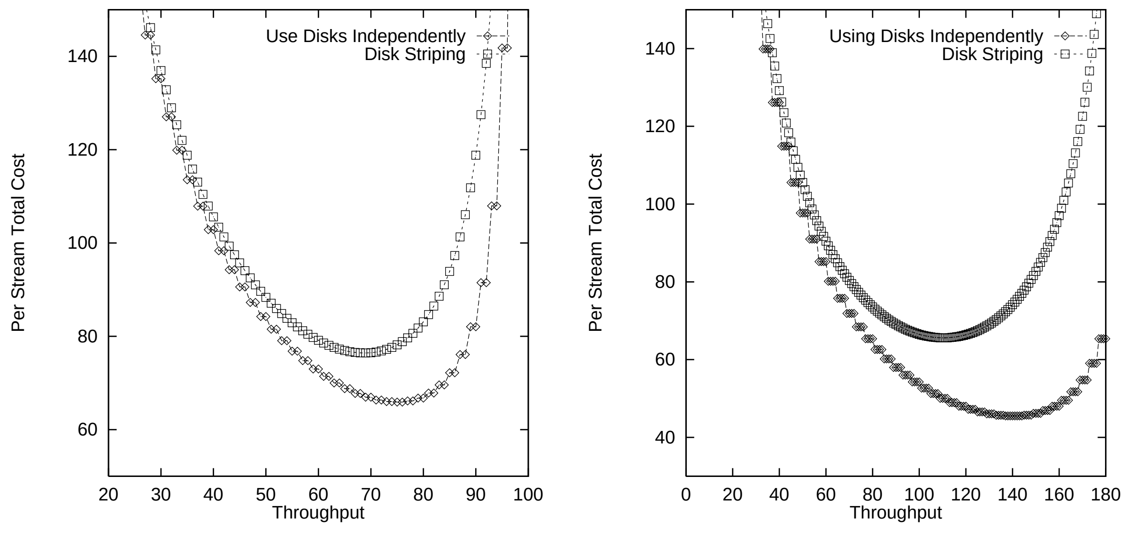
The second way to use disks, called *stripped disks* [5], treats a group of disks as one storage unit, with each segment broken into several subsegments, each stored on a separate disk. The time to transfer one segment into memory is reduced since the subsegments can be fetched in parallel. With striping, 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. (The study of [11] proposes a coarse-grained striping technique that stores data on multiple disks but operates disks independently to balance workload and conserve memory. 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 4: 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.5 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.

To illustrate the impact of multiple disks, in our next experiment we compare the per stream costs for independent and stripped disks when we have $M = 2$ and $M = 4$ disks. We only show the per stream costs for Fixed-Stretch, but all schemes display the silimar pattern. We use the same cost figures as before except we add \$500 for each additional disk. Figure 8(a) shows the case with two disks, while Figure 8(b) shows the four disk scenario. The minimum per stream cost for disk striping over two disks is 15% (76 versus 65) higher than for independent disks. The minimum per stream cost for striping over four disks is 44% (65 versus 45) higher than



(a) 2 Disks (b) 4 Disks

Figure 8: Per Stream Cost With Multiple Disks

with independent disks. This confirms the higher memory costs of disk striping as shown in Theorem 4.

For the analysis of a multi-disk system with no striping, we need to partition the available memory among the disks, and assume there is no sharing between the partitions. This is because we are analyzing for the worst case, and this occurs when the memory consumption peaks for each disk overlap exactly. This means we can decouple of our analysis: first we can evaluate how many requests a single disk can support at minimal cost (using an evaluation like the one in Section 6.3), and then we can determine how many total disks we need to support the required throughput.

9 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.

When the data for each request is contiguous on disk, policy BubbleUp 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 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 to less than what is feasible in order to make the system more cost effective.

References

[1] Seagate barracuda 4lp family product specification. *URL: http://www.seagate.com*, 1996.

[2] E. Chang and Y.-Y. Chen. Minimizing memory requirement in a multimedia storage system. *Stanford Technical Report SIDL-WP-1996-0045 URL: http://www-diglib.stanford.edu*, August 1996.

[3] E. Chang and H. Garcia-Molina. Reducing initial latency in multimedia storage systems. *To Appear in IEEE Multimedia*, 1997.

[4] T. Chua, J. Li, B. Ooi, and K.-L. Tan. Disk striping strategies for large video-on-demand servers. *Proceedings of ACM Multimedia*, pages 297–306, November 1996.

[5] H. Garcia-Molina and K. Salem. Disk striping. *International Conference on Data Engineering*, pages 336–342, Feburary 1986.

[6] S. Ghandeharizadeh, S. Kim, and C. Shahabi. On configuring a single disk continuous media server. *SIGMETRICS PERFORMANCE EVALUATION*, 23(1):37–46, May 1995.

[7] S. Ghandeharizadeh, S. Kim, and C. Shahabi. On disk scheduling and data placement for viedo servers. *USC Technical Report*, December 1995.

[8] J. Hennessy and D. Patterson. *Computer Architecture, A Quantitative Approach*. Morgan Kaufmann, 2 edition, 1995.

[9] D. Makaroff and R. Ng. Schemes for implememting buffer sharing in continuous-media systems. *Information Systems*, 20(6):445–464, 1995.

[10] B. Ozden, A. Biliris, R. Rastogi, and A. Silberschatz. A low-cost storage server for movie on demand databases. *Proc. VLDB*, September 1994.

[11] B. Ozden, R. Rastogi, and A. Silberschatz. A framework for the storage and retrieval of continuous media data. *Proc. IEEE Multimedia*, pages 2–13, May 1995.

[12] A. Reddy and J. Wyllie. I/o issues in a multimedia system. *Computer*, 2:69–74, March 1994.

[13] C. Ruemmler and J. Wilkes. An intro to disk drive modeling. *Computer*, 2:17–28, March 1994.

[14] R. Steinmetz. Multimedia file systems survey: approaches for continuous media disk scheduling. *Computer Communications*, pages 133–44, March 1995.

[15] 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.

[16] 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.

[17] P. Yu, M.-S. Chen, and D. Kandlur. Grouped sweeping scheduling for DASD-based multimedia storage managemen. *Multimedia Systems*, 1(1):99–109, January 1993.

Appendix A: Proof of Theorems

A.1: Proof of Theorem 1

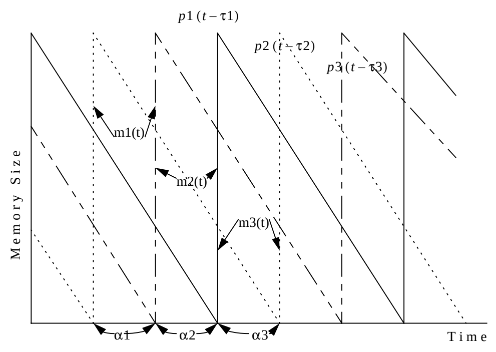


Figure 9: $p(t)$

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

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

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

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

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

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

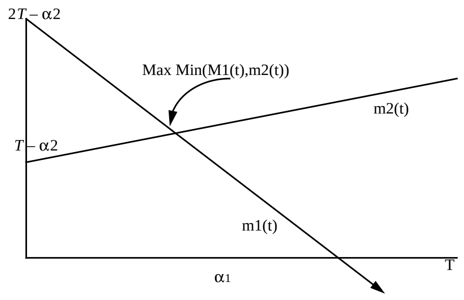


Figure 10: Maximize the Minimum Values of m'1(t) and m'2(t)

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

$$p(t) = \sum_{i=1}^{N_{Limit}} p_i(t - \frac{i \times T}{N_{Limit}}),$$
$$\text{where} \quad p_i(t - \frac{i \times T}{N_{Limit}}) = DR \times T - DR \times (t - \frac{i \times T}{N_{Limit}}). \tag{17}$$

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 17, obtaining

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

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

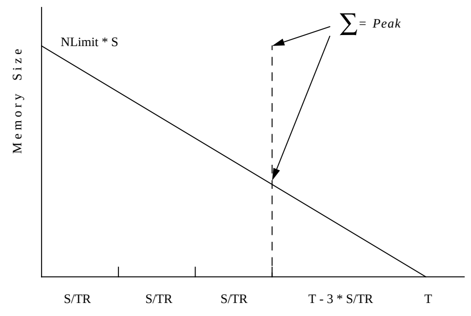


Figure 11: Memory Requirement for Sweep*

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*:

$$N_{Limit} \times S + N_{Limit} \times DR \times (T - \frac{(N_{Limit} - 1) \times S}{TR}). \qquad (18)$$

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

$$(N_{Limit} - 1) \times S + N_{Limit} \times DR \times (T - \frac{(N_{Limit} - 2) \times S}{TR}). \qquad (19)$$

$\square$

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*,

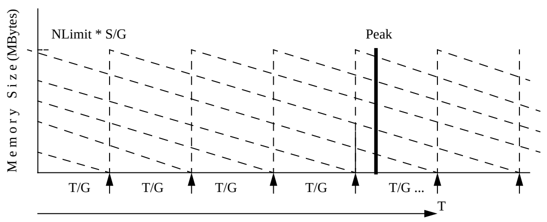


Figure 12: Memory Requirement of GSS*

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

$$(N_{Limit}/G) \times S \times \frac{G+1}{2} + N_{Limit} \times DR \times (T/G - (N_{Limit}/G - 1)\frac{S}{TR}). \quad (20)$$

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 20 by $(N_{Limit}/G - 2)$ and by subtracting S from the overall memory use. Thus, we obtain:

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

Note that because $TR \geq N_{Limit} \times DR$, the amount in Equation 21 will always be smaller than that in Equation 20.

A.5: Proof of Theorem 4

With independent disks, each disk must support N_{Limit}/M requests. In this case, the segment size is given by Equations 8, 25 or 14 (substituting N_{Limit} by N_{Limit}/M):

$$S = \frac{\frac{N_{Limit}}{M} \times \gamma(cyl_i) \times TR \times DR}{TR - (DR \times \frac{N_{Limit}}{M})}, \tag{22}$$

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 22 by M we see that $S' = S \times M$. □

Appendix B: Fixed-Stretch Derivations

B.1 Maximizing Throughput

As before, we call the maximum throughput for a given system N_{Max}. The value of N_{Max} is the value of N_{Limit} that solves our equations. To obtain it we solve for S in Equation 11. Assuming that N_{Limit} is large, we approximate Mem_{Avail} by $N_{Limit} \times (\frac{S}{2} + (T_{Rot} \times DR))$. As we have argued, $T_{Rot} \times DR$ is negligible compared with $S/2$. Thus, we simplify to $Mem_{Avail} = S \times N_{Limit}/2$, or $S = \frac{2Mem_{Avail}}{N_{Limit}}$. Substituting S and $T = S/DR$ (Eq. 1) into Equation 10, we obtain

$$N_{Limit} = \frac{2Mem_{Avail} \times TR}{(N_{Limit} \times \gamma(CYL) \times TR + 2Mem_{Avail}) \times DR}. \tag{23}$$

Dividing both sides by $TR \times DR$, and moving all terms to one side, we have the expression

$$\gamma(CYL) \times N_{Limit}^2 + \frac{2Mem_{Avail}}{TR} \times N_{Limit} - \frac{2Mem_{Avail}}{DR} = 0.$$

This second order polynomial can be solved by selecting the only positive root and rounding it down to the nearest integer. The maximum throughput is thus

$$N_{Max} = \left\lfloor \frac{\sqrt{(\frac{Mem_{Avail}}{TR})^2 + \frac{2 \times \gamma(CYL) \times Mem_{Avail}}{DR}} - \frac{Mem_{Avail}}{TR}}{\gamma(CYL)} \right\rfloor \tag{24}$$

B.2 Minimizing Memory

Instead of assuming some amount of memory Mem_{Avail}, we can also try to determine the minimum amount of memory, Mem_{Min}, required to support some desired N_{Limit}. Substituting $T = S/DR$ (Eq. 1) into Equation 10, we can solve for S, the segment size needed to support the N_{Limit} requests:

$$S = \frac{N_{Limit} \times \gamma(CYL) \times TR \times DR}{TR - (DR \times N_{Limit})}. \tag{25}$$

From equation 11 we can then solve for the required minimum memory for scheme Fixed-Stretch:

$$\begin{aligned} Mem_{Min} &= \frac{S \times (N_{Limit}+1)}{2} + (N_{Limit} \times T_{Rot} \times DR) \approx \frac{S \times (N_{Limit}+1)}{2} \\ &= \frac{(N_{Limit}+1)N_{Limit} \times \gamma(CYL) \times TR \times DR}{2(TR - (DR \times N_{Limit}))}. \end{aligned} \tag{26}$$

Again, as N_{Limit} grows, the denominator of Equation 26 approaches zero, causing Mem_{Min} to grow rapidly.

Appendix C: Policy BubbleUp

This appendix describes policy BubbleUp formally. Let s_i, for $i = 0, 2, ..., N_{Limit} - 1$, denote the IO slots. The slots are T/N_{Limit} apart (following Theorem 1). Let ψ denote the current IO slot that is being serviced by the disk. Figure 13 shows policy BubbleUp.

```
Policy BubbleUp

• Initialization:
    1. For $i$ from 0 to $N_{Limit} - 1$: $s_i \leftarrow$ empty
    2. $\psi \leftarrow 0$
• At beginning of every slot:
    1. If $s_\psi$ is occupied but playback ended:
       $s_\psi \leftarrow$ empty
    2. If $s_\psi$ is empty and a new request has arrived
         – Service the new request in $s_\psi$
         – $s_\psi \leftarrow$ occupied
       Else if $s_\psi$ is empty and no new request
         – Execute Procedure Swap (Figure 14)
       Else ($s_\psi$ is occupied)
         – Do IO for $s_\psi$
    3. $\psi \leftarrow (\psi + 1)$ mod $N_{Limit}$
```

Figure 13: Policy BubbleUp

Every T/N_{Limit} seconds, the system services an IO slot. The key in policy BubbleUp is the step where the current IO slot s_ψ is empty and there is no new requests. When this happens, policy BubbleUP swaps slot s_ψ with an occupied slot, and makes the empty slot available in the future. Procedure *Swap* in Figure 14 describes how a swap slot is chosen and the amount of data to retrieve for the swapped request.

In the second step of execution, procedure Swap determines which slot to swap. The slot to swap is the first occupied slot due up for service. If all slots are empty, then the procedure does nothing. After a swap candidate is chosen, procedure Swap calculates how much data to retrieve for the swapped request. First, procedure Swap determines how many slots (denoted as N_{Slots}) the swapped request has to be moved forward. Since the request was scheduled to run out of data N_{Slots} slots away, moving it N_{Slots} forward leaves $S \times N_{Slots}/N_{Limit}$ data in the buffer. To replenish the buffer to S to sustain a full cycle of display requires the data retrieval size to be:

$$S \times \frac{(N_{Limit} - N_{Slots})}{N_{Limit}}.$$

Procedure Swap

```
• Input:
    – Variables: $\psi$, $\kappa$, $N_{Slots}$;
    – $s_i$ for $0 = 1, 2, ..., N_{Limit} - 1$;
• Output:
    – $s_i$ for $i = 0, 2, ..., N_{Limit} - 1$;
• Execution:
    1. $\kappa \leftarrow (\psi + 1) \bmod N_{Limit}$
    2. While $\kappa \neq \psi$ do
        – If ($s_\kappa =$ occupied)
          Swap slot found ($s_\kappa$), exit loop
        – Else $\kappa \leftarrow (\kappa + 1) \bmod N_{Limit}$
    3. If $\kappa \neq \psi$ (swap slot found) then
        (a) Compute swap distance: $N_{Slots} \leftarrow (\kappa - \psi + N_{Limit}) \bmod N_{Limit}$.
        (b) The request serviced in slot $s_\kappa$ is now moved to be serviced in $s_\psi$.
        (c) Retrieve $S \times \frac{(N_{Limit} - N_{Slots})}{N_{Limit}}$ amount of data for the request.
        (d) $s_\psi \leftarrow$ occupied
        (e) $s_\kappa \leftarrow$ empty
```

Figure 14: Procedure Swap

To summarize, in each IO slot, the amount of data retrieved does not exceed S. This ensures that policy BubbleUp does not violate the minimum memory requirement. In addition, under policy BubbleUp, empty slots are always maintained right after s_ψ. When a new request arrives, it can be serviced in $s_{\psi+1}$ shortly. As stated in the main body of the paper, the worst initial latency happens when a request arrives just after the disk arm is scheduled to execute the data transfer for procedure Swap. The delay until we start transferring data for the new request is one seek and one data transfer time for executing procedure SWAP, and one more seek for the new request itself. Thus, the worst initial latency is

$$T_{Latency} = 2 \times \gamma(CYL) + S/TR.$$

10 Appendix D: Future Hardware Trends

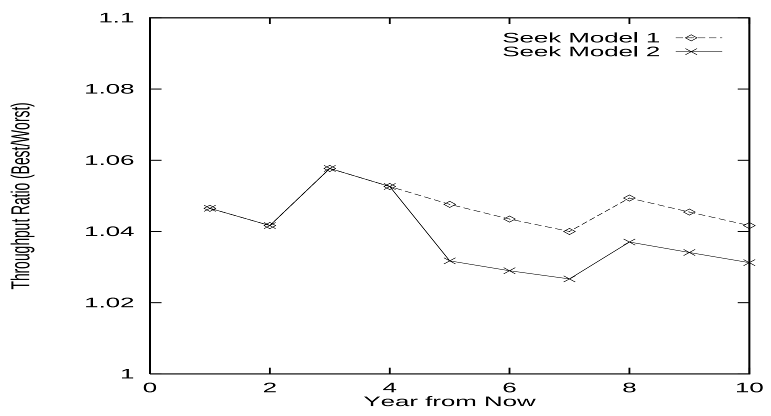


Figure 15: Throughput Ratio in Ten Years

In our last experiment we consider how changes in hardware parameters may affect throughput and the relative performance of the schemes we have studied. In particular, it is predicted [8, 13] that over the next few years memory capacity will grow at 60% annually, disk transfer rates at 40% per year, and disk rotational speeds at 12% yearly. In addition, it is expected that disk arm access times will be cut by a third over the next 10 years (equivalent to 4% yearly decrease). We also assume that DR grows at 20 to 30% yearly. This growth is to model future higher resolutions that may be used. Of course, growth will not be gradual but is likely to be in large steps. However, since we have no way of predicting when the steps will occur, we use a gradual growth model. Does this mean that all disk scheduling policies will still produce about the same relative throughput?

To illustrate, Figure 15 shows the results for two growth models. In the figure, the horizontal axis is the number of years from now, while the vertical axis shows the expected ratio of the throughput of the best and worst disk scheduling policies in the next ten years. A ratio of 1 in Figure 15 means that the schemes have the same throughput.

We study two cases:

- Model 1: We study the impact of the growth of TR, Mem_{Avail}, DR, and the reduction on the rotational delay and disk access time as forecast. In this case, we assume both the minimum seek time and maximum seek time in Table 2 are improved by 4% a year and use the same procedure described at the beginning of Section 6 to derive $\gamma(d)$.
- Model 2: The same as the previous case, except we assume no improvement for the minimum seek time. This is under the assumption that for short distance seeks, the speedup, slowdown, and settle phases dominate[13], and these factors are unlikely to be improved as the disk arm gets faster.

In Figure 15, we see that under both models, the ratio of the best and worst throughput because of the disk scheduling policies is insignificant (within less than 6%). This gives us more assurance that the analytical model of this study holds up for different disk parameters and lasts through trends. The results remain that the throughput of a media server is not very sensitive to the disk scheduling policy it employs.