Effective Memory Use in a Media Server

Edward Chang and Hector Garcia-Molina
Department of Computer Science
Stanford University

Abstract

A number of techniques have been developed for maximizing disk 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 memory use. We show that with proper memory use, maximizing disk utilization does not necessarily lead to optimal throughput. In addition, we study the impact of data placement policies including disk partitioning and multiple disks. Finally, our analysis shows that maximizing disk utilization and disk striping incur high system costs, and are not advisable in a media server.

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." This timely data retrieval requirement is also referred to as the continuous requirement or the real-time constraint [17]. 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!

In addition to studying the maximum throughput supported by media servers, we also consider the resources required and the per stream dollar cost. Our results show 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 propose to limit throughput to less than what is feasible in order to minimize per stream costs. We also briefly study the worst case initial latency before a new media request can be satisfied, which can be an important factor in an interactive system. The disk scheduling policies that allow the disk arm to move freely between requests (incurring lower bandwidth) surprisingly yield much lower initial latencies.

The rest of this paper is organized into seven 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 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 management, can lead to better performance than Sweep's. We call our modified scheme Fixed-Stretch,

Proceedings of the 23rd VLDB Conference
Athens, Greece, 1997

and we again present performance formulas that precisely account for memory sharing.

The third scheme we consider is a Group Sweeping Scheme (GSS) [18], 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 tradeoffs.

Section 7 analyzes data placement policies that are not considered in the first six sections. We 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, we offer our conclusions in Section 8.

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 [6, 8, 11, 13, 15] that optimize throughput by reducing disk seek overhead. Study [13] 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 7.) 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.)
- 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.

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}

[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.

request. Since γ is a concave function [9, 14, 16], 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:

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

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

$$T_{Transfer} = N_{Limit} \times S/TR. \quad (3)$$

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,

$$T = N_{Limit} \times (\gamma(CYL/N_{Limit}) + S/TR). \quad (4)$$

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 6.3.)

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,

$$Mem_{Avail} = 2 \times N_{Limit} \times S. \quad (5)$$

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

We can derive a closed form for the minimum memory requirement as follows. 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:

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

(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,

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

It is important to note in Equation 7 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 7 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 6.4, even if the system can support N_{Limit} concurrent streams, doing so is not cost effective.

In addition to deriving the minimum memory requirement, we can also derive the maximum system throughout for a given amount of memory (Mem_{Avail}). Please see [4] for details.

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 [10, 16]. 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 seek 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 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

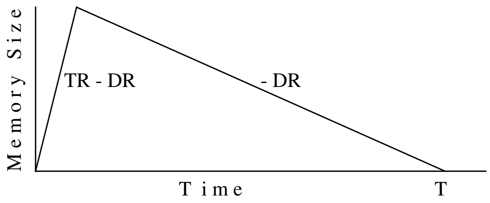


Figure 1: Memory Required in A Period

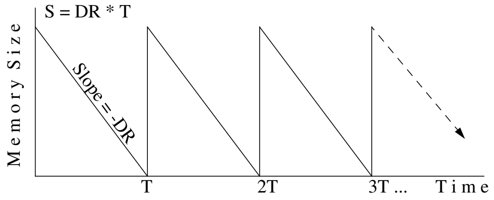


Figure 2: Memory Requirement Function

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 shows the actual memory used by a request. In practice, of course, memory is released in pages, so Figure 1 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 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 2. 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 2 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 2 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[3]. 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 [2, 4] the proofs of Theorem 1 and Corollary. 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 [4], 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 the formula for the minimum memory requirement of Sweep*.

[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 [10].

[3] For the theorem that deals with streams with different display rates please see [2].

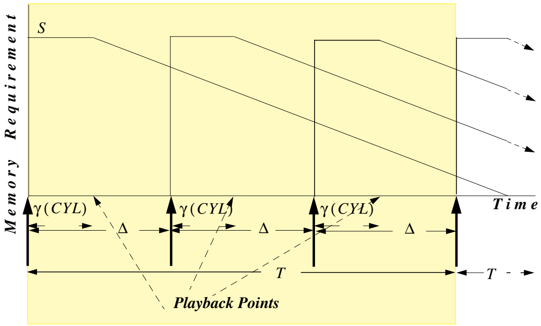
S 

Figure 3: Service Slots of Fixed-Stretch*

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 3 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 3 represents the amount of memory utilized by an individual stream.

Fixed-Stretch* executes in the following steps:

1. At the start of a service slot (indicated by the up-arrow in Figure 3), set the *end of slot timer* to expire in Δ.

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

3. Allocate S amount of memory for the request serviced in this time slot.[4]

4. Set the *IO timer* to expire in $\gamma(CYL)$, the worst possible seek overhead, and start 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 3), 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.[5] Go 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 if other requests depart or join the system.
- 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. As we discuss in Section 6.3, allowing the disk arm to move freely to service any request also leads to a very small initial latency, much better than that of the seek reduction schemes.

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 (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.

[4] 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 [10].

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

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

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

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 3 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 6, 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

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

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}. Following the same derivation steps in Section 2.2, we get the minimum memory requirement as follows:

$$Mem_{Min} = \frac{S \times (N_{Limit}+1)}{2}, \quad (10)$$

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

Please see [4] for the detailed derivations. If memory is *not* shard 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 [18] 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 [18].

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.

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:

$$S = \frac{N_{Limit}\times\gamma(CYL\times G/N_{Limit})\times TR\times DR}{TR-(DR\times N_{Limit})}. \quad (11)$$

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

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

$$+N_{Limit}\times DR\times(T/G-(N_{Limit}/G-2)S/TR). \quad (12)$$

Please refer to [4] for the proof.

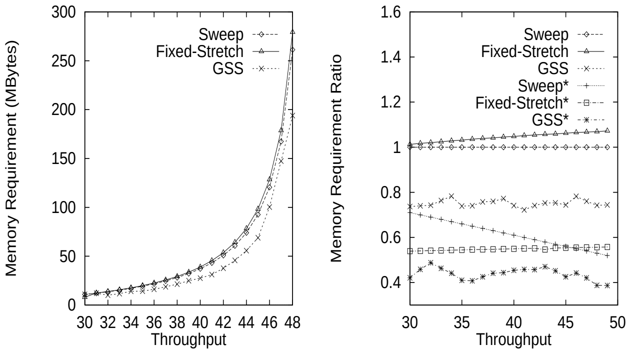
6 Evaluation

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 4. 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 [9, 14] 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

Figure 4: Seagate Barracuda 4LP Disk Parameters



(a) Memory Requirement (b) Ratio

Figure 5: Memory Requirement

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

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.

In each seek overhead we have included a full disk rotational delay 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 Memory Requirements

We set a desired throughput, and study how much memory each scheme needs to support it. Figure 5 shows the minimum amount of memory required, Mem_{Min}, to support a given number of requests (N_{Limit}). Figure 5(a) shows only the no-memory-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 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 5(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.

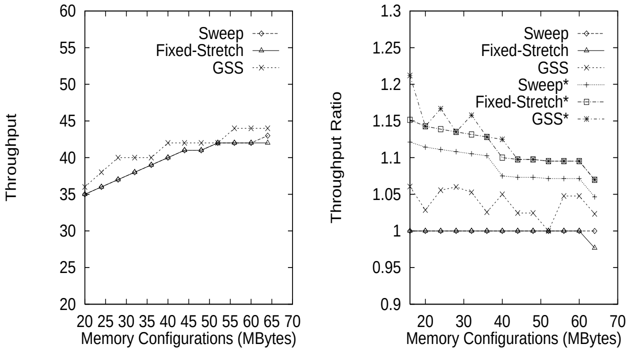
For all schemes, the marginal memory requirement starts dramatically increasing around $N_{Limit} = 38$ to 40. From $N_{Limit} = 38$ to the maximum achievable throughput 49, the memory requirement grows almost 20 fold. This suggests that although it is theoretically possible to use memory to reduce seek overhead and improve throughput, it may be economically unwise. We examine the cost issues in greater detail in Section 6.4.

6.2 Throughput

Instead of showing the minimum memory requirement, we can also show the maximum throughput the server can support given an amount of memory. The throughput is the maximum N_{Limit} the system can achieve given a memory configuration Mem_{Avail}. Figure 6 presents the throughput of our schemes for various memory sizes. 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 throughput to that of Sweep. For GSS and GSS* we again use an 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 6(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 6(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,



(a) Memory Not Shared (b) Throughput Ratio

Figure 6: Throughput Comparison

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.

6.3 Initial Latency

In this section we briefly 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. (For more discussion about why reducing initial latency is important, please consult references [3, 5].)

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.

In reference [3] we propose a resource management scheme, named *BubbleUp*, that builds upon Fixed-Stretch* to minimize the initial latency for servicing a newly arrived request. BubbleUp always makes the available disk bandwidth and memory resource ready for servicing a new request. This is achieved by using idle slots to execute "early" requests that are scheduled in the very near future. This creates some free time in the near future to handle new requests. If no new requests arrive, then those idle slots can also be used to service future scheduled requests. The worst latency to service a newly arrived is $2 \times \gamma(CYL) + S/TR$, independent of N_{Limit}. The evaluation in [3] shows that when the media server is heavily loaded ($N_{Limit} = 42$), BubbleUp that builds

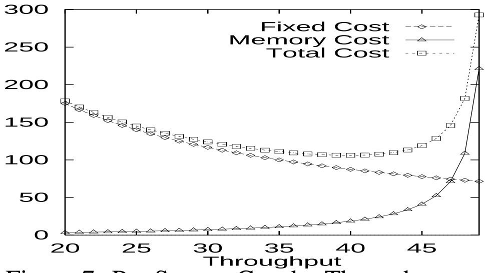


Figure 7: Per Stream Cost by Throughput

upon Fixed-Stretch* has a worst case initial latency of only a quarter of a second, while Sweep* suffers from more than eight seconds of delay. The GSS* scheme has even worse delay since its segment size is larger than that of Sweep*. Thus, for interactive applications that require fast response time, the results in Sections 6.1 and 6.2 together with the results in [3] show that a Fixed-Stretch* based scheme may be the choice.

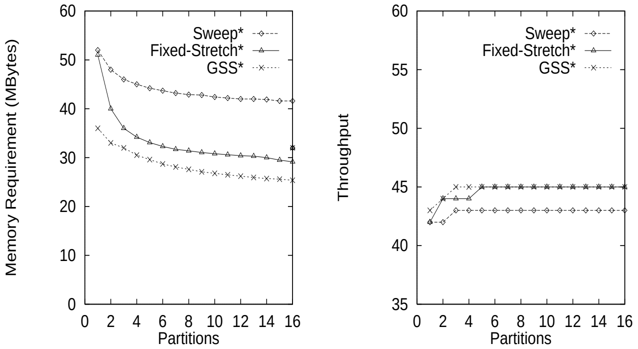
6.4 Minimize Per Stream Cost

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 5. Figure 7 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 7 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



(a) Mem_{Min} vs P (b) N_{Limit} vs P

Figure 8: Disk Partition

memory to purchase to support that one disk. This means that a single disk graph like Figure 7 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. Reference [4] presents results that support this claim, but due to space limitations they cannot be given here.

7 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.

7.1 Disk Partitions

Reference [8] 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. Dividing 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 8(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 8(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 5(a)). Note that since initial latency grows with P [5], a disk partition scheme may not be suitable for interactive applications.

7.2 Multiple Disks

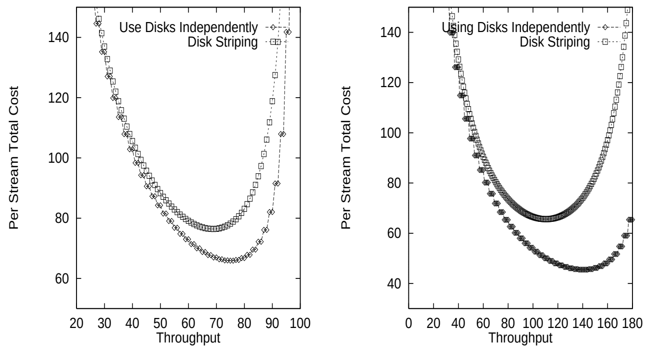
There are two common ways to allocate data when multiple disks are available in a system. With the first, which we call *independent disks*, a segment of a presentation is always stored within a single disk. (Although different segments of a presentation can be stored on multiple disks for the purpose 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 *striped disks* [7], 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 striped disks. For example, if we have a display rate that cannot be supported by a single disk, then striping 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 [12] proposes a coarse-grained striping technique that stores data on multiple disks but operates disks independently to balance workload and conserve memory.) 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 striping 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, striping requires M times as much memory as independent disks do. Please refer to [4] 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,



(a) 2 Disks (b) 4 Disks

Figure 9: Per Stream Cost With Multiple Disks

striping is not desirable.

To illustrate the impact of multiple disks, in our next experiment we compare the per stream costs for independent and striped 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 similar pattern. We use the same cost figures as before except we add $\$500$ for each additional disk. Figure 9(a) shows the case with two disks, while Figure 9(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 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 6.4), and then we can determine how many total disks we need to support the required throughput.

8 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 briefly pointed out that allowing the disk arm to move freely to service any request can reduce initial latency drastically, an important performance requirement of interactive applications. 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 requirements in a multimedia storage system. *Stanford Technical Report SIDL-WP-1996-0045 URL: http://www-diglib.stanford.edu*.

[3] E. Chang and H. Garcia-Molina. Bubbleup - low latency fast-scan for media servers. *Stanford Technical Report SIDL-WP-1997-0064 URL: http://www-diglib.stanford.edu*.

[4] E. Chang and H. Garcia-Molina. Effective memory use in a media server (extended version). *Stanford Technical Report SIDL-WP-1996-0050 URL: http://www-diglib.stanford.edu*.

[5] E. Chang and H. Garcia-Molina. Reducing initial latency in multimedia storage systems. *IEEE Multimedia, Fall 97*.

[6] 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.

[7] H. Garcia-Molina and K. Salem. Disk striping. *ICDE*, pages 336–342, Feburary 1986.

[8] S. Ghandeharizadeh, S. Kim, and C. Shahabi. On configuring a single disk continuous media server. *Sigmetrics Performance Evaluation*, 23(1):37–46, May 1995.

[9] 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.

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

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

[12] 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.

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

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

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

[16] 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.

[17] 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.

[18] 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.