Scheduling for Interactive Operations in Parallel Video Servers

Min-You Wu
Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260
Phone: (716)645-3180
Fax: (716)645-3464
wu@cs.buffalo.edu

Scheduling for Interactive Operations in Parallel Video Servers

Min-You Wu
Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260
wu@cs.buffalo.edu

Abstract— Providing efficient support for interactive operations such as fast-forward and fast-backward is essential in video-on-demand and other multimedia server systems. In this paper, we present two basic approaches for scheduling interactive operations, the prefetching approach and the grouping approach. Scheduling algorithms are presented for both fine-grain and coarse-grain data blocks. These algorithms can precisely schedule video streams for both normal playout and interactive operations.

keywords— parallel video servers, real-time scheduling, interactive operations, fast forward, Quality-of-Service.

1. Introduction

There is an increasing demand on capacity of video servers in large-scale video-on-demand systems [8, 1]. Interactive scan operations, such as fast-forward and fast-backward [1] are desirable features in video-on-demand and other multimedia servers. It is necessary to provide efficient supports for interactive operations with Quality-of-Service (QoS) guarantee. There can be a simple solution, that is, a separate partition in the disk can be reserved for interactive operations. Whenever the playout mode switches from normal playout to interactive operations, this section will be used. This solution requires extra disk space for the interactive operations. Moreover, the ratio of fast-forward (fast-backward) is fixed. A better solution is to retrieve the fast-forward (fast-backward) signal from the normal video stream with some scheduling technology. However, supporting these interactive operations is more difficult than supporting simple playout because an interactive operation usually consumes higher bandwidth. Moreover, it requires a different pattern and is more difficult to schedule.

Two possible admission policies can be employed. The first one guarantees normal playout and interactive operations separately. When a normal playout changes to an interactive operation, it is treated as a new request. It is called *two-step* policy. In this policy, an interactive operation could be rejected even if the corresponding normal playout has been accepted. The second one is a *one-step* policy. It admits normal

[1] Fast-backward is often called *rewind*.

playout and interactive operations at once. That is, when a request is accepted, it can change from playout to an interactive operation, or vice versa without re-admission. Because an interactive operations usually requires higher bandwidth, more bandwidth needs to be allocated when admitting playout. Alternatively, the same bandwidth of normal playout could be allocated for a request and an interactive operation is performed with lower quality, that is, lower resolution or less number of frames displayed per second. The two-step policy can be more efficient because it admits requests of different bandwidths separately. However, some problems may occur in the two-step policy. That is, what are we going to do when an interactive operation is rejected? We may display a "rejection" signal and continue the normal playout or display a blank frame. Other the other hand, the one-step policy could suffer some inefficiency. However, with proper scheduling, the efficiency degradation can be minimized. In this paper, we assume the *one-step* policy.

We will discuss different approaches to schedule interactive operations. Two basic approaches are the *prefetching* approach and the *grouping* approach. In the prefetching approach, a buffer is required to store some video data blocks and an interactive operation can be performed with some delay. In the grouping approach, different operations, such as normal playout, fast-forward, and fast-backward, are clustered into different groups with different paces. We apply the two approaches to fine-grain and coarse-grain blocks. In the next section, the video server architectures are described. The data layout and scheduling are discussed in section 3. The prefetching approach and grouping approach are presented in sections 4 and 5, respectively. Section 6 reviews previous work and section 7 concludes the paper.

2. Video Server Architectures

There are two major types of parallel video servers: shared memory multiprocessors and distributed memory clustered architectures. In a multiprocessor system, there are a set of storage nodes, a set of computing nodes, and a shared memory. The video data is sent to the memory buffer through a high-speed network or bus, and then to the clients. A mass storage system has presented the capacity of supporting hundreds of requests [9]. However, it is not yet clear that a multiprocessor video server can be scalable. A clustered architecture is easy to scale to thousands of server nodes. In such a system, a set of storage nodes and a set of delivery nodes are connected by an interconnection network. Data is retrieved from the storage nodes and sent to the delivery nodes which send the data to clients. A number of works describe clustered systems [13, 10, 11]. The clustered architecture can be extended to the direct-access architecture which provides an interface between the storage system and the network. Project MARS uses an ATM-based interconnect to connect storage devices to an ATM-based broadband network [3].

In this paper, we assume a clustered architecture. Figure 1 shows a diagram of clustered video servers. The storage nodes, each with a local disk array, are responsible for storing video data in some storage medium, such as disks. Each storage node deals with its disk scheduling problem separately to provide enough bandwidth. The delivery nodes are responsible for client's requests. On receiving a request from a client, a delivery node will schedule it to a time interval and deliver appropriate data blocks within some time deadline to the client during the playout of a video. The logical storage and delivery nodes can be mapped to different physical nodes of the cluster. This configuration is called the "two-tier" architecture in [13]. Also, a node can be both a storage node and a delivery node, called the "flat" architecture.

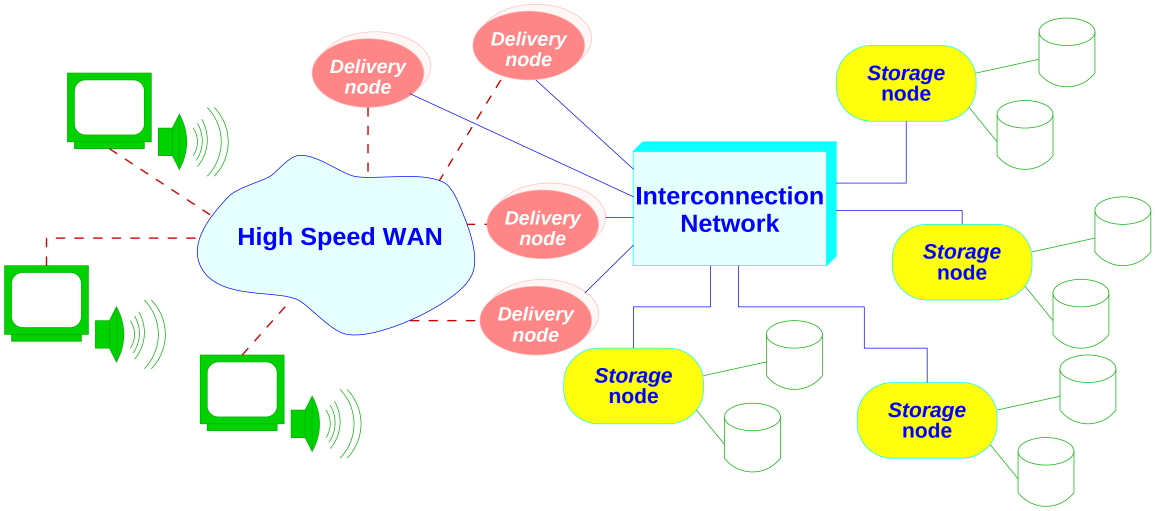

Figure 1. Diagram of a Clustered Video Server.

3. Data Layout and Scheduling

We assume that a number of video files are stored in a parallel storage server of N storage nodes. A video file is a sequence of ordered video frames. To facilitate distribution of video data, these video frames are partitioned into video blocks, where every video block contains one or more video frames.

3.1. Data partitioning

There are various configurations possible to partition video frames into video blocks. On one extreme, every video block contains only one video frame, which is called a *fine-grain* video block. The size of a video frame can vary. For example, an uncompressed video frame for HDTV is about 700 Kbytes, while the average frame size can be as small as 6 Kbytes in a compressed NTSC. Such a fine-grain partition presents the largest flexibility of video data distribution. However, it may suffer large overhead depending on disk seek and rotation penalty, as well as disk scheduling. This scheme has been used in project MARS [2].

In case of MPEG compression [7], the video frames are varied in size and possess some dependencies. Usually, several frames can be grouped together so that the frames that depend on each other are confined into the same group as a *group of picture*. It turns out that such a group can define a natural boundary for partitioning. Therefore, a video block can consist of a group of video frames.

A *coarse-grain* video block contains many video frames, which are either independent frames or frames belonging to several groups. In the case of coarse-grain video blocks, the size of video blocks can be determined for optimal I/O access time and without loss of parallelism. The fine- or coarse-grain video blocks need different treatments to serve the general interactive operations.

3.2. Distribution of video blocks

Video blocks are distributed to the storage nodes in order to 1) explore parallelism and hence increase the bandwidth provided for video stream retrieval; 2) maintain load balance over the storage nodes; and 3) reduce the capacity requirement of each single storage node and hence construct a scalable massively parallel video-on-demand storage system. In order to support a high bandwidth video-on-demand storage system and be able to serve thousands of clients simultaneously, we can evenly distribute video blocks over N storage nodes in a round-robin fashion, so called *wide striping*. When video blocks are distributed to a subset of nodes, it is called *short striping*. In this paper, we assume wide striping. The results obtained here can be easily extended to short striping.

3.3. Time-slot partition

Accessing a video block is accomplished by a process with a timer. When the access is done, the process will be put into sleep until the time when the next video block of the same video stream needs to be accessed. The scheduling module must be integrated with real time capability to handle these processes with soft-deadline constraints. When more than one process accesses the same storage node at the same time, some processes must wait. For this reason, a conventional storage server is typically lightly loaded in order to maintain a certain level of performance guarantee. Such a system usually leaves sufficient room for unexpected bursts, resulting in an efficient use of resources.

When a storage node is dedicated to the retrieval of video streams, the access time and access duration of video blocks can be known in prior. To take advantages of this knowledge, retrieval of video streams can be prescheduled. If the available bandwidth of a storage node can retrieve x video blocks in a time cycle t_c, the storage node can be time-multiplexed by x, which is equivalent to partitioning a time cycle into x time slots. It is assumed that all the video streams are partitioned into video blocks of the same size, and retrieved at the same speed, in which case we call it *homogeneous* time-slot partition.

If the stored video blocks are of different sizes, for examples, video files are compressed by different schemes, or a fast-forward operation accesses only partial video blocks, the time cycle can be partitioned into unequal-size slots, which is named as *heterogeneous* time-slot partition. Furthermore, if we need to support different retrieval rates, or other general purpose tasks, such as addition of a new video file or deletion of an old video file, the partition can be done hierarchically. It is more like the multiple-queue scheduling used in operating system, where each queue is associated with a certain percentage bandwidth of a storage node, within a queue the time can be partitioned into either *homogeneous* or *heterogeneous* slots, and one of queues can be dedicated to serve the general-purpose tasks in a round-robin fashion.

With explicit partition of the available bandwidth and preschedule of video streams onto specific time slots, we construct a more deterministic scheme that can utilize the full capacity of storage nodes, provide a better quality of service, reduce the buffer size, and save the real-time scheduling cost. Such a system is economically more attractive in a massively parallel video-on-demand system.

In this paper, we only study the homogeneous time-slot partition. The starting blocks of video files are uniformly distributed over all storage nodes. Depending on a selected block size s and the base stream rate R, time is divided into *time cycles*, where the length of a cycle is $t_f = s/R$. The optimal block size is around 256 KBytes to 512 KBytes [11, 13, 9], and an MPEG-2 compressed video rate is about 0.5 MBytes/sec. In general, the data transfer rate of a single disk or a disk array can be much higher than base stream rate R. Therefore, in a time cycle, multiple requests can be serviced by a storage node while the individual stream rate is still preserved. The time cycle is thus divided into *time slots*, where the length of the slot, t_s, is equal to or longer than the time required for retrieving a block from the storage node or transmitting to the delivery node, whichever is larger. The number of slots in a cycle, m, is determined by $m = \lfloor t_f/t_s \rfloor$. Then t_s is adjusted to t_f/m.

3.4. Scheduling for normal playout

Assume that a large-scale video-on-demand server consists of N storage nodes and N delivery nodes, interconnected by a high-speed network. An individual request r is handled by a delivery node $i = D(r)$, which is responsible for delivering the data blocks retrieved from storage nodes to the client via network during the entire life-time of request r unless a dynamic load balancing is required. Since the blocks of a video is consecutively distributed in all N storage nodes, if request r, at time cycle t, retrieves a data block from storage node $j = S(r, t)$, it will retrieve a data block from node $(S(r, t + \tau) \ mod \ N)$ at time cycle $(t + \tau)$. In each time cycle, at most $(N \times m)$ requests can be scheduled.

Video block scheduling can be illustrated by a simple example. Figure 2 shows a schedule, where $N = 4$ and $m = 3$. For a balanced load, each video stream can start from different storage nodes. An entry in the figure shows the request number r_k, the delivery node number i, and the storage node number j. That is,

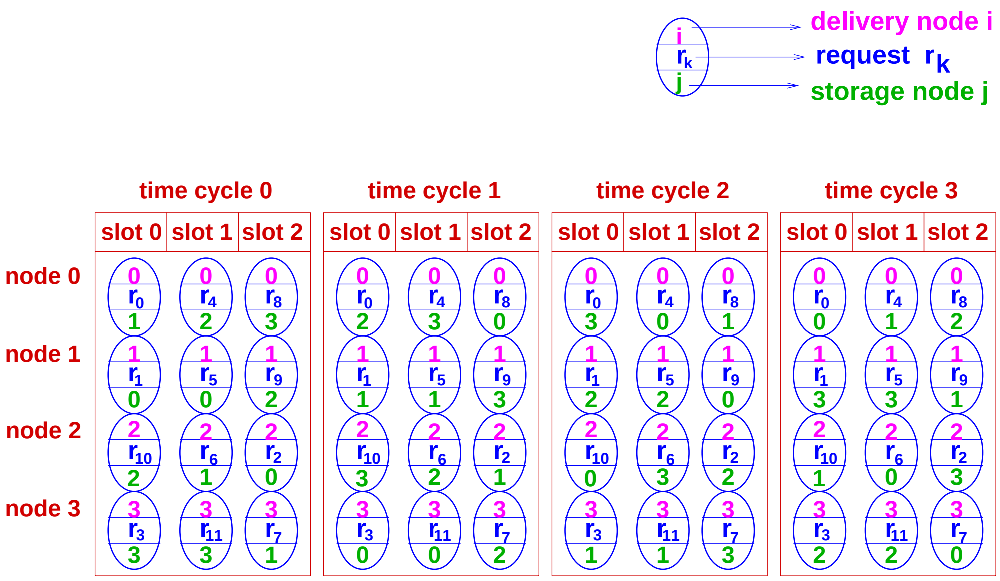

Figure 2. Scheduling of Video Blocks for Normal Playout.

request r_k retrieves a block from storage node j to delivery node i. A video stream has its access pattern listed horizontally in a row. The blocks of a single stream are separated by m time slots. For example, request r_0 is scheduled to time slot 0 in the first row. For this request, delivery node 0 retrieves a block from storage node 1 in time cycle 0. It then retrieves blocks from storage nodes 2, 3, and 0 in next three cycles. The schedule table is wrapped around. In a time slot, if more than one request needs to retrieve blocks from the same storage node, they compete for the resource. In order to avoid such a conflict, only the requests that access different storage nodes can be scheduled onto the same time slot. Thus, in every time slot, at most N requests can be scheduled, each of that retrieves a block from a different storage node. The *conflict-free schedule* is a schedule where in each time slot, no two video streams request a block from the same storage node.

The algorithms for scheduling normal playout have been presented in [14]. In this paper, we will propose scheduling algorithms for interactive operations. Since fast-forward (*ff*) and fast-backward (*fb*) impose similar requirements, the techniques developed for *ff* can easily be extended to *fb*. Thus, we will only discuss the *ff* operation. To present scheduling algorithms for the *ff* operations succinctly, we first define the *ff* ratio. That is, the *ff ratio* is f if the *ff* speed is f times the normal playout speed.

4. The Prefetching Approach

When all clients require normal playout, their access patterns are the same. As long as we have a conflict-free schedule at the first time cycle there will be no conflict at the following time cycles. If at a time, one request moves from a normal playout to a *ff* operation, the access pattern changes. The changed access pattern may cause conflict in some time cycles.

One solution is *prefetching*. That is, instead of accessing the storage node that contains the next demanded block but causes a conflict, we access a different storage node that does not cause a conflict. Retrieved from the storage node is the block that will be used in the nearest future. Since the retrieved block is not delivered immediately, it is buffered in the delivery node. This process will continue until the demanded block is retrieved. Thus, the request will be delayed for a few time cycles in general.

4.1. Inter-block skipping

The fine-grain video block has a single video frame. The small granularity may lead to high bookkeeping overhead and storage access cost. Though this assumption might not be realistic, this study provides a basic strategy for scheduling interactive operations. Furthermore, this method can be extended to segments which includes a number of continuous frames. An experiment described in [4] found that the client got the impression of watching each scene (segment) at regular speed with jumps between scenes, similar to watching a slide projector operating at a high speed. Clients can see the details in each scene so that they are able to locate the position of interest.

With this assumption, when performing a *ff* operation of ratio f, a frame (segment) is retrieved after skipping $f-1$ frames (segments). It is called *inter-block skipping*. To avoid hot spot, the number of storage nodes and *ff* ratio must be relatively prime. A good choice is to select the number of storage nodes N to be a prime. Thus, the *ff* ratio can be anything except the multiples of N. An example is shown in Figure 3, where N is 5 and the *ff* ratio is 3.

The prefetching approach for inter-block skipping can be described as follows. Assume the first video block is stored in storage node n_0, storage node n_i contains all video block with index

$$i + N \times x, \quad x = 0, 1, 2, ...$$

Assume that the *ff* ratio is f, where f is larger than one and relatively prime to N. Without loss of generality, we assume that a normal playout request changes to the *ff* operation starting from storage node n_0 at time cycle t_0. Thus, only the blocks with index $((i + N \times x) \bmod f = 0)$ will be retrieved. The stream of these video blocks to be retrieved from node n_i is

$$z_i, \quad z_i + N \times f, \quad z_i + 2N \times f, ...,$$

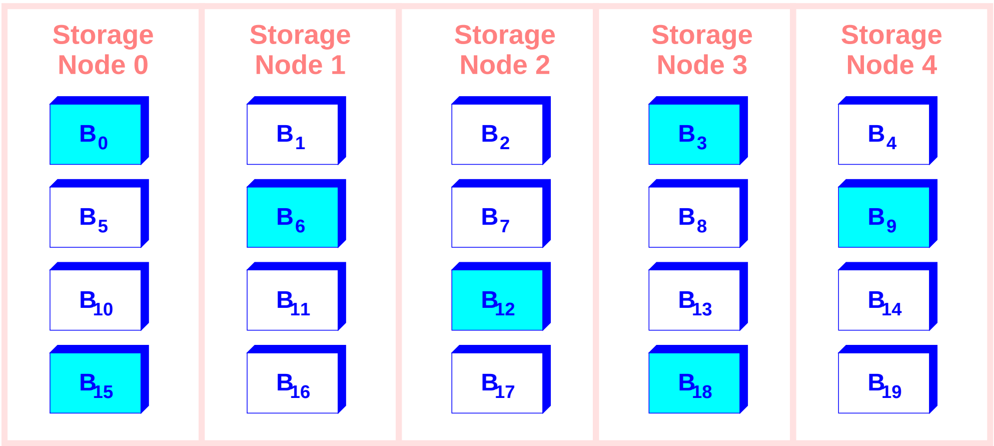

Figure 3. Layout and Access Pattern of Fast Forward for Inter-block Skipping.

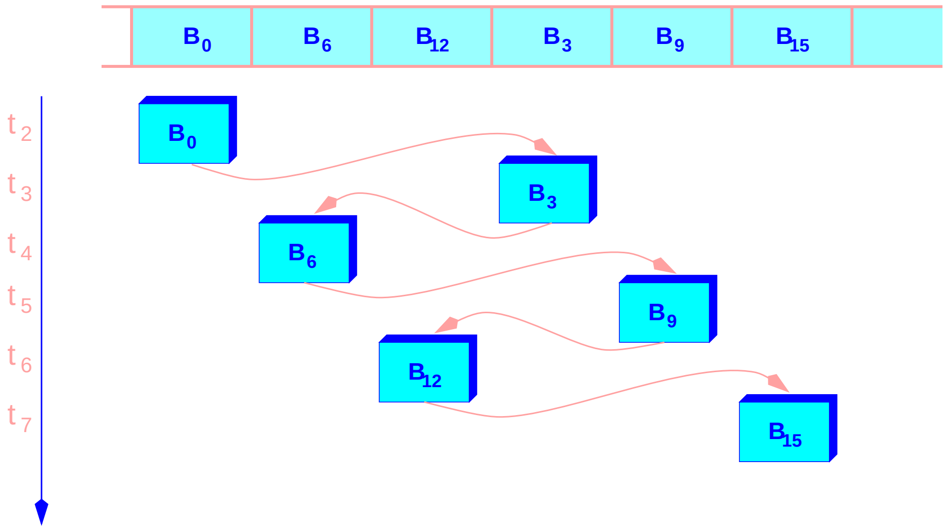

Figure 4. Prefetching Approach for Inter-block Skipping.

where

$$z_i = i + N \times \min_x \{x|(i + N \times x) \bmod f = 0,\ x = 0, 1, 2, ...\}$$

Thus, the delivery node retrieves blocks in the sequence of

$$z_0,\ z_1,\ z_2, ...$$

and delivers blocks in the sequence of

$$0,\ k,\ 2k, ...$$

The retrieved blocks will be stored in a buffer before they are delivered. Figure 4 shows the prefetching approach for the example in Figure 3. The delivery node retrieves blocks in the sequence of

$$0, 6, 12, 3, 9, 15, ...$$

and it delivers blocks in the sequence of

$$0, 3, 6, 9, 12, 15, ...$$

Assume block 0 is retrieved at time t_0, and block 3 at time t_3. The block 0 needs to be delayed to time t_2, so that at time t_3, block 3 can be delivered. Thus, in this example, the delay is two time cycles.

Let's consider the maximum delay. For a *ff* operation of ratio f, at time cycle t_k, the video block kf is supposed to be delivered, which is available from storage node n_{u_k} at time cycle t_{v_k}, where

$$u_k = (kf) \bmod N, \quad v_k = u_k + N \times \lfloor\frac{k}{N}\rfloor = ((kf) \bmod N) + N \times \lfloor\frac{k}{N}\rfloor$$

In general, t_{v_k}, the time cycle when the kth video block is available, is not equal to t_k, the time cycle when the kth video block is needed. Their difference is represented by d_k:

$$d_k = v_k - k = ((kf) \bmod N) + N \times \lfloor\frac{k}{N}\rfloor - k = ((kf) \bmod N) - (k \bmod N)$$

When $d_k < 0$, which implies that the video block is available before it is needed, the block will be prefetched and stored in the buffer. When $d_k > 0$, which implies that the video block is needed but not available, the video stream must be delayed. In general, d_k varies in period of N, because for any $k = xN + y$, where $0 \leq y < N$ and $x = 0, 1, 2, ...$,

$$d_k = (((xN + y)f) \bmod N) - ((xN + y) \bmod N) = ((yf) \bmod N) - (y \bmod N) = d_y$$

For given f and N, the maximum delay δ can be computed by

$$\delta = \max_{0 \leq k < N} d_k = \max_{0 \leq k < N} \{((kf) \bmod N) - k\}$$

Thus, the video stream is to be delayed by δ. Figure 5 shows the maximum delays when the *ff* ratio f changes from 1 to 12 with $N = 13$. Because δ varies in period of N, all the *ff* ratios with the same value of $(f \bmod N)$ have the same value of δ.

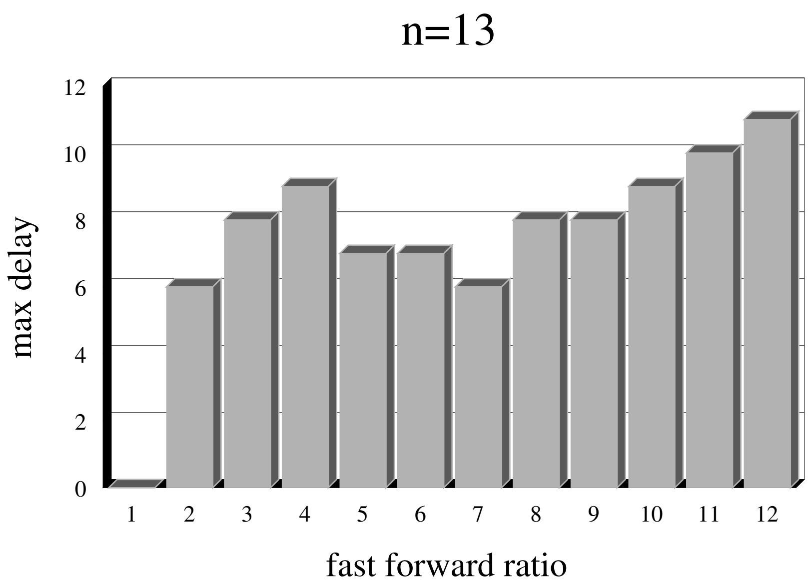

Figure 5. Example of Maximum Delay for Different Ratio f

When a video stream is delayed by δ time cycles, δ data blocks are to be stored in the delivery node's buffer. Therefore, the required buffer size is proportional to the maximum delay. Minimizing the maximum delay will minimize the buffer size too. There are a number of strategies to reduce the maximum delay and the buffer size. First, when $(f \mod N) = 1$, $\delta = \max_{0 \leq k < N} \{((kf) \bmod N) - k\} = 0$. That is, no delay is imposed. A simple design for *ff* operations can be based on this observation: if we distribute video blocks to N' storage nodes, then we can support all *ff* ratios such that $f \mod N = 1$ without having any delay. Also, no buffer is required. However, this design limits the available parallelism and the choice of the *ff* ratios. We may also vary the combination of f and N to minimize the maximum delay and the buffer size. Since δ is bounded by $N - 1$, the video stream can be delayed $(N - 1)$ time cycles and a buffer of size $(N - 1) \times B_s$ is needed for each request, where B_s is the block size. A delivery node handles m requests, where m is the number of time slots in a time cycle. When all requests are in the *ff* mode, the total buffer size can be as large as $m \times (N - 1) \times B_s$. However, multiple *ff* operations can share buffer space so that the buffer size can be further reduced.

4.2. Intra-block skipping

For coarse-grain blocks, instead of skipping blocks, $1/f$ frames in a block will be retrieved when the *ff* ratio is f. It is called *intra-block skipping*. This is illustrated in Figure 6, where $f = 3$.

A problem may occur with a compressed video stream, such as MPEG. That is, the size of $1/f$ frames

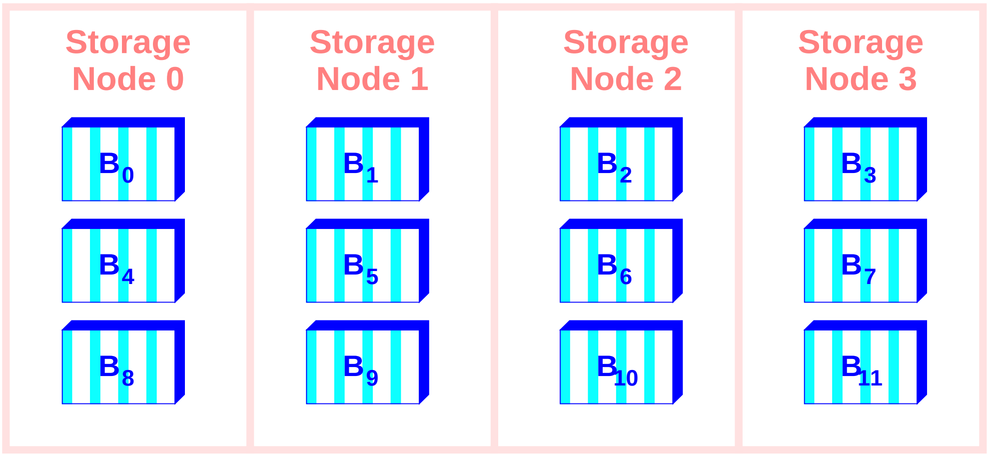

Figure 6. Access Pattern of Fast Forward for Intra-block Skipping.

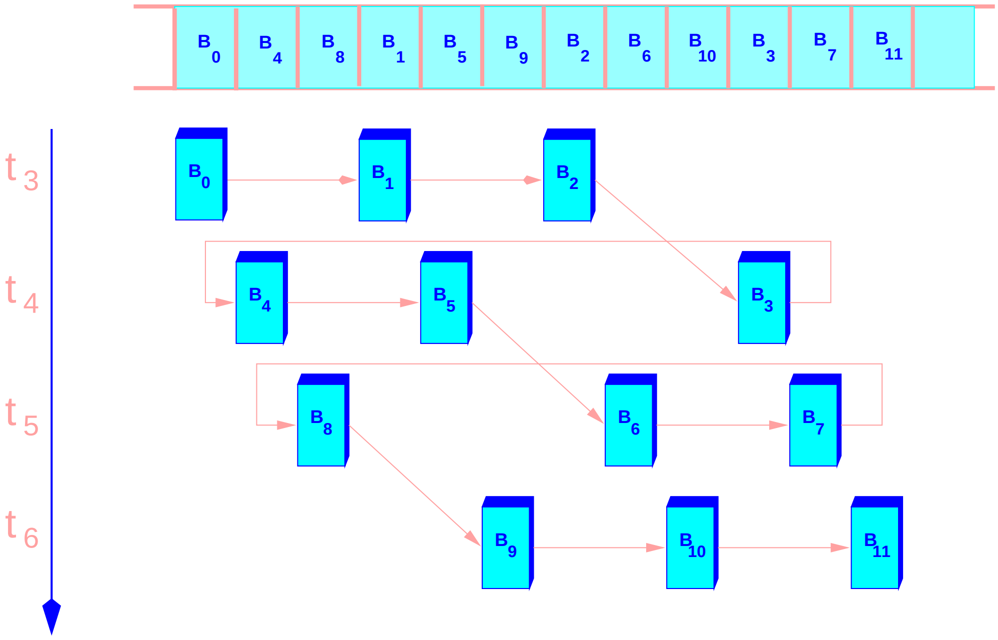

Figure 7. Prefetching Approach for Intra-block Skipping.

may be larger than $1/f$ of the size of all frames. Moreover, repositioning the disk head after every frame retrieval for large skipping distances and small block sizes will suffer severe seek and rotation penalty. Therefore, either higher bandwidth must be allocated or the quality of interactive operations must be reduced. Here, we assume that the low-resolution component will be retrieved to keep the bandwidth invariant [12]. Thus, the size of $1/f$ frames can be the same of $1/f$ of the size of all frames.

The prefetching approach can be applied to intra-block skipping. In each time slot, f blocks are accessed and f small blocks are retrieved, where the size of the small block is $1/f$ of the size of the entire block. These small blocks are buffered in the delivery nodes. When first f blocks are in the buffer, the delivery node sends them to the client. In this way, the number of time cycles delayed is $N - \lceil \frac{N}{f} \rceil$. The required buffer size for one *ff* operation is $(N - \lceil \frac{N}{f} \rceil) \times B_s$, and for m *ff* operations is $m \times (N - \lceil \frac{N}{f} \rceil) \times B_s$, where B_s is the block size. Again, multiple *ff* operations can share buffer space so that the buffer size can be further reduced.

An example is shown in Figure 7. Here, we have four storage nodes and the *ff* ratio $f = 3$. In time cycle t_0, blocks 0, 4, 8 are retrieved; and in time cycle t_1, blocks 1, 5, 9 are retrieved. In time cycle t_2, blocks 2, 6, 10 are retrieved and first three blocks can be sent out. The time delay is two time cycles.

5. The Grouping Approach

An important characteristics of video playout is its pace. The *pace* is defined as the speed of playout. When pace $p = 1$, it is in the normal playout mode. Other operations can be defined by the pace as follows:

$-\infty < p < -1$	fast-backward
$p = -1$	play-backward
$-1 < p < 0$	slow-backward
$p = 0$	pause
$0 < p < 1$	slow-forward
$p = 1$	play-forward (normal playout)
$1 < p < \infty$	fast-forward

For fast-forward, p is equal to the *ff* ratio f.

Different playout modes have different paces which may lead to access conflict on storage nodes. When requests for normal playout are scheduled into a time cycle without conflict, it is conflict-free in the following time cycles because their paces are the same. If some requests change from normal playout to the *ff* operation which has different pace, conflicts may occur. Different from the prefetching approach which changes the retrieval order to avoid conflicts, the grouping approach divides requests into different groups based on their paces. For example, we may have all normal playout operations in one group $\mathcal{G}_1$ with $p = 1$

and all *ff* operations of the same *ff* ratio f in another group $\mathcal{G}_f$. Furthermore, *ff* operations with different *ff* ratios are in different groups. In this way, all requests in the same group will not conflict each other as long as they do not conflict in the first time cycle.

5.1. Inter-block skipping

For inter-block skipping, the *ff* operation is carried out by skipping some blocks. Requests that skip different number of blocks are in different groups in addition to the normal playout group $\mathcal{G}_1$. With interactive operations supported, the system must allow a request dynamically change its pace p, and change its group membership consequently. Therefore, the key of supporting interactive operations turns to be how to handle dynamic reconfiguration of groups.

Assume that there are N storage nodes in the system and there are m time slots in each time cycle. The system can handle $N \times m$ requests when there exists only a single group. With multiple groups existing in the system, not every time slots can be fully utilized since the number of requests in a group is not necessarily a multiple of N. In general, all requests of the same pace p are in group $\mathcal{G}_p$, and the number of requests in group $\mathcal{G}_p$ is $g_p = \mid \mathcal{G}_p \mid$. The number of time slots required for the requests in group $\mathcal{G}_p$ is

$$s_p = \lceil \frac{g_p}{N} \rceil$$

In the worst case ($g_k \bmod N) = 1$, the largest fragment of a time slot is wasted. Figure 8 shows a worst case with $N = 3$, $m = 5$, and $K = 3$. Here, there are three groups. The first and second groups have four requests each and the third group has one request.

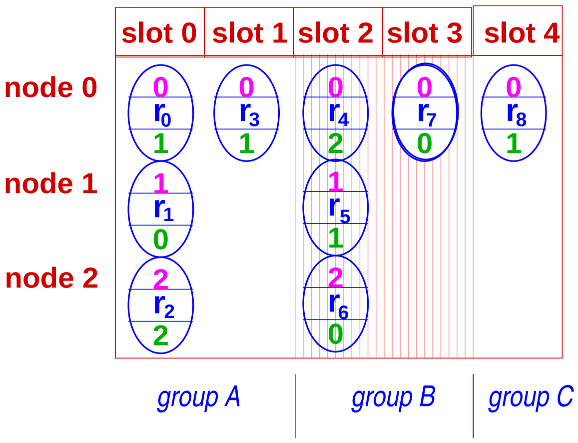

Figure 8. Groups of Requests with Different Paces.

Since a request can dynamically change its group membership, s_p, the number of time slots allocated to group $\mathcal{G}_p$ needs to be varied too. We define two operations for changing the membership of a request, *adding* and *removing*, and two operations to change the number of time slots allocated to a group, *expanding* and *shrinking*. First, the expanding and shrinking operations are described as follows.

Expanding.

If a request is to be added to group $\mathcal{G}_k$, which currently has g_k members and $(g_k \bmod N) = 0$, then one more time slot needs to be allocated to group $\mathcal{G}_k$. □

Shrinking.

When a group $\mathcal{G}_k$ currently has g_k members such that $(g_k \bmod N) = 1$, if a request is to be removed from the group, one time slot can be deallocated from group $\mathcal{G}_k$ by re-scheduling the rest of $(g_k - 1)$ requests into $(s_k - 1)$ time slots. Scheduling algorithms can be found in [14]. □

Assume there are K groups in the system. The following lemma gives the sufficient condition for expanding.

Lemma: An expanding operation can always allocate one more time slot if the number of requests in the system is no more than $N \times m - (K-1) \times (N-1)$. leaving at least $(K-1) \times (N-1)$ empty spaces.

Proof: When a request r_k changes its membership from $\mathcal{G}_i$ to $\mathcal{G}_j$, the condition for *expanding* group $\mathcal{G}_j$ is $(g_j \bmod N) = 0$, that is, the number of unused spaces in group $\mathcal{G}_j$ is 0. Assume there is an empty time slot, then group $\mathcal{G}_j$ can take this slot. If there is at least $(K-1) \times (N-1)$ empty spaces but no empty time slot, groups other than $\mathcal{G}_i$ and $\mathcal{G}_j$ can have at most $(K-2) \times (N-1)$ empty spaces. Since $\mathcal{G}_j$ has no empty space, $\mathcal{G}_i$ has at least $((K-1) \times (N-1)) - ((K-2) \times (N-1)) = (N-1)$ empty spaces. Then removing r_k will free a time slot which can be used for the expanding operation. □

When a request r_k moves from group $\mathcal{G}_i$ to group $\mathcal{G}_j$, it is removed from $\mathcal{G}_i$ and added to group $\mathcal{G}_j$. The *adding* and *removing* operations can be described as follows.

Removing.

Remove r_k from group $\mathcal{G}_i$.

If $(g_i \bmod N) = 1$, apply *shrinking*. □

Adding.

If $(g_j \bmod N) = 0$, apply *expanding* to allocate one time slot and schedule request r_k to the time slot.

Otherwise, schedule r_k to one of the empty spaces. If all empty spaces have conflict, algorithms described in [14] can be applied to obtain a conflict-free schedule. □

Sometimes the shrinking operation can be postponed until an expanding operation needs a free time slot. This is called *lazy shrinking*. Lazy shrinking can avoid unnecessary re-scheduling.

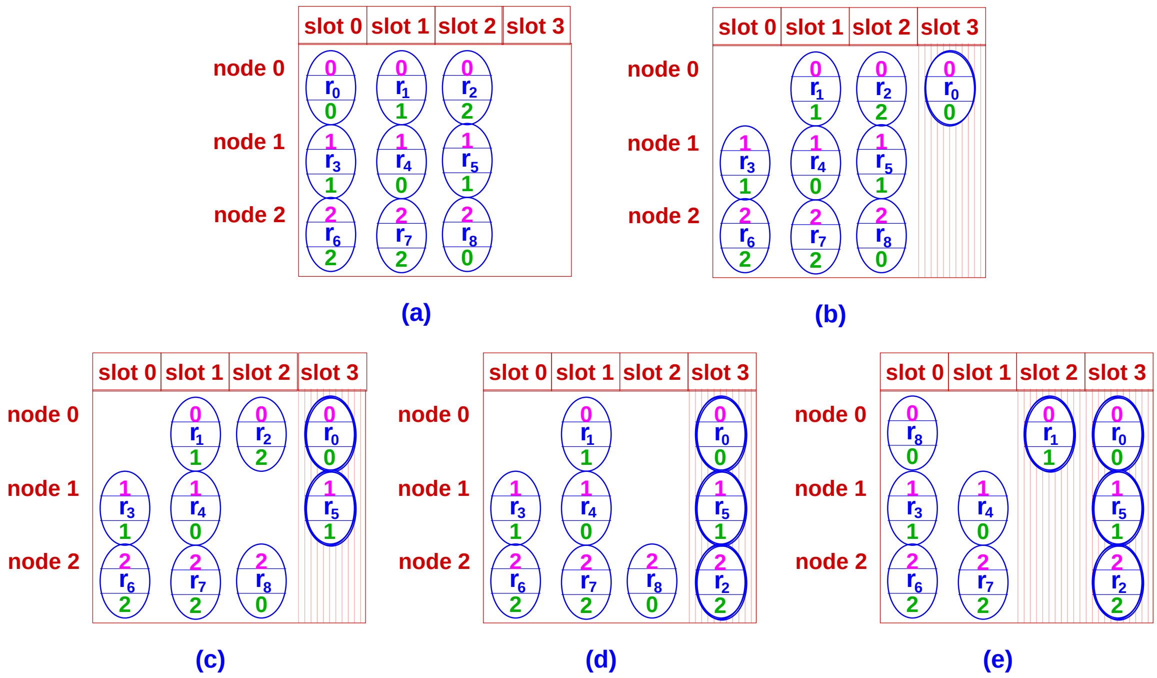

Figure 9. Grouping Approach for Inter-block Skipping.

An example is shown in Figure 9. This system is assumed to handle only two groups, the normal playout and *ff* operation. Since there are 3 nodes and 4 time slots, it is able to handle $N \times m - (K-1) \times (N-1) = 3 \times 4 - (3-1)(2-1) = 10$ requests. However, it is not used in its full capacity as there are only 9 requests. In Figure 9(a), all requests are normal playout. In Figure 9(b), request r_0 changes to *ff*. The expanding operation is invoked which creates a group $\mathcal{G}_f$ consisting of time slot 3. Request r_0 is removed from $\mathcal{G}_1$ and added to $\mathcal{G}_f$, which retrieves blocks at a faster pace. In Figure 9(c), request r_5 changes to *ff* and moves to time slot 3 too. In Figure 9(d), request r_2 changes to *ff* and moves to time slot 3. Since the only space left is in delivery node 2, the delivery node of r_2 is changed from node 0 to node 2. At this time, the shrinking operation can be invoked. However, this operation can be delayed because no expanding operation requests a free time slot. In Figure 9(e), request r_1 changes to *ff*. The expanding operation is invoked. It triggers the shrinking operation which re-schedules all requests of normal playout to time slots 0 and 1 to free time slot 2 for group $\mathcal{G}_f$. Now $\mathcal{G}_f$ consisting of time slots 2 and 3 and r_1 moves to time slot 2.

5.2. Intra-block skipping

The pace of the *ff* operation is f times faster than the normal playout. Thus, for intra-block skipping, a delivery node accesses f blocks in a time slot. We further divide a time slot into f subblocks, each for a different block in a different storage node. Similar to the grouping approach for the inter-block skipping,

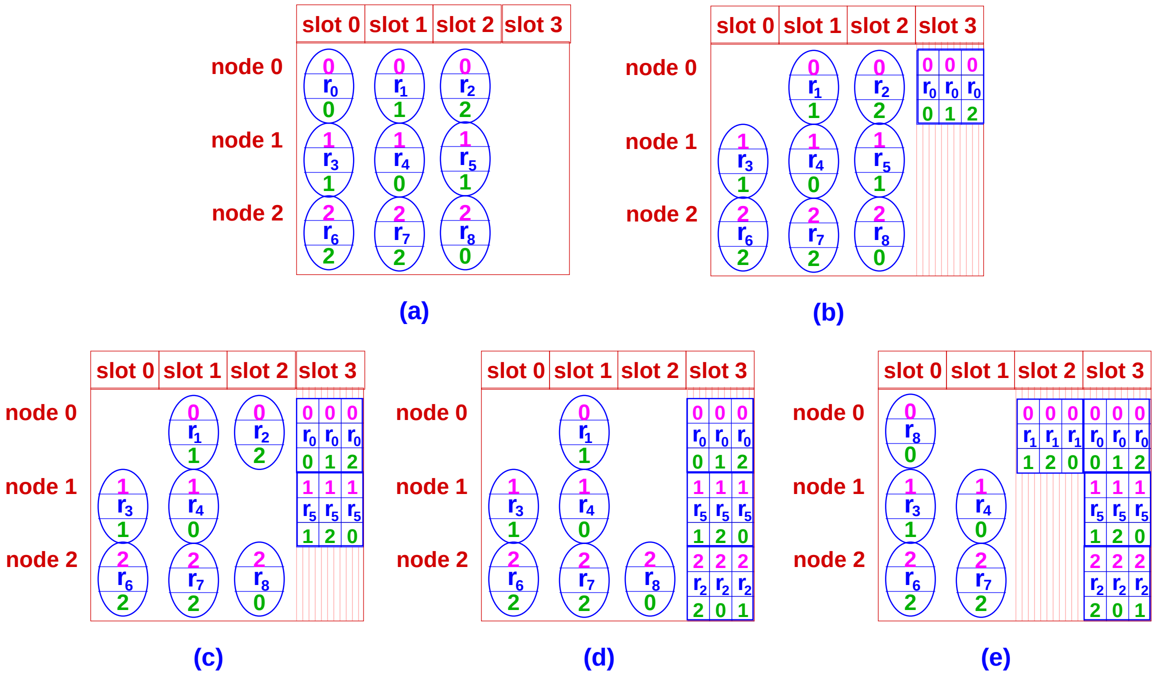

Figure 10. Grouping Approach for Intra-block Skipping.

when one client changes from normal playout to the *ff* operation, it is scheduled to group $\mathcal{G}_f$.

An example is shown in Figure 10. The group expanding and shrinking are the same as in Figure 9. The only difference is when a request moves to group $\mathcal{G}_f$, it accesses f blocks. For example, in Figure 10(b), request r_0 changes to *ff*. It is moved to time slot 3, where it retrieves blocks from storage nodes 0, 1, and 2.

6. Previous Work

Interactive operations can be accommodated by display frames at a rate higher than the normal playout. An analysis shows that statistical guarantees can be provided by reserving only a small portion of its total bandwidth if a small percentage of requests are interactive [5]. Bandwidth smoothing techniques are used to solve the bandwidth problem for interactive operations [6]. It provides the *ff* and *fb* capacity within a VCR-window by using large buffers. Scans to points outside of the VCR-window requires renegotiation.

Displaying frames at a higher speed yields an f-fold increase in the bandwidth. The bandwidth requirement can be reduced by skip frames. A segment skipping approach for interactive operations has been proposed in [4]. This approach can solve the bandwidth problem and is easy to implement. It is found that the segment skipping is an acceptable method for fast-forward and fast-backward. Moreover, a general segment layout scheme is proposed for arbitrary values of the *ff* ratio and the number of storage nodes for a balanced load.

Different data layout schemes for interactive operations to maximize support of different *ff* ratios have been described in [2]. Their technique is applied to fine-grain blocks or segments. Also, a method is descibed to divide the sequence of frames into two sequences: a sequence of frames from a single storage node for normal playout followed by frames from possibly all storage nodes for fast-forward. This idea is similar to the grouping approach.

A method has been described by using the characteristics of video streams and human perceptual tolerances to reduce the bandwidth of interactive operations [12]. Combining sub-stream block placement and access of low-resolution component of the base sub-stream, the bandwidth of interactive operations can be reduced to the same as the normal playout. This technique is applied to the coarse-grain blocks.

7. Concluding Remarks

In this paper, we defined two basic approaches for scheduling interactive operations, the prefetching approach and the grouping approach. Different scheduling algorithms were presented for inter-block and intra-block skipping. With these algorithms, video streams can be precisely scheduled for Quality-of-Service guarantee.

The prefetching approach can fully utilize the system bandwidth to support $N \times m$ requests of video streams. A request is allowed to change its pace at any time, without affecting other requests that do not change their paces. However, it may delay some requests of interactive operations and requires a large amount of buffer space. The grouping approach divides the requests into groups. It does not have those disadvantages such as request delay and extra buffer space. However, the system needs to set aside some bandwidth to accommodate transitions between groups. The shrinking operation may also lead to reallocation and delay of some requests.

Acknowledgments

This research was partially supported by NSF grants CCR-9505300 and CCR-9625784.

References

[1] D. W. Brubeck and L. A. Rowe. Hierarchical storage management in a distributed VOD system. *IEEE Multimedia*, 3(3):37–47, Fall 1996.

[2] M. Buddhikot and G.M. Parulkar. Efficient data layout, scheduling and playout control in MARS. In *Proceedings of the 5th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, 1995.

[3] M. M. Buddhikot, G. M. Parulkar, and J. R. Cox Jr. Design of a large scale multimedia storage sever. *Journal of Computer Networks and ISDN Systems*, pages 504–524, December 1994.

[4] M.S. Chen, D.D. Kandlur, and P.S. Yu. Support for fully interactive playout in a disk-array-based video server. In *ACM Multimedia '94*, October 1994.

[5] J.K. Dey-Sircar, J.D. Salehi, J.F. Kurose, and D. Towsley. Providing vcr capabilities in large-scale video servers. In *ACM Multimedia '94*, pages 25–32, October 1994.

[6] W.C. Feng, F. Jahanian, and S. Sechrest. Providing vcr functionality in a constant quality video-on-demand transportation server. In *ACM Multimedia '96*, pages 127–135, June 1996.

[7] D. Le Gall. A video compression standard for multimedia applications. *Communications of ACM*, 34(4):46–58, April 1991.

[8] D. J. Gemmell, H. M. Vin, D. D. Kandlur, P. V. Rangan, and L. A. Rowe. Multimedia storage servers: A tutorial. *IEEE Computer*, 28(5):40–49, May 1995.

[9] J. Hsieh, M. Lin, J. C. L. Liu, D. Du, and T. Ruwart. Performance of a mass storage system for video-on-demand. *Journal of Parallel and Distributed Computing*, 30(1):147–167, November 1995.

[10] D. Jadav, C. Srinilta, A. Choudhary, and P. B. Berra. Techniques for scheduling I/O in a high performance multimedia-on-demand server. *Journal of Parallel and Distributed Computing*, 30(1):190–203, November 1995.

[11] A. L. N. Reddy. Scheduling and data distribution in a multiprocessor video server. In *the second IEEE Int'l. Conf. on Multimedia Computing and Systems*, 1995.

[12] P.J. Shenoy and H.M. Vin. Efficient support for scan operations in video servers. In *Proceedings of the Third ACM Conference on Multimedia*, November 1995.

[13] R. Tewari, D. Dias, R. Mukherjee, and H. Vin. Design and performance tradeoffs in clustered multimedia servers. In *the Third IEEE Int'l Conf. on Multimedia Computing Systems*, June 1996.

[14] M.Y. Wu and W. Shu. Scheduling for large-scale parallel video servers. In *The Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 126–133, October 1996.