1

SCHEDULING IN MULTIMEDIA SYSTEMS

A. L. Narasimha Reddy

IBM Almaden Research Center,
650 Harry Road, K56/802,
San Jose, CA 95120, USA

ABSTRACT

In video-on-demand multimedia systems, the data has to be delivered to the consumer at regular intervals to deliver smooth playback of video streams. A video-on-demand server has to schedule the service of individual streams to ensure such a smooth delivery of video data. We will present scheduling solutions for individual components of service in a multiprocessor based video server.

1 INTRODUCTION

Several telephone companies and cable operators are planning to install large video servers that would serve video streams to customers over telephone lines or cable lines. These projects envision supporting several thousands of customers with the help of one or several large video servers. These projects aim to store movies in a compressed digital format and route the compressed movie to the home where it can be uncompressed and displayed. These projects aim to compete with the local video rental stores with better service; offering the ability to watch any movie at any time (avoiding the situation of all the copies of the desired movie rented out already) and offering a wider selection of movies. Providing a wide selection of movies requires that a large number of movies be available in digital form. Currently, with MPEG-1 compression, a movie of roughly 90 minute duration takes about 1 GB worth of storage. For a video server storing about 1000 movies (a typical video rental store carries more), we would then have to spend about $500, 000 just for storing the movies on disk at a cost of $0.5/MB. This requirement of large amounts of storage implies that the service providers need to centralize the resources and provide service to a large number of customers to amortize costs. Hence the requirement to build large video servers that can provide service a large number of customers. See [1, 2, 3, 4] for some of the projects on video servers.

If such a large video server serves about 10,000 MPEG-1 streams, the server has to support 10,000 * 1.5 Mbits/sec or about 2 GBytes/sec of I/O bandwidth. Multiprocessor systems are suitable candidates for supporting such large amounts of real-time I/O bandwidth required in these large video servers. We will assume that a multiprocessor video server is organized as shown in Fig. 1. A number of nodes act as *storage nodes*. Storage nodes are responsible for storing video data either in memory, disk, tape or some other medium and delivering the required I/O bandwidth to this data. The system also has *network nodes*. These network nodes are responsible for requesting appropriate data blocks from the storage nodes and routing them to the customers. Both these functions can reside on the same multiprocessor node, i.e., a node can be a storage node, or a network node or both at the same time. Each request stream would originate at one of the several network nodes in the system and this network node would be responsible for obtaining the required data for this stream from the various storage nodes in the system and delivering it to the consumer. The data transfer from the network node to the consumer's monitor would depend on the medium of delivery, telephone wire, cable or the LAN.

We will assume that the video data is stored on disk. Storing the video data on current tertiary mediums such as tapes is shown to be not attractive from price performance analysis [5]. Storing the video in memory may be attractive for frequently accessed video streams. We will assume that the video data is stored on disk to address the more general problem. The work required to deliver a video stream to the consumer can then be broken down into three components: (1) the disk service required to read the data from the disk into the memory of the storage node, (2) the communication required to transfer the data from the storage node memory to the network node's memory and (3) the communication required over the delivery medium to transfer the data from the network node memory to the consumer's monitor. These three phases of service may be present or absent depending on the system's configuration. As pointed out already, if the video data is stored in memory, the service in phase 1 is not needed. If the video server does not employ a multiprocessor system, the service in phase 2 is not needed. In this chapter, we will deal with the scheduling problems in phases 1 and 2. If the consumer's monitor is attached to the network node directly, the service in phase 3 is not needed. Service in phase 3 is dependent on the delivery medium and we will not address it here.

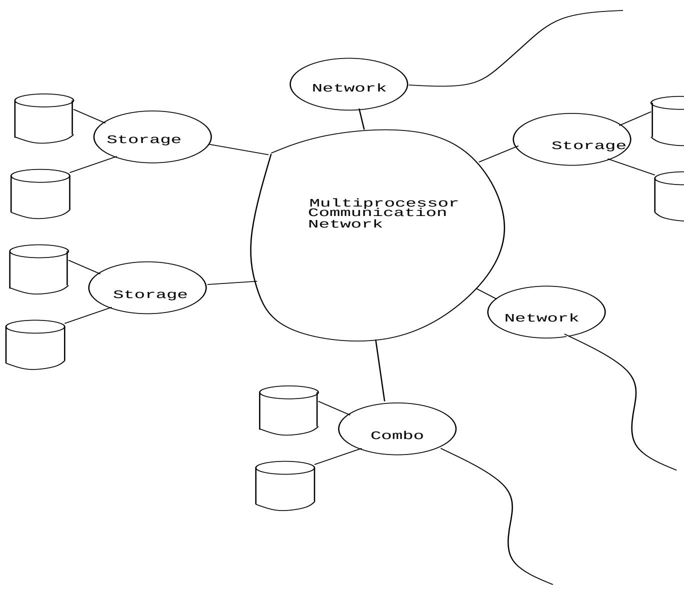

Figure 1 System model of a multiprocessor video server.

Deadline scheduling [6, 7] is known to be an optimal real-time scheduling strategy when the task completion times are known in advance. The disk service time (in phase 1) is dependent on the relative position of the request block with respect to the head on the disk. The communication time in the network (in phase 2) is dependent the contention for network resources and this contention varies based on the network load. Service for one video stream requires multiple resources (disk at the storage node, links, input and output ports in the network) unlike the assumption of requiring only one resource in these studies. Hence, these results cannot be directly applied to our problem.

The organization of the system and the data distribution over the nodes in the system impact the overall scheduling strategy. In the next section, we will describe some of the options in distributing the data and their impact on the different phases of service. In Section 3, we will discuss scheduling algorithms for (phase 1) disk service. In Section 4, we will describe a method for scheduling the multiprocessor network resources. Section 5 concludes this chapter with some general remarks and future directions.

2 DATA ORGANIZATION

If a movie is completely stored on a single disk, the supportable number of that movie will be limited by the bandwidth of a single disk. As shown earlier by [8], a 3.5" 2-GB IBM disk can support upto 20 streams. A popular movie may receive more than 20 requests over the length of the playback time of that movie. To enable serving a larger number of streams of a single movie, each movie has to be striped across a number of nodes. As we increase the number of nodes for striping, we increase the bandwidth for a single movie. If all the movies are striped across all the nodes, we also improve the load balancing across the system since every node in the system has to participate in providing access to each movie.

The width of striping (the number of disks a movie may be distributed on) determines a number of characteristics of the system. The wider the striping, the larger the bandwidth for any given movie and the better the load balancing. A disk failure affects a larger number of movies when wider striping is employed. When more disk space is needed in the system, it is easier to add a number of disks equal to the width of striping. Hence, wider striping means a larger unit of incremental growth of disk capacity. All these factors need to be considered in determining the width of striping. For now, we will assume that all the movies are striped across all the disks in the system. In a later section, we will discuss the effects of employing smaller striping widths. The unit of striping across the storage nodes is called a block.

Even though movies are striped across the different disks to provide high bandwidth for a movie, it is to be noted that a single MPEG-1 stream bandwidth of 1.5 Mbits/sec can be sufficiently supported by the bandwidth of one disk. Requests of a movie stream can be served by fetching individual blocks at a time from a single disk. Striping provides simultaneous access to different blocks of the movie from different disks and thus increases the bandwidth available to a movie. Higher stream rates of MPEG-2 can also be supported by requests to individual disks. We will assume that a single storage node is involved in serving a request block.

Data organization has an impact on the communication traffic within the system. During the playback of a move, a network node responsible for delivering that movie stream to the user has to communicate with all the storage nodes where this movie is stored. This results in a point to point communication from all the storage nodes to the network node (possibly multiple times depending on the striping block size, the number of nodes in the system and the length of

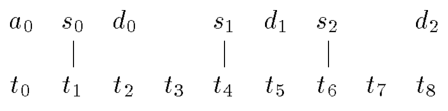

Figure 2 Progess of disk service of a request.

the movie) during the playback of the movie. Since each network node will be responsible for a number of movie streams, the resulting communication pattern is random point-to-point communication among the nodes of the system. It is possible to achieve some locality by striping the movies among a small set of nodes and restricting that the network nodes for a movie be among this smaller set of storage nodes.

3 DISK SCHEDULING

A real-time request can be denoted by two parameters (c, p), where p is the period at which the real-time requests are generated and c is the service time required in each period. The earliest-deadline-first (EDF) [6] algorithm showed that tasks can be scheduled by EDF if and only if the task utilization $\sum_{i=1}^{n} c_i/p_i < 1$. We will specify the real-time requests by specifying the required data rate in kbytes/sec. The time at which a periodic request is started is called the release time of that request. The time at which the request is to be completed is called the deadline for that request. Requests that do not have real-time requirements are termed aperiodic requests.

Fig. 2 shows the progress of disk service for a stream. Request for block 0 is released at time t_0 and the request is actually scheduled at time t_1 and is denoted by event s_0. The block 0 is consumed (event d_0) at time beginning t_2. The time between the consumption of successive blocks of this stream $d_{i+1} - d_i$ has to be maintained constant for providing glitch-free service to the user. For example, when 256 Kbyte blocks are employed for MPEG-1 streams, this is equal to about 1.28 seconds. The time between the scheduling events of successive blocks need not be constant. Only requirement is that the blocks be scheduled sufficiently in advance to guarantee that $d_{i+1} - d_i$ can be maintained constant. This is shown in Fig. 2. The vertical bars in the picture represent the size of the request block.

In real-time systems, algorithms such as earliest deadline first, and least slack time first are used. As pointed out earlier, strict real-time scheduling policies such as EDF may not be suitable candidates because of the random disk service times and the overheads associated with seeks and rotational latency.

Traditionally, disks have used seek optimization techniques such as SCAN or shortest seek time first (SSTF) for minimizing the arm movement in serving the requests [9]. These techniques reduce the disk arm utilization by serving requests close to the disk arm. The request queue is ordered by the relative position of the requests on the disk surface to reduce the seek overheads. Even though these techniques utilize the disk arm efficiently, they may not be suitable for real-time environments since they do not have a notion of time or deadlines in making scheduling decisions.

Video-on-demand systems may have to serve aperiodic requests also. It is necessary to ensure that periodic requests do not miss their deadlines while providing reasonable response times for aperiodic requests. A similar problem is studied in [10]. I/O requests are known to be bursty. A burst of aperiodic requests should not result in missing the guarantees for the periodic requests.

The scheduling algorithm should be fair. For example, shortest seek time first, is not a fair scheduling algorithm since requests at the edges of the disk surface may get starved. If the scheduling algorithm is not fair, an occasional request in the stream may get starved of service and hence will result in missing the deadlines.

To guarantee the service of scheduled real-time requests, worst-case assumptions about seek and latency overheads can be made to bound random disk service times to some constant service time. Another approach to making service times predictable is to make the request size so large that the overheads form a smaller fraction of the request service time. This approach may result in large demands on buffer space. Our approach to this problem is to reduce the overheads in service time by making more efficient use of the disk arm either by optimizing the service schedule and/or by using large requests. By reducing the random overheads, we make the service time more predictable. We will describe two techniques in the next section, larger requests and delayed deadlines, for reducing the variances in the service time.

We will consider three scheduling algorithms, CSCAN, EDF and SCAN-EDF. CSCAN is a seek optimizing disk scheduling algorithm which traverses the disk surface in one direction from innermost pending request to the outermost pending request and then jumps back to serving the innermost request [9].

EDF is the earliest deadline first policy. SCAN-EDF is a hybrid algorithm that incorporates the real-time aspects of EDF and seek optimization aspects of SCAN. CSCAN and EDF are well known algorithms and we will not elaborate on them further.

3.1 SCAN-EDF scheduling algorithm

SCAN-EDF disk scheduling algorithm combines seek optimization techniques and EDF in the following way. Requests with earliest deadline are served first. But, if several requests have the same deadline, these requests are served by a seek-optimizing scheduling algorithm.

SCAN-EDF applies seek optimization to only those requests that have the same deadline. Its efficiency depends on how often these seek optimizations can be applied, or on the fraction of requests that have the same deadlines. SCAN-EDF serves requests in batches or rounds. Requests are given deadlines at the end of a batch. Requests within a batch then can be served in any order and SCAN-EDF serves the requests within a batch in a seek optimizing order. In other words, requests are assigned deadlines that are multiples of the period p.

When the requests have different data rate requirements, SCAN-EDF can be combined with a periodic fill policy [11] to let all the requests have the same deadline. Requests are served in a cycle with each request getting an amount of service time proportional to its required data rate, the length of the cycle being the sum of the service times of all the requests. All the requests in the current cycle can then be given a deadline at the end of the current cycle.

A more precise description of the algorithm is given below.

```
SCAN-EDF algorithm
Step 1: let T = set of tasks with the earliest deadline
Step 2: if $|T| = 1$, (there is only a single request in T), service that request.
else let $t_1$ be the first task in T in the scan direction, service $t_1$.
go to Step 1.
```

The scan direction can be chosen in several ways. In Step 2, if the tasks are ordered with the track numbers of tasks such that $N_1 <= N_2 <= ... <= N_l$, then we obtain a CSCAN type of scheduling where the scan takes place only from smallest track number to the largest track number. If the tasks are ordered such that $N_1 >= N_2 >= ... >= N_l$, then we obtain a CSCAN type of scheduling where the scan takes place only from largest track number to the smallest track number. If the tasks can be ordered in either of the above forms depending on the relative position of the disk arm, we get (elevator) SCAN type of algorithm.

SCAN-EDF can be implemented with a slight modification to EDF. Let D_i be the deadlines of the tasks and N_i be their track positions. Then the deadlines can be modified to be $D_i + f(N_i)$, where $f()$ is a function that converts the track numbers of the tasks into small perturbations to the deadlines. The perturbations have to be small enough such that $D_i + f(N_i) > D_j + f(N_j)$, if $D_i > D_j$. We can choose $f()$ in various ways. Some of the choices are $f(N_i) = N_i/N_{max}$ or $f(N_i) = N_i/N_{max} - 1$, where N_{max} is the maximum track number on the disk or some other suitably large constant. For example, let tasks A, B, and C have the same deadline 500 and ask for data from tracks 347, 113, and 851 respectively. If $N_{max} = 1000$, the modified deadlines of A, B, and C become 499.347, 499.113 and 499.851 respectively when we use $f(N_i) = N_i/N_{max} - 1$. When these requests are served by their modified deadlines, they are served in the track order. A request with a later deadline will be served after these three requests are served. Other researchers have proposed similar scheduling policies [12, 13, 2].

3.2 Buffer space tradeoff

Available buffer space has a significant impact on the performance of the system. Real-time requests typically need some kind of response before the next request is issued. Hence, the deadlines for the requests are made equal to the periods of the requests. The multimedia I/O system needs to provide a constant data rate for each request stream. This constant data rate can be provided in various ways. When the available buffer space is small, the request stream can ask for small pieces of data more frequently. When the available buffer space is large, the request stream can ask for larger pieces of data with correspondingly larger periods between requests. This tradeoff is significant since the efficiency of the disk service is a varying function of the request size. The disk arm is more efficiently used when the request sizes are large and hence it may be possible to support larger number of multimedia streams at a single disk. Fig. 3(a) shows two streams providing the same constant stream rate, the second request stream scheduling twice as large requests at half the frequency of the first stream. A (c,p) request supports the same data rate as a (2c,2p) request if larger buffers are provided, at the same time improving the efficiency of the disk. However, this improved efficiency has to be weighed against the increased buffer space requirements. Each request stream requires a buffer for the consuming process and one buffer for the producing process (disk). If we

$$\begin{array}{ccccccccc}
a_0 & s_0 & d_0 & & s_1 & d_1 & s_2 & & d_2 \\
 & | & & & | & & | & & \\
t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 \\
 & || & & & & & || & & \\
a'_0 & s'_0 & d'_0 & & & & s'_1 & & d'_1
\end{array}$$

(a). Larger requests.

$$\begin{array}{cccccccccc}
a_0 & s_0 & d_0 & & s_1 & d_1 & s_2 & & d_2 & \\
 & | & & & | & & | & & & \\
t_0 & t_1 & t_2 & t_3 & t_4 & t_5 & t_6 & t_7 & t_8 & t_9 \\
 & | & | & & | & & & & & \\
a'_0 & s'_0 & s'_1 & d'_0 & s'_2 & & d'_1 & & & d'_2
\end{array}$$

(b). Delayed deadlines.

Figure 3 Request streams with same data rate requirements.

decide to issue requests at the size of S, then the buffer space requirement for each stream is $2S$. If the I/O system supports n streams, the total buffer space requirement is $2nS$.

There is another tradeoff that is possible. The deadlines of the requests need not be chosen equal to the periods of the requests. For example, we can defer the deadlines of the requests by a period and make the deadlines of the requests equal to $2p$. This gives more time for the disk arm to serve a given request and may allow more seek optimizations than that are possible when the deadlines are equal to the period p. Fig. 3(b) shows two streams providing the same constant stream rate, but with different charecteristics of progress along the time scacle. The stream with the deferred deadlines provides more time for the disk to service a request before it is consumed. This results in a scenario where the consuming process is consuming buffer 1, the producing process (disk) is reading data into buffer 3 and buffer 2 is filled earlier by the producer and waiting consumption. Hence, this raises the buffer requirements to $3S$ for each request stream. The extra time available for serving a given request allows more opportunities for it to be served in the scan direction. This results in more efficient use of disk arm and as a result, larger number of request streams can be supported at a single disk. A similar technique called work-ahead is utilized in [12]. Scheduling algorithms for real-time requests when the deadlines are different from the periods are reported in [14, 15].

Both these techniques, larger requests with larger periods and delayed deadlines, increase the latency of service at the disk. When the deadlines are delayed, the data stream cannot be consumed until two buffers are filled as opposed to waiting for one filled buffer when deadlines are equal to periods. When larger requests are employed, longer time is taken for reading a larger block and hence a longer time before the multimedia stream can be started. Larger requests increase the response time for aperiodic requests as well since the aperiodic requests will have to wait for a longer time behind the current real-time request that is being served. The improved efficiency of these techniques needs to be weighed against the higher buffer requirements and the higher latency for starting a stream.

3.3 Performance Evaluation

In this section, we compare the three scheduling algorithms CSCAN, EDF and SCAN-EDF through simulations. We present the simulation model used to obtain these results.

Simulation model

A disk with the parameters shown in Table 1 is modeled. It is assumed that the disk uses split-access operations or zero latency reads. In split-access operation, the request is satisfied by two smaller requests if the read-write head happens to be in the middle of the requested data at the end of the seek operation. The disk starts servicing the request as soon as any of the requested blocks comes under the read-write head. For example, if a request asks for reading blocks numbered 1,2,3,4 from a track of eight blocks 1,2,...8, and the read-write head happens to get to block number 3 first, then blocks 3 and 4 are read, blocks 5,6,7,8 are skipped over and then blocks 1 and 2 are read. In such operation, a disk read/write of a single track will not take more than one single revolution. Split access operation is shown to improve the request response time considerably in [16]. Split-access operation, besides reducing the average service time of a request, also helps in reducing the variability in service time.

Each real-time request stream is assumed to require a constant data rate of 150 kB/sec. This roughly corresponds to the data rate requirements for a CDROM data stream. Each request stream is modeled by an independent request generator. The number of streams is a parameter to the simulator.

Time for one rotation	11.1 ms
Avg. seek	9.4 ms
sectors/track	84
sector size	512 bytes
tracks/cylinder	15
cylinders/disk	2577
seek cost function	nonlinear
Min. seek time s_0	1.0 ms

Table 1 Disk parameters used in simulations.

Aperiodic requests are modeled by a single aperiodic request generator. Aperiodic requests are assumed to arrive with an exponential distribution. The mean time between arrivals is varied from 25 ms to 200 ms. If we allow unlimited service for the aperiodic requests, a burst of aperiodic requests can disturb the service of real-time requests considerably. It is necessary to limit the number of aperiodic requests that may be served in a given period of time. A separate queue could be maintained for these requests and these requests can be released at a rate that is bounded by a known rate. A multimedia server will have to be built in this fashion to guarantee meeting the real-time schedules. Hence, we modelled the arrival of aperiodic requests by a single request generator. In our model, if the aperiodic requests are generated faster than they are being served, they are queued in a separate queue.

The service policy for aperiodic requests depended on the scheduling policy employed. In EDF and SCAN-EDF, they are served using the immediate server approach [10] where the aperiodic requests are given higher priority over the periodic real-time requests. The service schedule of these policies allows a certain number of aperiodic requests each period and when sufficient number of aperiodic requests are not present, the real-time requests make use of the remaining service period. This policy of serving aperiodic requests is employed so as to provide reasonable response times for both aperiodic and periodic requests. This is in contrast to earlier approaches where the emphasis has been only on providing real-time performance guarantees. In CSCAN, aperiodic requests are served in the CSCAN order.

Each aperiodic request is assumed to ask for a track of data. The request size for the real-time requests is varied among 1, 2, 5, or 15 tracks. The effect of request size on number of supportable streams is investigated. The period between two requests of a request stream is varied depending on the request size to support a constant data rate of 150 kB/sec. The requests are assumed to be uniformly distributed over the disk surface.

Two systems, one with deadlines equal to the request periods and the second with deadlines equal to twice the request periods are modeled. A comparison of these two systems gives insight into how performance can be improved by deferring the deadlines.

Two measures of performance are studied. The number of real-time streams that can be supported by each scheduling policy is taken as the primary measure of performance. We also look at the response time for aperiodic requests. A good policy will offer good response times for aperiodic requests while supporting large number of real-time streams.

Each experiment involved running 50,000 requests of each stream. The maximum number of supportable streams n is obtained by increasing the number of streams incrementally till $n+1$ where the deadlines cannot be met. Twenty experiments were conducted, with different seeds for random number generation, for each point in the figures. The minimum among these values is chosen as the maximum number of streams that can be supported. Each point in the figures is obtained in this way. The minimum is chosen (instead of the average) in order to guarantee the real-time performance.

3.4 Results

Maximum number of streams

Fig. 4 shows the results from simulations. The solid lines correspond to a system with extended deadlines (=2p) and the dashed lines are for the system where deadlines are equal to request periods.

It is observed that deferring deadlines improves the number of supportable streams significantly for all the scheduling policies. The performance improvement ranges from 4 streams for CSCAN to 9 streams for SCAN-EDF at a request size of 1 track.

When deadlines are deferred, CSCAN has the best performance. SCAN-EDF has performance very close to CSCAN. EDF has the worst performance. EDF scheduling results in random disk arm movement and this is the reason for poor

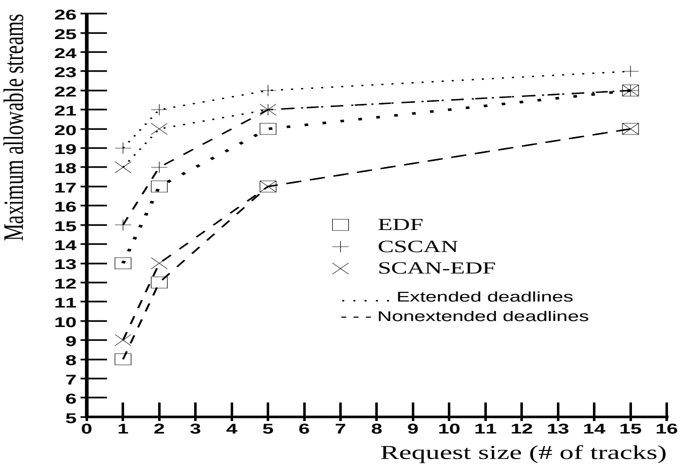

Figure 4 Performance of different scheduling policies.

performance of this policy. Fig. 4 clearly shows the advantage of utilizing seek optimization techniques.

Fig. 4 also presents the improvements that are possible by increasing the request size. As the request size is increased from 1 track to 15 tracks, the number of supportable streams keeps increasing. The knee of the curve seems to be around 5 tracks or 200 kbytes. At larger request sizes, the different scheduling policies make relatively less difference in performance. At larger request sizes, the transfer time dominates the service time. When seek time overhead is a smaller fraction of service time, the different scheduling policies have less scope for optimizing the schedule. Hence, all the scheduling policies perform equally well at larger request sizes.

At a request size of 5 tracks, i.e., 200 kbytes/buffer, minimum of 2 buffers/stream corresponds to 400 kbytes of buffer space per stream. This results in a demand of 400 kbytes * 20 = 8Mbytes of buffer space at the I/O system for supporting 20 streams. If deadlines are deferred, this corresponds to a requirement of 12 Mbytes. When such amount of buffer space is not available, smaller request sizes need to be considered.

At smaller request sizes, deferring the deadlines has a better impact on performance than increasing the request size. For example, at a request size of 1 track and deferred deadlines (with buffer requirements of 3 tracks) EDF supports 13 streams. When deadlines are not deferred, at a larger request size of 2 tracks and buffer requirements of 4 tracks, EDF supports only 12 streams. A similar trend is observed with other policies as well. A similar observation can be made when request sizes of 2 and 5 tracks are compared.

Aperiodic response time

Fig. 5 shows the response time for aperiodic requests. The figure shows the aperiodic response time when 8, 12, 15, 18 real-time streams are being supported in the system at request sizes of 1, 2, 5, and 15 tracks respectively. It is observed that CSCAN has the worst performance and SCAN-EDF has the best performance. With CSCAN, on an average, an aperiodic request has to wait for half a sweep for service. This may result in waiting behind half the number of real-time requests. In SCAN-EDF, EDF, aperiodic requests are given higher priorities by giving them shorter deadlines (100 ms from the issuing time). In these strategies, requests with shorter deadlines get higher priority. As a result, aperiodic requests typically wait behind only the current request that is being served. Among these three policies, the slightly better performance of SCAN-EDF is due to the lower arm utilizations.

From Figures 4 and 5, it is seen that SCAN-EDF performs well under both measures of performance. CSCAN performs well in supporting real-time requests but does not have very good performance in serving the aperiodic requests. EDF, does not perform very well in supporting real-time requests but offers good response times for aperiodic requests. SCAN-EDF supports almost as many real-time streams as CSCAN and at the same time offers the best response times for aperiodic requests. When both the performance measures are considered, SCAN-EDF has better characteristics.

Effect of aperiodic request arrival

Fig. 6 shows the effect of aperiodic request arrival rate on the number of sustainable real-time streams. It is observed that aperiodic request arrival rate has a significant impact on all the policies. Except for CSCAN, all other policies support less than 5 streams at an inter-arrival time of 25 ms. Figure 6 shows that the inter-arrival time of aperiodic requests should not be below 50 ms if more than 10 real-time streams need to be supported at the disk. CSCAN

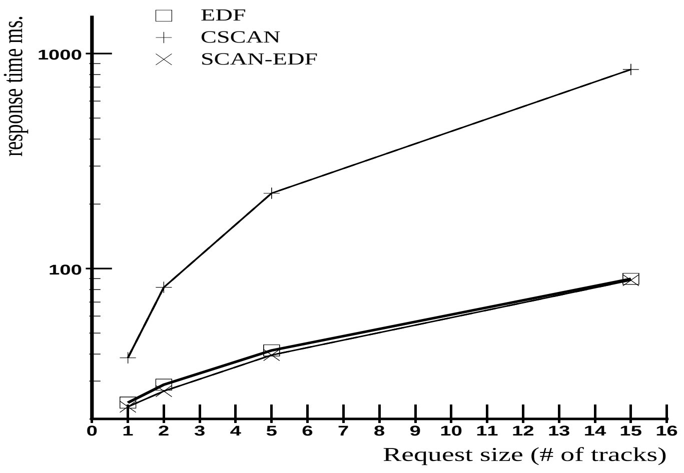

Figure 5 Aperiodic response time with different scheduling policies.

treats all requests equally and hence higher aperiodic request arrival time only reduces the time available for the real-time request streams and does not alter the schedule of service. In other policies, since aperiodic requests are given higher priorities, higher aperiodic request arrival rate results in less efficient arm utilization due to more random arm movement. Hence, other policies see more impact on performance due to higher aperiodic request arrival rate.

Multiple data rates

Fig. 7 shows the performance of various scheduling policies when requests with different data rates are served. The simulations modeled equal number of three different data rates of 150 kB/sec, 8 kB/sec and 176 kB/sec with aperiodic requests arriving at a rate of 200ms. The performance trends are similar to the earlier results.

A more detailed performance study can be found in [8] where several other factors such as the impact of a disk array are considered.

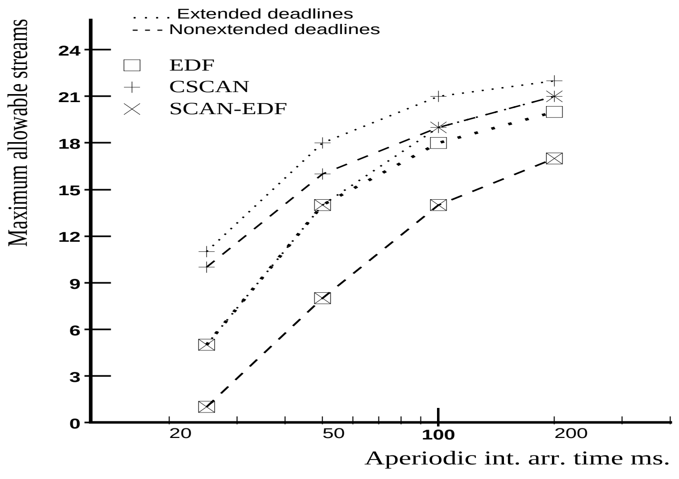

Figure 6 Effect of aperiodic request arrival rate on the number of streams.

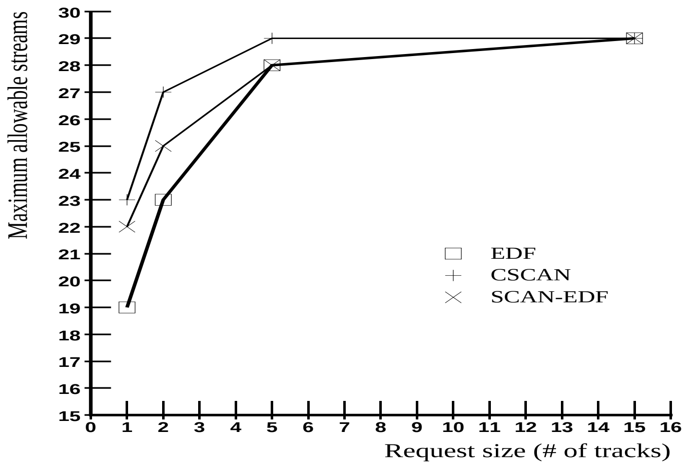

Figure 7 Performance of various policies with multiple data rates.

3.5 Analysis of SCAN-EDF

In this section, we will present an analysis of SCAN-EDF policy and show how request service can be guaranteed. We assume that the disk seek time can be modeled by the following equation s(m) = $s_0 + m * s_1$, where $s(m)$ is the seek time for m tracks, s_0 is the minimum seek time. This equation assumes that the seek time is a linear function of the number of tracks. This is a simplifying assumption to make the analysis easy (in simulations earlier, we used the actual measured seek function of one of the IBM disks). The value of s_1 can be chosen such that the seek time function $s(m)$ gives an upper bound on the actual seek time. Let M denote the number of tracks on the disk and T the track capacity. We will denote the required data rate for each stream by C. We also assume that the disk requests are issued at a varying rate, but always in multiples of track capacity. Let kT be the request size. Since C is the required data rate for each stream, the period for a request stream $p = kT/C$. If r denotes the data rate of the disk in bytes/sec, $r = T/(\mathit{rotation\ time})$. Disk is assumed to employ split-access operation and hence no latency penalty. This analysis assumes that there are no aperiodic requests. These assumptions are made so that we can establish an upper bound on performance.

SCAN-EDF serves requests in batches. Each batch is served in a scan order for meeting a particular deadline. We assume that the batch of n requests are uniformly placed over the disk surface. Hence the seek time cost for a complete sweep of n requests can be given by $s_1 * M + n * s_0$. This assumes that the disk arm sweeps across all the M tracks in serving the n requests. The read time cost for n requests is given by $n * kr$. The total time for one sweep is the time taken for serving the n requests plus the time taken to move the disk arm back from the innermost track to the outermost track. This innermost track to outermost track seek takes $s_0 + M * s_1$ time. Hence, the total time for serving one batch of requests is given by Q = $(n * s_0 + M * s_1 + n * kr) + s_0 + M * s_1$ $= n * (s_0 + kr) + 2M * s_1 + s_0$. The worst-case for a single stream results when its request is the first request to be served in a batch and is the last request to be served in the next batch of requests. This results in roughly 2Q time between serving two requests of a stream. This implies the number of streams n is obtained when p = 2Q or n = $(kT/C - 4M * s_1 - 2 * s_0)/2 * (s_0 + kr)$. However, this bound can be improved if we allow deadline extension. If we allow the deadlines to be extended by one period, the maximum number of streams n is obtained when $n = (kT/C - 2M * s_1 - s_0)/(s_0 + kr)$.

The time taken to serve a batch of requests through a sweep, using SCAN-EDF, has little variance. The possible variances of individual seek times could add

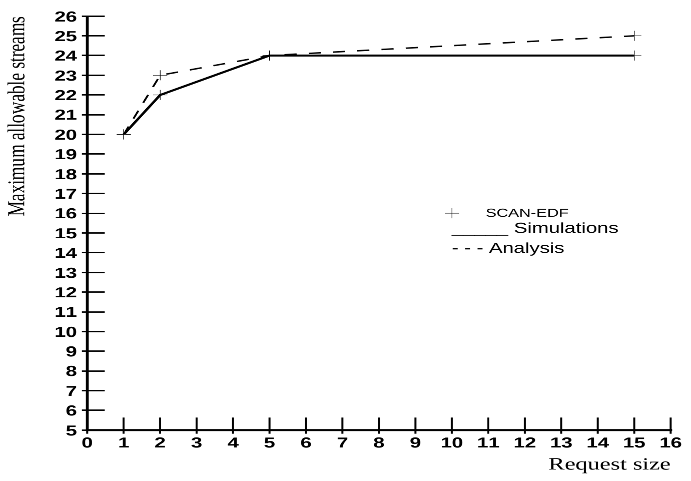

Figure 8 Comparison of analysis with simulation results.

up to a possible large variance if served by a strict EDF policy. SCAN-EDF reduces this variance by serving all the requests in a single sweep across the disk surface. SCAN-EDF, by reducing the variance, reduces the time taken for serving a batch of requests and hence supports larger number of streams. This reduction in the variance of service time for a batch of requests has a significant impact on improving the service time guarantees. Larger request sizes, split-access operation of disk arm also reduce the variance in service time by limiting the random, variable components of the service time to a smaller fraction.

Fig. 8 compares the predictions of analysis with results obtained from simulations for extended deadlines. For this experiment, aperiodic requests were not considered and hence the small difference in the number of streams supportable by SCAN-EDF from Fig. 4. It is observed that the analysis is very close to the simulation results. The error is within one stream.

3.6 Effect of SCSI bus contention

In today's systems, disks are connected to the rest of the system through a peripheral device bus such as a SCSI bus. To amortize the costs of SCSI controllers, multiple disks may be connected to the system on a single bus. SCSI bus, for example can support 10 MB/sec (also 20 MB/sec with wider buses). Since most disks have raw data rates in the range of 3-5 MB/sec, two to three disks can be attached to a single SCSI bus without affecting the total throughput of the disks. However, even when the raw data rate of the SCSI bus may be fast enough to support two to three disks, in a real-time environment, this shared bus could add delays to individual transfers and may result in missed deadlines. To study the affect of the SCSI bus contention on the throughput of the real-time streams in a system, we simulated 3 disks attached to a single bus. Each of these disks has the same characteristics as described earlier in Table 1. The raw data rate of these disks is 3.8 MB/sec. This implies that the total throughput of these disks slightly exceeds the rated bandwidth of the SCSI bus at 10 MB/sec. However, due to seek and latency penalties paid for each access, the disks do not sustain the 3.8 MB/sec for long periods of time.

SCSI bus is a priority arbitrated bus. If more than one disk tries to transfer data on the bus, disk with higher priority always gets the bus. Hence, it is possible that real-time streams being supported by the lower priority disks may get starved if the disk with higher priority continues to transmit data. Better performance may be obtained with other arbitration policies such as a round-robin policy. For multimedia applications, other channels such as the proposed SSA by IBM, which operates as a time division multiplexed channel, are more suitable.

Fig. 9 shows the impact of SCSI bus contention on the number of streams that can be supported. The number of streams supported is less than three times that of the individual disk real-time request capacity. This is mainly due to the contention on the bus. At a request size of 5 tracks, the ratio of the number of streams supported in a three disk configuration to that of a single disk configuation varies from 2.1 in the system with extended deadlines to 1.8 in the system without extended deadlines. This again shows that deadline extension increases the chances of meeting deadlines, in this case smoothing over the bus contention delays. Figure 9 assumes that the number of streams on the three disks differ at most by one. If the higher priority disk is allowed to support more real-time streams, the total throughput of real-time streams out of the three disks would be lower. We observed a sharp reduction in the number of streams supported at the second and third disks when the number

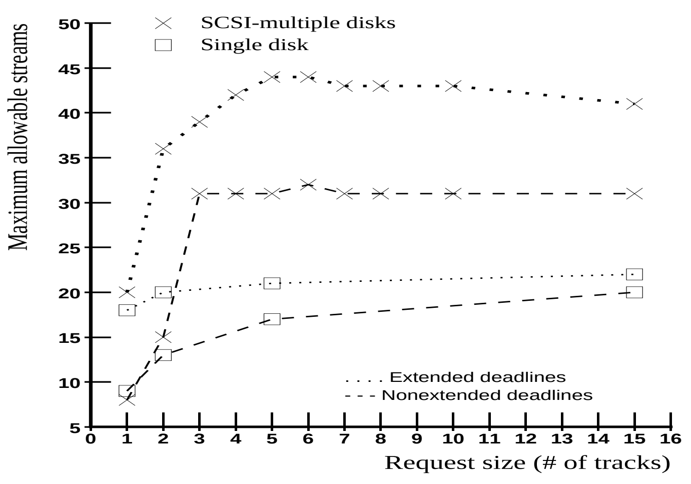

Figure 9 Performance of SCAN-EDF policy with SCSI bus contention.

of streams supported at the first disk is increased even by one. For example, at a request size of 5 tracks and extended deadlines, SCAN-EDF supported 15, 14 and 14 streams at the three disks but only supported 7 streams at the second and the third disks when the number is raised to 16 at the first disk.

Another key difference that is noted is that with SCSI bus contention, there is a peak in supportable request streams as the request size is increased. With larger blocks of transfer, the SCSI bus could be busy for longer periods of time when a disk with lower priority wants to access the bus and thus causing it to miss a deadline. From the figure it is found that the optimal request size for a real-time stream is roughly around 5 tracks.

The optimal request size is mainly related to the relative transfer speeds of the SCSI bus and the raw disk. When a larger block size is used, disk transfers are more efficient, but as explained earlier, disks with lower priority see larger delays and hence are more likely to miss deadlines. When a shorter block is used, disk transfers are less efficient, but the latency to get access to SCSI bus is shorter. This tradeoff determines the optimal block size.

Most of the modern disks have a small buffer on the disk arm for storing the data currently being read by the disk. Normally, the data is filled into this buffer by the disk arm at the media transfer rate (in our case, at 3.8 MB/sec) and transfered out of this buffer at the SCSI bus rate (in our case, at 10 MB/sec). If this arm buffer is not present, the effective data rate of SCSI bus will be reduced to the media transfer rate or lower. When the disk arm buffers are present, SCSI transfers can be intiated by the individual disks in an intelligent fashion such that the SCSI data rate can be maintained high while providing that the individual transfers are completed across the SCSI bus as they are being completed at the disk surface. IBM's Allicat drive utilizes this policy for transfering in and out of its 512 kbyte arm buffer and this is what is modeled in our simulations. Without this arm buffer, when multiple disks are configured on a single SCSI bus, the real-time performance will be significantly lower.

4 NETWORK SCHEDULING

We will assume that time is divided into a number of 'slots'. The length of a slot is roughly equal to the average time taken to transfer a block of movie over the multiprocessor network from a storage node to a network node. Average delivery time itself is not enough in choosing a slot; we will comment later on how to choose the size of a slot. Each storage node starts transferring a block to a network node at the beginning of a slot and this transfer is expected to finish by the end of the slot. It is not necessary for the transfer to finish strictly within the slot but for ease of presentation, we will assume that a block transfer completes within a slot.

The time taken for the playback of a movie block is called a frame. The length of the frame depends on the block size and the stream rate. For a block size of 256 Kbytes and a stream rate of 200 Kbytes/sec, the length of a frame equals $256/200 = 1.28$ seconds. We will assume that a basic stream rate of MPEG-1 quality at 1.5Mbits/sec is supported by the system. When higher stream rates are required, multiple slots are assigned within a frame to achieve the required delivery rate for that stream. It is assumed that all the required rates are supported by transferring movie data in a standard block size (which is also the striping size).

For a given system, the block size is chosen first. For a given basic stream rate, the frame length is then determined. Slot width is then approximated by dividing the block size by the average achievable data rate between a pair of nodes in the system. This value is adjusted for variations in communication delay. Also, we require that frame length be an integer multiple of the slot width. From here, we will refer to the frame length in terms of number of slots per frame 'F'.

The complete schedule of movies in the system can be shown by a table as shown in Fig. 10. The example system has 4 nodes, 0, 1, 2, and 3 and contains 5 movies A, B, C, D, and E. The distribution of movies A, B, C, D, E across the nodes 0, 1, 2, and 3 is shown in Fig. 10 (a). For example, movie E is distributed cyclically across nodes in the order of 2, 1, 0, and 3. For this example, we will assume that the frame length F = 3. Now, if movie needs to be scheduled at node 0, data blocks need to be communicated from nodes 2, 1, 0 and 3 to node 0 in different slots. This is shown in Fig. 10(b) where the movie is started in slot 0. Fig. 10(c) shows a complete schedule of 4 requests for movies E, C, B, and E that arrived in that order at nodes 0, 1, 2, 3 respectively. Each row in the schedule shows the blocks received by a node in different time slots. The entries in the table indicate the movie and the id of the sending node. Each column should not have a sending node listed more than once since that would constitute a conflict at the sender. A movie stream has its requests listed horizontally in a row. The blocks of a single stream are always separated by F slots, in this case F = 3. Node 0 schedules the movie to start in time slot 0. But node 1 cannot start its movie stream in slot 0 as it conflicts with node 0 for requesting a block from the same storage node 2. Node 2 can also schedule its movie in slot 1. Node 3 can only schedule its movie in slot 2. Each request is scheduled in the earliest available slot. The movie stream can be started in any column in the table as long as its blocks do not conflict with the already scheduled blocks. The schedule table is wrapped around i.e., Slot 0 is the slot immediately after Slot 11. For example, if another request arrives for movie E at node 2, we can start that request in time Slot 3, and schedule the requests in a wrap-around fashion in time Slots 6, 9, and 0 without any conflict at the source and the destination. The schedule table has FN slots, where N is the number of storage nodes in the system.

When the system is running to its capacity, each column would have an entry for each storage node. The schedule in slot j can be represented by a set (n_{ij}, s_{ij}), a set of network node and storage node pairs involved in a block transfer in slot j. If we specify F such sets for the F slots in a frame (j = 1,2,...F), we would completely specify the schedule. If a movie stream is scheduled in slot j in a frame, then it is necessary to schedule the next block of that movie in slot j of the next frame (or in $(j + F)$ mod FN slot) as well. Once the movie distribution is given, the schedule of transfer (n_{ij}, s_{ij}) in slot j of one frame automatically determines the pair (n_{ij}, s_{ij}) in the next frame, $s_{i(j+F)mod\ FN}$ being the storage node storing the next block of this movie and $n_{i(j+F)mod\ FN}$ $= n_{ij}$. Hence, given a starting entry in the table (row, column specified), we can

10(a). Movie distribution.

Movie/Blocks	0	1	2	3
A	0	1	2	3
B	1	3	0	2
C	2	0	3	1
D	3	2	1	0
E	2	1	0	3

10 (b). Schedule for movie E.

Slot 0	1	2	3	4	5	6	7	8	9	10	11
E.2			E.1			E.0			E.3		

10(c). Complete schedule

Req	0	1	2	3	4	5	6	7	8	9	10	11
0	E.2			E.1			E.0			E.3		
1		C.2			C.0			C.3			C.1	
2		B.1			B.3			B.0			B.2	
3			E.2			E.1			E.0			E.3

Figure 10 An example movie schedule.

immediately tell what other entries are needed in the table. It is observed that the F slots in a frame are not necessarily correlated to each other. However, there is a strong correlation between two successive frames of the schedule and this correlation is determined by the data distibution. It is also observed that the length of the table (FN) is equal to the number of streams that the whole system can support.

Now, the problem can be broken up into two pieces: (a) Can we find a data distribution that, given an assignment of (n_{ij}, s_{ij}) that is source and destination conflict-free, can produce a source and destination conflict-free schedule in the same slot j of the next frame? and (b) Can we find a data distribution that, given an assignment of (n_{ij}, s_{ij}) that is source, destination and network conflict-free, produce a source, destination and network conflict-free schedule in the same slot j of the next frame? The second part of the problem, (b), depends on the network of the multiprocessor and that is the only reason for addressing the problem in two stages. We will propose a general solution that addresses (a). We then tailor this solution to suit the multiprocessor network to address the problem (b).

4.1 Proposed solution

Part (a)

Assume that all the movies are striped among the storage nodes starting at node 0 in the same pattern i.e., block i of each movie is stored on a storage node given by $i \ mod \ N$, N being the number of nodes in the system. Then, a movie stream accesses storage nodes in a sequence once it is started at node 0. If we can start the movie stream, it implies that the source and the destination do not collide in that time slot. Since all the streams follow the same sequence of source nodes, when it is time to schedule the next block of a stream, all the streams scheduled in the current slot would request a block from the next storage node in the sequence and hence would not have any conflicts. In our notation, a set (n_{ij}, s_{ij}) in slot j of a frame is followed by a set $(n_{ij}, (s_{ij} + 1) \ mod \ N)$ in the same slot j of the next frame. It is clear that if (n_{ij}, s_{ij}) is source and destination conflict-free, $(n_{ij}, (s_{ij} + 1) \ mod \ N)$ is also source and destination conflict-free.

This simple approach makes movie distribution and scheduling stright-forward. However, it does not address the communication scheduling problem. Also, it has the following drawbacks: (i) not more than one movie can be started in any given slot. Since every movie stream has to start at storage node 0, node 0 becomes a serial bottleneck for starting movies. (ii) when short movie clips are played along with long movies, short clips increase the load on the first few nodes in the storage node sequence resulting in non-uniform loads on the storage nodes. (iii) as a results of (a), the latency for starting a movie may be high if the request arrives at node 0 just before a long sequence of scheduled busy slots.

The proposed solution addresses all the above issues (i), (ii) and (iii) and the communication scheduling problem. The proposed solution uses one sequence of storage nodes for storing all the movies. But, it does not stipulate that every movie start at node 0. We allow movies to be distributed across the storage nodes in the same sequence, but with different starting points. For example movie 0 can be distributed in the sequence of 0, 1, 2, ..., N-1, movie 1 can be distributed in the sequence of 1, 2, 3, ..., N-1, 0 and movie k (mod N) can be distributed in the sequence of k, k+1, ..., N-1, 0, ..., k-1. We can choose any such sequence of storage nodes, with different movies having different starting points in this sequence.

When movies are distributed this way, we achieve the following benefits: (i) multiple movies can be started in a given slot. Since different movies have different starting nodes, two movie streams can be scheduled to start at their starting nodes in the same slot. We no longer have the serial bottleneck at the starting node (we actually do, but for 1/Nth of the content on the server). (ii) Since different movies have different starting nodes, even when the system has short movie clips, all the nodes are likely to see similar workload and hence the system is likely to be better load-balanced. (iii) Since different movies have different starting nodes, the latency for starting a movie is likely to be lower since the requests are likely to spread out more evenly.

The benefits of the above approach can be realized on any network. Again, if the set (n_{ij}, s_{ij}) is source and destination conflict-free in slot j of a frame, then the set $(n_{ij}, (s_{ij}+1) \; mod \; N)$ is given to be source and destination conflict-free in slot j of the next frame, whether or not all the movies start at node 0. As mentioned earlier, it is possible to find many such distributions. In the next section, it will be shown that we can pick a sequence that also solves problem (b), i.e., guarantees freedom from conflicts in the network.

Part (b)

The issues addressed in this section are specific to the network of the system. We will use IBM's SP2 multiprocessor with an Omega interconnection network as an example multiprocessor. The solution described is directly applicable to hypercube networks as well. The same technique can be employed to find suitable solution for other networks. We will show that the movie distribution sequence can be carefully chosen to avoid communication conflicts in the multiprocessor network. The approach is to choose an appropriate sequence of storage nodes such that if movie streams can be scheduled in slot j of a frame without communication conflicts, then the consecutive blocks of those streams can be scheduled in slot j of the next frame without communication conflicts.

With our notation, the problem is to determine a sequence of storage nodes $s_0, s_1, ..., s_{N-1}$ such that given a set of nodes (n_{ij}, s_{ij}) that are source, destination and network conflict-free, it is automatically guaranteed that the set of nodes $(n_{ij}, s_{((i+1)\ mod\ N)j})$ are also automatically source, destination and network conflict-free.

First, let us review the Omega network. Fig. 11. shows a multiprocessor system with 16 nodes which are interconnected by an Omega network constructed out of 4x4 switches. To route a message from a source node whose address is given by $p_0p_1p_2p_3$ to a destination node whose address is given by $q_0q_1q_2q_3$, the following procedure is employed: (a) shift the source address left circular by two bits (log of the switch size) to produce $p_2p_3p_0p_1$, (b) use the switch in that stage to replace p_0p_1 with q_0q_1 and (c) repeat the above two steps for the next two bits of the address. In general, steps (a) and (b) are repeated as the number of stages in the network. Network conflicts arise in step (b) of the above procedure when messages from two sources need to be switched to the same output of a switch.

Now, let's address our problem of guaranteeing freedom from network conflicts for a set $(n_{ij}, s_{(i+1)\ mod\ N\ j})$ given that the set (n_{ij}, s_{ij}) is conflict-free. Our result is based on the following theorem of Omega networks.

Theorem: If a set of nodes (n_i, s_i) is network conflict-free, then the set of nodes $(n_i, (s_i + a) mod N)$ is network conflict-free, for any a.

Proof: Refer to [17].

The above theorem states that given a network conflict-free schedule of communication, then a uniform shift of the source nodes yields a network conflict-free schedule.

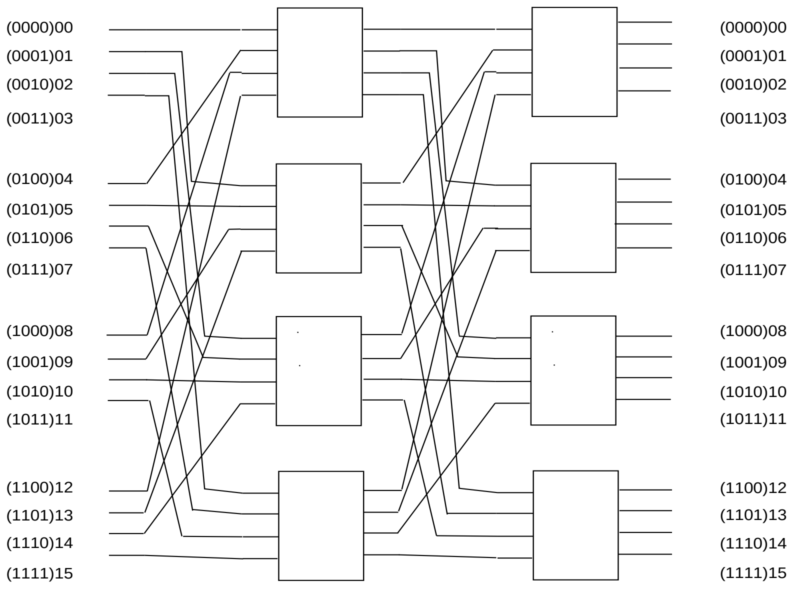

Figure 11 A 16-node Omega network used in IBM's SP2 multiprocessor.

There are several possibilities for choosing a storage sequence that guarantees the above property. A sequence of 0, 1, 2, ..., N-1 is one of the valid sequences - a simple solution indeed! Let's look at an example. The set S_1 = (0,0), (1,1), (2,2), ..., (14,14), (15,15) of network-storage nodes is conflict free over the network (identity mapping). From the above theorem, the set S_2 = (0,1), (1,2), (2,3), ..., (14,15), (15,0) is also conflict-free and can be so verified. If S_1 is the conflict-free schedule in a slot j, S_2 will be the schedule in slot j of the next frame, which is also conflict-free.

We have shown in this section a simple round-robin distribution of movie blocks in the sequence of 0, 1, 2, ..., N-1 yields an effective solution for our problem. This data distribution with different starting points for different movies solves (a) the movie scheduling problem, (b) the load balancing problem, (c) the problem of long latencies for starting a movie, and (d) the communication scheduling problem.

Now, the only question that remains to be addressed is how do we schedule the movie stream in the first place, i.e., in which slot should a movie be started. When the request arrives at a node n_i, we first determine its starting node s_0 based on the movie distribution. We look at each available slot j (where n_i is free and s_0 is free) to see if the set of already scheduled movies do not conflict for communication with this pair. We search until we find such a slot and schedule the movie in that slot. Then, the complete length of that movie is scheduled without any conflicts.

4.2 Other issues

Choosing a slot size

Ideally, we would like all block transfers to complete within a slot. However, due to variations in delivery time (due to variations in load and contention in the network), all the block transfers may not finish in the slot they are initiated. One option is to choose the slot to be large enough that it accommodates the maximum delivery time for a block. This approach, however, may not use the network as effectively since it allocates larger amount of time than the average delivery time for a block. If the slot is chosen to be the average delivery time, how do we deal with the transfers that take larger than average delivery delays?

Fig. 12. shows some results from simulation experiments on a 256-node 4-dimensional torus network with 100 MB/s link transfer speeds. These results

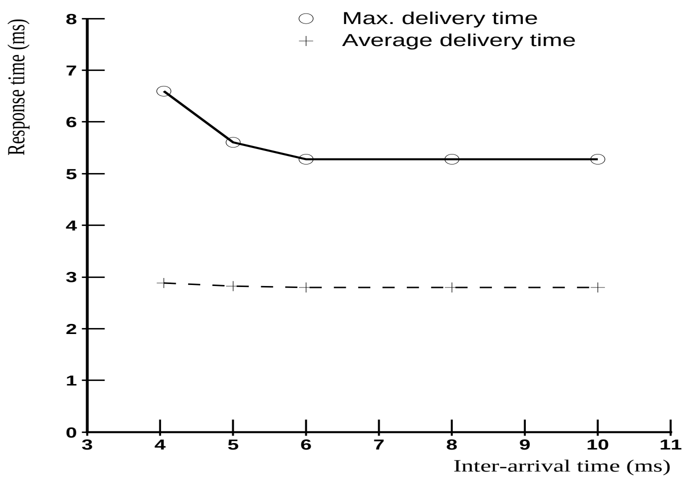

Figure 12 Observed delays in a 4-dim. 256-node system.

are only being presented as an example and similar results have to be obtained for the network under consideration. In the simulations, block arrival rates are varied until the deadlines for those block transfers could be met by the network. The figure shows the average time taken for message delivery and the maximum block delivery time at different request arrival times. It is observed that the average message delivery time is nearly constant and varies from 2.8 ms to 2.89 ms over the considered range of arrival times. However, the maximum delay observed by a block transfer goes up from 5.3 ms to 6.6 ms. Even though the average message completion time didn't vary significantly over the considered range of arrival rates, the maximum delays are observed to have a higher variation. If we were to look at only the average block transfer times, we might have concluded that it is possible to push the system throughput further since the request inter-arrival time of 4 ms is still larger than the average block transfer delay of 2.89 ms. If we were to look at only the maximum block transfer times, we would have concluded that we could not reduce the inter-arrival times to below 6 ms. However, the real objective of not missing any deadlines forced us to choose a different peak operating point of 4 ms of inter-arrival time (slot width).

It is clear from the above description that we need to carry out some experiments in choosing the optimal slot size. Both the average and the maximum delays in transferring a block over the network need to be considered. As mentioned earlier, the slot size is then adjusted such that a frame is an integer multiple of the width of the slot. Since the block transfers are carefully scheduled to avoid conflicts, it is expected that the variations in communication times will be lower in our system.

Different stream rates

When the stream rate is different from the basic stream rate, multiple slots are assigned within a frame to that stream to achieve the required stream rate. For example, for realizing a 3Mbits/sec stream rate, 2 slots are assigned to the same stream within a frame. These two slots are scheduled as if they are two independent streams. When the required stream rate is not a multiple of the basic stream rate, a similar method can be utilized with the last slot of that stream not necessarily transferring a complete block.

Reducing the stream startup latency

It is possible that when a stream A is requested, the next slot where this stream could be started is far away in time resulting in a large startup latency. In such cases, if the resulting latency is beyond certain threshold, an already scheduled stream B may be moved around within a frame to reduce the requested stream's latency. If stream B is originally scheduled at time T, then stream B can be moved to any free slot within $T + F - 1$ while maintaining guarantees on its deadlines. Fig. 13 shows the impact of such a strategy on the distribution of startup latencies.

When network nodes and storage nodes are different

It is possible to find mappings of network nodes and storage nodes to the multiprocessor nodes that guarantee freedom from network conflicts. For example, assigning the network nodes the even addresses and the storage nodes the odd addresses in the network, and distributing the movies in round-robin fashion among the storage nodes yields similar guarantees in an Omega network.

Node failures

Before, we can deal with the subject of scheduling after a failure, we need to talk about how the data on the failed data is duplicated elsewhere in the system.

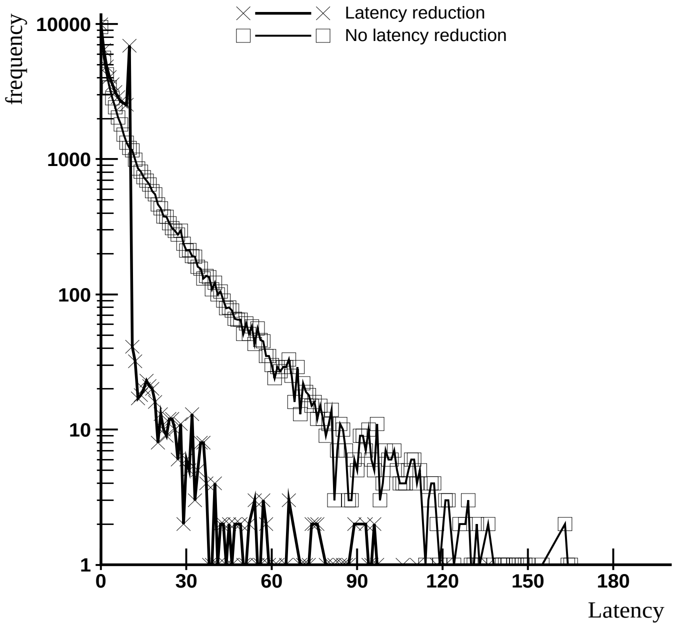

Figure 13 An example of the effectiveness of latency reduction techniques.

There are several ways of handling data protection, RAID, and mirroring being two examples. RAID increases the load on the surviving disks by 100% and this will not be acceptable in a system that has to meet real-time guarantees unless the storage system can operate well below its peak operating point. Mirroring may be preferred because the required bandwidths from the data stored in the system are high enough that the entire storage capacity of a disk drive may not be utilized. The un-utilized capacity can be used for storing a second copy of the data. We will assume that the storage system does mirroring. We will also assume that the mirrored data of a storage node is evenly spread among some set of K, $K < N$, storage nodes.

Let the data on the failed node f_0 be mapped to nodes $m_0, m_1, ..., m_{K-1}$. Before the failure, a stream may request blocks from nodes $0, 1, 2, ..., f_0, ...N-1$ in a round-robin fashion. The mirrored data of a movie is distributed among $m_0, m_1, ..., m_{K-1}$ such that the same stream would request blocks in the following order after a failure: $0, 1, 2, ..., m_0, ..., N-1, 0, 1, 2, ..., m_1, ..., N-1, ..., 0, 1, 2, ..., m_{K-1}, ..., N-1, 0, 1, 2, ..., m_0, ..., N-1$. The blocks that would have been requested from the failed node are requested from the set of mirror nodes of that failed node in a round-robin fashion. With this model, a failure increases the load on the mirrored set of nodes by a factor of (1+1/K) since for every request to the failed node, a node in the set of mirrored nodes observes 1/K requests. This implies that K should be as large as possible to limit the load increases on the mirror nodes.

Scheduling is handled in the following way after a failure. In the schedule table, we allow l slots to be free. When the system has no failures, the system is essentially idle during these l slots. After a failure, we will use these slots to schedule the communication of movie blocks that would have been served by the failed node. A data transfer (n_i, f_0) between a failed node f_0 and a network node n_i is replaced by another transfer of (n_i, m_i) where m_i is the storage node that has the mirror copy of the block that should have been transfered in (n_i, f_0). If we can pack all the scheduled communication with the mirror nodes into the available free slots, with some appropriate buffer management, then we can serve all the streams that we could serve before the failure. Now, let's examine the conditions that will enable us to do this.

Given that the data on the failed node is now supported by K other nodes, the total number of blocks that can be communicated in l slots is given by $K * l$. The failed node could have been busy during $(FN - l)$ slots before the failure. This implies that $Kl \geq FN - l$, or $l \geq FN/(K+1)$ - (1).

It is noted that no network node n_i can require communication from the failed node f_0 in more than $(FN - l)/N$ slots. Under the assumptions of system wide striping, once a stream requests a block from a storage node, it does not request another block from the same storage node for another $N - 1$ frames. Since each network node can support at most $(FN - l)/N$ streams before the failure, no network node requires communication from the failed node f_0 in more than $(FN - l)/N$ slots. Since every node is free during the l free slots, the network nodes require that $l \geq (FN - l)/N$, or $l \geq FN/(N + 1)$ - (2). The above condition (1) is more stringent than (2).

Ideally, we would like $K = N - 1$ since this minimizes the load increase on the mirror nodes. Also, we would like to choose the mirror data distribution such that if block transfer from the mirrored nodes is guaranteed to be conflict-free during a free slot j, then it will also be conflict-free in the slot $j + FN$ (the same free slot in the next schedule table) when the transfers would require data from the next node in the mirror set. In our notation if the set (n_i, m_i) is conflict-free in a free slot j, then we would like the set $(n_i, m_{(i+1)modK})$ to be conflict-free in slot $j + NF$.

Schedule of block transfers during the free slots is explained below. A maximal number of block transfers are found that do not have conflicts in the network. This set is assigned one of the free slots. With the remaining set of required block transfers, the above procedure is repeated until all the communication is scheduled. This algorithm is akin to the problem of finding a minimal set of matchings of a graph such that the union of these matchings yields the graph.

We can show an upper bound on the number of free slots required. We can show that at least 4 blocks can always be transferred without network conflicts as long as the source and destinations have no conflicts, when the Omega network is built out of 4x4 switches. If a set of four destinations are chosen such that they differ in the most significant 2 bits of the address, it can be shown that as long as the source and destinations are different, the block transfers do not collide in the network. The proof is based on the procedure for switching a block from a source to a destination and if the destinations are so chosen it can be shown that these four transfers use different links in the network. Since at most $FN - l$ blocks need to be transferred during the free slots, $l \leq (FN - l)/4$. This gives $l \leq FN/5$. This implies that if the network nodes requiring communication from the failed node are equally distributed over all the nodes in the system, we can survive a storage node failure with about 20% overhead.

Network node failures can be handled in the following way. The movie streams at the failed node are rerouted (redistributed) evenly to the other network nodes in the system. This assumes that the delivery site can be reached through any one of the network nodes. The redistributed streams are scheduled as if the requests for these streams (with a starting point somewhere over the length of the movie, not necessarily at the beginning) are new requests.

If a combo node fails, both the above procedures for handling the failure of a storage node and a network node need to be invoked.

Clock Synchronization

Throughout this section, it is assumed that the clocks of all the nodes in the system are somehow synchronized and that the block transfers can be started at the slot boundaries. If the link speeds are 40MB/sec, a block transfer of 256 Kbytes requires 6.4 ms, quite a large period of time compared to the precision of the node clocks which tick every few nanoseconds. If the clocks are synchronized to drift at most, say 600 us, the nodes observe the slot boundaries within $\pm 10\%$. During this time, it is possible that the block transfers observed collisions in the network. But during the rest of the 90% transfer time, the block transfers take place without any contention over the network. This shows that the clock synchronization requirements are not very strict. It is possible to synchronize clocks to such a coarse level by broadcasting a small packet of data at regular intervals to all the nodes through the switch network.

Other Interconnection Networks

The proposed solution may be employed even when the multiprocessor system is interconnected by a network other than an omega network. To guarantee conflict-free transfers over the network, appropriate data distributions for those networks have to be designed. For hypercube type of networks that can emulate an omega network, same data distribution provides similar guarantees as in Omega network. It can be shown that if movie blocks are distributed uniformly over all nodes in a hypercube in the same order $0, 1, 2, ..., n-1$ (with different starting nodes), a conflict free schedule in one slot guarantees that the set of transfers required a frame later would also be conflict free.

For other lower degree networks such as a mesh or a two dimensional torus, it can be shown that similar guarantees cannot be provided. For example, in a two dimensional nxn torus, the average path length of a message is 2* n/4 = n/2. Given that the system has a total of $4 * n^2$ unidirectional links, the average number of transmissions that can be in progress simultaneously is given by $4 * n^2/(n/2) = 8 * n$, which is less than the number of nodes n^2 in the system for $n > 8$. However, n simultaneous transfers are possible in a 2-dimensional torus when each node sends a message to a node along a ring. If this is a starting position of data transfer in one slot, data transfer in the following frames cannot be sustained because of the above limitation on the average number of simultaneous transfers through the network. In such networks, it may be advantageous to limit the data distribution to a part of the system so as to limit the average path length of a transfer and thus increasing the number of sustainable simultaneous transfers.

Incremental growth

How does the system organization change if we need to add more disks for putting more movies in the system? In our system, all the disks are filled nearly to the same capacity since each movie gets distributed across all the nodes. If more disk capacity is required, we would require that at least one disk be added at each of the nodes. If the system has N nodes, this would require N disks. The newly added disks can be used as a set to distribute movies across all the nodes to obtain similar guarantees for the new movies distributed across these nodes. If the system size N is large, this may pose a problem. In such a case, it is possible to organize the system such that movies are distributed across a smaller set of nodes. For example, the movies can be distributed across the two sets 0, 2, 4, 6 and 1, 3, 5, 7 in an 8-node machine to provide similar guarantees as when the movies are distributed across all the 8 nodes in the system. (This result is again a direct consequence of the above Theorem 1.) In this example, we only need to add 4 new disks for expansion as opposed to adding 8 disks at once. This idea can be generalized to provide a unit of expansion of K disks in an N node system, where K is a factor of N.

This shows that the width of striping has an impact on the system's incremental expansion. The wider the movies are striped across the nodes of the system, the larger the bandwidth to a single movie but also the larger the unit of incremental growth.

5 GENERAL DISCUSSION

5.1 Admission Control

Admission control is used to make sure that the system is not forced to operate at such a point that it cannot guarantee service to the scheduled streams. Requests are allowed only until a point that the scheduled streams can be guaranteed to meet their deadlines. Admission control policy can be based on analysis or through simulations. Each component of the service can be analyzed and the interaction of these components on the total service can be studied. Analysis presented in section 3.5 can be used for the disk service component. The communication component also has to be analyzed similarly.

Alternately, we could determine the maximum number of streams that can be supported by the system thorough simulations. After determining the capacity of the system, we could rate the usable capacity of the system to be a fraction of that to ensure that we don't miss too many deadlines. In a real system, a number of other factors such as the CPU utilization, the multiprocessor network utilization have to be considered as well for determining the capacity of the system. Analyzing all these factors may become cumbersome and may make simulations the only available method for determining the capacity of the system.

5.2 Future work

A number of problems in the design of a video-on-demand server require further study.

We presented a preliminary study of tolerating disk failures in this chapter. More work needs to be done in this area. If it is not possible to guarantee precise scheduling in the presence of failures, alternative scheduling strategies during normal operation may be attractive.

When the system is expanded, the newly added disks may have different performance characterisitcs than the already installed disks. How do we handle the different performance charateristics of different disks?

Providing fast-forward and rewind operations has not been discussed in this chapter. Depending on the implementation, these operations may result in varying demands on the system. It is possible to store a second version of the movie sampled at a higher (fast-forward) rate and then compressed on the disk for handling these operations. Then, fast-forward and rewind operations will not cause any extra demands on the system resources but will introduce the problems of scheduling the proper version of the movie at the right time. These strategies remain to be evaluated.

Acknowledgements

The work reported here has benefited significantly from discussions and interactions with Jim Wyllie and Roger Haskin of IBM Almaden Research Center.

REFERENCES

[1] R. Haskin. The shark continuous-media file server. *Proc. of IEEE COMPCON*, Feb. 1993.

[2] F. A. Tobagi, J. Pang, R. Biard, and M. Gang. Streaming raid: A disk storage system for video and audio files. *Proc. of ACM Multimedia Conf.*, pages 393–400, Aug. 1993.

[3] D. Anderson, Y. Osawa, and R. Govindan. A file system for continuous media. *ACM Trans. on Comp. Systems*, pages 311–337, Nov. 1992.

[4] H. M. Vin and P. V. Rangan. Designing file systems for digital video and audio. *Proc. of 13th ACM Symp. on Oper. Sys. Principles*, 1991.

[5] A. Chervenak. Tertiary storage: an evaluation of new applications. *Ph.D Thesis, Univ. of Calif., Berkeley*, 1994.

[6] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of ACM*, pages 46–61, 1973.

[7] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. *Proc. of Real-time Systems Symp.*, pages 129–139, Dec. 1991.

[8] A. L. Narasimha Reddy and Jim Wyllie. Disk scheduling in a multimedia I/O system. *Proc. of ACM Multimedia Conf.*, Aug. 1992.

[9] H. M. Deitel. An introduction to operatins systems. *Addision Wesley*, 1984.

[10] T. H. Lin and W. Tarng. Scheduling periodic and aperiodic tasks in hard real-time computing systems. *Proc. of SIGMETRICS*, pages 31–38, May 1991.

[11] J. Yee and P. Varaiya. Disk scheduling policies for real-time multimedia applications. *Tech. report, Univ. of California, Bekeley*, Aug. 1992.

[12] D. P. Anderson, Y. Osawa, and R. Govindan. Real-time disk storage and retrieval of digital audio/video data. *Tech. report UCB/CSD 91/646, Univ. of Cal., Berkeley*, Aug. 1991.

[13] P. S. Yu, M. S. Chen, and D. D. Kandlur. Grouped sweeping scheduling for dasd-based multimedia storage management. *Multimedia Systems*, 1:99–109, 1993.

[14] J.P. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. *Proc. of Real-time Systems Symp.*, pages 201–212, Dec. 1990.

[15] W. K. Shih, J. W. Liu, and C. L. Liu. Modified rate monotone algorithm for scheduling periodic jobs with deferred deadlines. *Tech. Report, Univ. of Illinois, Urbana-Champaign*, Sept. 1992.

[16] A. L. Narasimha Reddy. A study of I/O system organizations. *Proc. of Int. Symp. on Comp. Arch.*, May 1992.

[17] Duncan H. Lawrie. Access and alignment of data in an array processor. *IEEE Trans. Comput.*, C-24(12):1145–1155, Dec. 1975.