

Optimal Scheduling for Parallel CBR Video Servers

MIN-YOU WU

wu@ece.engr.ucf.edu

WEI SHU

shu@ece.engr.ucf.edu

Department of Electrical and Computer Engineering, University of Central Florida, Orlando, FL 32816

Abstract. Parallel video servers are necessary for large-scale video-on-demand and other multimedia systems. This paper addresses the scheduling problem of parallel video servers. We discuss scheduling requirements of constant bit rate (CBR) video streams. Optimal algorithms are presented for conflict-free scheduling, delay minimization, request relocation, and admission control. With these algorithms, video streams can be precisely scheduled for Quality of Service requirements. Performance of these algorithms is also presented.

Keywords: large-scale VoD, video servers, conflict-free scheduling, constant bit rate (CBR) video streams, Quality of Service (QoS).

1. Introduction

There is an increasing demand on capacity of video servers in large-scale video-on-demand systems [9, 3]. Parallel video servers become inevitable to provide large capacity required by the system that can service thousands of concurrent clients.

There are two major types of parallel video servers: shared memory multiprocessors and distributed memory clustered architectures. Figure 1 shows the diagram of shared-memory architecture. In this architecture, a set of storage nodes and a set of computing nodes are connected to 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 [13]. 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. Figure 2 shows the diagram of the clustered architecture. A number of works describe clustered systems [18, 14, 15]. The clustered architecture can be extended to the direct-access architecture which provides an interface between the storage system and the network [4, 7]. Figure 3 shows the diagram of direct-access architecture. It eliminates the delivery nodes in the clustered architecture. Storage nodes send video data directed to the high speed WAN through a network interface. If a video file is not stored in a single storage node this architecture cannot guarantee the order of data arrival from different storage nodes. This problem is solved in project MARS by using a chain connection, an ATM-based interconnection within the server to connect storage devices to an ATM-based broadband network [4].

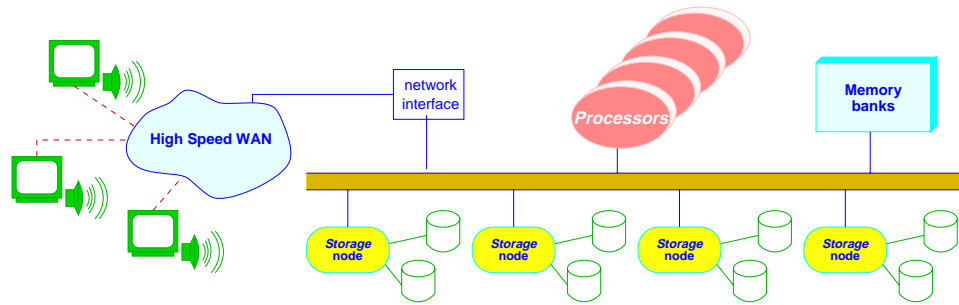


Figure 1. Shared-memory Architecture for Video Servers.

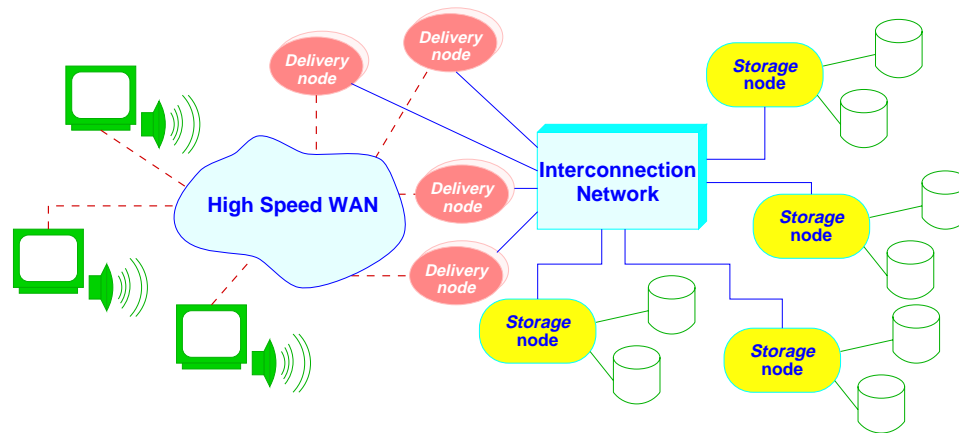


Figure 2. Clustered Architecture for Video Servers.

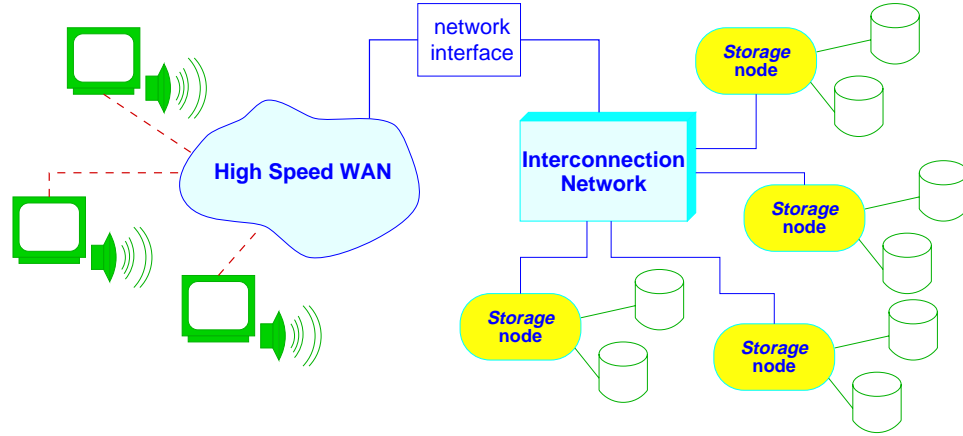


Figure 3. Direct-access Architecture for Video Servers.

In this paper, we evaluate scheduling strategies for the clustered architecture. As shown in Figure 2, an interconnection network connects *storage nodes* and *delivery nodes*. The storage nodes 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 the specified bandwidth. The delivery nodes are responsible for client's requests. On receiving a request from a client, the 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. Each delivery node has an output buffer as the interface to the high-speed WAN. If the WAN has congestion and the output buffer becomes full, a feedback signal is sent to corresponding storage nodes to stop sending video blocks. 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 [18]. Also, a node can be both a storage node and a delivery node, called the “flat” architecture. In this paper, the flat architecture is assumed.

Scheduling policies are critical to performance of video servers. Currently, there are not many scheduling algorithms for parallel video servers, in particular, for clustered video servers. Without a good scheduling algorithm, video streams may conflict each other, resulting in delays and rejections. A larger buffer size may be required to tolerate this delay. Good scheduling algorithms can guarantee the quality-of-service, efficiently utilize resources and reduce the buffer size, and increase system throughput.

This paper is organized as follows. Section 2 briefly describes an underlying model of a parallel CBR video server. A conflict-free scheduling algorithm and delay minimization are presented in sections 3 and 4. In section 5, a request relocation

algorithm is presented. Admission control is discussed in section 6. Performance comparison is presented in section 7. Section 8 gives related works and section 9 discusses future works.

2. System Architecture and Background

In this section, we model the scheduling problem for video servers. First, we assume *wide striping*, that is, video data is striped across all nodes in a round-robin fashion. Different video files start from different nodes for a balanced load. For example, the first file starts at node 0, the second file starts at node 1, etc. The results obtained here can be extended to *short striping*, which distributes video blocks to a subset of 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 time to access n blocks, one from each storage node, is called *time period*. Thus a time period includes n time cycles. 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 video streams 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 value of t_s can be determined by experiments. Similar to other Quality-of-Service (QoS) problems, there are two type of guarantees, the deterministic guarantee and statistical guarantee. Deterministic guarantee determines t_s by using the the worst case numbers. Such an approach is described in [15]. This approach guarantees QoS for all requests but the system may be underutilized. The statistical guarantee determines t_s by using the the average numbers over a period of time. It increases the system utilization but some data may miss its deadline and more buffer space is necessary to smooth the delay jitter. 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 . A similar model has been used in the Tiger system [7]. The experiment results showed that this model worked well on real systems.

Assume that a video 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 $i = D(r)$, where node i 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 request relocation is required. The blocks of a video file 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 $(S(r, t + \tau) \bmod N)$ at time cycle $(t + \tau)$.

Video block scheduling can be illustrated by a simple example. Figure 4 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

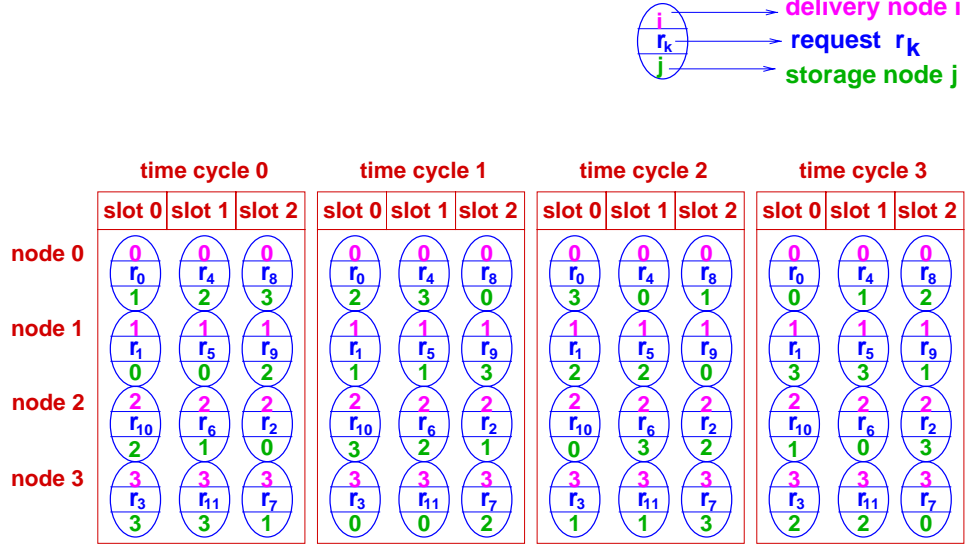


Figure 4. A Complete Video Schedule in a Time Period.

number r_k , the delivery node number i , and the storage node number j . 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. At most $(N \times m)$ requests can be scheduled.

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 retrieves a block from different storage nodes. Once the first cycle has a conflict-free schedule, the following cycles will not have conflict. Therefore, when discussing the scheduling problem, we can only show the first time cycle as long as the access patterns of video streams do not change.

The same model can be extended to scheduling of interactive operations, such as fast forward and rewind. The access patterns of interactive operations are irregular. A set of scheduling algorithms have been designed which can retain the access order of storage nodes [20].

Now, we define the *conflict-free schedule* as follows.

Definition 1: Conflict-free schedule.

A conflict-free schedule is a schedule where in each time slot, no two video streams request a block from the same storage node. \square

A conflict-free schedule imply two constraints. The first one is imposed by the disk retrieving bandwidth of storage nodes which is measured by the total number of requests retrieving blocks from storage node j , at time cycle t :

$$v_j^t = | \{ r \mid S(r, t) = j \} | .$$

Because no more than one request can access the same storage node in each time slot, no more than m requests can access the same storage node in a time cycle. The second one is limited by the I/O bandwidth of delivery nodes. Within a time slot, a delivery node cannot handle more than one stream from different storage nodes. The total number of requests handled by delivery node i

$$w_i = | \{ r \mid D(r) = i \} | .$$

is bounded by m . These requests may access arbitrary storage nodes which are not necessarily distinct. These two constraints are summarized in the following Lemma.

Lemma 1: The necessary condition for a conflict-free schedule in a time cycle is

- (1) $v_j^t \leq m$ for $j = 0, 1, \dots, N - 1$; and
- (2) $w_i \leq m$ for $i = 0, 1, \dots, N - 1$. \square

Now, the question is that whether the necessary condition is also the sufficient condition. In the other word, given a set of requests satisfying this necessary condition, does there exist a conflict-free schedule? In the next section, we will prove that Lemma 1 is also the sufficient condition for a conflict-free schedule by giving an algorithm to find out a conflict-free schedule.

3. Conflict-free Scheduling of Video Streams

We are going to generate a conflict-free schedule for a given set of requests such that $v_j^t = m$ and $w_i = m$ for every j and i . A greedy algorithm has been proposed in [15], which schedules requests in their arriving order. Whenever a request arrives, the next available slot that is not in conflict with the existing ones will be assigned to the request. An example illustrates this algorithm.

Example 1: Assume a set of requests arrive in the order as shown in Figure 5(a). The greedy algorithm schedules these requests as shown in Figures 5(b) to (d). Request r_1 cannot be scheduled to slot 0, so it is scheduled to slot 1; same is r_3 . Requests r_4 and r_7 must be scheduled to slot 2. Requests r_8, r_{10} , and r_{11} are not able to be scheduled. Thus, only nine requests have been scheduled. \square

The greedy algorithm cannot schedule some requests even if the Lemma 1 is satisfied. By rearranging the allocation of some existing requests, more requests can be accommodated. Let's revisit Example 1. In Figure 5(c), when request r_8 arrives, it cannot be scheduled to the only empty slot, slot 1, due to its conflict with r_3 . However, if we exchange r_3 and r_7 , r_8 can be scheduled to slot 1 as shown in Figure 6(a). Then r_9 is scheduled to slot 2 as shown in Figure 6(b). When

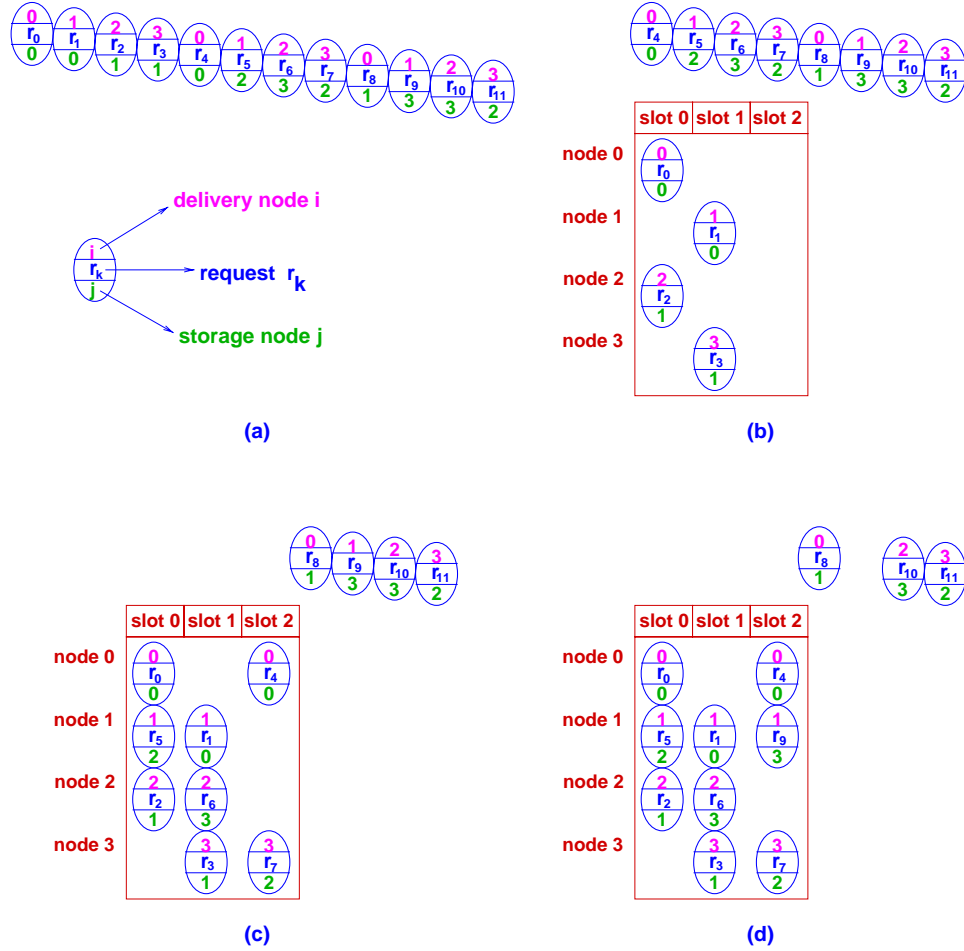


Figure 5. Example of Greedy Scheduling Algorithm.

request r_{10} arrives, it cannot be scheduled to slot 2 since it conflicts with r_9 . The table is rearranged by moving r_2 from slot 0 to slot 2 and r_3 from slot 2 to slot 0. Thus, r_{10} can be scheduled to slot 0 as shown in Figure 6(c). Finally, r_{11} is scheduled to slot 2 as shown in Figure 6(d). This example shows that a better schedule can be achieved by rearranging requests. We need a systematic approach to schedule requests without conflict. Now, we present an optimal algorithm for the conflict-free scheduling problem. This algorithm converts the scheduling problem to a matching problem on bipartite graphs. Then, the perfect matching algorithm can be applied to solve the problem.

Algorithm 1: Conflict-free scheduling.

Construct a bipartite graph G with bipartition (X, Y) , where $X = x_0, x_1, \dots, x_{N-1}$, $Y = y_0, y_1, \dots, y_{N-1}$, and x_i is joined to y_j if and only if delivery node i handles a request that retrieves a block from storage node j . Because each delivery node handles m requests and there are m requests retrieve blocks from storage node j , the constructed graph is an m -regular bipartite graph. For the first time slot, whether there exists a set of requests each of which accesses a different storage node from a different delivery node is equivalent to find a perfect matching in G . According to the marriage theorem [12], if G is a k -regular bipartite graph with $k > 0$, then G has a perfect matching. After determining a perfect matching for time slot 0, eliminate the matched edges, the original problem of scheduling Nm requests to m time slots is reduced to a problem of scheduling $N(m-1)$ requests to $(m-1)$ time slots. Thus, applying the perfect matching algorithm m times, a schedule for all time slots can be generated. \square

The perfect matching algorithm, so called the Hungarian method, can be found in [2]. Here we give a brief description of the algorithm. Start with an arbitrary matching M . If M is not a perfect matching yet, an M -unsaturated node u is chosen. We search for an M -augmenting path with origin u to construct a larger matching. This procedure is repeated until a perfect matching is found.

An example of the conflict-free algorithm is shown in Figure 7. It uses the same set of requests in Figure 5. After three iterations, the resultant schedule is shown in Figure 7(d) which is equivalent to Figure 6(d).

This algorithm can also be used for the situation that less than Nm requests exist. When $v_j^t < m$ or $w_i < m$, dummy requests can be inserted. From this algorithm, it is easy to see that Lemma 1 is also the sufficient condition of conflict-free scheduling. This algorithm should be called when a request arrives but there is no conflict-free slot. When multiple requests arrive simultaneously, these requests are scheduled to empty slots one by one. When a conflict occurs, this algorithm is called to scheduling all the requests instead of only a single request.

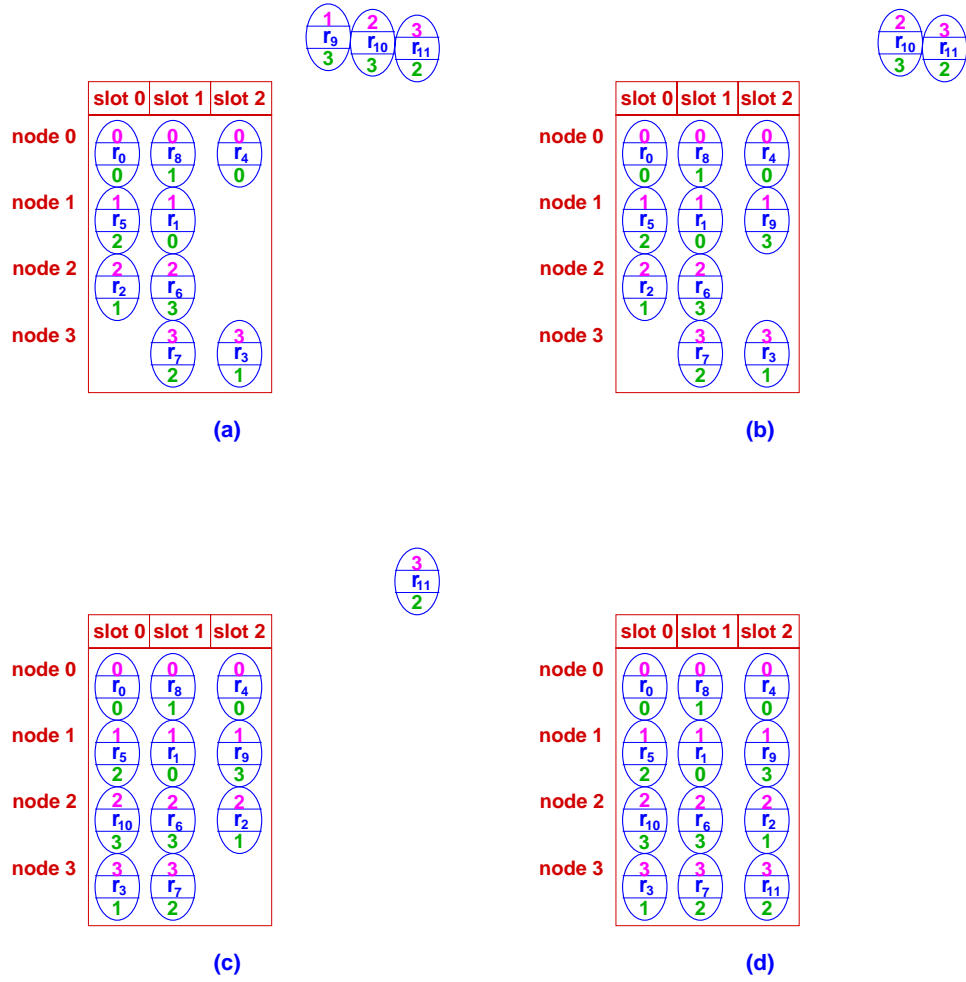


Figure 6. Improvement of Example 1.

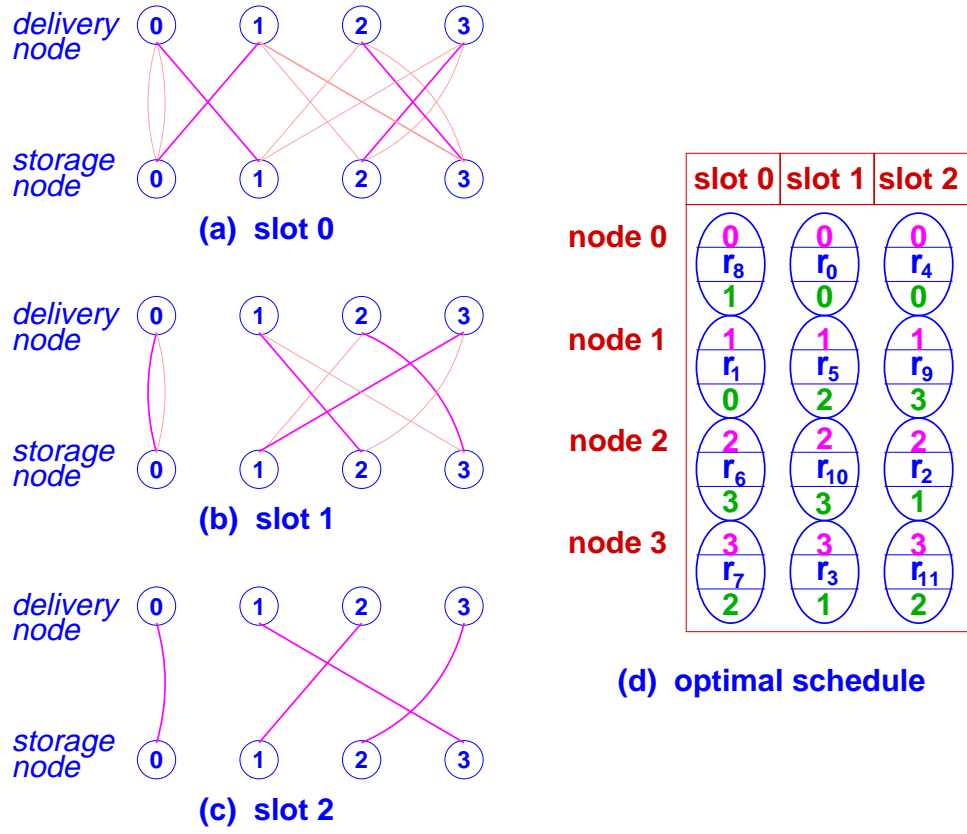


Figure 7. Bipartite Graph for Request Set in Example 1.

4. Request Delay Minimization

Now consider the situation that the total number of requests is less than or equal to Nm and $w_i \leq m$ for all i , but there exists some j such that $v_j^t > m$. In this case, only m requests can be fulfilled by storage node j , and $(v_j^t - m)$ requests must be delayed. Here, node j is called a “peg.” On the other hand, there can be other storage node j where $v_j^t < m$. That leaves $(m - v_j^t)$ empty spaces in node j which is called a “hole.” The “peg” requests can be fulfilled by delaying them to “holes”. In a large system, the delay can be significant. The problem is how to minimize the delay of these “peg” requests. This problem can be solved with a simple *peg-and-hole* algorithm which is presented as follows.

Algorithm 2: Peg-and-hole.

Let $R(j, k)$ hold the k th requests with $S(r, t) = j$, $v_j^t = |\{r \mid S(r, t) = j\}|$ and $\beta(r_k) = 0$ for all k

```

 $i_h = N - 1;$ 
for node  $i_p = N - 1$  to 0
  if ( $i_p == i_h$ )  $i_h = i_h - 1$ 
  if  $v_{i_p} > m$  for each of  $(v_{i_p} - m)$  extra requests
     $x = R(i_p, v_{i_p});$  remove request  $x$  from node  $i_p$ 
    while ( $v_{i_h} \geq m$ )  $i_h = (i_h - 1 + N) \bmod N;$  found a hole
     $v_{i_p} = v_{i_p} - 1;$ 
     $v_{i_h} = v_{i_h} + 1;$ 
     $R(i_h, v_{i_h}) = x;$  transfer it to node  $i_h$ 
     $\beta(x) = \beta(x) + ((i_p - i_h + N) \bmod N);$ 
     $S(x, t) = (S(x, t) - 1 + N) \bmod N;$ 

```

□

In this algorithm, each request r_i is represented by $\{S(r_i, t), \beta(r_i)\}$, where $\beta(r_i)$ stands for the number of cycles to be delayed on request r_i and initialized to zero. An entry of the two-dimension array $R(j, k)$ indicates the k th request on storage node j . Variables i_p and i_h are indices pointing to the “peg” and “hole” storage nodes, respectively. Basically, a request that cannot be fulfilled in the current time cycle is delayed to its nearest unpacked time slot. This algorithm guarantees the minimum delay.

Example 2: A set of requests are given as shown in Figure 8(a), where $v_0 = 1$, $v_1 = 5$, $v_2 = 4$, and $v_3 = 2$. Thus storage nodes 1 and 2 are pegs while storage nodes 0 and 3 are holes. Requests r_6 and r_{11} are delayed by two time cycles; request r_8 is delayed by one time cycle, as given in in Figure 8(b). The resultant schedule is shown in Figure 8(c). □

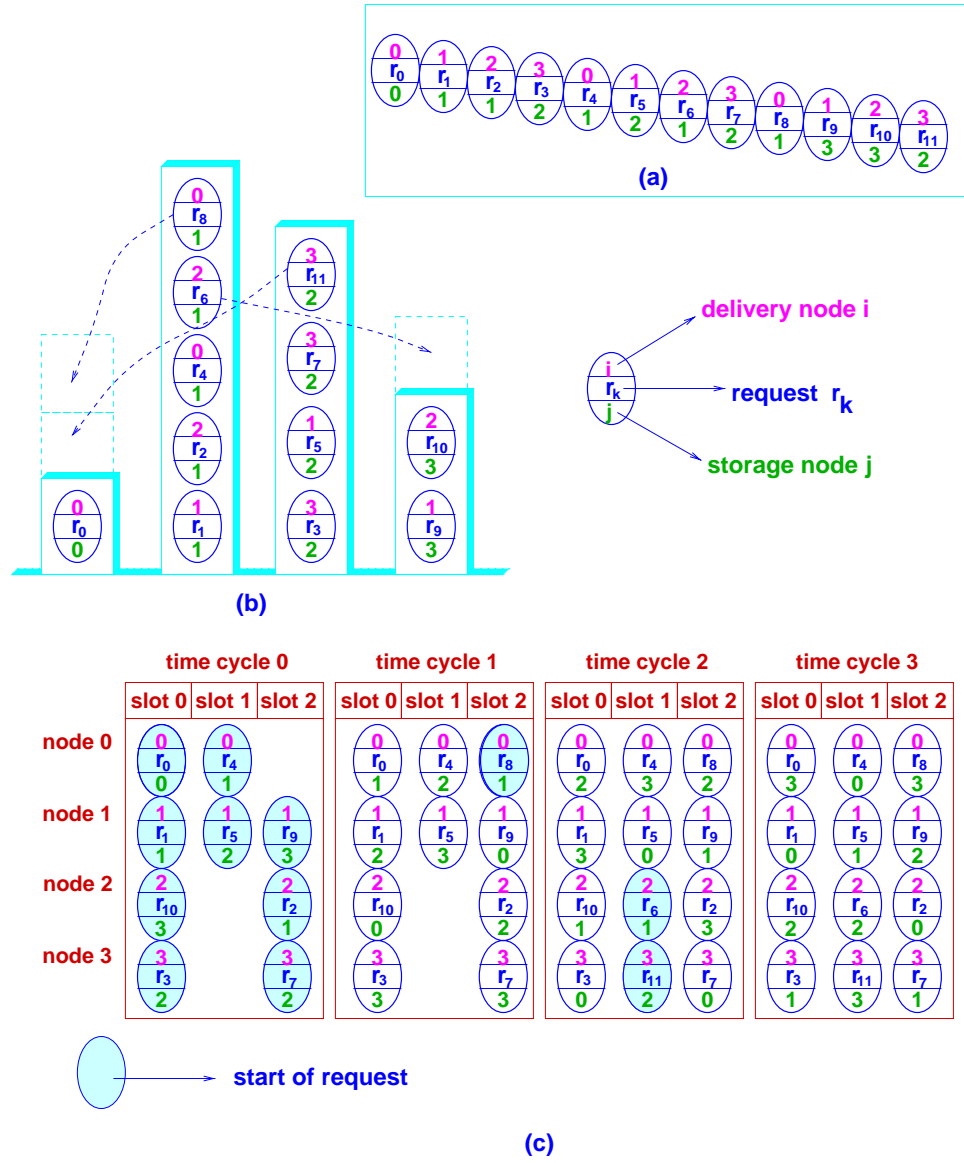


Figure 8. Example for Request Delay

5. Request Relocation

In this section, we consider the situation where the load is unbalanced. The definition of load balance is different from the traditional definition because the video stream scheduling is a real-time problem. In this system, as long as every $w_i \leq m$, it is considered as a balanced load. Otherwise, some requests need to be relocated. We assume that the total number of requests is less than or equal to Nm . In the case where the total number of requests is more than Nm , only Nm requests can be fulfilled. Many existing load relocation algorithms can be applied to this problem. The objective is to move requests from those overloaded nodes to the underloaded nodes with minimal cost. RIPS [17], runtime incremental parallel scheduling, is an algorithm designed to balance independent tasks on large-scale parallel machines. The algorithm can be so modified for parallel video servers that every node that presently handles more than m requests will send excess load to its nearest underloaded nodes. In Figure 9, we present a modified algorithm with assumption that the nodes are arranged in a linear chain. Algorithms for other topologies can be developed similarly.

The first step is to obtain the accumulated number of requests by using the *scan with sum* operation from node $N - 1$ to node 0. Each node records a partial sum $T_i = \sum_{j=i}^{N-1} w_j$. In the second step, each node calculates an accumulation quota Q_i . This algorithm can be seen as another “peg-and-hole” algorithm, where the “peg” is the overloaded node and the “hole” is the underloaded node with respect to its quota. The corresponding flows are also calculated. A positive flow means that a flow from node $i - 1$ to node i and a negative flow means that a flow from node i to node $i - 1$. In the third step, proximities are calculated which are the distance from nearest “hole” nodes. Assume every hole has unlimited capacity, the minimal flow $f_{i-1,i}$ is the result of delivering every extra work to its nearest hole, in case of a tie, to its left-nearest hole. Hence, there is no flow across any hole node. Therefore, between nodes $i - 1$ and i , the resultant minimal flow $f_{i-1,i}$ can be determined only by all nodes from its left-nearest hole to its right-nearest hole, which are bounded by indices from b_i^l to b_i^r . That is, $f_{i-1,i}$ is a summation of possible flows from its left-handed nodes if node b_i^r is their nearest hole and its right-handed nodes if node b_i^l is their nearest hole. In the fourth step, with two flows $y_{i-1,i}$ and $f_{i-1,i}$ available, we can construct an optimal flow $x_{i-1,i}$ such that it is closed to $f_{i-1,i}$ as much as possible, but does not overload any hole node. In the case of $T_0 = Q_0$, the flow $y_{i-1,i}$ is an optimal one. Otherwise, an auxiliary variable g_i is used to record how many streams can be adjusted. Initially, g_0 is the maximum number of streams allowed to be adjusted. Once g_i becomes zero, the rest of optimal flows, $f_{i,i+1}, \dots, f_{N-2,N-1}$ must be the same as $y_{i,i+1}, \dots, y_{N-2,N-1}$. According to the values of x , the workload is exchanged so that each node has no more than m requests. The number of communication steps in this algorithm is $5N$, where step 1 takes N steps to calculate T_i , step 3 takes $2N$ steps, one for calculating p , one for f , and step 4 takes $2N$ steps, one for calculating x , one for load exchange. This algorithm minimizes the number of communications and communication distance. It also maximizes locality.

Algorithm 3: Request Relocation.

Let w_i be the number of video streams in node i , where $i = 0, 1, \dots, N-1$.

1. **Global Reduction:** Perform the *scan* with *sum* operation of w_i :

$$T_i = \sum_{j=i}^{N-1} w_j$$

2. **Quota Calculation:** An accumulation quota and corresponding flow for each node are computed:

$$Q_i = (N - i)m$$

$$g_0 = Q_0 - T_0; \text{ for } i > 0, y_{i-1,i} = Q_i - T_i$$

3. **Proximity Calculation:**

For each node i , its proximity to the left/right-nearest hole is defined as p_i^l/p_i^r as follows:

$$p_i^l = \min(\{i - j \mid 0 \leq j \leq i \text{ and } w_j < m\} \cup \{\infty\})$$

$$p_i^r = \min(\{j - i \mid i \leq j < N \text{ and } w_j < m\} \cup \{\infty\})$$

For each node i , a segment from its left-nearest hole to its right-nearest hole are bounded by indices from b_i^l to b_i^r :

$$b_i^l = \max(\{j \mid 0 \leq j < i \text{ and } w_j < m\} \cup \{-1\})$$

$$b_i^r = \min(\{j \mid i < j < N \text{ and } w_j < m\} \cup \{N\})$$

Assume every hole has unlimited capacity, the minimal flow $f_{i-1,i}$ is the result of delivering every extra work to its nearest hole, and in case of a tie, to its left-nearest hole.

For $i = 1, \dots, N-1$:

$$f_{i-1,i} = \sum_{b_i^l < j < b_i^r} \begin{cases} m - w_j & \text{if } j \geq i \text{ and } p_j^l \leq p_j^r \\ w_j - m & \text{if } j < i \text{ and } p_j^l > p_j^r \\ 0 & \text{otherwise} \end{cases}$$

4. **Load Exchange:**

For $i = 1, \dots, N-1$:

$$g_i = \begin{cases} y_{i-1,i} - f_{i-1,i} & \text{if } g_{i-1} \geq (y_{i-1,i} - f_{i-1,i}) \geq 0 \\ 0 & \text{if } (y_{i-1,i} - f_{i-1,i}) < 0 \\ g_{i-1} & \text{if } (y_{i-1,i} - f_{i-1,i}) > g_{i-1} \end{cases}$$

$$x_i = \begin{cases} f_{i-1,i} & \text{if } g_{i-1} \geq (y_{i-1,i} - f_{i-1,i}) \geq 0 \\ y_{i-1,i} & \text{if } (y_{i-1,i} - f_{i-1,i}) < 0 \\ y_{i-1,i} - g_{i-1} & \text{if } (y_{i-1,i} - f_{i-1,i}) > g_{i-1} \end{cases}$$

if $i > 0$ and $x_{i-1,i} > 0$, wait to receive $x_{i-1,i}$ tasks from node $i-1$.

if $i < k-1$ and $x_{i,i+1} < 0$, wait to receive $|x_{i,i+1}|$ tasks from node $i+1$.

if $i > 0$ and $x_{i-1,i} < 0$, send $|x_{i-1,i}|$ tasks to node $i-1$.

if $i < k-1$ and $x_{i,i+1} > 0$, send $x_{i,i+1}$ tasks to node $i+1$.

Figure 9. Request Relocation Algorithm for a Chain.

Example 3: Table 1 and Figure 10 show an example for the load-balancing algorithm, where $m = 4$. T_i , Q_i , and $y_{i-1,i}$ are calculated in steps 1 and 2. In step 3, p_i^l and p_i^r are calculated so that by comparing their values each overloaded node can determine its intention of sending extra load, which is the value of f . For example, node 3 intends to send a request to node 2 so that $f_{2,3} = -1$. Similarly, node 4 intends to send a request to node 5 so that $f_{4,5} = 1$. When $p_i^l = p_i^r$, the node intends to send its extra load to left. In step 4, the value of x is calculated according to the values of y , f and g . Since the value of x must be in range $(y - g, y)$, the final flow could be different from the value of f . For example, node 4 intends to send a request to node 5 according to the value of f , but finally it sends a request to node 3. Otherwise, nodes 5, 6, or 7 would be overloaded. \square

Table 1. Example for Request Relocation

Node i	0	1	2	3	4	5	6	7
w_i	3	4	0	5	5	1	8	3
T_i	29	26	22	22	17	12	11	3
Q_i	32	28	24	20	16	12	8	4
$y_{i-1,i}$	-	2	2	-2	-1	0	-3	1
p_i^l	0	1	0	1	2	0	1	0
p_i^r	0	1	0	2	1	0	1	0
b_i^l	-1	0	0	2	2	2	5	5
b_i^r	0	2	2	5	5	5	7	7
$f_{i-1,i}$	-	0	0	-1	0	1	-4	0
g_i	3	2	2	0	0	0	0	0
$x_{i-1,i}$	-	0	0	-2	-1	0	-3	1

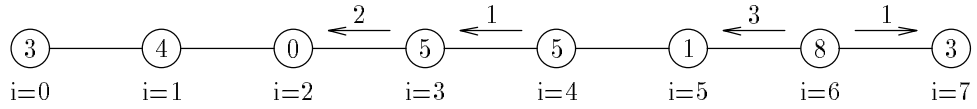


Figure 10. Example for Request Relocation.

The complexity of each algorithm is listed below:

Conflict-free scheduling	$O(mN \log N)$
Request delaying	$O(mN)$
Request relocation	$O(N)$

6. Admission Control

With the conflict-free scheduling algorithm, request delaying algorithm, and load-balancing algorithm, we are ready to define the admission control policy for new requests. A new request can be admitted if and only if it can be scheduled without conflict. It could be scheduled to a time slot immediately upon arrival at a delivery node if there is a conflict-free time slot available. Otherwise, the new request must be delayed, or other video streams need to be rescheduled to accommodate the new request.

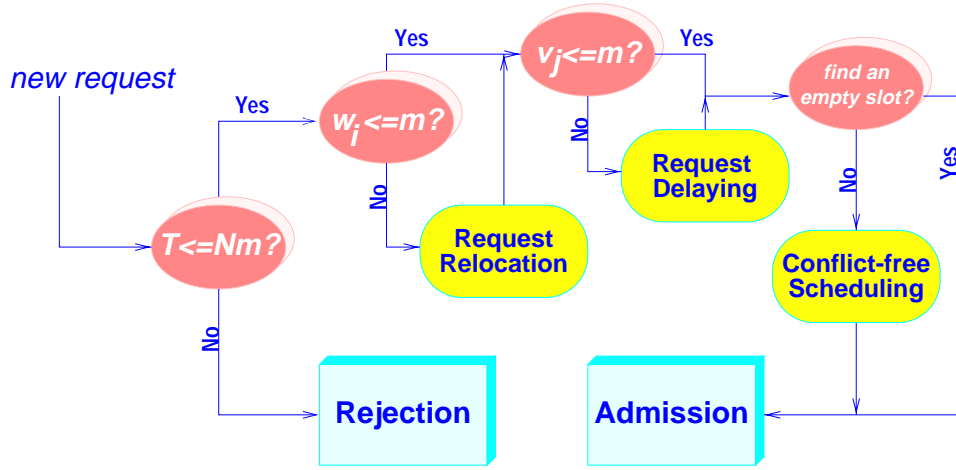


Figure 11. Admission Control Algorithm

The admission control algorithm is shown in Figure 11. Let T' be the current total number of video streams being serviced. When one or more requests arrive, the condition $T \leq Nm$ is checked, where $T = T' + r$ and r is the number of newly arrived requests. If the condition holds, the requests are admitted. Otherwise, only $r' = Nm - T'$ requests can be admitted and others must be rejected. For the admitted requests, it is to test whether some delivery nodes are overloaded. If so, some requests are to be relocated. The next step is to check whether some storage nodes are overloaded. If so, some requests are to be delayed. Finally, it is to check whether there are non-conflict time slots for the new requests. If not, the conflict-free scheduling algorithm is applied to rearrange video streams.

The admission control algorithm deals with the single request and the multiple requests separately. The following steps are applied to multiple requests arriving the system simultaneously:

- if for any delivery node i , $w_i > m$, the request relocation algorithm is applied;
- if for any storage node j , $v_j^t > m$, the request delay algorithm is applied;

- if any request cannot find an empty conflict-free slot, the conflict-free scheduling algorithm is applied.

In the situation that only a single request arrives the system, a simple policy can be applied which is described as follows:

- assume the new request r_k arrives, if $w_i > m$, the left-nearest and the right-nearest underloaded nodes of node i are searched and the new request is transferred to its nearest underloaded node;
- if $v_j^t > m$, the request is to be delayed to its nearest hole;
- if there is a conflict-free time slot for the new request, it is scheduled to the time slot, otherwise the conflict-free scheduling algorithm is applied.

The policy that schedules multiple requests together can reduce the scheduling overhead and sometime lead to a better schedule.

7. Simulation and Performance

We have conducted a simulation study on various scheduling algorithms, which include *Greedy algorithm*, *Conflict Free Scheduling (CFS)*, *CFS with Delay (CFS/D)*, *CFS with Delay and Relocation (CFS/D&R)*. The simulation is configured with five parameters: the total number of storage/delivery nodes N , the number of slots per time cycle m , the load measurement L based on the available capacity, the average file size Z specified in terms of number of slots required to retrieve the entire file, and the duration of simulation in terms of number of time cycles. In the following simulation, the average file size is 200 time slots, and the simulation duration is 20,000 time cycles.

The arrival rate, R , can be calculated from the following equation:

$$R = \frac{N * m * L}{Z}$$

At each step, the probability that there are exactly r new requests is given by:

$$P(r) = \rho^r (1 - \rho),$$

where, ρ can be calculated as:

$$R = \sum_{i=0}^{\infty} i \cdot \rho^i (1 - \rho) \quad \text{and} \quad \rho = R / (1 + R).$$

The simulation proceeds step by step for each time cycle as follows:

- At each step the number of new requests is calculated according to probability ρ .

- (b) For each new request, randomly generate its delivery node, its starting storage node where the first block of the file is stored, and the size of the file in terms of number of slots.
- (c) The admission control policy is applied to determine which requests are to be admitted or rejected, and if admitted, the scheduling algorithm is used to schedule the requests. If a single new request is generated in this step, the algorithms for the single request are employed. If more than one new requests are generated, the algorithms for multiple requests are employed.

At the end of the simulation, statistics obtained include the total number of requests generated, the number of rejected requests, the number of delayed requests, the number of relocated requests, etc.

The different versions of the CFS algorithm have first been studied. They are tested with different arrival rates. Table 2 shows performance of algorithms CFS, CFS/D, and CFS/D&R on different loads. For each instance, listed in the table are: (1) the percentage of delayed requests, (2) the average delay per delayed request in terms of time cycles, (3) the percentage of relocated requests, (4) the average distance per relocated request in terms of hops, and (5) rejection rate, which is the ratio of the number of rejected requests and the total number of requests. Here, items (2) and (4) give more detailed information about the delayed requests and the relocated requests, respectively. It is shown that when the system is lightly loaded, the delay is around one time cycle. As the load increases, the average delay may increase to a few time cycles. The number of delayed requests and the average delay of the CFS/D&R algorithm are substantially larger than the CFS/D algorithm, because CFS/D&R admits much more requests that are rejected by CFS/D. These requests must be relocated and delayed. Also, the number of relocated requests and the average distance of the CFS/D&R algorithm increase with the load. For a heavily-loaded system, such as 90% or more, CFS or CFS/D exhibits a high rejection rate, whereas CFS/D&R can reduce the rejection rate significantly. The penalty paid for a low rejection rate is the large number of relocations and delays to be enforced for incoming requests due to the highly saturated system.

Next, we compare performance of Greedy algorithm and different versions of the CFS algorithm. Figure 12 shows performance for different values of m , the number of time slots in a time cycle. When m increases, the rejection rate for all four algorithms decreases. The partially reason for this phenomenon is that when each delivery node handles more requests the difference between the number of requests arrived becomes smaller. Figure 13 shows performance for different number of nodes, N . There is no substantial change in rejection rates when N varies for the Greedy, CFS, and CFS/D algorithms. But for CFS/D&R the rejection rate decreases when N increases, since there could be more chances to relocate the newly arrived requests. Finally, Figure 14 shows performance for different loads. Obviously, the rejection rates of all four algorithms become high as load increases. For CFS/D&R, the rejection rate remains around 0 until the load becomes 80%, and at load of 80%, the rejection rate is 0.64%. In general, a rejection rate of less than 1% is considered to be acceptable.

Table 2. Performance Variances with Different Loads for m=10 and N=16.

Load	Algorithms	Delayed requests	Avg. delay per delayed request	Relocated requests	Avg. distance per relocated request	Rejection rate
30%	CFS	-	-	-	-	0.21%
	CFS/D	.08%	1.00	-	-	0.12%
	CFS/D&R	.08%	1.00	.12%	1.00	0.00%
40%	CFS	-	-	-	-	1.32%
	CFS/D	.70%	1.00	-	-	0.72%
	CFS/D&R	.70%	1.00	.72%	1.00	0.00%
50%	CFS	-	-	-	-	3.80%
	CFS/D	2.25%	1.07	-	-	2.12%
	CFS/D&R	2.54%	1.09	2.32%	1.00	0.00%
60%	CFS	-	-	-	-	7.05%
	CFS/D	4.87%	1.15	-	-	4.07%
	CFS/D&R	6.49%	1.28	5.49%	1.04	0.00%
70%	CFS	-	-	-	-	12.32%
	CFS/D	7.65%	1.24	-	-	8.43%
	CFS/D&R	13.1%	1.58	11.8%	1.11	0.05%
80%	CFS	-	-	-	-	18.32%
	CFS/D	11.7%	1.36	-	-	13.32%
	CFS/D&R	31.4%	2.36	31.7%	1.49	0.64%
90%	CFS	-	-	-	-	21.46%
	CFS/D	14.1%	1.44	-	-	15.97%
	CFS/D&R	42.4%	2.89	42.1%	1.75	2.57%
100%	CFS	-	-	-	-	26.61%
	CFS/D	17.0%	1.55	-	-	21.27%
	CFS/D&R	53.3%	3.69	52.6%	2.19	8.25%

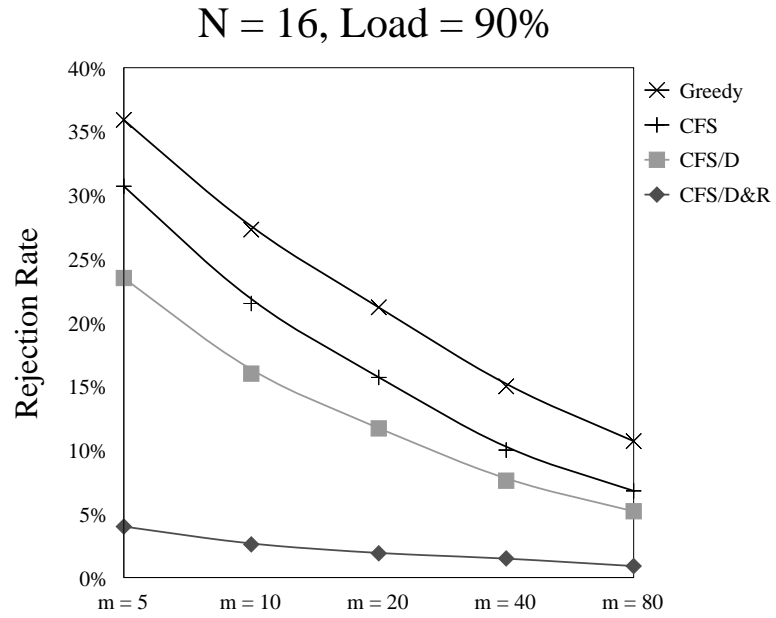


Figure 12. Rejection Rate as Number of Time Slots Varies.

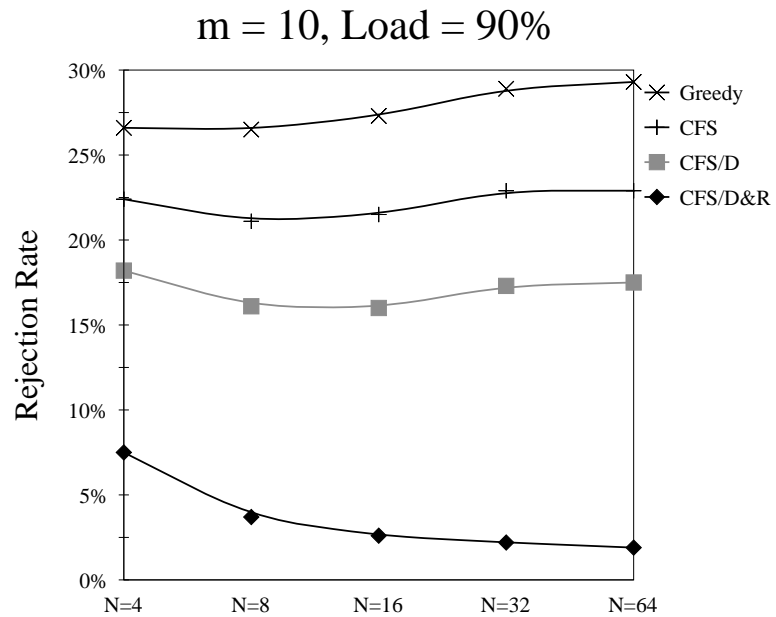


Figure 13. Rejection Rate as Number of Storage Nodes Varies.

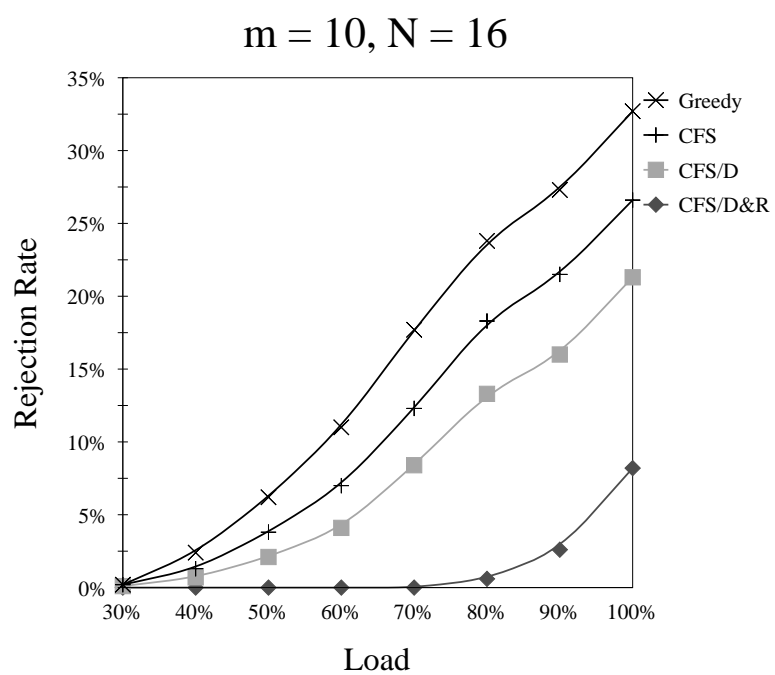


Figure 14. Rejection Rate as Load Increases.

8. Related Work

During the past few years, several research projects investigated techniques for designing video servers. Related issues include disk layout and scheduling strategies, admission control strategies, real-time support, etc. [5, 10, 16, 19, 6]. Most research works assumed single disk storage system. Several research projects investigated the design techniques for parallel video servers [15, 4, 13, 18, 14, 7, 11]. Video data stripping schemes have been studied [15, 1, 5, 10, 7, 11]. However, video stream scheduling problem has not been formally addressed. Many systems rely on statistical multiplexing which cannot truly guarantee the Quality of Service. The system resources cannot be fully utilized without a sophisticated scheduling strategy. Not arranging the video streams properly, large buffer space is required. In particular, the prefetch technique requests huge memory space [8]. A greedy scheduling for clustered architectures has been proposed by Reddy [15]. Our approach aims at precise scheduling of video streams to maximize system throughput and to minimize the usage of buffers.

Mitra, a scalable media server, is a cluster of multi-disk workstations connected using an ATM switch [11]. The data is partitioned into many blocks, each of them is distributed across d disks. Each piece of this block is termed as fragment. The time period is fixed for all media types. In each time period, the scheduler issues a read request for the block to the d disks. Then each disk transfers its fragment to the client. No scheduling has been applied to the transfer of these fragments. Tiger, a direct-access architecture, uses the wide striping strategy to balance the load and the similar delay strategy described in this paper. It assumes that all of the files have the same bit rate [7]. However, Tiger does not have a conflict-free scheduling algorithm to avoid unnecessary delay. Instead of scheduling to the current time slot, a newly arrived request is delayed to next available slot. For a large system, the delay can be significant. After the system load reaches 50% to 80%, the delay increases drastically. Our algorithms can be applied to direct-access architecture equally well. By using the conflict-free scheduling algorithm, only requests that exceed the disk capacity are delayed. Our experimental result shows that if a request has to be delayed, the average delay is less than three time cycles and rejection rate is less than 3% when the system load is 90%.

9. Conclusion and Future Work

This paper addressed the scheduling problem for parallel CBR video servers. A number of algorithms including conflict-free scheduling, request delay minimization, request relocation, and admission control have been presented. Combining these algorithms, we are able to achieve optimal scheduling in distributed memory clustered architectures.

Most current scheduling techniques are based on statistical multiplexing. To guarantee timely video signal delivery, a system is not able to reach its maximum capacity with statistical multiplexing. Our method precisely schedules video streams for Quality of Service requirement. This method maximizes the video server capacity,

and minimizes the delay time. It can minimize the buffer size required to reduce the effect of unpredictable network delay. The scheme described in this paper has been extended to support interactive operations such as fast forward and rewind [20].

We did not discuss the network conflict problem in this paper. Network traffic scheduling is an important issue to be addressed. For some topologies such as the Omega network and hypercube network, once a set of requests has been scheduled without conflict for a time cycle, there is no conflict for other cycles. However, for other network topologies, it might be necessary to do scheduling for each individual time cycle.

The paper only addressed scheduling problem for the CBR video. In some applications, clients may request the retrieval of variable bit rate encoded media streams. Scheduling this type of requests is much more difficult which is to be addressed in the future work.

Acknowledgments

The authors would like to thank Xin He for his helpful discussion and Karthikeyan Samuthiram for his simulation work. This research was partially supported by NSF grants CCR-9505300 and CCR-9625784.

References

1. S. Berson, S Ghandeharizadeh, R. Muntz, and X. Ju. Staggered striping in multimedia information systems. In *The fifth Int'l conf. on management of data*, May 1994.
2. J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. Macmillian Press, 1976.
3. D. W. Brubeck and L. A. Rowe. Hierarchical storage management in a distributed VOD system. *IEEE Multimedia*, 3(3):37-47, Fall 1996.
4. 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.
5. E. Chang and A. Zakhor. Scalable video data placement on parallel disk array. In *Proceedings of storage and retrieval for image and video databases II*, October 1994.
6. J. K. Dey, C. S. Shih, and M. Kumar. Storage subsystem in a large multimedia server for high-speed network environments. In *IS&T/SPIE symposium on electronic imaging science and technology*, February 1994.
7. W.J. Bolosky *et al.* The Tiger video fileserver. In *NOSSDAV 96*, April 1996.
8. C. Freedman and D. DeWitt. The SPIFFI scalable video-on-demand server. In *SIGMOD*, June 1995.
9. 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.
10. D. Ghandeharizadeh and L. Ramos. Continuous retrieval of multimedia data using parallelism. *IEEE Trans. Knowledge Data Engineering*, 5(4):658-669, August 1993.
11. S. Ghandeharizadeh, R. Zimmermann, W. Shi, R. Rejaie, D. Ierardi, and T-W Li. Mitra: A scalable continuous media server. *Multimedia Tools and Applications*, 5(1):79-108, July 1997.
12. P. Hall. On representatives of subsets. *J. London Math. Soc.*, 10:26-30, 1935.
13. 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.
14. 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.

15. 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.
16. A. L. N. Reddy and J. Wyllie. Disk-scheduling in a multimedia I/O system. In *the first ACM Int'l. Conf. on Multimedia*, pages 225–233, August 1993.
17. W. Shu and M.Y. Wu. Runtime Incremental Parallel Scheduling (RIPS) on distributed memory computers. *IEEE Trans. Parallel and Distributed System*, 7(6):637–649, June 1996.
18. 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.
19. H. Vin and V. Rangan. Admission control algorithm for multimedia on-demand servers. In *The third international workshop on network and operating system support for digital audio and video*, pages 56–69, November 1992.
20. M.Y. Wu and W. Shu. Scheduling for interactive operations in parallel video servers. In *IEEE Conference on Multimedia Computing Systems*, pages 178–185, June 1997.

Contributing Authors

Min-You Wu Min-You Wu received the M.S. degree from the Graduate School of Academia Sinica, Beijing, China, and the Ph.D. degree from Santa Clara University, California. Before he joined the Department of Electrical and Computer Engineering, University of Central Florida, where he is currently an Associate Professor, he has held various positions at University of Illinois at Urbana-Champaign, University of California at Irvine, Yale University, Syracuse University, and State University of New York at Buffalo. His research interests include parallel and distributed systems, compilers for parallel computers, programming tools, VLSI design, and multimedia systems. He has published over 70 journal and conference papers in the above areas and edited two special issues on parallel operating systems. He is a member of ACM and a senior member of IEEE. He is listed in International Who's Who of Information Technology.

Wei Shu Wei Shu received the Ph.D. degree from the University of Illinois at Urbana-Champaign in 1990, the M.S. degree from Santa Clara University in 1984, and the B.S. degree from Hefei Polytechnic University, China, in 1982. Since then, she worked at Yale University and the State University of New York at Buffalo. She is currently an Associate Professor in the Department of Electrical and Computer Engineering, University of Central Florida. Her current interests include dynamic scheduling, resource management, runtime support systems for parallel and distributed processing, multimedia networking, and operating system support for large-scale distributed simulation. She is a member of ACM and IEEE.