A Novel Video-On-Demand Storage Architecture for Supporting Constant Frame Rate with Variable Bit Rate Retrieval

S.W. Lau swlau@cs.cuhk.hk

John C.S. Lui* cslui@cs.cuhk.hk

Department of Computer Science, The Chinese University of Hong Kong

Abstract. One of the quality of service (QOS) factors in video-on-demand (VOD) applications is to provide high resolution quality to end users. One way to achieve this is to provide a constant display frame rate (e.g., 30 frames/sec) at the display station. However, due to the nature of video files and compression technique applied, video frame sizes vary significantly from frame to frame. Therefore, although the display frame rate is fixed, data retrieval is a variable bit rate process. Conventional VOD storage servers assume a peak rate retrieval of video files. Therefore, the number of concurrent requests to the VOD server cannot be maximized. In this paper, we consider a VOD storage server which can support a fixed frame rate at the display and at the same time, variable bit rates retrieval of compressed video files. We describe 1) video files layout strategy, 2) request scheduling algorithm, 3) buffering issues and, 4) various VCR features support such that the number of concurrent requests can be maximized.

1 Introduction

With recent advances in networking technologies and vast improvement of storage systems, it is now feasible to provide multimedia services such as multimedia mail, news distribution, computer animation advertisement, library information system and home entertainment to users via high bandwidth network. Due to this reason, research in multimedia storage systems has received a lot of attention in the past few years.

Usually video object, by itself, has large storage and large bandwidth requirement. For example, a 100 minutes HDTV video object requires an average of 800 megabits per second (Mbps) bandwidth and around 60 Gbytes of storage [1]. Typically, compression (e.g, MPEG) is used to reduce the storage as well as the network bandwidth requirement so that video object can be delivered to end users. However, it is important to observe that the frame size (bits) varies significantly from frame to frame. This is due to:

- Variations of information content among frames. For example, if there is a sudden change of scene in the video file, then the size of a group of frames from one scene may be much larger than the size of group of frames in another scene.
- Nature of the compression algorithm. For example, MPEG compression files have three types of frames, namely, I-frame, P-frame, and B-frame. I-frames are coded independent of any other frames. On the other hand, B-frames are coded by interpolating between previous and a future I or P-frame. Therefore, the I-frames might have size which is 100 to 200 times the size of the B-frame[6].

* This work was supported in part by the Croucher Foundation Research Grant and the CUHK Direct Grant.

Most recent research works [12, 11, 1, 10] concentrated on the study of multimedia storage systems which support the retrieval of a multimedia object at a *peak* display bandwidth (bits/sec); for example, assuming that the display bandwidth of object is fixed at $B_{display}$ Mbps throughout the duration of the display, the storage server retrieves object using $B_{display}$ Mbps disk I/O bandwidth. This approach may be applicable for low-quality video files, for example, if the frame size of these video objects is not highly variable, or if users do not require a high quality video display at their viewing stations.

However, designing a multimedia storage server using a peak display bandwidth assumption has the following drawbacks:

- It is often desirable to provide users with high speed browsing functions such as *fast forward* or *fast rewind* with viewing. To support these type of functions, the retrieve rate may be much higher than regular display bandwidth. If designers use the worst case bandwidth allocation[2], then storage I/O bandwidth is not efficiently utilized.
- For HDTV typed of video applications, there will be a high variability in the display bandwidth. For example, the MPEG-III for supporting HDTV has display bandwidth that can range from 5 to 20 Mbps [3]. Again, storage I/O bandwidth is not efficiently utilized if the peak bandwidth allocation policy is used.
- Using the worst case display bandwidth allocation as the disk I/O retrieval rate, it is possible to have *buffer build up problem.* For example, if buffer is allocated for *each* disk retrieval, since the object display bandwidth is less than the retrieval rate, the number of allocated buffer will build up, thereby demanding more buffer from the storage server. Since the total number of buffers in the storage server is finite, this implies that the number of concurrent requests that the system can support is reduced.

In this paper, our aim is to design a VOD storage server which can support a fixed frame rate at the display station and at the same time, variable bit rate retrieval in the storage subsystem. In this way, I/O bandwidth is efficiently utilized and therefore, the number of concurrent requests to the VOD storage server can be increased compared to traditional VOD architectures. Also, the VOD storage server is implemented on parallel disks such that load balancing can be achieved. We also illustrate 1) video files layout policy, 2) parallel disk scheduling policy for incoming request, 3) admission control policy and, 4) how to support VCR typed function such as fast forward with viewing.

It is interesting to point out that in [4], authors described different ways of reducing I/O demand for the VOD storage server. The class of techniques used is based on *dynamically merging* of requests to the same video object during their display periods such that system resources (e.g., disk I/O operations, buffers ... etc) can be reduced. The main idea is to dynamically adjust the frame rates of similar requests so that there is possibility to merge two or more requests into one system request. It is important to point out that the VOD storage server we describe in this paper can support this interesting I/O reduction techniques with zero storage overhead. For more information about dynamic merging techniques, please refer to [4].

The organization of the paper is as follows. In Section 2, we describe the architecture of VOD storage server and video file layout policy. In Section 3, we describe a request scheduling algorithm that optimizes disk utilization and

[2] worst case bandwidth allocation is defined as taking the peak display bandwidth requirement during the duration of display, including possibility of support fast forward or fast rewind with viewing.

buffer space. In Section 4, we describe how to support VCR typed function such as fast forward with viewing. Section 5 describes some related work in VOD storage server and performance study is carried out in Section 6. Conclusion is given in Section 7.

2 The VOD Storage Server Architecture

The VOD storage architecture we consider is a hierarchical storage architecture with a tertiary tape storage device and a parallel disk array as illustrated in Figure 1. All video objects reside permanently on the tape cartridge in the

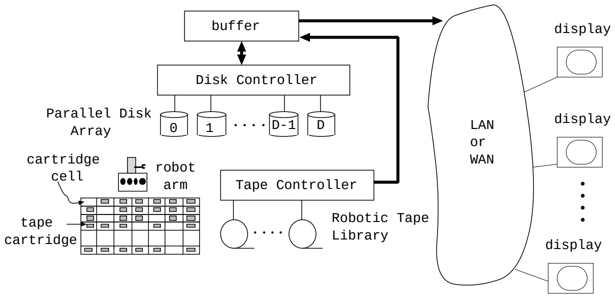

Fig. 1. Cost-effective multimedia storage server

tape system. This type of two-tier storage architecture provides a cost-effective solution for VOD since the size of each video file may be very large and it is prohibitive (in terms of cost) to store all of them in a magnetic disk storage system.

Upon request arrival to the VOD storage server, if the video file is not available in the disk storage system, then the video object is retrieved from the tape system, via the robot arm, to the disk storage system. Efficient tape scheduling algorithms for VOD storage system have been studied in [8]. Upon retrieval from the disk storage system, video frames are put into the buffer so that they can be packetized and be delivered through the network (either in form of LAN or WAN) to different display stations. In this study, we focus on the I/O bottleneck phenomena and we assume that the bandwidth of the network exceeds the display bandwidth requirement of a video object. The assumption is justified considering the current high-speed networking technologies.

Before we talk about the video file layout policy. We assume that there is a database within the video filesystem such that upon encoding and compression of video object, the size of each frame of the video object is kept in the database for future access and reference. This database is important so later on we can have an efficient parallel disk scheduling algorithm.

A file layout policy is as follow. The compressed video file $\mathcal{F}$ is broken down into equal size fragments F_i, such that $\mathcal{F} = \{F_0 \cup F_1 \cup \ldots \cup F_n\}$ and $F_i \cap F_j = \emptyset$. Fragments are then assigned to the parallel disk in a round-robin manner, for example, fragment F_i is assigned to disk $i\,(mod\,D)$ where D is the number of disks in the storage subsystem. Upon request, retrieval is done in unit of fragment. The size of the fragment F_i is currently chosen to be two cylinders. Note that the rationale for choosing this size is to minimize each seek and rotation overhead for each fragment transfer[3]. For simplicity of presentation, we assume that each frame is stored in exactly one fragment. That is, a frame will not be split such that part of it is stored in fragment F_i and remaining part of the frame

[3] Based on the parameters we used in our study, the seek overhead for each fragment transfer is about 10%.

is stored in fragment F_{i+1}. Also, due to compression applied, different fragments may contain different number of video frames.

A multimedia object is retrieved by reading fragments of the object in a pipe-lining manner, for example, the retrieval of fragment i will be completed before the end of the playback period of fragment $i-1$. Since the file system has a database containing frame sizes of video object and the fragment size is fixed. To support a constant frame rate at the display station, playback time for each fragment can be easily computed. Therefore, once the start time of a request is determined, the deadline of a fragment retrieval of the request can be determined by adding up the playback periods of all preceding fragments and the request start time. We term the above scheme as *round-robin striping*.

3 Round-robin Striping

We define a *task* as an operation of retrieving a fragment from a disk. The execution time of a task has three components: the seek time, the rotational latency and the fragment transfer time. The first two components are the overhead within a task. Since our goal is to design a storage server which supports high display bandwidth multimedia objects, we bound the task's overhead such that it is less than 10% of the transfer time of a task. For a typical 1.2 Gbytes hard disk, when the fragment size is two cylinders, about 10% of the disk bandwidth is wasted in switching overheads. In the rest of the discussion, we assume that the execution time of a task is a constant of τ seconds.

The algorithm we propose, the round-robin striping algorithm, has several components. It first finds an optimal *execution order* for the tasks in each disk in the parallel disk subsystem. With the execution order, the *execution schedule* of the tasks in a disk is then determined in a way such that the fragment buffer requirement is minimal. This execution schedule defines the start time of the request and the retrieval and completion time of each task in a request.

Upon arrival of a request, the algorithm *ScheduleRequest* will be executed to test whether the storage server can accept this new request. If the system can accept this new request, it finds the execution order and execution schedule of all tasks within the request. The algorithm ScheduleRequest is defined as follows.

Algorithm ScheduleRequest *Assume the number of buffer (which is used for storing fragment after its retrieval) in the storage server is B. Given a new playback request R for a multimedia object O and the latest time, t, such that the requester can wait until, the algorithm determines whether object O can start to display on or before time t and the earliest possible start time of the playback request R.*

1 **procedure** ScheduleRequest(R : request;
        var starttime : **real**; **var** success : **boolean**);
  { If object O can start on or before t, success is set to
    TRUE and starttime is set to the earliest start time
    of the request, otherwise, success is set to FALSE. }
2 **begin**
3   call *StartTime()* to find the earliest start time of R
    and the optimal execution order of tasks in each disk
    without considering the fragment buffer requirement;
4   **If** the earliest start time $\leq t$ **then**
5   **begin**
6   call *ScheduleTask()* to find the execution
        schedule of all tasks in the system which
        minimizes buffer requirement;

```
7        while (the maximum required number of
         buffer > B) and(the earliest start time ≤ t) do
8        begin
9            increment the start time of R by a fixed value;
10           call ScheduleTask() to find the execution
             schedule of all tasks in the system which
             minimizes buffer requirement;
11       end
12   end;
13   if the earliest start time ≤ t then
14   begin
15       success := TRUE;
16       starttime := earliest start time;
17   end
18   else
19       success := FALSE;
20 end
```

We propose two algorithms for routines in line 3 and line 6 (or line 10) of algorithm *ScheduleRequest*. They are the *StartTime* and *ScheduleTask* algorithms respectively.

3.1 Algorithm StartTime

Algorithm *StartTime* is responsible to find the earliest start time of a request R as well as the execution orders of all tasks in R in the parallel disks subsystem. To schedule tasks within a disk, the earliest deadline first (EDF) algorithm is used. We choose the EDF algorithm because the EDF algorithm is found to be optimal for scheduling tasks in a single disk[9]. Temporal relation for related tasks (belong to the same request) that were assigned to different disks are maintained by the algorithm (via each fragment deadline). The system maintains a doubly linked-list which contains all tasks scheduled for the particular disk. Tasks are arranged in an ascending order of deadlines. Algorithm *StartTime* determines the start time of a new request R by the following steps: 1) set the start time of request R to the current time; 2) given that current start time and execution time of each tasks, determines the deadlines of each tasks of R; 3) find the earliest completion time for each task of R by scheduling the task in the unused period between the scheduled tasks (of other requests) in the disk; 4) let LT_i be the late time task i as:

$$LT_i\text{=(completion time of task } i - \text{deadline of tasks } i)^+$$

then the start time of request R is:

$$\text{current time+execution time of the first task of } R + \max_{i \in R}\{LT_i\}$$

Algorithm StartTime *Given the linked lists of scheduled tasks in the disk (pointed by p_1) and the linked lists of tasks of a new request R (pointed by p_2), the following algorithm finds the earliest start time of request R. Please refer to [7] for the detail of algorithm.*

```
function StartTime: real;
begin
    i : integer;
    S : Set of Task;
    {MaxDelay is equal to max_{i∈R}{LT_i} }
    MaxDelay, RemainTime : real;
    p1, p2 : pointer to a task in a linked list;
    MaxDelay := 0.0;
    for i := 1 to D do
    begin
        p1 := points to the head of the list L1 of all scheduled task of other
            requests not equal to R in disk i in ascending order of deadlines;
        p2 := points to the head of the list L2 of tasks in R for disk i in ascending
            order of deadlines;
        while p1 <> nil do
        begin
            S = ∅;
            put task pointed by p1 into set S;
            RemainTime := deadline of task pointed by p1 - current time -
                the total execution time of all tasks in set S;
            while((RemainTime >= execution time of task pointed by p2)
                and (p2 <> nil) do
            begin
                put task pointed by p2 into set S;
                MaxDelay = max {MaxDelay,Late Time(LT) for task pointed by p2};
                update p2 to point to next task in L2;
            end;
            if RemainTime < 0 then
                move p2 to a previous task and remove minimum number of
                tasks in R from set S such that RemainTime ≥ 0 ;
            update p1 to point to the next task in L1;
        end;
        while (p2 <> nil) do
        begin
            put p2 into the set S;
            MaxDelay = max {MaxDelay,Late Time(LT) for task pointed by p2};
            update p2 to point to next task in L2;
        end;
    end;
    return (current time + execution time of the first task of R + MaxDelay)
end
```

Lemma 1. *Algorithm StartTime requires $\Theta(2N_{\overline{R}} + N_R)$ link traversals to find the earliest start time of request R where $N_{\overline{R}}$ and N_R are the total number of tasks of other requests not equal to R in the parallel disks and the number of tasks of the new request R respectively.*

Proof: please refer to [7]. ∎

3.2 Algorithm ScheduleTask

Given the execution order of tasks, Algorithm *ScheduleTask* minimizes the required buffer space by scanning the task list of each disk in the reverse order of task deadlines and determines the latest retrieval time of each task. The retrieval time of a task is the time at which the task starts to execute.

Algorithm ScheduleTask Let N_i be the number of tasks in disk i. Let $J_{i,1}, \ldots, J_{i,N_i}$ be the tasks in disk i listed in ascending order of deadlines. The retrieval time of all tasks in the disks are determined by the following procedure:

```
procedure ScheduleTask()
begin
  i, j : integer;
  CurTime : float;
  for i:= 1 to D do
  begin
    CurTime := +∞;
    for j := N_i down to 1 do
    begin
      if (CurTime > deadline of J_{i,j}) then
        CurTime := deadline of J_{i,j};
      CurTime := CurTime - execution time of J_{i,j};
      retrieval time of J_{i,j} := CurTime;
    end
  end
end
```

Theorem 2. *Given a set of tasks that can be scheduled by the EDF algorithm, Algorithm ScheduleTask finds a feasible schedule of tasks that requires a minimal buffer space.*

Proof: please refer to [7]. ∎

4 Interactive Playback Support

In this section, we describe how to support VCR typed functions. For simplicity of presentation, we only describe support of fast forward with viewing. Rewind with viewing can be supported using similar technique.

In order to support the fast forward with viewing operation, we can break it up into two sub-operations, 1) viewing fast forward frames when the fast forward button is pushed, and 2) resume normal display after the fast forward button is released.

Let us discuss the second sub-operation first. If the peak rate retrieval policy is used, it is very simple to support resuming the normal display after the fast forward button is released. To illustrate this point, Figure 2 illustrates the frag-

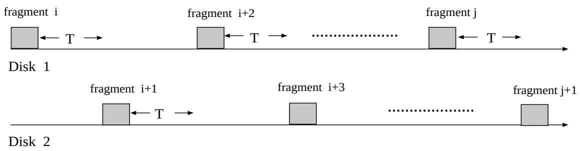

Fig. 2. Fast Forward with Peak Allocation

ment retrieval schedule if the peak rate retrieval policy is used and D (number of parallel disks in the system) is equal to two. Since the system assumes a peak retrieval rate when a request arrives at the system, the time between successive fragment retrieval is fixed with time T. If the fast forward operation is to skip fragment i to fragment $j-1$, it is a simple operation to re-map the retrieval schedule such that fragment k will move up $\frac{j-i}{D}$ units. For example, for disk 1 in Figure 2, instead of retrieving fragment $\{ i, i+2, i+4, \ldots \}$, fragment $\{ j, j+2, j+4, \ldots \}$ are retrieved instead. This is always possible because:

– the size of each fragment is the same.

- the inter-fragment time is fixed as T (due to peak retrieval rate policy).
- retrieval schedule of fragment can always be re-mapped (e.g., fragment j re-map to fragment i) because all fragments retrieval are scheduled during request arrival. Therefore, we are guaranteed that there is a *slot* available for re-mapping fragment retrieval.

On the other hand, to support variable bit rate retrieval, time between fragments retrieval is not fixed. Figure 3 illustrates a variable bit rate retrieval. Now,

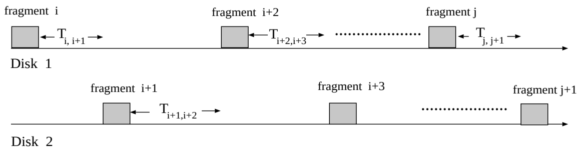

Fig. 3. Fast Forward with Peak Allocation

since the time between fragments is not fixed, direct re-mapping of fragments may not be possible because, 1) since $T_{i,i+1}$ (time between retrieval of fragment i and fragment $i+1$) is not equal to $T_{j,j+1}$, therefore the video continuity may not be capable of being satisfied, 2) disk storage system is servicing other requests (e.g., requests to other video files) and the presence of these requests' schedules (which were determined at each request arrival instant) might make the re-mapping impossible. Therefore, direct re-mapping will create a situation such that after the fast forward button is released, normal video display cannot be continued.

To overcome the problem stated above, we first defined an advanced unit, $\mathcal{U}_a$ (e.g., each $\mathcal{U}_a$ is equivalent to 30 seconds of normal display time). Note that $\mathcal{U}_a$ is a system tunable parameter. User can now specify the number N_a, which indicates the number of $\mathcal{U}_a$ advance units he/she wants to fast forward. Once this information is entered, the system treats this as a new request and try to schedule it using the algorithm we described in the previous section. If the request can be scheduled, it is accepted and previous allocated resources (e.g., I/O schedule for fragment retrieval) can be deleted. On the other hand, if request cannot be scheduled, the system has the following options:

- reject the request and continue display the video file.
- increment (or decrement) N_a and try to schedule this request again. We can repeat this approach several times. If the system still cannot schedule the request, the system simply informs the user that the fast forward operation cannot be scheduled.

Using this approach, the worst case that can happen is that the system rejects fast forward with viewing request and normal video display can be continued. It is important to point out that in our performance study, we observe that with very high probability, fast forward with viewing request can be accepted for a large range of system load.

Now we are in the position to answer how to support fast forward with viewing operation. In general, there are three approaches: 1. *separated replicated files* in [1]; 2) *fragment sampling* in [2] and 3) *frame sequence resampling*. Each has different degrees of storage overhead and quality of service. For detail description, please refer to [7].

5 Related Work

In [1], staggered striping was proposed for file layout architecture in support of video on demand. We choose staggered striping for comparison because it uses the similar concept of file declustering, that is, splitting up a video file and laying the file across the parallel disk array.

First, let us briefly define the concept of staggered striping. A video object O is split into separate sub-objects $O_1, O_2, \ldots, O_n$ where $O_i \cap O_j = \emptyset$. Each sub-object is composed of M fragments such that fragments of the same subobject are stored in M adjacent disks[4]. A stride is defined to be the distance (in term of number of disk) between the first fragment of sub-object O_i and the first fragment of sub-object O_{i+1}. Request retrieval is accomplished by retrieving a sub-object at a time; therefore, for each sub-object retrieval, M adjacent disks are involved in retrieving M fragments.

Staggered striping suffers from the following problems, 1) *time fragmentation* [1] where there are idle disks but they cannot process waiting requests; 2) it can only support constant retrieval rates; and 3) request may be starved (or be forced to wait forever in the request queue) if non first-in-first-out (non-FIFO) queueing discipline is used for reducing request waiting time.

In contrast, round-robin striping is starvation-free because all requests are scheduled when they arrive. Also, round-robin striping has the advantage that the start time of playback of a request can be determined when the request arrives. This information is very useful for system resource management.

In [10], a storage layout method that optimizes several concurrent requests of the same video object is proposed. With this method, fragments of several requests of different phases of the same object can be retrieved with one disk movement (seek and rotational movement). The phase of a request is the point in time at which the object is being displayed. The method can improve the system performance when several requests retrieve the same objects. However, the approach performs worse when many different video objects are retrieved concurrently because only a small fragment of an object is retrieved for each disk movement. In [10, 2], several ways to support interactive playback function are proposed, such as fast forward and backward functions by skipping fragments. Lastly, there is an on going research work on staggered striping for a compressed video object [5].

6 Performance Evaluation

In this section, we evaluate the performance of round-robin striping and staggered striping by computer simulation for four different cases: Case 1) the required display bandwidth for the playback of an object is constant and it is equal to 5 Mbytes/second; Case 2) The required display bandwidth for the playback of an object varies over time. The playback time is split into intervals of 0.6 seconds. The display bandwidth requirement within an interval is uniformly distributed between 2.5 Mbytes/second and 5 Mbytes/second; Case 3) The required display bandwidth for the playback of an object is constant and it is equal to 10 Mbytes/second; Case 4) The required display bandwidth for the playback of an object varies over time. The playback time is split into intervals of 0.6 seconds. The display bandwidth requirement within an interval is uniformly distributed between 5 Mbytes/second and 10 Mbytes/second. The main parameters of the

Parameter	Cases 1 & 2	Cases 3 & 4
Number of disks	200	400
Disk bandwidth	2.5 MB/s	2.5 MB/s
Number of objects	100	100
Playback period of an object	600 secs	600 secs
Fragment size	1.5 MB	1.5 MB

Table 1. Main Simulation Parameters.

simulation are shown in Table 1. The simulation runs for round-robin striping

[4] M is also known as the degree of declustering in Staggered Striping

and staggered striping with various parameters are abbreviated in Table 2.

SS(i, k, n)	staggered striping with non-fifo request queueing discipline, stride = k, n fragment buffers in storage server, and the bandwidth requirement is that of case i.
RS(i, n)	round-robin striping with n fragment buffers in storage server and the bandwidth requirement is that of case i.

Table 2. Abbreviations for simulation runs.

We measured the average waiting time of round-robin striping and staggered striping by an open queueing network in which the request arrival process is a Poison process. And we measured the maximum throughput of round-robbin striping and staggered striping by a closed queueing network in which the number of requests is fixed. Tables 3 (Table 4) presents the average waiting time (in seconds) of a playback request for Cases 1 and 2 (Cases 3 and 4). The average response time of a playback request is the average waiting time plus the execution time of the retrieval of the first fragment of an object. Tables 5 and 6 show the maximum throughput of staggered striping and round-robin striping for different bandwidth requirements and different number of fragment buffers.

Arrival rate (req/hr)	100	200	300	400	450	500
SS(1, 1, 400) or SS(2, 1, 400)	0.684	1.368	2.795	7.199	14.049	33.016
SS(1, 2, 400) or SS(2, 2, 400)	0.423	0.669	1.160	2.509	4.325	9.707
SS(1, 3, 400) or SS(2, 3, 400)	0.506	0.916	1.892	5.459	12.574	42.961
RS(1, 400)	0.186	0.520	1.232	4.047	13.624	95.746
RS(1, 480)	0.186	0.518	1.232	3.346	8.694	28.568
RS(1, ∞)	0.186	0.518	1.210	3.346	7.662	19.525
RS(2, 400)	0.142	0.353	0.667	1.304	2.005	10.171
RS(2, 480)	0.142	0.353	0.667	1.304	1.865	3.507
RS(2, ∞)	0.142	0.353	0.667	1.304	1.836	2.721

Table 3. Average waiting times (in seconds) for Cases 1 and 2.

Arrival rate (req/hr)	100	200	300	400	450	500
SS(3, 1, 800) or SS(4, 1, 800)	1.187	2.749	6.314	16.886	35.255	92.823
SS(3, 4, 800) or SS(4, 4, 800)	0.433	0.669	1.160	2.509	4.325	9.707
SS(3, 7, 800) or SS(4, 7, 800)	0.549	1.089	2.593	9.877	27.664	95.110
RS(3, 800)	0.186	0.518	1.210	3.362	8.389	28.533
RS(3, ∞)	0.186	0.518	1.210	3.228	7.607	19.978
RS(4, 640)	0.140	0.355	0.676	1.410	3.525	20.489
RS(4, 800)	0.140	0.355	0.676	1.307	1.872	4.595
RS(4, ∞)	0.140	0.355	0.676	1.307	1.866	2.814

Table 4. Average waiting times (in seconds) for Cases 3 and 4.

For fast forward with viewing, we measured the waiting time of fast forward request of multimedia objects. The main parameters of the simulation is shown in Table 7. Each multimedia object is a 90-minute video object. The playback period of each object is divided into 0.6 second intervals. In each interval, the display bandwidth requirement is uniformly distributed between 2.5 Mbytes/second and 5.0 Mbytes/second. In this simulation, a request will demand a fast forward with viewing operation at time t where t is a random variable uniformly distributed between 0 to 60 minutes. For each fast forward with viewing operation, the object is scanned at 3 times the normal playback speed, and the display time of fast forward with viewing is 1 minute. Hence,

# of Fragment Buffers (n)	400	480	∞
SS(1, 1, n) or SS(2, 1, n)	572.215	572.215	572.215
SS(1, 2, n) or SS(2, 2, n)	593.910	593.910	593.910
SS(1, 3, n) or SS(2, 3, n)	558.361	558.361	558.361
RS(1, n)	510.405	545.433	595.724
RS(2, n)	577.315	623.028	793.334

Table 5. Maximum throughput (in requests/hour) for Cases 1 and 2.

# of Fragment Buffers (n)	800	960	∞
SS(3, 1, n) or SS(4, 1, n)	563.728	563.728	563.728
SS(3, 4, n) or SS(4, 4, n)	594.593	594.593	594.593
SS(3, 7, n) or SS(4, 7, n)	542.630	542.630	542.630
RS(3, n)	568.330	587.028	596.013
RS(4, n)	613.455	655.730	798.090

Table 6. Maximum throughput (in request/hour) for Cases 3 and 4.

the average execution time of a request is equal to the display time $\times$ average retrieval bandwidth/disk bandwidth or $(87+1) \times 3.75/2.5$minutes $= 132$ minutes Assuming that the VOD server has 200 parallel disks, the theoretical maximum throughput is $60 \times 200/132$requests/hour $= 90.91$ requests/hour.

We measured the waiting time distributions for four combinations of arrival rate and number of fragment buffers. Let $FF(n, r)$ denote a simulation run with n fragment buffers in the storage server and request arrival rate r (unit is in requests/hour). Figure 4 shows probability distribution functions of waiting time for the four combinations.

In general, we see that round-robin striping outperforms staggered striping in most situations. Even under the constant display bandwidth assumption (case 1 and 3), round-robin striping has better response time for low to moderate arrival rate. It is important to point out that video file varies in fragment sizes (as we have argued in Section 1. Therefore, what we are showing is to compare round-robin against staggered striping in some rare cases. For system throughput, round-robin is comparable to staggered striping. For VCR-typed functions such as fast forward, we see the system accepts fast forward with viewing request with high probability.

7 Conclusion

In this paper, we have proposed a VOD storage server that can support constant display frame rate at the display station and at the same time, variable bit rate retrieval. We discussed the video file layout policy, the admission control, request scheduling, buffer management as well as VCR-typed functions support. Performance studies have been carried out and we have shown that it can have good request response and system throughput. Also, the type of VOD can support techniques as described in [4] to reduce the I/O cost.

Parameter	Case 5
Number of disks	200
Disk bandwidth	2.5 MB/sec
Number of objects	20
Normal playback period of an object	90 minutes
Fragment size	1.5 MB

Table 7. Simulation Parameters for Fast Forward Playback.

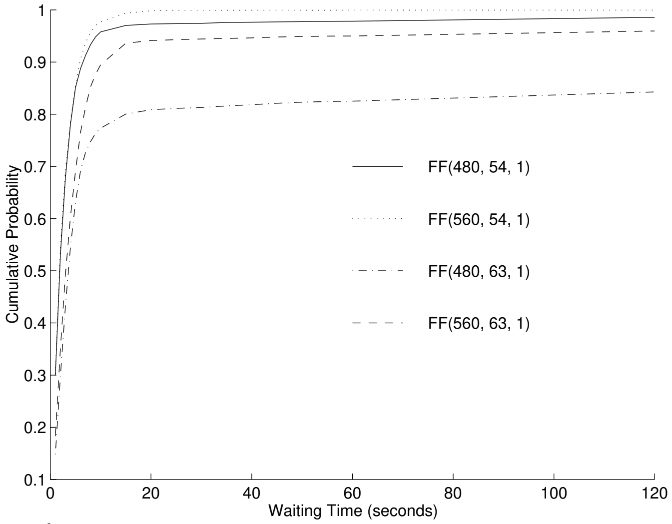

Fig. 4. Probability distribution functions of waiting time.

References

1. S. Berson, S. Ghandeharizadeh, R.R. Muntz, X. Ju. "Staggered Striping in Multimedia Information Systems", In *ACM SIGMOD Conf.*, pp. 79-90, June, 1994.
2. M. S. Chen, D. D. Kandlur, and P. S. Yu, "Support for Fully Interactive Playout in a Disk-Array-Based Video Server." In *Proceedings of the 2nd Annual ACM Multimedia Conference*, October 1994.
3. Borko Furht. "Multimedia Systems: An Overview." In *IEEE Multimedia Magazine*, Spring Issue, 1994.
4. L. Golubchik, John C.S. Lui, R.R. Muntz, "Reducing I/O demands in Video-On-Demand Storage Servers." In *Proceedings of the ACM SIGMETRICS/PERFORMANCE '95 Conference. May, 1995.*
5. L. Golubchik, R.R. Muntz. *Private Communication.*
6. D. Le Gall. "MPEG: A Video Compression Standard for Multimedia Applications" *Communications of the ACM, April, 1991.*
7. S.W. Lau, John C.S. Lui. "A Novel Video-On-Demand Storage Architecture for Supporting Constant Frame Rate with Variable Bit Rate Retrieval", Technical Report CS-TR-94-12, Department of Computer Science, the Chinese University of Hong Kong, 1994.
8. S.W. Lau, John C.S. Lui, P.C. Wong. "A Cost-effective Near-line Storage Server for Multimedia System" *In Proceedings of the* 11^{th} *International Conference on Data Engineering. March, 1995.*
9. C.L. Liu and J.W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", *JACM, Vol. 20, No.1, Jan. 1973. pp. 47-61.*
10. B. Ozden, A. Biliris, R. Rastogi and A. Silberschaz. "A Low-cost Storage Server for Movie on Demand Databases", *Proceedings of the 20th International Conference on Very Large Databases, September 1994.*
11. P. V. Rangan and H. M. Vin, "Efficient Storage Techniques for Digital Continuous Multimedia." In *Transactions on Knowledge and Data Engineering*, August 1993.
12. H. M. Vin and P. V. Rangan, "Designing a Multi-User HDTV Storage Server." In *IEEE Journal on Selected Areas in Communication*, Vol. 11, No. 1, January 1993.

This article was processed using the LaTeX macro package with LLNCS style