A Novel Data Caching Scheme for Multimedia Servers

James Z. Wang Rantan K. Guha

School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL 32816-2362, U. S. A.
E-mail: {zwang, guha}@cs.ucf.edu

Abstract

In this paper, we propose a *Bi-directional Fragmental Pipelining (BFP)* technique and its variable buffer size data-caching scheme BFP_V to reduce the disk I/O bandwidth requirement for multimedia servers. Our mathematical analysis shows that the BFP technique is superior to the traditional unidirectional pipelining technique in terms of memory buffer space requirement. We further demonstrate that the memory buffer management using BFP_V is better than that of using the fixed buffer size approach BFP_F. We have mathematically proved that BFP_V saves more disk I/O bandwidth than BFP_F does using the same memory buffer space. Our simulation results have quantitatively confirmed our analysis.

Key Words: Multimedia Servers, Pipelining, Data caching scheme, Buffer Management, Disk I/O Bandwidth

1 Introduction

Internet and multimedia applications have driven the storage technologies improving in an incredible pace. Two most important features of multimedia data are the large storage and I/O bandwidth requirement. To satisfy the demand, the storage capacity of the magnetic-platter disks has increased dramatically while the cost per byte of the disks is constantly dropping. Unlike the storage capacity, the disk I/O bandwidth increases in a rather slower speed due to the mechanical movement of the disk read/write heads. Furthermore, the effective disk I/O bandwidth is much less than the advertised I/O bandwidth. There are many factors, such as operating system, file system, application software and concurrent disk I/O requests, which prevent the effective disk I/O bandwidth to reach its advertised one. For instance, the performance study for Microsoft SQL 7.0 has shown that the typical Wide Ultra SCSI-3 hard drive is capable of providing Windows and SQL server with about 75 non-sequential (random) and 150 sequential I/O operations per second [9]. Advertised transfer rates in terms of megabytes (MB) for these hard drives range around 40 MB/second.

However, it is much more likely for a database server to be constrained by the 75/150 I/O transfers per second than the 40 MB/second transfer rate. Based on this observation, it is reasonable to expect at most 9.6 MB/second I/O processing capability from a given hard drive by doing strictly sequential read or write SQL server operations on it [9]. To prove this observation, we have done some experimental studies over a set of hard disks in terms of their effective disk I/O bandwidths. Those various disk models are from different vendors, including IBM, Seagate and Quantum. Our experiments are all conducted under Microsoft operating systems with NTSF file system. We create a single process to use the file I/O functions provided by the operating systems to sequentially read through a large data file and monitor the actual data transfer rate. The results are depicted in Table 1.

Table 1: Effective disk I/O bandwidth for various disk drives.

Disk Model	Operating System	I/O Interface	ETR	SDR	ADR
IBM DHEA-28451	Win 2000 Server	IDE	33.3	10	4.3
IBM DHEA-28451	Win NT Server 4.0	IDE	33.3	10	3.8
IBM DHEA-28451	Win 2000 Professional	IDE	33.3	10	1.7
IBM DTTA-371010	Win NT Server 4.0	IDE	33.3	13	4.5
IBM DDRS-34560D	Win 2000 Server	SCSI	40	13.3	7.4
IBM DTLA-307030	Win 2000 Professional	Ultra SCSI	100	37	23.9
Seagate ST-39216W	Win 2000 Server	Ultra Wide SCSI	40	N/A	11.5
Seagate ST-29102LC	Win NT Server 4.0	Ultra 2 SCSI	40	N/A	14.5
Quantum Viking II 9.1WLS	Win NT Server 4.0	Ultra SCSI	40	14	8.9

Besides the disk models, we also list the operating systems and the disk I/O interfaces, because those are very important factors in determining the effective disk I/O bandwidths. We note here that ETR in the table stands for External Transfer Rates (MB/Sec). They are the advertised disk I/O bandwidth commonly used by disk drive vendors. Usually those numbers are their maximum transfer rates that cannot be sustained. So some vendors provide Sustained Data Rates (MB/Sec) in their product specification. They are listed in column SDR of our table. However, those sustained data rates have not taken the effects of operating system, file system and the actual used disk I/O interface. Those factors create more overheads to the disk I/O operations. The Actual Data Rates (MB/Sec) obtained by our experiment are listed in column ADR.

Our experimental results have shown that the effective disk I/O bandwidths are far less than the advertised external transfer rates. The actual data rates are also less than the advertised sustained data rates. The actual data rates in Table 1 are sequential disk I/O bandwidth. If concurrent disk I/O operations are involved, we expect the actual data rates to drop even further. Thus the disk I/O bandwidth sets a hard limitation on the number of customers who can be served simultaneously in a multimedia system. To serve more concurrent video streams, one approach is to increase the number of hard disks in the multimedia systems. Those disks can be organized as disk array. To fully utilize the aggregate I/O bandwidth of the disks, striping techniques are widely used [1, 2, 3, 15], though some researchers question the cost of the striping techniques for interactive video services [12].

Although multimedia data are permanently stored in the hard disks, they must be processed in the memory buffer before sending out to the client's workstations. So the most recently accessed data are temporarily cached in memory buffer. The most famous memory cache replacement algorithm is the LRU algorithm. Using LRU, the least recently accessed pages will be replaced out of memory cache first. Several research studies tried to adopt various LRU algorithms into the *video-on-demand* systems [8, 10, 11, 13]. However, those aforementioned LRU variants are all focused on the replacement of the individual memory pages. We categorize them as *page-level memory buffer management*. Those page-level buffer management schemes have their limitations in dealing with continuously and simultaneously displaying video objects to multiple users. For instance, video object *V* consists of *n* pages, P_1, P_2, …, P_n. We assume there is only one client *A* currently accessing video *V*. We also assume the traditional LRU algorithm is used. If the second client *B* requests this video when client *A* is accessing page P_k, it is more likely that page P_1 is out of the memory buffer than page P_{k-1} does. However, client *B* actually requests the first page P_1 to start the playback of object *V*.

To address the problem, we propose a *Bi-directional Fragmental Pipelining (BFP)* technique for video object playback to reduce the disk I/O bandwidth requirement and discuss the data caching scheme for memory buffer management based on the proposed pipelining technique. Using the proposed pipelining technique, the data caching scheme for memory buffer management is no longer focused on the individual memory pages. The replacement of an individual page ties to its context in the video object. Unlike in the traditional LRU algorithms, the most recent used pages might be replaced in our cache replacement scheme. The rest of this paper is organized as follows. In Section 2, the traditional unidirectional pipelining technique and its related memory buffer management strategy are discussed. In Section 3, we propose the BFP technique and discuss its advantages over the traditional unidirectional pipelining technique. The variable cache size buffer management scheme BFP_V has also been discussed in the section. In Section 4, we present the simulation results to show the superiority of the BFP_V scheme. The memory cache replacement scheme based on BFP_V is discussed in Section 5. We have our concluding remarks and discuss the future works in Section 6.

2 Reducing the I/O Bandwidth Requirement by Pipelining

Pipelining is a well-known technique used in cache management. This technique is also very natural for processing video objects. We can always prefetch the trailing pages of the video object into the memory for processing while the system is displaying the current pages. In this section, we discuss how to use the traditional pipelining techniques to reduce the disk I/O bandwidth requirement for displaying the video objects.

2.1 Traditional Pipelining Technique

The idea of pipelining is to divide the whole multimedia object X into a sequence of slices $(X_0, X_1, X_2, \cdots, X_{n-1}, X_n)$ such that the playback time of X_{i-1} eclipses the time required to materialize X_i, where $1 \le i \le n$. This strategy ensures that X_i is in memory ready for continuous playback when the system finishes X_{i-1}. This traditional pipelining technique is presented in Figure 1.

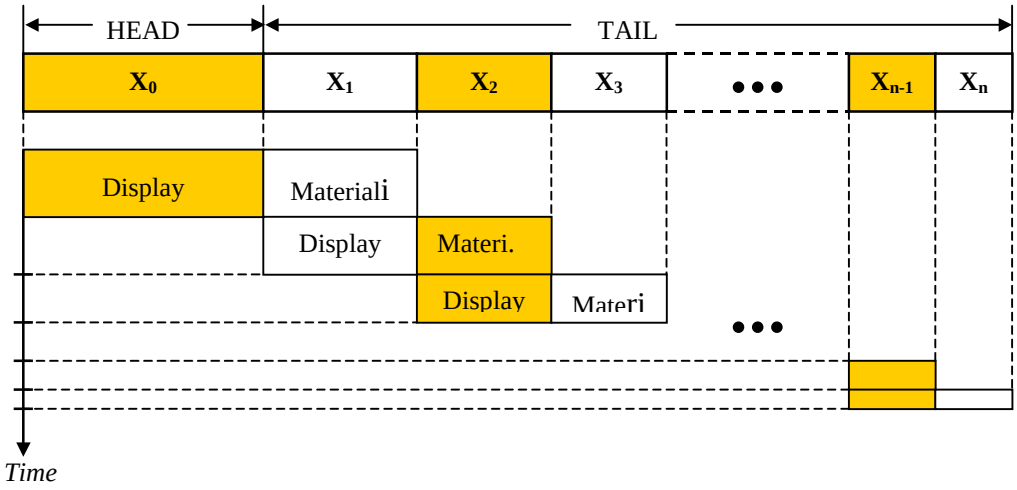

Figure 1: Pipelining technique.

We define PCR (*Production Consumption Rate*) as the ratio between the materialization rate (B_m) of reading data from disks to memory buffer for an object and the playback rate (B_p) of sending data from the server to a playback station, i.e., $PCR = \frac{B_m}{B_p}$. If $B_m > B_p$, not only the memory buffer space but also the disk I/O bandwidth is wasted, because the system materializes more data pages into the memory than that the display station can consume. Generally we have $B_m \le B_p$ and $PCR \le 1$. To simplify our analysis, we assume that B_m and B_p are constant through the entire video object X. Under this condition, the sizes of the data slices can be computed as follows:

$$Size(X) = Size(X_0) + Size(X_1) + \cdots + Size(X_n)$$
$$Size(X_i) = PCR \cdot Size(X_{i-1}), 0 < i \le n$$

If $B_m = B_p$, i.e., $PCR = 1$, we have

$$Size(X_0) = Size(X_1) = \cdots = Size(X_n) = \frac{1}{n+1} \cdot Size(X).$$

However, this situation does not reduce the disk I/O bandwidth requirement. To reduce the disk I/O bandwidth requirement, we need $B_m < B_p$. In this case, we have

$$Size(X_i) = PCR^i \cdot Size(X_0) \text{ and } Size(X_0) = \frac{1 - PCR}{1 - PCR^{n+1}} \cdot Size(X).$$

It is necessary that slice X_0 is resident in the memory buffer to allow the pipelining to take place. We call slice X_0 as the *HEAD* of the object (or pipeline) and the rest of slices as the *TAIL*. So X_0 needs to be as small as possible, that means letting n be as large as possible. To achieve this, the last slice X_n must be as small as possible because

$$\text{Size}(X_n) = \text{PCR}^n \cdot \text{Size}(X_0).$$

Therefore, we should choose the size of the last slice to be that of a minimal I/O unit (normally one disk page or memory page). Also, we know the time to display the sequence $(X_0, X_1, X_2, \cdots, X_{n-1})$ is equal to the time to materialize the sequence $(X_1, X_2, \cdots, X_{n-1}, X_n)$. That means

$$\frac{\text{Size}(X) - \text{Size}(X_0)}{B_m} = \frac{\text{Size}(X) - \text{Size}(X_n)}{B_p}.$$

Based on this equation and taking the factor that $\text{Size}(X_n)$ is very small, we can easily derive:

$$\text{Size}(X_0) \approx (1 - \text{PCR}) \cdot \text{Size}(X). \qquad (1)$$

2.2 Memory Buffer Management

Using the traditional unidirectional pipelining technique discussed above, we can save the disk I/O bandwidth by $1 - \text{PCR}$, if the head segment X_0 is cached in the memory. To start the pipeline, we have to have head segment X_0 memory resident. We start reading the next segment X_1 to the memory while the system is displaying the head segment X_0. Usually we do not want the head X_0 to be replaced out of the memory buffer. Otherwise, it is not possible for the new requests to start the pipeline immediately. This is the same problem as the traditional LRU replacement algorithm we pointed out in Section 1. So besides the head segment X_0, we need additional memory buffer space to store segment X_1, since we cannot reclaim the buffer space from X_0. However, we can replace the data starting from segment X_1 as soon as the data in this segment is displayed. Thus for each concurrent stream, a single circular buffer presented by Figure 2 can be used.

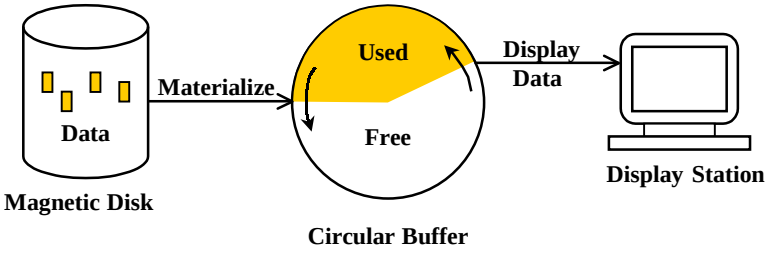

Figure 2: Circular memory buffer.

The initial size for the circular buffer should be the size of the second segment X_1 plus one more minimal I/O unit (disk page or memory page). So if we use the minimal I/O unit as our space measurement unit, the circular buffer size would be

$$\text{Initial Buffer Size} \approx (1 - \text{PCR}) \cdot \text{PCR} \cdot \text{Size}(X).$$

Because we have $\text{PCR} < 1$, i.e., $X_{i-1} > X_i$ for any $i > 1$, the required circular buffer size is shrinking during the entire playback process. We can reclaim the freed memory from the shrinking buffer. If we have several users accessing the same object starting at different times, the average shrinking buffer size will be

$$\text{Average Buffer Size} \approx \frac{1}{2} \cdot (1 - \text{PCR}) \cdot \text{PCR} \cdot \text{Size}(X).$$

We notice that $(1 - \text{PCR}) \cdot \text{PCR} \leq \frac{1}{4}$. It means the maximum initial size for circular buffer is around $\frac{1}{4}$ of the object size.

There are several problems with this unidirectional pipelining technique and its buffer management scheme. First, besides the head segment X_0 shared by the concurrent streams accessing the same object, each stream requires a circular buffer initially equal to $(1 - \text{PCR}) \cdot \text{PCR} \cdot \text{Size}(X)$. Sometimes, this buffer is too large. Second, the freed memory fragments from the shrinking buffers are not easy to be reclaimed as another circular buffer because of their fragmental nature. Third, the traditional pipelining technique and its circular buffer management scheme are not suitable for VCR functions such as fast reverse and fast forward. If the user wants to reverse to a previous viewed point, most of data have to be read from the hard disks because they might have been replaced out of the circular buffer. On the other hand, if the user wants to skip a portion of the video and to start at some later point, the system still has to read many data from the current point to the specified later point with a faster speed. Otherwise, the pipelining cannot be restarted. To solve those problems, we propose a novel *Bi-directional Fragmental Pipelining* (BFP) technique in next section.

3 Bi-directional Fragmental Pipelining Technique

The motivation of solving the problems associated with the traditional pipelining technique and its buffer management scheme requires us to find a better way to handle the video data. The unidirectional nature of the traditional pipelining technique is its inherent problem. This problem restricts us from handling the video data more efficiently. We attack this problem by fragmenting the video data as to be explained in this section.

3.1 Video Object Fragmentation

We divide the whole video object into two categories of fragments. One category is called *Disk-resident fragment* (or D-fragment) and the other is called *Cached fragment* (or C-fragment). Although the entire video object is permanently stored in the hard disks, the C-fragments are also temporarily cached in memory buffer. A pair of D, C fragments consist of a processing unit P. The D and C-fragments interleave the data file as illustrated in Figure 3. With this file organization, the pipelining is performed as follows. As the system displays

the processing unit $P_i = (D_i, C_i)$, it materializes the next D-fragment D_{i+1} from the hard disks. Obviously, to maintain the continuous playback, the elapsed time of displaying P_i should be equal to the elapsed time of materializing D_{i+1}. Mathematically, we can express the above requirements as follows:

$$\frac{Size(D_{i+1})}{B_m} = \frac{Size(D_i) + Size(C_i)}{B_p} \quad \text{for } i \geq 0. \qquad (2)$$

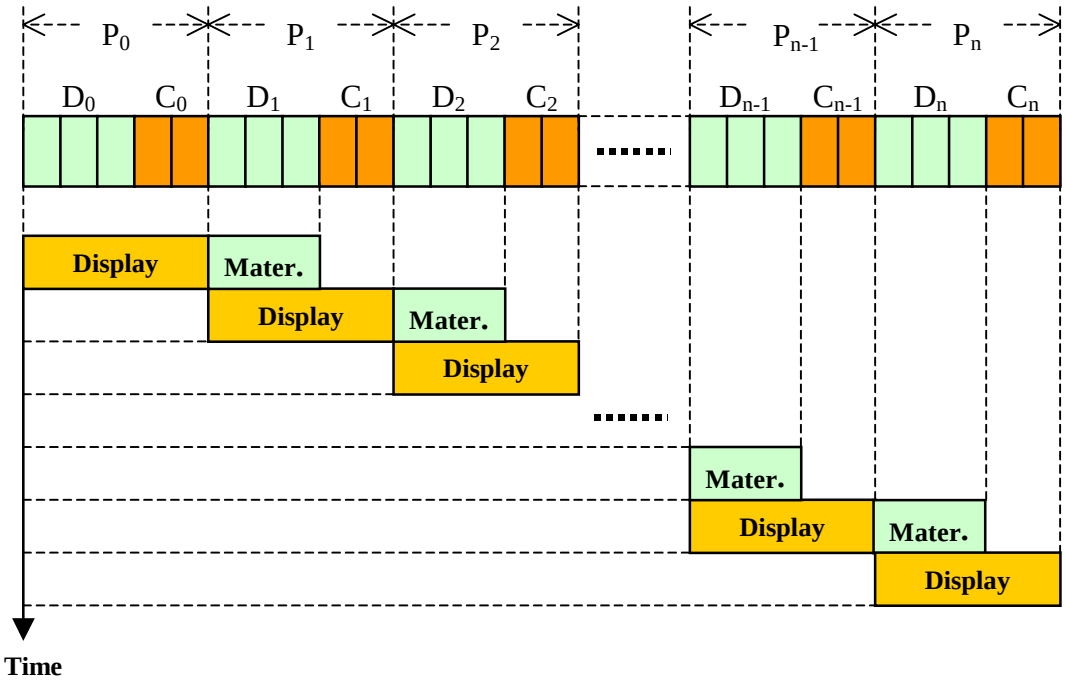

Figure 3: Bi-directional fragmental pipelining technique.

We call this pipelining technique as *Bi-directional Fragmental Pipelining* (BFP) technique. To ease the implementation, we make the sizes of the fragments more uniform by letting $Size(D_i) = S_D$ and $Size(C_i) = S_C$ for $i \geq 0$. Substituting the values S_D and S_C into Equation (2), we have:

$$\frac{S_D}{B_m} = \frac{S_D + S_C}{B_p}. \qquad (3)$$

thus,

$$\frac{S_D}{S_D + S_C} = \frac{B_m}{B_p} = PCR. \qquad (4)$$

The size of the entire object X can then be computed in terms of S_D and S_C as follows:

$$Size(X) = (n+1) \cdot (S_D + S_C).$$

Let *SD* and *SC* represent the accumulative size of all the D-fragments and the accumulative size of all the C-fragments, respectively. *SD* and *SC* can be computed as follows:

$$SD = (n+1) \cdot S_D = PCR \cdot Size(X), \qquad (5)$$

and

$$SC = (n+1) \cdot S_C = (1 - PCR) \cdot Size(X). \qquad (6)$$

3.2 Buffer Management Scheme

With the new pipelining scheme BFP, all C-fragments must be cached in the memory buffer before the pipeline can start. Comparing Equation (1) with Equation (6), the accumulated size for the cached fragments *SC* is equal to the size of the *HEAD* (i.e., X_0) in traditional pipelining technique.

There are two ways to implement the staging buffers:

Double Buffer: we use two staging buffers to load the D-fragments into the memory. This double buffer management scheme is depicted in Figure 4. Once buffer A is filled with D-fragment D_i, it forms a complete processing unit P_i with the cached C-fragment C_i. While the user is displaying the data in the processing unit P_i, the system materializes the data from D-fragment D_{i+1} into Buffer B. As soon as the user finishes displaying the processing unit P_i, the processing unit P_{i+1} is completely formed by the data loaded into buffer B and cached C-fragment C_{i+1}. So Buffer A and B can switch their roles. At this time, the user displays the data from the buffer B and C-fragment C_{i+1} while buffer A is used for materializing the next D-fragment. The buffer space requirement for this approach is $2 \cdot S_D$.

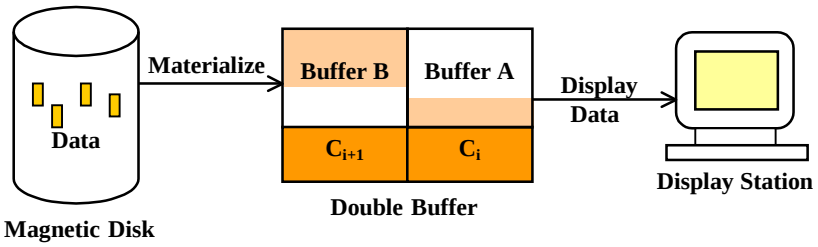

Figure 4: Double buffer management.

Single Buffer: This approach uses a similar circular buffer shared by both the consumption and production procedures as that used in the traditional pipelining technique. The buffer space requirement for this approach is $S_D + 1$. Unlike in the traditional pipelining technique, the size of this circular buffer does not shrink.

Obviously the double buffer scheme is much easier to implement than the circular buffer scheme. On the other hand, because the size of D-fragment can be very small, the single buffer approach won't save too much buffer space. Let us discuss the staging buffer space requirement in this situation. We should minimize S_D to keep the size of the staging buffer minimal. Equation (4) is repeated below:

$$\frac{S_D}{S_D + S_C} = \frac{B_m}{B_p} = PCR.$$

We can reduce the fraction $\frac{S_D}{S_D + S_C}$ to its irreducible form $\frac{p}{q} = \frac{S_D}{S_D + S_C}$, such that p is prime to q. Thus, the minimum size for the D-fragments is p blocks, where *block* is an efficient unit for I/O operations and display. Accordingly, the size for C-fragments should be $q - p$. For instance, if $PCR = 0.6$, we have $p = 3$ and $q = 5$. The sizes of D-fragments and C-fragments, therefore, are 3 blocks and 2 blocks, respectively. This example is illustrated in Figure 3. Let a block be 4 KBytes. The size of the staging buffer is $2 \cdot p$ blocks or 24 KBytes. On the other hand, using the traditional pipelining technique, the circular buffer space would be equal 24% of the object size. For a 10-minute MPEG-2-encoded NTSC-quality video clip using a playback rate of 6 Mbits/sec, the object size is about 450 MB. In this case, the circular buffer size would be 108 MB.

Besides the huge saving in the staging buffer, the BFP technique is natural for the implementation of VCR functions. Because the pipelining can be performed forwards as well as backwards, fast-forward and fast-reverse are very easy for BFP. Also the user can randomly jump into any middle point of the video object and start the playback from there without visible delay, because the system needs only read a small D-fragment into the memory to restart the pipeline. Furthermore, the frame skipping methods for fast-forward and fast reverse presented in [4, 7] are natural for BFP because it also skips through the file and caches in the memory buffer only every other fragment, i.e., skipping D-fragments and caching C-fragments. So the symmetrical nature of BFP allows fast-forward and fast-reverse to be done without even involving the disk I/O operations. If one needs to resume the normal play after a fast-forward or fast-reverse, the delay is essentially unnoticeable since this can be treated as a random access.

3.3 Managing Memory Space and Disk I/O Bandwidth

The novel BFP technique provides a systematic method for saving the disk I/O bandwidth. By caching the C-fragments of the video object in the memory buffer, we can save disk I/O bandwidth by $1 - PCR$. If all video objects in the database have the same PCR, we guarantee that the disk I/O bandwidth saving will be equal to PCR. For instance, we let all video objects have $PCR = 0.8$. That means $SC = 0.2 \cdot \text{Size(X)}$ and $B_m = 0.8 \cdot B_p$ for any object X. So we have saved 20% of disk I/O bandwidth by caching 20% of object fragments. Because every object uses the same PCR, we call this scheme as BFP_F, where subscript F stands for fixed PCR.

In multimedia systems, the access frequencies for different objects in the database vary. Some objects have more user requests. Some objects have less or even no user request. Usually, 80% of the user requests are contributed by only 20% of the video objects. To save the valuable memory cache space and disk I/O bandwidth, we should treat those different objects differently. Basically, we fragment more frequently requested objects (hotter objects) with larger C-fragments and less frequently requested objects (colder objects) with larger D-fragments. That means the hotter objects use smaller PCR than the colder objects do. We call this approach BFP_V, where subscript V stands for various PCR. The smaller PCRs for

hotter objects result in less disk I/O bandwidth requirement for each video stream accessing those objects. Because the majority of the concurrent streams are accessing those hotter objects, the total disk I/O bandwidth requirement is less. On the other hand, the larger PCRs for colder objects lead to smaller memory cached portion for those objects. Based on our discussion before, all concurrent streams accessing the same object share C-fragments, each object will only have one set of C-fragments in the memory buffer. Because cold objects consist of the majority of the entire database population, smaller C-fragments for those cold objects require less memory cache space. Furthermore, each video stream needs a staging buffer consists of one or two D-fragments. The majority of those video streams access the hot objects that have smaller D-fragments. That means the memory space for staging buffers is also less using the BFP_V approach. In general, we can mathematically prove that the BFP_V approach uses less disk I/O bandwidth than BFP_F does under the same memory cache space. The notations used in our discussion are listed as follows:

O_i :	Multimedia object i.
B_p :	Playback rate of each object. The system must be able to deliver data to a display station at this rate.
B_m :	Materialization rate for each stream if all objects use the same materialization rate.
C :	Concurrent streams currently supported by the video server.
M :	Number of objects currently used by concurrent streams.
k_i :	Number of concurrent streams that currently using object O_i. We have $\sum_{i=1}^{M} k_i = C$.
$B_m(i)$:	Materialization rate used for object O_i in the BFP_V approach.
$B_{disk}(S)$:	Disk I/O bandwidth required by scheme *S* to support the C concurrent streams that are using M multimedia objects. $B_{disk}(BFP_V)$ can be computed as $B_{disk}(BFP_V) = \sum_{i=1}^{M} k_i \cdot B_m(i)$.
Cache(*S*):	Cache space required by scheme *S* to cache the C-fragments of the M objects being currently used.
S_{cache}:	Total cache space.
$S_{database}$:	Database size for the M objects being currently used.
BFP_V:	BFP scheme using various materialization rates for objects.
BFP_F:	BFP scheme using the same materialization rate for all objects.
$PCR_v(i)$:	PCR used for object O_i in scheme BFP_V.
PCR_f:	PCR used for all objects in scheme BFP_F.

Without loss of generality, we assume that the multimedia objects are numbered so that $k_1 \le k_2 \le k_3 \le \cdots \le k_M$, where k_i denotes the concurrent streams using object O_i currently. We also assume the data size for each video object is S.

First we consider the BFP_F scheme. To minimize the disk I/O bandwidth requirement for BFP_F, we should let C-fragment be as large as possible. So the materialization rate assigned to each object is

$$B_m = (1 - \frac{S_{cache}}{S_{database}}) \cdot B_p ,$$

and

$$PCR_f = \frac{B_m}{B_p} = 1 - \frac{S_{cache}}{S_{database}} ,$$

So we have

$$B_{disk}(BFP_F) = C \cdot B_m = C \cdot PCR_f \cdot B_p .$$

Now we discuss the BFP_V scheme. We assign the materialization rate to those M involved objects as follows:

$$B_p \geq B_m(1) \geq B_m(2) \geq \cdots \geq B_m(M)$$

and

$$1 \geq PCR_v(1) \geq PCR_v(2) \geq \cdots \geq PCR_v(M), \text{ where } PCR_v(i) = \frac{B_m(i)}{B_p}$$

On the other hand, we let

$$\frac{1}{M} \cdot \sum_{i=0}^{M} PCR_v(i) = PCR_f \qquad (8)$$

Equation (8) guarantees that the memory cache space occupied by those M objects using the BFP_V scheme is the same as that using the BFP_F scheme. This claim is easy to prove. First we have,

$$Cache(BFP_V) = \sum_{i=1}^{M} [(1 - PCR_v(i)) \cdot S] \qquad (9)$$

Combining Equation (8) and (9), we have

$$Cache(BFP_V) = M \cdot (1 - PCR_f) \cdot S = Cache(BFP_F)$$

As for disk I/O bandwidth requirement for the BFP_V scheme, we have

$$B_{disk}(BFP_V) = \sum_{i=1}^{M} k_i \cdot B_m(i) . \qquad (10)$$

Based on Equation (8), we also have

$$\frac{1}{M} \cdot \sum_{i=0}^{M} B_m(i) = PCR_f \cdot B_p . \qquad (11)$$

Now we will prove that

$$B_{disk}(BFP_V) \leq B_{disk}(BFP_F)$$

To facilitate our discussion, we first prove the following theorem.

Theorem 1. For any $n > 0$, if $0 \le X_1 \le X_2 \le X_3 \le \cdots \le X_n$ and $Y_1 \ge Y_2 \ge Y_3 \ge \cdots \ge Y_n \ge 0$, then

$$\sum_{i=1}^{n} X_i \cdot Y_i \le \frac{1}{n} \cdot \sum_{i=1}^{n} X_i \cdot \sum_{i=1}^{n} Y_i \le \sum_{i=1}^{n} X_i \cdot Y_{n-i+1}$$

PROOF:

This proof is by induction on *n*. First, it is trivial that the theorem is true for $n = 1$. When $n = 2$, we can derive the following:

$$X_1 \cdot Y_1 + X_2 \cdot Y_2 - \frac{1}{2} \cdot (X_1 + X_2) \cdot (Y_1 + Y_2)$$
$$= \frac{1}{2} \cdot (X_1 \cdot Y_1 + X_2 \cdot Y_2 - X_2 \cdot Y_1 - X_1 \cdot Y_2)$$
$$= \frac{1}{2} \cdot (X_1 - X_2) \cdot (Y_1 - Y_2) \le 0.$$

Thus, we have:

$$X_1 \cdot Y_1 + X_2 \cdot Y_2 \le \frac{1}{2} \cdot (X_1 + X_2) \cdot (Y_1 + Y_2).$$

Similarly, we can obtain:

$$\frac{1}{2} \cdot (X_1 + X_2) \cdot (Y_1 + Y_2) \le X_2 \cdot Y_1 + X_1 \cdot Y_2.$$

Now, assuming that the induction hypothesis is true for $n < k$, i.e.,

$$\sum_{i=1}^{k-1} X_i \cdot Y_i \le \frac{1}{k-1} \cdot \sum_{i=1}^{k-1} X_i \cdot \sum_{i=1}^{k-1} Y_i \le \sum_{i=1}^{k-1} X_i \cdot Y_{k-i}.$$

We next prove that it is also true for $n = k$.

Using $\sum_{i=1}^{k-1} X_i \cdot Y_i \le \frac{1}{k-1} \cdot \sum_{i=1}^{k-1} X_i \cdot \sum_{i=1}^{k-1} Y_i$, We have,

$$\sum_{i=1}^{k} X_i \cdot Y_i - \frac{\sum_{i=1}^{k} X_i \cdot \sum_{i=1}^{k} Y_i}{k}$$

$$= \frac{1}{k}\left[k \cdot \sum_{i=1}^{k} X_i \cdot Y_i - \left(\sum_{i=1}^{k-1} X_i + X_k\right) \cdot \left(\sum_{i=1}^{k-1} Y_i + Y_k\right)\right]$$

$$= \frac{1}{k}\left[k \cdot \sum_{i=1}^{k} X_i \cdot Y_i - \sum_{i=1}^{k-1} X_i \cdot \sum_{i=1}^{k-1} Y_i - X_k \cdot \sum_{i=1}^{k-1} Y_i - Y_k \cdot \sum_{i=1}^{k-1} X_i - X_k \cdot Y_k \right] \qquad (12)$$

$$\leq \frac{1}{k}\left[k \cdot \sum_{i=1}^{k} X_i \cdot Y_i - (k-1) \cdot \sum_{i=1}^{k-1} X_i \cdot Y_i - X_k \cdot \sum_{i=1}^{k-1} Y_i - Y_k \cdot \sum_{i=1}^{k-1} X_i - X_k \cdot Y_k \right]$$

$$= \frac{1}{k}\left[\sum_{i=1}^{k-1} X_i \cdot Y_i + (k-1) \cdot X_k \cdot Y_k - X_k \cdot \sum_{i=1}^{k-1} Y_i - Y_k \cdot \sum_{i=1}^{k-1} X_i \right]$$

Considering $X_i \leq X_k$ and $Y_i \geq Y_k$ for $0 < i < k$, we have:

$$X_i \cdot Y_i + X_k \cdot Y_k \leq X_k \cdot Y_i + X_i \cdot Y_k$$

Using this inequality to (12), we can obtain:

$$\sum_{i=1}^{k} X_i \cdot Y_i - \frac{\sum_{i=1}^{k} X_i \cdot \sum_{i=1}^{k} Y_i}{k}$$

$$\leq \frac{1}{k}\left[\sum_{i=1}^{k-1} X_i \cdot Y_i + (k-1) \cdot X_k \cdot Y_k - X_k \cdot \sum_{i=1}^{k-1} Y_i - Y_k \cdot \sum_{i=1}^{k-1} X_i \right]$$

$$\leq \frac{1}{k}\left[\sum_{i=1}^{k-1} X_i \cdot Y_i + (k-1) \cdot X_k \cdot Y_k - \sum_{i=1}^{k-1} (X_i \cdot Y_i + X_k \cdot Y_k) \right] = 0.$$

Thus, we have:

$$\sum_{i=1}^{k} X_i \cdot Y_i \leq \frac{\sum_{i=1}^{k} X_i \cdot \sum_{i=1}^{k} Y_i}{k}.$$

Similarly, we can derive:

$$\frac{\sum_{i=1}^{k} X_i \cdot \sum_{i=1}^{k} Y_i}{k} \geq \sum_{i=1}^{k} X_i \cdot Y_{k-i+1}.$$

Hence the induction step is proven. We can conclude that the theorem is true for any $n \geq 1$. **Q.B.E.**

Applying Theorem 1 to Equation (10) and using Equation (11), we can derive the following:

$$
\begin{aligned}
B_{disk}(BFP_V) &= \sum_{i=1}^{M} k_i \cdot B_m(i) \\
&\le \frac{1}{M} \cdot \sum_{i=1}^{M} k_i \cdot \sum_{i=1}^{M} B_m(i) \\
&= C \cdot PCR_f \cdot B_p \\
&= B_{disk}(BFP_F)
\end{aligned}
$$

Thus, we conclude that $B_{disk}(BFP_V) \le B_{disk}(BFP_F)$.

We have proved that the BFP_V scheme uses less disk I/O bandwidth than the BFP_F scheme under the same disk cache space. Thus the BFP_V scheme also uses less memory cache space than the BFP_F scheme under the same disk I/O bandwidth. Because the staging buffer sizes used by different video objects in BFP_V vary based on the various PCR's, we also call this scheme *Variable Cache Size Buffer Management* scheme.

3.4 Materialization Rate Determination

In the previous section we have proved that BFP_V can save disk I/O bandwidth requirement for the video server if various PCRs are used for different video objects. Using the guidelines discussed previously, we have designed a *Materialize Rate Determination* algorithm, *MRD*, given in Figure 5. The following notations are used in the algorithm:

$Size(RAM)$:	The size of memory buffer.
$Size(O_i)$:	The size of object O_i.
$PCR(O_i)$:	PCR determined for object O_i.
$B_m(O_i)$:	Materialization rate for object O_i.
$Heat(O_i)$:	The access frequency of multimedia object O_i. We have $0 < Heat(O_i) \le 1$ and $\sum_{i=1}^{N} Heat(O_i) = 1$ where N is the number of objects in the database. Without loss of generality, we assume $Heat(O_1) \ge Heat(O_2) \ge \cdots \ge Heat(O_N)$.

Algorithm *MRD* determines the materialization rate for each multimedia object in the database according to its relative access frequency. For each object, *MRD* first determines the appropriate caching space for its C-fragments. A more frequently referenced object is allowed to cache a larger percentage of its data blocks, i.e., having larger C-fragments. As a result, this object needs a smaller materialization rate to support its pipelining. We note that this strategy follows the BFP_V scheme guidelines discussed previously. Our mathematical analysis ensures the BFP_V scheme uses less disk I/O bandwidth. But the analysis does not provide the amount of disk I/O bandwidth savings. This saving is depended on the materialization rate determination algorithm being used. We use simulation study to quantify the benefits of the BFP_V scheme, using algorithm *MRD* in the next section.

Algorithm: MRD

$SR = \text{Size}(\text{RAM})$

$TH = 1.0$

for $i = 1$ to N do

if $\left(\frac{\text{Heat}(O_i)}{TH} \cdot SR \geq \text{Size}(O_i) \right)$

then

$\text{PCR}(O_i) = 0$ /* The entire object O_i is cached */

else

$\text{PCR}(O_i) = 1 - \frac{\text{Heat}(O_i) \cdot SR}{TH \cdot \text{Size}(O_i)}$

$SR = SR - (1 - \text{PCR}(O_i)) \cdot \text{Size}(O_i)$

$B_m(O_i) = \text{PCR}(O_i) \cdot B_p$

$TH = TH - \text{Heat}(O_i)$

Figure 5: Materialization rate determination algorithm.

4 Performance Study

To quantitatively evaluate the benefits of BFP_V using algorithm *MRD*, we have developed a detailed simulator to compare its performance with the BFP_F scheme. We investigate the disk I/O bandwidth savings contributed by the memory buffer management and compare the performance of those two schemes with that of no memory buffer situation. In this section, we first describe the simulation model, and then discuss the simulation results.

4.1 Simulation Model

Our simulator consists of three components. The structure of the simulator is depicted in Figure 6. The *Request Generator* is responsible for creating requests. Each request is characterized by its arrival time and the object requested. Requests arrive at the FIFO queue where they wait for admission. The *Video Server* schedules the requests in the queue in FCFS manner. When disk I/O bandwidth becomes available for serving the pending request at the head of the queue, the server accepts the request and creates a video stream for this request. The server reserves the requested disk I/O bandwidth till the video playback completes. The requested disk I/O bandwidth for each individual object is determined by the PCR used by this object. In the BFP_F scheme, all objects use the same PCR. That means memory buffer rate for every objects are the same. So the PCR used by the BFP_F scheme is

$$\text{PCR} = 1 - \frac{\text{Memory buffer space}}{\text{Database size}}.$$

In the BFP_V scheme, the PCR and requested disk I/O bandwidth are determined by algorithm *MRD*.

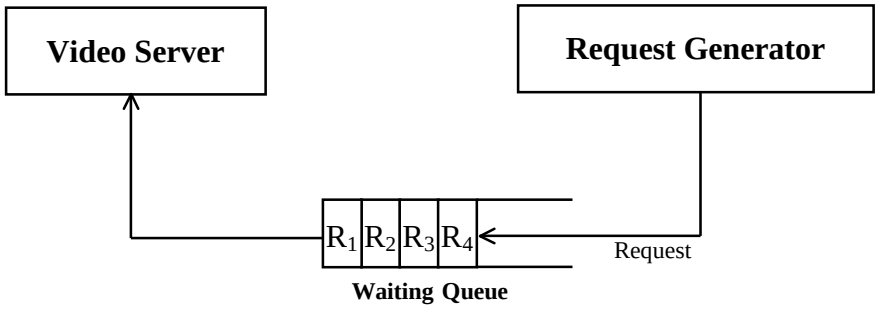

Figure 1: Structure of the simulator.

There are many ways to evaluate the performance of the data caching scheme for memory buffer management. However, the ultimate measurement for a video system is to measure the number of concurrent streams supported by the system. We define that a data caching scheme is better than a second one if the video server can support more concurrent streams using the first data caching scheme under the same hardware (disk I/O bandwidth and memory buffer size). So we use the maximum number of concurrent streams supported as our performance metric. In terms of workloads, we assume all video objects are MPEG-1 encoded with a playback rate of 1.5 Mbits/sec. The video database consists of 1000 video objects with their playback duration uniformly distributed between 10 minutes and 20 minutes. That means their sizes vary between 112.5 MB and 225 MB. User request inter-arrival time is modeled using a Poisson process with the average inter-arrival time varying between 0.5 and 5 seconds in different simulations.

For each simulation, we ensure the user request inter-arrival time is small enough so that the FIFO queue always has requests available whenever the video server is ready to accept a request. The access frequencies of objects in the database follow a Zipf-like distribution. The access frequency for each video object, O_i, is determined as follows:

$$f_i = \frac{1}{i^z \cdot \sum_{j=1}^{n} \frac{1}{j^z}},$$

where n is the number of objects in the system, and $z\ (0 \le z \le 1)$ is the Zipf factor [14]. A larger z value corresponds to a more skew condition, i.e., some objects are accessed considerably more frequently than other objects. When $z = 0$, the distribution is uniform, i.e., all the objects have the same access frequency. A study conducted in [6] and confirmed in [14] has shown that the video rental pattern in video store follows the Zipf-like distribution with a Zipf factor $z = 0.7$.

We summarize the simulation parameters in Table 2. We use those parameters to perform our simulation unless otherwise stated. We vary the memory buffer space and disk I/O bandwidth to demonstrate how the memory buffer space and disk I/O bandwidth affect the system performance.

Table 2: Simulation parameters.

Playback rate B_p	1.5 Mbits/Sec
Memory buffer space	9% of database size
Disk I/O bandwidth	30 Mbytes/Sec
Zipf factor	0.7
Number of objects	1000
Minimum object size	112.5 MB (10 minutes)
Maximum object size	225 MB (20 minutes)
Number of requests	20,000

4.2 Performance Under Various Disk I/O Bandwidth

The disk I/O bandwidth is the decision factor of the system performance in terms of maximum concurrent streams supported. The number of concurrent streams supported by the system is proportional to the disk I/O bandwidth. In this simulation study, we fix the memory buffer space at 9% of the entire database size and vary the disk I/O bandwidth between 10 MB/sec and 50 MB/sec. The simulation results are plotted in Figure 7.

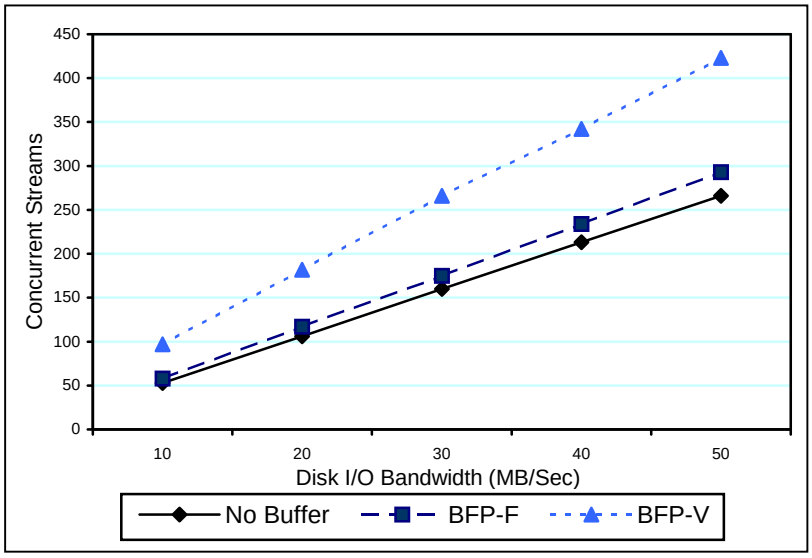

Figure 7: Performance under various disk I/O bandwidth.

Since the memory buffer space is fixed at 9% of the database size, the PCR used for BFP_F should be 0.91. Hence the disk I/O bandwidth saving using BFP_F will be about 9%, comparing to the no memory buffer situation. Our simulation results show that the BFP_V scheme dramatically increases the system performance over the BFP_F scheme and the no memory buffer situation. Even when the disk I/O bandwidth is very small, the performance improvement using the BFP_V scheme is outstanding. When the disk I/O bandwidth is only 10 MB/sec, the BFP_V scheme outperforms the BFP_F scheme significantly. In this case, the

system can support 97 concurrent streams using the BFP_V scheme, comparing to supporting 58 concurrent streams using the BFP_F scheme. The number of concurrent streams supported increases by 39 using BFP_V, comparing to using BFP_F. In the meantime, the system can only support 53 concurrent streams without memory buffer. When the disk I/O bandwidth increases to 50 MB/sec, the system can support 423 concurrent streams using the BFP_V scheme while it can merely support 293 concurrent streams using BFP_F. The number of concurrent streams supported increases by 130 using the BFP_V scheme.

4.3 Effect of Memory Buffer Space

Caching video data in the memory buffer reduces the disk I/O bandwidth requirement. Increasing the memory buffer space should improve the system performance. In this simulation study, we fix the disk I/O bandwidth at 30 MB/sec. By varying the memory buffer space from 3% to 15% of the database size, we plot the performance results in Figure 8.

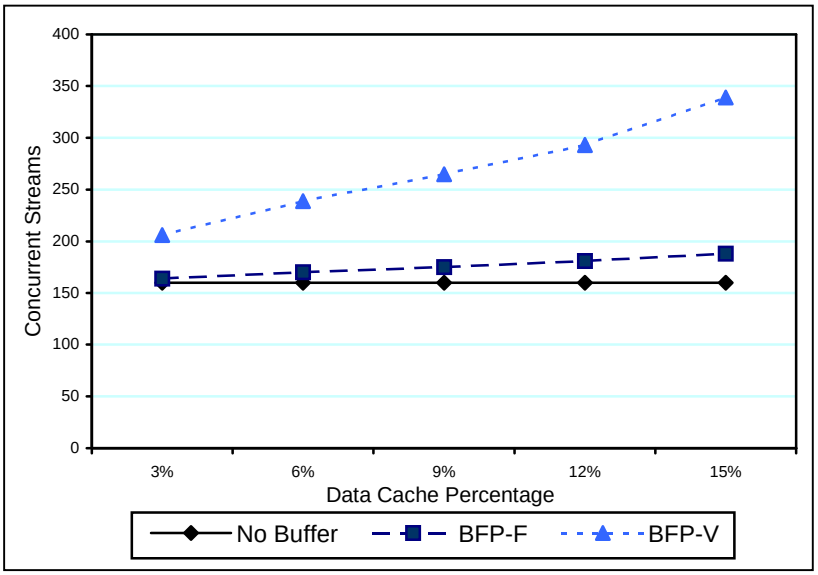

Figure 8: Performance under different cache space.

As we expected, the increase in memory buffer space improves the system performance under the BFP_F and BFP_V schemes. The performance improvement using the BFP_F scheme is proportional to the increase of memory buffer space, while the BFP_V scheme improves over the BFP_F scheme by smartly managing the memory cached video data. Increasing the memory buffer space gives more room for the BFP_V scheme to manipulate data and hence enhance its performance over the BFP_F scheme. Our simulation results confirm this expectation. When the memory buffer space is only 3% of the database size, the BFP_V scheme outperform the BFP_F scheme by only 42 concurrent streams. When the memory buffer space increases to 15% of the database size, the BFP_V scheme display a 151

concurrent stream increase over the BFP_F scheme. In the meantime, the system performance using the BFP_V scheme is 179 concurrent streams better than that not using memory buffer which can only support 160 concurrent streams.

5 Memory Cache Replacement Scheme

Algorithm *MRD* determines the individual materialize rates based on the access frequency of each object. The access frequency of an object changes from time to time. For instance, in a News-On-Demand system, new headlines are accessed more than the old ones. Some special events, such as Olympics, attract more people to request the related video objects. In a Video-On-Demand system, cartoons are very popular in the United States between 4 PM and 7 PM after children have returned home from schools. Also, adults typically watch movies between 7 PM and 10 PM. Usually those user access patterns are predictable [5]. To efficiently manage the system resources (memory buffer space and disk I/O bandwidth), we need to detect the access frequency changes of the video objects. Based on the new access frequencies of the objects, we re-evaluate the materialization rates for all video objects using algorithm *MRD* and hence determine the C-fragment and D-fragment sizes for every object.

Similar to the LRU algorithms where the least recently accessed pages are replaced, the pages to be replaced are in those less frequently accessed objects in our memory cache replacement scheme. However, within the object, the replaced pages might be the most recently accessed pages. Essentially our replacement scheme not only considers the access frequency of the objects but also considers the location of the pages in the object context. This context sensitive replacement makes our scheme superior to the traditional LRU replacement algorithms. When an object becomes less frequently accessed, we reduce the size of its C-fragments, i.e., some pages in each C-fragment are cast out of memory. To simplify our presentation, we use an example to demonstrate our cache replacement scheme. As we discussed before, each object is fragmented into many processing units, i.e., pairs of D, C fragments. We assume each playback unit consists of *G* pages as depicted in Figure 9. There are two kinds of pages in those *G* pages. D-pages form the D-fragment and C-pages form the C-fragment.

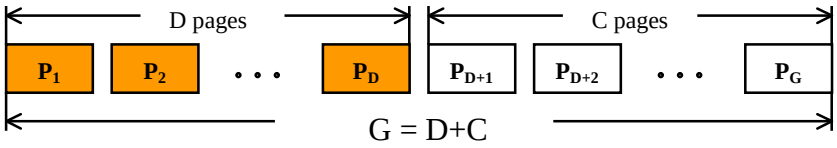

Figure 9: A G-page processing unit.

For each object, the sizes of the D-fragment and C-fragment in each processing unit are dynamically adjusted by converting the D-pages into C-pages and vice versa. For instance, if *G* is chosen to be 10, 11 combinations of D-fragment and C-fragment depicted in table 5 can be supported. We note that *Combination 0* corresponds to the case when the entire object is kept in the memory buffer. On the contrary, *Combination 10* corresponds to the case when

caching is not used. When the system determines some object *O* becomes less frequently accessed, the C-fragment size of this object has to be reduced by converting some C-pages into D-pages.

Table 3: Combinations of D-fragment and C-fragment.

Combination	0	1	2	3	4	5	6	7	8	9	10
PCR	0	1/10	2/10	3/10	4/10	5/10	6/10	7/10	8/10	9/10	1
D-fragment	0	1	2	3	4	5	6	7	8	9	10
C-fragment	10	9	8	7	6	5	4	3	2	1	0

Let us say the system has determined object *O* to be changed from combination $Comb_i \equiv (\mathrm{D} = i, \mathrm{C} = \mathrm{G} - i)$ to combination $Comb_j \equiv (\mathrm{D} = j, \mathrm{C} = \mathrm{G} - j)$ where $i < j$. Then the following pages can be replaced by the system:

$$Free(i, j) = \{p_k \mid i < k \bmod G \le j\}$$

To illustrate this strategy, let us consider the following example: an object *O* is currently changing from combination $Comb_3 \equiv (\mathrm{D} = 3, \mathrm{C} = 7)$ to combination $Comb_6 \equiv (\mathrm{D} = 6, \mathrm{C} = 4)$. ($Comb_3$ and $Comb_6$ correspond to $\mathrm{PCR} = 0.3$ and $\mathrm{PCR} = 0.6$, respectively.) In this case, the buffer space occupied by the following set of C-pages can be returned to the buffer pool:

$$Free(3,6) = \{p_{04}, p_{05}, p_{06}, p_{14}, p_{15}, p_{16}, p_{24}, p_{25}, p_{26}, \cdots\cdots\}$$

On the other hand, when system detects some object *O* becomes more frequently accessed, we have to increase the C-fragment size for this object to reduce the disk I/O requests. That means some D-pages will be converted into C-pages. In this case, object *O* is changed from combination $Comb_i \equiv (\mathrm{D} = i, \mathrm{C} = \mathrm{G} - i)$ to combination $Comb_j \equiv (\mathrm{D} = j, \mathrm{C} = \mathrm{G} - j)$ where $i > j$. Then the following pages have to be cached in the memory buffer:

$$Cache(i, j) = \{b_k \mid j < k \bmod G \le i\}$$

In this case, we can use the object *O* changing from combination $Comb_6 \equiv (\mathrm{D} = 6, \mathrm{C} = 4)$ to combination $Comb_3 \equiv (\mathrm{D} = 3, \mathrm{C} = 7)$ as the example. Thus the following pages have to be cached:

$$Cache(6,3) = \{p_{04}, p_{05}, p_{06}, p_{14}, p_{15}, p_{16}, p_{24}, p_{25}, p_{26}, \cdots\cdots\}$$

6 Conclusion and Future Works

To manage the memory buffer for the video objects, we propose a *Bi-directional Fragmental Pipelining* (BFP) technique and its variable buffer size data caching scheme, BFP_V, to reduce the disk I/O bandwidth requirement for video servers. Our analysis shows the proposed *Bi-directional Fragmental Pipelining* (BFP) technique and related data caching scheme for memory buffer management is natural for video objects. Not only does it provide excellent memory buffer management for video servers, but also it is able to efficiently implement the

special video functions such as Fast-Forward and Fast-Reverse. Most importantly, it significantly saves the disk I/O bandwidth by caching a very small portion of video fragments. Our simulation results show the BFP_V scheme improves the system performance by 26% using a surprisingly small memory buffer space (3% of the database size), comparing to no memory buffer situation. When memory buffer space is 15% of the database size, the system performance improvement over no memory buffer case is 112% using the BFP_V scheme. Overall the BFP_V scheme is significantly better than the BFP_F scheme. Based on the proposed BFP_V scheme, we have discussed the memory cache replacement method for video objects. The memory pages to be replaced are determined not only by the access frequency of the related object but also by the context of the pages in the object. However, this memory cache replacement scheme is based on the access frequencies of the video objects. Those access frequencies are average values over a certain period of time. In a more dynamic environment, sometimes the access frequencies of the video objects change more often. The average over a long duration of time might not reflect the conditions of its constituent periods. Although our simulation study shows that the memory buffer management based on average access frequencies significantly improves the system performance, we are still seeking a method that can dynamically adjust the C-fragment and D-fragment of each object to achieve the optimal system performance. The system cost issue of the video server under the BFP_V scheme is another topic for future investigation.

Acknowledgment

The authors acknowledge the partial support provided by the Army Research Office (Contract No. DAAH04-95-1-0250). The views and conclusions herein are those of the authors and do not represent the official policies of the funding agency.

Reference:

[1] S. Berson, S. Ghandeharizadeh, R. Muntz, and X. Ju (1994). Staggered striping in multimedia information systems. *In Proc. Of ACM SIGMOD Conference*, pages 79-90, May 1994.

[2] E. Chang and H. Garcia-Molina (1998). Cost-based media server design. *In Proc. Of the International Conference on Multimedia Computing and Systems*, pages 76-83, September 1998.

[3] T.-S. Chau, J. Li, B. Ooi and K.-L. Tan (1996). Disk striping strategies for large video-on-demand servers. *In Proc. Of ACM Multimedia*, pages 297-306, 1996.

[4] M. Chen, D. Kandlur and P. S. Yu (1993). Support for fully interactive playout in a disk-array-based video server. *In Proc. Of ACM Multimedia*, pages 391-398, 1994.

[5] Tae uk Choi, Young-Ju Kim and Ki-Dong Chung (1999). A prefetching scheme based on analysis of user pattern in news-on-demand system. *In Proc. Of ACM Multimedia*, pages 145-148, October 1999.

[6] A. Dan, D. Sitaram and P. Shahabuddin (1994). Scheduling policies for an on-demand video server with batching. *In Proc. Of ACM Multimedia*, pages 15-23, October 1994.

[7] Dey-Sircar et al. (1994). Providing VCR capabilities in large-scale video servers. *In Proc. Of ACM Multimedia*, pages 25-32, October 1994.
[8] C. S. Freedman and D. J. DeWitt (1995). The SPIFFI scalable video-on-demand system. *In Proc. Of the SIGMOD conference*, pages 352-363, San Jose, CA, May 1995.
[9] Henry Lau (1998). Microsoft SQL server 7.0 performance tuning guide. *Technical Report 098-81529*, http://www.microsoft.com/sql, 1998.
[10] F. Moser, A. Kraiβ and W. Klas (1995). L/MRP: A buffer management strategy for interactive continuous data flows in a multimedia DBMS. *In Proceedings of the 21st VLDB conference*, pages 275-286, Zurich, Switzerland, September 1995.
[11] Raymond T. Ng and Jinhai Yang (1994). Maximizing buffer and disk utilizations for news-on-demand. *In Proceedings of the 20th VLDB conference*, Santiago, Chile, 1994.
[12] Martin Reisslein, Keith W. Ross and Subin Shrestha (1999). Striping for interactive video: Is it worth it?. *In Proc. Of the International Conference on Multimedia Computing and Systems'99*, volume 2, pages 635-640, Florence, Italy, June 1999.
[13] Doron Rotem and J. L. Zhao (1995). Buffer management for video database systems. *In Proc. Of the Int'l Conf. On Data Engineering*, pages 439-448, Taipei, Taiwan, March 1995.
[14] James Z. Wang, Kien A. Hua, and Honesty C. Young (1996). SEP: a space efficient pipelining technique for managing disk buffers in multimedia servers. *In Proc. of IEEE Int'l Conf. on Multimedia Computing and Systems*, pages 598-607, Hiroshima, Japan, June 1996.
[15] Y. Wang and D. Du (1997B). Weighted striping in multimedia servers. *In Proc. Of the International Conference on Multimedia Computing and Systems'97*, pages 102-109, Ottawa, Ontario, Canada, June 1997.