Continuous Display of Presentations Sharing Clips*

Cyrus Shahabi and Shahram Ghandeharizadeh

Technical Report: *USC-CS-94-587*

Abstract

Databases were introduced to remove redundancy from conventional file systems to encourage *sharing* of information. The same idea is extended in this study to support sharing for continuous media data types (i.e., video and audio). Sharing in conventional databases results in update anomalies when information is modified. With presentations (movies) sharing clips (sequence of frames), continuous display becomes challenging as well. To ensure a continuous display, a system should retrieve data at a pre-specified rate. Otherwise, a display might suffer from disruptions or delays, termed *hiccups*. To ensure a continuous display using a multi-disk hardware platform, a video object is striped into a number of subobjects. The system enforces a regular schedule on retrieval of each subobject by controlling the placement of the subobjects across the disks. Now if different presentations share subobjects, each presentation will enforce its own restrictions on the placement of the data. This might result in irregular schedules for alternative presentations, resulting in hiccups. One approach to this problem is to replicate the shared subobjects as many times as they appear in different presentations. An alternative is to maintain these subobjects memory resident. Due to read-only characteristic of the application, *flash* memory is appropriate for storing these subobjects. These alternative approaches result in different storage costs. In this paper we investigate each approach as well as a wide spectrum of hybrid approaches that minimize the storage cost while ensuring a continuous display. In addition, this study investigates two alternative paradigms to support multiple users: Demand and Data driven. In the former, the display of a DAG starts once a request references it. In the latter, all possible *paths* of a DAG are displayed periodically. A price analysis is provided for each paradigm in order to configure the cheapest storage structure.

*A version of this paper appeared in the *ACM/Springer-Verlag Multimedia Systems Journal, v3, n2, May 95.* This research was supported in part by the National Science Foundation under grants IRI-9203389, IRI-9258362 (NYI award), and CDA-9216321, a Hewlett-Packard unrestricted cash/equipment gift.

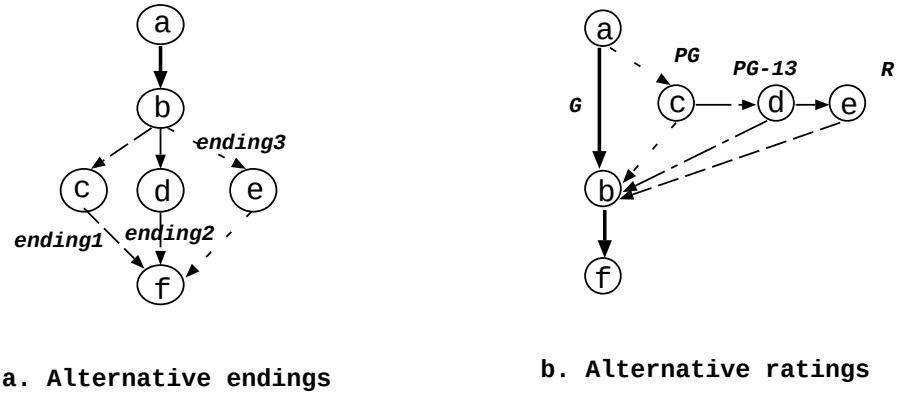
1 Introduction

Consider a *database* consisting of a number of digitized presentations. Each *presentation* consists of a sequence of *video clips*, where a clip is a sequence of *frames*. Alternative presentations might share clips. This sharing is currently demonstrated by applications from the entertainment industry. For example, three different versions of the movie Clue were distributed, each with a different ending in order to maintain the audience's suspense. Similarly, music channels (e.g., MTV, VH-1) produce shows where a single music video clip is presented in each show at a different time, e.g., while the video clip might be presented at the beginning of one show, in a second show the same video might be presented ten minutes into the show. Figure 1.a demonstrates different versions of a movie conceptualized as a Directed Graph (DG). In this case, audience(s) after watching the sequences a and b of video clips, can potentially proceed to c, d or e for three different endings. Note that clips have variable sizes and hence variable durations. For example, the display of c might require 5 minutes while the duration of d might be 2 minutes. The sequence f is the epilogue of the movie. A DG might have branches in the middle as well. To illustrate, consider Figure 1.b as a movie with four different ratings (i.e., G, PG, $PG-13$, R). An adult audience may select to preview sequences c, d, and e, while children are restricted to only watch sequence c. In general, a DG might be a combination of **different** movies sharing many clips.

We use the object-oriented terminology in order to abstract out the structure of a presentation[1]. A movie can be viewed as a collection of video clips. Each clip is an *object*, where objects have variable sizes (clips have different durations). Moreover, these objects are members of continuous media data type and should be retrieved and displayed at a pre-specified continuous rate. To illustrate, a video object based on NTSC (network quality) should be retrieved and displayed at a rate of 45 megabits per second (mbps) [Has89]. (Lossy compression techniques can be employed to reduce both the size of video objects and their bandwidth requirements, see [Fox91] for an overview.) A set of presentations sharing objects are represented as a Directed Graph (DG). Each vertex in the graph represents an object while an edge represents a link from one object to another. Thus, an object may lead to several different objects and be reached via different objects. Each path of the DG corresponds to a single presentation. Since, we expect finite presentations, we do not allow cycles and focus on Directed Acyclic Graphs, DAG. Finite number of loops in a cycle can be represented by instantiating each cycle in a DAG. The user may select a path either before the display of a DAG has started (termed *static*) or interactively while viewing a clip (termed *interactive*). To separate the storage manager (which is the focus of this study) from the user interface, this study concentrates on static presentations. However, to make the problem more challenging and in order to extend the solutions for a special type of interactive presentation in Section 6, we assume no advance knowledge about the users' decisions. Given a DAG, a link from object X to object Y implies that the display of Y should start immediately after the display of X ends. This is identical to the *meet* temporal relationship found in [All83]. To simplify the problem, this study considers only these two simple relations by using a DAG to represent the database. However, the extension to support other relations defined in [All83] is straightforward. This is described further in Section 6.

In this study, we concentrate on the **placement** of a DAG on a hierarchy of storage media in order to guarantee a continuous display for all the plausible paths of the DAG. One approach to store multiple presentations sharing objects (clips), is to replicate each clip as many times as the clip appears in different presentations. An alternative approach is to maintain the clip memory resident. Both approaches might result in storage space requirements that is not economically justified. Based on a number of parameters, it might be more appropriate to replicate some of the clips while storing others in memory (a combined approach). Moreover, it might be cheaper to store (or replicate) a portion of a clip on magnetic disk drive(s) while the rest resides in memory (a hybrid approach). In each case, the system should ensure a continuous display for all the presentations sharing those objects. This study investigates alternative object placement strategies in order to satisfy two major objectives. First, **guarantee** a continuous display of all the presentations in a database. Second, minimizing the cost of storage space required by the database.

[1] The terms *movie* and *presentation* are used as synonyms in this paper.



a. Alternative endings **b. Alternative ratings**

Figure 1: Alternative versions

We investigate two alternative paradigms for displaying a DAG: Demand and Data driven. With the demand driven paradigm, the system displays a presentation when a user references that presentation. An example of this paradigm is our everyday use of a VCR device. Using this device, a user display a movie whenever desirable. Its primary advantage is a low latency time. However, in a multi-user environment, to maintain the low latency time, there should be a one-to-one relationship between the active users and available resources (one VCR per user). If the number of resources is less than the number of active requests then a queue of pending requests is formed, causing each queued request to observe a latency time. (The average latency is dependent on the display time of the object referenced by each request and the number of pending requests.)

With the data driven paradigm, the data is retrieved and displayed periodically. The *pay per view cable service* is an example of this paradigm. A single movie is displayed periodically (say once every two hours), regardless of the number of requests. This paradigm may suffer from a high average latency time (maximum of two hours) observed by those requests arriving after the display has started. However, a single resource (a cable channel in this example) is sufficient to service a large number of users, theoretically an infinite number of users[2]. To reduce the latency time in this paradigm, one can increase the number of resources (e.g. two cable channels) and display the movie with different starting time (say one hour apart) on each channel. In this case, the user observes a one hour latency time by switching channels. We describe the placement of the objects in a DAG for each paradigm that realizes both of our objectives.

Recently, a number of studies have investigated retrieval of presentations consisting of continuous media in order to support their hiccup-free display [BGMJ94, AH91, RVR92, RV93, CL93, TPBG93, RW94]. We will extend one of the proposed techniques, namely *simple striping* [BGMJ94] to support a hiccup-free display of a DAG. To realize a system that is economically viable, these studies have focused on a storage hierarchy that consists of: dynamic random access memory (DRAM), and magnetic disk storage. DRAM provides a short latency time (typically 60 nano-seconds), a high bandwidth (500 megabytes per second for Synchronous DRAM [PNHV92]) and is fairly expensive ($30 per megabyte of storage at the time of this writing). Magnetic disk drives are mechanical devices that provide a cheap form of storage ($0.6 per megabyte of storage). However, upon the arrival of a request, they incur a high latency time (typically 10 to 15 milli-seconds) and provide a low bandwidth (3 to 5 megabytes per second). We extend this hierarchy with solid state memory [DH93] (termed *flash*) due to both its increased popularity and suitability of its physical characteristics to this application. Flash memory provides a transfer rate and a latency time almost identical to DRAM. It differs from DRAM as follows: 1) it is cheaper, 2) is suitable for read-only data; the write operations on flash are slow, moreover, they slow down the read performance of this medium if performed frequently[3]. As compared to magnetic disk storage, the low latency of flash (with 60 to 120

[2]There are other differences between the two paradigms, such as the ability to control the display (e.g., fast forward). These features do not constitute the focus of this study.

[3]Note that DRAM cannot be replaced with flash because the data must be temporarily staged somewhere before its display,

nano-seconds read-access-time [DH93]) and its high transfer rate allow it to be multiplexed among multiple requests for data retrieval while ensuring a continuous display for each request. The read-only characteristic of a presentation render the use of flash memory attractive.

In this paper, we report performance results on neither a hiccup-free display nor a low storage cost. As far as the first objective is concerned, the designs are constructed using simple striping. As shown in [BGMJ94], the placement of objects using simple striping guarantees a hiccup-free display. For the second objective, we propose analytical models that quantify the storage cost based on the characteristics of an application and system parameters. These models can be employed for a variety of applications. The results obtained for one application is specific to that application and cannot be generalized to others.

2 Overview

Placing a combination of different objects with different bandwidth requirements (forming a directed graph) on a multi-disk environment is an open research area. In this paper, the following assumptions are made to simplify the problem:

1. A directed graph has no cycles (i.e., Directed Acyclic Graph (DAG)).
2. A DAG has a single *source*. In practice, this can be achieved by adding a common title to the beginning of any DAG. Note that this assumption is made to simplify the description of the techniques. In general, a user can display a presentation starting from any of its objects.
3. We concentrate on a single DAG. For multiple DAGs, if they do not share objects then each of them can be considered separately. Otherwise, an artificial *super_source* is used to reduce the problem to a single DAG.
4. Bandwidth required to display an object remains unchanged during its display.
5. Bandwidth required to display each object in a DAG is identical. If this is not true, *staggered striping* [BGMJ94] should be employed instead of simple striping. This has no impact on the proposed solutions and cost functions.
6. The bottleneck resource is the disk subsystem. We assume that the bandwidth of both the network and connection buses is significantly higher than the bandwidth of the disk subsystem.
7. Price per megabyte of flash memory is lower than that of DRAM [Wal94] (otherwise, replace flash with DRAM).
8. The focus is to display a presentation consisting of video objects. Features such as pause, fast-forward, and rewind constitute our future research direction.

Now consider the display of a DAG with each of the demand and data driven paradigms. With the demand driven paradigm, although a single resource per active request is still sufficient to display the DAG, a number of active requests arriving at the same time cannot be grouped together. This is because each user might elect to view a different presentation, pursuing a different path. Obviously, this is impossible in the presence of a single resource (VCR). Hence, the number of active requests is a function of the available bandwidth and a fraction of the bandwidth required by each request; otherwise, a latency time is observed. This latency time is independent of the number of alternative paths in the DAG, and is a function of the display time of the DAG and the number of pending requests. Similarly, employing the data driven paradigm requires more resources to display a DAG as compared to that of a single movie. Consider a DAG with p alternative paths. To display this DAG (say in a pay per view cable service), p cable channels (resources) are

using flash for this purpose is a mistake because frequent write operations to this medium renders it useless.

Subobject:	a stripe of an object. It represents a contiguous portion of the object. Its size is determined at system configuration time.
DAG:	a Directed Acyclic Graph, used to present a database of presentations sharing clips.
Channel:	A fraction of available disk bandwidth, required to display an object.
Platform:	a physical storage with sufficient bandwidth to support the display of at least one subobject.
Time Interval:	The duration of time required to display a subobject from a channel.

Table 1: Defining terms

Term	Definition
p	Number of all possible paths in a DAG
$Max\_Display$	The duration of the longest path of a DAG
$B_{Display}$	Bandwidth required to display an object
B_{Disk}	Disk Effective Bandwidth ($B_{Disk} = tfr \times \frac{size(subobject)}{size(subobject) + (tfr \times (max\ seek + max\ latency))}$ where tfr is the transfer rate of the disk drive. see [BGMJ94])
M	Degree of declustering ($= \frac{B_{Display}}{B_{Disk}}$)
R	Number of channels
D	Number of disk drives
$H(x)$	Optimal head size (flash resident portion) of object x
$\mathcal{C}(x_i)$	The channel number x_i resides on
$\mathcal{C}(x_i, j)$	The channel number x_i is displayed from when interval j is employed to display x
$\|DAG\|$	Number of objects in a DAG
$\$flash$	Price of flash storage per megabyte in \$
$\$disk$	Price of Disk storage per megabyte in \$
$\$ram$	Price of DRAM storage per megabyte in \$
k	Number of groups of p displays with data driven

Table 2: List of terms used repeatedly in this paper and their respective definitions

required, each to display one path. Note that the maximum latency time is now the duration of the longest path ($Max\_Display$) of the DAG. One may reduce the latency time by increasing the number of resources as a multiple of p.

In order to describe the details of these two paradigms, we need to focus on a technique that retrieves video (and audio) objects in a manner that supports its continuous display. We have elected to focus on simple striping. This technique minimizes the amount of memory required to display an object by establishing a pipeline between the magnetic disk drive(s) and a display station. Since the bandwidth of the disk drives is the bottleneck resource, it partitions the aggregate bandwidth of the available disks into R channels (C_1, C_2, ..., C_R) in order to support R simultaneous displays. In order to distribute the load of a display referencing an object x across the channels, it stripes object x into $\frac{\text{SIZE}(x)}{\text{SIZE}(subobject)}$ subobjects, and assigns the subobjects of x to the channels in a round-robin manner. (The size of each subobject is fixed for all objects and determined at system configuration time[4].) To illustrate, in Figure 2.a, 9 subobjects of x (x_1, x_2, ..., x_9) are assigned to an 8 channel system, starting with C_1. To ensure a continuous display, the bandwidth

[4]This size depends on system available memory size, magnetic disk drive characteristics, and some other factors which is beyond the scope of this paper.

Figure 2: Logical channels in simple striping

of a channel should be equal to the display bandwidth of the objects ($B_{Display}$), minimizing the amount of memory required by pipelining the data from the disk to a display. Hence, denoting the **effective** disk bandwidth (considering the maximum seek and latency time, see [BGMJ94]) with B_{Disk}, if $M = \frac{B_{Display}}{B_{Disk}}$, then the number of channels (R) is computed as:

$$R = \frac{D}{M} \tag{1}$$

where D is the number of disk drives. Moreover, R determines the number of simultaneous requests supported by the system.

When $M < 1$, each channel provides a fraction of a disk bandwidth (see Figure 2.a). If $M \geq 1$ then each subobject is declustered [GRAQ91] into M fragments and the aggregate bandwidth of M disk drives determine the required bandwidth of a channel (see Figure 2.b). In Figure 2.b, $x_{i.j}$ represents the jth fragment of the ith subobject of x. To ensure a continuous display of an object x, the display of a fraction of x_i is overlapped with the retrieval of a portion of x_{i+1} (for details see [BGMJ94]). The duration of the retrieval of a subobject is fixed for all subobjects regardless of their media type, and is termed a **Time Interval**. When the system displays x, it employs a single channel during each time interval. Hence, the maximum number of available Time Intervals is equal to the number of channels and defines the maximum number of displays that can be supported simultaneously ($= R$). Each display iterates over the available channels employing each in a round-robin manner in order to distribute its work evenly across the channels. The round-robin scheduling of channels avoids multiple requests from colliding with one another once their display has been initiated. To illustrate, in Figure 2.a, assume a time interval that corresponds to C_1 is available. Employing that interval, x_1 can be displayed. In the next interval the system employs C_2 to retrieve and display x_2. This process repeats itself until x is displayed entirely. At the same time, $R - 1$ other intervals can be utilized to service $R - 1$ other requests. Therefore, to display x (assuming only one time interval is available), in the worst case $(R - 1) \times Time\_Interval$ seconds are required until the idle interval reaches C_1, where $Time\_Interval$ is the duration of a time interval in seconds.

Let a $platform$ define a physical storage with sufficient bandwidth to display at least one subobject. For example, in Figure 2.a, a platform consists of a single disk drive, while in Figure 2.b, a cluster of four disk drives constitute a platform. When $M \geq 1$ there is a one-to-one relationship between a platform and a channel. This relationship is one-to-many when $M < 1$. For instance in Figure 2.a (where $M < 1$), a subobject placed on platform 1 can be displayed from C_1, C_2, C_3 and C_4; whereas, in Figure 2.b (where $M \geq 1$), x_1 which is placed on platform 1 can only be displayed from C_1. For the rest of this paper we assume $M \geq 1$. This assumption is made not to simplify the problem, but to consider the situation which introduces the most challenging problems. To observe, lets illustrate the placement of a DAG with simple striping. In Figure 3, both DAGs consist of three objects, x, y and z where each has a different size. Figure 3.a demonstrates the placement of a simple DAG for a system that consists of eight channels, $R = 8$. In this case, a time interval might be reserved by a user (say U_{xy}) who decides to display y after x (path xy). Another user may desire to display z after x (U_{xz}). To avoid hiccups for either user, once the display of x is

Figure 3: Placement of DAG with simple striping

complete, the first subobject of both y and z are placed on the channel following the one that contains the last subobject of x. However, this may not be possible for all DAGs. For example, the extended DAG of Figure 3.b is defined such that users are allowed to display z after both x and y. If z starts from C_2 (see the first placement in Figure 3.b), a user choosing path xyz might observe a hiccup because the regular schedule of simple striping dictates that this display employs C_4 while the display needs C_2 to satisfy the user and C_2 might be busy servicing other users. On the other hand if the display of z starts from C_4 (see the second placement in Figure 3.b), then a user choosing path xz might observe a hiccup because another request might be using C_4 when the current request is done with the display of x_9 using C_1. Hence, to be able to schedule either time intervals (employed by U_{xyz} or U_{xz}) regularly in order to display z immediately after x or y, it should be possible to display z_1 using both C_2 and C_4. If $M \geq 1$, C_2 and C_4 correspond to two different platforms (due to a one-to-one relationship), requiring z_1 to be placed on both platforms[5]. However, when $M = 0.25$, C_2 and C_4 might both correspond to a single platform, say P_i (as in Figure 2.a). Hence, placing z_1 on P_i will simply ensure continuous display of the DAG independent of users' decision. Note that if C_2 and C_4 correspond to two different platforms, then the problem would be identical to the case when $M \geq 1$. Since problems for $M < 1$ is a subset of problems for $M \geq 1$, the techniques described in this study can be applied to $M < 1$ as well. However, taking advantage of the flexibilities of $M < 1$, the proposed techniques can be optimized further. For example, one might first detect if two different channels correspond to a single platform. If so, then there is no constraint on object placement; otherwise, the problem is solved similar to the case when $M \geq 1$. We consider these optimizations as implementation techniques and will not consider $M < 1$ any further. However, if a single disk is partitioned into regions or zones (restricting the assignment of an object on a single disk drive) in order to reduce the seek time or maximize the utilization of the disk bandwidth [GKS95, GKS94], the problem for $M < 1$ becomes equivalent to that of $M \geq 1$.

Since there is always a one-to-one relationship between a platform and a channel when $M \geq 1$, the terms *channel* and *platform* are used interchangeably for the rest of this paper. Hence, we use C_i for both channel i and the platform corresponding to channel i.

[5]Due to this constraint, one might replicate z_1 twice on the two platforms. This strategy and some other alternatives are discussed in Section 3.1.

Our proposed solutions to the above problem include intelligent placement of the objects, replication, using DRAM as a staging area and maintaining a portion of objects flash resident. The most trivial solution is to replicate the object as many times as it appears in a path. An alternative is to use flash memory. The read-only characteristic of the application encourages the use of flash memory due to its physical characteristics as discussed in Section 1. The flash memory is used to store the first portion of an object, termed *head*, while the rest resides on the disks. In this case, independent of the location of the free interval, the display of the object starts immediately. Meanwhile, the free interval is employed to retrieve the rest of the object that is disk resident into the DRAM. Once the display of *head* ends, enough data is accumulated in DRAM in order to ensure a hiccup-free display. To reduce the amount of flash and DRAM requirement, objects should be placed intelligently on the disk channels. In order to decrease the total storage cost further, a combination of replication and flash resident portion might be appropriate. In this paper, these alternatives are investigated in detail for both the demand and data driven paradigms.

The rest of this paper is organized as follows. Section 3 describes the demand driven paradigm. It introduces four alternative object placement strategies and provides a price analysis in order to choose the cheapest alternative. The data driven paradigm is described in Section 4. It demonstrates that the same placement strategies can be employed here with some modifications. In Section 5, the two paradigms are compared with each other and a system that combines the two paradigms are discussed. The option of maintaining a DAG entirely flash resident is also briefly investigated. Section 6 concludes this paper and lists our future research directions.

3 Demand Driven Paradigm

With the *Demand Driven paradigm*, once a request arrives, an empty time interval is assigned to that request. The request occupies this time interval for the entire display time of the user's desired path. Therefore, in a R channel system, at most R requests can be serviced simultaneously[6]. The $(R+1)th$ request should wait until a time interval dedicated to an active request becomes available (the active request terminates), resulting in the worst latency time of $Max\_Display$ (the duration of the longest path[7]). If the number of requests in every $Max\_Display$ seconds exceeds R, a request queue is formed and the latency time becomes a function of the number of pending requests and $Max\_Display$. This paradigm is independent of the complexity of the DAG. That is, once a time interval is assigned, regardless of the number of alternative links chosen by the user, that same interval is sufficient to complete the user request. Moreover, this paradigm is appropriate when the number of requests during $Max\_Display$ is less than or equal to R and the DAG is complex.

3.1 Object Placement For Demand Driven

With this paradigm, similar to the discussion of Section 2, **the placement of each object of the DAG depends on the placement of its predecessor(s)**. To illustrate, assume that a traversal from three alternative objects u, v and w leads to object x (i.e., u, v and w are the three predecessors of x). The last subobject of u, v and w reside on three different disk channels: $\mathcal{C}(u_k) = 1$, $\mathcal{C}(v_l) = 5$, and $\mathcal{C}(w_m) = 7$ respectively (see Figure 4.a and Table 3.1). The objective is to place x intelligently in order to ensure regular scheduling for all three paths (ux, vx, and wx).

A complete DAG is a combination of objects. Each object might be: 1) replicated, 2) have a portion in flash, 3) be entirely flash resident, or 4) be replicated and have a portion in flash. For object x of the DAG in Figure 4.a consider each strategy in turn (Section 3.2 provides a complete algorithm to traverse a DAG object by object and place each object by choosing one of the following strategies based on a price analysis):

[6]Note that if some number of users request an identical path, they can be grouped together. In this study, however, we assume no prior knowledge about the user choice.

[7]Note that in the worst case the active request might be referencing the longest path.

Figure 4: Alternative object placement

3.1.1 Replication

One alternative is to replicate x three times (see Figure 4.b and Table 3.1.1). In this case, the number of copies required for an object x is not $fan\_in(x)$ (number of incoming edges to x), instead it is $path\_in(x)$ (number of paths from *source* of the DAG to x). This explains why both objects x and y should be copied 3 ($= path\_in(y)$) times instead of just once ($= fan\_in(y)$). Note that, $path\_in(x)$ defines the **maximum** required number of copies. The reason is that the last subobject of both u and v might reside on the same channel (i.e., $\mathcal{C}(u_k) = \mathcal{C}(v_l)$). Moreover, if x has R copies, each should start from a different channel, then independent of $path\_in(x)$, all paths to x can be scheduled regularly. Hence, the maximum number of replica of x is bounded by $Min(R, path\_in(x))$.

3.1.2 Read-ahead Buffering

We start by providing a general idea of this technique. Subsequently, we formalize its details. The general concept is as follows. Suppose x is placed after u (see Table 3.1.2) and $x_1..x_4$ are permanently staged in flash memory. Now consider path vx once the display of v is finished and the time interval corresponding to C_6 becomes available. The system uses the free time interval to read-ahead x_5 into DRAM and at the same time display x_1 from flash memory (see Figure 5). In the next interval, the system reads-ahead x_6

C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
u_k				v_l		w_m	
	x_1	x_2	x_3	x_4	x_5	x_6	x_7
x_8	x_9						
					x_1	x_2	x_3
x_4	x_5	x_6	x_7	x_8	x_9		
							x_1
x_2	x_3	x_4	x_5	x_6	x_7	x_8	x_9

Table 4: Replication: x is replicated three times.

Figure 5: Read-ahead Buffering schedule

into DRAM and displays x_2 from flash. This process is repeated until the time interval corresponding to C_2 becomes available (time 5 in Figure 5). At this point, the system displays x_5 from DRAM and reads-ahead x_9 into DRAM. After that, nothing is retrieved from the disk channels for four time intervals (they are either idle or employed by a non-real-time request). The display of the next object for this request employs C_7 due to the regular schedule requiring the system to shift one channel to the right during each time interval (regardless of whether they are employed or not). One might be tempted to start the placement of the next object (say y) from C_3, and instead of rendering channels idle during the shrinking state utilize them to retrieve y. However, this is not possible because we assumed no advance knowledge about the user decision. Therefore, the identity of the next object is unknown (it might be either y or some other object). Besides, placing y on C_3 might be beneficial for path vxy, but not for wxy which is also a valid path.

During a display, the read-ahead buffering impacts the contents of DRAM (the contents of flash is predetermined during the system placement of the object and remains unchanged). The DRAM requirements of a display has three states: a growing state, a steady state and a shrinking state (see Figure 5). Let $\text{HEAD}_i(v, x)$ denote the number of subobjects of x required to be flash resident for path vx when x_1 resides on C_i (in this example $\text{HEAD}_2(v, x) = 4$). $\text{HEAD}_i(v, x)$ defines both the maximum number of subobjects required to be DRAM resident (DRAM is stable at $\text{HEAD}_i(v, x)$ subobjects) and the number of time intervals during which the channel is not employed (see Figure 5). The number of time intervals DRAM is in steady state (stable intervals) can be computed as $\text{SIZE}(x) - (2 \times \text{HEAD}_2(v, x)) + 1$.

The same idea can be repeated for object w. However, since x_7 resides on C_8 (i.e., $\mathcal{C}(w_m)+1 = \mathcal{C}(x_7) = 8$), $\text{HEAD}_2(w, x)$ will be 6. Hence, the number of subobjects of x that should be flash resident is: $Max(\text{HEAD}_2(v, x), \text{HEAD}_2(w, x)) = H_2(x) = 6$. Note that once we have $H_2(x)$ subobjects of x flash resident, it is no longer necessary to maintain them on disk. Hence, the number of subobjects of x that become disk resident is reduced to:

$$disk\_space = \text{SIZE}(x) - H_2(x) \tag{2}$$

Similar to the discussion of Section 3.1.1, $H_i(x)$ is bounded by $R - 1$ ($H_i(x) \leq R - 1$). That is, once $R - 1$ subobjects of x become flash resident, independent of the channel containing the last subobject of its predecessors, the scheduling of all the incoming paths to x can be done regularly.

With demand driven, in the worst case there are R simultaneous requests for an object x, where the memory might be in steady state for all of them. To illustrate, assume $R = 3$ and $\text{HEAD}(t, z) = 5$ and $\text{SIZE}(z) = 20$. Hence, for path tz memory is stable at 5 subobjects for 11 ($= 20 - 2 * 5 + 1$) time intervals. Suppose 3 different requests to z: the first one requires z_6 to z_{10} be DRAM resident, the second requires z_{12} to z_{16} be DRAM resident, and finally the last requires z_{17} to z_{20} be DRAM resident. Thus, totally $R \times \text{HEAD}(t, z) = 15$ memory pages (the size of each page is equivalent to the size of a subobject) of DRAM is required (this is the worst case scenario that results in maximum DRAM requirement). However, if $\text{SIZE}(z) - H_i(z) < R \times H_i(z)$ then in the worst case z will become memory resident in its entirety ($H_i(z)$ in flash and the rest in DRAM). Hence, the maximum amount of required DRAM is:

$$max\_dram = Min(R \times H_i(z), \text{SIZE}(z) - H_i(z)) \tag{3}$$

In practice, instead of keeping read-ahead subobjects in DRAM during the growing and steady state, a dynamic scheduler can employ idle intervals to flush them onto the disk. In this case, they can be retrieved later by employing other idle intervals. For example, in Figure 5, consider time 2 when the system is retrieving x_6 from C_7 (see Table 3.1.2). Suppose the time interval corresponding to C_3 is free, x_6 can be flushed onto C_3 to free up DRAM. Later, at time 6 (during the shrinking state), instead of rendering a time interval idle to display x_6 from DRAM, the time interval can be utilized to retrieve x_6 from C_3 (C_3 was selected to respect the regularity of the display schedule). In general, the duration of time from when x_l becomes DRAM resident until it is displayed is defined as $H_i(x)$ time intervals. In the best case, if x_l is read-ahead from C_k at time t it can be flushed immediately to $C_{(k+H_i(x)) \bmod R)+1}$ in order to be retrieved and displayed later at time $t + H_i(x)$. Observe that retrieval does not always occur during the shrinking state. Hence, the existence of an idle interval to be utilized for retrieval is not always guaranteed. Therefore, to achieve the

Figure 6: Computation of optimal head size

i	$\text{HEAD}_i(u,x)$	$\text{HEAD}_i(v,x)$	$\text{HEAD}_i(w,x)$	$H_i(x)$
1	1	5	7	7
2	0	4	6	6
3	7	3	5	7
4	6	2	4	6
5	5	1	3	5
6	4	0	2	4
7	3	7	1	7
8	2	6	0	6

Table 6: A table to compute the optimal head size of x ($H(x) = 4$ in this example).

best case scenario, the scheduling should be done dynamically and intelligently. To illustrate, assume at time 1, x_5 is retrieved from C_6 and flushed onto C_2 ($= C_{(6+4) \bmod 8}$). However, at time 5 ($= 1 + H_i(x)$) the idle interval corresponding to C_2 is employed to retrieve x_9 and not x_5. At this point the system should retrieve x_5 and postpone the retrieval of x_9. Furthermore, it should guarantee that the time interval corresponding to C_2 will become available no later than time 9 to retrieve x_9. This technique is called *Dynamic Subobject Shuffling* and it is useful to reduce the DRAM requirement when the system is under-utilized (the number of active requests is smaller than R). Note that since this process is dynamic, to configure the system, the worst case DRAM requirement (i.e., $Min(R \times H_i(x), \text{SIZE}(x) - H_i(x))$) should still be considered.

Minimum Head

$H_i(x)$ should be minimized for each object x in the DAG in order to reduce the amount of required flash and DRAM. The problem is how to place object x in a R channel system such that it results in minimum head for x. More specifically which channel should contain the first subobject of x. Consider edges (u,x), (v,x), ... (z,x) in a DAG. If x is placed starting with channel one, then it will have the following head sizes for alternative paths: $\text{HEAD}_1(u,x)$, $\text{HEAD}_1(v,x)$, ..., $\text{HEAD}_1(z,x)$ (see Figure 6). For that placement, $H_1(x)$ will be the maximum of the above sizes. We can repeat this procedure for each channel i, where $1 \leq i \leq R$, to determine the channel j that minimizes $H_i(x)$ (denoted as the *optimal head size*, $H(x)$). The complexity of the above optimal algorithm is $O(path\_in(x) \times R)$ for each object of the DAG. Table 3.1.2 illustrates the algorithm to find $H(x)$ for the example in Figure 4.a.

Appendix A demonstrates how the complexity of the algorithm can be significantly reduced to $O(\log(fan\_in(x)) \times fan\_in(x))$.

C_1	C_2	C_3	C_4	C_5	C_6	C_7	C_8
u_k				v_l		w_m	
	x_1	x_2	x_3	x_4	x_5	x_6	x_7
x_8	x_9						
					x_1	x_2	x_3
x_4	x_5	x_6	x_7	x_8	x_9		

Table 7: Read-ahead Buffering and Replicate: x is replicated twice, path wx requires to read-ahead x.

3.1.3 Read-ahead Buffering and Replication

The idea is to both replicate an object and keep its head flash resident. This reduces the size of the head of x even further in order to minimize the required amount of both flash and DRAM (see Figure 4.c and Table 3.1.3). For example, by eliminating the first column of Table 3.1.2, the optimal head size of x ($H(x)$) is now reduced to 2 subobjects. Note that, this reduction is also advantageous for $H(y)$ (see Equation 24 in Appendix A); however, it increases $fan\_in(y)$ (see Figure 4.c).

3.1.4 Flash Resident

An alternative approach to all the above strategies, is to maintain the entire object flash resident. This placement needs neither DRAM nor disk storage. Intuitively for those objects that their optimal head size are approximately equal to their entire size, this placement will be superior.

3.2 A Price Analysis to Choose the Cheapest Placement Strategy

To decide which of the above placement options to choose from, we provide a comparison strategy. There are many factors that influence a final choice: the size of the object, its fan_in, its path_in and its optimal head size as well as architectural factors such as number of channels (R) and the available DRAM and flash memory.

Assume $\$disk$, $\$ram$, and $\$flash$ are the price per megabyte of disk, flash and DRAM storage, respectively. Moreover, assume n is the number of copies of an object x on disk, where $0 \leq n \leq Min(path\_in(x), R)$. Hence:

$$\begin{array}{ll} n = 0 & \equiv \text{flash resident} \\ n = 1 & \equiv \text{Read-ahead Buffering} \\ n = Min(path\_in(x), R) & \equiv \text{Replication} \\ n = otherwise & \equiv \text{Read-ahead Buffering and Replication} \end{array} \quad (4)$$

The objective is to find the best value of n in order to minimize the price of storage. Note that $H(x)$ is changing for different values of n, hence $Hx(n)$ (optimal head size of x when x has n copies on disk) is used instead, where

$$\begin{array}{l} Hx(Min(path\_in(x), R)) = 0 \\ Hx(0) = \text{SIZE}(x) \end{array} \quad (5)$$

The following defines the price for each resource as a function of n. The disk price is computed using Equation 2 as

$$disk\_price(n) = n \times (\text{SIZE}(x) - Hx(n)) \times \$disk \quad (6)$$

Since the required amount of flash is identical to the optimal head size,

$$flash\_price(n) = Hx(n) \times \$flash \tag{7}$$

The required amount of dram from Equation 3 is

$$dram(n) = Min(R \times Hx(n), \text{SIZE}(x) - Hx(n)) \tag{8}$$

However, assuming Mem is the amount of memory required by the objects assigned thus far, the amount of extra DRAM required by the assignment of a new object x is:

$$Extra\_Mem(n) = \begin{cases} 0 & \text{If } dram(n) \leq Mem \\ dram(n) - Mem & \text{otherwise} \end{cases} \tag{9}$$

Hence,

$$ram\_price(n) = Extra\_Mem(n) \times \$ram \tag{10}$$

The amount of required Mem is now updated to be:

$$Mem \leftarrow Mem + Extra\_Mem(n) \tag{11}$$

The value of n for each object is determined in order to minimize $storage\_price(n)$, where:

$$storage\_price(n) = disk\_price(n) + flash\_price(n) + ram\_price(n) \tag{12}$$

Figure 7 is a greedy algorithm[8] for object placement in the Demand Driven paradigm. The reason that the algorithm is greedy is because it starts from *source* and place the objects independent of their successors as it traverses the DAG. Figure 8 defines the *Place* subroutine used in Figure 7. The input to the algorithm is a DAG, $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, and R, where $\mathcal{V}$ and $\mathcal{E}$ are the sets of the DAG's vertices and edges, respectively.

$Mem \leftarrow 0$
$x \leftarrow source$
$\mathcal{PLACED} \leftarrow \emptyset$
while $(\mathcal{V} - \mathcal{PLACED} \neq \emptyset)$
    $Place(x)$
    $\mathcal{PLACED} \leftarrow \mathcal{PLACED} \cup \{x\}$
    Choose an object x from $\mathcal{V} - \mathcal{PLACED}$, such that all the predecessors
        of x are in $\mathcal{PLACED}$
END

Figure 7: The main placement algorithm

Algorithm in Figure 8 might execute lines 15 and 17 for the first series of the objects, because of the initial value of Mem that might result in a high value of ram_price. However, later in the DAG other objects might increase Mem. It would be advantageous for the starting objects to take advantage of this price_payed memory; this can be achieved by re-running the algorithm with the value of Mem produced from the first

[8]We believe this is also an optimal algorithm. In Appendix A we informally proved the optimality of the greedy algorithm (using *fan_in* instead of *path_in*) for read-ahead buffering. We expect this to be true when the combination of strategies are employed. However, we do not as yet have a proof.

```
Place(x)
{
    Compute Hx(1) (using the algorithm described in Section 3.1.2)
    Compute storage_price(1) (using Equation 12)
    n ← 0
    /* Consider replication in combination with read-ahead buffering */
    repeat
        n ← n + 1
        Find that predecessor of x which results in the Hx(n) head size (say y),
            using the method described in Section 3.1.3
        Compute Hx(n + 1) and subsequently storage_price(n + 1) for the case
            that x is replicated after y
    until (storage_price(n + 1) ≥ storage_price(n))
    if storage_price(0) is the minimum then
        place x in the flash and return
    if storage_price(Min(path_in(x), R)) is the minimum then
        replicate x, Min(path_in(x), R) times on disk channels and return
    Place the first Hx(n) subobjects of x in flash
    Place n copies of the last SIZE(x) − Hx(n) subobjects of x on disk channels
    Mem ← Mem + Extra_Mem(n)
    return
}
```

Figure 8: *Place* subroutine

run.

4 Data Driven Paradigm

A limitation of the demand driven is that once the number of users exceeds the number of available resources ($= R$), the latency time increases as a function of number of the pending requests. To avoid this for DAGs with a few paths, we investigate the *Data Driven paradigm*. With this paradigm, each time interval is assigned to each possible path. This is identical to the *pay per view cable service* analogy, where each cable channel was assigned to a single path. Hence, for a DAG with p possible paths, at least p time intervals are required. Recall from Section 2, that $R = p$ channels are required to provide the system with p time intervals. Furthermore, to reduce the latency time (again similar to pay per view analogy) the number of intervals must increase as a multiple of p. Therefore, k groups of p time intervals are dedicated to display all possible paths of a DAG periodically. In this case, (in theory) an infinite number of users can be grouped together and employ one of the k groups of displays. If a request arrives once the display of a group has been initiated, it must wait until the next period when the system starts to display the next group. Therefore, the larger the value of k, the lower the latency time. However, the lower bound for the worst latency time is p time intervals. This is because a maximum of p time intervals are required until the time interval corresponding to the channel containing the first subobject of the source of the DAG becomes available. Moreover, the longer $Max\_Display$, the higher the value of k should be to reach the lower bound of p time intervals latency time. Since $R = k \times p$, the more complex the path (large p and $Max\_Display$), the more channels (R) are required to reduce the latency time. While the maximum latency time cannot become less than p time intervals. This paradigm is appropriate when the number of requests in every p time intervals exceeds p and the application can tolerate p time intervals latency time.

Figure 10: Alternative time interval assignments

The **maximum** latency observed by a user before the display of the DAG starts is ℓ intervals:

$$\ell = Max(p, Max\_Display - R + p) \tag{13}$$

where the unit of $Max\_Display$ is in number of time intervals rather than seconds. In [KRT94], it is suggested to reduce the latency time by using more memory. The problem is that due to high bandwidth of multimedia objects, to reduce the latency a little, a large amount of memory is required. However, we use Equation 13, to reduce the latency by adding more disk channels (i.e. increasing R). Note that from Equation 13, the upper bound of R is $Max\_Display$ (i.e., adding more channels will not reduce the latency time any further).

4.1 Object Placement For Data Driven

The object placement for this paradigm is different from that of demand driven paradigm. However, the alternative placement strategies (using replication, read-ahead buffering, a combination of replication and read-ahead buffering, and entirely flash resident) are still applicable.

To illustrate how this paradigm might be implemented, consider the DAG in Figure 9 where $p = 6$. Recall that with this paradigm, one time interval is required per possible path. Hence, 6 time intervals ($i1$, $i2$, ..., $i6$) are required to display all possible paths of the DAG in Figure 9. Assuming a six channel system that can support these intervals, the assignment of the paths to the intervals is important because it determines: 1) the placement of the data, and hence 2) the amount of resources required. Figure 10 demonstrates two possible time interval assignments for the DAG in Figure 9. Table 4.1 demonstrates the desired display schedule based on time interval assignment in Figure 10.a.

In Table 4.1, x is displayed by employing one of the six available intervals (i.e., $i1$) and its display is shared by all the users. This is achieved by multiplexing one stream from a channel among the users. One might be tempted to display x six times once using each time interval. This wastes resources in two ways. First, it wastes five intervals and the bandwidth of five channels: these intervals could be employed by a non-real time application executing simultaneously. Second, it increases the amount of storage required by replicating x five additional times. To observe, recall that each time interval corresponds to a different channel. Hence, to be able to display x_1 six times (by employing six intervals), it must be replicated on 6 different channels. This wastes storage. The minimum number of distinct intervals employed to display an object o is equal to $path\_in(o)$ ($path\_in(source) = 1$). This is because, x should be displayed at least $path\_in(o)$ times in order to ensure a continuous display of all the paths. For example, the display of y employs one interval ($i1$) (see Table 4.1) while display of w employs $path\_in(w) = 3$ intervals (i.e., $i3$, $i5$, and $i6$).

Figure 11: Impact of time interval assignment and round-robin allocation of intervals on object placement

Once the display of x completes, the display of y, z, and w should start simultaneously. This is because at this point the initial group of users can potentially break into three different groups, each pursuing a different path (say, first group continues with display of y, second group with z and the last group with w). Now the problem is how to place y, z, and w to ensure their simultaneous display after x.

With the data driven paradigm, the placement of each object depends on: 1) the placement of its predecessors (similar to demand driven), and 2) the *distance* between the time interval employed to display the object and the one employed to display its predecessor. Figure 11 illustrates how both placement of x and the assigned time intervals ($i1$, $i2$, and $i3$ in Figure 10.a) impact the placement of y_1, z_1, and w_1. Since y employs the same interval employed by x (see the first row of Table 4.1), it can start immediately after x. To present this by an equation, define $\mathcal{C}(x_j, i)$ as the channel containing x_j, when x is displayed by employing time interval i (where $1 \leq i \leq p$). Hence, $\mathcal{C}(y_1, 1) = \mathcal{C}(x_6, 1) + 1$, which means y_1 should be placed immediately after x_6 when the same interval ($i1$) is assigned to display both. However, to display z_1, the second time interval ($i2$) is employed. Since the *distance* between the time intervals assigned to x ($i1$) and z ($i2$) is 1, z_1 should be placed on $\mathcal{C}(z_1, 2) = \mathcal{C}(x_6, 1) + 1 - distance = \mathcal{C}(x_6, 1)$. In general,

$$\mathcal{C}(z_1, j) = \mathcal{C}(x_m, q) + 1 - (j - q) \tag{14}$$

where x_m is the last subobject of x, j and q are the identity of the time intervals employed to display z and x, respectively. Using Equation 14, the placement of w_1 can be computed as follows. From Table 4.1, $i3$ is assigned to display w (i.e., $j = 3$ in Eq. 14) immediately after the display of x completes from $i1$ (i.e., $q = 1$ in Eq. 14). Hence,

$$\begin{aligned} \mathcal{C}(w_1, 3) &= \mathcal{C}(x_6, 1) + 1 - (3 - 1) \\ &= \mathcal{C}(x_6, 1) - 1 \end{aligned} \tag{15}$$

Note that $\mathcal{C}(x_6, 1)$ can be computed directly from $\mathcal{C}(x_1, 1)$. Generally,

$$\mathcal{C}(x_k, j) = ((\mathcal{C}(x_1, j) + k - 2) \bmod R) + 1 \tag{16}$$

At this point the difference between $\mathcal{C}(y_5)$ and $\mathcal{C}(y_5, 1)$ might not be obvious. The reason is that $path\_in(y) = 1$ and hence it is displayed by employing only **one** time interval ($i1$). However, from Table 4.1, it should be

	Reserved Time Interval/path					
	$i=1/xyzu$	$i=2/xzu$	$i=3/xwu$	$i=4/xyu$	$i=5/xzwu$	$i=6/xyzwu$
$\mathcal{C}(x_1,i)$	-	-	-	-	-	-
$\mathcal{C}(y_1,i)$	$\mathcal{C}(x_6,1)+1$	-	-	-	-	-
$\mathcal{C}(z_1,i)$	$\mathcal{C}(y_5,1)+1$	$\mathcal{C}(x_6,1)$	-	-	-	-
$\mathcal{C}(w_1,i)$	-	-	$\mathcal{C}(x_6,1)-1$	-	$\mathcal{C}(z_6,2)-2$	$\mathcal{C}(z_6,1)-4$
$\mathcal{C}(u_1,i)$	$\mathcal{C}(z_6,1)+1$	$\mathcal{C}(z_6,2)+1$	$\mathcal{C}(w_4,3)+1$	$\mathcal{C}(y_5,1)-2$	$\mathcal{C}(w_4,5)+1$	$\mathcal{C}(w_4,6)+1$

Table 9: Placement constraint table for the first time interval assignments

obvious that for example $\mathcal{C}(u_1,1) \neq \mathcal{C}(u_1,4)$. This is because $i1$ is employed to display u_1 after z while $i4$ is employed to display u_1 after y.

By applying Equation 14, $\mathcal{C}(o_1,i)$ can be determined for each object $o \in DAG$ where $1 \leq i \leq p$. The result can be summarized in *placement constraint tables*. *Placement constraint tables* specify the candidate channel(s) from which the placement of each object should start. For example, Table 4.1 demonstrates a *placement constraint table* based on the display schedule in Table 4.1. As an example, lets construct Table 4.1 row-by-row. The first row ($\mathcal{C}(x_1,i)$) is empty. This is always true for the source of a DAG because the *source* has no predecessor. Hence, there is no constraint on the placement of the source (*source* can be placed starting from an arbitrary channel). To construct the second row ($\mathcal{C}(y_1,i)$), observe that since $path\_in(y)=1$, it is displayed only once by employing $i1$ (see the first row of Table 4.1). Therefore, $\mathcal{C}(y_1)$ for all other intervals is empty. For $i1$, however, y_1 should be placed immediately after x_6. This is because the same interval which was employed to display x, is now employed to display y. In other words, from Equation 14,

$$\mathcal{C}(y_1,1) = \mathcal{C}(x_6,1) + 1 - (1-1) = \mathcal{C}(x_6,1) + 1 \tag{17}$$

By an identical discussion

$$\mathcal{C}(z_1,1) = \mathcal{C}(y_5,1) + 1 \tag{18}$$

To compute $\mathcal{C}(z_1,2)$, observe that although z_1 should be displayed immediately after x_6, its assigned time interval ($i2$) is different from that of x ($i1$). Hence,

$$\begin{aligned} \mathcal{C}(z_1,2) &= \mathcal{C}(x_6,1) + 1 - (2-1) \\ &= \mathcal{C}(x_6,1) \end{aligned} \tag{19}$$

Similarly other elements of Table 4.1 can be computed. As the final example lets compute $\mathcal{C}(u_1,4)$. Since u_1 should be displayed from $i4$ immediately after the display of y_5 from $i1$ completes, then

$$\begin{aligned} \mathcal{C}(u_1,4) &= \mathcal{C}(y_5,1) + 1 - (4-1) \\ &= \mathcal{C}(y_5,1) - 2 \end{aligned} \tag{20}$$

Now that the constraint table is constructed, placing objects to satisfy a data driven display schedule (similar to Table 4.1) is trivial. For example, the third row of Table 4.1 denotes that the display of z should be scheduled from both $\mathcal{C}(y_5,1)+1$ and $\mathcal{C}(x_6,1)$. This can be achieved by (say) replicating z two times. An alternative is to use read-ahead buffering. To compute the optimal head size of z, assume $m_1 = \mathcal{C}(y_5,1) < m_2 = \mathcal{C}(x_6,1) - 1$. Now it is sufficient to compare $m_2 - m_1$ with $R - (m_2 - m_1)$ (applying Equations 22 and 23 from Appendix A) and start the placement of z from either $C_{m_1+1} = C_{\mathcal{C}(y_5,1)+1}$ or $C_{m_2+1} = C_{\mathcal{C}(x_6,1)}$. Generally, for each object similar to Section 3.1, the system can choose to do read-ahead buffering and/or replication, as well as maintaining the entire object flash resident.

Observe that if assignment of time intervals is modified (see Figure 10.b, and Table 4.1), then the placement constraint table would change as well (see Table 4.1). Different placement constraint tables result in different storage requirements. One can generate all the possible constraint tables and choose the one that

Reserved Interval/ For path:	⟶ Time ⟶																				
$i1/xzwu$	x_1	x_2	x_3	x_4	x_5	x_6	z_1	z_2	z_3	z_4	z_5	z_6	w_1	w_2	w_3	w_4	u_1	u_2	u_3	□	
$i2/xwu$		*	*	*	*	*	w_1	w_2	w_3	w_4	u_1	u_2	u_3	□							
$i3/xzu$			*	*	*	*	*	*	*	*	*	*	u_1	u_2	u_3	□					
$i4/xyzwu$				*	*	*	y_1	y_2	y_3	y_4	y_5	z_1	z_2	z_3	z_4	z_5	z_6	w_1	w_2	w_3	...
$i5/xyu$					*	*	*	*	*	*	*	u_1	u_2	u_3	□						
$i6/xyzu$						*	*	*	*	*	*	*	*	*	*	*	*	u_1	u_2	u_3	□
	□ = End of path display * = the time interval is reserved,but is idle Framed subobjects (e.g. u_1, w_1) in a column are displayed in parallel.																				

Table 10: The display of 6 possible paths corresponds to the second time interval assignment

	Reserved Time Interval/path					
	$xzwu$	xwu	xzu	$xyzwu$	xyu	$xyzu$
$\mathcal{C}(x_1,i)$	-	-	-	-	-	-
$\mathcal{C}(y_1,i)$	-	-	-	$\mathcal{C}(x_6,1)-2$	-	-
$\mathcal{C}(z_1,i)$	$\mathcal{C}(x_6,1)+1$	-	-	$\mathcal{C}(y_5,4)+1$	-	-
$\mathcal{C}(w_1,i)$	$\mathcal{C}(z_6,1)+1$	$\mathcal{C}(x_6,1)$	-	$\mathcal{C}(z_6,4)+1$	-	-
$\mathcal{C}(u_1,i)$	$\mathcal{C}(w_4,1)+1$	$\mathcal{C}(w_4,2)+1$	$\mathcal{C}(z_6,1)-1$	$\mathcal{C}(w_4,4)+1$	$\mathcal{C}(y_5,4)$	$\mathcal{C}(z_6,4)-1$

Table 11: Placement constraint table for the second time interval assignment

results in minimum storage requirement (lower cost). Since there are $p!$ possible time interval assignments, the complexity of this algorithm is $O(|DAG| \times p! \times p)$, where $|DAG|$ is the number of objects in the DAG and p is for the maximum number of constraints per object[9]. Note that this algorithm is applied only once to place the objects. The following paragraphs show that considering read-ahead buffering is sufficient for constraint tables.

For data driven paradigm, the exact amount of DRAM requirement can be computed in advance. The reason is that all the paths are displayed periodically. Hence, it is not required to consider the worst case scenarios. Thus, the price analysis can be done accurately. One heuristic is to first choose the placement constraint table which results in total minimum DRAM requirement (*optimal constraint table*) considering only read-ahead buffering. Next, draw a curve for DRAM requirement for that constraint table, and then eliminate curve peaks by replication and/or storing larger fractions of an object x in flash (i.e., $flash(x) > H(x)$) to reduce the total storage cost. This is true because, if $flash(x) = H(x) + d$, then the system can postpone read-ahead buffering d time intervals and reduce the number of stable intervals. This might also eliminate the DRAM curve peaks.

Since the *optimal constraint table* is determined based on read-ahead buffering, the complexity of the algorithm ($O(|DAG| \times p! \times p)$) can be reduced further. For example, in the last row of Table 4.1, there are three $\mathcal{C}(w_4)$ (i.e., $\mathcal{C}(w_4,3), \mathcal{C}(w_4,5), \mathcal{C}(w_4,6)$). However, since read-ahead buffering is employed, there will be only one replica of w on disks. Subsequently, there will be only one physical channel containing w_4 (i.e., $\mathcal{C}(w_4)$). This is independent of $path\_in(u)$ which includes three different paths from w. Hence, from Section 3.1.2 it is sufficient to only consider $\mathcal{C}(w_4)$. In other words, it is sufficient to consider $fan\_in(u)$ which includes only one edge from w.

There are some predetermined idle intervals in Tables 4.1 and 4.1 (denoted as $*$). In addition, the exact location of those intervals during which channel is not employed (discussed in Section 3.1.2, Figure 5) are now predictable beforehand. Therefore, a scheduler can assign these idle intervals for another application running on the same system.

[9]To be specific, it is $O(path\_in(o))$ for each object o of the DAG per time interval assignment.

5 Analysis and Discussion

In this section, we outline the tradeoffs with the alternative paradigms and how to choose one based on an application's characteristics. Next, a combination of both paradigms is introduced for applications that have a variable system load. Finally, we investigate situations where it is more appropriate to make the DAG entirely flash resident.

5.1 Data Driven versus Demand Driven

To decide which paradigm to employ, assume $\aleph_{request}$ is the number of requests referencing the DAG every $Max\_Display$ intervals. If the system employs the demand driven paradigm, with $R = \aleph_{request}$ channels, a new request can find an empty time interval most of the time. Thus, in the worst case, it incurs a latency time of $R-1$ time interval, and at best no latency, until the empty time interval reaches the desired channel. Hence, on the average, the latency time observed is $\frac{\aleph_{request}-1}{2}$. On the other hand, if $R = \aleph_{request}$ channels are used with the data driven paradigm, (from Equation 13) the average latency time would be: $\frac{Max(p, Max\_Display+p-\aleph_{request})}{2}$. One can compare the above average latency times and decide which paradigm to choose.

An alternative is to restrict R and observe how many requests can be supported by each paradigm with an identical average latency time. The paradigm with the highest number of requests can thus be chosen.

5.2 Data Driven combined with Demand Driven

If the number of requests varies through the life of an application, a combination of both paradigms can be employed. In this case the system should be configured for the data driven paradigm. Subsequently, the system can accumulate some statistics about the number of users referencing the DAG per ℓ intervals ($\aleph_{user}$). If $\aleph_{user} > p$, then it executes the data driven paradigm; otherwise, it employs the demand driven paradigm. This is because with data driven, p time intervals are reserved every ℓ intervals. Therefore, if there are fewer than p users every ℓ intervals, it is more efficient to reserve time intervals based on the exact number of users (demand driven).

Once the system decides to switch paradigms, it should abandon the current paradigm promptly. To observe, assume the system is currently using the data driven paradigm. Moreover, assume that the statistics suggest that the demand driven paradigm should be employed ($\aleph_{user} < p$). Once the display of one of the groups of the possible paths completes, the p available time intervals are employed to service $\aleph_{user}$ requests that are waiting for this group (one time interval per request). Obviously, except the first user, the rest will observe a higher latency time as expected (their display would have been started, if the system had continued to use the data driven paradigm). Similarly the display of other groups are terminated.

During the demand driven paradigm, the system continues accumulating statistics until $\aleph_{user} > p$. At this point, once an active request is terminated, its corresponding time interval is reserved. Note that servicing of new requests are stopped, resulting in higher latency. As soon as p adjacent time intervals are reserved, the system starts one group of displays of all possible paths. In a similar manner, the display of other groups is started incrementally as p time intervals become available. Paradigm switching results in higher latency time observed by the requests arriving during the switching period, in favor of reducing the latency time observed by the future requests. Therefore, it might not be wise to switch paradigm in short intervals, even if the number of users varies radically.

To be able to combine the two paradigms, the DAG should be placed with respect to both paradigms. A simple solution is to have two different copies of the DAG. An alternative is to place the DAG with respect to both paradigms. To achieve this, it is sufficient to add more columns to the *placement constraint tables* (described in Section 4) because of the demand driven paradigm. These columns add more constraints (each

object should start immediately after the last subobject of its predecessors). For example, in the second row of Table 4.1, y should also start after x_6. This can be confirmed by adding $\mathcal{C}(x_6) + 1$ to that row. Note that sometimes the new columns do not introduce additional constraints. For example, in the third row of Table 4.1, new columns are required to confirm that z should start after x_6 and y_5. However, these two constraints have already been introduced for the data driven paradigm.

5.3 Comparison with One Level Storage Structure

If there are potentially large number of users referencing a DAG, the question is, should the DAG become entirely flash resident. Since a large number of users is assumed, demand driven is not appropriate. As compared to the data driven paradigm, if the application cannot tolerate $p \times Time\_Interval$ (the lower bound on the maximum latency time (see Equation 13)), then the DAG should become flash resident. In the rest of this section, we assume the maximum latency tolerated by an application is higher than $p \times Time\_Interval$.

One can limit the latency time by substituting ℓ with (say) $Max\_Latency$ in Equation 13, and compute R. Subsequently, for this value of R, we can apply the algorithm in Section 4.1 and compute the total $storage\_price$. This price can be compared with the price of required flash to contain the entire DAG. Finally, the cheaper one is chosen (of course the expected number of simultaneous references to the DAG should also be considered).

6 Conclusion and Future Research Directions

In this study, presentations sharing clips are represented as a DAG. The DAG consists of objects that should be retrieved at a pre-specified bandwidth in order to support their continuous display. Alternative users might display different paths of a DAG. A path is one possible presentation of information selected by the user. Two alternative paradigms, demand driven and data driven, were introduced. The former is appropriate when: 1) the number of resources is greater than or equal to the number of users, and 2) the number of possible paths is significantly higher than the amount of available resources. The latter, is appropriate when the demand for data (number of users) is significantly higher than the number of resources. Assuming the simple striping placement strategy, we demonstrate how each paradigm can be implemented. Each paradigm introduces different placement constraint for the objects of a DAG. The best placement in a hierarchy of storage structure (consisting of DRAM, flash, and disk drive) was investigated based on the price analysis for each paradigm.

There are two straightforward extensions to this study. First, by choosing a DAG to represent a database, we restricted ourselves to *meet* temporal relationship as defined in [All83]. This is illustrated in Figure 12.a. Now assume each link bears information specifying other types of relationships defined in [All83]. For example, in Figure 12.b, display of object X overlaps with the display of Y for two subobjects (or $2 \times \frac{\text{SIZE}(subobject)}{B_{Display}}$ seconds). Hence, Y should be placed such that its display can start immediately after X_4. It is trivial that the techniques described in this paper can support these types of constraints with minor modifications. This is because the problem is the same as before, except that previously Y should have been placed such that its display can start immediately after X_6. Second, in this study we assumed that the user choose his/her desired path prior to its display. However, a DAG is placed with no prior knowledge about the probability that a path might be chosen. That is, the objects of the DAG are placed in order to provide a hiccup-free display for all plausible paths. Once this is done, the system can support an interactive application as well. That is, a user is not required to choose the complete path beforehand. To illustrate, consider Figure 12.a. The user can be allowed to choose to display Y or Z no later than the end of X's display. This is because, Y and Z are already placed such that both can be displayed immediately after X.

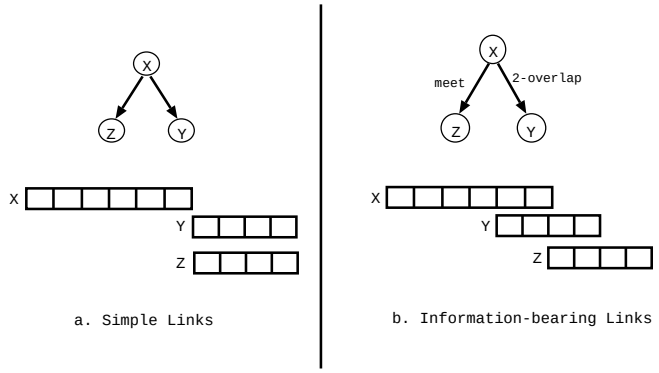


Figure 12: Straightforward Extensions

References

[AH91] D. Anderson and G. Homsy. A cotinuous media I/O server and its synchronization. *IEEE Computer*, October 1991.

[All83] James F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, 26(11):832–843, November 1983.

[BGMJ94] S. Berson, S. Ghandeharizadeh, R. Muntz, and X. Ju. Staggered Striping in Multimedia Information Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1994.

[CL93] H.J. Chen and T. Little. Physical Storage Organizations for Time-Dependent M ultimedia Data. In *Proceedings of the Foundations of Data Organization and Algorithms (FODO) Conference*, October 1993.

[DH93] Brian Dipert and Lou Hebert. Flash memory goes mainstream. *IEEE Spectrum*, 11(5):48–52, 1993.

[Fox91] E. A. Fox. Advances in Interactive Digital Multimedia Sytems. *IEEE Computer*, pages 9–21, October 1991.

[GKS94] S. Ghandeharizadeh, S. H. Kim, and C. Shahabi. Display of Continuous Media with Multi-Zone Magnetic Disks. Technical report, USC, 1994.

[GKS95] S. Ghandeharizadeh, S. H. Kim, and C. Shahabi. On Configuring a Single Disk Continuous Media Server. In *To appear in Proceedings of the ACM SIGMETRICS*, 1995.

[GRAQ91] S. Ghandeharizadeh, L. Ramos, Z. Asad, and W. Qureshi. Object Placement in Parallel Hypermedia Systems. In *Proceedings of the International Conference on Very Large Databases*, 1991.

[Has89] B. Haskell. International standards activities in image data compression. In *Proceedings of Scientific Data Compression Workshop*, pages 439–449, 1989. NASA conference Pub 3025, NASA Office of Management, Scientific and technical information division.

[KRT94] M. Kamath, K. Ramamritham, and D. Towsley. Buffer Management for Continuous Media Sharing in Multimedia Database Systems. In *Proceedings of the International Conference on Very Large Databases*, 1994.

[PNHV92] B. Prince, R. Norwood, J. Hartigan, and W. Vogley. Synchronous dynamic ram. *IEEE Spectrum*, 29(10):44–47, 1992.

[RV93] P. Rangan and H. Vin. Efficient Storage Techniques for Digital Continuous Media. *IEEE Transactions on Knowledge and Data Engineering*, 5(4), August 1993.

[RVR92] P. Rangan, H. Vin, and S. Ramanathan. Designing an On-Demand Multimeida Service. *IEEE Communications Magazine*, 30(7), July 1992.

[RW94] A. L. N. Reddy and J. C. Wyllie. I/O Issues in a Multimedia System. *IEEE Computer Magazine*, 27(3), March 1994.

[TPBG93] F.A. Tobagi, J. Pang, R. Baird, and M. Gang. Streaming RAID-A Disk Array Management System for Video Files. In *First ACM Conference on Multimedia*, August 1993.

[Wal94] S. Wallace. Managing mass storage. *Byte*, 19(3):78–88, 1994.

A On Demand Driven Complexity

As described in Section 3.1.2, the complexity of the algorithm to locate the best channel to start an object x in order to minimize its head size is $O(path\_in(x) \times R)$. This can be reduced to $O(\log(fan\_in(x)) \times fan\_in(x))$. In this section we describe this reduction in two steps. In the first step we reduce the complexity to $O(\log(path\_in(x)) \times path\_in(x))$. Subsequently, in the next step we reduce it further to $O(\log(fan\_in(x)) \times fan\_in(x))$.

Step 1:
Consider two important observations from Table 3.1.2:

- **Observation 1:** The first row of Table 3.1.2 is identical to the channel numbers containing the last subobject of u, v, and w (i.e., $\mathcal{C}(u_k)$, $\mathcal{C}(v_l)$, and $\mathcal{C}(w_m)$).
- **Observation 2:**
  $\text{HEAD}_i(vertex, x) = (\text{HEAD}_{i+1}(vertex, x) + 1) \bmod R$. This informally means that the values in each column is deterministic as a function of the rows: it is the value of the previous row minus one, except when the previous value is 0 at which point it becomes $R - 1$.

Using the above observations, the optimal placement of x and the minimum head size ($H(x)$), can be determined directly without constructing the entire table. The idea is illustrated in Figure 13, where m_1, m_2, m_3, and m_4 are the channels containing the last subobject of four different predecessors of an object x (e.g., if v is a predecessor of x and its last subobject (v_l) reside on C_3 then $m_1 = \mathcal{C}(v_l) = 3$). Moreover, assume they are sorted such that $m_1 < m_2 < m_3 < m_4$. Figure 13 is a generalized version of Table 3.1.2. The table can be divided into 5 ($= path\_in(x) + 1$) sections (see Figure 13). The first section includes the rows where $m1$ is reduced to 0. Since $m1$ is the smallest, it reduces to 0 sooner than m_2, m_3, and m_4. Subsequently, since m_4 is the largest and in each iteration all m's are reduced by 1, the maximum of each row always occur in m_4-*column* for the first section. Similarly in the second section $m2$ reduces to 0 sooner. The same is true for the third and forth section. Since we are interested in the maximum value in each row, it is sufficient to consider the maximum column in each section. However, we want to find the minimum of these maximums. The minimum of maximums in each section, obviously occurs in the last row of that section. For example, in the first section the maximum is always in m_4-*column* and it is decremented in each iteration. Hence, its minimum value occurs in the last iteration of that section. The value of this minimum of maximums for each five sections are $m_4 - m_1$, $R - (m_2 - m_1)$, $R - (m_3 - m_2)$, $R - (m_4 - m_3)$, and $m_4 + 1$, respectively. Moreover, since $m_4 - m_1 < m_4 + 1$:

$$\begin{aligned} H(x) = \text{Min}(&m_4 - m_1, \\ &R - (m_2 - m_1), R - (m_3 - m_2), R - (m_4 - m_3)) \end{aligned} \tag{21}$$

Figure 13: Faster method to compute minimum head size

or in general for n predecessors of x:

$$H(x) = \text{Min}(m_n - m_1, \\ R - (m_2 - m_1), ..., R - (m_n - m_{n-1})) \tag{22}$$

The channel number that x should start from is simply identical to the row number that the above minimum value appears. That is:

$$\begin{array}{lll} \text{if } H(x) = m_n - m_1 & \text{then} & \mathcal{C}(x_1) = m_1 + 1 \\ \text{if } H(x) = R - (m_2 - m_1) & " & \mathcal{C}(x_1) = m_2 + 1 \\ : & : & : \\ \text{if } H(x) = R - (m_n - m_{n-1}) & " & \mathcal{C}(x_1) = m_n + 1 \end{array} \tag{23}$$

Therefore, once the channels containing the last subobject of $path\_in(x)$ predecessors of x is known, by sorting them in $O(\log(path\_in(x)) \times path\_in(x))$ and applying Equations 22 and 23 in $O(path\_in(x))$ we can determine the optimal placement of x and $H(x)$. Hence, the total complexity of this algorithm is $O(\log(path\_in(x)) \times path\_in(x))$.

An important observation is that the minimum of maximums for each section occurs in a row that the value of one of its column is zero. Observe that if (say) $m_1 = \mathcal{C}(v_l)$, then its value reduces to 0 in row $\mathcal{C}(v_l) + 1$. This means that x is placed on $\mathcal{C}(v_l) + 1$, immediately after v, for that row. The above discussion proves that the placement of an object x should always start with the channel following the one that contains the last subobject of one of x's predecessors to ensure an optimal head size.

Step 2:
To place y after x (see Figure 14) there is no need to consider all three paths ux, vx, and wx. Using the last observation, it is sufficient to start the placement of y immediately after x independent of the incoming paths to x. Hence, the complexity of the above algorithm is reduced further to $O(\log(fan\_in(x)) \times fan\_in(x))$. However, note that $H(y) = H(x)$ and it is not computed from Equation 22. Generally, if the algorithm

Figure 14: Optimal head size, considering $fan\_in(y)$ instead of $path\_in(y)$

considers $fan\_in(y)$ instead of $path\_in(y)$, then Equation 22 will not compute the correct head size. To observe, assume z as another predecessor to y (see Figure 14), and let the last subobject of z (say z_k) resides on C_7. To place y, if the algorithm uses $m_1 = \mathcal{C}(x_9) = 6$ and $m_2 = \mathcal{C}(z_k) = 7$ in Equation 22, it computes $Fan\_in\_H(y) = 1$. This is not correct because $H(y)$ needs to be at least 4 due to paths uxy, vxy, and wxy. Hence,

$$H(y) = Max(H(x), H(z), Fan\_in\_H(y)) \tag{24}$$

However, the placement of y starts after either x or z (depends on whether $m_2 - m_1$ is smaller or $R - (m_2 - m_1)$ (from Equation 23)).