Avoiding Retrieval Contention for Composite Multimedia Objects

Surajit Chaudhuri
Hewlett-Packard Labs
Palo Alto, CA 94304

Shahram Ghandeharizadeh, Cyrus Shahabi*
University of Southern California
Los Angeles, California 90089

Abstract

An important requirement for multimedia presentations is the ability to compose new multimedia objects from the existing ones using temporal relationships. When compositions of continuous media objects are specified dynamically, the task of displaying these objects poses new challenges. These challenges are addressed in this paper. We show that in the case of a single composite object retrieval, a prefetching technique, *simple sliding*, provides an approach to reduce latency and buffering requirements. We extend this prefetching technique to the problem of retrieving multiple composite objects simultaneously. This new technique is termed *buffered sliding*. We consider several variants of the buffered sliding algorithm. A simulation-based study is used to compare their usage pattern of available memory and in determining their relative merits in reducing latency and increasing disk bandwidth utilization.

*Research supported in part by the National Science Foundation under grants IRI-9203389, IRI-9258362 (NYI award), and CDA-9216321, and a Hewlett-Packard unrestricted cash/equipment gift.

**Proceedings of the 21st VLDB Conference
Zurich, Swizerland, 1995**

1 Introduction

An important requirement for multimedia information systems is the ability to compose new multimedia objects from the existing multimedia objects [LG91]. Temporal primitives (e.g., before, after, overlaps [All83]) provide one of the most powerful and natural ways of authoring composition. Such composition is necessary in the domain of electronic publishing, computer music, news editing and many other applications.

In this paper, we investigate how a multimedia storage system can display a composite object. We focus on composite objects that are authored dynamically. To illustrate an example environment, consider a TV-news editor preparing to present new footage on unrest in Bosnia. He requires background material to provide the audience with a context. He considers playing a sequence of clips *one after another* from different footage taken at different times to author a thirty second presentation. He may decide to accompany a footage with appropriate music in parts (i.e., music overlaps video). He may conclude his presentation with split windows that concurrently display short clips that leave us with the images of diverse scenes in Bosnia. During editing of such a presentation, he would try several possible composition, possibly picking different sets of clips or music from the repository. Surely, the editor would like to display his composition during the authoring process to evaluate his choice. Thus, the process of editing a news story consisted of specifying composite objects using temporal relationships and then displaying those.

Note that displaying atomic objects of high-bandwidth continuous media objects, such as video (requiring no composition) is a challenging task in itself. Video clips require a continuous bandwidth for their display. For example, the bandwidth re-

quired by NTSC[1] for "network-quality" video is about 45 megabits per second (mbps) [Has89]. Even with a compression technique that provides a reasonable quality of presentation, the bandwidth requirement of the compressed object is typically quoted as 3 to 6 mbps [Nat95, Tri95]. Video objects are large in size and almost always resident on a secondary storage device. To support a continuous display, an object should be retrieved at its pre-specified rate. Otherwise, its display may suffer from frequent disruptions and delays, termed *hiccups*. Several studies have described techniques to support continuous display of a video clip [Pol91, SG95, TPBG93, CL93, BGMJ94, GKS95]. These studies control the layout of data on a secondary storage device to ensure the continuous display of an object.

When displaying a composite object, the system should ensure that the temporal constraints are satisfied. Since the composition is specified dynamically, no assumption can be made about the placement of the atomic objects. Furthermore, we would not be able to modify the placement of data each time a new presentation is authored because it may force the user to observe a long wait time before the display starts. In fact, retrieval of the atomic objects that constitute the composite object may conflict with one another (making significant demands on buffering), as will be indicated later. It is this unique aspect of retrieval contention that distinguishes this study from the previous work in both real-time scheduling and synchronization in distributed multimedia information systems (See [Buf94] for a survey). However, our results need to be used in conjunction with the past techniques for a complete solution to the problem of displaying composite objects.

The general problem of displaying composite objects is a complex one. In this paper, we focus on an important special case of this problem where the atomic objects that comprise the composite object have the same bandwidth requirements (e.g., overlapping video newsclips). However, we must still take into account that the possibility that the bandwidth requirements can be high. For the sake of ease of exposition, we will present the results in the context of binary composite objects, i.e., where the composite object consists of only two atomic objects. However, we have included sketches of generalizations necessary to retrieve arbitrary composite objects.

We begin with a discussion on the challenges of retrieval of composite objects (Section 2). We show how by using a technique, termed simple sliding, we need no more than a constant extra buffering, i.e., independent of the sizes of the individual atomic objects and the extent of overlap, to support display of a single composite object (Section 3). Our new technique uses the properties of layouts of atomic objects to compute a schedule of retrieval. We propose algorithms based on prefetching (using a generalization of simple sliding) that enable the system to display multiple composite objects (Section 4). The performance characteristics of these algorithms, using the metrics of usage patterns for available memory, bandwidth utilization and latency are studied in Section 5. The rest of this section provides a discussion of the framework for our proposed techniques.

1.1 Framework

As mentioned in the introduction, we assume that all objects belong to a single media type with identical bandwidth requirement, $B_{Display}$. We assume that the *simple striping* [SGM86, Pat93] technique is used to retrieve video (and audio) objects for continuous display. In this technique, we partition the aggregate bandwidth of the available disks into **channels** each with the bandwidth requirement of $B_{Display}$. Hence, denoting the *effective* disk bandwidth (considering the maximum seek and latency time, see [BGMJ94]) with B_{Disk} the number of channels (R) is computed as:

$$R = \lfloor \frac{D}{M} \rfloor$$

where D is the number of disk drives and $M = \frac{B_{Display}}{B_{Disk}}$. Note that R determines the number of simultaneous requests supported by the system. We will denote these R channels by C_1, C_2, ..., C_R. When $M < 1$, each channel provides a fraction of a disk bandwidth (see Figure 1.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 1.b).

In order to distribute the load imposed by a display of object X evenly across the channels, simple striping technique stripes X into 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. For example, in Figure 1.a, 9 subobjects of X (X_1, X_2, ..., X_9) are assigned to an 8 channel system, starting with C_1. Simple striping also minimizes the amount of memory required by pipelining the data from the disk to a display. When the system displays an object X, it employs a single channel during each time interval. Thus, to ensure a continuous display of an object X, the display of X_i is overlapped with the retrieval of X_{i+1} [BGMJ94]. The duration of the retrieval of a subobject is fixed for all subobjects and is termed a

[1] The US standard established by the National Television System Committee.

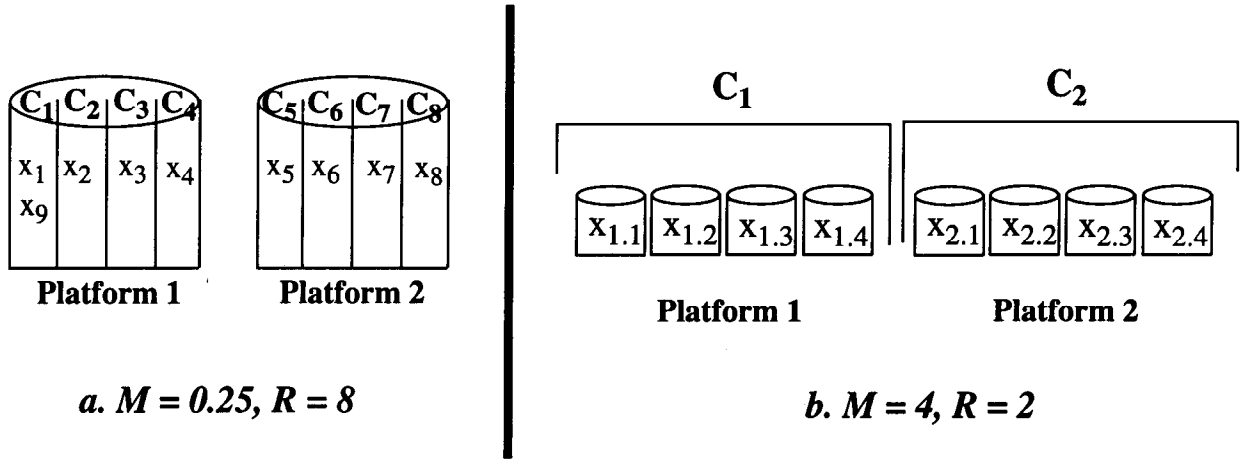


Figure 1: Logical channels in simple striping

time interval. Each display iterates over the channels, employing each in a round-robin manner. To illustrate, in Figure 1.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.

1.2 Atomic Objects

An atomic object X is a video clip that is retrieved in a sequential manner. Its subobjects are assigned to the channels in a round robin manner starting with an arbitrary channel. We use the function $channel(X_i)$ to determine the channel that contains subobject X_i. Let $channel(X_i)$ be c_i. In a system that consists of R channels, if X consists of k subobjects then we can also represent the layout of X as the list: $(c_1, ..c_k)$ where each $c_i \in \{1, .., R\}$. To illustrate, assume a system consisting of three channels ($R = 3$). The database consists of two objects: X and Y. Both objects are assigned to the channels in a round-robin manner starting with channel 1. Each of X and Y consist of five and eight subobjects respectively. In this case, X is represented as `(1 2 3 1 2)` while Y is represented as `(1 2 3 1 2 3 1 2)`.

1.3 Schedule

The *schedule* for the storage system reflects the state of the disks. It can be represented by specifying the set of busy channels for each time interval. We represent a schedule S as an array. The index of the array is time varying from "now" to $\mathcal{T}$, where $\mathcal{T}$ is time until which at least one of the channels is busy. Each element of the array is the set of busy channels at that time interval. For example, assume that $\mathcal{T} = 4$ and consider a schedule for the time period 1 through 4: S[1] = $\{1, 2\}$, S[2] = $\{2, 3\}$, S[3] = $\{3, 1\}$, S[4] = $\{2\}$. This states that at time 1, channels 1 and 2 were busy, at time 2, channels 2 and 3 were busy etc.

2 Problem Definition: Retrieval of Composite Objects

In this paper, we consider the class of composite multimedia objects that consist of two atomic objects. The temporal relationships between the display of two atomic objects may be specified in several ways. For simplicity, we assume the following model of specification:

A *composite object* is a triplet (X, Y, j) indicating that the composite object consists of atomic objects X and Y. The parameter j is the *lag parameter*: It indicates that the start time of object Y (i.e., display of Y_1) is synchronized with the display of subobject X_j. For example, to designate a complex object where the display of X and Y must start at the same time, we will use the notation $(X, Y, 1)$. Likewise, the composite object specification $(X, Y, 3)$ indicates that the display of Y is initiated with the display of the third subobject of X (see Figure 3a). This definition of a composite object supports the alternative temporal relationships described in [All83]. Table 2 lists these temporal relationships and their representation using our notation of a composite object. Our proposed techniques support all temporal constructs because they solve for: (1) arbitrary j values, (2) arbitrary sizes for both X and Y, and (3) placement of X and Y starting with an arbitrary channel.

Since two overlapping objects need to be displayed at the same time during retrieval of the overlapping part of a composite object, request for retrieval of such

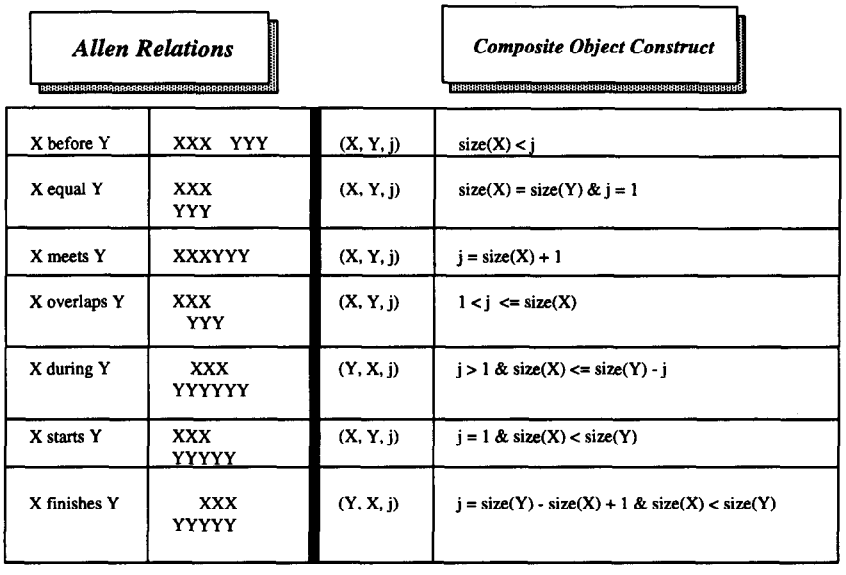


Figure 2: Allen temporal relationships and their representation using our notation of a composite object

an object must reserve the right number of channels at appropriate times in order to ensure a continuous display. Moreover, the identity of reserved channels is important due to placement of data. Without proper precautions, the system may fail to support continuous display of composite object if the placement of its participating atomic objects collide and reference subobjects that are stored on the same channel. To illustrate, assume a system consisting of 3 channels. Consider a single request displaying the composite object $(X, Y, 4)$ where X is (1 2 3 1 2 3 1 2) and Y is (1 2 3 1 2 3 1). Note that the retrieval of X_4 and Y_1 conflict, resulting in hiccups. This illustrates the problem of retrieval of composite objects. There could be *contentions within* a single composite object because each atomic object is laid out independently. Such contention arises even though there are no other active requests. This contention is formalized below.

Definition 2.1: A composite object (X, Y, j) has an *internal contention* if there is some $i \geq 1$ such that $channel(X_{i+j-1}) = channel(Y_i)$. ∎

Note that our notation extends naturally to specification of composite objects that contain more than one atomic objects. Thus, a composite object containing n atomic objects can be characterized by $(n-1)$ lag factor, e.g., $(X^1, ..X^n, j^2, ..j^n)$ where j^i denotes the lag factor of X^i with respect to the beginning of the display of X^1. Note that Definition 2.1 can be generalized in a straight-forward fashion for the case where the composite object contains multiple atomic objects.

One might attempt to address the internal contention problem similar to retrieval of atomic objects in a multi-user environment. This is *not* appropriate for the following reason. The display of each atomic object in a multi-user environment is *independent*. Different scheduling policies may result in increased latency, but it is always possible to obtain a schedule that requires *no* additional buffer. However, because of the synchronization constraint between the displays of overlapping objects, treating the objects in the composite object as independent objects may require significant *buffering*. Indeed, when there is contention within a composite object, buffering might be unavoidable. In our previous example where X and Y collide, while scheduling X and Y independently does not need any buffer (if latency is acceptable), retrieval of $(X, Y, 4)$ requires at least one buffer.

In the rest of the paper, we will address the question of retrieval of composite objects. We will study the solutions using three metrics: latency time, memory and disk utilization. Latency time is defined as the amount of time elapsed from the arrival time of a request to the onset of the display of its referenced composite object. Both memory and disk utilization are defined as the amount of time that the memory and disks are busy supporting displays as a function of total time, respectively.

In the following section, we study the problem of displaying a single composite object. Section 4 extends the study to environments that support simultaneous display of several composite objects.

3 Single Display Environment

In this section, we describe how to minimize both the buffer requirements and latency of a display referencing a single composite multimedia object.

Naive Prefetching Strategy: Let us assume that the specification of the composite object is (X, Y, j) where X and Y have $k1$ and $k2$ subobjects respectively, i.e., $X = (X_1, .., X_{k1})$ and $Y = (Y_1, .., Y_{k2})$. The naive solution will be to prefetch the entire overlapped part, i.e., subobjects $(Y_1..Y_u)$ where $u = min(k1 - j, k2)$. Having fetched the overlapping part, we can now begin fetching the sequence of subobjects for X and then conclude by fetching the rest of Y. Thus, the sequence of retrieval may be depicted as follows:

$$Y_1, .., Y_u, \quad X_1, ..., X_{k1} \quad Y_{u+1}, ..., Y_{k2}$$

However, note that this naive method incurs a latency of u time intervals and requires u buffers. Thus, in the example of Figure 3.a, we will incur an overhead of 3 buffers and a latency of 3 time intervals. Note that if a system is configured with a single channel ($R = 1$) then naive prefetching is the only alternative that is available. In such a case, naive prefetching reduces to retrieving the two objects one after the other. The **contention prefetching** strategy is the obvious improvement over the naive prefetching. Given a composite object (X, Y, j), it prefetches only those subobjects Y_i for which there is a retrieval contention with the corresponding subobjects of X, i.e., $channel(Y_i) = channel(X_{i+j-1})$. However, naive prefetching and contention prefetching behave in an identical manner because the subobjects of each atomic object are assigned to the channels in a round-robin manner. However, contention prefetching is a desirable method when the layout is not round-robin. Assume that we have to display the composite object $(X, Y, 4)$ where the layout of X and Y are (2 1 3 2 1 3 2 1 3) and (1 2 3 1 2 3 1) respectively. In that case, the naive prefetching would prefetch the first 6 subobjects of Y, which is the overlapping part of the object Y. In contrast, the contention prefetching strategy will prefetch only the third and the sixth subobjects of Y since these are only subobjects whose retrieval would conflict with the retrieval of the corresponding subobjects of X. Contention prefetching is appropriate when a presentation consists of several composite objects. The scenarios where contention prefetching is desirable are described in [CGS].

Simple Sliding Algorithm: The amount of prefetching in naive (and contention prefetching) technique grows as the overlap between the atomic objects increase. Moreover, neither of these techniques exploit the round-robin assignment of subobjects. In contrast, by exploiting the layout, simple sliding technique minimizes both latency and buffer requirement of a display *independent* of the size of overlap as long as $R > 1$.

To illustrate the idea of simple sliding consider Figure 3. Figure 3.a shows a composite object Z which comprises of two atomic objects X and Y where the lag parameter is 3. Assuming X_3 and Y_1 reside on two different channels, the retrieval and display of Z can be depicted as in Figure 3.b. Trivially, no extra buffer is required in this case. Now assume that $channel(Y_1) = channel(X_3)$. Exploiting the round-robin layout of the objects and assuming $R > 2$, we know that both $channel(Y_1) \neq channel(X_2)$ and $channel(Y_1) \neq channel(X_4)$. Thus, the system has two choices: either (1) slide the retrieval of Y **up** for one interval (upslide Y) and prefetch one subobject of Y, see Figure 3.c, or (2) slide the retrieval of Y **down** for one interval (downslide Y) and prefetch one subobject of X, see Figure 3.d. Note that downsliding Y introduces latency. The rest of this section provides a formal description of simple sliding.

We say that an object X has a *regular layout cycle* if the subobject X_{j+1} is placed at a channel "adjacent" to X_j:

$$channel(X_{j+1}) = (channel(X_1) + j) \ mod \ R + 1$$

From this definition it follows that regular layouts differ from one another only in the placement of their first subobjects. Observe that simple striping enforces precisely this property. For example, two regular layouts of X over five channels might be (1 2 3 4 5 1 2 3). An alternative layout for X might be (3 4 5 1 2 3 4 5). Thus, regular layout and simple striping are identical.

Lemma 3.2 : *Consider the retrieval of the composite object (X, Y, j), where both X and Y have a regular layout. There is a retrieval contention iff $channel(X_j) = channel(Y_1)$.*

As mentioned earlier, naive prefetching is the only option when there is single channel. However, the more interesting case is where $R > 1$. In such a case, the following identity holds ($j > 1$) since the subobject X_{j-1} must be on a different channel from the subobject Y_1:

$$(channel(X_j) = channel(Y_1)) \Rightarrow$$

$$(channel(X_{j-1}) \neq channel(Y_1))$$

In other words, when $R > 1$, then we know that $(X, Y, j - 1)$ has no contention if (X, Y, j) has contention. Therefore, in such a case, we can *prefetch* subobject Y_1 during retrieval of X_{j-1}. Such "upsliding" results in *zero latency*. However, it requires *one extra buffer* during the display of Y.

In case $j = 1$, then we use the observation (similar to above) that $(Y, X, 2)$ does not have any contention whenever $(X, Y, 1)$ does have contention. This

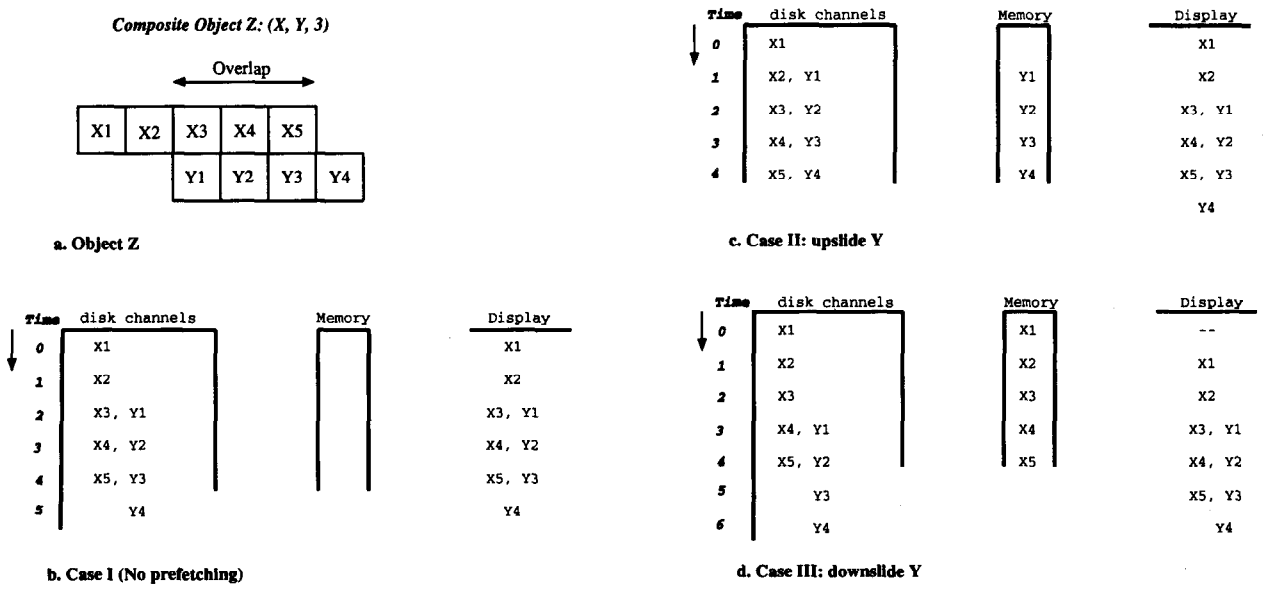


Figure 3: Simple Sliding

motivates a technique that prefetches Y_1 and then begin fetching subobjects of X concurrently with the retrieval of subobjects of Y. This "downsliding" results in *latency of one* time interval. In addition, it requires *one extra buffer* (during the display of X).

We refer to this strategy as *simple sliding*. For the class of binary composite objects of the form (X, Y, j), it is the *optimal* algorithm for retrieval when (1) only a single display request is active and (2) objects have regular layouts (as in simple striping). It is noteworthy that both the buffer requirement and latency of a display is *independent* of the size of the composite object. This algorithm is summarized below. We have omitted the details of buffering as they have already been explained.

Algorithm *Simple_Sliding*(X,Y,j)
if there is a single channel **then**
    do naive prefetching
else
    if $j = 1$ **then** fetch $(Y, X, 2)$
    else fetch $(X, Y, j-1)$

Assuming that the system must overlap the display of two atomic objects that constitute a composite object, this algorithm requires at most one extra buffer. Moreover, its incurred latency is bounded by one time interval. The technique of simple sliding is surprisingly robust even where the atomic objects do not satisfy the regular layout assumption. For example, the technique can be adapted with modifications for the case where new channels are added.

We now briefly sketch the generalization of *Simple_Sliding* when the composite object consists of more than two atomic objects, e.g., $(X^1, ..X^n, j^2, ..j^n)$. For simplicity, assume that all the objects X^i $(i = 1..n)$ are mutually overlapping. The generalization ensures that the composite object can be displayed with the number of extra buffers (as well as latency) independent of the *duration* of overlap among atomic objects. Intuitively, since the sliding technique must use buffers independent of the extent of overlap, it must assign a channel for display of each of the n objects. Therefore, $R \geq n$. For binary composite objects, the condition reduces to $R > 1$, which we have already identified as a necessary condition for use of simple sliding for binary composite objects. It can be shown that the buffer requirements are at most $n(n-1)/2$ and the latency is at most $n-1$. The details of the generalization of the algorithm will be reported in [CGS].

4 Multi Display Environment

The problem of retrieval contention becomes challenging in the presence of multiple users displaying objects simultaneously. This is due to competition for disk bandwidths by multiple independent requests arriving at random times. In particular, in the case of a single user environment, we established that using an extra buffer, it is possible to bound the latency for retrieval with simple sliding. Such is no longer the case for multi-user case. While a single extra buffer is still sufficient to remove internal contention, there may be significant latency due to: 1) demands on bandwidths by active users, and 2) active users colliding and referencing data items that reside on one channel. Increased availability of memory may be utilized to prefetch additional objects that lead to contention. Therefore, a key issue in multi-user environment will be to study the role of memory to reduce latency.

We assume that the scheduler in a multi-user system consists of two modules (1) A priority governing and (2) A task assignment module. The priority governing module is responsible for determining the priority of tasks among the queued jobs. Examples of heuristics that govern the priority governor may include First Come First Serve, Shortest Job First. Given a request to display a composite multimedia object, the task assignment module schedules it by taking into account the current status of the channels. The focus of our study is on the task assignment module. For simplicity, we will assume that scheduling is *non preemptive*, i.e., once a task is scheduled, it's schedule remains unchanged.

4.1 Buffered Sliding Algorithms

The class of algorithms that we present here try to minimize latency by using available buffers. Let us assume that due to contention, we can*not* schedule (X, Y, j) $(j > 1)$ beginning at time t but can schedule $(X, Y, j-1)$ beginning at time t. In that case, we can use an extra buffer to prefetch Y_1 during retrieval of X_{j-1}. By virtue of prefetching, we are then able to display (X, Y, j) beginning at time t. We can generalize the idea to the case where if the display of $(X, Y, j-b)$ is possible beginning at time t, then using b buffers, we can display (X, Y, j) beginning at time t (assuming $j > b$). We can think of such use of buffers as *sliding upwards*. Let us now assume that it is also possible to schedule $(X, Y, j+1)$ beginning at time t. In such a case, display of X will be ahead of display of Y by $(j+1)$ and by using an extra buffer to prefetch X, it is possible to display (X, Y, j) beginning at time $t+1$. We can refer to such a scheduling technique as *sliding downward*. In the general case, given b extra buffers, if display of $(X, Y, j+b)$ is possible beginning at time t, then display of (X, Y, j) can start beginning at $t+b$. The above technique of sliding upwards and downwards is a generalization of prefetching based on simple sliding algorithm that was presented in Section 3. However, we are now allowing use of multiple buffers for prefetching. Informally, we refer to these as *buffered sliding*. Observe that if for scheduling of a display, sliding upwards and downwards require the same number of buffers, then we must prefer sliding upwards because it does *not* adversely impact latency.

We will now describe the algorithms that use buffered sliding to avoid contention so that latency can be reduced and disk utilization can be increased. Given the current schedule S, these algorithms take as input a parameter B, which is the maximum number of extra buffers used for sliding, as well as the specification of the composite object (X, Y, j) to be scheduled. A key function invoked by buffered sliding algorithms is $atomic\_schedule\_after(X, S, u, S', u')$. It consumes as its parameters an atomic object X, the current schedule S, a time u. It outputs a time u' which is the earliest point in time when X can be displayed, i.e., u' is the minimum value greater than u, such that $channel(X_i)$ is free during interval $u'+i-1$ for each $i \in [1, k1]$. The other output parameter S' indicates the updated schedule. The procedure $atomic\_schedule\_before(X, S, u, S', u')$ is similar to $atomic\_schedule\_after$ except that it computes an u' that precedes u but is closest to it. Thus, $atomic\_schedule\_after$ and $atomic\_schedule\_before$ correspond to sliding up and down respectively using at most B extra buffers.

The key functions, both $atomic\_schedule\_after$ and $atomic\_schedule\_before$, should examine the availability of both channels and buffers per interval. Although the corresponding channels might be free, the system may not be able to schedule X because of the unavailability of buffers. This is because by sliding object Y upward (downward), the system is using prefetching. During prefetching, the buffer requirements of a display has three states: a growing state, a steady state and a shrinking state. Once the time intervals that the object's display and retrieval starts are fixed, the buffer requirement per interval can be computed accurately. The functions $atomic\_schedule\_after$ and $atomic\_schedule\_before$ employ the models introduced in [SG95] to compute the buffer requirement for every interval. Subsequently, if the buffer requirement for an interval exceeds the maximum available memory they continue searching for another candidate interval to initiate the display of a candidate object.

4.2 First_Match and Exhaust_B

Given B extra buffers, there could be a family of algorithms based on alternative approaches that traverse the search space to incorporate the schedule of a display with the current system schedule (corresponding to active displays). In our work, we have investigated two algorithms. The algorithm *First_Match* is *conservative* in its approach of using extra buffers. On the other hand, the algorithm *Exhaust_B* tries to *greedily* use all B extra buffers for sliding. Thus, the latter tries to minimize latency at the cost of increased use of buffers whereas the former tries to minimize the use of extra buffers for the current display. Given a workload, it is interesting to see which of the two approaches lead to lower overall latency for a given amount of available buffers. In the performance section, we will address these questions.

The algorithm *First_Match* (see Figure 4.a) assigns object X to the earliest location where it can be sched-

Algorithm First_Match
$u_0 = 0$;
repeat
    $atomic\_schedule\_after(X, S, u_0, S_1, u_1)$;
    $atomic\_schedule\_before(Y, S_1, u_1 + j, S_2, u_2)$;
    if $u_2 \geq u_1 + j - B$ then **return**(S_2);
    $atomic\_schedule\_after(X, S_1, u_1 + j, S_2, u_2)$;
    if $u_2 \leq u_1 + j + B$ then **return**(S_2);
    $u_0 = u_1$;
forever

a. First_Match Algorithm

Algorithm Exhaust_B
$u_0 = 0$;
repeat
    $atomic\_schedule\_after(X, S, u_0, S_1, u_1)$;
    $atomic\_schedule\_after(Y, S_1, u_1 + j - B, S_2, u_2)$;
    if $u_2 \leq u_1 + j + B$ then **return**(S_2);
    $u_0 = u_1$;
forever

b. Exhaust_B Algorithm

Figure 4: Algorithms for Multi-User Environment

uled as an atomic object (say u). It then tries to see whether Y can be scheduled at $u+j$. If so, then none of the extra B buffers will be needed. Otherwise, it looks for a match nearest to $u + j$, sliding upwards until it exhausts all B buffers. The reason for sliding upwards is to avoid increasing latency. If we fail to find a suitable match, then attempts are made to schedule Y by sliding downwards using at most B buffers. This increases latency. If this also fails, a new time interval is sought to display X and the above steps are repeated.

In contrast, the algorithm *Exhaust_B* (see Figure 4.b) uses the extra B buffers greedily. Using the buffers, it slides Y up as high as possible $(u + j - B)$ for scheduling. On failure, it slides down incrementally until $i + j + B$. This results in utilization of channels prior to the current interval in favor of freeing up future channels. For example, Figure 5 demonstrates a portion of the *schedule* array for an eight channel system. Composite object Z is defined as $(X, Y, 4)$ and the retrieval of X is scheduled to start during the 3rd interval. First_Match will schedule the retrieval of Y from the 6th interval utilizing no buffers for prefetching. This is done although $B > 3$. However, Exhaust-B slides Y upward for 2 intervals. It cannot slide it higher because the 6th channel is busy for higher intervals. Finally, observe that for a given B, in the worst case, the space of possibilities examined by these two algorithms is identical.

Efficient implementations of First_Match and Exhaust_B require careful indexing of information on availability of cahnnels so that repeated calls to *atomic_schedule_before* and *atomic_schedule_after* inexpensive. The implementations exploit the regular layout property of simple striping and for large B also employ approximating techniques to reduce the search space of schedules [CGS].

4.3 Estimating B

If the composite object has no internal contention, then the minimum *required* value of B is zero. In the worst case, such a value of B would result in starting

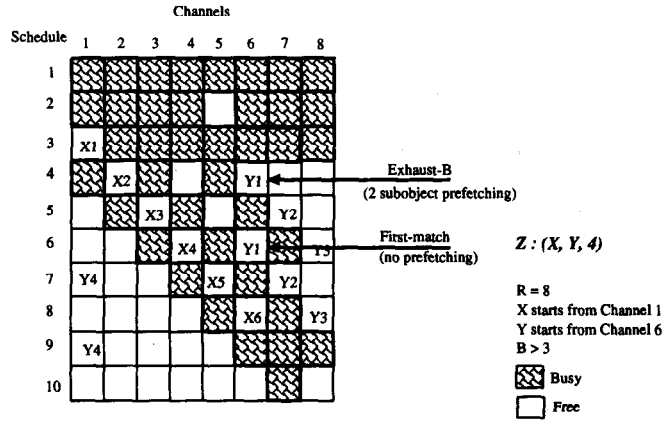


Figure 5: Exhaust-B vs. First_Match

the display after all the currently scheduled displays have terminated. However, as explained in the single user case, if the composite object has internal contention, then the minimum *required* value of B is one. Increasing B potentially reduces latency. By adjusting B, the algorithm can be *dynamically* adjusted for available buffers. We use the following three simple heuristics to compute the value of B. See Figure 6.

- **MinB:** In this case the value of B is fixed at 1 which is the minimum buffer requirement. This limits the extent of sliding permitted.

- **MaxB:** This is a greedy approach where the value of B is always equal to the entire system memory (all buffers). This results in the maximum flexibility for the buffered sliding algorithms. As shown in Section 5 this flexibility cannot be rendered effective if we use the First_Match algorithm.

- **FunB:** The value of B is a function of the number of availabel buffers and the maximum number of displays supported by a system:

$$B = Max(1, 2 \times \frac{MaxMem}{R}) \qquad (1)$$

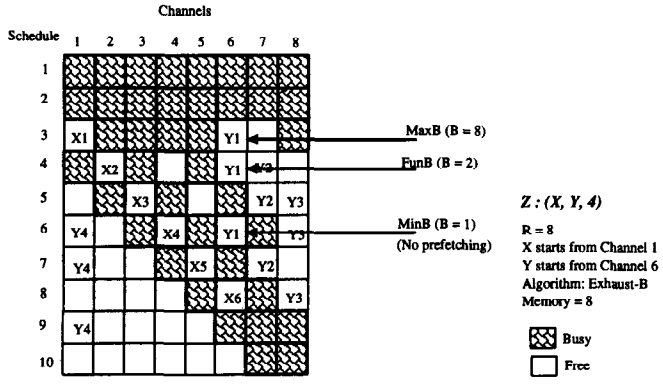


Figure 6: Impact of B

The above function is based on the following intuition. During a time interval, each object's share of buffers is $2 \times \frac{MaxMem}{R}$. This is because during each time period at most R atomic objects are active simultaneously. Moreover, each composite object consists of 2 atomic objects where only one of them is using buffers for prefetching[2].

4.4 Generalization

The algorithms for scheduling display of a composite object (X, Y, j) that we discussed have the following key steps. First, the atomic object X is scheduled. Next, the atomic object Y is scheduled while ensuring that the lag factor j is respected and the number of buffers utilized does not exceed B. Finally, if scheduling of Y fails to meet the above constraints, then X is rescheduled.

When we consider a composite object consisting of more than two atomic objects, then the above steps can be generalized. Thus, we can attempt scheduling atomic objects X^1, X^2,..,X^n in sequence while ensuring that the *cumulative* number of extra buffers assigned to respect lag factors does not exceed B. If a partial consistent schedule (say, obtained by scheduling $X^1..X^k$) cannot be extended, then X^k is rescheduled. However, such an approach may have significant overhead in determining a schedule. Therefore, we are investigating other alternatives as well. Regarding assignment of B to a display request, we can use the alternatives discussed in the previous section. Observe that **minB** is no longer 1, but may depend on n as well as on the lag factors.

[2] An alternative heuristic is to define B as a function of the composite object structure. Assuming a composite object (X,Y,j), let's define B as $Max(1, \frac{j}{2})$. We investigated this heuristic in our experiments and eliminated it from this discussion because it did not result in good performance.

5 Performance Evaluation

We implemented a simulation model to: 1) compare *First_Match* with *Exhaust-B*, and 2) investigate the impact of B and system memory on their performance. Our experiments focused on binary composite objects only. Therefore, it remains an open question whether our conclusions generalize to arbitrary composite objects. First, we describe the simulation model. Next, we report the results of our experiments.

5.1 Simulation Model

For the purposes of this evaluation, we assumed a platform of 20 channels (i.e., $R = 20$). The effective bandwidth of each channel is 4 mbps (supports a single display of MPEG-2 compressed clips). The system was configured with a 512 kilobyte block size. In other word, the size of each subobject is 512 Kbytes and the duration of a time interval is 1 seconds.

The database consists of 200 atomic objects, each with a 4 mbps bandwidth requirement. The size of the objects was fixed at 65 subobjects, except for the last experiment where the objects had different sizes. Consequently, the display time of each atomic object is 1.08 minutes. The rationale is that every 1.08 minutes the scene is changed (for example by using fade-in or fade-out requiring a small amount of overlap). The objects are assigned to the channels in a round-robin manner, starting with a random channel.

The total number of requests submitted to the system is 3000. Each request references a composite object consisting of two atomic objects. We inquired five types of overlaps for the two atomic objects within a composite object:

1. Small overlap: the two atomic objects overlap for 2 seconds (2 subobjects).

2. Moderate overlap: the two atomic objects overlap for 30 seconds (30 subobjects).

3. Complete overlap: the two atomic objects overlap for 1.05 minutes (63 subobjects).

4. Zero overlap: the two atomic objects do not overlap, but meet [All83].

5. Variable overlap: the amount of overlap between two atomic objects X and Y are chosen randomly between 1 and $Min(size(X), size(Y))$ subobjects.

We employed an open simulation model for our evaluation: requests arrive every *think* time intervals. We manipulated the parameter *load* to model two alternative loads on the system: Heavy and light system

load. The value of *think* is a function of R, *Obj_Size*, *Overlap* (the amount of overlap), and *load*:

$$think0 = \frac{(2 \times Obj\_Size) - Overlap}{\frac{R}{2}} + 2$$

$$think = \frac{think0}{load}$$

If $load = 1$, the requests arrive so far apart that at least one idle channel is available when a request arrives (lightly loaded system). For $load > 1$, the compartment of requests arrival is tighter (smaller think time), imposing a higher load on the system.

5.2 Experiment Results

For all the experiments we vary the amount of available buffers from one to 400 subobjects (the x-axis of all the graphs) and we measured the average latency time in number of intervals (the y-axis). Note that the disk utilization is inversely proportional to the average latency time. That is as the average latency time decreases, the total disk utilization increases. This is because, for lower average latency, the same number of requests were serviced in a relatively smaller number of intervals. Hence, the total disk utilization was higher.

Figure 7 demonstrates the results of the first set of experiments where the system was heavily loaded[3] ($load = 2$) and First_Match algorithm was employed. In Figure 7.a, all the composite objects consisting of atomic objects with two subobject overlap. When $B = 1$ (MinB) the graph levels off at $MaxMem = 5$ because the extra amount of buffer cannot be utilized. However, both MaxB and FunB continue reducing the latency time by utilizing the buffers. MaxB also levels off at $MaxMem = 100$ because the first match interval can be found with smaller value of B. Hence, the extra amount of B is not beneficial. However, since the value of B for FunB is a function of the available buffers, it will reach the level off point later than MaxB. Note that the effective search space of FunB is smaller than that of both MinB and MaxB. MinB suffers from repeated failures because B is too small to provide sufficient flexibility for prefetching. Therefore, many invocations of *atomic_schedule_before* and *atomic_schedule_before* need to be made. On the other hand, a large B value requires a significant number of possible sliding positions. FunB by having a flexible value of B always tries to provide each request with no more than its own share of buffers. Hence, it provides a smaller search space as compared to both MinB and MaxB. Figures 7.b and 7.c show the same observations when the amount of overlap is 30 and 63 subobjects, respectively.

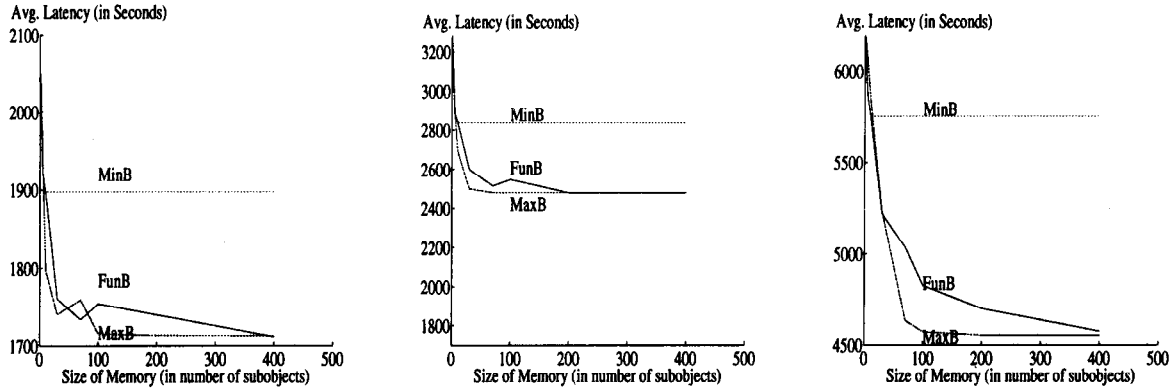
We repeated the above set of experiments for the Exhaust-B algorithm. The basic observations for MinB, MaxB, and FunB remained the same. Figure 9a compares First_Match with Exhaust-B for FunB and MaxB with 2 overlap objects. An interesting phenomenon is that with Exhaust-B neither MaxB nor FunB levels off. Instead, they continue reducing the latency time as the system memory grows. This is because Exhaust-B does not stop when it finds the first match and continues searching for an available time interval earlier than that determined by First_match. By doing so, it utilizes more of the available buffers. Note that this will result in a large search space for Exhaust-B. To reduce its search space, we used a heuristic. Instead of sliding the second object downward one interval at a time, the heuristic first detects the time interval that was responsible for the unavailability of buffers (failed-interval) and then slides the second object all the way to the failed-interval.

Exhaust-B in combination with MaxB is a greedy algorithm that tries to allocate all the available buffers to currently active requests. When memory is a scarce resource this might force other requests to starve since display of composite objects with internal contention requires at least one extra buffer. In other words, these requests require at least one buffer to be scheduled. Figure 8.a supports this fact and it shows that for small amount of system buffers (less than 100 subobjects) FunB outperformed MaxB. Even MinB outperformed MaxB when the amount of system buffer is less than 70 subobjects. An interesting observation is that the behavior of First_Match and Exhaust-B are identical for large overlaps (compare Figure 7.c with 8.b). This is because First_Match differs from Exhaust-B when the second object slides upward. For large overlaps the amount of upward sliding is very small (usually 2 intervals in our experiments). Hence, both algorithms end up finding the same interval for scheduling the second object.

In order to show that prefetching reduces the latency even when there is no overlap (simulating the *meet* temporal relationship proposed by [All83]) we performed an experiment when $Overlap = 0$ (see Figure 8.c). Since in this case there was no internal contention, we investigated $B = 0$ (no prefetching) as well. As shown in Figure 8.c, heuristics based on buffered sliding achieves a significantly lower average latency time as compared to no prefetching.

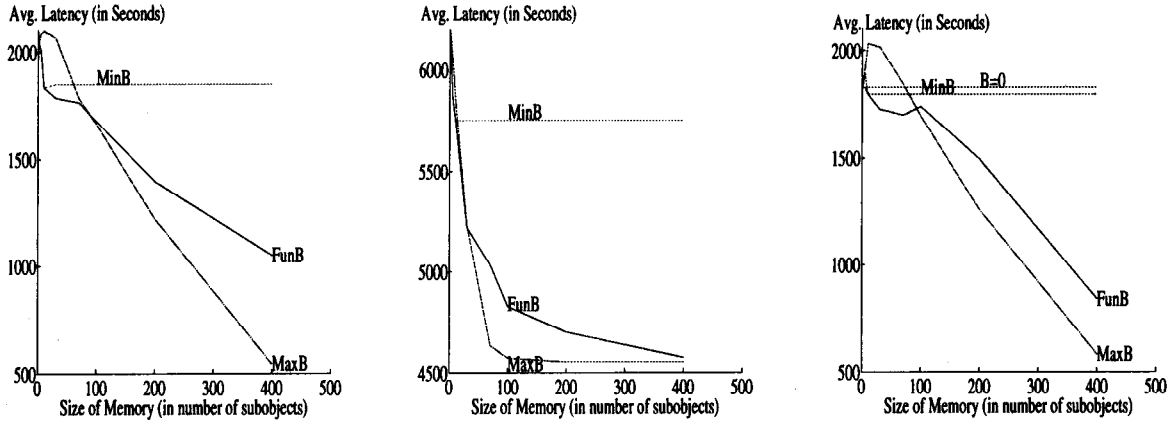
Finally, to ensure that our techniques are independent of the fixed structure of composite objects, we conducted another experiment. In this experiment, the atomic objects were no longer equi-sized. Instead,

[3] We performed the same set of experiments for a lightly loaded system and although the average latency time was reduced significantly, the basic observations were identical.



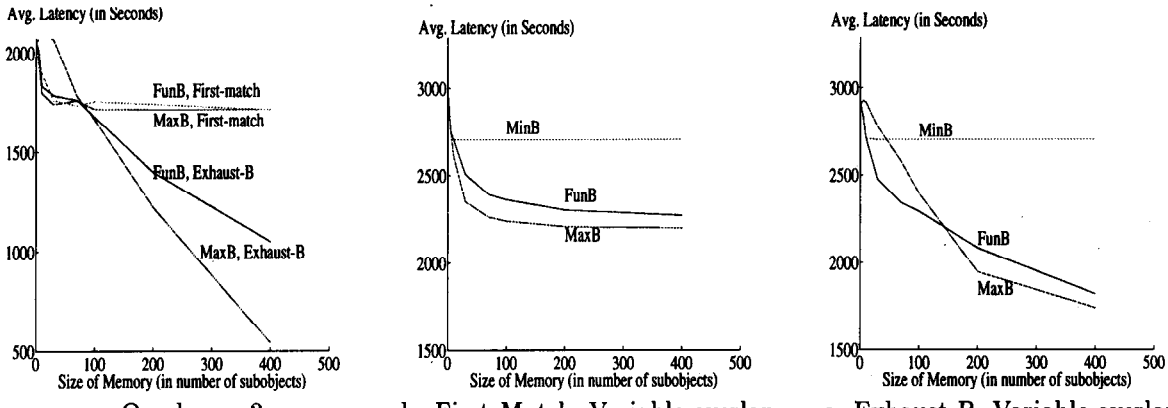
a. Overlap = 2 b. Overlap = 30 c. Overlap = 63

Figure 7: First_Match, Heavy loaded system



a. Overlap = 2 b. Overlap = 63 c. Overlap = 0

Figure 8: Exhaust-B, Heavy loaded system



a. Overlap = 2
Fixed Obj_Size

b. First_Match, Variable overlap
Variable Obj_Size

c. Exhaust-B, Variable overlap
Variable Obj_Size

Figure 9: Exhaust-B vs. First_Match, Heavy loaded system

we randomly chose the size of atomic objects to be from 30 to 65 subobjects. Similarly, the amount of overlap between two objects X and Y was a random number between 1 and $Min(size(X), size(Y))$ subobjects. The observations remain valid for both Exhaust-B and First_Match. The results are reported in Figures 9b and 9c.

6 Conclusion

We investigated the problem of continuously displaying composite objects that are dynamically specified. The ability to display such objects is likely to be important in many multimedia applications. The problem is challenging because a composite object consists of *overlapped* atomic objects. Therefore, in order to support continuous display of composite objects, we need to solve the problem arising due to contention during retrieval of overlapped objects.

In this paper, we proposed techniques based on simple sliding and buffered sliding that help support continuous display by *partial* prefetching of overlapping objects instead of the naive strategy of prefetching overlapped objects entirely. The key idea in both these methods is to exploit the striped layout of the multimedia objects. Our strategies apply for single display and multi display environments.

Our study can be extended in several ways. We mention a few of those here. First, our experimental study in the multi-display scenario has focused on composite objects consisting of only two atomic objects. Identifying new approaches to scheduling when the composite object has more complex structure and comparing them experimentally are important problems. Next, we need to extend our approach to the case where not all atomic objects in the composite object have the same bandwidth.

As a final remark, we should note that the problem of displaying a composite object is a complex problem since temporal synchronization is subject to several other system parameters that have not been considered in our retrieval model. Thus, our proposed technique should be viewed as a tool for retrieving composite objects and not as a complete solution in itself.

Acknowledgement: We thank Umesh Dayal for useful discussions and comments on the draft.

References

[All83] James F. Allen. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM*, pages 832–843, November 1983.

[BGMJ94] S. Berson, S. Ghandeharizadeh, R. Muntz, and X. Ju. Staggered Striping in Multimedia Information Systems. In *In Proc. of SIGMOD94*, 1994.

[Buf94] John Buford, editor. *Multimedia Systems*. Addison Wesley, 1994.

[CGS] S. Chaudhuri, S. Ghandeharizadeh, and C. Shahabi. Retrieval of Composite Multimedia Objects. Technical report, HP Lab. Technical Report, In preparation.

[CL93] H.J. Chen and T. Little. Physical Storage Organizations for Time-Dependent Multimedia Data. In *In Proc. of FODO93*, October 1993.

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

[GRAQ91] S. Ghandeharizadeh, L. Ramos, Z. Asad, and W. Qureshi. Object Placement in Parallel Hypermedia Systems. In *In Proc. of VLDB91*, 1991.

[Has89] B. Haskell. International standards activities in image data compression. In *Proceedings of Scientific Data Compression Workshop*, pages 439–449, 1989.

[LG91] Thomas D.C. Little and Arif Ghafoor. Spatio-temporal composition of distributed multimedia objects for value-added networks. *IEEE Computer*, pages 42–50, November 1991.

[Nat95] K. Natarajan. Video Servers Take Root. *IEEE Spectrum*, 32(4):66–69, April 1995.

[Pat93] D. Patterson. Massive Parallelism and Massive Storage: Trends and Predictions for 1995 to 2000 (Keynote Speaker). In *Second International Conference on PDIS*, January 1993.

[Pol91] V.G. Polimenis. The Design of a File System that Supports Multimedia. Technical Report TR-91-020, ICSI, 1991.

[SG95] C. Shahabi and S. Ghandeharizadeh. Continuous Display of Presentations Sharing Clips. *ACM Journal of Multimedia Systems*, 3(2):76–90, May 1995.

[SGM86] K. Salem and H. Garcia-Molina. Disk striping. In *Proceedings of International Conference on Database Engineering*, February 1986.

[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.

[Tri95] C. Tristram. Bottleneck Busters. *Newmedia*, pages 53–56, April 1995.