Multimedia Systems (1996) 4: 140–155

Multimedia Systems
© Springer-Verlag 1996

Adaptive piggybacking: a novel technique for data sharing in video-on-demand storage servers*

Leana Golubchik[1,**], **John C.S. Lui**[2,***], **and Richard R. Muntz**[3,†]

[1] Department of Computer Science, Columbia University, New York, NY, 10027, USA
[2] Department of Computer Science, The Chinese University of Hong Kong, Shatin, NT, Hong Kong
[3] Computer Science Department, University of California at Los Angeles, Los Angeles, CA 90095, USA

Abstract. Recent technology advances have made multimedia on-demand services, such as home entertainment and home-shopping, important to the consumer market. One of the most challenging aspects of this type of service is providing access either instantaneously or within a small and reasonable latency upon request. We consider improvements in the performance of multimedia storage servers through data sharing between requests for *popular* objects, assuming that the I/O bandwidth is the critical resource in the system. We discuss a novel approach to data sharing, termed adaptive piggybacking, which can be used to reduce the aggregate I/O demand on the multimedia storage server and thus reduce latency for servicing new requests.

Key words: Multimedia storage systems – Video-on-demand – Data sharing – Resource management – Merging policies

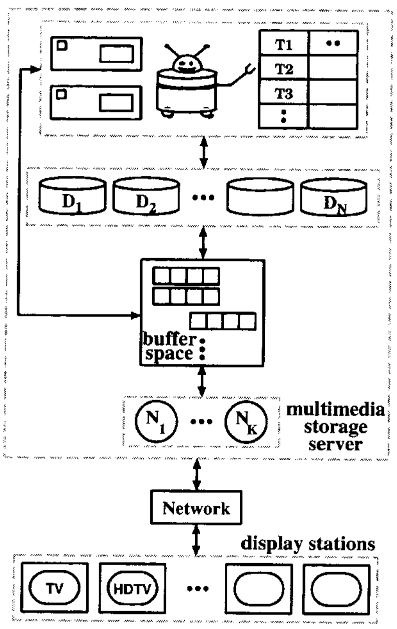


Fig. 1. Multimedia storage server architecture

1 Introduction

Recent technological advances in information and communication technologies have made multimedia on-demand services, such as movies-on-demand and home-shopping, feasible. Information systems today can not only store and retrieve large multimedia objects, they can also meet the stringent real-time requirements of continuously delivering an object at a specified bandwidth for the entire duration of its display. Multimedia systems already play a major role in educational applications, entertainment technology, and library information systems.

In this paper, we consider a video-on-demand storage server, such as the one depicted in Fig. 1, which archives many objects of long duration, such as movies, music videos, and educational training material. The storage server consists of a set of disks ($D_1 \ldots D_N$), a set of processors ($N_1 \ldots N_K$), buffer space, and a tertiary storage device. The entire database resides on tertiary storage, and the more frequently accessed objects are cached on disks. (We assume that the caching on disks is done on-demand, i.e., a non-disk-resident object is fetched from tertiary storage only when it is referenced. Some form of the Least Recently Used (LRU) policy can be used to purge objects from disks to create space for the newly retrieved object.) We also assume that a request for an object must be serviced from the disk subsystem. The size of the objects (on the order of 4.5 GB for a 100-min MPEG-2-encoded movie) precludes them from being stored in main memory. The long latency and high bandwidth cost of tertiary storage precludes objects from being transmitted directly from tertiary devices. (The seek latency for a 1.3 GB tape on a U.S. $1000-tape drive can be on the order of 20 s (Drapeau & Katz 1993), whereas a simi-

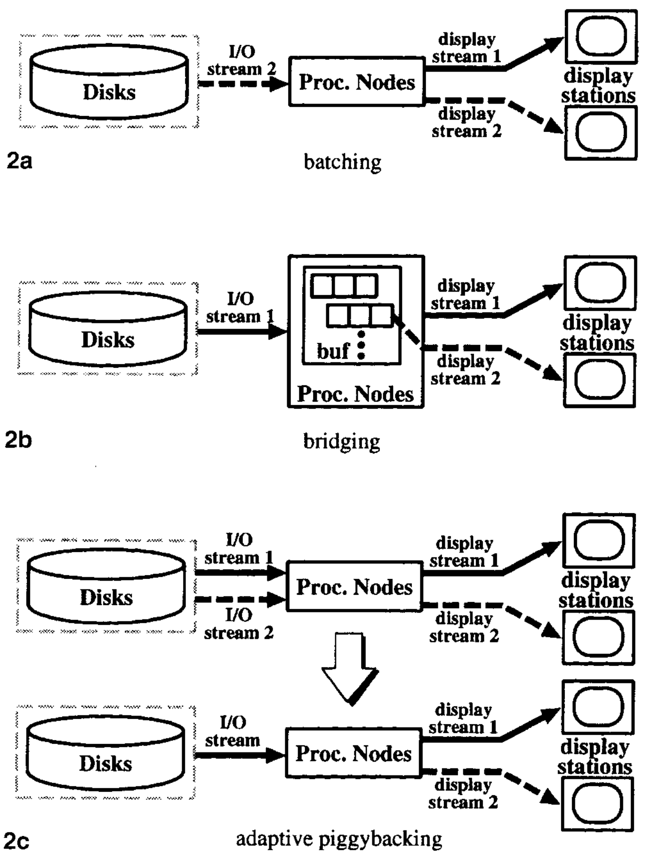
* A version of this paper appeared in SIGMETRICS/Performance '95, under the title "Reducing I/O Demand in Video-On-Demand Storage Servers"
** e-mail: leana@cs.columbia.edu
*** e-mail: cslui@cs.cuhk.hk
† e-mail: muntz@cs.ucla.edu
Correspondence to: L. Golubchik

larly priced disk, of a similar capacity, has a maximum seek time on the order of 35 ms and more than 16 times the transfer rate. Tape systems with significantly higher transfer rates and tape capacities, although not with a much lower seek latency, do exist, but at a cost of U.S.$40 000–$300 000.) If the requested object is not disk-resident, then it must be retrieved from the tertiary store and placed on disks before its display can be initiated. This can result in one or more objects being purged from disks due to lack of space. A disk-resident object is displayed by scheduling an I/O stream and reading the data from the appropriate disks.

One of the most challenging aspects of such systems is providing *on-demand* service to multiple clients simultaneously, thus realizing economies of scale. That is, users expect to access objects such as movies within a small and "reasonable" latency, upon request. We define the latency for servicing a request as the time from the request's arrival to the time the system initiates the reading of the object (from a disk). The additional delay until data are actually delivered to the display device is considered relatively negligible. Latency can be attributed to: (a) insufficient bandwidth for servicing the request, (b) insufficient buffer space for scheduling its reading from the disks, or (c) insufficient disk storage, that is the object in question may not be disk resident and hence may have to be retrieved from tertiary storage before it can be scheduled for display.

For ease of exposition, we can assume that the server, depicted in Fig. 1, can be described by the following three parameters: (1) total available I/O bandwidth, (2) total available disk-storage space, and (3) total available buffer space. (We will not consider the characteristics of the tertiary device in this paper.) These parameters, in conjunction with the data layout and scheduling schemes, determine the cost of the server, as well as the quality of service it can offer. Although quality of service is a somewhat vague term, the *latency* in servicing a video request is one useful measure. In general, the more video streams a system can support simultaneously, the lower the average latency is for starting the service of a new request (at least for the disk-resident objects).

There are several basic architectures that can be used for constructing a video-on-demand server (Berson et al. 1994; Ozden et al. 1994; Tobagi et al. 1993). The distinctions between these architectures can be largely attributed to the data layout and scheduling techniques used. Let us consider one such system, in which the workload can be described by $\lambda = (\lambda_1, \lambda_2, \ldots, \lambda_K)$, where λ_i is the arrival rate of requests for object i and K is the total number of objects available on the storage server (including the non-disk-resident objects). Informally, we expect a skewed distribution of access frequencies with a relatively small subset of objects accessed very frequently, and the rest of the objects exhibiting fairly small access rates. (For instance, a movie server would have such characteristics, where a small subset of popular movies (for that week, perhaps) is accessed simultaneously by relatively many users. Furthermore, we assume that the change in access frequency is relatively slow, e.g., the set of popular movies should not change more often than once per week.) In such a system, it is fair to assume that there is at least sufficient disk storage to hold the popular objects. Moreover, it is very likely that I/O bandwidth is the critical resource that contributes to increases in latency. One way to reduce the latency is simply to purchase more disks. A more interesting and more economical approach might be either to attempt to improve the data layout and scheduling techniques or to reduce the I/O demand of each request in service by *sharing* data among requests for the same object.



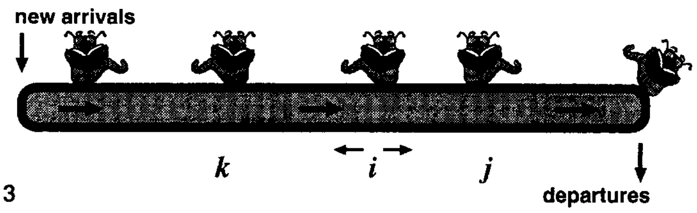


Fig. 2. Data sharing

Fig. 3. Conveyor belt analogy

There are several approaches to reducing the I/O demand on the storage server by sharing, or, in effect, increasing the number of user requests that can be served simultaneously. For example (Fig. 2):

1. *Batching*. Requests are delayed for up to T_i time units in hopes that more requests, for the same object i will arrive during the batching interval, and the entire group is serviced with a single I/O stream. In Fig. 2a, two displays are serviced by a single I/O stream that begins delivery after the arrival of the *second* request. Of course, a drawback of this approach is the additional latency created in the system.

2. *Bridging*. The temporal "gaps" between successive requests are closed by the use of buffer space, i.e., data read for a leading stream are held in buffers, and trailing requests are serviced from the buffer rather than by issuing another I/O stream. In Fig. 2b, the second display is serviced from the buffer space. A drawback of this approach is the need for additional buffer space.
3. *Adaptive piggybacking*. The display rates of requests *in progress*, for the same object, are adjusted until their corresponding I/O streams can be "merged" into one. In Fig. 2c, two displays are initially serviced by two individual I/O streams, which eventually "merge" into a single I/O stream.

In this paper, we concentrate on adaptive piggybacking. It is a more innovative approach and, to the best of our knowledge, has not been studied (or even proposed) before. Some work on batching (Dan et al. 1994a,b; Ozden et al. 1994; Yu et al. 1995) and bridging (Dan et al. 1995; Kamath et al. 1994, 1995) does exist.

An adaptive piggybacking procedure is defined to be a policy for altering display rates of requests *in progress* (for the same object), for the purpose of "merging" their respective I/O streams into a single stream that can serve the entire group of merged requests. The idea is similar to that of batching, with one notable exception. The grouping is done *dynamically* and while the displays are *in progress*, i.e., no latency is experienced by the user. Note that, the reduction in the I/O demand is not quite as large as in the case of batching, since some time must pass before the streams can merge. (The display adjustment must be gradual (or slow) enough to insure that it is not noticeable to the user. We assume that altering the quality of the display as perceived by an "average" user is not an acceptable solution.) Thus, the trade-off between these two techniques is between the latency for starting the service of a request and the amount of I/O bandwidth saved. These techniques are not mutually exclusive; in this paper, we present results of using adaptive piggybacking in conjunction with batching.

Consider an analogy of servicing video requests for a particular movie to a collection of bugs sitting on a moving conveyor belt (Fig. 3). The conveyor belt represents one particular movie. Its length corresponds to the duration of the movie display, and the rate at which the conveyor moves corresponds to the *normal* display rate of the movie (e.g., 30 frames/s for American television). Each bug represents a single I/O stream, servicing one or (as we shall see later) more display requests for that movie. The position of the bug on the conveyor belt represents the part of the movie being displayed by the corresponding I/O stream. If a bug chooses to remain still on the conveyor belt, then the corresponding stream displays the movie at the normal rate. If the bug chooses to crawl forward at some speed, then the corresponding movie is displayed at a slightly higher rate. Similarly, if the bug chooses to crawl backwards at some speed, then the corresponding movie is displayed at a slightly lower rate.

We elaborate on the technicalities involved in altering display rates in Sect. 2. For the remainder of this section we assume that display rates can be altered and concentrate on the possible benefits of this approach. These benefits are as follows. If two bugs, one crawling forward and one crawling backward, can "merge" at time t, before either one falls off the conveyor belt, then starting at time t the system can support both displays with only a single I/O stream. (Clearly, there is a problem of providing VCR functionality. Dan et at. (1994a) and Dey-Sircar (1994) solved a similar problem in the context of batching. Their solution of reserving channels for this purpose can be used here as well. Furthermore, adaptive piggybacking has one additional benefit. After obtaining a reserved channel and resuming display, we can again attempt merging. If we are successful, the reserved channel can be returned and reused by another stream.) Now, consider for the moment bug i in Fig. 3, which must make a decision, namely, whether to crawl forward toward bug j and piggyback on its I/O stream or to crawl backward toward bug k and, instead, piggyback on its stream. If i crawls forward, it takes less time to merge. However, after the merge, a smaller portion of the movie remains to be displayed, and hence the benefits of merging are not as great. In contrast, if i crawls backward toward k, it takes longer to merge. However, greater benefits might be reaped from this merger, if it can be achieved at an earlier portion of the conveyor belt.

In this paper, we consider several merging policies and evaluate them with respect to reduction in I/O bandwidth utilization. In general, the following parameters can be used to improve the number of simultaneous requests that a system can serve: (1) delay time for batching, (2) merging policy for adaptive piggybacking, (3) buffer allocation policy, and (4) display-rate altering techniques. Reduction in the I/O bandwidth consumed by the aggregate requests for a movie is considered to be the main goal of these policies. While other resources are affected, disk bandwidth is likely to be the most important and costly. This will remain so for the foreseeable future since disk capacity is increasing at a faster rate than disk bandwidth.

The remainder of the paper is organized as follows. In Sect. 2, we describe the feasibility of supporting multiple display rates. In Sect. 3, we briefly state the batching policy assumed in the remainder of this paper. In Sect. 4, we describe several adaptive piggybacking policies. Performance analysis of these policies can be found in Sect. 5, and the discussion of results can be found in Sect. 6. Our conclusions and directions for future work are given in Sect. 7.

2 Altering video display rates

Adaptive piggybacking is a viable technique for reducing I/O demand on a video storage server and consequently improving the response time of the system, if the storage server can dynamically alter the display rate of a request, or, rather, dynamically *time compress* or *time expand* some portion of an object's display. In this section we discuss how this can be done. Although not discussed here, the necessary time adjustments can be performed on the audio portion of an object as well, using techniques such as audio pitch correction. (We remind the reader that the word "video" (or "movie") in this context usually includes audio channels also.) For instance, when the movie *Amadeus* was transferred from film to television, it was "time compressed" by 3% (personal communication – R. Igo and B. Carpenter, Ampex). Clearly, the rate of this adjustment must be chosen accordingly to insure the necessary synchronization (Rubin 1991) with the video portion of the object.

We make the basic assumption that the display units being fed by the storage server conform to the National Television Standards Committee (NTSC) standard and display at a rate of 30 frames/s. Therefore, any time expansion or contraction must be done at the storage server. The effective display rate can be slowed down by adding additional frames to the video since the display occurs at a fixed rate. For example, if one additional frame is added for every ten of the original frames, the effective display rate (original frames/s) will be $30 \times \frac{10}{11}$. Similarly, by removing frames the effective display rate can be increased. There is ample evidence that effective display rates that are $\pm 5\%$ of the nominal rate can be achieved in such a way that it is not perceivable by the viewer. For example:

- A movie shot on film is transferred to video using a telecine machine that adapts to the 30 frames/s required for the video from the 24 frames/s that is standard for films. This is done with a 3–2 pulldown algorithm (Ohanian 1993; Rubin 1991), that for every four movie frames, creates five video frames. Two of the five frames produced are interpolations of a pair of the original frames. A similar type of interpolation could be used in our application.
- Ampex makes a product called Zeus(TM) (product description) that can be used to produce high quality video that has been time compressed or expanded by up to 8%. According to the product literature it can accomplish this without bounce or blur.
- Personal contacts within the the video editing industry have assured us that alterations of the actual display rate in the 2–3% range (personal communication – C. Shott, Lightwors USA and Digital Images) or expansion and contraction (through interpolation) in the 8% range (personal communication – R. Igo and B. Carpenter, Ampex) can be accomplished without being noticeable to the viewer.

There are two approaches to providing the altered stream of frames to be transmitted to the display stations.

- The altered version of the video can be created on-line. In this case, the I/O bandwidth required from the disk varies with the effective display rate. There are two possible disadvantages of the on-line alteration: (1) the layout of the data on disk is often tuned to one delivery bandwidth, and having to support multiple bandwidths can complicate scheduling and/or require additional buffer storage; and (2) supporting on-the-fly modification may require the expense of specialized hardware to keep up with the demand.
- The altered version of the video is created off-line and stored on disk with the original version. The obvious disadvantage of this approach is the additional disk storage required.

Based on the preceeding discussion, in the remainder of this paper, we assume that we can alter the effective display rate by $\pm 5\%$ without sacrificing video quality, and we consider both the on-line generation approach to providing the altered stream of frames and the off-line approach. In either case we assume that when frames are inserted, the additional frames are some interpolation of existing frames (not simply duplicates). Similarly, when a frame is deleted, the preceding and succeeding frames are altered to reduce the abruptness of the change (e.g., each becomes an interpolation of the original and the deleted frame). For the off-line approach, we include additional considerations in the scheduling policies for limiting the amount of additional disk space required for storing replicates of a video.

3 Batching

One way to reduce the I/O demand (Mb/s) on the storage server is to *batch* requests, for the same object, into a single I/O request to the storage server. The trade-off for the batching approach is the amount of latency experienced by a request versus the corresponding reduction in I/O demand on the storage server. Here, we concentrate on controlling utilization, and more specifically, on controlling utilization of the I/O subsystem. For reasonably busy systems (the only really interesting case), the lower the utilization of a system, the lower is its response time for servicing requests. There are several ways to batch requests into a single I/O stream. (We briefly discuss batching policies here and in the remainder of the paper assume that the policy of *batching by timeout*, as described below, is used.)

In this section, we study (1) batching by size and (2) batching by time out. (Note that the following are simple batching policies introduced mainly for the purpose of exploring the possibilities of combining batching with adaptive piggybacking. Combinations of these batching policies, as well as many other variations on batching, are possible. Some of them have already been studied by Dan et al. (1994a, b) and Ozden et al. (1994); since batching is not the focus of this paper, we do not pursue these any further.) For the remainder of this section, we assume that the request-arrival process for a particular object j, is Poisson with rate λ_j.

3.1 Batching by size

Let B_j be the predefined batching size, and let λ_j be the arrival rate of requests for object j. The system initiates an I/O stream only when B_j such requests accumulate in the system. Let $E[N_j]$ be the expected reduction (due to batching) in the number of I/O streams issued. Then:

$$E[N_j] = B_j - 1 \tag{1}$$

Let L_j be a random variable denoting the latency experienced by each request for object j. Then the expected latency is:

$$E[L_j] = \frac{1}{B_j} \sum_{i=1}^{B_j} \frac{B_j - i}{\lambda_j} = \frac{B_j - 1}{2\lambda_j}\,. \tag{2}$$

Although this policy reduces the I/O demand on the storage server, it can result in long delays for requests, particularly for low-to-moderate arrival rates. (Depending on the network characteristics, it might be wise to batch by size, since this can result in lower network traffic; however, we do not consider network characteristics in this paper, and therefore will assume batching by timeout (described next) in the remainder of the paper.)

3.2 *Batching by timeout*

Another policy we consider is batching by timeout. The timer is set when a request arrives to the storage server and there exists no other outstanding request for the same object j. The system issues an I/O request to the storage server T_j time units after the initiation of the timer. Any request, for the same object, arriving during these T_j time units is *batched* and serviced when the timer expires. Let N_j be a random variable denoting the number of I/O streams saved due to batching, and let p_k^j be the probability of k arrivals during time T_j. Since the arrival process is Poisson, we have:

$$E[N_j] = \lambda_j T_j \tag{3}$$

To evaluate the expected latency experienced by each request, we can view the system as an $M/G/1$ queue with a constant setup time (where the setup time is the duration of the timer T_j) and a deterministic service time distribution with a mean of zero. The expected latency for this type of a system can be found in Takagi (1991):

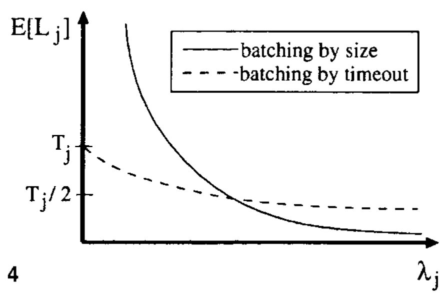
$$E[L_j] = \frac{T_j(2 + \lambda_j T_j)}{2(1 + \lambda_j T_j)} \ . \tag{4}$$

Since $E[N_j] = \lambda_j T_j$, the I/O demand on the secondary storage can be reduced tremendously under moderate-to-high request arrival rates. This is also a reasonable policy for movie-on-demand applications because each request for object j does not experience more than T_j units of delay due to batching. Figure 4 illustrates the expected latency for the two batching policies described in this section.

4 Adaptive piggybacking

We now describe several adaptive piggybacking policies. Consider a storage system, where for each request for an object there exists a display stream and a corresponding I/O stream. The processing nodes use the I/O streams to retrieve the necessary data from disks, possibly modify the data in some manner, and then use the display streams to transmit the data through the network to the appropriate display stations. For example, in Fig. 5, we service display streams 1 and 2 with the corresponding I/O streams 1 and 2. We can reduce the I/O demand on the storage server by using a single I/O stream to service several display streams corresponding to requests for the same object. For example, in Fig. 5, display streams 3 and 4 correspond to requests for the same object and are serviced using a single I/O stream, 3. As stated in Sect. 1, this can be done in a *static* manner, i.e., by batching requests (see Sect. 3) and in a *dynamic* or *adaptive* manner; adaptive piggybacking is the topic of this section. (Note that, depending on the network characteristics, it may

4



5

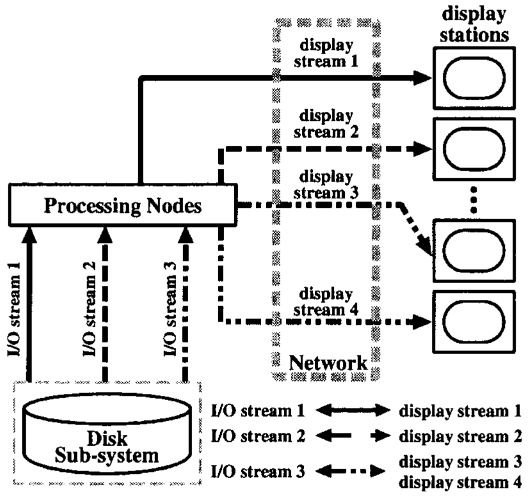


Fig. 4. Expected latency for batching policies for object j

Fig. 5. A simplified view of the system

be wiser to delay "splitting" display streams 3 and 4 (Fig. 5) until the last possible moment, i.e., transmit them through the network as a single stream for as long as possible to obtain a similar reduction in the *network* bandwidth demand as we would in the I/O bandwidth demand; however, we do not consider network characteristics or alternative transmission policies in this paper any further.)

A dynamic approach initiates an I/O stream, for each display stream, on-demand, and then allows one display stream to adaptively piggyback on the I/O stream of another display stream for the same object. We can also view this as a *dynamic merging* of two I/O streams into one. Before the merge, there are two I/O streams, each serving one (or more) display stream(s), where the display streams correspond to two temporally separated displays of the same object. After the merge, there is only one I/O stream, which can service both display streams. Furthermore, the corresponding displays are then "in synch". As described in Sect. 1, this merging can be accomplished by adjusting the display rates of the requests to close the temporal gap between the displays. Although adaptive piggybacking and batching are *not* mutually exclusive techniques, for ease of exposition, in this section we concentrate on adaptive piggybacking policies only. (The results of using adaptive piggybacking policies in conjuction with batching policies are reported in Sect. 6.)

Our goal in this paper is to investigate the benefits, namely, the reduction in I/O bandwidth utilization, attributable to adaptive piggybacking rather than due to a particular storage server architecture. Therefore, we do not specify data layout and/or scheduling schemes. Furthermore, in the following development we do *not* make assumptions about which method is used to achieve different display rates. Instead, in the following derivation, we associate an I/O cost with each I/O stream, where the cost is a function of the corresponding display rate. In other words, the I/O cost for servicing a slow- (or a fast-) rate display can differ from the I/O cost for servicing a normal-rate display. For instance, the speed up (or slow down) can be achieved by replicating data, in which case, the total number of bytes read from disks may differ, depending on the display rate of a stream. If, on the other hand, dropping (or duplication) of frames is used, then the total number of bytes read from disks remains the same, regardless of the display rate of a stream. (Note that, there could be costs other than I/O bandwidth associated with reading data at higher or lower rates, such as additional buffering space and scheduling complexity. For instance, one might consider using only two alternate display rates (e.g., normal and fast) to reduce the scheduling complexity. However, since we do not consider a specific architecture, we will not evaluate such costs in this paper.)

We can view the duration of the object's display as a *continuous* line of finite length and consider the problem of adaptive piggybacking as a decision problem. Given the global state of the system, i.e., the position, relative to the beginning of the display, of each display stream in progress, we must choose a display rate for each of the requests, such that the total average I/O demand on the system is minimized. (We take minimization of the average I/O demand as the objective. Such reductions, if small, would not necessarily be a good measure of how latency is decreased. However we will show that large reductions are obtainable, and therefore the reduction in I/O bandwidth requirements will translate directly to latency reduction.) Since merging is only possible for I/O streams corresponding to displays of the same object, we can consider each group of requests for the same object separately. For the remainder of this section, we consider requests for a particular, *single* object only.

We begin by deriving the general conditions under which I/O streams i and j can be merged in a way that reduces the total I/O demand on the storage server. Initially, we assume that merging can occur at any time during the object display. This assumption is removed at the end of this section. We define the following notation (Fig. 6). for derivation purposes:

$S_k^{'}$	=	The display speed (in frames/s) of display stream k if no attempt to merge is made, where $k \in \{i, j\}$.
S_k	=	The adjusted display speed (in frames/s) of display stream k if merging attempts are made, where $k \in \{i, j\}$.
S_k^{*}	=	The display speed (in frames/s) of display stream k after merging, where $k \in \{i, j\}$.
p_M	=	The total number of frames in a video object.
p_k	=	The current position in an object's display (in frames) of I/O stream k, where $k \in \{i, j\}$.
p_m	=	The position (in frames) in an object's display where I/O streams i and j merge.
$C_k^{'}$	=	The I/O bandwidth (in bits/s) of the I/O stream corresponding to display stream k, with a display speed of $S_k^{'}$.
C_k	=	The I/O bandwidth (in bits/s) of the I/O stream corresponding to display stream k with a display speed of S_k.
C_k^{*}	=	The I/O bandwidth (in bits/s) of the I/O stream corresponding to display stream k, with a display speed of S_k^{*}.
d	=	The distance (in frames) between I/O streams i and j, which is equal to $p_j - p_i$.
d_m	=	The distance (in frames) between the merge point and the current position of j, which is equal to $p_m - p_j$.

Figure 6 represents the duration of an object's display as a continuous line of length p_M. Each display stream, e.g., stream i, is identified by its position p_i in the object's display. The display stream is moving at a particular display speed S_i. To merge I/O streams i and j, we must first ensure that $S_i > S_j$. Secondly, we can define the following distance and cost constraints that can be used in any adaptive piggybacking policy to identify merging opportunities, i.e., to determine, whether or not it is possible and cost-effective to merge I/O streams i and j. The cost constraint ensures that the total I/O demand (measured in bits read from the disk) with merging is less than the total I/O demand without merging. This I/O cost constraint is:

$$\frac{dC_i}{S_i} + \frac{d_m C_i}{S_i} + \frac{d_m C_j}{S_j} + \frac{(p_M - d - d_m - p_i)C_j^*}{S_j^*} \leq \frac{(p_M - p_i)C_i^{'}}{S_i^{'}} + \frac{(p_M - p_i - d)C_j^{'}}{S_j^{'}} . \tag{5}$$

(Since I/O stream i is merged with j, after the merge, only the I/O cost of stream j need be considered beyond the merge point.) The above constraint is only meaningful when the number of bits read from the disk is *not* independent of the display rate, i.e., in our case, it is meaningful only when replication is used. Otherwise, any merging prior to the end of a display results in savings. Then Eq. 6 becomes the only constraint, namely, the object length (or duration of its display) is finite and hence requires the following distance constraint:

$$p_i + d + d_m \leq p_M . \tag{6}$$

Finally, the merge-time constraint is:

$$\frac{d + d_m}{S_i} = \frac{d_m}{S_j} . \tag{7}$$

Let d_1 be the maximum d such that the I/O cost condition in Eq. 5 is satisfied. We obtain d_1 by using Eq. 7 to obtain $d_m = d\left(\frac{S_j}{S_i - S_j}\right)$ and then setting the equality in Eq. 5. Hence:

$$d_1 = \frac{\left[\frac{(p_M - p_i)C_i^{'}}{S_i^{'}} + \frac{(p_M - p_i)C_j^{'}}{S_j^{'}} - \frac{(p_M - p_i)C_j^*}{S_j^*}\right]}{\left[\left(\frac{C_i}{S_i} - \frac{C_j^*}{S_j^*} + \frac{C_j^{'}}{S_j^{'}}\right) + \left(\frac{S_j}{S_i - S_j}\right)\left(\frac{C_i}{S_i} + \frac{C_j}{S_j} - \frac{C_j^*}{S_j^*}\right)\right]} . \tag{8}$$

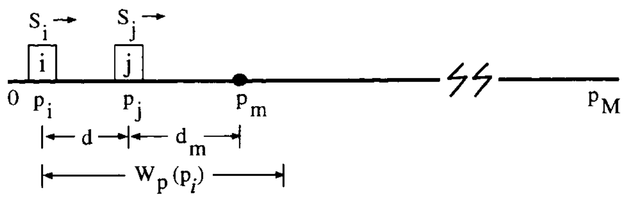


Fig. 6. State of the system

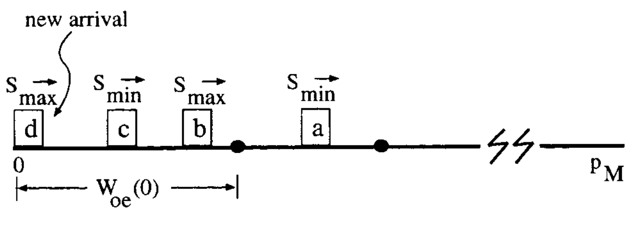


Fig. 7. Scenario of the odd-even reduction policy

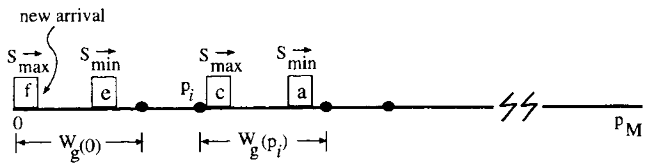


Fig. 9. Scenario of the greedy merging policy

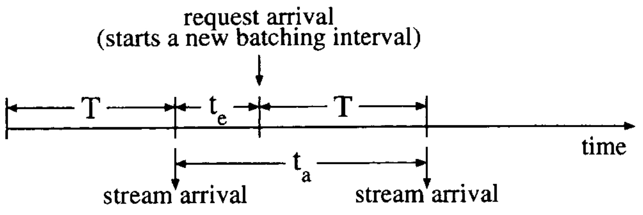


Fig. 10. Arrival of I/O streams after a delay

Let d_2 be the maximum d such that the distance constraint in Eq. 6 is satisfied. Again, d_2 can be obtained by substituting the expression for d_m in Eq. 6 and solving for equality:

$$d_2 = \frac{(p_M - p_i)(S_i - S_j)}{S_i} . \qquad (9)$$

Let d^* be the maximum distance between two I/O streams such that merging these two streams (at d_m) results in a reduction of I/O demand on the storage server. Therefore:

$$d^* = \min(d_1, d_2) . \qquad (10)$$

We can now apply this result to the various adaptive piggybacking policies, which we describe next. Our goal is to find adaptive piggybacking policies that have a significantly lower expected I/O demand than that of the baseline policy. (A baseline policy is one that does not use display adjustment – each I/O stream is displayed at its normal display rate.) Thus in the remainder of this section we present several adaptive piggybacking or "merging" policies, which differ in how the stream merging decisions are made, as well as in how many merges are attempted. These policies are not intended to be "optimal", but rather to be simple heuristics, designed to investigate the possible benefits that can be gained from adaptive piggybacking; these benefits are discussed in Sect. 6.

We make the following observations about the display adjustment decisions. Consider again the system state depiction in Fig. 6. Clearly, the only stochastic events in the system are the arrival points. Such events as merging of two streams or end of a display are predictable. Hence, an optimal policy can evaluate all possible display rates, make appropriate decisions with respect to minimizing the average system I/O demand, and then not re-evaluate these decisions until the next arrival point. However, this would be computationally intensive and hence impractical. Instead, we consider a class of (simpler) policies that make speed adjustments when one of the following four types of events occurs: (1) *arrival*, (2) *merge*, (3) *dropoff*, and (4) *window crossing*. An *arrival* event corresponds to an initiation of a new I/O stream. A *merge* event corresponds to the merge of two I/O streams, and a *dropoff* event corresponds to the end of a display of an object – a "departure" of an I/O stream. A *window-crossing* event refers to passing the boundary of a catch-up window, which is illustrated in Fig. 6. We define a *catch-up window*, $W_p(p_i)$, for policy p to be the maximum possible distance between stream i and stream j, ahead of stream i, such that "profitable" merging is possible. $W_p(p_i)$ is computed relative to position p_i in the display of an object. We shall see shortly how the catch-up window is used in the merging policies. $W_p(p_i)$ can be computed using Eq. 10.

The sooner merging occurs in the object's display, the more resources (disk bandwidth, buffer space, etc.) can be conserved and used by the storage system to service other requests. Hence, in the remainder of this paper we shall assume the maximum possible deviations from the normal speed both, for the slower and faster than normal display rates. In other words, we limit our policies to consider only three possible display rates: (1) the slowest rate, S_{min}, (2) the normal rate, S_n, and (3) the fastest rate, S_{max}. The corresponding I/O demands, or cost rates, are C_{min}, C_n, and C_{max}.

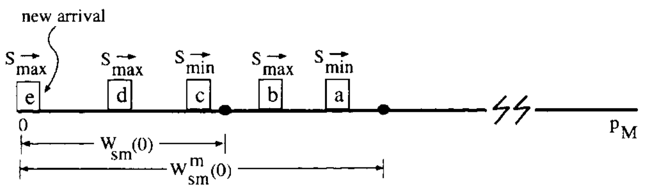


Fig. 8. Scenario of the simple merging policy

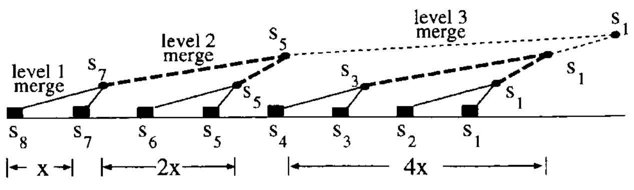


Fig. 11. Merging pattern for streams under the greedy policy

4.1 Baseline policy

This is the normal situation. When requests arrive, there is no attempt to adjust the display rates, i.e., all requests are assigned the normal display speed of S_n and there are no merging events in the system. Note that the lack of merging does not exclude the possibility of initial batching.

4.2 Odd-even reduction policy

A simple display rate adjustment policy that attempts to reduce I/O demand by at most 50% is the *odd-even reduction* policy. The basic approach is to pair up consecutive arrivals for merging whenever possible; the algorithm is given below. Let us define $W_{oe}(0)$, measured relative to the beginning of an object's display (Fig. 7), to be the catch-up window for the odd-even reduction policy. The algorithm for odd-even reduction is:

```
**Algorithm** odd-even reduction policy
**Case**: arrival of stream $i$
    **If** ((no stream, in front, is within $W_{oe}(0)$ frames) **or**
        (stream immediately in front is moving at $S_{max}$))
        $S_i = S_{min}$;
    **else**
        $S_i = S_{max}$;
**Case**: merge of $i$ and $j$
    drop stream $i$;
    $S_j = S_n$;
**Case**: window crossing, $W_{oe}(0)$ (by stream $i$)
    **If** ($S_i == S_{min}$) **and**
        (no stream behind, in $W_{oe}(0)$, moving at $S_{max}$)
        $S_i = S_n$;
    **else**
        $S_i$ is unchanged;
**end**
```

In this merging policy, each original stream participates in at most a single merge. Such simplicity in merging decisions or such a "limited" form of merging could be advantageous if the particular display adjustment technique used is complex or resource consuming; an approach to "limiting" the merges is discussed in Sect. 4.5. The "slowing down" of a new arrival (in this algorithm) is, of course, motivated by the anticipation of future arrivals.

Figure 7 illustrates one possible scenario of this policy. When an I/O stream d arrives to the system, I/O stream c is still in the catch-up window $W_{oe}(0)$ "moving" at the display speed of S_{min}. In this case, the display speed of request d is set to S_{max}. Likewise, when stream b arrives to the system, I/O stream a is within the catch-up window $W_{oe}(0)$. Therefore, the display speed of b is set to S_{max}. In this scenario, I/O streams a and b merge into a single I/O stream, and streams c and d also merge into a single I/O stream.

$W_{oe}(0)$ can be computed using Eq. 10, where the values of d_1 and d_2 can be found, using Eqs. 8 and 9, respectively, by simply setting $p_i = 0$, $C_i = C_{max}$, $S_i = S_{max}$, $C_j = C_{min}$, $S_j = S_{min}$, $C_i' = C_j' = C_n$, $S_i' = S_j' = S_n$, $C_i^* = C_n$, $S_i^* = S_n$. Then we have:

$$d_1 = \frac{\left[\frac{p_M C_n}{S_n}\right]}{\left[\left(\frac{C_{max}}{S_{max}}\right) + \left(\frac{S_{min}}{S_{max}-S_{min}}\right)\left(\frac{C_{max}}{S_{max}} + \frac{C_{min}}{S_{min}} - \frac{C_n}{S_n}\right)\right]} \tag{11}$$

$$d_2 = \frac{p_M(S_{max} - S_{min})}{S_{max}} \tag{12}$$

4.3 Simple merging policy

As in the case of the odd-even reduction policy, we first define $W_{sm}(0)$ to be the catch-up window for the *simple merging* policy, measured relative to the beginning of an object's display (Fig. 8). In addition, we define $W_{sm}^m(0)$ to be the maximum merging window for the simple merging policy, also measured relative to the beginning of an object's display (Fig. 8). $W_{sm}^m(0)$ indicates the latest possible position where two streams can merge, i.e., if i arrives to the system and finds j $W_{sm}(0)$ frames ahead of it, then i and j can still merge. Moreover, they will merge at the right-hand boundary of $W_{sm}^m(0)$ (Fig. 8). The basic rationale behind the simple merging policy is to assign streams to "merging groups", where one stream, e.g., stream i, initiates the group, and all streams that arrive to the system while stream i is in $W_{sm}(0)$ eventually merge with stream i. The last stream will merge "into the group" before leaving $W_{sm}^m(0)$. The algorithm for the simple merging policy is:

```
**Algorithm** simple merging policy
**Case**: arrival of stream $i$
    **If** no stream within $W_{sm}(0)$ is moving at $S_{min}$
        $S_i = S_{min}$;
    **else**
        $S_i = S_{max}$;
**Case**: merge of $i$ and $j$
    drop stream $i$;
    $S_j = S_{min}$;
**Case**: window crossing, $W_{sm}^m(0)$
    $S_i = S_n$;
**end**
```

The rationale for keeping the display speed at S_{min} until the right boundary of $W_{sm}^m(0)$ is crossed is to allow all streams in the merging group to eventually merge. Unlike in the odd-even reduction policy, in this merging policy, each original stream could eventually merge with more than one other (original) stream. However, this merge might occur in a "later portion" of a video, as compared to the odd-even case. We expect that the relative performance of these algorithms will depend on the distribution of the arrival process.

Figure 8 illustrates one possible scenario under this policy. When I/O stream c arrives to the system, I/O stream a has already moved outside of the catch-up window $W_{sm}(0)$. Therefore, the display speed of I/O stream c is set to S_{min}. When stream b (streams d and e) arrives to the system, stream a (stream c) is within the catch-up window $W_{sm}(0)$. Therefore, their display speeds are set to S_{max}. In this scenario, stream b eventually merges with stream a, and streams d and e merge with stream c. All merges occur within $W_{sm}^m(0)$.

Both $W_{sm}(0)$ and $W^{m}_{sm}(0)$, can be computed using Eq. 10. The values of d_1 and d_2 can be found using Eqs. 8 and 9, respectively, by simply setting $p_i = 0$, $C_i = C_{max}$, $S_i = S_{max}$, $C_j = C_{min}$, $S_j = S_{min}$, $C_i' = C_j' = C_n$, $S_i' = S_j' = S_n$, $C_i^* = C_n$, $S_i^* = S_n$. Then we have:

$$d_1 = \frac{\left\lceil \frac{p_M C_n}{S_n} \right\rceil}{\left[\left(\frac{C_{max}}{S_{max}}\right) + \left(\frac{S_{min}}{S_{max}-S_{min}}\right)\left(\frac{C_{max}}{S_{max}} + \frac{C_{min}}{S_{min}} - \frac{C_n}{S_n}\right)\right]} \tag{13}$$

$$d_2 = \frac{p_M(S_{max} - S_{min})}{S_{max}} \tag{14}$$

and

$$W_{sm}(0) = \min(d_1, d_2) \tag{15}$$

$$\begin{aligned} W^{m}_{sm}(0) &= W_{sm}(0) + d_m \\ &= W_{sm}(0)\left(\frac{S_{max}}{S_{max} - S_{min}}\right) . \end{aligned} \tag{16}$$

4.4 Greedy policy

If the request arrival rate to the system is moderate to high, then it is advantageous to merge requests as early as possible, thereby reducing the I/O demand sooner. Both the *odd-even reduction* and the *simple merging* policies attempt to accomplish this. But, it is still possible to further merge I/O requests, which have accomplished some form of "early merging". The greedy policy attempts to merge I/O requests as many times as possible, during the entire duration of an object's display. Therefore, in addition to the *initial* catch-up window $W_g(0)$, measured relative to the beginning of an object's display, we shall also use $W_g(p_i)$, a catch-up window measured relative to position p_i in an object's display. The greedy policy uses this "current" catch-up window (described below) as an indication of opportunity for further merging.

The greedy policy works as follows. Upon arrival of a request for the object, the speed adjustment is performed as in the odd-even reduction policy. If, on crossing the catch-up window, the stream determines that it has not yet been paired up for merging, then it checks $W_g(W_g(0))$ for possibility of merging with some stream in front. When merging occurs at position p_i, a new catch-up window $W_g(p_i)$ is computed. If there is no I/O request within this window, the speed of the request is set to S_n. If there are some requests within the catch-up window $W_g(p_i)$ and the I/O request immediately in front has a display speed of S_n, then that request's speed is set to S_{min}, and the speed of the request at position p_i is set to S_{max}. In algorithmic form, the greedy policy is described as follows:

```
**Algorithm** greedy policy
**Case**: arrival of stream $i$
  **If** ((no stream in front is within $W_g(0)$ frames) **or**
      (stream immediately in front has display speed $S_{max}$))
      $S_i = S_{min}$;
  **else**
      $S_i = S_{max}$;
**Case**: merge of streams $i$ and $j$
  drop stream $i$;
  compute $W_g(p_j)$, where $p_j$ is the position of stream $j$;
  **If** ((no stream $k$ with speed $S_n$ is immediately in front
      within $W_g(p_j)$ frames)
      $S_j = S_n$;
  **else**
      $S_k = S_{min}$;
      $S_j = S_{max}$;
**Case**: window crossing, $W_g(0)$ (by stream $i$)
  compute $W_g(p_i)$;
  **If** (($S_i == S_{max}$) **or**
      ($S_j == S_{max}$, where $j$ is stream immediately
                       behind $i$ in $W_g(0)$))
      $S_i$ is unchanged;
  **else If** (stream $k$ with speed $S_n$ immediately in front
      is within $W_g(p_i)$)
      $S_k = S_{min}$;
      $S_i = S_{max}$;
  **else**
      $S_i = S_n$;
**end**
```

Clearly, we expect the greedy policy to perform better than the odd-even or the simple-merge policies, since it attempts merges during the entire duration of an object's display. (On the other hand, as already mentioned in Sect. 4.2, a greater simplicity in merging decisions or a more limited form of merging could be advantageous if the particular display adjustment technique used is complex or resource consuming.)

Figure 9 illustrates one possible scenario of this policy. I/O streams b and d (not shown) have been already merged with I/O streams a and c, respectively. This occurs in the first catch-up window $W_g(0)$. After the merging of I/O streams d and c, I/O stream c attempts to merge with I/O stream a in catch-up window $W_g(p_i)$. At the same time, a newly arrived I/O stream f attempts to merge with I/O stream e, which is within its catch-up window $W_g(0)$.

$W_g(p_i)$ can be derived from Eq. 10. The values of d_1 and d_2 (now both functions of the i's current position, p_i) can be found by simply setting $C_i = C_{max}$, $S_i = S_{max}$, $C_j = C_{min}$, $S_j = S_{min}$, $C_i' = C_j' = C_n$, $S_i' = S_j' = S_n$, $C_i^* = C_n$, $S_i^* = S_n$. Then we have:

$$d_1 = \frac{\left\lceil 2\frac{(p_M - p_i)C_n}{S_n} + \frac{p_M C_n}{S_n} \right\rceil}{\left[\left(\frac{C_{max}}{S_{max}}\right) + \left(\frac{S_{min}}{S_{max}-S_{min}}\right)\left(\frac{C_{max}}{S_{max}} + \frac{C_{min}}{S_{min}} - \frac{C_n}{S_n}\right)\right]} \tag{17}$$

$$d_2 = \frac{p_M(S_{max} - S_{min})}{S_{max}} . \tag{18}$$

4.5 Limited merging

At this point we remove the assumption that merging can occur at any time. If replication of data is necessary to perform the display rate alteration, then we must consider another parameter, namely, the amount of additional disk space that would be necessary to store replicated data. As already mentioned, there is a tradeoff between the amount of additional storage necessary to replicate data and the reduction in I/O demand that can result. (Note that, we do not necessarily have to store three different versions of an object, each corresponding to a different display rate. For instance, in the simple merging policy, we only need the slow and the fast versions while in the maximum merging window, $W^m_{sm}(0)$, and only the normal version outside of the maximum merging window.) We can evaluate the above tradeoff by placing an additional constraint on the merging policies, namely, the constraint of a maximum merging point (in the display of an object). In other words, we can control the amount of data that must be replicated by allowing merging only if it can occur within a specified amount of time or, rather, within a certain distance (in frames) from the beginning of an object's display. We refer to this distance as p^{max}_m. Consider again Fig. 6 and Eqs. 5–7 which describe the distance and cost constraints that must be met to attempt merging of two display streams. To control the amount of replication, we enforce the additional constraint that the merge must occur before p^{max}_m rather than before p_M, i.e., $p_m \le p^{max}_m$. Thus, Eqs. 6 and 9 are replaced by Eqs. 19 and 20, respectively, as follows:

$$p_i + d + d_m \le p^{max}_m \tag{19}$$

$$d_2 = \frac{(p^{max}_m - p_i)(S_i - S_j)}{S_i} \; . \tag{20}$$

All other equations can remain unchanged. Of course, these modifications must be carried through for *all* the adaptive piggybacking policies described above. (Results of studies of adaptive piggybacking in conjunction with batching, both with and without a constraint on the maximum merging point, are reported in Sect. 6; performance analysis of these policies can be found in Sect. 5.)

5 Performance analysis

In this section we present analytic solutions for computing the I/O demand on a storage server that uses adaptive piggybacking policies in conjunction with batching. We define the following notation (also see Fig. 10) used in the derivation of this section. All computation is done with respect to a particular multimedia object j. Unless otherwise stated, we drop the subscript j for simplicity of illustration.

p_M	= number of frames in a movie.
T	= batching delay time (deterministic).
λ	= mean arrival rate.
t_e	= mean time between the end of one batching delay interval and the beginning of the next one (Fig. 10).
t_a	= random variable representing the time between I/O stream initiation.
$W_p(p_i)$	= the catch-up window for policy p, relative to position p_i.
$W^m_p(p_i)$	= the maximum merging window for policy p, relative to position p_i.
BW_p	= the mean total I/O demand under policy p (bits/s).

The equations for $W_p(p_i)$ and $W^m_p(p_i)$ where p could be the odd-even (oe), simple (sm), or greedy (g) policy, can be found in Sect. 4; these equations already allow for the limited merging case which limits the maximum allowed merging time.

First let us derive the density function of t_a, which is the interarrival time between two streams arriving to the storage server. Since the request arrival process is Poisson with rate λ, $t_e = \frac{1}{\lambda}$. Therefore, the density function of t_a is:

$$f_{t_a}(x) = \lambda e^{-\lambda(x-T)} \qquad \text{for } x \ge T \; . \tag{21}$$

Since the normal duration of a movie object is p_M/S_n, the expected number of I/O streams, N, that the storage server has to support is:

$$N = \left(\frac{p_M}{S_n}\right)\left(\frac{1}{\int_T^\infty x f_{t_a}(x)dx}\right) = \frac{p_M/S_n}{1/\lambda + T} \; . \tag{22}$$

5.1 Analysis of the baseline policy

We begin with the analysis of the baseline policy, which is very simple, since there are no merges and each stream carries a fixed cost of C_n. The expected bandwidth demand is:

$$BW_b = NC_n = \frac{p_M/S_n}{1/\lambda + T} C_n \; . \tag{23}$$

The expected bandwidth demand without batching can be obtained by setting $T = 0$.

5.2 Analysis of the odd-even reduction policy

The behavior of the odd-even policy is such that pairs of consecutive I/O streams are statistically identical. We can therefore analyze the mean I/O demand for one such pair and then compute the average I/O bandwidth by multiplying half the rate of intensity of I/O streams by the average demand per pair. Under the odd-even policy, merges are possible for certain ranges of interarrival times and batching delays. Consider two consecutive streams s_1 and s_2 that arrive to the system x time units apart (assume that s_2 is the lagging stream). Assume for the moment that it is possible for these streams to merge, and let t_m be the time it would take s_1 and s_2 to merge, computed from the arrival time of s_2. Let t_f be the time from the merge point of these two streams until the end of the object's display. Then:

$$t_m = \frac{xS_{min}}{S_{max} - S_{min}} \tag{24}$$

$$t_f = \frac{p_M - (t_m + x)S_{min}}{S_n} \; . \tag{25}$$

Note that merging is possible only if two streams arrive within the catch-up window $W_{oe}(0)$. Therefore, the combined I/O demand for streams s_1 and s_2, given that they arrived x time units apart and that they can merge (i.e., that $x \le \frac{W_{oe}(0)}{S_{min}}$) is:

$$BW^m_{oe} = (t_m + x)C_{min} + t_m C_{max} + t_f C_n \tag{26}$$

The three costs correspond to the bandwidth demands of: (a) the leading stream s_1, first moving at display speed S_{min}; (b) the trailing stream s_2, first moving at display speed S_{max};

and (c) the remaining I/O demand, after merging, and continuing display at the speed of S_n.

Similarly, if $x > \frac{W_{oe}(0)}{S_{min}}$, then the I/O demand of the pair of streams is:

$$BW_{oe}^{nm} = 2 * \left[\frac{W_{oe}(0)}{S_{min}} C_{min} + \frac{p_M - W_{oe}(0)}{S_n} C_n \right] . \qquad (27)$$

The expression corresponds to each of the streams at first having a display speed of S_{min} and, after moving beyond the catch-up window, reseting the display speed to S_n. At this point, we can compute BW_{oe}, the total mean bandwidth demand in the system:

$$BW_{oe} = \frac{\int_T^{\frac{W_{oe}(0)}{S_{min}}} (BW_{oe}^m * \frac{N}{2} * f_{t_a}(x))\, dx}{\frac{p_M}{S_n}} + \frac{\int_{\frac{W_{oe}(0)}{S_{min}}}^{\infty} (BW_{oe}^{nm} * \frac{N}{2} * f_{t_a}(x))\, dx}{\frac{p_M}{S_n}} . \qquad (28)$$

5.3 Analysis of the simple merging policy

The analysis of the simple merging policy is similar to that of the odd-even policy except that, instead of looking at pairs of streams, we consider "merging groups" of streams, i.e., groups of streams that eventually all merge together (see Sect. 4). Similarly to the odd-even policy, we note that all "merging groups" are statistically identical. Hence we can analyze the mean I/O demand for one such group and compute the average I/O demand by multiplying the rate of intensity of such groups by the I/O demand for each group. Under the simple merging policy, merging is possible if, upon initiation of a stream, there exists another stream within the catch-up window $W_{sm}(0)$, that is moving at speed S_{min}. Let β be the number of streams, within the window $W_{sm}(0)$ that can (eventually) be merged. We call this set of streams a "merging group". We approximate β by:

$$\beta = \max \left\{ \lfloor \frac{W_{sm}(0)/S_{min}}{T + 1/\lambda} + 1 \rfloor , 2 \right\} . \qquad (29)$$

The first component corresponds to the number of streams that can fall within window $W_{sm}(0)$. By setting $\beta \geq 2$, we consider the (merging) effect when at least two streams are available for merging.

Assume that all streams in a merging group are separated by time x and that there are β merging streams within the catch-up window $W_{sm}(0)$. The second stream needs t_m (or $\frac{xS_{min}}{S_{max} - S_{min}}$) time units to catch up to the leading (first in the group) stream, the third stream needs $2t_m$ time units to catch up, etc. The leading stream keeps the display speed at S_{min} until it reaches position $W_{sm}^m(0)$. Then the display speed is reset to S_n. Therefore, the amount of time during which the leading stream has a display speed of S_n is:

$$t_f = \frac{p_M - W_{sm}^m(0)}{S_n} . \qquad (30)$$

The I/O demand for the merging group, given that the streams are separated by time x and that merging is possible, can be expressed as:

$$BW_{sm}^m = \frac{W_{sm}^m(0)}{S_{min}} C_{min} + C(\beta) t_m C_{max} + t_f C_n \qquad (31)$$

where $C(\beta) = \beta(\beta - 1)/2$. The cost terms correspond to the cost of the leading stream moving at S_{min} and all other streams, originally within the catch-up window $W_{sm}(0)$, moving at speed S_{max}, trying to catch-up. The last cost term represents the remaining time after the last merge, when the leading stream moves at speed S_n.

If merging is not possible for a given interarrival time x, then the I/O demand for the merging group is:

$$BW_{sm}^{nm} = \beta * \left[\frac{W_{sm}^m(0)}{S_{min}} C_{min} + \frac{p_M - W_{sm}^m(0)}{S_n} C_n \right] . \qquad (32)$$

The expected I/O demand for the simple merging policy is:

$$BW_{sm} = \frac{\int_T^{\frac{W_{sm}(0)}{S_{min}}} (BW_{sm}^m * \frac{N}{\beta} * f_{t_a}(x))\, dx}{\frac{p_M}{S_n}} + \frac{\int_{\frac{W_{sm}(0)}{S_{min}}}^{\infty} (BW_{sm}^{nm} * \frac{N}{\beta} * f_{t_a}(x))\, dx}{\frac{p_M}{S_n}} . \qquad (33)$$

5.4 Analysis of the greedy policy

The performance analysis of the greedy policy is more complex. Let us first refer to Fig. 11 and consider the merging pattern. This figure depicts a system with eight streams. All of them start out x time units apart, and eventually, all eight streams can be merged into one. For the first-level merge (Fig. 11), the system reduces the number of streams by half, but all remaining streams (s_1, s_3, s_5, s_7) are $2x$ time units apart. After the second-level merge, only two streams remain, s_1 and s_5, and they are $4x$ time units apart. With this observation, let l be the highest level of merges under the greedy policy. The expression for l is:

$$l = \max \{ k : g(k) > 0 \} \qquad (34)$$

where

$$g(k) = p_M - 2^{(k-1)} \tau \left[S_n + \left(\frac{S_{min}}{S_{max} - S_{min}} \right) S_{min} \right] \qquad (35)$$

and

$$\tau = \int_T^{\frac{W_g(0)}{S_{min}}} x f_{t_a}(x)\, dx = \left[T + \frac{1}{\lambda} \right] - \left[\frac{(S_{min} + \lambda W_g(0)) e^{(-\lambda \frac{W_g(0) - S_{min} T}{S_{min}})}}{\lambda S_{min}} \right] . \qquad (36)$$

Given that the streams can go through l levels of merges, the leading stream, after the last merging point, will have t_f time units of display left, at a speed of S_n, where t_f is:

$$t_f = \left[p_M - \tau \left(1 + \frac{S_{min}}{S_{max} - S_{min}} \right) S_{min} - \sum_{j=2}^{l} 2^{j-1} \tau \left[S_n + \left(\frac{S_{min}}{S_{max} - S_{min}} \right) S_{min} \right] \right] \left[\frac{1}{S_n} \right] \qquad (37)$$

where the first term represents the remaining frames of the object, after the last merging event, displayed at the speed of S_n.

Given that the interarrival time between streams participating in an l-level merge is x, and that there are $l \geq 2$ levels of merges, the I/O demand is:

$$BW_g^m = \left\lceil \frac{(t_m + x)C_{min} + t_m C_{max}}{p_M/S_n} \right\rceil \left\lceil \frac{N}{2} \right\rceil + \sum_{j=2}^{l} \left[h(j) \left\lceil \frac{S_n}{p_M} \right\rceil \left\lceil \frac{N}{2^j} \right\rceil \right] + \left\lceil \frac{t_f C_n}{p_M/S_n} \right\rceil \left\lceil \frac{N}{2^l} \right\rceil \qquad (38)$$

where

$$h(j) = 2^{(j-1)}\tau \left[C_n + \left(\frac{S_{min}}{S_{max} - S_{min}} \right) (C_{min} + C_{max}) \right] . \qquad (39)$$

The first term in Eq. 38 represents the bandwidth demand for the first-level merge while pairing up $\frac{N}{2}$ streams. The second term represents the bandwidth demand for the second level, etc., until the l^{th} level merge while pairing up $\frac{N}{2^l}$ pairs. (For level two and up, the leading stream first moves at S_n because it finishes merging earlier than the next pairs of trailing streams. When the trailing streams finally finish their merge, its display speed is reset from S_n to S_{min}, while the trailing stream resulting from the merge attempts to catch up at the speed of S_{max}.) The third term represents the bandwidth demand for the leading stream, moving at the display speed of S_n, all the way until the end of the object's display.

If merges are *not* possible, given x, then the I/O demand is:

$$BW_g^{nm} = \left\lceil \frac{\frac{W_g(0)}{S_{min}} C_{min} + \frac{p_M - W_g(0)}{S_n} C_n}{p_M/S_n} \right\rceil N . \qquad (40)$$

Unconditioning on the interarrival time x, we have:

$$BW_g = \int_T^{\frac{W_g(0)}{S_{min}}} (BW_g^m * f_{t_a}(x))\, dx + \int_{\frac{W_g(0)}{S_{min}}}^{\infty} (BW_g^{nm} * f_{t_a}(x))\, dx . \qquad (41)$$

Finally, we constrain the bandwidth demand of the odd-even policy to be the upper bound for the bandwidth demand of the greedy policy:

$$BW_g = \min(BW_g, BW_{oe}) . \qquad (42)$$

5.5 Validation of analytic results

In conclusion of this section, we validate our our analytic results by comparing them to results obtained from simulation. Comparisons of all three policies in conjunction with batching by timeout are depicted in Figs. 12–14, in which "delay" refers to the batching interval (in minutes) and each curve represents the percentage improvement in I/O demand, as compared to the baseline policy. The comparisons indicate that the largest divergence from the simulation occurs when the arrival rate is low. The analytic results match the simulation closely when the arrival rate is moderate to high. This is sufficient for our purposes since we are interested in applying our techniques to video objects with relatively high access rates, i.e., *popular* objects.

6 Discussion of results

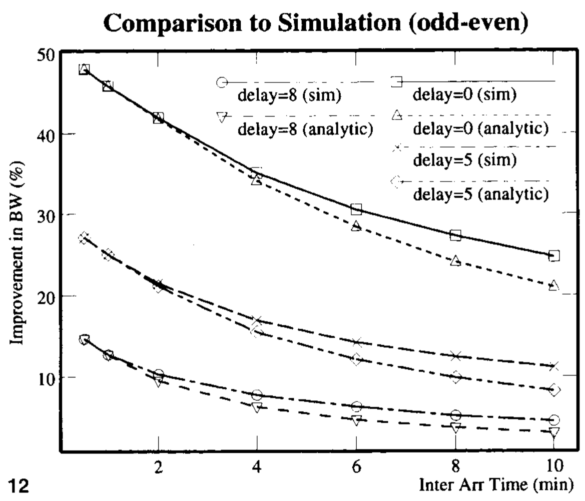
In this section we present results of studies of adaptive piggybacking policies in conjunction with batching policies. To avoid degradation in display quality, we assume that the adjusted rates S_{min} and S_{max} are within 5% of the normal display rate, S_n. In the remainder of this discussion, we use the the following values for the parameters presented in Sect. 4:

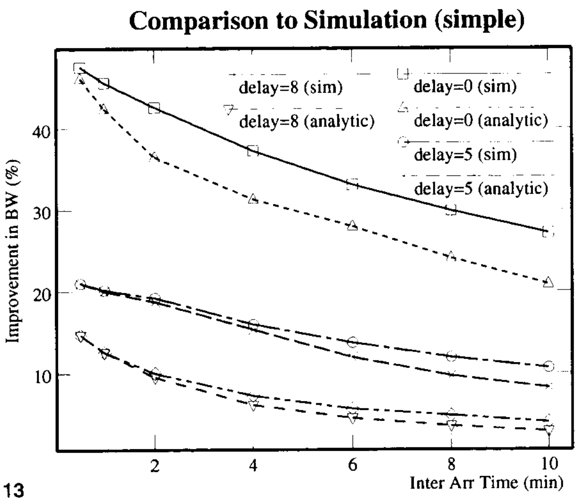
S_{min}	= 28.5 frames/s
S_n	= 30.0 frames/s
S_{max}	= 31.5 frames/s
C_{min}	= 1.425 Mbits/s
C_n	= 1.5 Mbits/s
C_{max}	= 1.575 Mbits/s

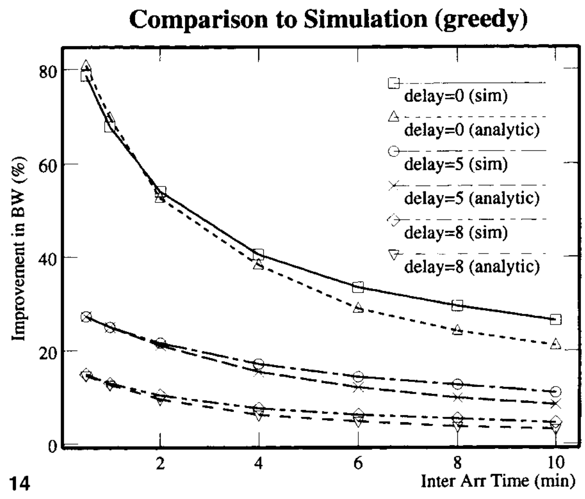
Batching by timeout is used as the batching policy in all results presented in this section. The delay time is varied between 0 and 10 min and the mean interarrival time (between consecutive requests for the same movie) is varied between 0.5 and 10 min. In the following discussion, we consider the total average I/O bandwidth demand on the storage server as the measure of interest. More specifically, in each graph, we present the percentage improvement of the various policies, as compared to the baseline policy. For ease of exposition, we initially assume no restrictions on the maximum allowed merging time. At the end of this section, we consider the effect of restricting merges to occur within a specified time interval.

We first consider the effects of batching, i.e., the decrease in I/O demand on the storage server due to batching and the corresponding increase in the average latency for starting the service of a request. This comparison is illustrated in Fig. 15, in which the interarrival time is kept at 4 min and the batching delay is varied between 0 and 10 min. This graph indicates that, as the batching delay increases, the decrease in I/O demand quickly shows diminishing returns while the average latency, which grows almost linearly with the batching delay (Eq. 4), continues to grow.

Next, we compare the adaptive piggybacking policies, but without batching. The results of this comparison are depicted in Fig. 16 as a percentage improvement over the baseline policy, where the interarrival time is varied between 0.5 and 10 min. This graph indicates that the odd-even policy results in a significant reduction in I/O demand. Recall that the odd-even policy allows each I/O stream to participate in (at most) a single merge, and hence it can result in (at most) a 50% decrease in I/O demand. For the cases presented in Fig. 16, the reduction in I/O demand compared to the baseline policy ranges from 47.92%, corresponding to a fairly small interarrival time of 0.5 min to 20.92%, corresponding to a fairly large interarrival time of 10 min. Further reduction can be achieved by allowing each I/O stream to participate in multiple merges, for instance, by using the greedy policy. The results for the greedy policy (without batching) are also illustrated in Fig. 16, where we achieve a further reduction in I/O demand. Again, compared to the baseline policy, the results for the greedy policy range from 81.0% for the fairly small interarrival time of 0.5 min to 20.92% for the fairly large interarrival time of 10 min. The results are qualitatively similar when batching is used in conjunction







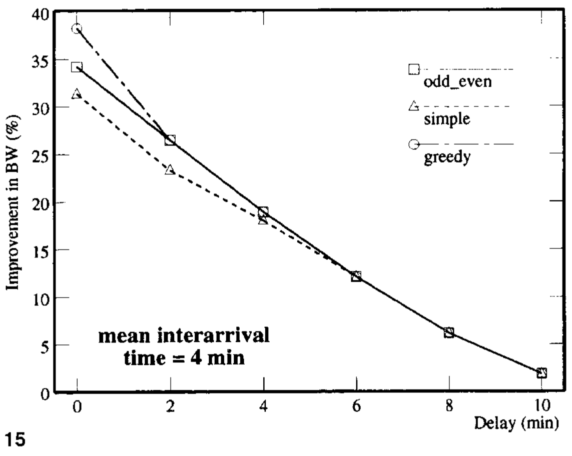


Fig. 12. The odd-even policy with comparison to simulation
Fig. 14. The greedy policy with comparison to simulation

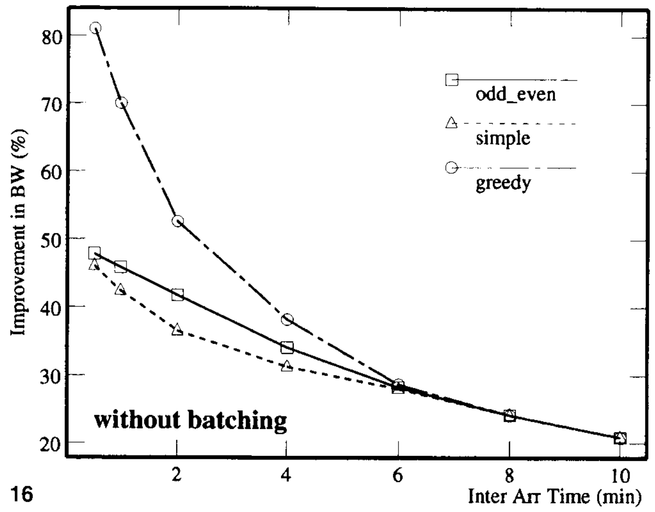
Fig. 13. The simple policy with comparison to simulation
Fig. 15. Varying delay

with adaptive piggybacking. This is illustrated in Figs. 17a, b which correspond to batching intervals of 2 and 5 min respectively. (Clearly, as the batching interval increases, the range of workloads over which these policies exhibit significantly different behavior decreases.)

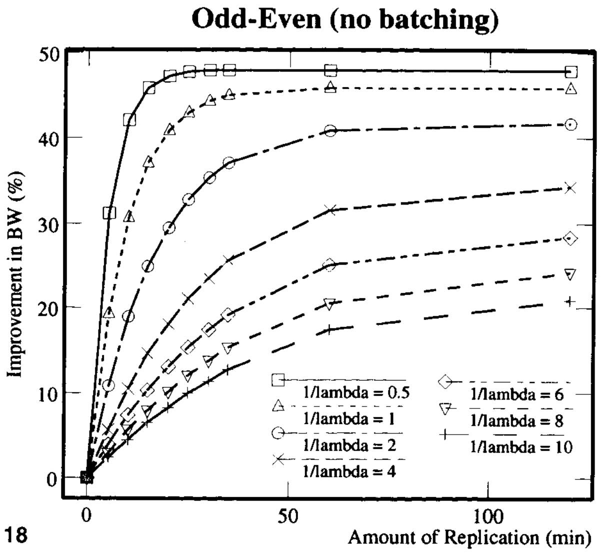
Although a greater reduction in I/O demand is achieved by the greedy policy, than by the odd-even policy, allowing more than a single merge per I/O stream could be costly in terms of other resources. For instance, if display rate alterations are done through replication of appropriate data, then we can reduce the amount of replication needed to support adaptive piggybacking by constraining the merges to occur before a specified maximum allowed merging time. Therefore, in Fig. 18 we investigate the benefits of adaptive piggybacking under an additional constraint of a limited merging time. In this figure, the percentage reduction in I/O demand, as compared to the baseline policy, is depicted as a function of the maximum allowed merging time (each curve corresponds to a different interarrival time). We obtained the results using the odd-even policy, without batching. Qualitatively similar results can be obtained for systems with batching delays as well as other adaptive piggybacking policies; however, in the interests of brevity, we do not illustrate them here.

As expected, given fairly small interarrival times, most of the reduction in I/O demand (Fig. 16), can be achieved using relatively small maximum merging times. The implication is that, if replication of data is used to support merging, then most of the benefits of "unrestricted" merging can be achieved with relatively little increase in disk storage cost. For instance, given an interarrival time of 0.5 min and a maximum merging time of 5 min the reduction in I/O demand is 31.1%, as compared to 47.92% with unlimited maximum merging time. However, the corresponding increases in disk storage for a 120-minute MPEG-I compressed video would be $\approx$ 56 MB for the 5-min maximum merging time, and $\approx$ 1.35 GB for the unlimited merging time. (In calculating increases in storage space, in this section, we assume that only one additional copy of the data would be necessary, i.e., for any portion of an object, only two copies need to be maintained (see Sect. 4).) Of course, as the interarrival times increase, so does the maximum merging time necessary to obtain a "significant" reduction in I/O demand.

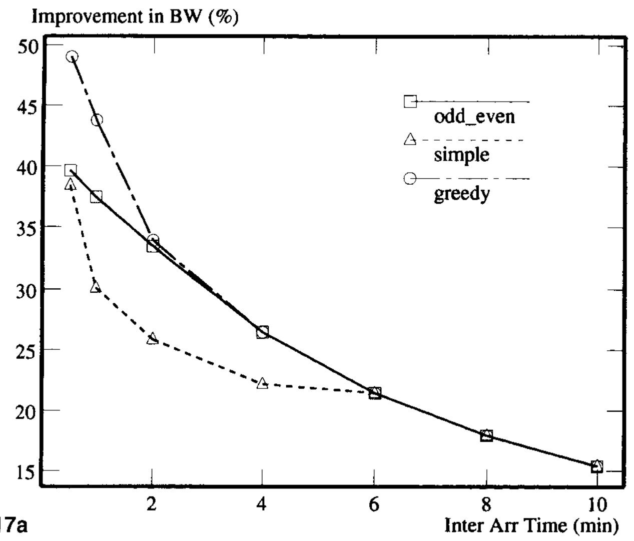
Lastly, one interesting issue to consider is why should we not use the batching approach alone, which is a relatively



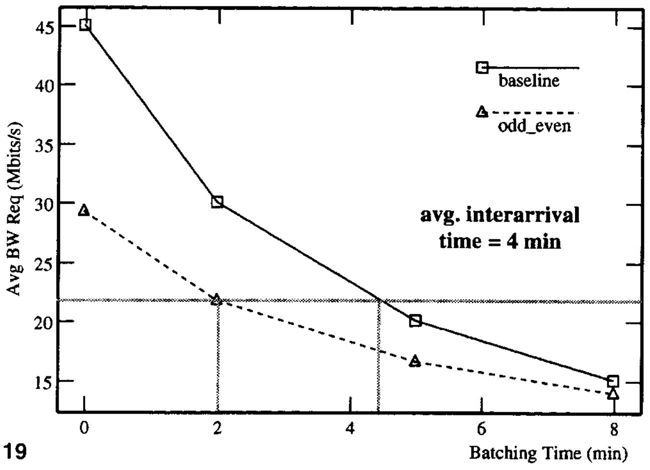
16



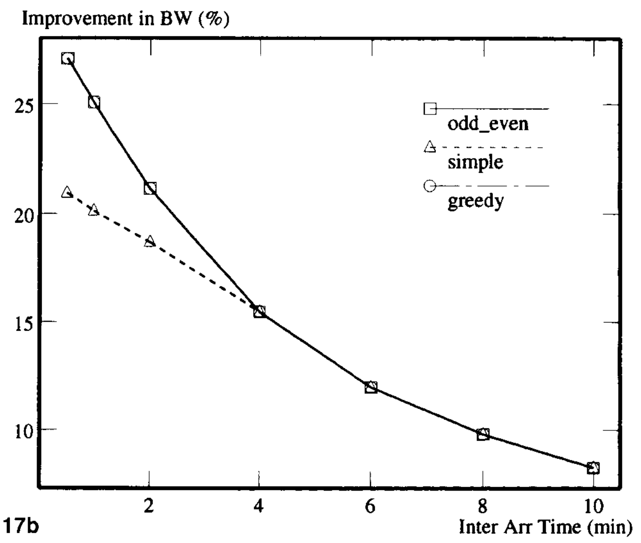
18



17a



19



17b

Fig. 16. Varying arrival rate (without batching)

Fig. 17. Varying arrival rate with a batching interval of **a** 2 min and **b** 5 min

Fig. 18. Benefit vs. maximum merging time

Fig. 19. Batching vs. batching and merging

simple technique to implement. Clearly, we can achieve greater and greater improvements in I/O bandwidth demand by increasing the batching interval. One problem, of course, is that increasing the batching interval also contributes to additional latencies in the system. Thus, a better approach is to combine techniques. In other words, use a small batching interval to obtain some initial benefit from requests for the very popular objects, and then accomplish the rest of the performance improvement through merging without creating additional latencies in the system. This is illustrated in Fig. 19, in which we compare a baseline policy that uses batching alone with the odd-even reduction policy that includes both batching and merging. (In this figure we depict the "average bandwidth requirement" of the system, rather than the "percentage improvement in bandwidth requirement", as in all other graphs in this section.) Here we can see that with 2 min of batching plus merging we can achieve the same performance as with over 4 min of batching alone.

7 Conclusions

On-demand video servers present some interesting performance problems. Part of the effort is simply to understand the constraints and goals well enough to appreciate what is possible. In this paper we have considered a novel method of reducing the I/O bandwidth required, while at the same time providing a guaranteed maximum latency. We have exploited the fact that video-stream rates can be varied by small amounts without perceptible degradation in video quality. We have analyzed several heuristic policies. The results indicate convincingly that small variations in the delivery rate can enable enough merging of I/O streams that significant reduction of I/O bandwidth is realized.

Future work includes a search for better heuristics and an optimal policy. More specifically, we would like to be able to classify an optimal merging policy. Although it might not be practical to implement such a policy, classifying it would give us a better way of designing heuristics. For instance, if an optimal policy were a threshold policy, then this would "justify" the use of threshold-type heuristics. We have presented an approach for combining batching and adaptive piggybacking techniques. Future directions also include considerations of bridging policies in conjunction with batching and adaptive piggybacking, for instance, by using buffer space to accomplish merges earlier.

Acknowledgement. The research done by L.G. was supported in part by the National Science Foundation (NSF) graduate fellowship while the author was a graduate student at the University of California at Los Angeles. The research done by J.C.S.L. was supported in part by the Chinese University of Hong Kong Direct Grant and by the Croucher Foundation. The research done by R.R.M. was supported in part by a Hewlett Packard equipment grant.

References

Berson S, Ghandeharizadeh S, Muntz RR, Ju X (1994) Staggered striping in multimedia information systems. ACM SIGMOD Conference on the management of data, Minneapolis, Minnesota. ACM, New York, pp 79–90

Dan A, Shahabuddin P, Sitaram D, Towsley D (1994a) Channel allocation under batching and VCR control in movie-on-demand servers. Technical Report RC19588, IBM Research Report, Yorktown Heights, NY

Dan A, Sitaram D, Shahabuddin P (1994b) Scheduling policies for an on-demand video server with batching. Proceedings of the 2nd ACM International Conference on Multimedia, San Francisco, CA. ACM, New York, pp 15–23

Dan A, Dias D, Mukherjee R, Sitaram D, Tewari R (1995) Buffering and caching in large-scale video servers. Proceeding of COMPCON, San Francisco, CA, IEEE Conputer Society Press, Los Alamitos, CA, pp 217–224

Dey-Sircar JK, Salehi JD, Kurose JF, Towsley D (1994) Providing VCR capabilities in large-scale video servers. Proceedings of the 2nd ACM International Conference on Multimedia, San Francisco, CA. ACM, New York, pp 25–32

Drapeau AL, Katz R (1993) Striped tape arrays. Proceedings of the 12th IEEE Symposium on Mass Storage Systems, Monterey Calif., IEEE Computer Society Press, Los Alamitos, CA, pp 257–265

Kamath M, Ramamritham K, Towsley D (1994) Buffer management for continuous media sharing in multimedia database systems. Technical Report 94-11; University of Massachusetts, Amherst, Mass

Kamath M, Ramamritham K, Towsley D (1995) Continuous media sharing in multimedia database systems. Proceedings of the 4th International Conference on Database Systems for Advanced Applications (DASFAA'95), Singapore, World Scientific, N.J., pp 112–119

Ohanian TA (1993) Digital nonlinear editing: new approaches to editing film and video. Focal Press, Boston, Mass.

Ozden B, Biliris A, Rastogi R, Silberschatz A (1994) A low-cost storage server for movie-on-demand databases. Proceedings of the 20th International Conference on Very Large Databases, Santiago, Chile. Morgan Kaufmann Pub. Co., San Mateo, CA, pp 594–605

Product description, Zeus (TM) Video Processor, Ampex Systems Corp., Redwood City, CA

Rubin M (1991) Nonlinear: a guide to electronic film and video editing. Triad, Gainesville, FL

Takagi H (1991) Queueing analysis: a foundation of performance evaluation, vol 1: vacation and priority systems, part 1, North-Holland, New York, NY

Tobagi FA, Pang J, Baird R, Gang M (1993) Streaming RAID – a disk array management system for video files, Proceedings of the 1st ACM Multimedia Conference, Anaheim, CA, ACM, New York, pp 393–399

Yu PS, Wolf JL, Shachnai H (1995) Design and analysis of a look-ahead scheduling scheme to support pause-resume for video-on-demand applications. Technical Report RC19683, IBM Research Report, Yorktown Heights, NY