Adaptive Piggybacking: A Novel Technique for Data Sharing in Video-On-Demand Storage Servers*

Leana Golubchik† John C.S. Lui‡ Richard R. Muntz§

Abstract

Recent technological 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. In this paper, we consider improvements in performance of multimedia storage servers through data sharing between requests for popular *objects, where I/O bandwidth is assumed to be 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.*

1 Introduction

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

In this paper, we consider a video-on-demand storage server, e.g., as the one depicted in Figure 1, which archives many objects of long duration, such as movies, music videos, educational training material, etc. The storage server consists of a set of disks ($D_1 \ldots D_N$), a set of processors (N_1

*A version of this paper appeared in SIGMETRICS/Performance '95, under the title "Reducing I/O Demand in Video-On-Demand Storage Servers".

†Department of Computer Science, Columbia University (leana@cs.columbia.edu). This research was supported in part by the NSF graduate fellowship and the IBM graduate fellowship while the author was a graduate student at UCLA.

‡Department of Computer Science, The Chinese University of Hong Kong (cslui@cs.cuhk.hk). This research was supported in part by the CUHK Direct Grant and the Croucher Foundation.

§Computer Science Department, UCLA (muntz@cs.ucla.edu). This research was supported in part by Hewlett Packard through an equipment grant.

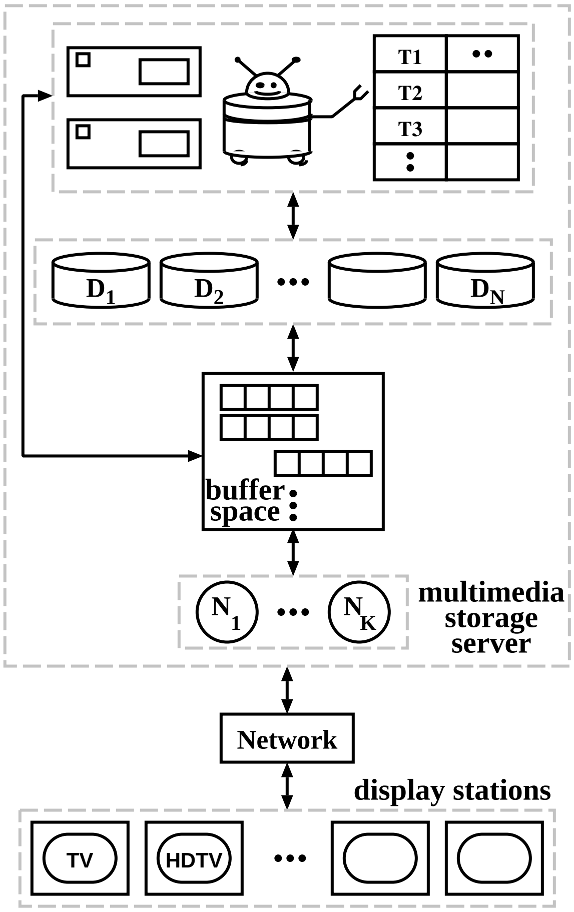


Figure 1: Multimedia Storage Server Architecture.

$\ldots N_K$), buffer space, and a tertiary storage device. The entire database resides on tertiary storage, and the more frequently accessed objects are cached on disks[1]. We assume that a request for an object must be serviced from the disk sub-system; the size of the objects (on the order of 4.5 GB for a 100 minute MPEG-2 encoded movie) precludes them from being stored in main memory, and the long latency and high bandwidth cost of tertiary storage[2] precludes objects from being transmitted directly from tertiary devices. If the requested object is not disk-resident, then it has

[1]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 LRU policy can be used to purge objects from disks (in order to create space for the newly retrieved object).

[2]The seek latency for a 1.3GB tape on a \$1000 tape drive can be on the order of 20 seconds [7], whereas a similarly priced disk, of a similar capacity, has a maximum seek time on the order of 35 milliseconds and more than 16 times the transfer rate. Tape systems with significantly higher transfer rates and tape capacities although not with much lower seek latency do exist, but at a cost \$40,000-\$300,000.

to be retrieved from the tertiary store and placed on disks before its display can be initiated; this could 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, e.g., movies, within a small and "reasonable" latency, upon request. We define the latency for servicing a request as the time between the request's arrival to the time the system initiates the reading of the object (from a disk); the additional delay until data is 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, i.e., 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 Figure 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[3]. These parameters, in conjunction with 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 is the average latency 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 [2, 14, 11]. The distinctions between these architectures can be (mostly) attributed to the data layout and scheduling techniques used. Let us consider one such system, where 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[4]. In such a system, it is fair to assume that there is sufficient disk storage to at least hold the popular

[3]We will not consider the characteristics of the tertiary device in this paper.

[4]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.

objects; moreover, it is very likely that I/O bandwidth is the critical resource which contributes to increases in latency. One way to reduce the latency is to simply purchase more disks. A more interesting and more economical approach might be to either attempt to improve the data layout and scheduling techniques or to reduce the I/O demand of each request in service through "sharing" of data between requests for the same object.

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

1. *batching*: delaying requests for up to T_i time units in hopes of more requests, for the same object i, arriving during the batching interval and servicing the entire group using a single I/O stream (refer to Figure 2(a) where two displays are serviced by a single I/O stream, which 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*: closing the temporal "gaps" between successive requests through the use of buffer space, i.e., holding data read for a "leading" stream and servicing "trailing" requests out of the buffer rather than by issuing another I/O stream (refer to Figure 2(b), where the second display is serviced from the buffer space); of course, a drawback of this approach is the need for additional buffer space

3. *adaptive piggybacking*: adjusting display rates of requests *in progress* (for the same object) until their corresponding I/O streams can be "merged" into one (refer to Figure 2(c), where 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 [4, 5, 11, 17] and bridging [9, 8, 3] 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, which 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

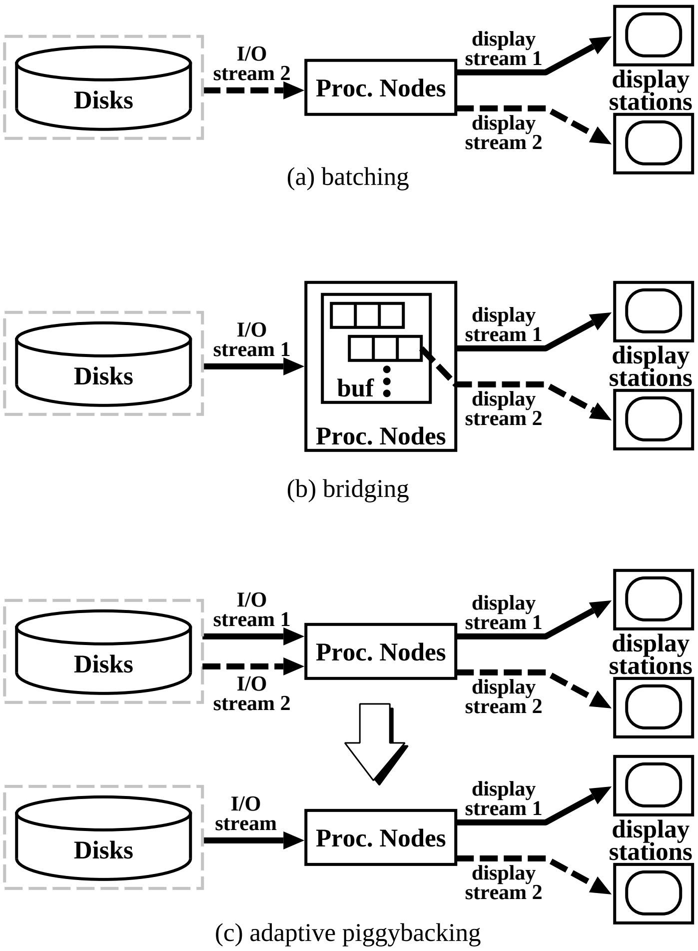


Figure 2: Data Sharing

demand is not quite as high as in the case of batching, since some time must pass before the streams can merge[5]; hence, the tradeoff (between these two techniques) is between latency for starting the service of a request and the amount of I/O bandwidth saved. Note also that, 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 (refer to Figure 3). The conveyor belt represents one particular movie; its length corresponds to the duration of the movie's display, and the rate at which the conveyor moves corresponds to the *normal* display rate of the movie (e.g., 30 frames/sec for U.S.

[5]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.

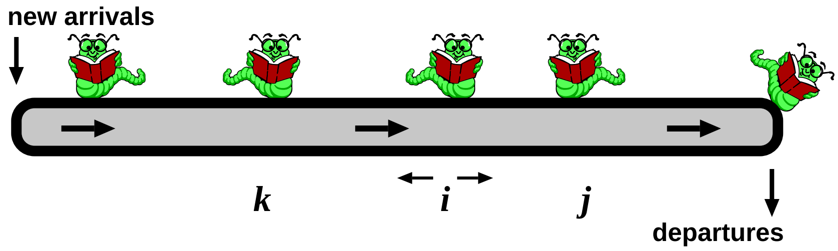


Figure 3: Conveyor Belt Analogy.

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 (within the bounds not perceptible by a human observer) in Section 2; for the remainder of this section we assume that it can be done and concentrate on the (possible) benefits of this approach. These benefits are as follows; if two bugs, one crawling forward and one crawling backward, are able to "merge" at time t, before either one falls off the conveyor belt, then starting at time t the system is able to support both displays using only a single I/O stream[6]. Consider for the moment bug i in Figure 3, which must make a decision, namely, whether to crawl forward, toward bug j, and piggyback on its I/O stream or whether to crawl backward, toward bug k, and instead piggyback on its stream. If i crawls forward, then it will take less time to merge; however, after the merge, a smaller portion of the movie will remain (to be displayed), and hence the benefits of merging would not be as great. On the other hand, if i crawls backward, toward k, then it will take longer to merge; however, greater benefits might be reaped from that 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

[6]Clearly, there is a problem of providing VCR functionality; A similar problem was solved in the context of batching in [4, 6], and their solution of reserving channels for this purpose, can be used here as well; furthermore, adaptive piggybacking has one additional benefit. After obtaining reserved channel and resuming display, further attempts at merging can be made; if successful, the reserved channel can be returned and reused by another stream.

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

2 Altering Video Display Rates

As stated in Section 1, 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 has the capability to dynamically alter the display rate of a request, or, rather, to dynamically *time compress* or *time expand* some portion of an object's display In this section we discuss how this can be done. Note that, although we do not discuss it in detail here, the necessary time adjustments can be performed on the audio portion of an object as well[7], using techniques such as audio pitch correction [15]; for instance, when the movie *Amadeus* was transferred from film to television, it was "time compressed" by 3% [15]. Clearly, the rate of this adjustment must be chosen accordingly to insure the necessary synchronization [12] with the video portion of the object.

We make the basic assumption that the display units being fed by the storage server are NTSC standard and display at a rate of 30 frames per second (fps). Therefore any time expansion or contraction must be done at the storage server. Slow down in the effective display rate can be done by adding additional frames to the video since the display device displays at a fixed rate. For example, if 1 additional frame is added for every 10 of the original frames, the effective display rate (orig-frames/sec) 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

[7] We remind the reader that the word "video" (or "movie") in this context usually includes audio channels as well.

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 which adapts to the 30 fps required for the video from the 24 fps which is standard for films; this is done using a 3-2 pulldown algorithm [12, 10], which for every 4 movie frames creates 5 video frames, where 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) [1] which 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 alterations of the actual display rate in the $2-3\%$ range [16] or expansion and contraction (through interpolation) in the 8% range [15] can be accomplished without being noticeable to the viewer.

There are two approaches to actually 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) to support 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 above discussion, we will, in the remainder of this paper, assume that we can alter the effective display rate by $\pm5\%$ without sacrificing video quality, and we will consider both the on-line generation approach to providing the altered stream of frames and the off-line approach[8] For

[8] 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).

the latter, we will include additional considerations in the scheduling policies that are motivated by the desire to limit the amount of additional disk space required for storing replicates of a video.

3 Batching

As already mentioned in Section 1, 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 tradeoff 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. In this paper, 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 utilization a system has, 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 *batching by timeout* policy, which is described below, is used.)

In this section, we study the following batching policies[9]: (1) batching by size and (2) batching by timeout. 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 pre-defined batching size and 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:

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

[9]Note that the following are simple batching policies which are introduced mainly for the purpose of exploring the possibilities of combining batching with adaptive piggybacking, in later sections. Combinations of these batching policies as well as many other variations on batching are possible, some of which have already been studied in [4, 5, 11]; since batching is not the focus of this paper, we do not pursue these any further.

$$= \frac{B_j - 1}{2\lambda_j} \tag{2}$$

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[10].

3.2 Batching by Timeout

Another policy we consider is that of 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 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 [13] as:

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

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 requests, 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

In this section, we 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

[10] Depending on the network characteristics, it might be wise to batch by size, since it 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.

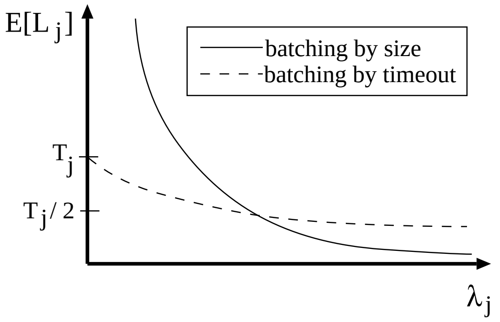


Figure 4: Expected Latency for Batching Policies (for object j).

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 appropriate display stations (e.g., in Figure 5, display streams 1 and 2 are serviced using the corresponding I/O streams 1 and 2). The I/O demand on the storage server can be reduced by

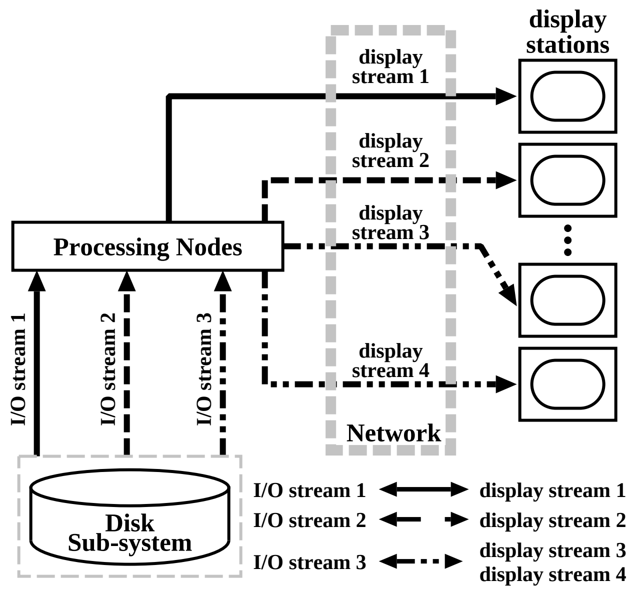


Figure 5: Simplified View of the System.

using a single I/O stream to service several display streams corresponding to requests for the same object, e.g, in Figure 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 Section 1, this can be done in a *static manner*, i.e., by batching requests (see Section 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 be wiser to delay "splitting" display streams 3 and 4 (in Figure 5) until the last possible moment, i.e., transmit them through the network as a single stream for as long as possible in order to obtain 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, and furthermore the corresponding displays are then "in synch". As described in Section 1, this merging can be accomplished by adjusting requests' display rates (see Section 2), in order 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 conjunction with batching policies are reported in Section 6.

Our goal in this paper is to investigate the benefits, namely, the reduction in I/O bandwidth utilization, attributable to the 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 be different from the I/O cost for servicing a normal-rate display[11]. For instance, the speed up (or slow down) can be achieved by replicating data (see Section 2), 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 (see Section 2), then the total number of bytes read from disks will remain the same, regardless of the display rate of a stream.

[11]Note that, there could be other costs, other than I/O bandwidth, associated with reading data at higher or lower rates, e.g., additional buffering space, scheduling complexity, etc.; 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 these requests, such that the total average I/O demand on the system is minimized[12]. 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 object only, i.e., the remainder of the discussion is in terms of a single object.

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

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

[12]Note that 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.

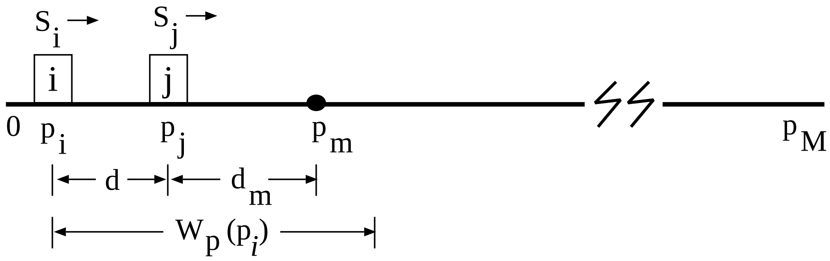


Figure 6: State of the system.

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 it's position in the object's display, p_i, and is moving at a particular display speed, S_i. In order to merge I/O streams i and j, firstly, we have to insure that $S_i > S_j$. Secondly, we can define the following distance and cost constraints which can be used, in any adaptive piggybacking policy, to identify merging opportunities, i.e., whether or not it is possible and cost effective to merge I/O streams i and j. The cost constraint insures 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 as follows[13]:

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

Note that this 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 Equation 6 becomes the only constraint, namely, the object length (or duration of its display) is finite and hence requires the following distance constraint:

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

Finally, the merge time constraint is:

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

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

[13]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.

hence:

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

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

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

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,

$$d^* = \min(d_1, d_2) \tag{10}$$

We can now apply this result to the various adaptive piggybacking policies, which are described next. Our goal is to find adaptive piggybacking policies which have significantly lower expected I/O demand compared to that of the baseline policy[14]. 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 simple heuristics, designed to investigate the possible benefits that can be gained from adaptive piggybacking; these benefits are discussed in Section 6.

We make the following observations about the display adjustment decisions. Consider again the system state depiction in Figure 6; clearly, the only stochastic events in the system are the arrival points; such events as merging of two streams, end of a display, etc., 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 which 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, i.e., to 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 Figure 6. We define a *catch-up window*, $W_p(p_i)$, for policy p, to

[14]A policy that does not use display adjustment, i.e., each I/O streams is displayed at its normal display rate.

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 an object's display; we shall see shortly how the catch-up window is used in the merging policies below. $W_p(p_i)$ can be computed using Equation 10.

The sooner (in the object's display) merging occurs the more resources (e.g., 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 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}.

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

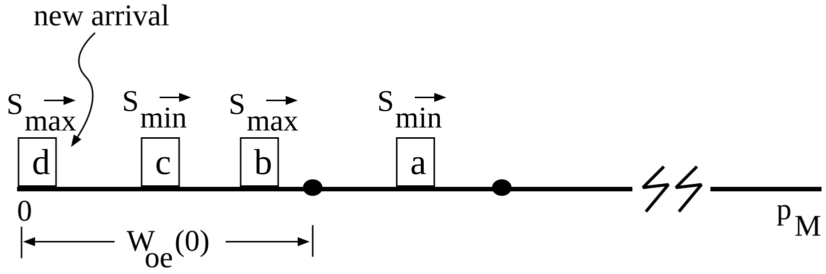


Figure 7: Scenario of Odd-even Reduction Policy.

Algorithm Odd-even reduction

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 would participate in at most a single merge. (Note that, 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 Section 4.5). The "slowing down" of a new arrival (in the above 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 arrived to the system, I/O stream c was 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 arrived to the system, I/O stream a was 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 Equation (10), where the values of d_1 and d_2 can be found (using Equations (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 beginning of an object's display (see Figure 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 (see Figure 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, and moreover they will merge at the right-hand boundary of $W_{sm}^m(0)$ (see Figure 8). The basic rationale behind simple merging policy

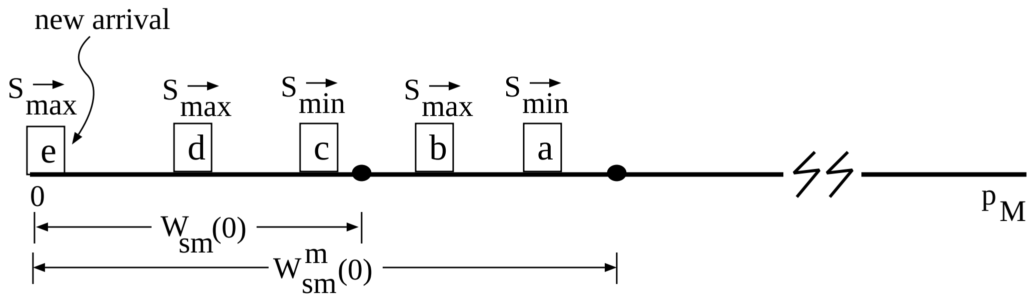


Figure 8: Scenario of Simple Merging Policy.

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, $W^m_{sm}(0)$

$S_i = S_n$;

end

Note that the rationale for keeping the display speed at S_{min} until crossing the right boundary of $W^m_{sm}(0)$ 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 arrived to the system, I/O stream a had already moved outside of the catch-up window $W_{sm}(0)$; therefore, the display speed of I/O stream c was set to S_{min}. When stream b (streams d and e) arrived to the system, stream a (stream c) was within the catch-up window, $W_{sm}(0)$; therefore, their display speeds were 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 $W^m_{sm}(0)$).

$W_{sm}(0)$ and $W^m_{sm}(0)$, can both be computed using Equation (10). The values of d_1 and d_2 can be found (using Equations (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:

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

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

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 *odd-even reduction* and *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. This "current" catch-up window is used by the greedy policy (described

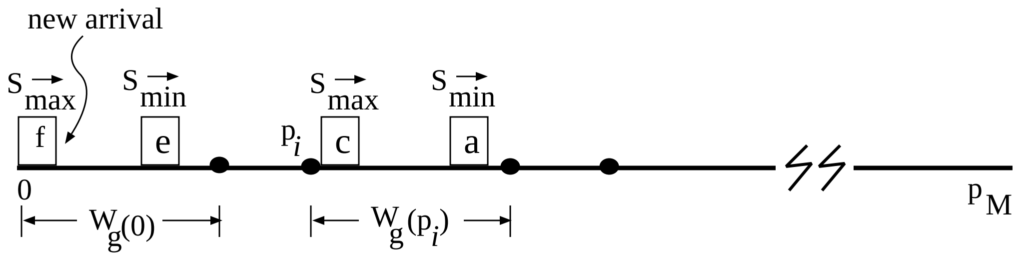


Figure 9: Scenario of Greedy Merging Policy.

below) as an indication of opportunity for further merging.

The greedy policy works as follow. 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 request's speed 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 Algorithm
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 Section 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 occurred in the first catch-up window $W_g(0)$. After 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 Equation (10). The values of d_1 and d_2 (now both functions of the i's current position, i.e., 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:

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

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

4.5 Limited Merging

At this point we remove the assumption that merging can occur at any time. If replication of data is necessary in order to perform the display rate alteration (see Section 2), 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[15]. We can evaluate the 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_m^{max}. Consider again Figure 6 and Equations (5)-(7) which describe the distance and cost constraints that must be met in order 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_m^{max} rather than before p_M, i.e., $p_m \leq p_m^{max}$. Thus, Equations (6) and (9) are replaced by Equations (19) and (20), respectively, as follows:

$$p_i + d + d_m \quad \leq \quad p_m^{max} \tag{19}$$

$$d_2 \quad = \quad \frac{(p_m^{max} - p_i)(S_i - S_j)}{S_i} \tag{20}$$

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 piggy-

[15]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 ($W_{sm}^m(0)$) and only the normal version outside of the maximum merging window.

backing, in conjunction with batching, both with and without a constraint on the maximum merging point, are reported in Section 6; performance analysis of these policies can be found in Section 5.

5 Performance Analysis

In this section we present analytic solutions for computing I/O demand on a storage server which uses adaptive piggybacking policies in conjunction with batching. We define the following notation (also see Figure 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 (see Figure 10)
t_a	=	r.v. representing the time between I/O stream initiation
$W_p(p_i)$	=	catch-up window for policy p, relative to position p_i
$W_p^m(p_i)$	=	maximum merging window for policy p, relative to position p_i
BW_p	=	mean total I/O demand (under policy p) (bits/sec)

Note that, below we do not give equations for either $W_p(p_i)$ or $W_p^m(p_i)$, where p could be the odd-even (oe), simple (sm), or greedy (g) policy. These equation can be found in Section 4. Note also, that the above mentioned equations, for $W_p(p_i)$ and $W_p^m(p_i)$, already allow for the limited merging case, i.e., the case where there is a limit on 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 rate is Poisson with rate λ, $t_e = \frac{1}{\lambda}$. Therefore, the density function of t_a is:

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

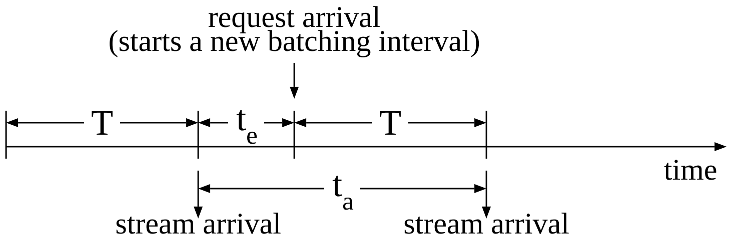


Figure 10: Arrival of I/O streams after a delay.

Since the normal duration of a movie object is p_M/S_n, N, the expected number of I/O streams 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 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 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 under certain ranges of interarrival times and batching delays. Consider two consecutive streams s_1 and s_2 which 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 time of s_2's arrival. 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 $x \leq \frac{W_{oe}(0)}{S_{min}}$) is:

$$BW_{oe}^{m} \quad = \quad (t_m + x)C_{min} + t_m C_{max} + t_f C_n \tag{26}$$

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:

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

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}, i.e., the total mean bandwidth demand in the system:

$$\begin{aligned} BW_{oe} \quad = & \quad \frac{\int_{T}^{\frac{W_{oe}(0)}{S_{min}}} (BW_{oe}^{m} * \frac{N}{2} * f_{t_a}(x))\, dx}{\frac{p_M}{S_n}} \\ + & \quad \frac{\int_{\frac{W_{oe}(0)}{S_{min}}}^{\infty} (BW_{oe}^{nm} * \frac{N}{2} * f_{t_a}(x))\, dx}{\frac{p_M}{S_n}} \end{aligned} \tag{28}$$

5.3 Analysis of 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 Section 4). Similarly to the odd-even policy, we note that all "merging groups" are statistically identical, and 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, $W_{sm}(0)$, which 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:

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

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 2 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 stream (i.e., the first stream in the group), the third stream needs $2t_m$ time units to catch up, etc. The leading stream will keep the display speed at S_{min} until it reaches position $W^m_{sm}(0)$, then the display speed will be reset to S_n. Therefore, the amount of time during which the leading stream has the display speed of S_n is:

$$t_f = \frac{p_M - W^m_{sm}(0)}{S_n} \tag{30}$$

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

$$BW^m_{sm} = \frac{W^m_{sm}(0)}{S_{min}} C_{min} + C(\beta) t_m C_{max} + t_f C_n \tag{31}$$

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

$$BW^{nm}_{sm} = \beta * \left[\frac{W^m_{sm}(0)}{S_{min}} C_{min} + \frac{p_M - W^m_{sm}(0)}{S_n} C_n \right] \tag{32}$$

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

$$\begin{aligned} BW_{sm} &= \frac{\int_T^{\frac{W_{sm}(0)}{S_{min}}} (BW^m_{sm} * \frac{N}{\beta} * f_{t_a}(x))\, dx}{\frac{p_M}{S_n}} \\ &+ \frac{\int_{\frac{W_{sm}(0)}{S_{min}}}^{\infty} (BW^{nm}_{sm} * \frac{N}{\beta} * f_{t_a}(x))\, dx}{\frac{p_M}{S_n}} \end{aligned} \tag{33}$$

5.4 Analysis of Greedy Policy

The performance analysis of the greedy policy is more complex. Let us first refer to Figure 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. Note that for the first level merge (refer to Figure 11, the system reduces the number of streams by half but all remaining streams (e.g., 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:

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

where

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

and

$$\begin{aligned} \tau &= \int_T^{\frac{W_g(0)}{S_{min}}} x f_{t_a}(x)\ dx \\ &= \left[T + \frac{1}{\lambda}\right] - \left[\frac{(S_{min} + \lambda W_g(0)) e^{(-\lambda \frac{W_g(0) - S_{min} T}{S_{min}})}}{\lambda S_{min}}\right] \end{aligned} \tag{36}$$

Given that the streams can go through l levels of merges, the leading stream, after the last merging

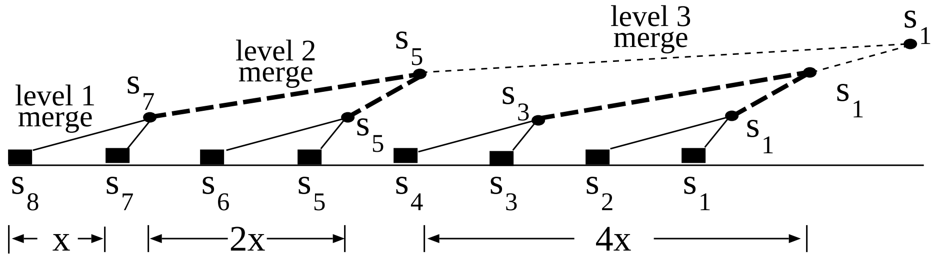


Figure 11: Merging pattern for streams under greedy policy.

point, will have t_f time units of display left, at a speed of S_n, where t_f is:

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

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 a l level merge, is x, and that there are $l \geq 2$ levels of merges, the I/O demand is:

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

where

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

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. (Note that for level two and up, the leading stream will first move at S_n because it will finish merging earlier than the next pairs of trailing streams; when the trailing streams finally finish their merge, its display speed will be reset, from S_n to S_{min}, while the trailing stream (resulting from the merge) will attempt 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 will be:

$$BW_g^{nm} = \left[\frac{\frac{W_g(0)}{S_{min}}C_{min} + \frac{p_M - W_g(0)}{S_n}C_n}{p_M/S_n}\right] N \quad (40)$$

unconditioning on the interarrival time x, we have:

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

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

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

5.5 Validation of Analytic Results

In conclusion of this section, we validate our our analytic results (see Section 5) by comparing them to results obtained from simulation. These comparisons, of all three policies in conjunction with batching by timeout (see Section 3) are depicted in Figures 12-14, where "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. They indicate that the largest divergence from the simulation occurs when the arrival rate is low; the analytic result 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.

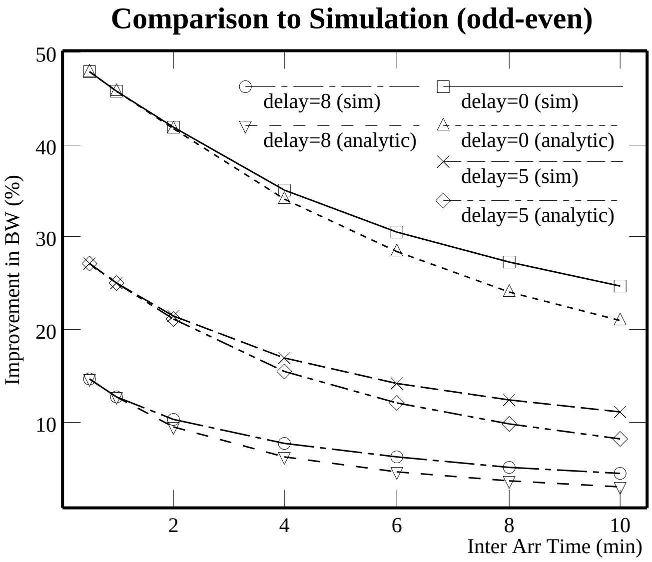


Figure 12: Odd-even policy.

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 (see Section 2). In the remainder of this discussion, we use the the following values for the parameters presented in Section 4:

$$\begin{aligned}
S_{min} &= 28.5 \text{ frames/sec} \\
S_n &= 30.0 \text{ frames/sec} \\
S_{max} &= 31.5 \text{ frames/sec} \\
C_{min} &= 1.425 \text{ Mbits/sec} \\
C_n &= 1.5 \text{ Mbits/sec} \\
C_{max} &= 1.575 \text{ Mbits/sec}
\end{aligned}$$

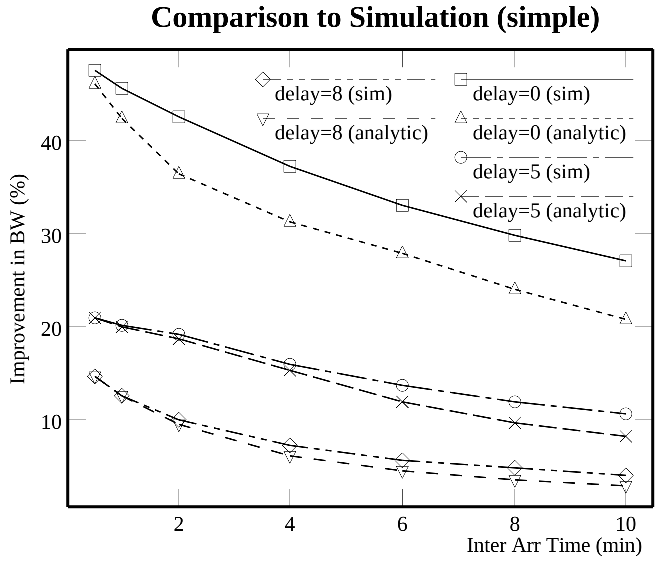


Figure 13: Simple policy.

Batching by timeout (see Section 3) is used as the batching policy in all results presented in this section. The delay time is varied between 0 and 10 minutes, and the mean interarrival time (between consecutive requests for the same movie) is varied between 0.5 and 10 minutes. 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 various policies, as compared to the baseline policy. For ease of exposition, we initially assume no restrictions on the maximum allowed merging time (see Section 4). At the end of this section, we consider the effect of restricting merges to occur within a specified time interval.

We first consider the affects 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 Figure 15, where the interarrival time is kept at 4 minutes and the batching delay is varied between 0 and 10 minutes. 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 (see Equation (4) in Section 3), continues to grow.

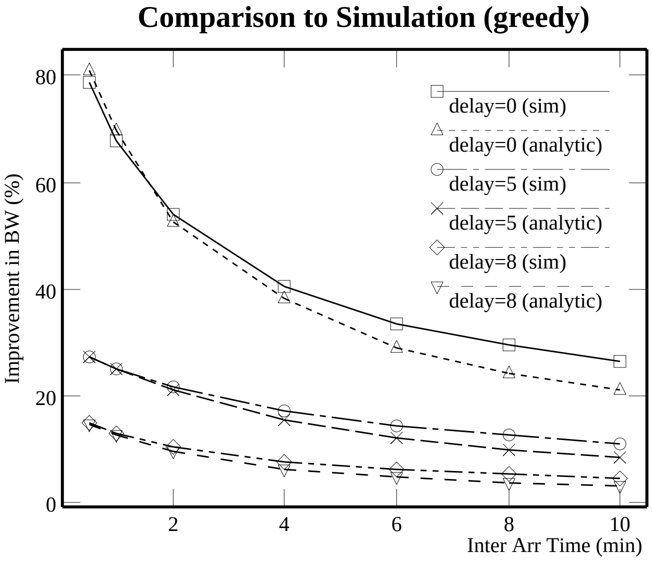


Figure 14: Greedy policy.

Next, we compare the adaptive piggybacking policies, but without batching; the results of this comparison are depicted in Figure 16 (as a percentage improvement over the baseline policy), where the interarrival time is varied between 0.5 and 10 minutes. 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 Figure 16, the reduction in I/O demand, as compared to the baseline policy, ranges from 47.92%, corresponding to a fairly small interarrival time of 0.5 minutes, and 20.92%, corresponding to a fairly large interarrival time of 10 minutes. 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 Figure 16, where we achieve a further reduction in I/O demand; again, as compared to the baseline policy, the results for the greedy policy range from 81.0%, for the fairly small interarrival time of 0.5 minutes, to 20.92%, for the fairly large interarrival time of 10 minutes. The results are qualitatively similar, when batching is used in conjunction with adaptive piggybacking[16]; this is illustrated in Figures 17(a) and 17(b) which correspond to batching intervals of 2 and 5 minutes, respectively.

[16] Clearly, as the batching interval increases, the range of workloads over which these policies exhibit significantly different behavior decreases.

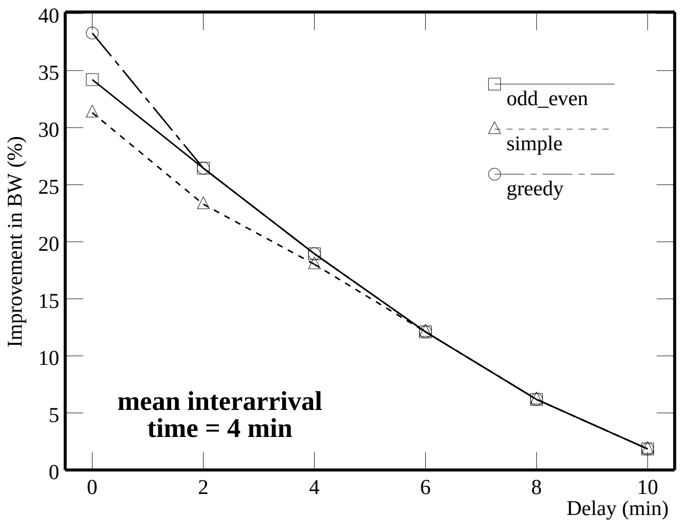


Figure 15: Varying Delay.

Although a greater reduction in I/O demand is achieved by the greedy policy, as compared to 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 (see Section 4). Therefore, in Figure 18 we investigate the benefits of adaptive piggybacking under an additional constraint of a limited merging time (see Section 4 for the new constraint). In this figure, the percent 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). The results are obtained 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, due to lack of space we do not illustrate them here.)

As expected, given fairly small interarrival times, most of the reduction in I/O demand, shown earlier in Figure 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 minutes and a maximum merging time of 5 minutes, 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

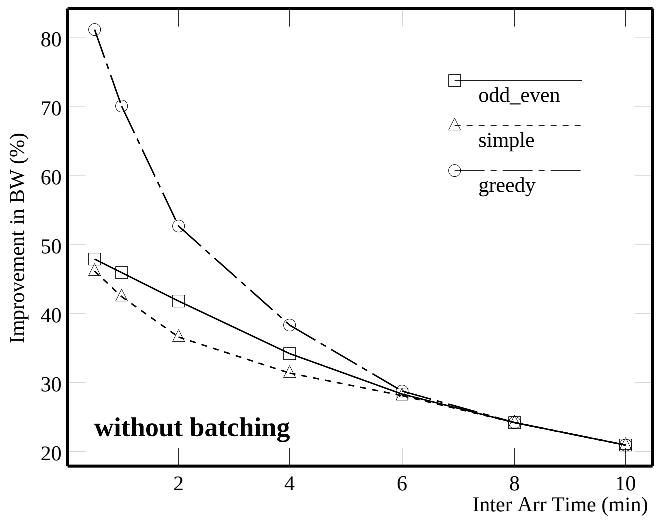


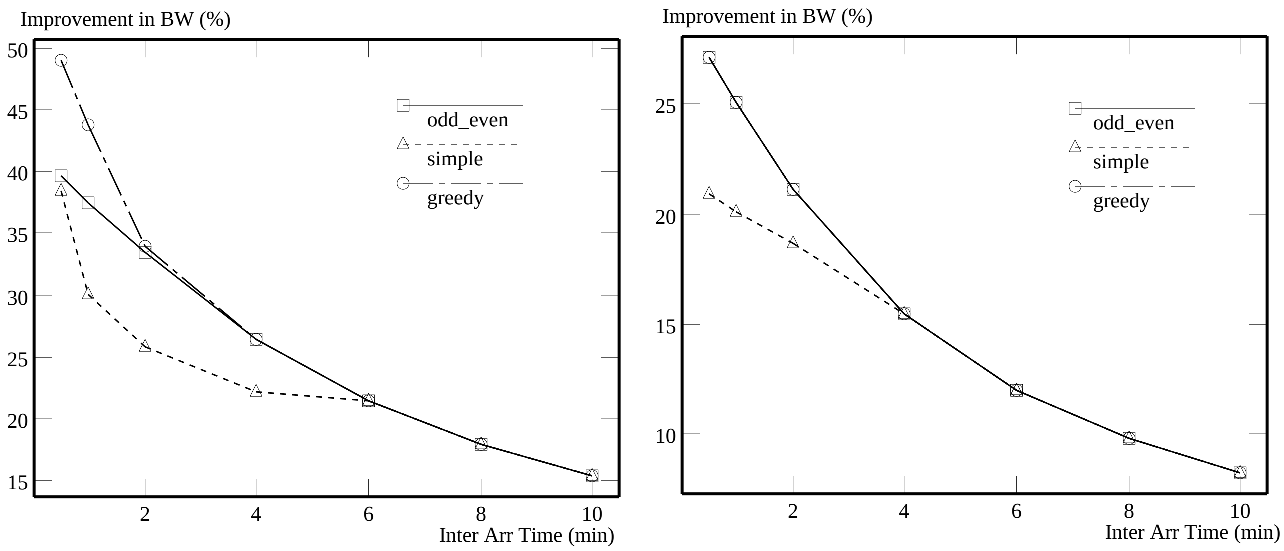
Figure 16: Varying Arrival Rate (no batching).

be ≈ 56 MB for the 5 minute maximum merging time, and ≈ 1.35 GB for the unlimited merging time[17]. 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 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 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 in Figure 19, where we compare a baseline policy, which uses batching alone, with the odd-even reduction policy, which includes both batching and merging[18]. Here we can see that with 2 minutes of batching plus merging we can achieve the same performance as with over 4 minutes of batching alone.

[17] 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 Section 4).

[18] Note that, in this figure we are depicting the "average bandwidth requirement" of the system, rather than the "% improvement in bandwidth requirement", as in all other graphs in this section.



(a) batching interval = 2min

(b) batching interval = 5min

Figure 17: Varying Arrival Rate (with batching).

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 hueristic 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 was a threshold policy, then this would "justify" the use of threshold-type heuristics. In this paper we presented an approach for combining batching and adaptive piggybacking techniques; future directions also include considerations of bridging policies (see Section 1) in conjunction with batching and adaptive piggybacking, for instance, by using buffer space to accomplish merges earlier.

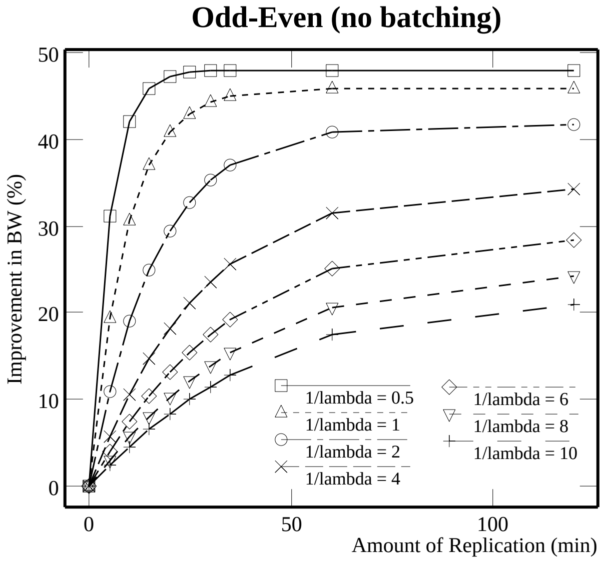


Figure 18: Benefit vs. Maximum Merging Time.

References

[1] Product Description, Ampex DST800 Automated Cartridge Library.

[2] Steven Berson, Shahram Ghandeharizadeh, Richard R. Muntz, and Xiangyu Ju. Staggered Striping in Multimedia Information Systems. *SIGMOD*, 1994.

[3] A. Dan, , D. Dias, R. Mukherjee, D. Sitaram, and R Tewari. Buffering and Caching in Large-Scale Video Servers. In *Proc. of COMPCON*, 1995.

[4] A. Dan, P. Shahabuddin, D. Sitaram, and D. Towsley. Channel Allocation under Batching and VCR Control in Movie-On-Demand Servers. Technical Report RC19588, IBM Research Report, Yorktown Height, NY, 1994.

[5] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling Policies for an On-Demand Video Server with Batching. In *Proceedings of the Second ACM International Conference on Multimedia*, October 1994.

[6] J. K. Dey-Sircar, J. D. Salehi, J. F. Kurose, and D. Towsley. Providing VCR Capabilities in Large-Scale Video Servers. In *Proceedings of the Second to ACM International Conference on Multimedia*, October 1994.

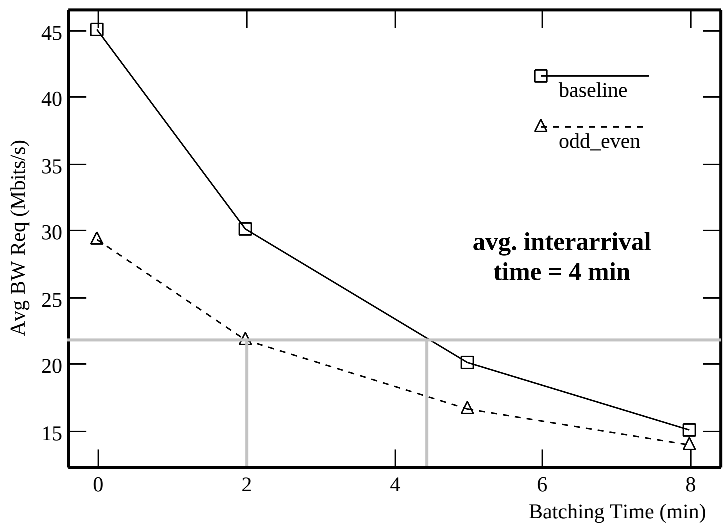


Figure 19: Batching vs. Batching & Merging

[7] A. L. Drapeau and R. Katz. Striped Tape Arrays. In *Proc. of the 12th IEEE Symposium on Mass Storage Systems*, pages 257–265, Monterey, California, April 1993.

[8] M. Kamath, K. Ramamritham, and D. Towsley. Continuous Media Sharing in Multimedia Database Systems. In *Proc. of the 4th Intl. Conference on Database Systems for Advanced Applications (DASFAA '95)*, Singapore, April 10-13, 1995.

[9] M. Kamath, K. Ramamritham, and D. Towsley. Buffer Management for Continuous Media Sharing in Multimedia Database Systems. Technical Report 94-11, University of Massachusetts, February 1994.

[10] T. A. Ohanian. *Digital Nonlinear Editing: new approaches to editing film and video.* Focal Press, 1993.

[11] B. Ozden, A. Biliris, R. Rastogi, and A. Silberschatz. A Low-Cost Storage Server for Movie on Demand Databases. *Proceedings of the 20th International Conference on Very Large Data Bases*, September 1994.

[12] M. Rubin. *Nonlinear: a guide to electronic film and video editing.* Triad Publishing Co., 1991.

[13] H. Takagi. *Queueing Analysis: A Foundation of Performance Evaluation. Volume 1: Vacation and Priority Systems, Part 1.* North-Holland, 1991.

[14] F. A. Tobagi, J. Pang, R. Baird, and M. Gang. Streaming RAID - A Disk Array Management System For Video Files. *ACM Multimedia Conference*, pages 393–399, 1993.

[15] Personal Communication with Mr. Rich Igo and Mr. Bill Carpenter at Ampex Corp.

[16] Personal Communication with Ms. Cary Shott at Lightwors USA and Digital Images.

[17] P. S. Yu, J. L. Wolf, and H. Shachnai. 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 Height, NY, 1995.