On Optimal Piggyback Merging Policies for Video-On-Demand Systems

Charu Aggarwal*, Joel Wolf† and Philip S. Yu†
* Massachusetts Institute of Technology, Cambridge, Massachusetts
† IBM T.J. Watson Research Center, Yorktown Heights, New York
charu at mit.edu, jlw at watson.ibm.com, psyu at watson.ibm.com

Abstract A critical issue in the performance of a video-on-demand system is the I/O bandwidth required in order to satisfy client requests. A number of techniques have been proposed in order to reduce these bandwidth requirements. In this paper we concentrate on one such technique, known as adaptive piggybacking. We develop and analyze piggyback merging policies which are optimal over large classes of reasonable methods.

1 Introduction

In a video-on-demand (VOD) system, subscribers can choose both the movie they wish to view and the time at which they wish to view it. Such systems are becoming feasible because of recent technological advances, and will presumably become popular in the consumer market. The quality of service can be characterized in terms of the *latency time* of a customer request, defined as the length of time between the arrival of the request and the initiation of service. The latency of a request is influenced by a number of factors, which we shall outline in this section.

A VOD system may be modeled as a client-server architecture. The clients essentially consist of the customers, who access the videos stored on disks in the server. (For the purposes of this paper we shall consider only videos stored on disk. Less frequently accessed videos may reside on tertiary storage. The most frequently accessed videos may possibly be stored in memory.) Whenever there is a request for a particular video, it is accessed from the disks in the storage server, transmitted to the central processor, and then routed to the client. Thus an I/O stream needs to be scheduled. Since I/O bandwidth is costly, such streams are key resources in a VOD system, and need to be managed carefully.

One simple way of reducing the bandwidth requirements is known as *batching*. In batching, we intentionally delay the initiation of requests by some amount of time, called a *batching interval*, so that subsequent requests for the same video arriving during the current batching interval may be serviced using a single I/O stream. This trades off reduced I/O stream requirements for increased latency, of course, so large batching intervals would seem to be incompatible with the notion of a VOD system. More work on batching may be found in [1, 2, 4].

A second technique to reduce the I/O bandwidth requirements is called *bridging*. In this technique, we use memory in the central processor as a buffer. If some fixed number of frames behind a particular video stream is buffered, then any subsequent request for that video within the corresponding time interval can be read from buffer rather than from disk. This technique has the disadvantage that a considerable amount of buffer space may be required in order to build bridges large enough to yield substantial savings in bandwidth. More details on the bridging technique may be found in [6, 8].

Recently an elegant technique called *adaptive piggy-*

Permission to make digital/hard copy of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.

SIGMETRICS '96 5/96 PA, USA
© 1996 ACM 0-89791-793-6/96/0005...$3.50

backing was proposed by Golubchik, Lui, and Muntz [5]. This approach assumes the capability of altering the display rates of videos while they are in progress. (It has been established that small differences in the display rates, for example those which deviate at most 5% from the normal display rate, are not perceived by the viewer. So from the customer's perspective this notion appears feasible. We comment a bit about technical feasibility later in the paper.) Suppose two streams are displaying the same video a small number of frames apart. The idea is to display the leading stream at a slower rate, and the trailing stream at a faster rate. Then, assuming this interval is sufficiently small, the faster stream will eventually catch up with the slower stream. At that point the streams can be piggybacked, or *merged.* That is, they can be played thereafter at a single speed, and one stream can be dropped. In a sense adaptive piggybacking is similar in spirit to batching, but it avoids the extra latency which is inherent in the batching interval.

In this paper we will concentrate on adaptive piggybacking. The issue, of course, is to find piggybacking policies for which there are maximum savings in bandwidth. Three basic types of piggybacking policies were discussed in the seminal paper [5]. In approximate order of worst to best performance, these include the *simple merging* policy, the *odd-even* policy, and the *greedy* policy. The first two of these can be characterized as *elementary* in the sense that they involve at most a single change of speed for each video stream. (The greedy algorithm is not elementary.) We should note that by its nature the odd-even policy can have at most 50% improvement in the number of streams saved, because it pairs off subsequent streams.

The contribution of the current paper is twofold. First, we develop a generalization and optimal variant of the simple merging policy which appears to perform better (both empirically as well as analytically) than the version presented in [5]. Second, we propose an entirely new policy, called the *snapshot algorithm*, which will be seen to be optimal over a large class of reasonable piggybacking policies.

Ultimately, our revised simple merging algorithm, which is still elementary, will be seen to have performance nearly equal to that of greedy, the best overall algorithm of [5]. Our snapshot algorithm, which is not elementary, appears to have better performance than any known adaptive piggybacking algorithm.

This paper is organized as follows. In Section 2 we shall develop the generalized simple merging policy. We then focus on the optimal variant, and show its properties. Section 3 describes the snapshot algorithm, which is based on dynamic programming. We show that this policy is optimal over a large class of piggybacking policies. Experimental results are presented in Section 4, and a conclusion is presented in Section 5.

2 Generalized Simple Merging Policy

We shall begin by describing a slightly generalized version of the *simple merging policy* developed in [5]. Consider a single video whose length, in frames, is given by L. Initially we will not consider special features such as pause-resume, fast-forward or rewind. Assume there are two possible display speeds (in frames/second) at which the display may take place – a *slow* speed denoted by S_{min}, and a *fast* speed denoted by S_{max}. (A third, *normal* speed is also considered in [5]. We prefer to assume that the normal and slow speeds are identical, giving customers the most for their money. Less charitable authors might adopt the maxim originally attributed to P.T. Barnum: "This way to the egress." In other words, they would employ the fast speed in normal situations, simultaneously shortening the videos and encouraging customers to watch more of them. However, this and other minor changes in assumptions to the original algorithms presented in [5] are not critical. The reader can easily modify the algorithms and analysis as appropriate.) Define the *maximum catchup window size* W_m, measured in frames, as the latest position in the video at which a slow stream can be overtaken by a fast stream starting at the beginning of the video by the time the video completes at frame L. Given the difference in speed, this can be computed as

$$W_m = \frac{S_{max} - S_{min}}{S_{max}} \cdot L. \tag{1}$$

We define the generalized simple merging policy in terms of a parameter W also measured in frames, called the *window size.* (We require that $0 \leq W \leq W_m$.)

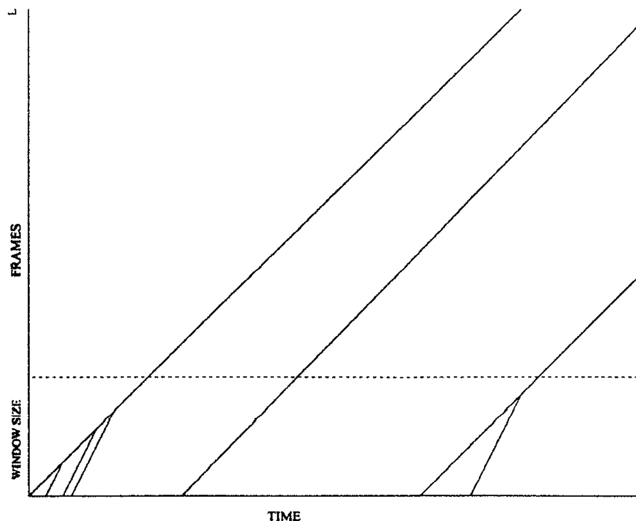

Figure 1: Generalized Simple Merging Policy

Specifically, a new arrival is designated to be a fast stream if a slow stream exists within W frames of it. Otherwise, the stream is designated to be a slow stream. If a fast stream merges with a slow stream, the fast stream is dropped, and the slow stream proceeds. Figure 1 illustrates the algorithm, the x-axis representing (increasing) time and the y-axis representing the position of the video in frames. (The window size and length L of the video are also shown.) Note that there is always a single slow stream associated with each distinct window. On the other hand there can be any number of fast streams, including 0.

Pseudocode for the generalized simple merging policy is as follows:

Algorithm Generalized Simple Merging Policy (W)
Case: Arrival of stream i
If no stream within W is moving at S_{min}
  Set Speed = S_{min}
Else
  Set Speed = S_{max}
Case: Merge of streams i and j
Drop stream i
Set Speed = S_{min}

Now in [5] the window size W used is exactly equal to the maximum catchup window size. That is, they set $W = W_m$. We plan, instead, to optimize W as a function of the forecasted arrival rate. Assume, for simplicity, that requests for the video arrive according to a simple Poisson process with rate λ. (This assumption will not, of course, be perfectly accurate.)

The tradeoffs for different size values of W are as follows:

(1) When the window size is big, a larger number of fast streams can be merged into one slow stream. But they tend to be merged at later stages, with less benefit.

(2) When the window size is small, merges tend to occur at earlier stages. But there are fewer of them.

In order to quantify the savings due to piggybacking, recall that whenever a slow stream is merged with a fast stream, both streams combine into one slow stream. In effect, we assume that the fast stream exists only until that time. Thus we will charge a fast stream only the number frames needed to reach the merge point.

We first proceed to build a model which expresses the expected number of frames for a randomly chosen display stream as a function of the window size W. Consider a new video stream arrival, which may be either fast with probability P_{fast} or slow with probability $P_{slow} = 1 - P_{fast}$. The expected number $E[F]$ of frames read by a randomly chosen display stream is the weighted average of the expected number $E[F_{fast}]$ of frames if the stream is fast and the expected number $E[F_{slow}]$ of frames if the stream is slow. In other words,

$$E[F] = P_{fast} \cdot E[F_{fast}] + P_{slow} \cdot E[F_{slow}]. \quad (2)$$

By our frame charging assumption we have that F_{slow} is deterministically equal to L, and hence $E[F_{slow}] = L$ as well. It is only slightly more complicated to calculate the number of frames charged when the stream is fast. Suppose the nearest slow stream beyond it is p frames ahead. The number of frames required by this fast stream to catch up with the slow stream is given by

$$F_{fast} = \frac{p \cdot S_{max}}{S_{max} - S_{min}}. \quad (3)$$

Note that the algorithm is designed in such a way to ensure that $p \leq W$. Since the arrival rate is uniform

it follows by symmetry that p is uniformly distributed between zero and W. Thus

$$E[F_{fast}] = \frac{W/2 \cdot S_{max}}{S_{max} - S_{min}}. \quad (4)$$

It now remains to calculate the probability that a randomly chosen stream will be fast. Note that all streams which are within W frames of a slow stream (or, equivalently, arrive within W/S_{min} time units of a slow stream) are fast. Hence for each slow stream, the expected number of fast streams following it consecutively equal to $\lambda \cdot W/S_{min}$. Consequently, the fraction of fast streams in the system is approximately equal to

$$P_{fast} = \frac{\lambda W/S_{min}}{\lambda W/S_{min} + 1}. \quad (5)$$

Substituting the above values in Equation 2, we obtain the following relationship:

$$E[F] = \frac{\lambda W}{\lambda W + S_{min}} \cdot \frac{W S_{max}}{2(S_{max} - S_{min})} + \frac{S_{min}}{\lambda W + S_{min}} \cdot L. \quad (6)$$

We now minimize this equation subject to the constraint that a new fast stream must always be able to catch up with a slow stream if the slow stream is at most W frames ahead the fast one. This constraint amounts to:

$$W \cdot \frac{S_{max}}{S_{max} - S_{min}} \leq L. \quad (7)$$

Ignoring the constraint for the time being, we set

$$\frac{dE[F]}{dW} = 0. \quad (8)$$

On expanding the resulting equation for W and simplifying, we obtain:

$$W^2 + \frac{2S_{min}}{\lambda} \cdot W - \frac{L S_{min}(S_{max} - S_{min})}{\lambda \cdot S_{max}} = 0. \quad (9)$$

Solving the above quadratic for W (and ignoring the negative root), we obtain:

$$W^* = -\frac{S_{min}}{\lambda} + \sqrt{\left(\frac{S_{min}}{\lambda}\right)^2 + 2 \cdot \frac{L S_{min}(S_{max} - S_{min})}{\lambda S_{max}}}. \quad (10)$$

The second derivative is positive, and an easy check shows that this value of W^* automatically satisfies constraint 7. Consequently, W^* is the optimal window size.

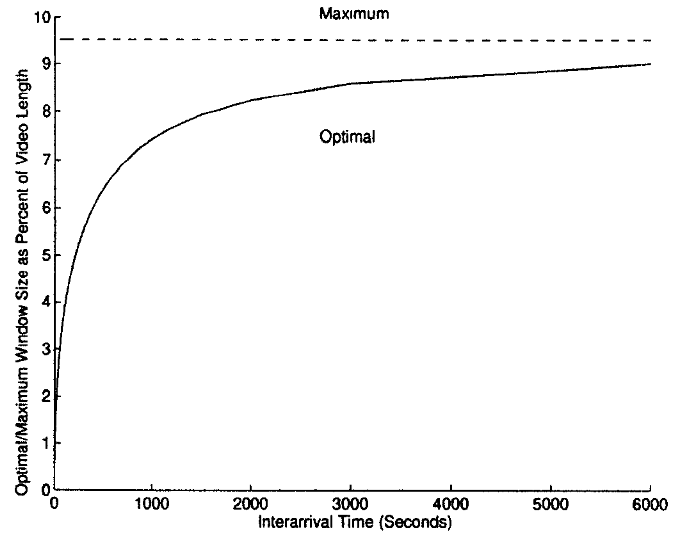

Figure 2: Optimal and Maximum Catchup Window Sizes

Figure 2 shows the relative values of W_m and W^* as a function of interarrival time (the reciprocal of λ). (These are normalized as a percentage of the total number of frames in the video.) These numbers were evaluated for a two hour video with $S_{min} = 28.5$ and $S_{max} = 31.5$. Notice that the optimal window size is always smaller than the maximum catchup window size, sometimes considerably so. However, W^* asymptotically approaches W_m as the interarrival time increases.

We briefly discuss the modifications required to the generalized simple merging policy to handle special customer features such as pause, resume, fast-forward and rewind. One must be able to accomodate the addition of new streams at arbitrary positions within the video, and similarly the elimination of existing streams from arbitrary positions. To handle the addition of a new stream, notice that currently playing streams can be partitioned at any point in time into groups of two different types. The first group will consist of a slow stream followed by one or more consecutive fast streams which will ultimately merge with the slow one. The second group consists of solitary slow streams. If a new video arrives at a position between the first and last members of a group of the first kind, it should be assigned a fast speed and eventually merged with the slow stream. A new video arriving outside the range of a group of the first kind should be assigned a fast speed if it is behind

a slow stream at position less than W, and a slow speed otherwise. To handle the elimination of an existing slow stream, do nothing if the stream trailing it is also slow (or nonexistent). If the stream trailing it is fast, change its speed to slow. Nothing need be done to handle the elimination of an existing fast stream. Remember, of course, that many customers may be piggybacked onto a single stream, so the removal of one such customer does not necessarily imply that the stream itself will be dropped. The policy just described remains elementary in the sense that streams will change speeds at most once. We have developed an approximation algorithm to compute the optimal window size for such a scenario, but details are complicated and we omit them in the current paper.

3 The Snapshot Algorithm

Consider again a single video consisting of L frames. Suppose that at a fixed point T in time there are a total of n streams of this video playing. Denote the positions of these streams, measured in terms of frames, by $f_1, ..., f_n$, respectively. Without loss of generality we can assume that $f_1 \geq ... \geq f_n$. Ignore for the time being any other requests for this video which may appear later, and the manner in which the streams reached their current positions. Also assume that there are no pauses, resumes, fast-forwards or rewinds. This scenario is entirely deterministic, and it is therefore meaningful to attempt to find the precise piggybacking strategy which minimizes the total number of frames required from time T onward. We begin the section by solving this optimization problem via a dynamic programming algorithm.

As before, the two speeds are denoted by S_{max} and S_{min}. We can assume in an optimal solution that the stream farthest along (in this case the one initially corresponding to f_1) proceeds at speed S_{min}, while the stream least farthest along (corresponding initially to f_n) proceeds at speed S_{max}: It is never *more* profitable not to do so. For the same reason, of course, we always merge two streams which coalesce. While we will certainly have to account for the costs correctly, pretend for the moment that merges can occur at any point, including possibly past the length L of the video. We can then envision each potentially optimal piggybacking policy as a binary tree. The leaf nodes correspond to the original streams, while interior nodes correspond to merges. The root node corresponds to the final merge of all the n original streams. Left arcs correspond to the fast speed, and right arcs correspond to the slow speed. *Past* the root node there exists only one stream, which can proceed at either speed. We don't explicitly consider this as part of the binary tree, but assume the speed is S_{min} as before. Some of the merges close to the root node may never actually take place. This depends on whether or not they would occur past position L.

Looked at in this light there is a one-to-one correspondence between the set of binary trees with n leaf nodes and all potentially optimal piggybacking policies for n streams.

Figure 3 shows the 5 possible binary trees for a scenario in which there are $n = 4$ original streams. Here we have reversed the roles of the x- and y-axes from that of Figure 1, in order to draw the binary trees in something like standard orientation. Thus the x-axis corresponds to position and the y-axis to time. (We actually show a little more structure, namely the relative positions of the initial streams, the two speeds, and so on. The area of the histogram underneath each binary tree illustrates the cost, in frames, of implementing that particular piggybacking strategy, assuming the final merge at the root occurs *before* L. Note that the root always occurs at the same position, and the cost remaining is the difference between that position and L. This is a constant, and like the remaining single stream is not illustrated. Being a constant, this term is also irrelevant to the optimization problem. If L occurs before the final merge, the actual cost would correspond to integrating the curve *up to* L.)

We note in passing that the *odd-even* policy described in [5] pairs up successive streams which start no more than a fixed window size apart, by playing one at S_{min} and the other at S_{max} until they merge. The *greedy* policy developed in [5] iterates this process recursively with successive merged pairs. Thus, roughly speaking, the greedy policy results in binary trees of the form shown in (d) of the figure.

We recall that the number of binary trees with n leaf nodes (streams) is given by the $(n-1)$st *Catalan*

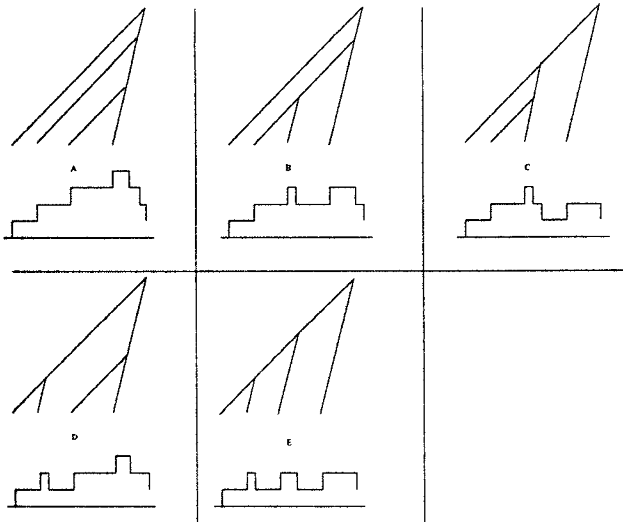

Figure 3: Typical Binary Tree Alternatives and Costs

number

$$b(n-1) = \frac{1}{n}\binom{2n-2}{n-1}. \tag{11}$$

See [3] for details. The Catalan number $b(n)$ can be approximated via Stirling's approximation as

$$b(n) \approx \frac{4^n}{\sqrt{\pi}n^{3/2}}. \tag{12}$$

So the Catalan numbers grow very rapidly, and searching all binary trees for any reasonable value of n will be impractical. Fortunately, there is a better way:

Let i and j denote two streams between 1 and n, with $i \leq j$. Let $P(i,j)$ denote the hypothetical position in frames at which streams i and j would merge in an optimal policy for the case in which only the arrivals $i, \ldots, j$ occur. This value may possibly be greater than L, and is also the position at which streams i and j would merge if they were the *only* streams. The point is that $P(i,j)$ is well-defined because this optimal policy would involve stream i moving at the maximum speed and stream j moving at the minimum speed. So we obtain

$$P(i,j) = f_j + \frac{S_{min} \cdot (f_j - f_i)}{S_{max} - S_{min}} \tag{13}$$

via our standard analysis if $i < j$, and

$$P(i,i) = f_i. \tag{14}$$

This value can thus be computed for each relevant pair i and j, and is independent of all other streams. Now let $C(i,j)$ denote the cost of an optimal policy in which only the arrivals $i, \ldots, j$ occur. Denote the corresponding binary tree by $\mathcal{T}(i,j)$. It is easy to see that

$$C(i,i) = L - f_i \tag{15}$$

for each i. In order to compute $C(i,j)$ for $i < j$ we observe that the principal of optimality holds here: For the optimal policy there will exist a stream k with $i \leq k < j$ such that the left subtree will contain the leaf nodes corresponding to streams $i, \ldots, k$ and the right subtree will contain the leaf nodes corresponding to streams $k+1, \ldots, j$. Furthermore, both the left and right subtrees themselves will be optimal. That is, they will be $\mathcal{T}(i,k)$ and $\mathcal{T}(k+1,j)$, respectively. Such a binary tree has cost $C(i,k) + C(k+1,j) - max(L - P(i,j), 0)$, the last term indicating the (potential) savings of the final merge at position $P(i,j)$. Thus the optimal policy in which only arrivals $i, \ldots, j$ occur has a left subtree with leaf nodes corresponding to $i, \ldots, k^*$ and a right subtree with leaf nodes corresponding to $k^*+1, \ldots, j$, where

$$k^* = \operatorname{argmin}_{i \leq k < j}\{C(i,k) + C(k+1,j) - (L - P(i,j))^+\}. \tag{16}$$

The overall optimal cost $C(1,n)$ and its corresponding piggybacking policy can therefore be calculated in a bottom up fashion by dynamic programming: Starting with the initial trees $\mathcal{T}(i,i)$ and costs $C(i,i)$, compute all trees $\mathcal{T}(i,i+1)$ and costs $C(i,i+1)$, then all trees $\mathcal{T}(i,i+2)$ and costs $C(i,i+2)$, and so on. Ultimately, we compute the optimal tree $\mathcal{T}(1,n)$ and its optimal cost $C(1,n)$.

Pseudocode for the dynamic programming algorithm is as follows:

Algorithm Dynamic Programming
For $i = 1$ to n do
  Initialize $P(i,i), C(i,i)$ and $\mathcal{T}(i,i)$ via Equations 14 and 15
End
For $m = 1$ to $n-1$ do
  For $i = 1$ to $n-m$ do
    Compute $P(i,i+m), C(i,i+m)$ and $\mathcal{T}(i,i+m)$ via Equations 13 and 16
  End
End

It remains to compute the computational complexity:

Theorem 3.1 *The dynamic programming algorithm finds the optimal merging policy and requires $O(n^3)$ iterations, where n is the number of streams to be merged.*

Proof After initialization there are $O(n)$ iterations of the outer loop and $O(n)$ iterations of the inner loop. The calculation of the positions, costs and trees requires $O(n)$ comparisons.

We comment that the dynamic program algorithm presented has analogues in the problem of optimal polygon triangulation via the natural correspondence between binary trees and convex polygons [3, 7]. This, in turn, has led to algorithms for parse trees and the like.

We now incorporate the deterministic dynamic programming technique discussed above into a practical window-based piggybacking policy known as the ***snapshot*** algorithm. Assume at first that there are no pauses, resumes, fast-forwards or rewinds. The algorithm is based on the idea of taking snapshots of the positions of the streams at fixed time intervals, say of I units each. We will call these the *snapshot* intervals. The first stream arriving within a snapshot interval is assigned a speed of S_{min}. All other arriving streams within this same snapshot interval are assigned a speed of S_{max}. Suppose there are n such streams, with stream 1 being slow and streams $2, ..., n$ being fast. This mimics the original and generalized simple merging policy described in [5] and the previous section. Notice that all n streams will lie within a window, measured in frames, of length $W = I \cdot S_{max}$. We shall refer to W as the *snapshot window size.* We will choose I in a way that ensures that the snapshot window size is less than or equal to the maximum catchup window size W_m. By the end of the snapshot interval some of our initial n streams may have merged. We shall use our dynamic programming algorithm in order to modify the speeds of all the remaining streams that were initiated in the interval. We do not affect the speeds of streams from previous snapshot intervals.

Actually, many variants of this snapshot algorithm are possible. One could, for example, solve the overall dynamic programming problem for all currently playing streams, not just the ones within the most recent snapshot interval. This approach would appear to be too costly, given the complexity of the dynamic programming algorithm. Of course, the effectiveness of the algorithm itself will cause the number of surviving streams to be significantly reduced relative to the number of original customer requests. Conversely, the ratio of surviving streams relative to original requests should decrease throughout the lifetime of the video. Thus it is more important to perform the dynamic programming algorithm for earlier rather than later streams anyway. A second apparently reasonable algorithmic variant might group streams together according to their arrival snapshot interval, and resolve the dynamic programming problem for each such group at the end of every snapshot interval during its lifetime. These groupings would appear plausible in the sense that the last stream from one snapshot interval and the first stream from the next snapshot interval are naturally moving away from each other anyway. But a little thought will show that all subsequent solutions to the dynamic programming problem will be identical to the original one. Thus the snapshot algorithm we have presented appears to represent the best tradeoff among various reasonable alternatives.

Pseudocode for the snapshot algorithm is as follows:

Algorithm Snapshot Policy (W)
Compute snapshot interval I
Start interval counter
Case: Arrival of stream i
If first stream is within interval
  Set Speed = S_{min}
Else
  Set Speed = S_{max}
Case: End of interval counter
Solve dynamic programming problem on remaining new streams
Reset interval counter
Case: Merge of streams i and j
If within initial interval follow generalized simple merging rules
Else follow dynamic programming rules

The dynamic programming policy is optimal for streams which survive past the snapshot interval. Because it employs a generalized simple merging policy to

deal with non-deterministic arrivals until the end of that interval it may not be quite optimal during that region of time. But if the snapshot interval is small relative to the total time of the video (or, equivalently, if W is small relative to L), then we may ignore this effect. In other words, no window-based policy will outperform the snapshot algorithm under these conditions. As we shall see, the snapshot algorithm gives the best experimental results of any of the tested piggyback policies as well.

We should point out that we do not have an analytic method for deriving the optimal snapshot window size. So we have adopted the optimal window size of the generalized simple merging policy instead. Fortunately, we will show experimentally that the performance of the snapshot algorithm is fairly insensitive to the choice of window size.

Special customer features such as pause, resume, fast-forward and rewind can be accomodated in the context of a snapshot algorithm. However, they involve complicated heuristics, and we omit details.

4 Experimental Results

In this section we discuss the empirical results of simulations in which we compare the various piggybacking policies. No batching or bridging was considered in any of these experiments, in order to isolate the benefits of piggybacking policies from any other effects. In all our experiments we assumed a fast speed of $S_{max} = 31.5$ frames per second and a slow speed of $S_{min} = 28.5$ frames per second. All simulations were performed assuming Poisson arrivals with a parameter of λ. Each graph discussed in this section shows the percentage of frames saved as a result of piggybacking.

Figure 4 shows the performance of the original simple merging and greedy policies of [5] and our new optimal simple merging policy. Here we fix the length of the video to be 2 hours, and vary the interarrival time $\gamma = 1/\lambda$ between 15 and 500 seconds. As we see, employing the optimal rather than maximum catchup window size can have a large effect on the performance of the generalized simple merging policies, at least when the interarrival time is small (or equivalently, when the arrival rate is large). Furthermore, the optimal simple

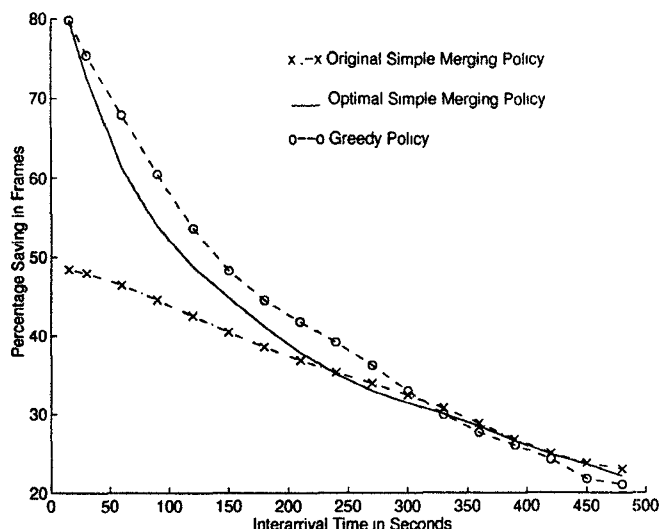

Figure 4: Comparisons for Varying Interarrival Rates

merging policy now appears to be nearly competitive with the more elaborate greedy policy. Recall that the former algorithm is *elementary*, while the latter is not. For high interarrival times all three algorithms have essentially equal performance. As one would expect, the performance of each algorithm decreases as a function of increasing interarrival time.

Figure 5 shows the performance of the generalized simple merging policy as a function of window size W. Here we fix the length of the video to be 2 hours, and the interarrival time to be 30 seconds. The optimal window size W^* is also shown, and matches the best performance of the generalized simple merging policy. Differing from the optimal window size by too much in either direction causes the performance to degrade. Note that our optimal window size is computed in terms of the arrival rate λ, which is by necessity a forecast. Fortunately the figure shows relatively stable performance for values of W close to optimal, so errors in forecasts will probably not have major negative effects on the optimal policy.

Figure 6 shows the performance of the original and optimal simple merging policies and the snapshot algorithm as a function of video length. Here we fix the interarrival time to be 30 seconds. Notice that optimal simple merging policy always outperforms the original policy, and the improvement grows as a function of the video length. In fact, the performance of the original

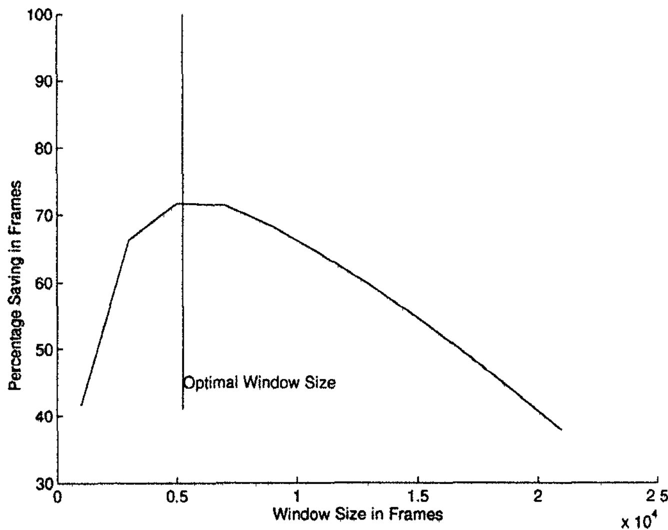

Figure 5: Generalized Simple Merging Performance for Varying Window Sizes

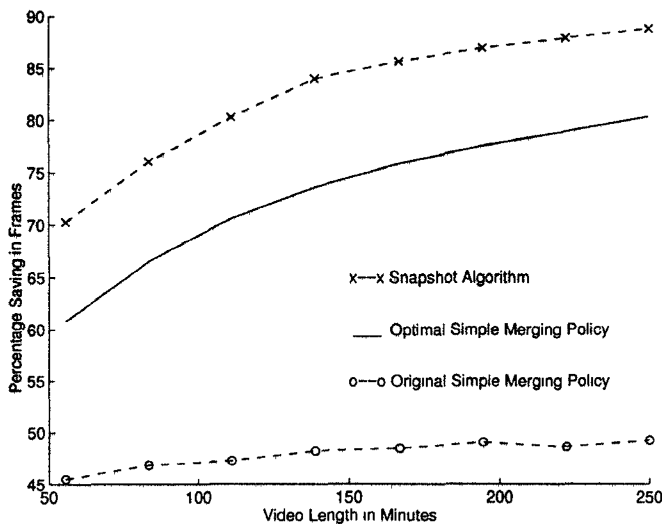

Figure 6: Comparisons at Varying Video Lengths

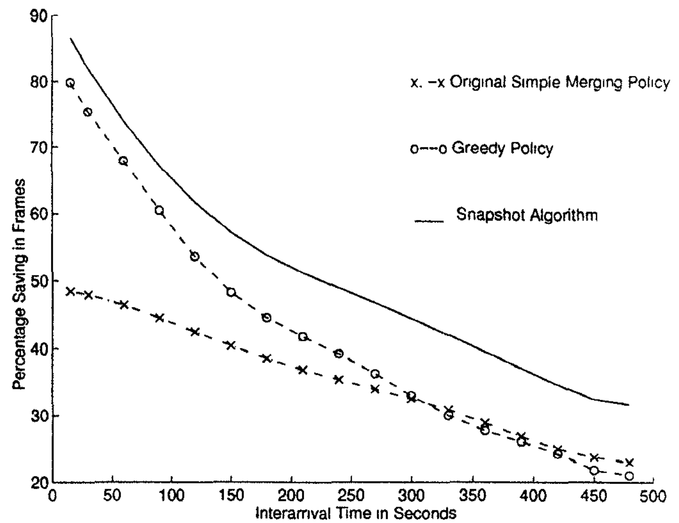

Figure 7: Comparisons for Varying Interarrival Times

simple merging policy is relatively flat. The snapshot algorithm outperforms both of the other policies, and the improvement relative to the optimal simple merging policy remains more or less constant.

Figure 7 shows the performance of the original simple merging and greedy policies of [5] and the snapshot algorithm. Again we fix the length of the video to be 2 hours, and vary the interarrival time γ between 15 and 500 seconds. Notice that snapshot does clearly better than the greedy algorithm, and this difference is even more pronounced for higher interarrival times.

Figure 8 shows the performance of the snapshot algorithm as a function of window size W, which we normalize as a fraction of the maximum catchup window size. Again we fix the length of the video to be 2 hours, and the interarrival time to be 30 seconds. Recall that this is important because we do not have a method to determine the optimal value of W for snapshot. Fortunately, the curve is quite flat in the range we are concerned about.

5 Conclusions

Based on the original idea of Golubchik, Lui and Muntz, we have in this paper devised and analyzed two piggybacking algorithms for VOD systems. The first is a generalized version of their simple merging policy, for which

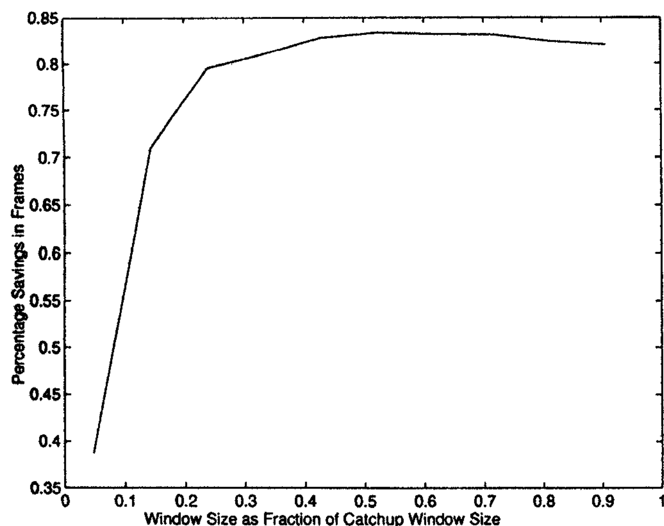

Figure 8: Snapshot Performance for Various Window Sizes

we find the optimal window size. The second is a snapshot algorithm based on dynamic programming. The snapshot algorithm apparently outperforms all existing policies, while the performance of the optimal simple merging policy is nearly equal to that of the best previously known policy.

Some concern has been raised regarding the ease of implementation of on-the-fly piggybacking schemes, given current MPEG standards. That is, it may be difficult to create a fast version of a video in real time from a slow version. (There certainly exists equipment today which can vary speeds of videos off-line.) We believe that the notion of piggybacking is very appealing, however, especially as an alternative and/or companion to batching. Piggybacking results in improvements similar to those of batching, and yet does not require additional latency for customers. Thus piggybacking would seem to be more in the spirit of VOD. In any case, there is an alternative to on-the-fly piggybacking. As noted in [5], one could pre-store on disk fast speed initial segments of the most popular videos. Since most of the good effects of piggybacking will occur early in video showing, it should be possible to derive most of the benefits, at the cost of a small amount of additional disk space.

References

[1] C. Aggarwal, J. Wolf and P. Yu, "On Optimal Batching Policies for Video-on-Demand Servers", *IEEE Multimedia Computing and Systems Conference*, Hiroshima, Japan, 1996.

[2] D. Anderson, "Metascheduling for Continious Media", *ACM Transactions on Computer Systems*, Vol. 11, No. 3, 1993, pp. 226-252.

[3] T. Cormen, C. Leiserson and R. Rivest, *Introduction to Algorithms*, McGraw Hill, 1986.

[4] A. Dan, D. Sitaram and P. Shahabuddin, "Scheduling Policies for an On-Demand Video Server with Batching", *ACM Multimedia Conference*, San Francisco, CA, 1994, pp. 15-24.

[5] L. Golubchik, J. Lui and R. Muntz, "Reducing I/O Demand in Video-on-Demand Storage Servers", *ACM Sigmetrics Conference*, Ottawa, Canada, 1995, pp. 25-36.

[6] M. Kamath, D. Towsley D., and Ramamritham, "Buffer Management for Continuous Media Sharing in Multi-Media Database Systems *Technical Report 94-11*, University of Massachusetts, 1994.

[7] D. Sleator, R. Tarjan and W. Thurston, "Rotation Distance, Triangulations and Hyperbolic Geometry", *JACM*, 1986, pp. 122-135.

[8] P. Yu, J. Wolf and H. Shachnai, "Design and Analysis of A Look-Ahead Scheduling Scheme to Support Pause-Resume for Video-on-Demand Applications", *ACM Multimedia Systems Journal*, Vol. 3, No. 4, 1995, pp. 137-150.