Optimization Issues in Multimedia Systems

Charu C. Aggarwal, Joel L. Wolf and Philip S. Yu
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598

May 23, 1997

Abstract

Multimedia systems must meet stringent real-time performance criteria in order to satisfy customer requirements. Because of these criteria, and because of the rich structure inherent in video-on-demand (VOD) applications, there is both need and opportunity to employ sophisticated mathematical optimization techniques. In this paper we present an overview of several recent optimization algorithms for multimedia systems, concentrating on the techniques themselves. In particular, we will describe a VOD batching algorithm known as the *maximum factored queue length* policy, based in part on solving a simple instance of a so-called mathematical resource allocation problem. Next we will describe an optimization problem arising when employing a VOD technique known as adaptive piggybacking. This problem can be solved as a dynamic program, and the scheme which results is known as the *snapshot* algorithm. Finally we will describe a so-called *DASD dancing* algorithm for VOD disk load balancing, which depends on the solution to a set of three somewhat more advanced resource allocation problems.

1 Introduction

Recent advances in communications technology have generated considerable interest in multimedia applications in general, and in video-on-demand (VOD) systems in particular. In such a system, viewers have the flexibility of choosing both the video they want as well as the time at which they wish to watch it. Such a system can be implemented using a client-server architecture. A large set of videos is stored in a centralized server. The clients consist of viewers who make requests from the server. Whenever there is a request for a particular object, the video is accessed from the disks in the storage server and transmitted to the viewer isochronously. Failure to effectively manage scarce system resources when viewers request videos may result in overly long waits for the start of playback. Failure to manage these resources during the playback of the videos themselves may result in interruptions, or *hiccups*.

The point is that multimedia applications require the careful solution of real time scheduling and resource allocation problems. Fortunately, VOD applications also have a rich inherent structure, making careful resource management feasible. The point of this paper is that many problems which arise in the design of VOD algorithms can be posed and neatly solved as optimization problems. We illustrate this with a number of examples. It is the purpose of this paper to focus on the optimization techniques themselves. It is our thesis that these techniques belong in the toolkit of multimedia systems designers. Naturally, we provide pointers to papers which provide further systems-related details, other similar algorithms, experimental comparisons and so on.

We next motivate and give an overview of the three main optimization algorithms developed in this paper.

Because of the isochronous requirement, there exists a maximum number of concurrent video streams that a given disk or striped disk array can support [28, 26]. Hence there automatically exists a constraint on the maximum number of video streams which can be supported concurrently by a video server. This maximum is referred to as the server *stream capacity.* If the number of outstanding viewer requests is larger than the available server stream capacity, a new viewer request cannot be satisfied immediately. The elapsed time between the arrival of a video request and the time when the service to the display device is actually initiated is known as the *latency* of the request. A viewer may *defect* from the queue if the wait time becomes excessive, and may cancel the service completely if such problems persist. So the goal is to provide good quality of service with few defections while consuming the least amount of server capacity. One way to accomplish this goal is to develop techniques which handle multiple requests using a single video stream. Three such techniques are standardly employed, either alone or in combination.

One common approach to reducing video stream requirements is known as *batching.* The method of batching was proposed originally in [3]. The basic idea is to *intentionally* delay the requests for the different videos for an amount of time known as the *batching interval*, so that additional requests for the same video arriving during the current batching interval can be serviced using the same stream. Thus, requests made by many different viewers for the same video can share a common video stream if these requests are spaced closely enough. We assume that the viewer has a choice of selecting from a relatively large number of videos. Consequently, whenever a stream becomes available at the server end, the question arises as to which video if any is best to schedule at that particular moment in time. There has been a considerable amount of research devoted to batching policies [8, 14, 23]. In this paper we describe the *maximum factored queue length* algorithm introduced in [1], which is based on solving a well-known optimization problem. Specifically, we formulate and solve a simple version of what is known mathematically as a *resource allocation problem.*

Another common video stream reduction technique is known as *bridging.* Bridging involves the use of a buffer space in memory to retain moving windows of certain videos behind as the videos play. Suppose that a contiguous segment of a particular video is being buffered as it plays for a particular viewer. Then a viewer who is trailing the original viewer by an amount less than this segment can be serviced from this buffered stream instead of from disk. More details on the technique of bridging may be found in [19, 29]. The key issue is to determine which segments of which instances of video streams should be buffered. One can actually derive an optimal bridging algorithm as a greedy solution to an extremely simple optimization problem, but we shall omit this in the current paper.

Recently, a third technique for reducing video stream requirements was introduced. The technique is called *adaptive piggybacking* [15, 16]. In piggybacking the display rates are altered even while the video streams 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. Suppose two streams are displaying a common video are in progress. Then the leading stream can be played at a slower rate, and the trailing stream at a faster rate. Assuming the interval between the two videos 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. We describe a dynamic programming algorithm for finding an optimal adaptive piggybacking strategy. (Another straightforward optimization problem needs to be solved as well.) This so-called *snapshot* algorithm was originally proposed in [2].

We have already indicated that each disk or disk array can only support some fixed number of video streams. This maximum depends on the performance characteristics of the disks in question. We return to this issue now. Because the I/O subsystem is generally the performance and cost bottleneck of a VOD server, the challenge is to balance the load on the existing disks effectively, so as to maximize the throughput the system can achieve. Overutilization of disks can cause either video service interruptions to current customers or rejection of new customer demands, neither of which is desirable. On the other hand, underutilization is wasteful. Said differently, VOD systems present a real-time disk scheduling problem which is highly non-trivial but must be solved satisfactorily almost all of the time. We will describe an algorithm [27] known as *DASD dancing* for solving this as a disk load balancing problem. (Recall that disks used to be known, in IBM parlance, as *direct access storage devices* or *DASDs*). The problem can be posed, in part, as a set of three resource allocation problems. These problems are more elaborate and difficult to solve than the one described for the batching algorithm above.

The remainder of this paper is organized as follows. Section 2 describes the maximum factored queue length batching algorithm. The snapshot algorithm for adaptive piggybacking is described in Section 3. Section 4 describes the DASD dancing algorithm for disk load balancing. Section 5 contains a summary.

2 Batching

While initial viewer delay is a necessary evil in VOD batching schemes, one obviously wants to keep such delays as small as possible. In this section we present a simple batching optimization model which attempts to minimize average latency time subject to a constraint on server stream capacity. The resulting formulation is a particularly easy instance of a so-called *resource allocation problem* [17].

Assume that there are N videos, and that the request frequency of the ith video is f_i. If L_i denotes the length of video i, then $L = (\sum_{i=1}^{N} f_i \cdot L_i)/(\sum_{i=1}^{N} f_i)$ is the average video length. Assume that the server capacity (in terms of the number of streams) is S. Consequently, the average number of streams which are scheduled by the server per unit time at full capacity is S/L. Suppose that $t_1, t_2, \ldots, t_N$ are the *average time intervals* at which the videos $1, 2, \ldots, N$ are batched. (These are the decision variables of the optimization problem.) Then the average latency for a video of type i is equal to $t_i/2$. The objective in this model is to minimize the average latency of the viewers. Thus, modulo a constant which we can ignore, we want to minimize the sum of the latency times of all the requests which arrive in a unit interval of time. The expected number of such requests for video i is f_i. Consequently, the expected sum of the latency times of all the requests which arrive within a unit interval is equal to $\sum_{i=1}^{N} f_i \cdot t_i/2$. This is the objective function we wish to minimize.

We need a constraint which limits the number of video streams which can be scheduled at any moment of time. On average, the number of streams for video i scheduled per unit of time is approximately equal to $1/t_i$. (This is not entirely rigorous, but if X is a random variable with a relatively small deviation, the approximation $E[1/X] \approx 1/E[X]$ is fairly accurate.) Thus, at full capacity the total number of streams of all video types scheduled per unit of time is $\sum_{i=1}^{N} 1/t_i$, which must be equal to S/L. Thus, eliminating another constant, we wish to minimize

$$\sum_{i=1}^{N} f_i \cdot t_i \qquad (1)$$

subject to the constraints

$$\sum_{i=1}^{N} 1/t_i = S/L \tag{2}$$

$$\text{and } t_i > 0 \qquad \forall i. \tag{3}$$

The last constraint ensures the positivity of the variables. To solve this optimization problem most efficiently, we make a change of variables. Let $\rho_i = 1/t_i$. Then the objective function $\sum_{i=1}^{N} f_i/\rho_i$ becomes convex and separable. (The latter term means that the objective function is the sum of separate functions of the various decision variables.) The first constraint $\sum_{i=1}^{N} \rho_i = S/L$ becomes linear, and because of its special form is known as a *resource allocation* constraint. The positivity constraint becomes $\rho_i > 0$. This transformation thus yields a *continuous separable convex resource allocation problem.* This is perhaps the simplest possible resource allocation problem, and it is known that the optimal solution occurs when the derivatives of each f_i/ρ_i are equal [11, 17]. For the benefit of readers unfamiliar with this fact, we derive it briefly: Because of the nature of the problem, the Kuhn-Tucker conditions [4] are necessary and sufficient for the existence of a global optimum solution. These Kuhn-Tucker conditions are as follows:

$$f_i - u/t_i^2 = 0 \tag{4}$$

$$\text{and } \sum_{i=1}^{N} 1/t_i = S/L. \tag{5}$$

Here u is the Lagrange multiplier associated with the constraint. From the above set of variables it is easy to see that the average batching intervals for the different videos should be related as follows:

$$t_1 \cdot \sqrt{f_1} = t_2 \cdot \sqrt{f_2} = \ldots = t_n \cdot \sqrt{f_n}. \tag{6}$$

(We have ignored the positivity constraints, but it can be seen that the optimum solution automatically satisfies them.) Note that the above equations hold when the derivatives of the convex functions in the transformed problem are equal, as observed previously. We note that this derivation applies in general to any separable convex resource allocation problem.

In the absence of any defections from the system, the average queue length q_i of the ith video at the time of batching is equal to $q_i = t_i \cdot f_i$. Thus, using the above equations we get the following goals for the queue lengths at the time of batching:

$$\frac{q_1}{\sqrt{f_1}} = \frac{q_2}{\sqrt{f_2}} = \ldots = \frac{q_n}{\sqrt{f_n}}. \tag{7}$$

In other words, we have the following intuitive result: In order to achieve the smallest average latency, the batch queue lengths should be proportional to the square roots of the relative frequencies of the arrivals of the different videos. Let the *factored queue length* of a video be defined as its queue length divided by the square root of its request frequency. The result above suggests a good *greedy* scheduling policy for the VOD problem, as follows.

Whenever a stream becomes available, schedule the video with the largest factored queue length.

This scheme is called *Maximum Factored Queue Length (MFQL)* policy. To implement this policy in practice, however, one needs to robustly estimate the request frequencies of the various videos. To handle this issue, let Δt_i denote the interval since the last time that video i was scheduled. Consider the standardly employed approximation $f_i \approx q_i/\Delta t_i$. Making this substitution, the factored queue length of video i is given by $\sqrt{q_i \cdot \Delta t_i}$. Since we need to choose the largest among this set of numbers, we consider instead the square of these factored queue lengths. In this way one obtains a more naturally implementable variant of MFQL, as follows:

Whenever a stream becomes available, schedule the video with the largest value of $q_i \cdot \Delta t_i$, where q_i is the queue length and Δt_i is the time since the last scheduling of video i.

We refer the reader to [1] for further discussion of the MFQL policy. Among other things that paper discusses the effects of viewer defections on the algorithm. Other batching algorithms are also discussed there, and compared via simulation experiments to MFQL. Two of these are the *Maximum Queue Length (MQL)* and *First Come First Served (FCFS)* policies. For further information on VOD batching see [8, 14, 23].

3 Adaptive Piggybacking

An adaptive piggybacking merging policy must decide which video streams to play at a fast speed and which video streams to play at a slow speed. The issue here, of course, is to find a piggybacking policy for which the savings in bandwidth is maximized. The *snapshot* algorithm described in this section consists of two components. The first of these is a *generalized simple merging* policy, and in some ways it mimics the simple merging policies defined in [15]. This policy can be regarded as a piggyback merging policy in its own right, though it will form only the first portion of the overall snapshot policy. The roll of the generalized simple merging policy as used in the snapshot algorithm is to handle the merging of stochasticly arriving videos for a predetermined interval of time. The roll of the second component of the snapshot algorithm is to handle the deterministic problem which remains. (We intentionally omit consideration of pause/resume, fast forward/rewind capabilities here for the sake of simplicity.) Both components involve optimization algorithms. The snapshot algorithm turns out to be optimal over a large class of reasonable piggybacking policies.

3.1 Generalized Simple Merging Policy

Consider a single video whose length, in frames, is given by L. 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 [15]. We prefer to assume that the normal and slow speeds are identical.) 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

$$W_m = \frac{S_{max} - S_{min}}{S_{max}} \cdot L. \tag{8}$$

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

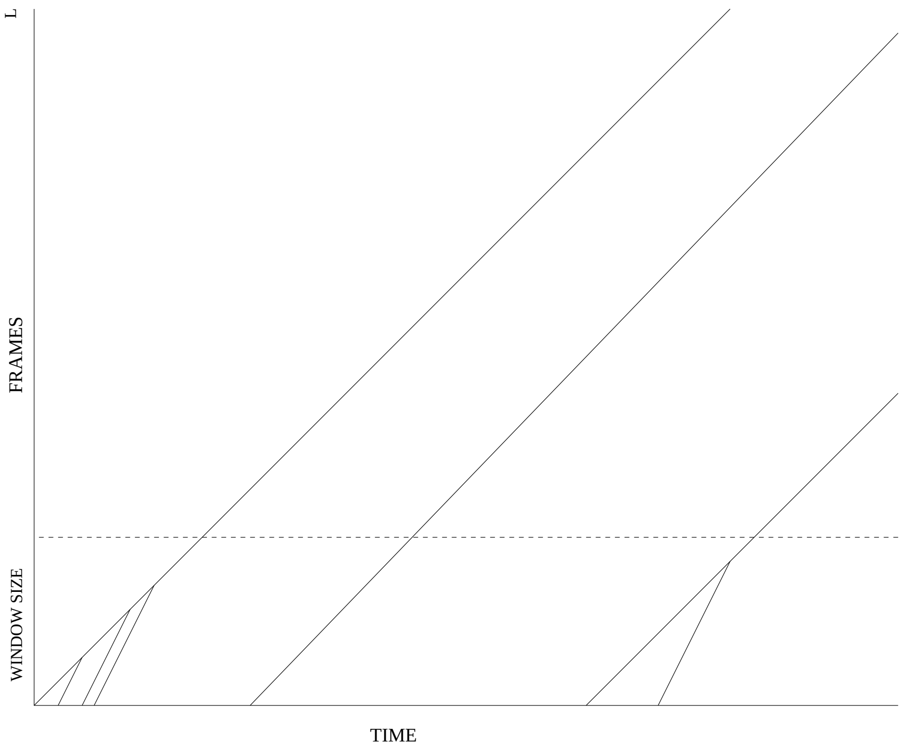

Figure 1: Generalized Simple Merging Policy

The goal of the generalized simple merging policy is to optimize W as a function of the forecasted arrival rate. Assume 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,

$$E[F] = P_{fast} \cdot E[F_{fast}] + P_{slow} \cdot E[F_{slow}]. \tag{9}$$

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

$$F_{fast} = \frac{p \cdot S_{max}}{S_{max} - S_{min}}. \tag{10}$$

Note that the algorithm must be 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

$$E[F_{fast}] = \frac{W/2 \cdot S_{max}}{S_{max} - S_{min}}. \tag{11}$$

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 is equal to $\lambda \cdot W/S_{min}$. Consequently, the fraction of fast streams in the system is approximately equal to

$$P_{fast} = \frac{\lambda W/S_{min}}{\lambda W/S_{min} + 1}. \tag{12}$$

Substituting the above values in Equation 9, we obtain the following relationship:

$$E[F] = \frac{\lambda W}{\lambda W + S_{min}} \cdot \frac{W S_{max}}{2(S_{max} - S_{min})} + \frac{S_{min}}{\lambda W + S_{min}} \cdot L. \tag{13}$$

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:

$$W \cdot \frac{S_{max}}{S_{max} - S_{min}} \leq L. \tag{14}$$

Ignoring the constraint for the time being, we set

$$\frac{dE[F]}{dW} = 0. \tag{15}$$

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

$$W^2 + \frac{2S_{min}}{\lambda} \cdot W - \frac{LS_{min}(S_{max} - S_{min})}{\lambda S_{max}} = 0. \tag{16}$$

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

$$W^* = -\frac{S_{min}}{\lambda} + \sqrt{\left(\frac{S_{min}}{\lambda}\right)^2 + 2 \cdot \frac{LS_{min}(S_{max} - S_{min})}{\lambda S_{max}}}. \tag{17}$$

The second derivative is positive, and an easy check shows that this value of W^* automatically satisfies constraint 14. Consequently, W^* is the optimal window size.

3.2 Dynamic Programming Algorithm

Here we describe the second component of the snapshot algorithm, which is based on dynamic programming. 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. In the presumed absence of pause/resume and fast forward/rewind this scenario is entirely deterministic. 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 shall solve 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 2 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.)

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

$$b(n-1) = \frac{1}{n}\binom{2n-2}{n-1}. \tag{18}$$

See [7] for details. The Catalan number $b(n)$ can be approximated via Stirling's approximation as

$$b(n) \approx \frac{4^n}{\sqrt{\pi}n^{3/2}}. \tag{19}$$

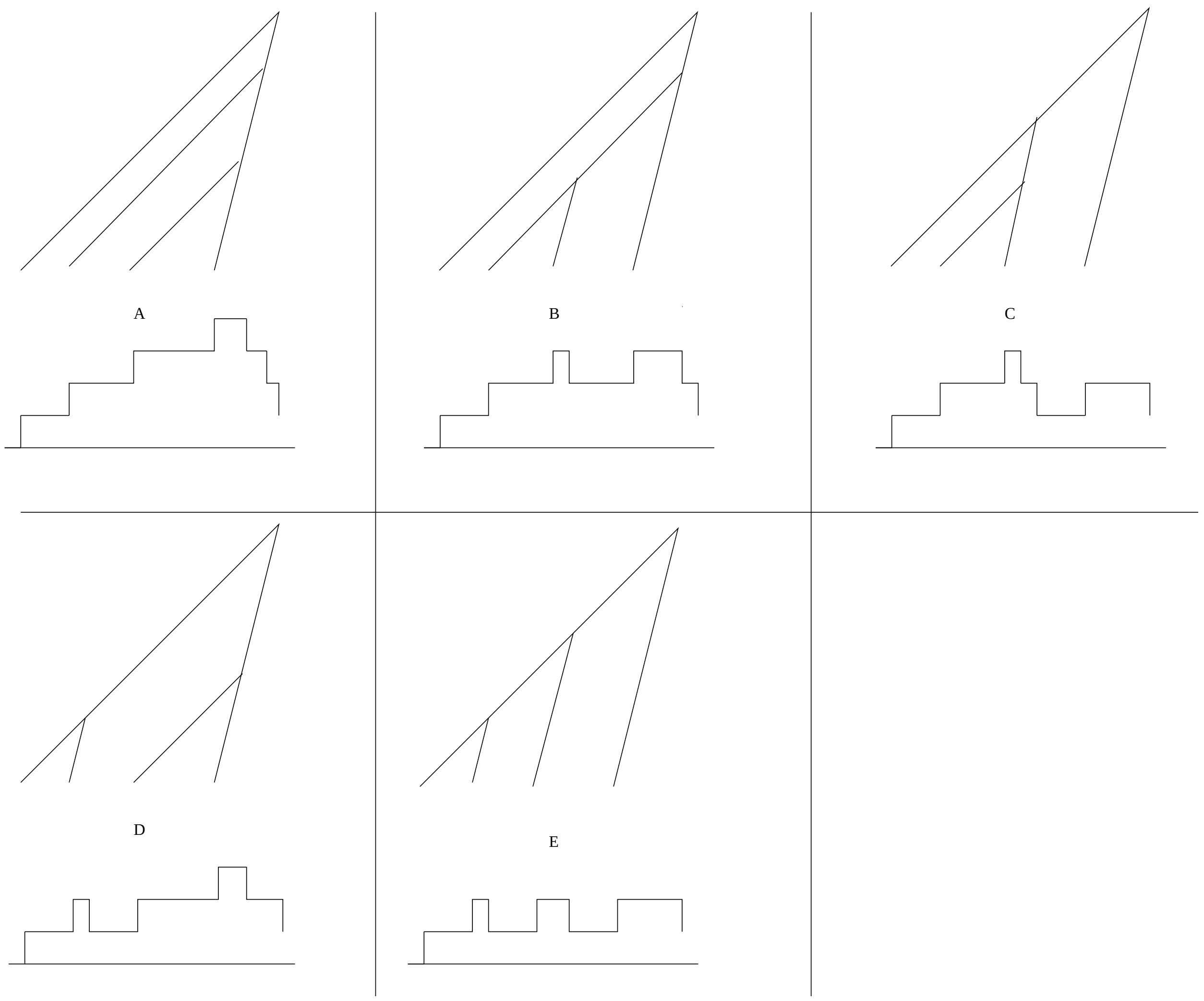

Figure 2: Typical Binary Tree Alternatives and Costs

Thus 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, which we now describe.

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 $i, ..., j$ 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 minimum speed and stream j moving at the maximum speed. So we obtain

$$P(i,j) = f_i + \frac{S_{min} \cdot (f_i - f_j)}{S_{max} - S_{min}} \tag{20}$$

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

$$P(i,i) = f_i. \tag{21}$$

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 $i, ..., j$ occur. Denote the corresponding binary tree by $\mathcal{T}(i, j)$. It is easy to see that

$$C(i,i) = L - f_i \tag{22}$$

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 $i, ..., k$ and the right subtree will contain the leaf nodes corresponding to streams $k + 1, ..., j$. 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 $C(i, k) + C(k + 1, j) - (L - P(i, j))^+$, the last term indicating the (potential) savings of the final merge at position $P(i, j)$. Therefore the optimal policy in which only arrivals $i, ..., j$ occur has a left subtree with leaf nodes corresponding to $i, ..., k^*$ and a right subtree with leaf nodes corresponding to $k^* + 1, ..., j$, where

$$k^* = \text{argmin}_{i \leq k < j}\{C(i,k) + C(k+1,j) - (L - P(i,j))^+\}. \tag{23}$$

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 $C(1,n)$.

Thus after an initialization step taking $O(n)$ steps to compute $P(i,i)$, $C(i,i)$ and $\mathcal{T}(i,i)$ for each i, there is an iterative step consisting of a pair of nested loops. Each step in the inner loop involves $O(n)$ comparisons to compute $P(i,i+m)$, $C(i,i+m)$ and $\mathcal{T}(i,i+m)$ for each i and fixed m. There are $O(n)$ such steps. In the outer loop we increase m from 1 to $n-1$, also a total of $O(n)$ steps. All told, we have:

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

The dynamic program algorithm presented has analogues in the problem of optimal polygon triangulation via the natural correspondence between binary trees and convex polygons [7, 24]. This, in turn, has lead to algorithms for parse trees and the like. (Ken Sevcik [22] has pointed out to us that under certain circumstances the dynamic programming solution can be accomplished via a slightly modified telescoping algorithm which has complexity $O(n^2)$ rather than $O(n^3)$. Such a case arises, for example, in the construction of binary search trees for common English words, as described in [20]. Unfortunately, the structure of the objective function for this problem does not quite match ours, and this approach will not work for the adaptive piggybacking problem.)

3.3 Snapshot Algorithm

We now combine the (stochastic) generalized simple merging policy and the (deterministic) dynamic programming technique into a practical window-based piggybacking policy known as the *snapshot* algorithm. Assume again 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 [15] 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.

We refer the reader to [2] for further discussion of the snapshot algorithm. In particular, that paper discusses special features such as pause/resume and fast forward/rewind. Pseudocode for all of the snapshot components is provided. Other adaptive piggybacking algorithms are also discussed there, and compared via simulation experiments to the snapshot algorithm. We remark again that adaptive piggybacking see [15, 16]. The piggyback merging algorithms *simple merging*, *odd-even* and *greedy* policies are taken from these papers.

4 Disk Load Balancing

We have already suggested that VOD disk load balancing and scheduling appears to be a complicated problem. On the positive side, video streams, once requested, represent a logically defined unit of load to the disks. They are read-only in nature, and basically predictable in length. Since pause/resume and fast forward/rewind are relatively infrequent events, one can assume that the viewer watches a video without interruption for long stretches at a time. Video on disks is typically stored in MPEG compressed format [18], and different videos require similar megabits per second rates. Thus, although a disk has a well-defined maximum acceptable I/O bandwidth, that limit can be achieved in a manner largely independent of which videos are actually being played.

The load balancing problem is made more complicated by the fact that some videos are vastly more popular than others at any given time. Furthermore, this highly skewed distribution varies on a weekly, daily, and even hourly basis, due to changing video popularity and customer mix.

We should observe that some videos may be hot enough to justify their being stored in main memory, not disk. Similarly, other levels of the storage hierarchy (such as tape) may be appropriate for cold videos. We will ignore these issues here, and consider only the videos which reside on disk. The popularity of the hottest such videos can often be so great that storing them on a single disk may not be feasible from a performance standpoint. Playing them from a single disk may cause that disk to be overloaded. A partial solution to this is to use *striped disks* [6]. By combining, for example, each group of 8 disks into an 8-way *disk striping group* (DSG), the load generated by each video stream can be cut correspondingly, and the overall load across each of the 8 disks essentially balanced. Nevertheless, striping does have its disadvantages, for example availability in the event of disk failures. These tradeoffs imply that the degree of striping should be limited to some extent. Thus, depending on the required throughput it will still typically be necessary to create multiple copies of some videos. Likewise, given a fixed striped disk configuration and a fixed number of videos to be offered, there may actually be spare disk space available for replicating certain videos.

The idea behind *DASD dancing* is to take advantage of multiple video copies on disk to solve the VOD load balancing problem effectively. However, the algorithm does work synergistically with disk striping, and is more effective than disk striping alone. The DASD dancing algorithm consists of two stages. One is static, and the other dynamic.

The static stage decides, based on video forecasts and the hardware configuration, which videos should reside in memory, if any, and which videos should reside on tape. For the remaining videos, to be stored on disk, it then employs an optimization technique for solving the so-called *apportionment problem* to determine the optimal number of copies per video. Again, this technique is borrowed from the theory of resource allocation problems [17]. Finally, there is an algorithm which makes good quality assignments of videos to DSGs. The static stage is meant to be run periodically, perhaps once per day. It can be run either from scratch or in incremental mode. The latter mode allows for constraints which limit the number of copy and assignment changes. While the various static stage components involve interesting optimization problem algorithms, we will only have space in this paper for a brief discussion of the apportionment problem.

The dynamic stage handles the on-line scheduling of videos to DSGs, based on the output of the static stage and on fluctuating video customer requests. These fluctuations occur because videos start and complete, and also because customers may pause and resume in-progress videos. The algorithm uses an optimization technique for solving so-called *class constrained resource allocation problems* [9, 25] to determine optimal load balancing goal at any given moment. Much of the time, the decision on which DSG should handle a new video request or resumed video can be performed on a *greedy* basis: Specifically, we play the video on that DSG which is relatively most underloaded among those DSGs which have a copy. However, periodically load balancing using this approach may degrade. When the quality of the disk load balancing differs from the goal by more than a predefined threshold, a *DASD dance* is initiated. This dance is based on an algorithm which is also graph-theoretic, and has the effect of shifting load from relatively overloaded to relatively underloaded DSGs. As an example, consider a situation in which the video A is assigned to DSGs 1 and 2, video B is assigned to DSGs 2 and 3, and video C is assigned to DSGs 3 and 4. If DSG 1 is overloaded and DSG 4 is underloaded, the DASD dancing algorithm might change a currently playing stream of video A from DSG 1 to 2, a currently playing stream of video B from DSG 2 to

3, and a currently playing stream of video C from DSG 3 to 4. The directed graph

$$1 \xrightarrow{\text{A}} 2 \xrightarrow{\text{B}} 3 \xrightarrow{\text{C}} 4$$

represents this scenario neatly, with the nodes corresponding to DSGs and the directed arcs corresponding to videos. The effect of this three-step "dance" is to lower the load on DSG 1 by one and raise the load on DSG 4 by one. There is no net effect on DSGs 2 and 3. The actual transfer of plays can be achieved via a simple *baton passing* primitive.

4.1 Determining Load Balancing Objectives

First we fix some notation. Let M denote the number of videos stored on disk, and D denote the number of DSGs. (If we let $\mathcal{S}$ denote the degree of striping employed, then $\mathcal{S}D$ is the number of actual disks.) Let $\mathcal{A} = (a_{i,j})$ denote the assignments of video copies to DSGs. Thus $\mathcal{A}$ is a $\{0,1\}$ $M \times D$ matrix defined by

$$a_{i,j} = \begin{cases} 1 & \text{if a copy of video } i \text{ exists on DSG } j, \\ 0 & \text{otherwise.} \end{cases}$$

Associated with each DSG j is a maximum acceptable number L_j of concurrent video streams. This number depends on the performance characteristics of the disks, and is chosen to ensure that the real-time scheduling problem of reading the videos within a required fixed deadline can be solved successfully. To avoid reaching this threshold and balance the load on the disks we shall employ a function F_j for each DSG j which progressively penalizes loads approaching L_j. Thus F_j can be any convex increasing function on the set $\{0, \ldots, L_j\}$ satisfying $F_j(0) = 0$. (An analytically or experimentally derived performance curve for the DSG will typically have these properties.) Assume at a given moment that there are λ_i streams of video i in progress. We break these down further into $\lambda_{i,j}$ streams playing on DSG j. Thus $\lambda_i = \sum_{j=1}^{D} \lambda_{i,j}$, and $\lambda_{i,j} = 0$ whenever $a_{i,j} = 0$. We let $\Lambda = \sum_{i=1}^{M} \lambda_i$ denote the total number of all video streams in progress.

The disk loads are optimally balanced given the current load and video-to-DSG assignments when the objective function

$$\sum_{j=1}^{D} F_j\left(\sum_{i=1}^{M} x_{i,j}\right) \tag{24}$$

is minimized subject to the constraints

$$x_{i,j} \in \{0, \ldots, L_j\}, \tag{25}$$

$$\sum_{j=1}^{D} x_{i,j} = \lambda_i \tag{26}$$

$$\text{and } x_{i,j} = 0 \quad \text{if } a_{i,j} = 0. \tag{27}$$

Note that for the optimal solution, $X_j = \sum_{i=1}^{M} x_{i,j}$ represents the desired load on DSG j. Our goal will be to ensure that X_j and $\sum_{i=1}^{M} \lambda_{i,j}$ are always close to each other for each j.

Now the optimization problem described above is a special case of the so-called *class constrained resource allocation problem.* The classes here correspond to the videos, and the constraints refer to the infeasibility of playing a video on a DSG on which that video does not exist. As shown independently in [9, 25], class constrained resource allocation problems can be solved exactly and efficiently using a graph-theoretic optimization algorithm.

Although the algorithm in [25] is too long to present here in full, we will present an overview of it as it applies to the special case above. There are two reasons to do so: First, the algorithm will be called as part of the dynamic phase scheme, in order to set the target DSG loads. Second, the graph technique of the original algorithm in mimicked in the next component of the dynamic phase scheme. Assuming a feasible solution exists, the algorithm proceeds in Λ steps. A directed graph is created and maintained throughout the course of the algorithm. The nodes of the graph are the DSGs $1, \ldots, D$, plus a dummy node which we label as node 0. We also create and modify a *partial* feasible solution $\{x_{i,j} | i = 1, \ldots M, j = 0, \ldots, D\}$. Initially, this partial feasible solution is set for each i to have $x_{i,0} = \lambda_i$, and $x_{i,j} = 0$ for all $j = 1, \ldots, D$. The directed graph at any step has a directed arc from a node $j_1 \in \{0, \ldots, D\}$ to a node $j_2 \in \{1, \ldots, D\}$ if there is at least one video i_1 satisfying

(1) $a_{i_1,j_1} = a_{i_1,j_2} = 1$,

(2) $x_{i_1,j_1} > 0$,

(3) $\sum_{i=1}^{M} x_{i,j_2} < L_{j_2}$.

(Note that there may be directed arcs *from* node 0, but there are no directed arcs *to* node 0.) The general step of the algorithm finds, among all nodes $j \in \{1, \ldots, D\}$ for which there is a directed path from 0 to j, the node for which the first difference

$$F_j(\sum_{i=1}^{M} x_{i,j} + 1) - F_j(\sum_{i=1}^{M} x_{i,j}) \tag{28}$$

is minimal. If no such node exists, the algorithm terminates with an infeasible solution. Otherwise, an acyclic directed path is chosen from 0 to the optimal node. For each directed arc (j_1, j_2) in this path, the value of x_{i_1,j_1} is decremented by 1 and the value of of x_{i_1,j_2} is incremented by 1 for an appropriate video i_1. Performing this step over all directed arcs has the effect of removing one unit of load from the dummy node, and adding one unit of load to the optimal node. There is clearly no net effect on the load of the intermediate nodes. Thus the dummy node serves as a staging area for the resources, one of which is released in each step into the DSG nodes. Bookkeeping is then performed on the graph, which may modify some directed arcs and potentially disconnect certain nodes, and the step is repeated. After Λ steps the algorithm terminates with an optimal solution to the original class constrained resource allocation problem. The complexity of this algorithm is $O(D(\Lambda D + D^2 + \Lambda M))$. See [25] for further details.

4.2 Dynamic Scheme

With these preliminaries, we are now ready to discuss the dynamic algorithm itself. Clearly, stream requests are increased by 1 when a customer starts a new video or resumes a currently paused video. (Similarly, stream requests are decreased by 1 when a customer finishes a video or pauses a currently playing video; we do not actively concern ourselves actively with these.) Normally, handling request increases can be accomplished by employing the obvious *greedy* algorithm. In other words, if a new stream of video i_1 is to be added, that DSG j satisfying $a_{i_1,j} = 1$ whose first difference

$$F_j(\sum_{i=1}^{M} \lambda_{i,j} + 1) - F_j(\sum_{i=1}^{M} \lambda_{i,j}) \tag{29}$$

is minimal is chosen. However, periodically this approach may degrade. To check this, we solve the class constrained resource allocation problem above to obtain optimal DSG loadings given the current video requests. Reindexing these DSGs according to decreasing values of $\sum_{i=1}^{M} \lambda_{i,j} - X_j$ puts them in order of most overloaded to most underloaded, relative to optimal. (To fix notation, suppose that the first D_1 DSGs are relatively overloaded, and the last D_2 DSGs are relatively underloaded.) If the values $\sum_{i=1}^{M} \lambda_{i,j} - X_j$ differ from zero by more than some fixed threshold T according to any reasonable norm, the DASD dancing component of the dynamic phase algorithm will be initiated. (The excess $\sum_{i=1}^{M} \lambda_{i,1} - X_1$ of the relatively most overloaded DSG is one such norm.) We let B denote the value of this norm, an indicator of load balancing badness.)

The DASD dancing component is also graph-theoretic, maintaining at all times a directed graph G defined as follows: The nodes correspond to the DSGs. (There is no dummy node.) For each pair j_1 and j_2 of distinct nodes, there is a directed arc from j_1 to j_2 provided there exists at least one video i_1 for which

(1) $a_{i_1,j_1} = a_{i_1,j_2} = 1$,

(2) $\lambda_{i_1,j_1} > 0$,

(3) $\sum_{i=1}^{M} \lambda_{i,j_2} < L_{j_2}$.

As before, the existence of a directed arc signifies the potential for reducing the load on one DSG, increasing the load on another without exceeding the load capacity, and leaving the loads on other DSGs unaffected. We try, of course, to move load from relatively overloaded to relatively underloaded DSGs. The algorithm has a main routine and one subroutine. The main routine wakes up whenever the badness threshold is exceeded, and continues to call the subroutine until either the load balancing is satisfactory or no further improvements can be made. The subroutine performs the DASD dance from the most overloaded possible DSG to the most underloaded possible DSG.

Note that the requirement to proceed along a shortest directed path implies that each directed arc involves the baton-passing transfer of a *different* video. For a single arc baton passing can be accomplished using a synchronization primitive. This is not difficult to implement, but we omit details here. The point is that this dynamic scheme will have the effect of balancing the load to a larger degree than would be possible without transfering videos dynamically.

We illustrate the DASD dancing dynamic load balancing scheme via a simple example. Consider Figure 3, which shows the directed graph for a scenario with 6 DSGs and 4 videos. (In the figure we show a single arc between DSGs whenever a video is stored on both of them. We draw in the

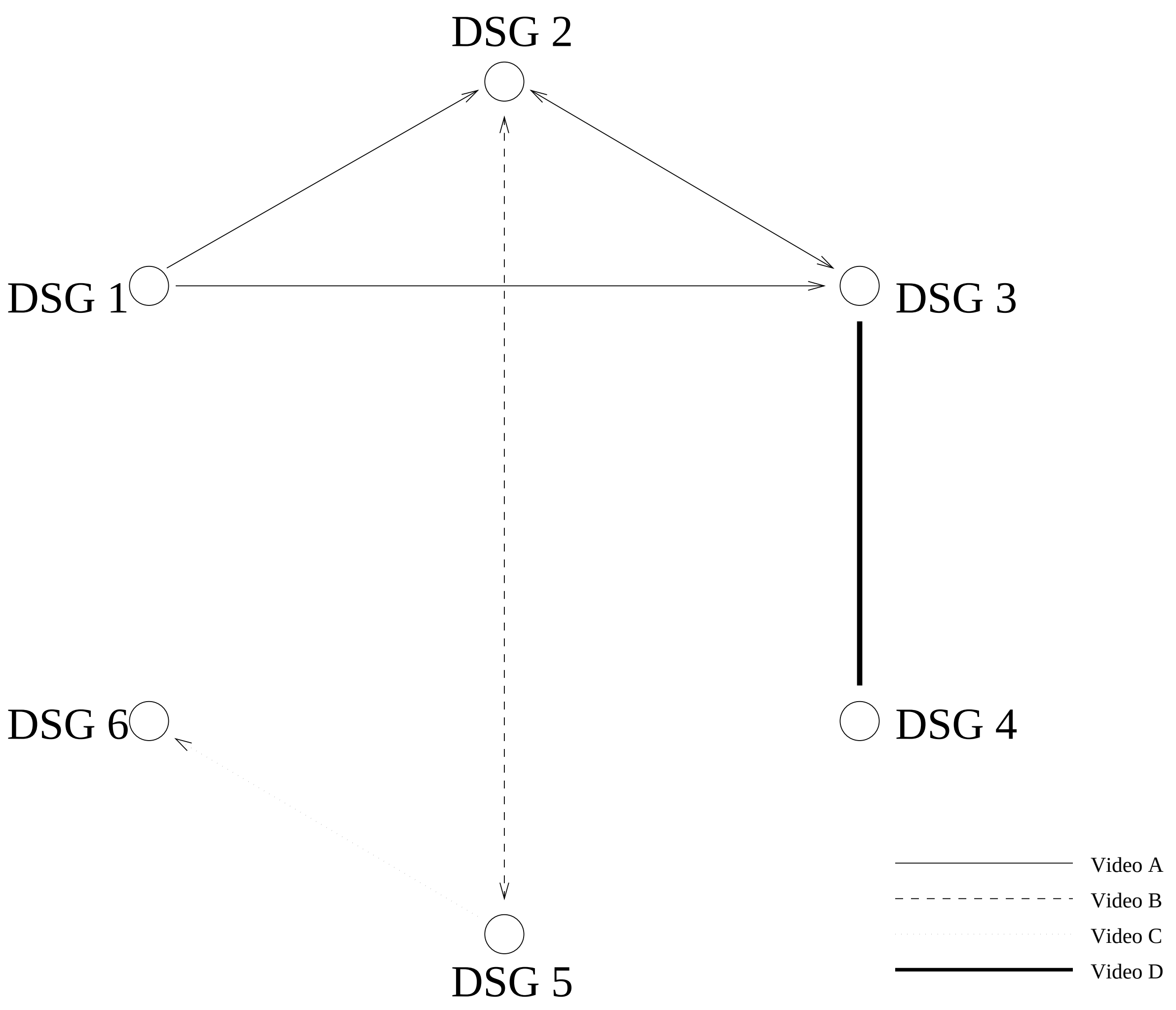

Figure 3: Sample DASD Dance

arrows (two way, one way or no way) as appropriate. Suppose that DSG 1 is overloaded, while DSG 6 is underloaded. Then a DASD dance to help fix this imbalance might consist of transfering of one stream for video A from DSG 1 to DSG 2, one stream for video B from DSG 2 to DSG 5, and one stream for video C from DSG 5 to DSG 6.

We should observe that it is possible to shorten the execution time of the dynamic phase scheme by eliminating the calls to the class constrained resource allocation problem algorithm. Specifically, consider the corresponding resource allocation problem in which the classes have been removed: Thus we wish to minimize the objective function

$$\sum_{j=1}^{D} F_j(x_j) \tag{30}$$

subject to the constraints

$$x_j \in \{0, \ldots, L_j\} \tag{31}$$

$$\text{and } \sum_{j=1}^{D} x_j = \Lambda. \tag{32}$$

By definition, the value of the objective function for this problem is less than or equal to the corresponding value for the class constrained problem. But if the video-to-DSG assignment algorithm described in the next section is done well, the optimistic assumption that the values will be close is generally justified. Thus we can use each value x_j as a surrogate for X_j. This new optimization problem is solvable by an algorithm [11] with computational complexity $O(D + \Lambda \log D)$. Because of its incremental nature, this algorithm computes the optimal solution for all values between 1 and Λ as it proceeds. Thus these can be stored and simply looked up as needed, rather than being computed each time. (There exist faster algorithms [12, 13] for this resource allocation problem, but they are not incremental in nature. See also [17] for a further details.)

If all the disks are homogeneous in the sense that they have identical performance, we can do better still. In this case we can assume that $L = L_j$ and $F = F_j$ for each DSG j, and then the resource allocation problem solves trivially (modulo integrality considerations), with each $x_j = \Lambda/D$.

4.3 Static Scheme

As indicated, we will not have space to discuss the actual assignments of videos to DSGs. We simply make the comment that in both the incremental and from scratch versions, the goal is to obtain high connectivity in the graph H whose vertices are the DSGs and whose arcs correspond to videos resident on both DSGs. This undirected graph mimics the directed graph G. The idea is that if H is highly connected then G will be also at most instants in time. This is highly desirable from the DASD dancing perspective. Full details on the static phase algorithm can be found in [27]. Here we content ourselves with a discussion of the first step in this process, namely the computation of the number $A_i \geq 1$ of required copies for each video i based on its forecasted request frequency $\hat{\lambda}_i$. Let A denote the (maximum) allowable number of video disk copies in the system. The problem of making each A_i roughly proportional to $\hat{\lambda}_i$ with the constraint that $A = \sum_{i=1}^{M} A_i$ is known as the *apportionment problem*, and arises naturally (as one might imagine) in the context of government representation. Many schemes have been proposed for this problem, and the DASD dancing algorithm adopts *Webster's Monotone Divisor Method.* (Ingenious alternative schemes are due, for example, to Hamilton, Adams and Jefferson, all figures from the American revolution. Details may be found in [17].)

We again refer the reader to [27] for further discussion of the DASD dancing algorithm. In addition to presenting the algorithmic details and pseudocode missing here, that paper compares DASD dancing via simulation experiments to a simple greedy load balancing algorithm. It also describes certain system related issues which arise in the implementation of DASD dancing, and discusses a VOD configuration problem which is in some sense dual to that of disk load balancing. See also [10] for a disk load balancing algorithm which depends more heavily on striping. The optimization of disk arm scheduling, while orthogonal to disk load balancing, is also an important topic. See [5, 21] for details.

5 Summary

In this paper we have focused on three multimedia problems which can be tackled in part by solving sophisticated optimization algorithms. The first and third of these problems pertain to batching and disk load balancing. The maximum factored queue length policy which solves the former problem and the DASD dancing policy which addresses the latter both rely on the solutions to resource allocation problems of one sort or another. The second problem pertains to adaptive piggybacking, and the snapshot algorithm which handles this problem relies on dynamic programming. We believe that VOD system designers may wish to become familiar with these optimization techniques.

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] C. Aggarwal, J. Wolf and P. Yu, “On Optimal Merging Policies for Piggybacking in Video-on-Demand Storage Systems”, *ACM Sigmetrics Conference*, Philadelphia, PA, 1996.

[3] Anderson, D. P., “Metascheduling for Continuous Media,” *ACM transactions on Computer Systems,* Vol. 11, No. 3, 1993, pp. 226-252.

[4] Avriel, M., “Nonlinear Programming Analysis and Methods,”

[5] Chen, M-S., Kandlur, D., and Yu, P., “Storage and Retrieval Methods to Support Fully Interactive Playout in a Disk-Array Based Video Server,” *Multimedia Systems*, Vol. 3, No. 3, pp. 126-135, 1995.

[6] P. Chen, E. Lee, G. Gibson, R. Katz and D. Patterson, “RAID: High Performance, Reliable Secondary Storage”, *ACM Computing Surveys*, vol. 26, no. 2, pp. 145-185, 1994.

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

[8] 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.

[9] A. Federgruen and H. Groenevelt, “The Greedy Procedure for Resource Allocation Problems: Necessary and Sufficient Conditions for Optimality”, *Operations Research*, vol. 34, pp. 909-918, 1986.

[10] R. Flynn and W. Tetzlaff, “Disk Striping and Block Replication Algorithms for Video File Servers”, *IEEE Multimedia Computing and Systems Conference*, Hiroshima, Japan, 1996.

[11] B. Fox, “Discrete Optimization via Marginal Analysis”, *Management Science*, vol. 13, pp. 210-216, 1966.

[12] G. Frederickson and D. Johnson, “The Complexity of Selection and Ranking in X+Y and Matrices with Sorted Columns”, *Journal of Computer and System Science*, vol. 24, pp. 197-208, 1982.

[13] Z. Galil and N. Megiddo, “A Fast Selection Algorithm and the Problem of Optimum Distribution of Efforts”, *Journal of the ACM*, vol. 26, pp. 58-64, 1981.

[14] Gelman, A. D., and Halfin, S., "Analysis of Resource Sharing in Information Providing Services," *Proceedings of IEEE Global Telecommunications Conference and Exhibition 1990,* Vol. 1, 1990.

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

[16] Golubchik, L., Lui J. C. S., and Muntz, R., "Adaptive Piggybacking: A Novel Technique for Data Sharing in Video-on-Demand Storage Servers", *Multimedia Systems*, Vol. 4, No. 3, 1996, pp. 140-155.

[17] Ibaraki, T., and Katoh, N., "Resource Allocation Problems: Algorithmic Approaches," MIT Press, Cambridge, MA, 1988.

[18] International Organization for Standardization, "DCT Coding of Motion Sequences Including Arithmetic Coder", *ISO-IEC/JTC1/SC2/WG8 N.*

[19] Kamath, M., Towsley D., and Ramamritham, K., "Buffer Management for Continuous Media Sharing in Multi-Media Database Systems,"

[20] D. Knuth, *The Art of Computer Programming*, Volume 3 (Sorting and Searching), Addison Wesley, 1973. Technical Report 94-11, University of Massachusetts, 1994.

[21] P. Rangan and H. Vin, "Designing File Systems for Digital Video and Audio", *12th ACM Symposium on Operating Systems*, 1991.

[22] K. Sevcik, *Personal communication.*

[23] Shachnai, H., and Yu, P., "The Role of Wait Tolerance in Effective Batching: A Paradigm for Multimedia Scheduling Systems," Research Report RC 20038, IBM T J Watson Research Center, Yorktown Heights, NY, 1995.

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

[25] A. Tantawi, D. Towsley and J. Wolf, "Optimal Allocation of Multiple Class Resources in Computer Systems", *ACM Sigmetrics Conference*, Santa Fe NM, 1988.

[26] Tobagi, F., Pang, J., Baird, R., and Gang, M., "Streaming RAID - A Disk Array Management System for Video Files," *Proceedings of the 1st ACM Multimedia Conference*, Anaheim, CA, 1993, pp. 393-400.

[27] J. Wolf, P. Yu and H. Shachnai, "DASD Dancing: A Disk Load Balancing Scheme for Video-on-Demand Computer Systems", *ACM Sigmetrics Conference*, Ottawa, Canada, 1995.

[28] Yu, P., Chen, M-S., and Kandlur, D., "Grouped Sweeping Scheduling for DASD-based Multimedia Storage Management," *Multimedia Systems*, Vol. 1, No. 3, pp. 99-109, 1993.

[29] Yu, P., Wolf, J., and Shachnai, H., "Design and Analysis of a Look-Ahead Scheduling Scheme to Support Pause-Resume for Video-on-Demand Applications," *Multimedia Systems*, Vol. 3, No. 4, pp. 137-149, 1995.