Multimedia Systems 5:358–370 (1997)

Multimedia Systems

Disk load balancing for video-on-demand systems

Joel L. Wolf[1], **Philip S. Yu**[1], **Hadas Shachnai**[2]

[1] IBM T.J. Watson Research Center, Yorktown Heights, NY 10598, USA
[2] Department of Computer Science, Technion IIT, Haifa 32000, Israel

Abstract. For a video-on-demand computer system, we propose a scheme which balances the load on the disks, thereby helping to solve a performance problem crucial to achieving maximal video throughput. Our load-balancing scheme consists of two components. The static component determines good assignments of videos to groups of striped disks. The dynamic component uses these assignments, and features a "DASD dancing" algorithm which performs real-time disk scheduling in an effective manner. Our scheme works synergistically with disk striping. We examine the performance of the proposed algorithm via simulation experiments.

1 Introduction

Consider a *video-on-demand* (VOD) computer system consisting of a central processor and a collection of shared disks, sometimes known as *direct access storage devices* (DASDs). VOD computer systems must be able to "play" multiple streams of many different videos simultaneously, based on customer demand. Most videos will be stored most cost effectively on disk. (A few videos may be popular enough to justify their being stored in main memory. Conversely, lower levels of the storage hierarchy, such as tape, may be appropriate for infrequently played videos. In this paper we will consider only those videos which are stored on disk.) 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 non-trivial but must be solved satisfactorily almost all of the time.

On the positive side, video streams, once demanded, represent a logically defined unit of load to the disks. They are read-only in nature, and basically predictable in length: customers are typically entitled to pause and then resume [22], but, generally speaking, they watch the videos without interruption for long stretches at a time. Video on disks is typically stored in MPEG-compressed format [25], and different videos require similar megabits-per-second rates. Thus, although a disk has a very 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 disk-scheduling 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. In fact, the popularity of the hottest 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*. As defined in [3], striping transparently distributes data over multiple disks to give the appearance of a single large and fast disk. By combining, for example, each group of eight disks into an eight-way *disk-striping group* (DSG), the load generated by each video stream can be cut correspondingly, and the overall load across each of the eight 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.

In this paper, we propose to take *advantage* of multiple video copies on disk to solve the VOD load-balancing problem very effectively. Our algorithm works synergistically with disk striping, and is more effective than disk striping alone. The load-balancing algorithm consists of two components. One is static, and creates the assignment of videos to DSGs. The other is dynamic, and performs the real-time video stream scheduling.

Correspondence to: J.L. Wolf
Part of this work was done while Hadas Shachnai was with IBM T.J. Watson Research Center

- The static component consists of two stages.
  - First, based on video demand forecasts, we employ an optimization technique for solving the so-called *apportionment problem* to determine the optimal number of copies per video. This technique is borrowed from the theory of resource allocation problems [5].
  - Second, we design an algorithm which makes good quality assignments of these optimal number of video copies to DSGs. This algorithm can be run either in *initial* or *incremental* mode. The initial mode is appropriate when configuring a new VOD system. Those videos with multiple copies are assigned first, using a graph-theoretic scheme based on a construct called *clique trees*. Then single-copy videos are assigned, using a *least loaded first* (LLF) scheme. The overall scheme is called CLLF. The incremental mode allows for constraints which limit the number of copy and assignment changes, and is thus practical for maintaining high-quality video-to-DSG assignments. A *neighborhood escape heuristic* [10] is employed. The incremental mode is meant to be run periodically, perhaps once per day or once per week. The exact frequency will depend on the volatility of the video demand forecasts.
- The dynamic component handles the real-time scheduling of videos streams to DSGs, based on the output of the static component and on fluctuating video customer demands. These fluctuations occur because videos start and complete, and also because customers may pause and resume in-progress videos. There are two stages to this component as well.
  - The first stage of the algorithm uses an optimization technique for solving so-called *discrete class-constrained resource allocation problems* [6, 18] to determine optimal load-balancing goals at any given moment, given the assignments of videos to DSGs. Specifically, the output of this stage is the optimal number of video streams on each DSG. This problem will need to be solved again whenever the overall load on the VOD system increases, namely whenever a new video request arrives, or a paused video resumes. Fortunately, the algorithm is incremental in nature.
  - The second stage attempts to achieve these load-balancing goals. Much of the time, the scheduling 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 the greedy approach alone may degrade relative to the optimal goal. When the quality of the DSG load-balancing differs from the goal by more than a predefined threshold, the *DASD dancing* algorithm is initiated. This algorithm is also graph-theoretic, and has the effect of shifting load from relatively overloaded to relatively underloaded DSGs.

A sample DASD dance will help to illustrate our technique. Consider a scenario in which the static component of the algorithm has assigned video A (*Dances with Wolves*) to DSGs 1 and 2, video B (*Yu Can't Cheat an Honest Man*) to DSGs 2 and 3, video C (*A Shachnai Phobia*) to DSGs 3 and 4, and a less popular video D (*Dirty Dancing*) to DSG 1 alone. Suppose a new request to play video D arrives. The greedy algorithm must schedule video D on DSG 1, thus increasing the load on that DSG by one. Suppose that this action overloads DSG 1 past the predefined threshold, relative to optimal. If DSG 4 is relatively 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 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.)

For simplicity, we will refer to our overall dynamic phase algorithm as DASD dancing, and our overall static-phase algorithm as CLLF.

In this paper, we shall examine the performance of our proposed disk-load-balancing algorithm via simulation experiments. In particular, we will show that DASD dancing combined with CLLF works well at more or less any degree of striping, and does better than striping alone. In many scenarios our scheme performs load-balancing equally well at lower striping degrees, albeit at a cost of additional dancing. We will also compare DASD dancing / CLLF with a variety of other load balancing schemes. For example, we consider a dynamic phase scheme which employs only a greedy scheduling algorithm in the dynamic component. (This can be regarded as a trivial special case of DASD dancing in which the threshold is set to infinity, and is certainly a reasonable strategy in its own right.) Similarly we consider a static phase scheme employing LLF alone, instead of the more elaborate CLLF. Our experiments show that DASD dancing / CLLF does better than either of three possible variant schemes, namely DASD dancing / LLF, Greedy / CLLF and Greedy / LLF. (These variants are in order of decreasing performance.) Finally, we consider a baseline scheme which assigns only single DSG copies of each video, so that the real-time scheduling algorithm becomes trivial. The static component is LLF. DASD dancing with CLLF performs dramatically better than this approach.

We briefly comment on some related work to support other aspects of VOD. Significant results were presented in [15] regarding admission control techniques and the choice of service size to support multimedia applications. The issue of collocational storage of multiple media segments on a disk is examined in [16]. In [23], a new formulation for disk-arm-scheduling schemes called grouped sweeping scheduling is proposed and analyzed. The goal is to minimize the buffer requirement. A similar concept is also considered in [11]. Furthermore, [19] studies stor-

age management and disk access algorithms in a disk array environment using this grouping approach. In [4], a cost/performance analysis of a video server with hierarchical storage is presented. Batching policies are considered in [2], while adaptive piggybacking schemes are discussed in [1, 12].

The remainder of this paper is organized as follows: In Sect. 2 we present the dynamic component of our algorithm, and in Sect. 3 we present the static component. We order the sections in this way because the load-balancing technique motivates the approach we take for the video-to-DSG assignments. In Sect. 4 we present the results of our simulation experiments. Section 5 contains conclusions, including some discussion of a VOD configuration planning problem, essentially dual to the load-balancing problem which is our focus.

2 Dynamic load balancing scheme

2.1 Preliminaries

In this section we describe the dynamic disk-load-balancing scheme called DASD dancing. The algorithm assumes the DSG assignments of videos as given. It then reacts dynamically to fluctuating video play demands, making decisions on which DSGs should handle the streams of each new or resumed video, as well as possibly changing decisions on which DSGs should handle currently playing videos. This transfering is accomplished via a baton-passing synchronization primitive, and gives the algorithm its name.

First, we fix some notation. (For the reader's convenience, we summarize the key notation employed in this paper in Table 1.) Let M denote the number of videos, and D denote the number of DSGs. (If we let $\mathscr{S}$ denote the degree of striping employed, then $\mathscr{S}D$ is the number of actual disks.) Let $\mathscr{A} = (a_{i,j})$ denote the assignments of video copies to DSGs. We make the reasonable assumption that any DSG will have at most one copy of any one video. Thus, $\mathscr{A}$ 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 $F_j(0) = 0$. (The function $F_j(x) = x/(L_j(L_j + 1 - x))$ is one such. A function which measures disk access times based on load would also be an appropriate choice.) 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 $a_{i,j} = 0$. (One cannot play a video from a DSG which has no copy.) We let $\Lambda = \sum_{i=1}^{M} \lambda_i$ denote the total number of all video streams in progress.

Table 1. Glossary of key DASD dancing variables

Variable	Section	Definition
M	2	Number of distinct videos stored on disk
D	2	Number of DSGs
$\mathscr{S}$	2	Degree of disk striping
$\mathscr{A}$	2	Video / DSG assignment matrix
L_j	2	Maximum stream capacity for DSG j
F_j	2	Penalty function for DSG j
$\lambda_{i,j}$	2	Actual number of video i streams playing on DSG j
λ_i	2	Total number of video i streams playing
Λ	2	Total number of videos streams playing
$x_{i,j}$	2	Optimal number of video i streams playing on DSG j
X_j	2	Optimal load on DSG j
D_1	2	Number of relatively overloaded DSGs
D_2	2	Number of relatively underloaded DSGs
B	2	Badness norm
T	2	Badness threshold
G	2	Directed DASD dancing graph
ϕ_i	3	Forecast demand for video i
Φ	3	Total forecast demand
K	3	Total allowable number of video copies
K_i	3	Number of copies of video i
H	3	Undirected DASD dancing graph
S_j	3	Storage capacity of DSG j
s_i	3	Size of video i
U	3	Maximal number of new video copies allowed
θ	4	Zipf-like distribution skew parameter
κ	4	Correlation coefficient

2.2 Optimal load balancing

The disk loads can be regarded as optimally balanced given the current load and video-to-DSG assignments when the objective function

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

is minimized subject to the constraints

$$x_{i,j} \in \{0,\ldots,L_j\},$$
$$\sum_{j=1}^{D} x_{i,j} = \lambda_i,$$
$$x_{i,j} = 0 \quad \text{if } a_{i,j} = 0.$$

Here, $x_{i,j}$ is a decision variable representing the hypothetical number of streams of video i which might be playing on DSG j. The first constraint limits the acceptable values of $x_{i,j}$. The second constraint ensures that the total number of video i streams equals the actual number of such streams in progress. The third constraint ensures that the video-to-DSG assignments are respected. Note that, for the optimal solution, $X_j = \sum_{i=1}^{M} x_{i,j}$ represents the desired load on DSG j. Our ultimate goal in DASD dancing will be to ensure that the optimal load X_j and the actual load $\sum_{i=1}^{M} \lambda_{i,j}$ are always close to each other for each DSG j.

Now the optimization problem described above is a special case of the so-called *discrete class-constrained resource allocation problem*. (The classes here correspond to

the videos. The problem is discrete because of the first constraint, a resource allocation problem because of the second, and class-constrained because of the third.) As shown independently in [6, 18], discrete class constrained resource allocation problems can be solved exactly and efficiently using a graph-theoretic optimization algorithm.

We now present an overview of the algorithm in [18] as it applies to the special case above. There are actually two reasons to do so. First, of course, the algorithm will be called as part of the dynamic component scheme, in order to set the target DSG loads. But second, the graph technique of the original algorithm is mimicked in the next component of the dynamic component scheme, namely in the DASD dancing algorithm itself. 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 $1, ..., D$, plus a dummy node, which we label as node 0. We also create and modify a *partial* feasible solution $\{x_{i,j} | i = 1, ...M, j = 0, ..., D\}$. Initially, this partial feasible solution is set for each i to have $x_{i,0} = \lambda_i$, and $x_{i,j} = 0$ for all $j = 1, ..., D$. Thus, all resources reside at the dummy node. The directed graph at any step has a directed arc from a node $j_1 \in \{0, ..., D\}$ to a node $j_2 \in \{1, ..., D\}$ 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 $j \in \{1, ..., D\}$ for which there is a directed path from 0 to j, the node for which the first difference

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

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 discrete class-constrained resource allocation problem. Feasibility is guaranteed because of the conditions on the arcs in the directed graph. The complexity of this algorithm is $O(D(\Lambda D + D^2 + \Lambda M))$. See [18] for further details.

2.3 DASD dancing

Given these optimal DSG loads, we are now ready to discuss the real-time scheduling algorithm itself. Clearly, stream demands are increased by one when a customer starts a new video or resumes a currently paused video. (Similarly, stream demands are decreased by one when a customer finishes a video or pauses a currently playing video; the scheduling algorithm does not react directly to these, however, since reductions in stream demand will not, by themselves, result in disk overloading.) Normally, handling demand 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

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

is minimal is chosen. However, periodically this approach may degrade. To check this, we solve the discrete class-constrained resource allocation problem of the previous subsection to obtain optimal DSG loadings given the current video demands. 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. The middle $D - D_1 - D_2$ DSGs must therefore be optimally loaded.) 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 component algorithm will be initiated. (Examples of reasonable norms include the value $\sum_{i=1}^{M} \lambda_{i,1} - X_1$ of the relatively most overloaded DSG, the sum $\sum_{j=1}^{D_1}(\sum_{i=1}^{M} \lambda_{i,j} - X_j)$ of all the relatively overloaded DSGs, the sum of squares $\sum_{j=1}^{D}(\sum_{i=1}^{M} \lambda_{i,j} - X_j)^2$, and so on. For simplicity we choose the first of these in our implementation. 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:

Procedure : **MAIN**
  While $B > T$
    Call DANCE
    If DANCE returns (0) then stop
    Perform bookkeeping on B, G, D_1, D_2
  End
End MAIN

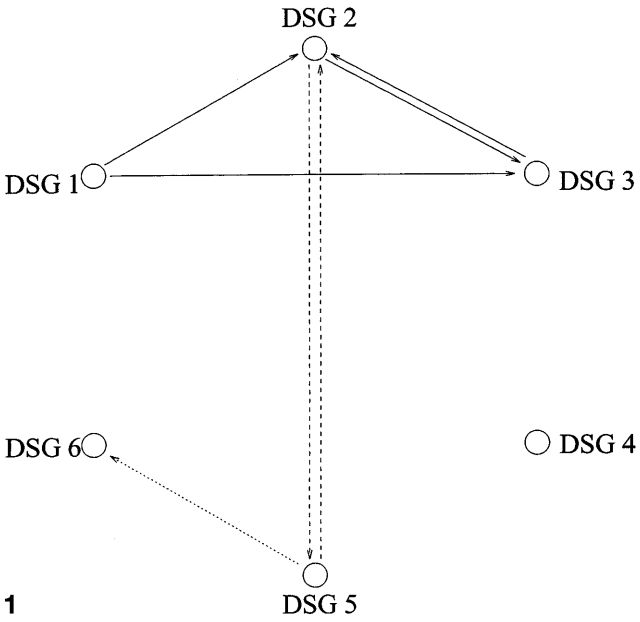
Procedure : **DANCE**
  Do for j_1 = 1 to D_1
    Do for $j_2 = D$ to $D - D_2 + 1$ by -1
      If there exists a directed path in G from j_1 to j_2 then transfer videos along the shortest such directed path using baton passing and return (1)
    End
  End
  Return (0)
End DANCE

The DASD dance corresponds to a successful call of the DANCE routine (a call which returns the value 1). The while loop in the MAIN routine continues to call the DANCE routine until either the threshold is no longer exceeded, or the call is not successful (no DASD dance is identified, and the value 0 is returned). The order of the nested do loops in the DANCE routine causes shifts of load from the most relatively overloaded DSGs to the most relatively underloaded DSGs to occur as early as possible. Note that the requirement to proceed along a shortest directed path keeps the dance lengths as small as possible, and 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 the details are not important to the main thrust of the paper. For a directed path of length greater than 1, the dance should be performed in forward order – from the first arc through the last: This will fix the overloading problem as quickly as possible, without overloading the other DSGs even temporarily. The DASD dancing scheme will have the effect of balancing the load to a larger degree than would be possible without transfering videos dynamically.

An example will help illustrate the DASD dancing algorithm. Consider a six DSG configuration supporting four videos, with video-to-DSG assignments listed in Table 2. Suppose the following scenario: DSG 1 is loaded to capacity with video streams, and is thus relatively overloaded. DSG 6 is relatively underloaded. DSGs 1, 2 and 3 are each playing *The Thin Man*. DSGs 2 and 5 are playing *Dashing Through*. *Thick as Thieves* is not currently playing. *Polka Dot Puss* is playing on DSG 6, but not on DSG 5. Consider Fig. 1, which shows the directed graph G at this moment. There are *thin* directed arcs from DSG 1 to DSGs 2 and 3, and in both directions between DSGs 2 and 3 themselves. There are no thin arcs directed towards DSG 1, because DSG 1 is at capacity. There are *dashed* directed arcs in both directions between DSGs 2 and 5. There are no *thick* directed arcs in either direction between DSGs 3 and 4, because of the lack of current plays of that video. There is a *dotted* directed arc from DSG 5 to DSG 6, but none in the opposite direction. Under these conditions, a DASD dance from DSG 1 to 2 to 5 to 6 would occur, transferring the plays of 3 videos.

Table 2. Sample video to DSG assignments

Video Name	DSGs
The Thin Man	1,2,3
Dashing Through	2,5
Thick as Thieves	3,4
Polka Dot Puss	5,6



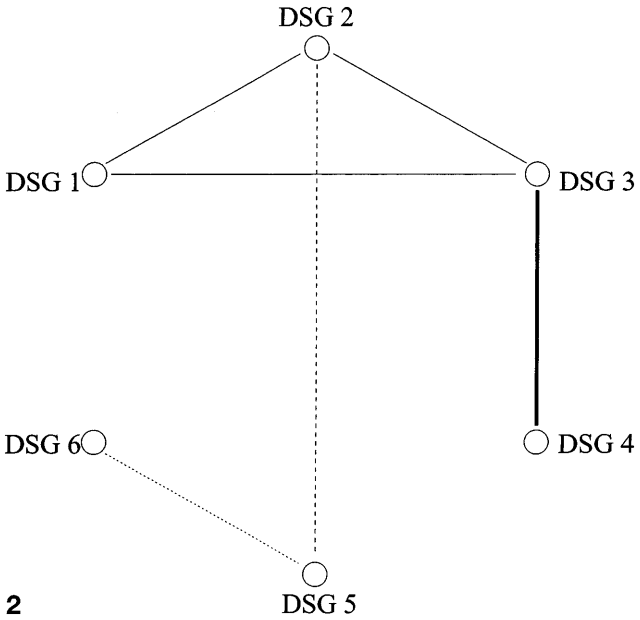


Fig. 1. The directed graph G

Fig. 2. The graph H

The shortest directed path between DSGs 1 and 6 happens to also be the only such directed path. As a result of this dance, the overall disk load-balancing would be improved.

2.4 Algorithmic shortcuts

We should observe that successive calls to find optimally balanced DSG loadings (Sect. 2.2) will involve substantially similar problem instances. Fortunately, there exist natural incremental variants of the described solution technique, so that the computational complexity can be kept within reason. Actually, it is also possible to shorten the execution time of the dynamic component scheme by replacing the calls to the discrete class-constrained resource allocation problem algorithm with calls to a simpler algorithm, at the expense of some accuracy in setting the DSG optimal loading goals. Specifically, consider the corresponding resource allocation problem in which the classes have been removed (or more precisely, conglomerated into a single class). Thus, we wish to minimize the objective function

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

subject to the constraints

$$x_j \in \{0, \ldots, L_j\},$$
$$\sum_{j=1}^{D} x_j = \Lambda.$$

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, since we have relaxed one constraint. Of course, relaxing this constraint also means that the solution obtained may not be actually implementable. But if the video-to-DSG assignment algorithm described in the next section is done well, the optimistic assumption that the values x_j and X_j will be close is generally justified. Thus, we can use each value x_j as a surrogate for X_j, even though it may not correspond exactly to a truly feasible solution. The point is that this new optimization problem is solvable by a fast algorithm [7] 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 even faster algorithms [8, 9] for this resource allocation problem, but they are not incremental in nature. See also [5] for further details.)

If all the disks are homogeneous in the sense that they have identical performance, we reduce complexity further still. In this case, we can drop the subscripts and define $L = L_j$ and $F = F_j$ for each DSG j, and then the resource allocation problem solves trivially (modulo integrality considerations), with each $x_j = \Lambda/D$. We will make this assumption and adopt this shortcut in our experimental section.

3 Efficient static assignment of videos to DSGs

3.1 Preliminaries

In this section, we describe the static scheme which assigns videos to DSGs, the goal being to facilitate the job of the real-time scheduler of Sect. 2. Using the notation of that section, the output of this stage is simply the $\{0, 1\}$ assignment matrix $\mathscr{A} = (a_{i,j})$. The static algorithm proceeds in two stages.

In the first stage, we decide how many copies of each video to create, given the forecast demands for each video and the total number of video copies allowed in the system. Thus, we determine the row sums of the matrix $\mathscr{A}$.

Specifically, assume we have indexed the M videos in terms of decreasing forecast demand ϕ_i. Therefore $\phi_1 \geq \ldots ... \geq \phi_M$. (The issue of providing good-quality forecasts is orthogonal to the main thrust of the paper. We assume these forecasts as given.) Let K denote the acceptable number of video DSG copies in the system. The goal is to compute the number $K_i \geq 1$ of required copies for each video i. Making each K_i roughly proportional to ϕ_i with the constraint that

$$K = \sum_{i=1}^{M} K_i$$

is a resource allocation problem known as the *apportionment problem*. The problem arises natually in the context of government representation. In the 435-member United States House of Representatives, for example, it is necessary to choose the size of the congressional delegation from each of the 50 states in proportion to its population. New York State has 31 representatives, for instance. Many schemes have been proposed for this apportionment problem, and we adopt *Webster's monotone divisor method*. (Alternative schemes are due, for example, to Hamilton, Adams and Jefferson, all figures from the American revolution.) We describe Webster's method in Sect. 3.2. Other details may be found in [5].

Next, we assign the numbers of video copies computed in the first stage to DSGs. This second stage has two possible modes. The "initial" mode is used to configure a new system from scratch, one for which no videos have yet been assigned to DSGs. This mode is described in Sect. 3.3. The "incremental" mode, described in Sect. 3.4, is used on a periodic basis to adjust existing video-to-DSG assignments based on revised video demand forecasts. In order to ensure that implementation of these adjustments is practical, we allow for a constraint on the number of new video copies which can be assigned to DSGs.

The primary goal in both modes is to achieve high connectivity of the undirected graph H defined as follows. The nodes correspond to the DSGs. For each pair j_1 and j_2 of distinct nodes, there is an arc between j_1 and j_2, provided there exists at least one video i_1 for which

$$a_{i_1,j_1} = a_{i_1,j_2} = 1 \ .$$

This condition mimics condition (1) in the definition for the directed graph G given in Sect. 2.3. The notion is that H serves as an effective surrogate for G, since a good dynamic component scheme will typically ensure that conditions (2) and (3) in the definition of G will be satisfied whenever (1) is. We attempt to increase connectivity by minimizing the *diameter* of the graph H, which is the maximum distance between any pair of nodes. Figure 2 shows the graph H for the example described in Sect. 2.3. Note the similarities to

the directed graph G shown in Fig. 1. There is, however, a *thick* arc in Fig. 2 between DSGs 3 and 4, even though Fig. 1 has no comparable directed arc.

A secondary goal will be to keep the forecast load on each DSG roughly proportional to its maximum stream capacity. To this end, approximate the per-copy forecast load on each video i by

$$\delta_i = \frac{\phi_i}{K_i}.$$

We will attempt to equalize the values of

$$\frac{\sum_{i=1}^{M} a_{i,j}\delta_i}{L_j}$$

for each DSG j.

Finally, we will need to ensure that the storage capacity of each DSG is not exceeded. Denote by S_j the storage capacity of DSG j, and let s_i denote the size of video i (in the same units). Then the storage capacity test amounts to checking that

$$\sum_{i=1}^{M} a_{i,j}s_i \leq S_j$$

for each DSG j.

3.2 Apportionment problem

Define $\Phi = \sum_{i=1}^{M} \phi_i$ to be the sum of all the video demand forecasts. We wish to keep K_i nearly proportional to ϕ_i for each video i, and there are $K = \sum_{i=1}^{M} K_i$ videos in all. Thus, we want to maintain the proportions $K_i/\phi_i \approx K/\Phi$. Said differently, in our context, the goal in solving the apportionment problem is to give each video i a number of DSG copies K_i as close as possible to its *quota* $K\phi_i/\Phi$, while creating $K = \sum_{i=1}^{M} K_i$ DSG copies altogether. Unfortunately, the quota for any given video need not be integral, and K_i must be. There are rounding algorithms which ensure that K_i always falls between the floor and ceiling of its quota. For example, Hamilton's scheme is one of these. But it turns out that there are interesting and subtle paradoxes which any such scheme must suffer. The Webster monotone divisor method avoids these paradoxes, and may be described as follows. Initialize each $K_i = 1$. The general step is to find the video i^* for which $\phi_i/(K_i + .5)$ is maximized, and increment K_{i^*} by 1. If $K = \sum_{i=1}^{M} K_i$, stop. Otherwise repeat the general step. Webster's method is basically a greedy algorithm, and the denominator $K_i + 0.5$ is one of several possible so-called *divisors* which can be used as criteria in selecting the next video to increment. (There are divisors due to Hamilton and Adams as well.) An excellent description of these algorithms and the various paradoxes which they avoid appears in [5].

3.3 Initial assignment algorithm

In the initial mode we achieve high connectivity of the graph H from scratch in three steps. We first produce a *clique tree* and initial graph, then use the greedy *ADD* scheme to reduce the diameter of the graph, and finally assign single copies, using a scheme we call *LLF*. The first two schemes will handle *multicopy* (MC) videos, in other words videos j for which $K_j > 1$. The last scheme will handle the remaining *single-copy* (SC) videos. We describe each of these steps in turn.

Each node of the *clique tree* will consist of a set of DSGs such that each has a copy of a common video. (These are the *cliques*.) Each DSG will be contained in at most one node in the tree. However, aside from the root node, one copy of this common video will also be placed on another preexisting node in the tree, and an edge will thereby be created. More specifically, we proceed as follows. All copies of the most popular MC video are first assigned to different DSGs and these DSGs form the root node of the clique tree. We then consider the next hottest MC video. Each copy except one is assigned different empty DSGs, and these DSGs form a *branch* node in the clique tree. The remaining copy is assigned to the relatively least loaded DSG with sufficient storage capacity in the root node. This copy is called the *connection copy*, because it creates an edge in the clique tree. We continue in this manner to create new clique tree nodes, assigning the connection copy to the node closest to the root which has a DSG with sufficient storage capacity. Within that node, the relatively least loaded DSG with sufficient storage capacity is chosen. The process is repeated until we run out of empty DSGs or we run out of MC videos. The leaf nodes represent less popular videos with replications. During the final step of the tree building, the video may have more copies (excluding the connection copy) than the number of DSGs remaining unassigned. The assignment of those excess copies will be addressed in the next step.

The *ADD* algorithm handles the remaining MC videos in two stages. Each stage is greedy in nature. For each MC video with copies left to be assigned, we can pair off all but at most one copy. We treat the paired copies in the first stage, in order of decreasing popularity. In the second stage, we treat the remaining single copies in the same order.

(i) For each *pair* of remaining video copies, we choose from the set of DSGs with sufficient capacity that pair (j_1, j_2) with maximal distance. Ties are adjudicated by picking a pair which is the *least connected* in the following sense. Let $M(j_i)$ denote the number of distinct copies of MC videos assigned to DSG j_i. Then the connectivity of the pair (j_1, j_2) is defined as

$$C(j_1, j_2) = M(j_1) + M(j_2).$$

Intuitively, we measure the connectivity of the graph by counting the number of potential DASD dancing paths originating from a given DSG j_i.

(ii) Any remaining *single* copy of an MC video is placed on a DSG so as to maximally decrease the diameter of the graph. Ties are adjudicated in a manner similar to that described in (i).

The remaining SC videos have no effect on the diameter of H. We assign them to balance the load only. The scheme is greedy, and called *Least Loaded First* (LLF). In other words, we assign the next single copy of a video to that

Table 3. Sample video forecast demand and number of copies

Video ID	Name	Forecast (%)	Copies
A	After the Dance	16	4
B	Bolero	13	3
C	Charleston	11	3
D	Dance Fever	10	3
E	Emperor Waltz	9	2
F	Flashdance	9	2
G	Go Into Your Dance	9	2
H	Hustle	8	2
I	I am a Dancer	8	2
J	Jo Jo Dancer, Your Life is Calling	4	1
K	Kickin' the Conga 'Round	3	1

DSG with sufficient capacity for which the ratio of forecast load to maximum stream capacity is minimal.

We now illustrate the above construction by an example. We first produce a clique tree and initial graph, then use the greedy ADD scheme to reduce the diameter of the graph, and finally assign single copies using the LLF scheme. Assume a system of $D = 13$ DSGs and $M = 11$ videos. For simplicity, assume all videos are of equal size, and that the DSGs themselves are identical, with a storage capacity of 2 videos per DSG and a stream capacity of 10. Assume that the allowable number of copies is $K = 25$. The forecast demands and numbers of required copies of each video are presented in Table 3. (Note that the forecasts are normalized, and the number of copies are computed using Webster's monotone divisor method.)

The ultimate assignment of videos to DSGs is given by the matrix

$$\mathscr{A} = \begin{pmatrix} 1&1&1&1&0&0&0&0&0&0&0&0&0 \\ 1&0&0&0&1&1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&1&1&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&1&1&0&0&0 \\ 0&0&0&1&0&0&0&0&0&0&1&0&0 \\ 0&0&0&0&1&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&1&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1&0&0&0&0&1&0 \\ 0&0&0&0&0&0&0&1&0&0&0&1 \\ 0&0&0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&0&0&1&0&0&0&0 \end{pmatrix}.$$

The graph H obtained after the greedy phase is illustrated in the top portion of Fig. 3. The four copies of the most popular video A are assigned to DSGs 1, 2, 3 and 4. These four DSGs form the root node of the clique tree. The next most popular video is B, which has 3 copies. One of these is assigned to one of the DSGs in the root node, DSG 1, as the connection copy of the video. The other two copies of the video B are assigned to DSGs 5 and 6, respectively. These two DSGs form a branch node of the clique tree. Video C also has three copies, one of which is assigned to DSG 2 as a connection copy to the branch node consisting of DSGs 7 and 8. Similarly, there are three copies of video D, one of which is assigned to DSG 3 as a connection copy to the branch node consisting of DSGs 9 and 10. There are two copies of video E. The connection copy is assigned to DSG 4, while the other is assigned to branch node DSG 11. At this point in the assignment process, the storage capacities of DSGs in the root node have been reached, and thus it is no longer possible to assign connection copies of the remaining videos to the root node. Therefore, one copy of video F is assigned to DSG 5 and another is assigned to DSG 12. Similarly, one copy of video G is assigned to DSG 6 and another to DSG 13. The clique tree is now complete.

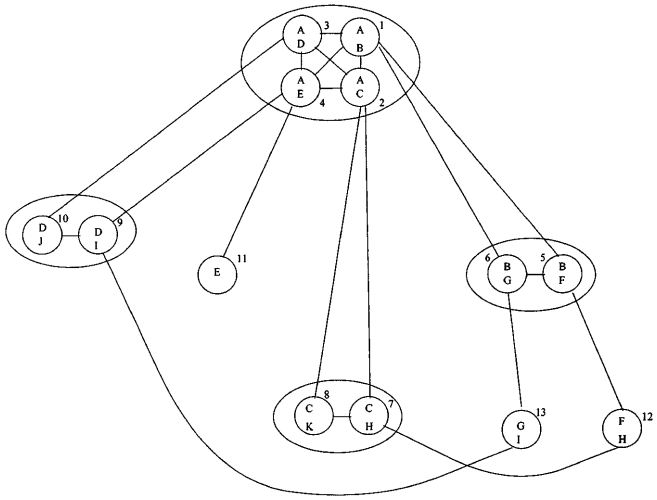


Fig. 3. Initial assignment of videos to DSGs

Now the ADD phase begins. The two copies of video H are assigned to DSGs 7 and 12. We note that these DSGs belong to different nodes in the clique tree. The assignments reduce the graph diameter, previously determined by the distance between these DSGs. The two copies of video I are assigned to DSGs 9 and 13. There are no SC MC videos to deal with, so the ADD phase is now complete. The graph H at this stage is illustrated in Fig. 3. (Again, the clique tree can also be seen in the top of this figure; the effect of the ADD phase is shown in the bottom.)

Finally, SC videos J and K are assigned to DSGs 10 and 8, repectively, by LLF.

3.4 Incremental assignment algorithm

The incremental algorithm will be used to retain high connectivity in the graph H on a day-by-day or week-by-week basis. The idea is that periodic revisions to the video demand forecasts will, in turn, cause changes in the solution to the apportionment problem. Thus, some videos will need to lose DSG copies, some will gain DSG copies, and some of the existing video copies may be moved from one DSG to another. The three steps of the algorithm are performed in this order. We keep track of the net number of new video copies on DSGs, which is forced to be bounded by some fixed threshold U. This threshold is designed to avoid excessive revisions to the video-to-DSG assignments. This, in turn, is intended to make the incremental mode scheme practical from the standpoint of implementation.

In step 1, we greedily remove DSG copies from net-loss videos using a scheme we call *DELETE*. In other words, we always remove that video copy which increases the diameter of the graph H least. Again there is a scheme for adjudicating ties, which we omit. In step 2, we greedily add DSG copies to net-gain videos, using the algorithm ADD. We then compute the number of new video copies. If this number is

greater than or equal to U, we stop. Otherwise, we perform step 3, a *neighborhood escape heuristic* [10] on the entire set of video copies. (Briefly, a neighborhood escape heuristic is an iterative improvement scheme which attempts to avoid being trapped in local minima, while achieving low computational costs. Assuming a predefined metric on the search space of feasible solutions, plus an existing initial feasible solution, the algorithm successively searches the neighborhoods of distance 1, 2, and, so on about that solution. If no improvement is found within a predetermined number of neighborhoods, the algorithm stops. Otherwise, the improved solution is adopted as the new solution, and the process repeats.) For our problem, one can impose a natural metric in which the distance between two sets of video-to-DSG assignments $\mathscr{A}_1$ and $\mathscr{A}_2$ is the number of matrix entries on which they are not equal. Thus, modifying an assignment by moving a single video copy results in a new assignment a distance of 1 away. Swaps of video copies are of distance 2, and so on. The feasibility constraints are on load-balancing and DSG storage capacity. If either

1. no further improvements to the diameter of H or its connectivity are possible, or
2. the number of moves would exceed U,

we stop. In the latter case, we abort the final move decision. We call this scheme ESCAPE. Details on a similar neighborhood escape heuristic may be found in [20].

4 Experimental results

4.1 Methodology

In this section we describe the simulation experiments designed to test the performance of the DASD dancing algorithm. First we list some of the key parameters used. We assume a total of M = 200 videos. Each disk has enough physical capacity to store 3 videos. We look at cases of 2-way, 4-way and 8-way striping. In other words, $\mathscr{S}$ varies between 2, 4 and 8. We examine configurations with 72, 80, 88 and 96 total disks. For example, in the case of 8-way striping, a 72-disk configuration means that the number of DSGs is D = 9. (Given 200 videos at 3 videos per disk, there is not enough physical capacity when D = 8. But at D = 9 there is enough spare capacity for 16 additional video copies.) We assume that the disks are identical in performance, with a maximum number of L = 10 concurrent video streams per disk. So the physical capacity of an 8-way DSG is 24 videos, while the maximum number of concurrent video streams is 80. We employ the shortcut resource allocation problem described in Sect. 2.4 to compute our load-balancing goals. As noted, this problem can be solved trivially, because the disks are homogeneous.

Next, we describe the structure of the simulation experiments themselves. We choose hour-of-day arrival patterns according to a Zipf-like distribution with N = 24 and θ = 0.3. (Briefly, a Zipf-like distribution [13, 24] takes two parameters, N and θ, the latter corresponding to the degree of skew. The distribution is given by $p_i = c/i^{1-\theta}$ for each $i \in \{1, ..., N\}$, where $c = 1/[\sum_{i=1}^{N} 1/i^{1-\theta}]$ is a normalization constant. Setting θ = 0 corresponds to a pure Zipf distribution, which is highly skewed. Setting θ = 1 corresponds to a uniform distribution.) The most busy hour is fixed to be 9 pm, followed by 10 pm, 11 pm, 8 pm, and, so on down to the least busy hour, which is fixed to be 4 am. Multiplying a variable overall daily throughput against this distribution determines the arrival rate of videos per hour. The relative popularity of each of the M = 200 videos varies hour by hour, and is chosen as the weighted average of four reasonably well-correlated Zipf-like distributions with $N = M$ and randomly chosen values of θ. (We model the *correlation* between two distributions by using a single parameter κ that can take on any integer value between 1 and M. First, the most popular video in distribution 1 is made to correspond to the r_1th most popular video in distribution 2, where r_1 is chosen randomly from between 1 and κ. Then, the second most popular video in distribution 1 is made to correspond to the r_2th most popular video in distribution 2, where r_2 is chosen randomly from between 1 and $min(M, \kappa+1)$, except that r_1 is not allowed, and so on. Thus, κ = 1 corresponds to perfect correlation, and $\kappa = M$ to the random case. In our case, we choose random values for κ which are less than or equal to 10. See [21] for details on a similar correlation scheme.) These four distributions are meant to correspond to the hours of 3 am, 9 am, 3 pm and 9 pm, respectively. The weightings for any other given hour are chosen to be inversely proportional to the time difference between that hour and these four. The idea is to provide different customer mixes throughout the day. Videos last 1.5 h, and are paused and resumed an average of once each. We evaluated our schemes on a prototypical simulated day in each of seven successive weeks, under the assumption that forecasts would be revised on a weekly basis. On each subsequent week, 5 new videos were added into the mix in randomly chosen positions, the positions of the existing videos were varied based on a correlation scheme with κ = 10 similar to the above, and 5 old videos (those whose positions would now be beyond M = 200) were removed. The demand forecast for each video in a given week is chosen randomly from a truncated normal distribution whose mean is the actual daily activity rate of that video. This is intended to model inacurracy in the forecasting process. While our simulations could not be exhaustive, they were extensive. (Experiments in which many of the above parameters were varied yielded results similar to those reported on. Examples include the number of videos, the physical and stream capacities of the disks, and the distribution of videos throughout the day.) We also believe that the simulation framework is realistic and robust enough to test the DASD dancing scheme with some level of confidence.

For example, Fig. 4 shows actual video grosses on six successive weekends in May and June of 1995 [26]. The relative gross dollar amounts should presumably approximately mimic video demand, though in a VOD environment the skew may be even more pronounced: Theatre customers cannot attend a showing of a hot movie if there are no remaining seats. Notice, in any case, that each week has a roughly Zipf-like distribution, with varying degrees of skew. Notice also how the hot videos fare week-by-week. For example, the hottest video of week 1 assumes positions 1, 3, 4, 6 and 7 in weeks 2 through 6, respectively.

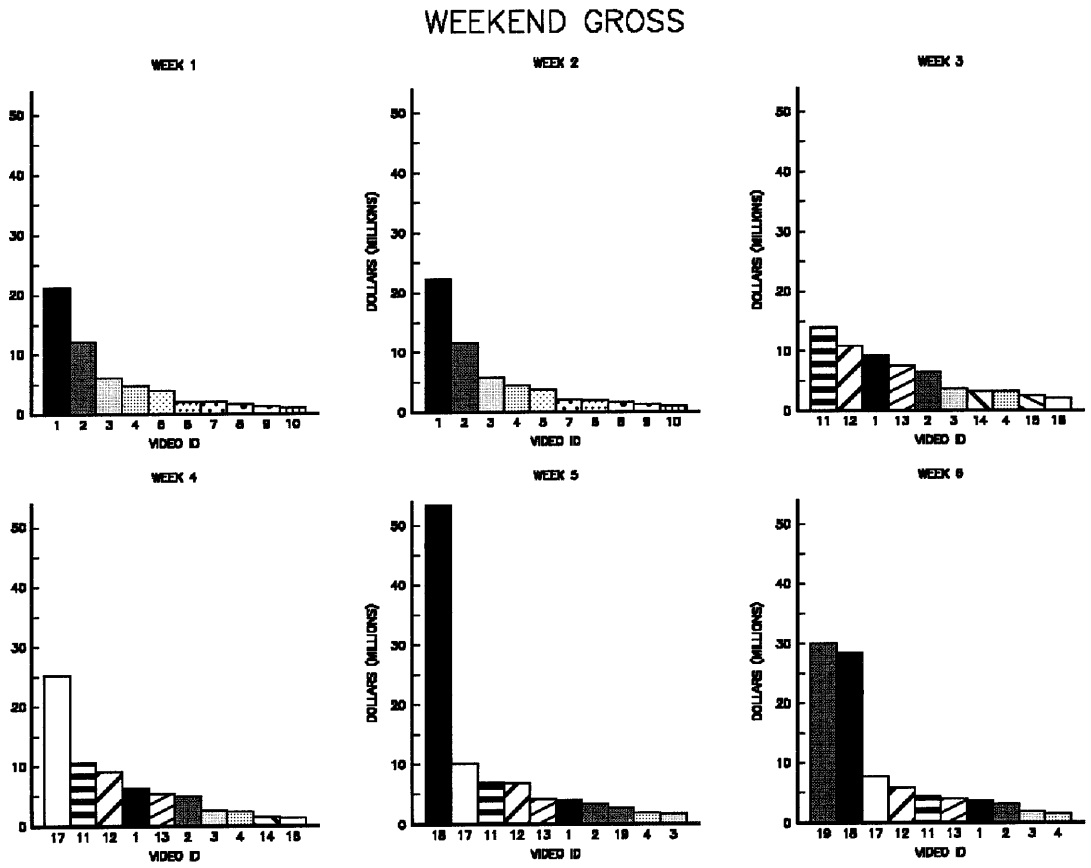
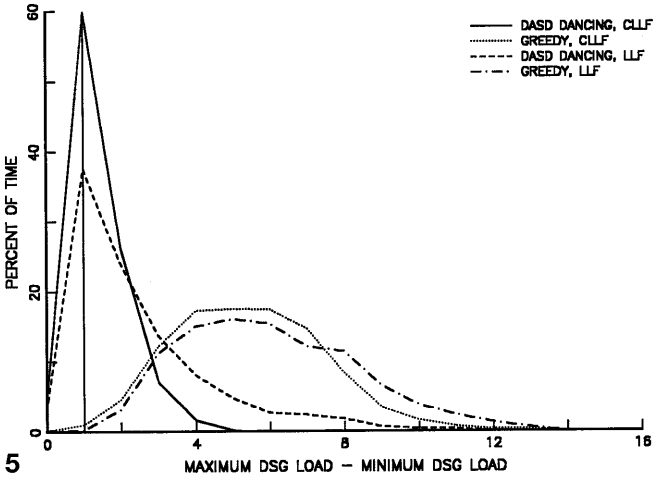


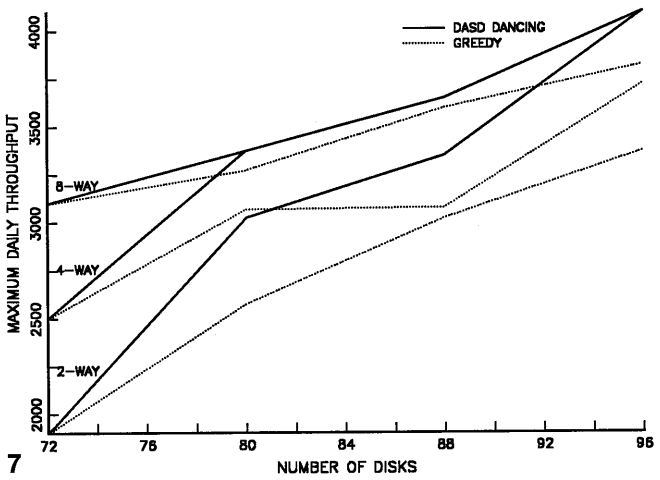
Fig. 4. Weekend video grosses

4.2 Results

We first describe a test comparing combinations of two dynamic strategies with two *initial* static strategies. The dynamic strategies are, of course, greedy and DASD dancing. (Remember that greedy is a special case of DASD dancing in which the threshold has been set to infinity.) The static strategies are CLLF and LLF alone. There are thus four load-balancing strategies overall. We choose the $D = 9$ case with 8-way striping. As noted, this means that there are 16 additional videos available to reduce the diameter of a nine node graph. We choose a single day for comparison, and study the distribution of time throughout this 24-h period of the maximum load per DSG minus the minimum load per DSG: If the system is properly load-balanced, the maximum and minimum DSG loads should be nearly identical, so that this distribution would be concentrated near 0. The results are shown in Fig. 5. Actually, a difference of 0 or 1 should be regarded as perfect load-balancing, based on integrality considerations. (We have emphasized this by drawing a vertical line in the figure at $x = 1$.) So we can see that the DASD dancing algorithm with CLLF does very well, producing a tight distribution, for which the difference between the maximally loaded DSG and minimally loaded DSG is never more than 5, and achieves 5 less than 0.2% of the day. DASD dancing with LLF alone does well also, but noticeably worse. Both of the greedy versions fare noticeably more poorly. For example, for greedy with CLLF, there are four times in the day when the difference reaches 16. (In fairness, the difference is more than 11 only 1% of the day.) As with DASD dancing, the performance of greedy with LLF is slightly worse than with CLLF. Returning to the DASD dancing algorithm with CLLF, there were 1343 dances of length 1, 386 dances of length 2, and 1 dance of length 3. This amounts to approximately 72 dances per hour, or just over one per minute. Given that the daily video throughput in this experiment was 3000, with a similar number of pause resumes, the number of dances does not appear to be very significant. We note that the badness threshold in these example was set very low. This has the cosmetic advantage of providing a very tight distribution in the figure. It turns out that setting it higher appears to still provide effective disk load-balancing while cutting the number of dances by a factor of nearly two.

Assuming now that CLLF is the right initial static component algorithm, we experiment with the performance of the $D = 9$, 8-way striping example over the course of 7 weeks. This allows us to test the *incremental* portion of the static component algorithm and see if performance degrades. The results of the simulation appear in Fig. 6. The figure shows the distribution of the maximally loaded DSG minus the minimally loaded DSG for both DASD dancing and greedy. Week 0 is the initial week, and weeks 1 through 6 are incremental. There does not appear to be any significant degradation in either dynamic scheme. The shapes of the distributions for DASD dancing are in fact quite similar, as are those for greedy.





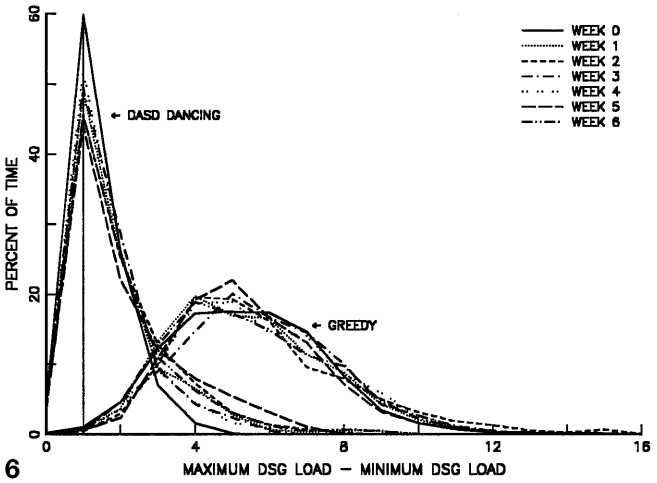


Fig. 5. Load imbalance distributions for different schemes

Fig. 6. Load imbalance distributions for different days

Fig. 7. Maximal daily throughputs for different configurations

Finally, we consider the maximum daily video throughput achievable by DASD dancing and greedy, given the stream capacities of the disks. From the standpoint of the VOD provider the throughput is perhaps the most important metric. However, by the design of our simulation experiments a maximum of approximately 23% of all daily videos will be playing concurrently at various times between 10 pm and 11 pm. Thus, maximum daily throughput effectively translates into maximum concurrent video streams. So these results can be understood in terms of the latter metric as well. We consider cases of 2-way, 4-way and 8-way striping, and scenarios with 72, 80, 88 and 96 actual disks. The maximal daily throughput was determined by combining our simulation code with a *bracket and bisection* algorithm [14].

Figure 7 shows the results of this study. The DASD dancing algorithm exhibits nearly linear growth, at least as the DSGs start to become more highly connected. Observe the striping degree, which is noted on the left side of the figure. For 72 disks and 2-way striping, there are 16 video copies available to connect 36 DSGs, so dancing will not help as much in this instance. Note also that as the number of actual disks grows, connecting the DSGs in low-striping scenarios occurs more slowly than in high-striping scenarios. This is because the number of DSGs is inversely proportional to the degree of striping. Nevertheless, by the 96-disk experiments the maximal throughput of the DASD dancing algorithm is identical for all three striping scenarios. This comes at the price of more dancing, however. The distribution of the lengths of the dances is shown in Table 4. In all cases DASD dancing does better than greedy alone. One would expect the performance of these two algorithms to be relatively close in scenarios with a small number of disks (because there is too little connectivity for dancing to work well), and in scenarios with a large number of disks (because there is so much connectivity that greedy works well). One would expect DASD dancing to perform best relative to greedy for numbers of disks between these extremes. The figure shows many of these effects. It should also be noted in the figure that DASD dancing does better than greedy in all scenarios, but that the performance of the two schemes is closest for highly striped cases.

As a baseline scheme, we also simulated the trivial scheduling policy in which there is always a single copy of each video available, assigned via LLF. (Clique trees are irrelevant in this case, and the scheduling policy amounts to playing the video on the DSG to which it is assigned.) One would hope that the throughput would grow with the number of disks. But, for the SC algorithm, the maximum throughput is nearly flat in each striping example. Since striping assists load-balancing, 8-way striping does better than 4-way, which

Table 4. DASD dancing length distributions for different degrees of striping

Length	8-Way	4-Way	2-Way
1	1878	1661	981
2	42	554	1100
3	0	21	139
4	0	2	11
5	0	0	1

does better than 2-way. But, in each case, the hottest video must be played from a single DSG, and this immediately becomes the bottleneck. Thus, the maximal throughput achievable using 8-way striping is approximately 1850 videos per day, independent of the number of disks. For 4-way striping, the throughput is roughly 1100, and, for 2-way striping, the throughput is approximately 550.

On the other end of the spectrum, we can compute the largest possible daily throughput theoretically obtainable by *any* load-balancing scheme as follows. Given our simulation design, we have noted that the maximum number of simultaneous videos is about 23% of the entire daily throughput. Given also a maximum stream capacity of 10 concurrent videos per disk, we determine that a system with 72 disks can accomodate a daily throughput of at most 3130 videos if it is operating at absolutely full capacity during its busiest period. This number grows in proportion to the number of disks, so that a system with 96 disks can handle a throughput of 4174 videos. Note how closely DASD dancing with 8-way striping comes to achieving this theoretical maximum throughput across all of our disk configurations.

5 Conclusions

In this paper, we have devised a real-time disk-scheduling algorithm for VOD computer systems. The algorithm consists of a dynamic and a static scheme. The dynamic scheme schedules videos to DSGs in order to balance the load on the disks. Typically, it does this scheduling in a greedy fashion, but occasionally it may transfer several in-progress videos between successive pairs of DSGs in order to deal with degrading load balance. This “DASD dance” is achieved through a baton-passing primitive. The CLLF static scheme assigns videos to DSGs on a periodic basis, perhaps once per day or once per week. Its mission is to optimize the load-balancing achievable by the dynamic scheme. The techniques in both the static and dynamic schemes are graph-theoretic, and are based primarily on resource allocation problem optimization algorithms.

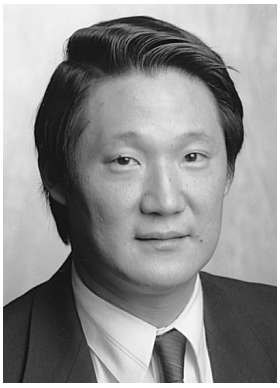
Based on our simulation results, DASD dancing with CLLF appears to be an effective load-balancing scheme. It works synergistically with disk striping, and outperforms the greedy scheduling policy alone in all examples tested. DASD dancing / CLLF appears to allow for video throughputs which grow in proportion to the number of disks.

The problem of *configuration planning* is in a sense dual to the load-balancing problem which has been our focus. In the latter, we wish to maximize the load we can handle in a fixed hardware configuration. In the former we wish to minimize the cost of the configuration, while handling a given video forecast demand. Thus, in principal, we can use our simulation code to solve VOD configuration planning problems as well. Given a suite of simulation tests, we explore the disk search space to find a VOD hardware configuration which passes the tests and has minimal cost.

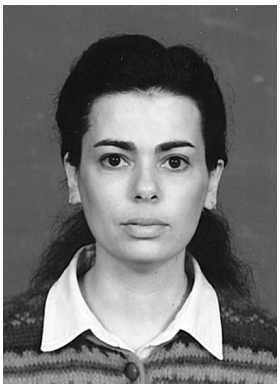
References

1. Aggarwal C, Wolf J, Yu P (1996) On Optimal Piggyback Merging Policies for Video-on-Demand Systems. ACM Sigmetrics Conference, Philadelphia Pa
2. Aggarwal C, Wolf J, Yu P (1996) On Optimal Batching Policies for Video-on-Demand Storage Servers. 3rd International Conference on Multimedia Computing and Systems, Hiroshima, Japan
3. Chen P, Lee E, Gibson G, Katz R, Patterson D (1994) RAID: high-performance, reliable secondary storage. ACM Comput Surv 26(2):145–185
4. Doganata Y, Tantawi A (1994) A Cost / Performance Study of Video Servers with Hierarchical Storage. 1st International Conference on Multimedia Computing and Systems, Boston Mass.
5. Ibaraki T, Katoh N (1988) Resource Allocation Problems - Algorithmic Approaches. MIT Press, Cambridge, Mass.
6. Federgruen A, Groenevelt H (1986) The greedy procedure for resource allocation problems: necessary and sufficient conditions for optimality. Oper Res 34:909–918
7. Fox B (1966) Discrete optimization via marginal analysis. Management Sci 13:210–216
8. Frederickson G, Johnson D (1982) The complexity of selection and ranking in X+Y and matrices with sorted columns. J Comput Syst Sci 24:197–208
9. Galil Z, Megiddo N (1981) A Fast Selection Algorithm and the Problem of Optimum Distribution of Efforts. J ACM 26:58–64
10. Garfinkel R, Nemhauser G (1972) Integer Programming, Wiley, New York
11. Gemmell D (1993) Multimedia Network File Servers: Multi-Channel Delay-Senstitive Data Retrieval. ACM Multimedia 93, Anaheim Calif.
12. Golubchik L, Lui J, Muntz R (1995) Reducing I/O Demand in Video-on-Demand Storage Servers. ACM Sigmetrics Conference, Ottawa, Canada
13. Knuth D (1973) The Art of Computer Programming, Volume 3: Sorting and Searching. Addison-Wesley, Reading, Mass.
14. Press W, Flannery B, Teukolsky S, Vetterling W (1986) Numerical Recipes. Cambridge University Press, Cambridge, UK
15. Rangan P, Vin H (1991) Designing File Systems for Digital Video and Audio. 12th ACM Symposium on Operating Systems
16. Rangan P, Vin H (1993) Efficient storage techniques for digital continuous media. IEEE Trans Knowl Data Eng 5(4):564–573
17. Sincoskie W (1991) System architecture for large scale video on demand. Comput Networks ISDN Syst 22:155–162
18. Tantawi A, Towsley D, Wolf J (1988) Optimal Allocation of Multiple Class Resources in Computer Systems. ACM Sigmetrics Conference, Santa Fe NM
19. Tobagi F, Pang J, Baird R, Gang M (1993) Streaming RAID – A Disk Array Management System for Video Files. ACM Multimedia 93, Anaheim Calif.
20. Wolf J (1989) The Placement Optimization Program. ACM Sigmetrics Conference, Berkeley Calif.
21. Wolf J, Dias D, Yu P (1993) A parallel sort merge join algorithm for managing data skew. IEEE Trans Parallel Distrib Syst 4:70–86
22. Yu P, Wolf J, Shachnai H (1995) Design and analysis of a look-ahead scheduling scheme to support pause-resume for video-on-demand applications. Multimedia Syst 3(4):137–149
23. Yu P, Chen M-S, Kandlur D (1993) Grouped sweeping scheduling for DASD-based multimedia storage management. Multimedia Syst 1(3):99–109
24. Zipf G (1949) Human Behavior and the Principle of Least Effort. Addison-Wesley, Reading, Mass.
25. International Organization for Standardization. (1991) DCT Coding of Motion Sequences Including Arithmetic Coder. ISO-IEC/JTC1/SC2/WG8 N
26. New York Times (1995) May 22 - June 25

JOEL L. WOLF received his Ph.D from Brown University in 1973 and his Sc.B. from the Massachusetts Institute of Technology in 1968, both in Mathematics. He is currently a Staff memeber at the IBM T.J. Watson research Center, with interests in mathematical optimization. In 1988, he won an IBM Outstanding Innovation Award for his work on the Placement Optimization technique to solve the disk file assignment problem. In 1994, he won another OIA for his work on parallel quera processing. He has also been an Assistant Proffessor of Mathematics at Havard University, as well as a Distinguished Member of Technical Staff and manager at Bell Laboratories. Dr. Wolf is a senior member of IEEE, INFORMS and ACM, and the author of numerous papers and patents.



PHILIP S. YU received the B.S. degree in E.E. from National Taiwan University, Taipe, Taiwan, Republic of China, in 1972, the M.S. and Ph.D. degrees in E.E. from Stanford University, in 1976 and 1978, respectively, and the M.B.A. degree from New York University in 1982. Since 1978 he has been with the IBM Thomas L. Watson Research Center, Yorktown Heights, NY. Currently he is manager of the Software tools and Techniques group. His current research interests include database systems, data mining, multimedia systems, parallel and distributed processing, disk arrays, computer architecture, performance modelling, and workload analysis. He has published more than 200 papers in refereed journals and conferences, and over 130 research reports and 90 invention disclosures. He holds or has applied for 32 US patents. Dr. Yu is a Fellow of the IEEE and the ACM. He is an editor of the IEEE Transactions on Knowledge and Data Engeneering. In addition to serving as program committee members on various conferences, he has served as the program chair of the 2nd International Workshop on Research Issues in Data Engeneering: Transaction ans Query Processing and the program co-chair of the 11th International Conference on Data Engeneering. He has received several IBM and external honors including Best Paper Award, 2 IBM Outstanding Innovation Awards, Outstanding Technical Achievement Award, Research Division Award, and 15 Invention Achievement Awards.



HADAS SHACHNAI received B.S. and Ph.D. degrees in Computer Science from the Israel Institute of Technologiy (Technion) in 1986 and 1991. In 1993–1995 she was a visiting postdoctoral fellow at the IBM T.J. Watson Research Center, Yorktown Heights, NY, and currently she is Assistant Professor in the department of Computer Science at the Technion. Her research interests are performance evaluation and probabilistic modeling of computing and multimedia systems.