DISTRIBUTED MULTIMEDIA SYSTEMS

Advances in computer processing and storage performance and in high speed communications has made it feasible to consider continuous media (e.g., audio and video) servers that scale to thousands of concurrently active clients. The principle characteristics of continuous media is their sustained bit rate requirement (1,2). If a system delivers a clip at a rate lower than its prespecified rate without special precautions (e.g., prefetching), the user might observe frequent disruptions and delays with video and random noises with audio. These artifacts are collectively termed *hiccups.* For example, CD-quality audio (2 channels with 16 bit samples at 44 kHz) requires 1.4 Megabits per second (Mbps). Digital component video based on the CCIR 601 standard requires 270 Mbps for its continuous display. These bandwidths can be reduced using compression techniques due to redundancy in data. Compression techniques are categorized into lossy and lossless. Lossy compression techniques encode data into a format that, when decompressed, yields something similar to the original. With lossless techniques, decompression yields the original. Lossy compression techniques are more effective in reducing both the size and bandwidth requirements of a clip. For example, with the MPEG standard (3), the bandwidth requirement of CD-quality audio can be reduced to 384 kilobits per second. MPEG-1 reduces the bandwidth requirement of a video clip to 1.5 Mbps. With some compression techniques such as MPEG-2, one can control the compression ratio by specifying the final bandwidth of the encoded stream (ranging from 3 to 15 Mbps). However, there are applications that cannot tolerate the use of lossy compression techniques, for example, video signals collected from space by NASA (4).

Most of the compression schemes are Constant Bit Rate (CBR) but some are Variable Bit Rate (VBR). With both techniques, the data must be delivered at a prespecified rate. Typically, CBR schemes allow some bounded variation of this rate based on some amount of memory at the display. With VBR, this variation is not bounded. The VBR schemes have the advantage that for the same average bandwidth as CBR, they can maintain a more constant quality in the delivered image by utilizing more megabits per second when needed, for example, when there is more action in a scene.

The size of a compressed video clip is quite large by most current standards. For example, a two hour MPEG-2 encoded video clip, requiring 3 Mbps, is 2.6 Gigabytes in size. (In this paper, we focus on video due to its significant size and bandwidth requirements that are higher than audio.) To reduce cost of storage, a typical architecture for a video server employs a hierarchical storage structure consisting of DRAM, magnetic disks, and one or more tertiary storage devices. (Magnetic tape typically serves as a tertiary storage device (5)). As the different levels of the hierarchy are traversed starting with memory, the density of the medium and its latency increases while its cost per megabyte decreases. It is assumed that all video clips reside on the tertiary storage device. The disk space is used as a temporary staging area for the frequently accessed clips in order to minimize the number of references to tertiary. This enhances the performance of the system. Once the delivery of a video clip has been initiated, the system is responsible for delivering the data to the settop box of the client at the required rate so that there is no interruption in service. The settop box is assumed to have little memory so that it is incumbent on the server and network to deliver the data in a "just in time" fashion. (We will specify this requirement more precisely and formally later.) Note that the system will have some form of admission control, and while it is reasonable to make a request wait to be

initiated, once started, delivery in support of a hiccup-free display is required. An important concept is that of a stream which is one continuous sequence of data from one clip. At any point in time, a stream is associated with a specific offset within the clip. Note that two requests for the same clip that are offset by some time are two different streams.

In this article, we start by assuming a set of videos stored on disk and describe techniques to stage data from disk to memory buffers. Subsequently, we explore the role of tertiary storage devices and address issues such as how a presentation can be displayed from a tertiary storage device, how data should be staged from tertiary onto the magnetic disk storage, and how pipelining can be used to minimize the incurred startup latency. The models are described assuming CBR encoded data. We do not investigate the networking issues (6) and assume, therefore, that the communication network will transmit a stream at a constant bit rate. Note that this is a simplifying assumption. There will be some amount of variation in the network transmission. To account for this, some amount of buffering is required in the network interface as well as the settop box. We do not consider this further. See (6) for a detailed treatment of buffer requirements to mask delays in end to end delivery of continuous media streams.

In this context, the disk subsystem is responsible for delivering data from the disk image of a clip to buffers such that if R_C is the play-out rate of the object in Mbps, then at τ seconds after the start of the stream, at least $R_C \times \tau$ megabits must have been delivered to the network interface. This ensures that the network never starves for data.

There are a number of ways of scheduling streams depending on the frequency of access to each video. The simplest would be to define a fixed schedule a priori of start times. At the other extreme, one could start a new stream for each request at the earliest possible time (consistent with available resources). Video-on-demand implies the latter policy, but there are variations that will be described in later. With n videos, we define $f_i(t)$ to be the frequency of access to the ith video as a function of time. This notation emphasizes that the frequency of access to videos varies over time. Some variations might be periodic over the course of a day, or some other period, while other variations will not be repetitive (e.g., a video becoming old over a couple of months). It is important, therefore, to design a system that can respond effectively to both variations and do so automatically. As we shall see, striping of objects across disks in an array alleviates this problem.

The following are the main performance metrics that constitute the focus of this article.

1. Throughout: number of simultaneous displays supported by the system.
2. Startup latency: amount of time elapsed from when a request arrives referencing a clip until the time the server starts to retrieve data on behalf of this request.

Startup latency corresponds roughly to the usual basic measure of response time. Alternative applications might sacrifice either startup latency in favor of throughput or vice versa. For example, a service provider that supports video-on-demand might strive to maximize its throughput by delaying a user by tens of seconds to initiate the display. If the same provider decides to support VCR functionalities, for example, pause, resume, fast forward and rewind with scan, then the startup latency starts to become important. This is because a client might not tolerate tens of seconds of latency for a fast forward scan. The impact of startup latency becomes more profound with nonlinear digital editing systems. To explain this, consider the role of such a system in a news organization, for example, CNN. With sports, a producer tailors a presentation together based on highlights of different clips, for example, different events at the Olympics. Moreover, the producer might add an audio narration to logically tie these highlights together. Upon the display of the presentation, the system is required to display (1) the audio narration in synchrony with video, and (2) the highlights in sequence, one after another, with no delay between two consecutive highlights. If the system treats the display of each highlight as a request for a stream, then the startup latency must be minimized between each stream to provide the desired effect. Later, we will describe scheduling techniques that hide this latency.

The rest of this paper is organized as follows. We provide an overview of the current disk technology. Subsequently, we describe scheduling techniques in support of a continuous display and the role of hierarchical storage structures. Then, we focus on optimization techniques that can enhance the performance of a continuous media server. Finally, describes an experimental prototype named Mitra that realizes a number of techniques in this paper.

OVERVIEW OF MAGNETIC DISKS

A magnetic disk drive is a mechanical device operated by its controlling electronics. The mechanical parts of the device consist of a stack of platters that rotate in unison on a central spindle; see (9) for details. A single disk contains a number of platters, as many as sixteen at the time of this writing. Each platter surface has an associated disk head responsible for reading and writing data. Each platter stores data in a series of tracks. A single stack of tracks at a common distance from the spindle is termed a cylinder. To access the data stored in a track, the disk head must be positioned over it. The operation to reposition the head from the current track to the desired track is termed *seek*. Next, the disk must wait for the desired data to rotate under the head. This time is termed *rotational latency*.

The seek time is a function of the distance traveled by the disk arm (7–9). Several studies have introduced analytical models to estimate seek time as a function of this distance. To be independent from any specific equation, this study assumes a general seek function. Thus, let Seek(c) denote the time required for the disk arm to travel c cylinders to reposition itself from cylinder i to cylinder $i + c$ (or $i - c$). Hence, Seek(1) denotes the time required to reposition the disk arm between two adjacent cylinders, while Seek(#*cyl*) denotes a complete stroke from the first to the last cylinder of a disk that consists of #*cyl* cylinders. Typically, seek increases linear distance except for small number of cylinders (7,8). For example, the model used to describe the seek characteristic of Seagate ST31200W disk, consisting of 2697 cylinders, is

$$Seek(c) = \begin{cases} 1.5 + (0.510276 \times \sqrt{c}) & \text{if } c < 108 \\ 6.5 + (0.004709 \times c) & \text{otherwise} \end{cases} \quad (1)$$

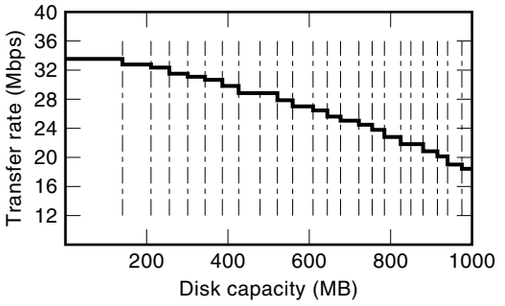

Figure 1. Zone characteristics of the Seagate ST31200W magnetic disk.

A trend in the area of magnetic disk technology is the concept of zoning. Zoning increases the storage capacity of a disk by storing more data on the tracks that constitute the outer regions of the disk drive. With a fixed rotational speed for the disk platters, this results in a disk with variable transfer rate where the data in the outermost region is produced at a faster rate. Figure 1 shows the transfer rate of the 23 different zones that constitute a Seagate disk drive. [Techniques employed to gather these numbers are reported in (10).]

CONTINUOUS DISPLAY

This section starts with a description of a technique to support continuous display of CM objects assuming a platform that consists of a single disk drive with one zone. Subsequently, we extend the discussion to incorporate multizone disks. Next, we describe the role of multiple disk drives in support of environments that strive to support thousands of CM streams. Finally, we show how the architecture can be extended to support a hierarchy of storage structures in order to minimize the cost of providing online access to petabytes of data.

Single Disk Drive

In this article, we assume that a disk drive provides a constant bandwidth, R_D. The approaches discussed can, however, be extended to multizone disk drives with variable transfer rates. Interested readers can consult (11). We also assume that all objects have the same display rate, R_C. In addition, we assume $R_D > R_C$ first assumption is relaxed in Mix of Media Types, and the second in Hierarchical Storage Management. To support continuous display of an object X, it is partitioned into n equisized blocks: $X_0, X_1, \ldots, X_{n-1}$, where n is a function of the block size ($\mathscr{B}$) and the size of X. We assume a block is laid out contiguously on the disk and is the unit of transfer from disk to main memory. The time required to display a block is defined as a time period (T_p):

$$T_p = \frac{\mathscr{B}}{R_C} \tag{2}$$

To support the continuous display of X, one can retrieve blocks of X, one after the other, and send it to the user display consecutively. This is a traditional production-consumption problem. Since R_D, rate of production, is larger than R_C, rate of consumption, a large amount of memory buffer is required at the user site. To reduce the amount of required buffer, one should slow down the production rate. Note that the consumption rate is fixed and dictated by the display bandwidth requirement of the object. Therefore, if X_i and X_j are two consecutive blocks of X, X_j should be in the user buffer once the consumption of X_i has been completed. This is the core of a simple technique, termed round-robin schema (12,2).

With round-robin schema, when an object X is referenced, the system stages X_0 in memory and initiates its display. Prior to completion of a time period, it initiates the retrieval of X_1 into memory in order to ensure a continuous display. This process is repeated until all blocks of an object have been displayed.

To support simultaneous displays of several objects, a time period is partitioned into fixed-size slots, with each slot corresponding to the retrieval time of a block from the disk drive. The number of slots in a time period defines the number of simultaneous displays that can be supported by the system ($\mathscr{N}$). For example, a block size of 1 MB corresponding to a MPEG-2 compressed movie (R_C = 4 Mb/s) has a 2 s display time (T_p = 2). Assuming a typical magnetic disk with a transfer rate of 68 Mb/s (R_D = 68 Mb/s) and maximum seek time of 17 ms, 14 such blocks can be retrieved in 2 s. Hence, a single disk supports 14 simultaneous displays. Figure 2 demonstrates the concept of a time period and a time slot. Each box represents a time slot. Assuming that each block is stored contiguously on the surface of the disk, the disk incurs a seek every time it switches from one block of an object to another. We denote this as T_{W_Seek} and assume that it includes the average rotational latency time of the disk drive. We will not dis-

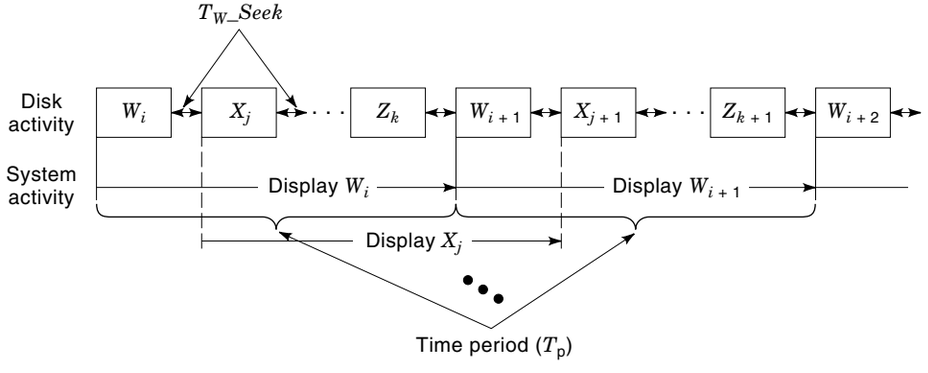

Figure 2. Time period.

cuss rotational latency further because it is a constant added to every seek time.

Since the blocks of different objects are scattered across the disk surface, the round-robin schema should assume the maximum seek time [i.e., *Seek*(#*cyl*)] when multiplexing the bandwidth of the disk among multiple displays. Otherwise, a continuous display of each object cannot be guaranteed. Seek is a wasteful operation that minimizes the number of simultaneous displays supported by the disk. In the worst case, the disk performs $\mathcal{N}$ seeks during a time period. Hence, the percentage of time that the disk performs wasteful work can be quantified as $[\mathcal{N} \times Seek(d)]/T_p \times 100$, where d is the maximum distance between two blocks retrieved consecutively (d = #*cyl* with round-robin). By substituting T_p from Eq. 2, we obtain the percentage of wasted disk bandwidth:

$$wasteful = \frac{\mathcal{N} \times Seek(d) \times R_C}{\mathcal{B}} \times 100 \tag{3}$$

By reducing this percentage, the system can support a higher number of simultaneous displays. We can manipulate two factors to reduce this percentage: (1) decrease the distance traversed by a seek (d), and/or (2) increase the block size ($\mathcal{B}$). A limitation of increasing the block size is that it results in a higher memory requirement. Here, we investigate display schema that reduce the first factor. An alternative aspect is that by manipulating d and fixing the throughput, one can decrease the block size and benefit from a system with a lower memory requirement for staging the blocks. The following paragraphs elaborate more on this aspect.

Suppose $\mathcal{N}$ blocks are retrieved during a time period; then, $T_p = \mathcal{N}\mathcal{B}/R_D + \mathcal{N} \times Seek(\#cyl)$. By substituting T_p from Eq. 2, we solve for $\mathcal{B}$ to obtain

$$\mathcal{B}_{\text{round-robin}} = \frac{R_C \times R_D}{R_D - \mathcal{N} \times R_C} \times \mathcal{N} \times Seek(\#cyl) \tag{4}$$

From Eq. 4, for a given $\mathcal{N}$, the size of a block is proportional to *Seek*(#*cyl*). Hence, if one can decrease the duration of the seek time, then the same number of simultaneous displays can be supported with smaller block sizes. This will save some memory. Briefly, for a fixed number of simultaneous displays, as the duration of the worst seek time decreases (increases) the size of the blocks shrinks (grows) proportionally with no impact on throughput. This impacts the amount of memory required to support $\mathcal{N}$ displays. For example, assume *Seek*(#*cyl*) = 17 ms, R_D = 68 Mb/s, R_C = 4 Mb/s, and $\mathcal{N}$ = 15. From Eq. 4, we compute a block size of 1.08 MB that wastes 12% of the disk bandwidth. If a display schema reduces the worst seek time by a factor of two, then the same throughput can be maintained with a block size of 0.54 MB, reducing the amount of required memory by a factor of two and maintaining the percentage of wasted disk bandwidth at 12%.

The maximum startup latency observed by a request, defined as the amount of time elapsed from the arrival time of a request to the onset of the display of its referenced object, with this schema is

$$\ell_{\text{round-robin}} = T_p \tag{5}$$

This is because a request might arrive a little too late to employ the empty slot in the current time period. Note that ℓ is the maximum startup latency (the average latency is $\ell/2$) when the number of active users is $\mathcal{N} - 1$. If the number of active displays exceeds $\mathcal{N}$, then Eq. 5 should be extended with appropriate queuing models. This discussion holds true for the maximum startup latencies computed for other scheme in this paper.

In the following sections, we investigate two general techniques to reduce the duration of the worst seek time. While the first technique schedules the order of block retrieval from the disk, the second controls the placement of the blocks across the disk surface. These two techniques are orthogonal, and we investigate a technique that incorporates both approaches. Three main objectives are (1) maximizing the number of simultaneous displayed streams (i.e., throughput), (2) minimizing the startup latency time, and (3) minimizing the amount of required memory. Since these objectives are conflicting, there is no single best technique. Each of the described techniques strive to strike a compromise among the mentioned objectives.

Disk Scheduling. One approach to reduce the worst seek time is Grouped Sweeping Scheme (13), GSS. GSS groups $\mathcal{N}$ active requests of a time period into g groups. This divides a time period into g subcycles, each corresponding to the retrieval of $\lceil \mathcal{N}/g \rceil$ blocks. The movement of the disk head to retrieve the blocks within a group abides by the SCAN algorithm in order to reduce the incurred seek time in a group. Across the groups, there is no constraint on the disk head movement. To support the SCAN policy within a group, GSS shuffles the order that the blocks are retrieved. For example, assuming X, Y, and Z belong to a single group, the sequence of the block retrieval might be X_1 followed by Y_4 and Z_6 (denoted as $X_1 \rightarrow Y_4 \rightarrow Z_6$) during one time period, while during the next time period, it might change to $Z_7 \rightarrow X_2 \rightarrow Y_5$. In this case, the display of (say) X might suffer from hiccups because the time elapsed between the retrievals of X_1 and X_2 is greater than one time period. To eliminate this possibility, (13) suggests the following display mechanism: the displays of all the blocks retrieved during subcycle i start at the beginning of subcycle $i + 1$. To illustrate, consider Fig. 3 where g = 2, and $\mathcal{N}$ = 4. The blocks X_1 and Y_1 are retrieved during the first subcycle. The displays are initiated at the beginning of subcycle 2 and last for two subcycles. Therefore, while it is important to preserve the order of groups across the time periods, it is no longer necessary to maintain the order of block retrievals in a group.

The maximum startup latency observed with this technique is the summation of one time period (if the request arrives when the empty slot is missed) and the duration of a subcycle (T_p/g):

$$\ell_{\text{gas}} = T_p + \frac{T_p}{g} \tag{6}$$

By comparing Eq. 6 with Eq. 5, it may appear that GSS results in a higher latency than round-robin. However, this is not necessarily true because the duration of the time period is different with these two techniques due to a choice of different block size. This can be observed from Eq. 2, where the duration of a time period is a function of the block size.

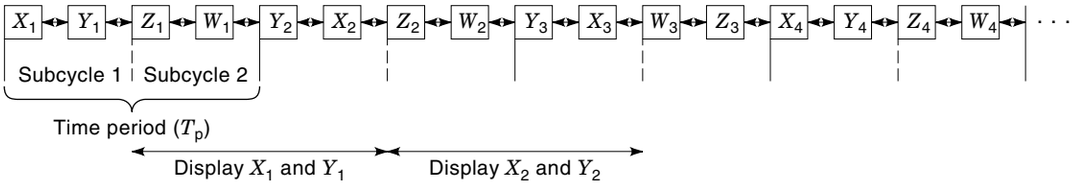

Figure 3. Continuous display with GSS.

To compute the block size with GSS, we first compute the total duration of time contributed to seek times during a time period. Assuming $\lceil \mathcal{N}/g \rceil$ blocks retrieved during a subcycle are distributed uniformly across the disk surface, the disk incurs a seek time of $Seek(\#cyl/(\mathcal{N}/g))$ between every two consecutive block retrievals. This assumption maximizes the seek time according to the square root model, providing the worst case scenario. Since $\mathcal{N}$ blocks are retrieved during a time period, the system incurs $\mathcal{N}$ seek times in addition to $\mathcal{N}$ block retrievals during a period; that is $T_p = \mathcal{N}\mathcal{B}/R_D + \mathcal{N} \times Seek[(\#cyl \times g)/\mathcal{N}]$. By substituting T_p from Eq. 2 and solving for $\mathcal{B}$, we obtain

$$\mathcal{B}_{\text{gas}} = \frac{R_C \times R_D}{R_D - \mathcal{N} \times R_C} \times \mathcal{N} \times Seek\left(\frac{\#cyl \times g}{n}\right) \qquad (7)$$

By comparing Eq. 7 with Eq. 4, observe that the bound on the distance between two blocks retrieved consecutively is reduced by a factor of $g/\mathcal{N}$, noting that $g \leq \mathcal{N}$.

Observe that $g = \mathcal{N}$ simulates the round-robin schema. (By substituting g with $\mathcal{N}$ in Eq. 7, it reduces to Eq. 4.) Other disk scheduling algorithms for continuous media objects are also discussed in the literature. Almost all of them (14–16) can be considered as special cases of GSS because they follow the main concept of scheduling within a time period (or round). The only exception is SCAN-EDF (17).

Constrained Data Placement. An alternative approach to reduce the worst seek time is to control the placement of the blocks across the disk surface. There are many data placement techniques described in the literature (18–22). In this section, we describe a technique termed Optimized REBECA (22) (OREO for short). OREO (23) reduces the worst seek time by bounding the distance between any two blocks that are retrieved consecutively. OREO achieves this by partitioning the disk space into $\mathcal{R}$ regions. Next, successive blocks of an object X are assigned to the regions in a round-robin manner as shown in Fig. 4. The round-robin assignment follows the efficient movement of disk head as in the scan algorithm (24). To display an object, the disk head moves inward (see Fig. 5) from the outermost region toward the innermost one. Once the disk head reaches the innermost region, it is repositioned to the outermost region to initiate another sweep. This minimizes the movement of the disk head required to simultaneously retrieve $\mathcal{N}$ objects because the display of each object abides by the following rules:

1. The disk head moves in one direction (inward) at a time.
2. For a given time period, the disk services those displays that correspond to a single region (termed active region, R_{active}).
3. In the next time period, the disk services requests corresponding to either $R_{\text{active}} + 1$.
4. Upon the arrival of a request referencing object X, it is assigned to the region containing X_0 (say R_X).
5. The display of X does not start until the active region reaches X_0 ($R_{\text{active}} = R_X$).

To compute the worst seek time with OREO, note that the distance between two blocks retrieved consecutively is bounded by the length of a region (i.e., $\#cyl/\mathcal{R}$). This distance is bounded by $2 \times \#cyl/\mathcal{R}$ when the blocks belong to two different regions. This only occurs for the last block retrieved during time period i and the first block retrieved during time period $i + 1$. To simplify the discussions, we eliminated this factor from the equations (see (22) for precise equations).

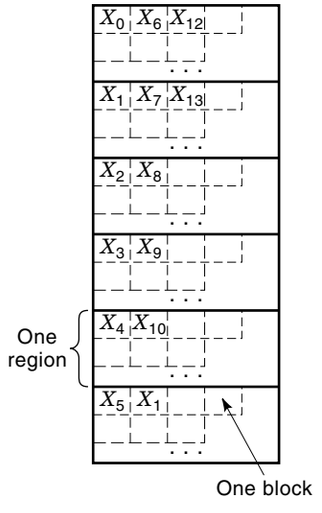

Figure 4. OREO.

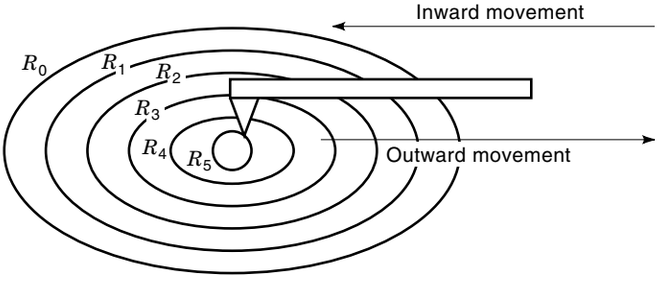

Figure 5. Disk head movement.

Thus, the worst incurred seek time between two block retrievals is $Seek(\#cyl/\mathscr{R})$. Furthermore, the system observes a long seek, Seek(#cyl), every $\mathscr{R}$ regions (to reposition the head to the outermost cylinder). To compensate for this, the system must ensure that after every $\mathscr{R}$ block retrievals, enough data has been prefetched on behalf of each display to eclipse a delay equivalent to Seek(#cyl). There are several ways of achieving this effect. One might force the first block along with every $\mathscr{R}$ other blocks to be slightly larger than the other blocks. We describe OREO based on a fix-sized block approach that renders all blocks to be equisized. With this approach, every block is padded so that after every $\mathscr{R}$ block retrievals, the system has enough data to eclipse the Seek(#cyl) delay. Thus, the duration of a time period is

$$T_p = \frac{\mathscr{N}\mathscr{B}}{R_D} + \mathscr{N} + Seek\left(\frac{\#cyl}{\mathscr{R}}\right) + \frac{Seek(\#cyl)}{\mathscr{R}}$$

By substituting T_p from Eq. 2, we solve for $\mathscr{B}$ to obtain

$$\mathscr{B}_{\text{oreo}} = \frac{R_C \times R_D}{R_D - \mathscr{N} \times R_C} \times \left[\mathscr{N} \times Seek\left(\frac{\#cyl}{\mathscr{R}}\right) + \frac{Seek(\#cyl)}{\mathscr{R}}\right] \quad (8)$$

By comparing Eq. 8 with Eq. 4, observe that OREO reduces the upper bound on the distance between two blocks retrieved consecutively by a factor of $1/\mathscr{R}$.

Introducing regions to reduce the seek time increases the average latency observed by a request. This is because during each time period, the system can initiate the display of only those objects that correspond to the active region. To illustrate this, consider Fig. 6. In Fig. 6(a), Y is stored starting with R_2, while the assignment of both X and Z starts with R_0. Assume that the system can support three simultaneous displays ($\mathscr{N} = 3$). Moreover, assume a request arrives at time T_1, referencing object X. This causes region R_0 to become active. Now, if a request arrives during T_1 referencing object Y, it cannot be serviced until the third time period even though sufficient disk bandwidth is available [see Fig. 6(b)]. Its display is delayed by two time periods until the disk head moves to the region that contains Y_0 (R_2).

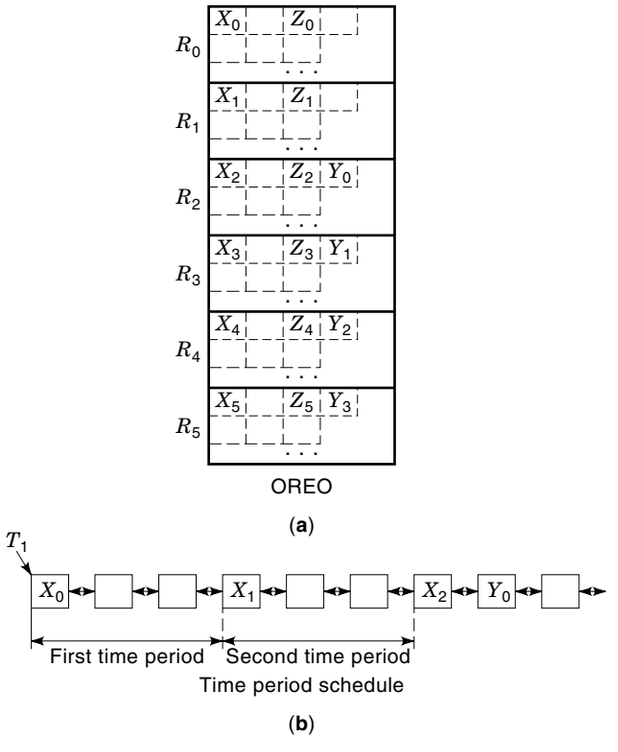

Figure 6. Latency time.

In the worst case, assume (1) a request arrives referencing object Z when $R_{\text{active}} = R_0$, and (2) the request arrives when the system has already missed the empty slot in the time period corresponding to R_0 to retrieve. Hence, $\mathscr{R} + 1$ time periods are required before the disk head reaches R_0 in order to start servicing the request. Hence, the maximum startup latency is computed as

$$\ell_{\text{oreo}} = \begin{cases} (\mathscr{R}+1) \times T_p & \text{if } \mathscr{R} > 2 \\ (2 \times T_p) & \text{if } \mathscr{R} = 2 \\ T_p & \text{if } \mathscr{R} = 1 \end{cases} \quad (9)$$

Hybrid: Disk Scheduling Plus Constrained Data Placement. In order to cover a wide spectrum of applications, GSS and OREO can be combined. Recall that with OREO, the placement of objects within a region is unconstrained. Hence, the distance between two blocks retrieved consecutively is bounded by the length of a region. However, one can introduce the concept of grouping the retrieval of blocks within a region. In this case, assuming a uniform distribution of blocks across a region surface, the distance between every two blocks retrieved consecutively is bounded by $\#cyl \times g/\mathscr{N}\mathscr{R}$. Hence

$$T_p = \frac{\mathscr{N}\mathscr{B}}{R_D} + \mathscr{N} + Seek\left(\frac{\#cyl \times g}{\mathscr{N}\mathscr{R}}\right) + \frac{Seek(\#cyl)}{\mathscr{R}}$$

By substituting T_p from Eq. 2, we solve for $\mathscr{B}$ to obtain

$$\mathscr{B}_{\text{combined}} = \frac{R_C \times R_D}{R_D - \mathscr{N} \times R_C} \times \left[\mathscr{N} \times Seek\left(\frac{\#cyl \times g}{\mathscr{N}\mathscr{R}}\right) + \frac{Seek(\#cyl)}{\mathscr{R}}\right] \quad (10)$$

Observe that, with OREO + GSS, both reduction factors of GSS and OREO are applied to the upper bound on the distance between any two consecutively retrieved blocks (compare Eq. 10 with both Eqs. 7 and 8).

The maximum startup latency observed with OREO + GSS is identical to OREO when $\mathscr{R} > 1$ (see Eq. 9).

Buffer Management. The technique employed to manage memory impacts the amount of memory required to support $\mathscr{N}$ simultaneous displays. A simple approach to manage memory is to assign each user two dedicated blocks of memory: one for retrieval of data from disk to memory and the other for delivery of data from memory to the display station. Trivially, the data is retrieved into one block while it is consumed from the other. Subsequently, the role of these two blocks is switched. The amount of memory required with this technique is:

$$M_{\text{unshared}} = 2 \times \mathscr{N} \times \mathscr{B} \quad (11)$$

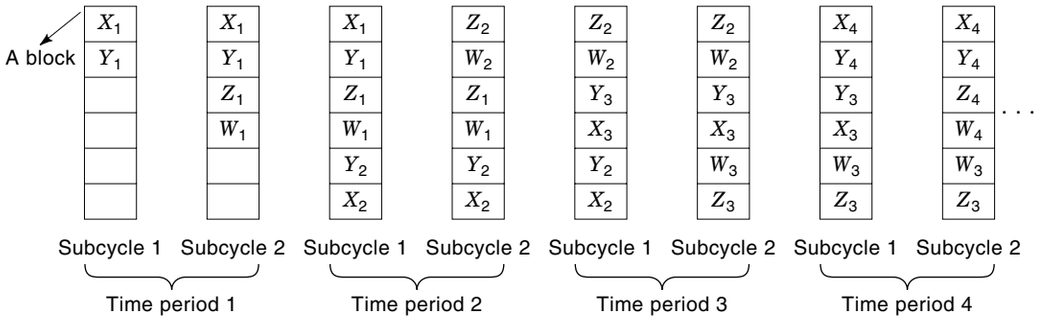

Figure 7. Memory requirement per subcycle.

Note that $\mathscr{B}$ is different for alternative display techniques: $\mathscr{B}_{gss}$ with GSS, $\mathscr{B}_{oreo}$ with OREO, and $\mathscr{B}_{combined}$ with OREO + GSS.

An alternative approach, termed coarse-grain memory sharing, reduces the amount of required memory by sharing blocks among users. It maintains a shared pool of free blocks. Every task (either retrieval or display task of an active request) allocates blocks from the shared pool on demand. Once a task has exhausted the contents of a block, it frees the block by returning to it a shared pool. As described later, when compared with the simple approach, coarse-grain sharing results in lower memory requirement as long as the system employs GSS with the number of groups (g) smaller than $\mathscr{N}$.

The highest degree of sharing is provided by fine-grain memory sharing. With this technique, the granularity of memory allocation is reduced to a memory page. The size of a block is a multiple of the page size. If $\mathscr{P}$ denotes the memory page size, then $\mathscr{B} = m\mathscr{P}$, where m is a positive integer. The system maintains a pool of memory pages (instead of blocks with coarse-grain sharing), and tasks request and free pages instead of blocks.

In the following, we describe the memory requirement of each display technique with both fine and coarse-grain sharing. The memory requirement of the round-robin schema is eliminated because it is a special case of GSS ($g = \mathscr{N}$) and OREO ($\mathscr{R} = 1$).

Coarse-Grain Sharing (CGS). The total amount of memory required by a display technique that employs both GSS and coarse-grain memory sharing is

$$M_{\text{coarse}} = \left(\mathscr{N} + \left\lceil \frac{\mathscr{N}}{g} \right\rceil\right) \times \mathscr{B} \tag{12}$$

To support $\mathscr{N}$ simultaneous displays, the system employs $\mathscr{N}$ blocks for $\mathscr{N}$ displays and $\lceil \mathscr{N}/g \rceil$ blocks for data retrieval on behalf of the group that reads its block from disk. To illustrate, consider the example of Fig. 7, where $g = 2$, and $\mathscr{N} = 4$. From Eq. 12, this requires 6 blocks of memory (see (13) for derivation of Eq. 12). This is because the display of X_1 and Y_1 completes at the beginning of subcycle 2 in the second time period. These blocks can be swapped out in favor of Z_2 and W_2. Note that the system would have required 8 blocks without coarse-grain sharing.

OREO and OREO + GSS can employ Eq. 12. This is because the memory requirement of OREO is a special case of GSS where $g =$ However, the block size ($\mathscr{B}$) computed for each approach is different: $\mathscr{B}_{gss}$ with GSS, $\mathscr{B}_{oreo}$ with OREO, and $\mathscr{B}_{combined}$ with OREO + GSS (see the earlier sections for the computation of the block size with each display technique).

Fine-Grain Sharing (FGS). We describe the memory requirement of fine-grain sharing with a display technique that employs GSS. This discussion is applicable to both OREO and OREO + GSS.

When compared with coarse-grain sharing, fine-grain sharing reduces the amount of required memory because during a subcycle, the disk produces a portion of some blocks while the active displays consume portions of other blocks. With coarse-grain sharing, a partially consumed block cannot be used until it becomes completely empty. However, with fine-grain sharing, the system frees up pages of a block that have been partially displayed. These pages can be used by other tasks that read data from disk.

Modeling the amount of memory required with FGS is more complex than that with CGS. While it is possible to compute the precise amount of required memory with CGS, this is no longer feasible with FGS. This is because CGS frees blocks at the end of each subcycle where the duration of a subcycle is fixed. However, FGS frees pages during a subcycle, and it is not feasible to determine when the retrieval of a block ends within a subcycle because the incurred seek times in a group are unpredictable. Therefore, we model the memory requirement within a subcycle for the worst case scenario.

Let t denote the time required to retrieve all the blocks in a group. Theoretically, t can be a value between 0 and the duration of a subcycle; that is, $0 \leq t \leq T_p/g$. We first compute the memory requirement as a function of t and then discuss the practical value of t. We introduce t to generate another end point (beside the end of a subcycle) where the memory requirement can be modeled accurately. The key observation is that between t and the end of subcycle, nothing is produced on behalf of a group, while display of requests in a subcycle continues at a fixed rate of R_C. Hence, we model the memory requirement for the worst case where all the blocks are produced in order to eliminate the problem of unpredictability of each block retrieval time in a subcycle. Assuming S_i is the end of subcycle i, the maximum amount of memory required by a group is at $S_i + t$ because the maximum amount of data is produced, and the minimum amount is consumed at this point. Observe that at a point x, where $S_i + t < x \leq S_{i+1}$, data is only consumed, reducing the amount of required memory. Moreover, at a point y where $S_i \leq y < S_i + t$, data is still being produced.

The number of pages produced (required) during t is

$$produced = \left\lceil \frac{\mathcal{N}}{g} \right\rceil \times m \tag{13}$$

The number of pages consumed (released) during t is

$$consumed = \left\lfloor \frac{t \times \mathcal{N}m}{T_p} \right\rfloor \tag{14}$$

This is because the amount of data consumed during a time period is $\mathcal{N}m$ pages, and hence, the amount consumed during t is t/T_p of $\mathcal{N}m$ pages. We use floor function for consumption and ceiling function for the production because the granularity of memory allocation is in pages. Hence, neither a partially consumed page (floor function) nor a partially produced page (ceiling function) is available on the free list. Moreover, m is inside the floor function because the unit of consumption is in number of pages, while it is outside the ceiling function because the unit of production is in blocks.

One might argue that the amount of required memory is the difference between the volume of data produced and consumed. This is an optimistic view that assumes everything produced before S_i has already been consumed. However, in the worst case, all the $\mathcal{N}$ displays might start simultaneously at time period j ($T_p(j)$). Hence, the amount of data produced during $T_p(j)$ is higher than the amount consumed. This is because the production starts during the first subcycle of $T_p(j)$, while consumption starts at the beginning of the second subcycle. It is sufficient to compute this remainder (rem) and add it to *produced* − *consumed* in order to compute the total memory requirement because all the produced data is consumed after $T_p(j)$.

To compute *rem,* Fig. 8 divides $T_p(j)$ into g subcycles and demonstrates the amount of produced and consumed data during each subcycle. The total amount that is produced during each time period is $\mathcal{N}m$ pages. During the first subcycle, there is nothing to consume. For the other $g - 1$ subcycles, $1/g$ of what have been produced can be consumed. Hence, from the figure, the total consumption during $T_p(j)$ is $\mathcal{N}m/g$ $1/g$ $(1 + 2 + \cdots + (g - 1))$. By substituting $(1 + 2 + \cdots + (g - 1))$ with $[g(g - 1)]/2$, *rem* can be computed as

$$rem = \mathcal{N}m - \frac{\mathcal{N}m(g-1)}{2g} \tag{15}$$

The total memory requirement is *produced* − *consumed* + *rem,* or

$$M_{\text{fine}} = \mathcal{N}m + \frac{\mathcal{N}}{g}m - \frac{t.\mathcal{N}m}{T_p} - \frac{Nm(g-1)}{2g} \tag{16}$$

Note that Eq. 16 is an approximation because we eliminated the floor and ceiling functions from the equation. For large values of m, the approximation is almost identical to the actual computation. An interesting observation is that if the size of a page is equal to the size of a block, then Eq. 16 can be reduced to Eq. 12. This is because the last two terms in Eq. 16 correspond to the number of pages released during t and the first time period, respectively. Since with coarse-grain, no pages are released during these two periods; the last two terms of Eq. 16 become zero, producing Eq. 12.

The minimum value of t is computed when all the $\lceil \mathcal{N}/g \rceil$ blocks are placed contiguously on the disk surface. The time required to retrieve them is the practical minimum value of t and is computed as

$$t_{\text{practical}} = \left\lceil \frac{\mathcal{N}}{g} \right\rceil \times \frac{\mathcal{B}}{R_D} \tag{17}$$

The number of groups g impacts the memory requirement with both coarse and fine-grain sharing in two ways. First, as one increases g, the memory requirement of the system decreases because the number of blocks staged in memory is $\lceil \mathcal{N}/g \rceil$. On the other hand, this results in a larger block size in order to support the desired number of users, resulting in higher memory requirement. Thus, increasing g might result in either a higher or a lower memory requirements. Yu et al. (13) suggests an exhaustic search technique to determine the optimal value of g $(1 \le g \le \mathcal{N})$ in order to minimize the entire memory requirement for a given $\mathcal{N}$.

An implementation of FGS (beyond the focus of this article) must address how the memory is managed. This is because memory might become fragmented when pages of a block are allocated and freed incrementally. With fragmented memory, either (1) the disk interface should be able to read a block into m disjoint pages, or (2) the memory manager must bring m consecutive pages together to provide the disk manage with m physically contiguous pages to read a block into. The first approach would compromise the portability of the final system because it entails modifications to the disk inter-

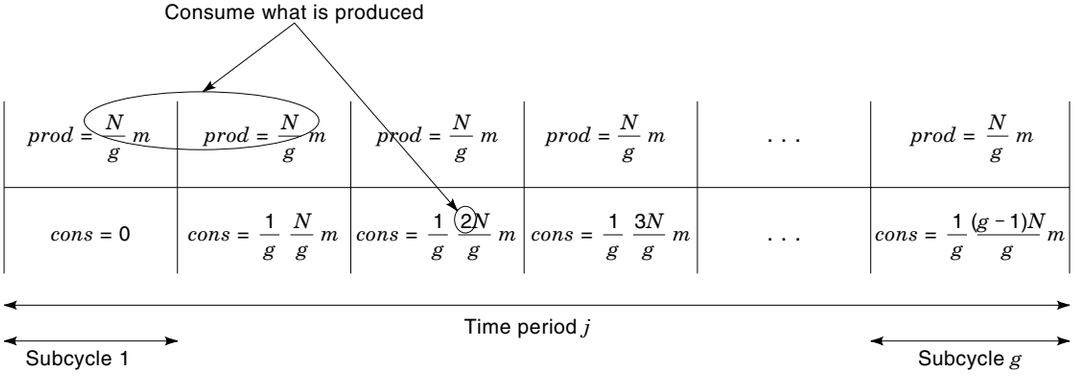

Figure 8. Memory requirement of the jth time period.

face. With the second approach, one may implement either a detective or preventive memory manager. A detective memory manager waits until memory becomes fragmented before reorganizing memory to eliminate this fragmentation. A preventive memory manager avoids the possibility of memory fragmentation by controlling how the pages are allocated and freed. When compared with each other, the detective approach requires more memory than the preventive one (and would almost certainly require more memory than the equations derived in this section). However, the preventive approach would most likely incur a higher CPU overhead because it checks the state of memory per page allocation/release.

Multi-Zone Disks. To guarantee a continuous display, the techniques described in the previous section must assume the transfer rate of the innermost zone. An alternative is to constrain the layout of data across the disk surface and the schedule for its retrieval (25–26, 11). This section describes three such techniques. With all techniques, there is a tradeoff between throughput, amount of required memory, and the incurred startup latency. We illustrate these tradeoffs starting with a brief overview of two techniques and then discussing the third approach in detail.

Track pairing (25), organizes all tracks into pairs such that the total capacity of all pairs is identical. When displaying a clip, a track-pair is retrieved on its behalf per time period (alternative schedules are detailed in (25). Similar to the GSS discussions earlier, the system can manipulate the retrieval of physical tracks on behalf of multiple active displays to minimize the impact of seeks.

Assuming that the number of tracks in every zone is a multiple of some fixed number, (26) constructs Logical Tracks (LT) from the same numbered physical track of the different zones. The order of tracks in a LT is by zone number. When displaying a clip, the system reads an LT on its behalf per time period. An application observes a constant disk transfer rate for each LT retrieval. This forces the disk to retrieve data from the constituting physical tracks in immediate succession by zone order. Recall that we assumed a logical disk drive that consists of d physical disks. If d equals the number of zones (m), the physical tracks of a LT can be assigned to a different disk. This facilitates concurrent retrieval of physical tracks that constitute an LT. Similarly, to facilitate concurrent retrieval with track pairing, if d is an even number, then the disks can be paired such that track i of one disk is paired with track $\#cyl - i$ track of another disk.

The third approach as detailed in (11) organizes a clip at the granularity of blocks (instead of tracks). We describe two variations of this approach that guarantee continuous display while harnessing the average transfer rate of m zones, namely, FIXed Block size (FIXB) and VARiable Block size (VARB) (11). These two techniques assign the blocks of an object to the available zones in a round-robin manner, starting with an arbitrary zone. With both techniques, there are a family of scheduling techniques that ensure a continuous display. One might require the disk to retrieve m blocks of an object assigned to m different zones in one sweep. Assuming that $\mathcal{N}$ displays are active, this scheduling paradigm would result in $\mathcal{N}$ disk sweeps per time period and substantial amount of buffer space. (With this scheduling paradigm, VARB would be similar to LT (26).) An alternative scheduling paradigm might multiplex the bandwidth of each zone among the $\mathcal{N}$ displays and visit them in a round-robin manner. It requires the amount of data retrieved during one scan of the disks on behalf of a display (m time periods that visit all the zones, starting with the outermost zone) to equal that consumed by the display. This reduces the amount of required memory. However, it results in a higher startup latency. We will focus on this scheduling paradigm for the rest of this section.

To describe the chosen scheduling technique, assume a simple display scheme (GSS with $g = \mathcal{N}$). We choose the block size (the unit of transfer from a zone on behalf of an active display) to be a function of transfer rate of each zone such that the higher transfer rate of fast zones compensates for that of the slower zones. The display time of the block that is retrieved from the outermost zone (Z_0) on behalf of a display exceeds the duration of a time period ($T_P(Z_0)$); see Fig. 9. Thus, a portion of the block remains memory resident in the available buffer space. During the time period when the innermost zone is active, the amount of data displayed exceeds the block size, consuming the buffered data. In essence, the display of data is synchronized relative to the retrieval from the outermost zone. This implies that if the first block of an

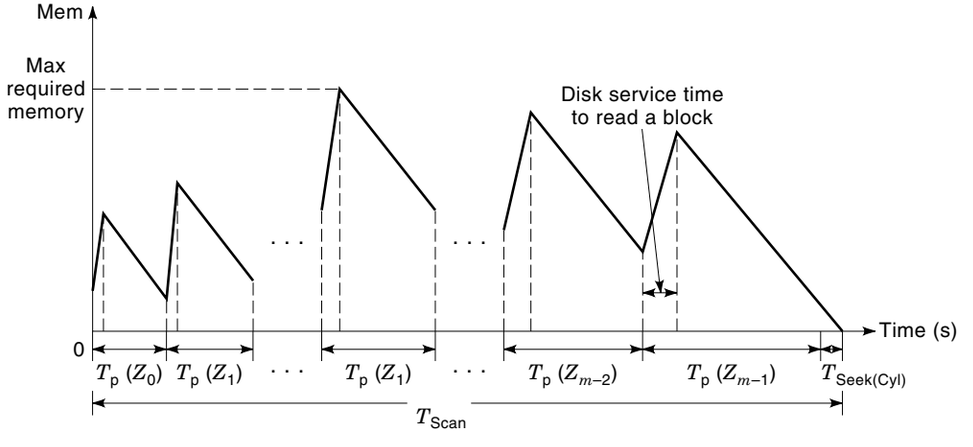

Figure 9. Memory required on behalf of a display with FIXB.

Table 1. Seagate ST31200W Disk

	FIXB				VARB			
$\mathcal{N}$	Mem. (MB)	Max l (s)	% Wasted disk space	% Avg wasted band.	Mem. (MB)	Max l (s)	% Wasted disk space	% Avg wasted band.
1	0.007	0.44	58.0	94.1	0.011	0.44	40.4	94.3
2	0.023	0.92	58.0	88.2	0.044	0.92	40.4	88.6
4	0.107	2.10	58.0	76.5	0.192	2.08	40.4	77.3
8	0.745	6.03	58.1	53.1	1.059	5.88	40.4	54.6
10	1.642	9.66	58.1	41.4	2.078	9.26	40.5	43.2
12	3.601	16.15	58.2	29.7	4.040	15.06	40.6	31.9
13	5.488	21.77	58.3	23.9	5.759	19.84	40.4	26.2
14	8.780	31.05	58.0	18.0	8.511	27.26	40.6	20.5
15	15.515	49.23	58.4	12.1	13.481	40.34	41.0	14.8
16	35.259	100.9	58.3	6.3	24.766	69.53	41.3	9.2
17	536.12	1392.5	73.8	0.4	72.782	192.4	42.2	3.5

object is assigned to the zones starting with a zone other than the innermost zone, then its display might be delayed relative to its retrieval in order to avoid hiccups.

Both FIXB and VARB waste disk space due to (1) a round-robin assignment of blocks to zones, and (2) a nonuniform distribution of the available storage space among the zones. (In general, the outermost zones have a higher storage capacity relative to the innermost zones.) Once the storage capacity of a zone is exhausted, no additional blocks can be assigned to the remaining zones because it would violate the round-robin assignment of blocks. Table 1 compares FIXB and VARB by reporting on the amount of required memory, the maximum incurred startup latency assuming fewer than $\mathcal{N}$ active displays, and the percentage of wasted disk space and disk bandwidth with the Seagate ST31200W disk. The percentage of wasted disk space with both FIXB and VARB is dependent on the physical characteristics of the zones. While VARB wastes a lower percentage of the Seagate disk space, it wastes a higher percentage of another analyzed disk space (HP C2247) (11).

If we assumed the transfer rate of the innermost zone as the transfer rate of the entire disk and employed the discussion earlier to guarantee a continuous display, the system would support twelve simultaneous displays, require 16.09 Mbytes of memory, and incur a maximum startup latency of 7.76 s. A system that employs either FIXB or VARB supports twelve displays by requiring less memory and incurring a higher startup latency; see sixth row of Table 1. For a high number of simultaneous displays (16 and 17, see last two rows of Table 1), when compared with FIXB, VARB requires a lower amount of memory and incurs a lower maximum startup latency. This is because VARB determines the block size as a function of the transfer rate of each zone, while FIXB determines the block size as a function of the average transfer rate of the zones, that is, one fix-sized block for all zones. Thus, the average block size chosen by VARB is smaller than the block size chosen by FIXB for a fixed number of users. This reduces the amount of time required to scan the disk (T_{Scan} in Fig. 9) which, in turn, reduces both the amount of required memory and the maximum startup latency, see (11) for details.

A system designer is not required to configure either FIXB or VARB using the vendor specified zone characteristics. Instead, one may logically manipulate the number of zones and their specifications by either (1) merging two or more physically adjacent zones into one and assuming the transfer rate of this logical zone to equal that of the slowest participating physical zone, or (2) eliminating one or more of the zones. For example, one might merge zones 0 to 4 into one logical zone, zones 5 to 12 in a second logical zone, and eliminate zones 13 to 23 altogether. With this logical zone organization, VARB supports 17 simultaneous displays by requiring 10 Mbytes of memory and observing a maximum startup latency of 6.5 s the amount of required memory is lower than that required by the approach that assumes the transfer rate of the innermost zone for the entire disk, see the previous paragraph. Computed with a system that assumes the transfer rate of the innermost zone as the transfer rate of the entire disk, the throughput of the system is increased by 40% of the expense of wasting 35% of the available disk space. A more intelligent arrangement might even outperform this one. The definition of outperform is application dependent. A configuration planner that employs heuristics to strike a compromise between the conflicting factors is described in (11).

Multiple Disk Drives

The bandwidth of a single disk is insufficient for those applications that strive to support thousands of simultaneous displays. One may employ a multidisk architecture for these applications. In this report, we assume a homogeneous set of disk drives; that is, all the disk drives have an identical transfer rate of R_D. The issues discussed here can be extended to a heterogeneous platform. Interested readers are encouraged to consult (27).

Multidisk environments raise the interesting research problem of data placement. That is, on which disk (or set of disks) a single clip should be stored. A simple technique is to assign a clip to a disk in its entirety. The drawback is that a single disk storing all the hot objects might become a bottleneck. In (28), we proposed a detective mechanism to detect a bottleneck and resolve it by replicating the hot object(s) into the other disk drives. However, we have learned that both partitioning the resources and a detective mechanism are not appropriate for such a setup. For the rest of this section, we describe three alternative data placement techniques in a

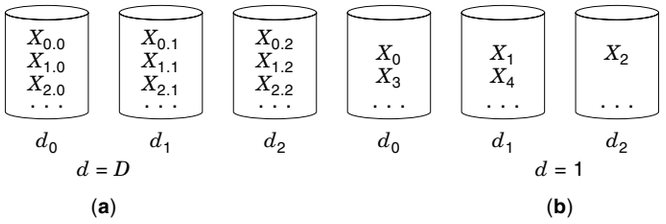

Figure 10. RAID ($d = D$) vs. round-robin retrieval ($d = 1$).

multidisk hardware platform. These techniques neither partition resources nor employ detective mechanisms to balance the load. That is, each CM object is striped across all the disk drives. Hence, resources (disk drives) are not partitioned, the load is distributed evenly across all the drives and there is no need to detect bottlenecks to resolve load imbalance (i.e., bottlenecks are prevented and not detected). The differences among the three techniques is mainly on their retrieval schedule.

RAID Striping. One way to render multiple disks is to follow the RAID (29) architecture. This has been done in Streaming RAID (30). Briefly, each block of an object is striped into fragments where fragments are assigned to the disks in a round-robin manner. For example, in Fig. 10a, block X_0 is declustered across the 3 disks. Each fragment of this block is denoted as $X_{0,i}$; $0 \leq i < D$. Given a platform consisting of D disk drives, to retrieve a block, all the D disk drives transfer the corresponding fragments simultaneously. Subsequently, the block is formed from the fragments in memory and sent to the client.

Conceptually, a RAID cluster can be considered as a single logical disk drive. Hence, the display techniques and memory requirements discussed earlier can be applied here with almost no modification. In theory, a RAID cluster consisting of D disk drives have a sustained transfer rate of $D \times R_D$. However, in practice, as one increases D, the seek time dominates the sustained transfer rate. This is because seek time is fixed and is not improved as D increases. Therefore, as D grows, the RAID system spends a higher percentage of time doing seeks (wasteful work) as opposed to data transfer (useful work). In summary, the RAID architecture is not scalable in throughput (see Fig. 11a).

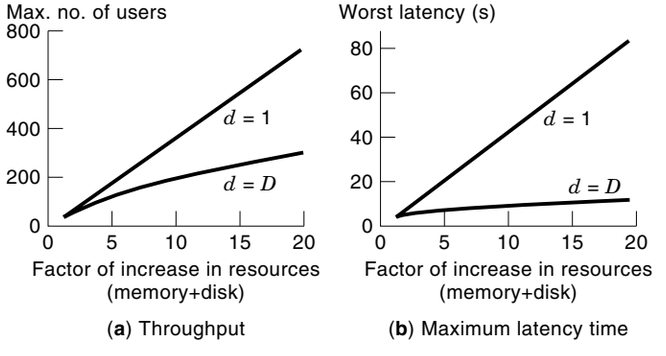

Figure 11. RAID ($d = D$) vs round-robin retrieval ($d = 1$).

Round-Robin Retrieval. With round-robin retrieval, the blocks of an object X are assigned to the D disk drives in a round-robin manner. The assignment of X_0 starts with an arbitrary disk. Assuming a system with three disk drives, Fig. 11(b) demonstrates the assignment of blocks of X with this choice of value. When a request references object X, the system employs the idle slot on the disk that contains X_0 (say d_i) to display its first block. Before the display of X_0 completes, the system employs cluster $d_{(i+1) \bmod D}$ to display X_1. This process is repeated until all blocks of an object have been retrieved and displayed. This can be considered as if the system supports D simultaneous time periods, one per disk drive. Hence, the display techniques and memory requirements discussed earlier can be applied here with straightforward modifications (see (31). The throughput of the system (i.e., maximum number of displays) scales linearly as a function of additional resources in the system. However, its maximum latency also scales linearly (see Fig. 11). To demonstrate this, assume that each disk in Fig. 10(b) can support three simultaneous displays. Assume that eight displays are active and that the assignment of object X starts with d_0 (X_0 resides on d_0). If the request referencing object X arrives too late to utilize the idle slot of d_0, it must wait three (i.e., D) time periods before the idle slot on d_0 becomes available again (see Fig. 12). Hence, the maximum latency time is $T_p \times D$.

Hybrid (Disk Clusters). As mentioned neither RAID striping nor round-robin retrieval scales as one increases resources. In (31–33), we proposed a hybrid approach. Hybrid striping partitions the D disks into k clusters of disks with each cluster consisting of d disks: $k = \lfloor D/d \rfloor$. Next, it assigns the blocks of object X to the clusters in a round-robin manner. The first block of X is assigned to an arbitrarily chosen disk cluster. Each block of an object is declustered (34) across the d disks that constitute a cluster. For example, in Fig. 13, a system consisting of six disks is partitioned into three clusters, each consisting of two disk drives. The assignment of the blocks of X starts with cluster 0. This block is declustered into two fragments: $X_{0.0}$ and $X_{0.1}$. When a request references object X, the system employs the idle slot on the cluster that contains X_0 (say C_i) to display its first block. Before the display of X_0 completes, the system employs cluster $C_{(i+1) \bmod k}$ to display X_1. This process is repeated until all blocks of an object have been retrieved and displayed. Note that the hybrid striping simulates RAID striping when $d = D$ and round-robin retrieval when $d = 1$.

The hybrid striping by varying the number of disk drives within a cluster as well as changing the number of clusters can strike a compromise between throughput and latency time. Given a desired throughput and latency ($\mathcal{N}_{\text{desired, desired}}$), (31) describes a configuration planner that determines a value for the configuration parameters of a system. The value of these parameters is chosen such that the total cost of the system is minimized.

Hierarchical Storage Management

The storage organization of systems that support multimedia applications is expected to be hierarchical, consisting of a tertiary storage device, a group of disk drives, and some memory (32). The database resides permanently on the tertiary storage device, and its objects are materialized on the

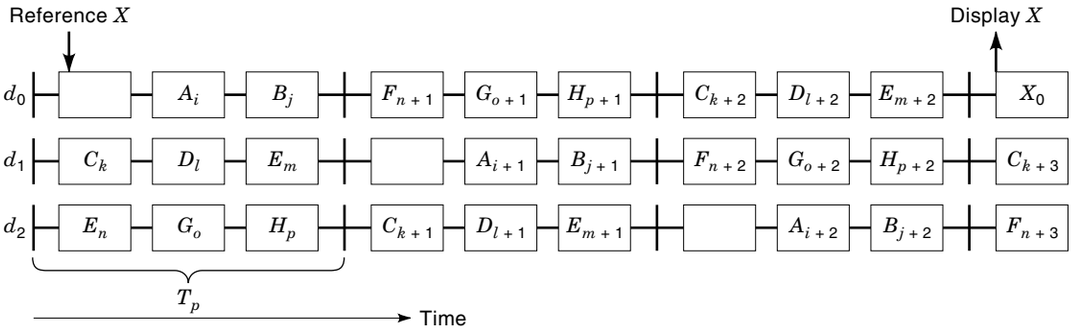

Figure 12. Maximum latency time with striping.

disk drives on demand (and deleted from the disk drives when the disk storage capacity is exhausted). A small fraction of a referenced object is staged in memory to support its display.

Assume a hierarchical storage structure consisting of random access memory (DRAM), magnetic disk drives, optical disks, and a tape library (35) (see Fig. 15). As the different strata of the hierarchy are traversed starting with memory (termed stratum 0), both the density of the medium (the amount of data it can store) and its latency increase, while its cost per megabyte of storage decreases. At the time of this writing, these costs vary from $40/Mbyte of DRAM to $0.6/Mbyte of disk storage to $0.3/Mbyte of optical disk to less than $0.05/Mbyte of tape storage. An application referencing an object that is disk resident observes both the average latency time and the delivery rate of a magnetic disk drive (which is superior to that of the tape library). An application would observe the best performance when its working set becomes resident at the highest level of the hierarchy: memory. However, in our assumed environment, the magnetic disk drives are the more likely staging area for this working set due to the large size of objects. Typically, memory would be used to stage a small fraction of an object for immediate processing and display. We define the working set (36) of an application as a collection of objects that are repeatedly referenced. For example, in existing video stores, a few titles are expected to be accessed frequently and a store maintains several (sometimes many) copies of these titles to satisfy the expected demand. These movies constitute the working set of a database system whose application provides a video-on-demand service.

One might be tempted to replace the magnetic disk drives with the tertiary storage devices in order to reduce the cost further. This is not appropriate for the frequently referenced objects that require a fraction of a second transfer initiation delays, that is, the time elapsed from when a device is activated until it starts to produce data. This delay is determined by the time required for a device to reposition its read head to the physical location containing the referenced data; this time is significantly longer for tertiary storage device (ranges from several seconds to minutes) as compared to that for a magnetic disk drive (ranges from 10 to 30 ms). Similarly, the tertiary storage device should not be replaced by magnetic disk drives because (1) the cost of storage increases, and (2) it might be acceptable for some applications to incur a high latency time for infrequently referenced objects.

In this section, we first describe different data flows among the three storage components of the system (memory, disk, tertiary). Next, a pipelining mechanism is explained to reduce the startup latency when a request references a tertiary resident CM object. Finally, we describe some techniques to manage the disk storage space. Note that the assumed hardware architecture is identical to that of Space Management.

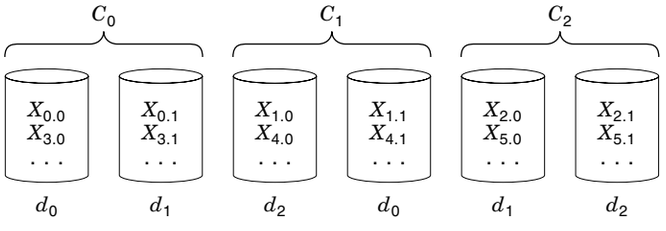

Figure 13. Hybrid striping.

Data Flows. Assuming an architecture that consists of some memory, several disk drives, and a tertiary storage device, two alternative organization of these components can be considered: (1) memory serves as an intermediate staging area between the tertiary storage device, the disk drives, and the display stations, and (2) the tertiary storage device is visible only to the disk drives via a fixed size memory. With the first organization, the system may elect to display an object from the tertiary storage device by using the memory as an intermediate staging area. With the second organization, the data must first be staged on the disk drives before it can be displayed. In (32), we capture these two organizations using three alternative paradigms for the flow of data among the different components:

- Sequential Data Flow (SDF): The data flows from tertiary to memory (STREAM 1 of Fig. 14), from memory to the disk drives (STREAM 2), from the disk drives back to memory (STREAM 3), and finally from memory to the display station referencing the object (STREAM 4).
- Parallel Data Flow (PDF): The data flows from the tertiary to memory (STREAM 1), and from memory to both the disk drives and the display station in order to materialize (STREAM 2) and display (STREAM 4) the object simultaneously. (PDF eliminates STREAM 3.)
- Incomplete Data Flow (IDF): The data flows from tertiary to memory (STREAM 1) and from memory to the

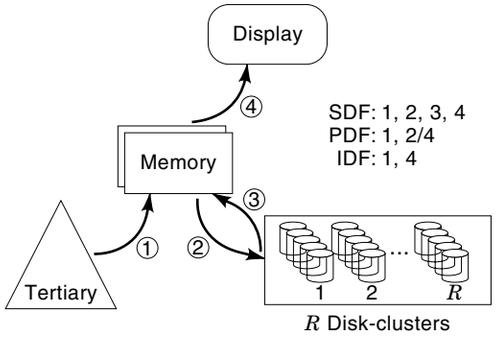

Figure 14. Three alternative dataflow paradigms.

display station (STREAM 4) to support a continuous retrieval of the referenced object. (IDF eliminates both STREAM 2 and 3.)

Figure 14 models the second architecture (tertiary storage is accessible only to the disk drives) by partitioning the available memory into two regions: one region serves as an intermediate staging area between tertiary and disk drives (used by STREAM 1 and 2), while the second serves as a staging area between the disk drives and the display stations (used by STREAM 3 and 4). SDF can be used with both architectures. However, neither PDF nor IDF is appropriate for the second architecture because the tertiary is accessible only to the disk drives. When the bandwidth of the tertiary storage device is lower than the bandwidth required by an object, SDF is more appropriate than both PDF and IDF because it minimizes the amount of memory required to support a continuous display of an object. IDF is ideal for cases where the expected future access to the referenced object is so low that it should not become disk resident (i.e., IDF avoids this object from replacing other disk resident objects).

Pipelining Mechanism. With hierarchical storage organization, when a request references an object that is not disk resident, one approach might materialize the object on the disk drives in its entirety before initiating its display. In this case, assuming a zero system load, the latency time of the system is determined by the time for the tertiary to reposition its read head to the starting address of the referenced object, the bandwidth of the tertiary storage device, and the size of the referenced object. Assuming that the referenced object is continuous media (e.g., audio, video) and requires a sequential retrieval to support its display, a superior alternative is to use pipelining (32) in order to minimize the latency time. Briefly, the pipelining mechanism splits an object into s logical slices (S_1, S_2, S_3, . . ., S_s) such that the display time of S_1 overlaps the time required to materialize S_2, the display time of S_2 overlaps the time to materialize S_3, and so on and so forth. This ensures a continuous display while reducing the latency time because the system initiates the display of an object once a fraction of it (i.e., S_1) becomes disk resident.

With pipelining, two possible scenarios might happen: the bandwidth of the tertiary is either (1) lower or (2) higher than the bandwidth required to display an object the discussion for the case when the bandwidth of tertiary is equivalent to the display is a special case of item (2). The ratio between the production rate of tertiary and the consumption rate at a display station is termed Production Consumption Ratio (PCR = $B_{\text{Tertiary}}/B_{\text{Display}}$).

When $PCR < 1$, the time required to materialize an object is greater than its display time. Neither PDF nor IDF is appropriate because the bandwidth of tertiary cannot support a continuous display of the referenced object (assuming that the size of the first slice exceeds the size of memory). With SDF, the time required to materialize X is $\lceil n/PCR \rceil$ time periods, while its display requires n time periods. If X is a tertiary resident, without pipelining, the latency time incurred to display X is $\lceil n/PCR \rceil + 1$ time periods. (Plus one because an additional time period is needed to both flush the last subobject to the disk cluster and to allow the first subobject to be staged in the memory buffer for display). To reduce this latency time, a portion of the time required to materialize X can be overlapped with its display time.

When $PCR > 1$, the bandwidth of tertiary exceeds the bandwidth required to display an object. Two alternative approaches can be employed to compensate for the fast production rate: either (1) multiplex the bandwidth of tertiary among several requests referencing different objects, or (2) increase the consumption rate of an object by reserving more time slots per time period to render that object disk resident. The first approach wastes the tertiary bandwidth because the device is required to reposition its read head multiple times. The second approach utilizes more resources in order to avoid the tertiary device from repositioning its read head. For more detailed description of pipelining please refer to (32).

Space Management. In general, assuming that the storage structure consists of n strata, we assume that the database resides permanently on stratum $n - 1$. For example, Fig. 15 shows a system with four strata in which the database resides on stratum 3. Objects are swapped in and out of a device at strata $i < n$, based on their expected future access patterns with the objective of minimizing the frequency of access to the slower devices at higher strata. This objective minimizes the average latency time incurred by requests referencing objects.

At some point during the normal mode of operation, the storage capacity of the device at stratum i will be exhausted. Once an object o_x is referenced, the system may determine

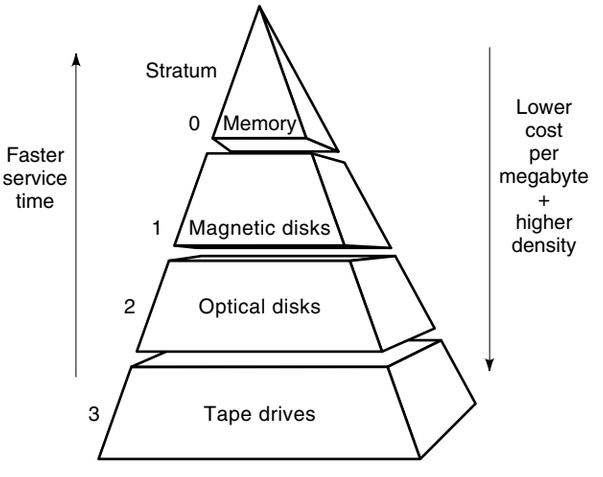

Figure 15. Hierarchical storage system.

that the expected future reference to o_x is such that it should reside on a device at this stratum. In this case, other objects should be swapped out in order to allow o_x to become resident here.

In this section, we focus on how to manage the disk space when objects are migrated in and out of the disk storage from the tertiary storage. We describe two orthogonal techniques. The first, termed EVEREST, manages the blocks of a continuous media object. It approximates a contiguous layout of a file (i.e., a block). The second technique, PIRATE, manages the entire CM object. Each block of the CM object, however, can be managed employing EVEREST. PIRATE is a replacement technique which replaces CM objects partially striving to keep the head of the objects disk resident. Therefore, PIRATE is a nice complement of pipelining. As explained later, however, PIRATE is only appropriate for single user systems. For multi-user systems, objects should be swapped in their entirety.

EVEREST. With η media types, a CM file system might be forced to manage η different block sizes. Moreover, the blocks of different objects might be staged from the tertiary storage device onto magnetic disk storage on demand. A block should be stored contiguously on disk. Otherwise, the disk would incur seeks when reading a block, reducing disk bandwidth. Moreover, it might result in hiccups because the retrieval time of a block might become unpredictable. To ensure a contiguous layout of a block, we considered four alternative approaches: disk partitioning, extent-based (37–39) multiple block sizes, and an approximate contiguous layout of a file. We chose the final approach, resulting in the design and implementation of the EVEREST file system (40). Below, we describe each of the other three approaches and our reasons for abandoning them.

With disk partitioning, assuming η media types with η different block sizes, the available disk space is partitioned into η regions, one region per media type. A region i corresponds to media type i. The space of this region is partitioned into fix sized blocks, corresponding to $\mathscr{B}(M_i)$. The objects of media type i compete for the available blocks of this region. The amount of space allocated to a region i might be estimated as a function of both the size and frequency of access of objects of media type i (41). However, partitioning of disk space is inappropriate for a dynamic environment where the frequency of access to the different media types might change as a function of time. This is because when a region becomes cold, its space should be made available to a region that has become hot. Otherwise, the hot region might start to exhibit a thrashing (42) behavior that would increase the number of retrievals from the tertiary storage device. This motivates a reorganization process to rearrange disk space. This process would be time consuming due to the overhead associated with performing I/O operations.

With an extent-based design, a fixed contiguous chunk of disk space, termed an extent, is partitioned into fix-sized blocks. Two or more extents might have different page sizes. Both the size of an extent and the number of extents with a prespecified block size (i.e., for a media type) is fixed at system configuration time. A single file may span one or more extents. However, an extent may contain no more than a single file. With this design, an object of a media type i is assigned one or more extents with block size $\mathscr{B}(M_i)$. In addition to suffering from the limitations associated with disk partitioning, this approach suffers from internal fragmentation with the last extent of an object being only partially occupied. This would waste disk space, increasing the number of references to the tertiary storage device.

With the multiple block size approach (MBS), the system is configured based on the media type with the lowest bandwidth requirement, say M_1. MBS requires the block size of each of media type j to be a multiple of $\mathscr{B}(M_1)$; i.e., $\mathscr{B}(M_j) = \lceil \mathscr{B}(M_j)/\mathscr{B}(M_1) \rceil \mathscr{B}(M_1)$. This might simplify the management of disk space to: (1) avoid its fragmentation, and (2) ensure the contiguous layout of each block of an object. However, MBS might waste disk bandwidth by forcing the disk to (1) retrieve more data on behalf of a stream per time period due to rounding up of block size, and (2) remain idle during other time periods to avoid an overflow of memory. These are best illustrated using an example. Assume two media types, MPEG-1 and MPEG-2 objects, with bandwidth requirements of 1.5 Mbps and 4 Mbps, respectively. With this approach, the block size of the system is chosen based on MPEG-1 objects. Assume, it is chosen to be 512 kbyte; $\mathscr{B}$(MPEG-1) = 512 kbyte. This implies that $\mathscr{B}$(MPEG-2) = 1365.33 kbytes. MBS would increase $\mathscr{B}$(MPEG-2) to equal 1536 kbytes. To avoid an excessive amount of accumulated data at a client displaying an MPEG-2 clip, the scheduler might skip the retrieval of data one time period every nine time periods. The scheduler may not employ this idle slot to service another request because it is required during the next time period to retrieve the next block of current MPEG-2 display. If all active requests are MPEG-2 video clips and a time period supports nine displays with $\mathscr{B}$(MPEG-2) = 1536 kbytes, then with $\mathscr{B}$(MPEG-2) = 1365.33 kbytes, the system would support ten simultaneous displays (10% improvement in performance). In summary, the block size for a media type should approximate its theoretical value in order to minimize the percentage of wasted disk bandwidth.

The final approach, EVEREST, employs the buddy algorithm to approximate a contiguous layout of a file on the disk without wasting disk space. The number of contiguous chunks that constitute a file is a fixed function of the file size and the configuration of the buddy algorithm. Based on this information, the system can either (1) prevent a block from overlapping two noncontiguous chunks or (2) allow a block to overlap two chunks and require the client to cache enough data to hide the seeks associated with the retrieval of these blocks. Until now, we assumed the first approach. To illustrate the second approach, if a file consists of five contiguous chunks, then at most, four blocks of this file might span two different chunks. This implies that the retrieval of four blocks will incur seeks with at most one seek per block retrieval. To avoid hiccups, the scheduler should delay the display of the data at the client until it has cached enough data to hide the latency associated with four seeks. The amount of cached data is not significant. For example, assuming a maximum seek time of 20 ms, with MPEG-2 objects (4 Mbps), the client should cache 10 kbytes to hide each seek. However, this approach complicates the admission control policy because the retrieval of a block might incur either one or zero seeks.

With EVEREST, the basic unit of allocation is a page, the size of a page has no impact on the granularity at which a process might read a section; this is detailed below, also termed a *section* of height 0. EVEREST organizes these sec-

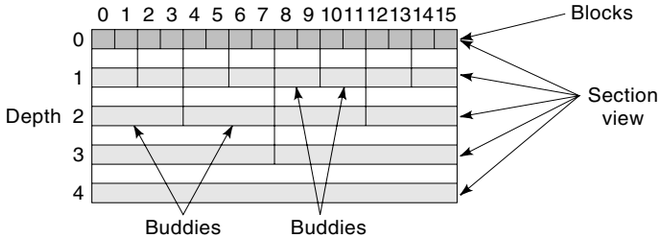

Figure 16. Physical division of disk space into pages and the corresponding logical view of the sections with an example base of $\omega = 2$.

tions as a tree to form larger, contiguous sections. As illustrated in Fig. 16, only sections of size(page) $\times\ \omega^i$ (for $i \geq 0$) are valid, where the base ω is a system configuration parameter. If a section consists of ω^i pages, then i is said to be the height of the section. ω height i sections that are buddies (physically adjacent) might be combined to construct a height $i + 1$ section.

To illustrate, the disk in Fig. 16 consists of 16 pages. The system is configured with $\omega = 2$. Thus, the size of a section may vary from 1, 2, 4, 8, up to 16 pages. In essence, a binary tree is imposed upon the sequence of pages. The maximum height, computed by

$$S = \left\lceil \log_\omega \left(\left\lfloor \frac{Capacity}{\text{size(page)}} \right\rfloor \right) \right\rceil$$

is 4. To simplify the discussion, assume that the total number of pages is a power of ω. The general case can be handled similarly and is described below.

With this organization imposed upon the device, sections of height $i \geq 0$ cannot start at just any page number, but only at offsets that are multiples of ω^i. This restriction ensures that any section, with the exception of the one at height S, has a total of $\omega - 1$ adjacent buddy sections of the same size at all times. With the base 2 organization of Fig. 16, each section has one buddy.

With EVEREST, a portion of the available disk space is allocated to objects. The remainder, should any exist, is free. The sections that constitute the available space are handled by a free list. This free list is actually maintained as a sequence of lists, one for each section height. The information about an unused section of height i is enqueued in the list that handles sections of that height. In order to simplify object allocation, the following bounded list length property is always maintained: For each height $i = 0, \ldots, S$, at most $\omega - 1$ free sections of i are allowed. Informally, this property implies that whenever there exists sufficient free space at the free list of height i, EVEREST must compact these free sections into sections of a larger height. A lazy variant of this scheme would allow these lists to grow longer and do compaction upon demand, that is, when large contiguous pages are required. This would be complicated as a variety of choices might exist when merging pages. This would require the system to employ heuristic techniques to guide the search space of this merging process. However, to simplify the description, we focus on an implementation that observes the invariant described above.

The materialization of an object is as follows. The first step is to check whether the total number of pages in all the sections on the free list is either greater than or equal to the number of pages (denoted no-of-pages(o_x)) that the new object o_x requires. If this is not the case, then one or more victim objects are elected and deleted. (The procedure for selecting a victim is based on heat (43). The deletion of a victim object is described further below.) Assuming enough free space is available at this point, o_x is divided into its corresponding sections as follows. First, the number m = no-of-pages(o_x) is converted to base ω. For example, if $\omega = 2$, and no-of-pages(o_x) = 13_{10}, then its binary representation is 1101_2. The full representation of such a converted number is $m = d_{j-1} \times \omega^{j-1} + \cdots + d_2 \times \omega^2 + d_1 \times \omega^1 + d_0 \times \omega^0$. In our example, the number 1101_2 can be written as $1 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$. In general, for every digit d_i that is non-zero, d_i sections are allocated from height i of the free list on behalf of o_x. In our example, o_x requires 1 section from height 0, no sections from height 1, 1 section from height 2, and 1 section from height 3.

For each object, the number ν of contiguous pieces is equal to the number of ones in the binary representation of m, or with a general base ω, $\nu = \sum_{i=0}^{j} d_i$ (where j is the total number of digits). Note that ν is always bounded by $\omega \lceil \log_\omega m \rceil$. For any object, ν defines the maximum number of sections occupied by the object. (The minimum is 1 if all ν sections are physically adjacent.) A complication arises when no section at the right height exists. For example, suppose that a section of size ω^i is required, but the smallest section larger than ω^i on the free list is of size ω^j ($j > i$). In this case, the section of size ω^j can be split into ω sections of size ω^{j-1}. If $j - 1 = i$, then $\omega - 1$ of these are enqueued on the list of height i, and the remainder is allocated. However if $j - 1 > i$, then $\omega - 1$ of these sections are again enqueued at level $j - 1$, and the splitting procedure is repeated on the remaining section. It is easy to see that, whenever the total amount of free space on these lists is sufficient to accommodate the object, then for each section that the object occupies, there is always a section of the appropriate size, or larger, on the list. This splitting procedure will guarantee that the appropriate number of sections, each of the right size, will be allocated, and that the bounded list length property is never violated.

When the system elects that an object must be materialized and there is insufficient free space, then one or more victims are removed from the disk. Reclaiming the space of a victim requires two steps for each of its sections. First, the section must be appended to the free list at the appropriate height. The second step ensures that the bounded list length property is not violated. Therefore, whenever a section is enqueued in the free list at height i, and the number of sections at that height is equal to or greater than ω, then ω sections must be combined into one section at height $i + 1$. If the list at $i + 1$ now violates bounded list length property, then once again, space must be compacted and moved to section $i + 2$. This procedure might be repeated several times. It terminates when the length of the list for a higher height is less than ω.

Compaction of ω free sections into a larger section is simple when they are buddies; in this case, the combined space is already contiguous. Otherwise, the system might be forced to exchange one occupied section of an object with one on the free list in order to ensure contiguity of an appropriate sequence of ω sections at the same height. The following algo-

rithm achieves space-contiguity among ω free sections at height i.

1. Check if there are at least ω sections for height i on the free list. If not, stop.
2. Select the first section (denoted s_j) and record its page number (i.e., the offset on the disk drive). The goal is to free $\omega - 1$ sections that are buddies of s_j.
3. Calculate the page-numbers of s_j's buddies. EVEREST's division of disk space guarantees the existence of $\omega - 1$ buddy sections physically adjacent to s_j.
4. For every buddy s_k, $k \leq 0 \leq \omega - 1$, $k \neq j$; if it exists on the free list, then mark it.
5. Any of the s_k unmarked buddies currently store parts of other object(s). The space must be rearranged by swapping these s_k sections with those on the free list. Note that for every buddy section that should be freed, there exists a section on the free list. After swapping space between every unmarked buddy section and a free list section, enough contiguous space has been acquired to create a section at height $i + 1$ of the free list.
6. Go back to Step 1.

To illustrate, consider the organization of space in Fig. 17(a). The initial set of disk resident objects is $\{o_1, o_2, o_3\}$, and the system is configured with $\omega = 2$. In Fig. 17a, two sections are on the free list at height 0 and 1 (addresses 7 and 14, respectively), and o_3 is the victim object that is deleted. Once page 13 is placed on the free list in Fig. 17(b), the number of sections at height 0 is increased to ω, and it must be compacted according to Step 1. As sections 7 and 13 are not contiguous, section 13 is elected to be swapped with section 7s buddy, that is, section 6 [Fig. 17(c)]. In Fig. 17(d), the data of section 6 is moved to section 13, and section 6 is now on the free list. The compaction of sections 6 and 7 results in a new section with address 6 at height 1 of the free list. Once again, a list of length two at height 1 violates the bounded list length property, and pages (4, 5) are identified as the buddy of section 6 in Fig. 17(e). After moving the data in Fig. 17(f) from pages (4, 5) to (14, 15), another compaction is performed with the final state of the disk space emerging as in Fig. 17(g).

Once all sections of a deallocated object are on the free list, the iterative algorithm above is run on each list, from the lowest to the highest height. The previous algorithm is somewhat simplified because it does not support the following scenario: a section at height i is not on the free list; however, it has been broken down to a lower height (say $i - 1$), and not all subsections have been used. One of them is still on the free list at height $i - 1$. In these cases, the free list for height $i - 1$ should be updated with care because those free sections have moved to new locations. In addition, note that the algorithm described above actually performs more work than is strictly necessary. A single section of a small height, for example, may end up being read and written several times as its section is combined into larger and larger sections. This is eliminated in the following manner. The algorithm is first performed virtually—that is, in main memory, as a compaction algorithm on the free lists. Once completed, the entire sequence of operations that have been performed determine the ultimate destination of each of the modified sections. The scheduler constructs a list of these sections. This list is inserted into a queue of house keeping I/Os. Associated with each element of the queue is an estimated amount of time required to perform the task. Whenever the scheduler locates one or more idle slots in the time period, it analyzes the queue of work for the element that can be processed using the available time.

The value of ω impacts the frequency of preventive operations. If ω is set to its minimum value (i.e., $\omega = 2$), then preventive operations would be invoked frequently because every time a new section is enqueued, there is a 50% chance for a height of the free list to consist of two sections (violates the bounded list length property). Increasing the value of ω will, therefore, relax the system because it reduces the probability that an insertion to the free list would violate the bounded list length property. However, this would increase the expected number of bytes migrated per preventive operation. For example, at the extreme value of $\omega = n$ (where n is the total number of pages), the organization of blocks will consist of two levels, and for all practical purpose, EVEREST reduces to a standard file system that manages fix-sized pages.

The design of EVEREST suffers from the following limitation: the overhead of its preventive operations may become significant if many objects are swapped in and out of the disk drive. This happens when the working set of an application cannot become resident on the disk drive.

In a real implementation of EVEREST, it might not be possible to fix the number of disk pages as an exact power of ω. The most important implication of an arbitrary number of pages is that some sections may not have the correct number of buddies ($\omega - 1$ of them). However, we can always move those sections to one end of the disk—for example, to the side with the highest page-offsets. Then, instead of choosing the first section in Step 2 in the object deallocation algorithm, the system can choose the one with the lowest page number. This ensures that the sections towards the critical end of the disk that might not have the correct number of buddies are never used in both Steps 4 and 5 of the algorithm.

In (40) we report on an implementation of EVEREST in our CM server. Our implementation enables a process to retrieve a file using block sizes that are at the granularity of 1/2 kbyte. For example, EVEREST might be configured with a 64 kbyte page size. One process might read a file at the granularity of 1365.50 kbyte blocks, while another might read a second file at the granularity of 512 kbyte.

The design of EVEREST is related to the buddy system proposed in (44–45) for an efficient main memory storage allocator (DRAM). The difference is that EVEREST satisfies a request for b pages by allocating a number of sections such that their total number of pages equals b. The storage allocator algorithm, on the other hand, will allocate one section that is rounded up to $2^{\lceil lgb \rceil}$ pages, resulting in fragmentation and motivating the need for either a reorganization process or a garbage collector (39). The primary advantage of the elaborate object deallocation technique of EVEREST is that it avoids internal and external fragmentation of space as described for traditional buddy systems [see (39)].

PIRATE. Upon the retrieval of a tertiary resident object (say Z), if the storage capacity of the disk drive is exhausted, then the system must replace one or more objects (victims) in order to allow Z to become the disk resident. Previous approaches (collectively termed Atomic) replace each of the vic-

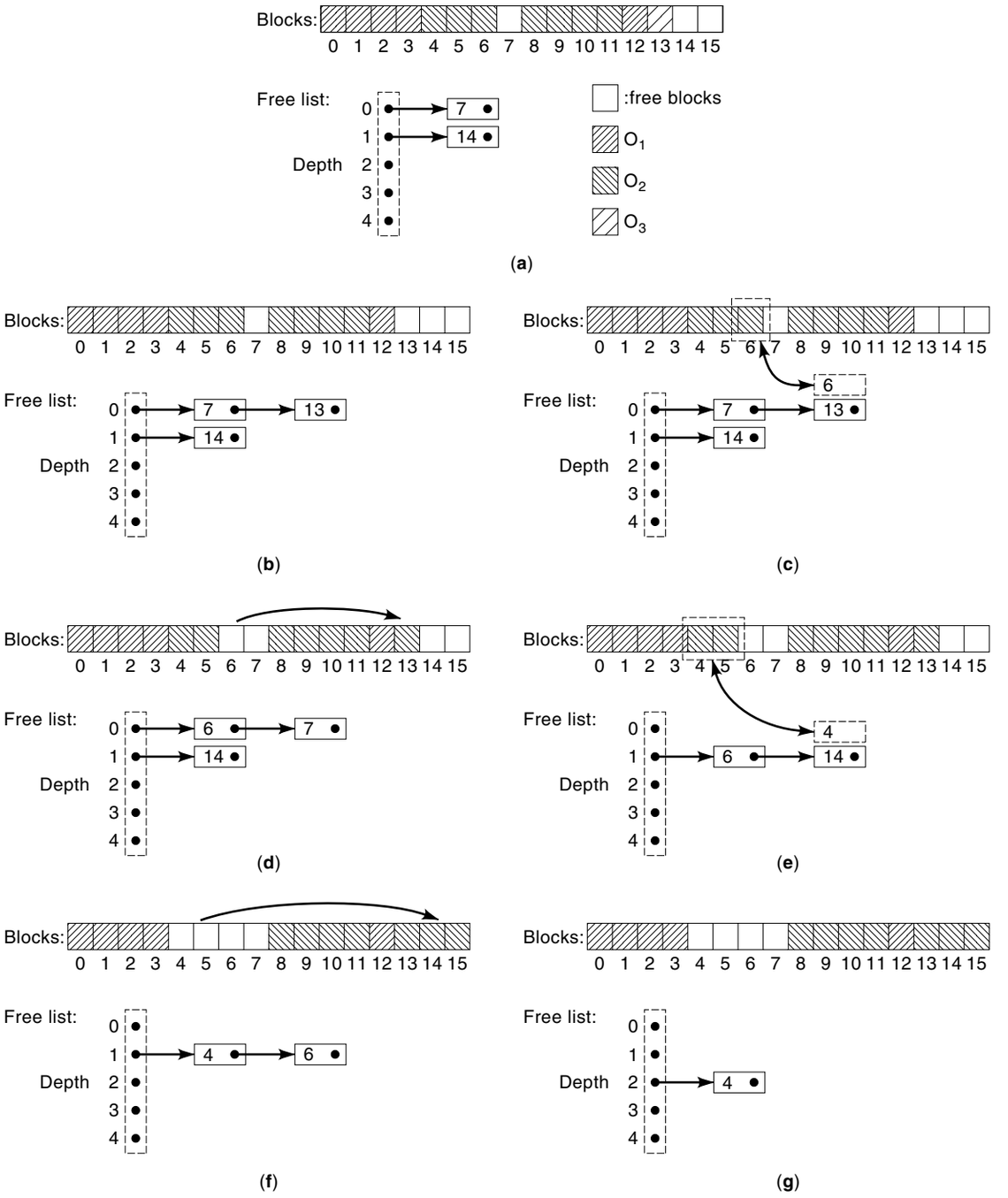

Figure 17. Deallocation of an object. The example sequence shows the removal of object o_3 from the initial disk resident object set $\{o_1, o_2, o_3\}$. Base two, $\omega = 2$. (a) Two sections are on the free list already (7 and 14), and object o_3 is deallocated. (b) Sections 7 and 13 should be combined; however, they are not contiguous. (c) The buddy of section 7 is 6. Data must move from 6 to 13. (d) Sections 6 and 7 are contiguous and can be combined. (e) The buddy of section 6 is 4. Data must move from (4, 5) to (14, 15). (f) Sections 4 and 6 are now adjacent and can be combined. (g) The final view of the disk and the free list after removal of o_3.

tim objects in their entirety, requiring each object to either be completely a disk resident or not disk resident at all. With PartIal ReplAcement TEchnique (PIRATE) (12), the system chooses a larger number of objects as victims; however, it replaces a portion of each of its victims in order to free up sufficient space for the blocks of Z. The input to PIRATE include (1) the size and frequency of access to each object X in the database, termed size(X) and heat(X) respectively (46), (2) a set of objects with a disk resident fraction except Z (denoted $\mathscr{T}$), and (3) the size of the object referenced by the pending request (size(Z)). Its side effect is that it makes enough of the disk space available to accommodate Z.

PIRATE deletes blocks of an object one at a time, starting with those that constitute the tail end of the object. For example, if PIRATE decides to replace those blocks that constitute 10 min of a 30 min video clip, it deletes those blocks that represent the last 10 min of the clip, leaving the first 20 min disk resident. The number of blocks that constitute the first portion of X is denoted DISK(X), while its deleted (non disk resident) blocks are termed ABSENT(X); ABSENT(X) = size(X) − DISK(X). (Note: the granularity of ABSENT(X), size(X), and DISK(X) are in blocks.) PIRATE complements pipelining because by keeping the head of the objects disk resident, their displays can start immediately minimizing the observed startup latency time. On the other hand, since all the requests will access the tertiary, it is possible that the tertiary becomes the bottleneck. Hence, PIRATE is suitable for single user environments (such as Personal Computers (PC) (12) or when the sustained bandwidth of the tertiary is higher than the average load imposed by simultaneous requests.

Formal Statement of the Problem. The portion of disk space allocated to continuous media data types consists of C blocks. The database consists of m objects $\{o_1, \ldots, o_m\}$, with heat(o_j) $\in (0, 1)$ satisfying $\sum_{j=1}^{m}$ heat(o_j) = 1, and sizes size(o_j) $\in (0, C)$ for all $1 \le j \le m$. The size of the database exceeds the storage capacity of the system (i.e., $\sum_{j=1}^{m}$ size(o_j) > C). Consequently, the database resides permanently on the tertiary storage device, and objects are swapped in and out from the disk. We assume that the size of each object is smaller than the storage capacity of the disk drive, size(o_j) < C for $1 \le j \le m$. Moreover, to simplify the discussion, we assume that the tertiary is not required to change tapes/platters or reposition its read head once it starts to transfer an object. Assume a process that generates requests for objects in which object o_j is requested with probability heat(o_j) (all independent). We assume no advance knowledge of the possible permutation of requests for different objects.

Let $\mathscr{T}$ denote the set of objects with a disk resident fraction except the one that is referenced by the pending request, size($\mathscr{T}$) = $\sum_{x \in \mathscr{T}}$ DISK(X). Moreover, assuming a new request arrives referencing object Z ($\mathscr{T} \leftarrow \mathscr{T} - \{Z\}$), we define *free_disk_space* as C − (size($\mathscr{T}$) + DISK(Z)). If ABSENT(Z) ≤ *free_disk_space,* then no replacement is required. In this study, we focus on the scenario where replacement is required; that is, ABSENT(Z) > *free_disk_space.*

We define latency time observed by a request referencing Z ($\ell(Z)$) as the amount of time elapsed from the arrival of the request to the onset of the display. It is a function of DISK(Z) and B_{Tertiary}. If DISK(Z) = size($S_{Z,1}$), then the maximum value for $\ell(Z)$ is the worst reposition time of the tertiary storage device. One may reduce this latency time to zero by incrementing size($S_{Z,1}$) with the amount of data corresponding to this time; that is,

$$\frac{\textit{worst reposition time} \times B_{\text{Display}}}{\textit{size}(\textit{block})}$$

(This optimization is assumed for the rest of this paper.) If DISK(Z) > size($S_{Z,1}$), then ($\ell(Z) = 0$ (due to assumed optimization). Otherwise (i.e., DISK(Z) < size($S_{Z,1}$)), the system determines the starting address of the nondisk resident portion of Z (*missing*), and $\ell(Z)$ is defined as the total sum of (1) the repositioning of tertiary to the physical location corresponding to missing, and (2) the materialization time of the remainder of the first slice,

$$\frac{\textit{size}(\textit{block})}{B_{\text{Tertiary}}} \times (\text{size}(S_{Z,1}) - \text{DISK}(Z))$$

The average (expected value of) latency as a function of requests can be defined as

$$\mu = \sum_{x} \text{heat}(x) * \ell(x) \tag{18}$$

The variance is

$$\sigma^2 = \sum_{x} \text{heat}(x) * (\ell(x) - \mu)^2 \tag{19}$$

By deleting a portion of an object, we may increase its latency time resulting in a higher μ and σ^2. However, once the disk capacity is exhausted, deletion of an object is unavoidable. In this case, it is desired for some x in $\mathscr{T}$ to reduce DISK(x) such that enough disk space becomes available to render object Z disk resident in its entirety. The problem is how to determine those x and their corresponding fractions to be deleted to minimize both the average latency time and its variance. Unfortunately, minimizing the average latency time might increase the variance and vice versa. In the next section, we present simple PIRATE and demonstrate that it minimizes the average latency. Subsequently, extended PIRATE is introduced, as a generalization of simple PIRATE, with a knob that can be adjusted by the user to tailor the system to strike a compromise between these two objectives.

Simple PIRATE. Figure 18 presents simple PIRATE. Logically, it operates in two passes. In the first pass, it deletes from those objects (say i) whose disk resident portion is greater than the size of their first slice ($S_{i,1}$). By doing so, it can ensure a zero latency time for requests that reference these objects in the future (by employing the pipelining mechanism). Note that PIRATE deletes objects at a granularity of a block. Moreover, it frees up only sufficient space to accommodate the pending request and no more than that. (For example, if ABSENT(Z) is equivalent to size(X)/10, and X is chosen as the victim, then only the blocks corresponding to the last 1/10 of X are deleted in order to render Z disk resident.)

If the disk space made available by the first pass is insufficient, then simple PIRATE enters its second pass. This pass deletes objects starting with the one that has the lowest heat (following the greedy strategy suggested for the fractional knapsack problem (47). One might argue that a combination of heat and size should be considered when choosing victims. However, ABSENT(Z) blocks (where Z is the object required to

```
define pfs: potential_free_space,
define rds: required_disk_space
rds ← ABSENT(Z) − free_disk_space
repeat
  victim ← object i from set 𝒯 with:
        1) the lowest heat, and
        2) DISK(i) > Size(S_i,1)
 if (victim is NOT null) then
   pfs ← DISK(victim) − size(S_victim,1)
 else
   victim ← object i from set 𝒯
              with the lowest heat
   pfs ← DISK(victim)
 if (pfs > rds) then
   DISK(victim) ← DISK(victim) − rds
   rds ← 0
 else
   DISK(victim) ← DISK(victim) − pfs
   rds ← rds − pfs
until (rds = 0)
```

Figure 18. Simple PIRATE.

become disk resident) are required to be deleted from the disk, independent of the size of the victims. The following proof formalizes this statement and proves the optimality of simple PIRATE in minimizing the latency time of the system.

Lemma 1: To minimize the average latency time of the system, during pass 2, PIRATE must delete those blocks corresponding to the object with the lowest heat, independent of the object sizes.

Proof: Without loss of generality, assume $\mathscr{T} = \{X, Y\}$, $size(block) = 1$ and $B_{\text{Tertiary}} = 1$. Assume a request arrives at t_0 referencing Z, and the disk capacity is exhausted. Let t_1 be the time when a portion of X and/or Y is deleted. We define μ_0 (μ_1) to be the average latency at t_0 (t_1), see Eq. 18. Subsequently, $\text{DISK}_0(i)$ and $\text{DISK}_1(i)$ represent the disk resident fraction of object i at time t_0 and t_1, respectively. Let δ_i denote the number of blocks of object i deleted from disk at time t_1 (i.e., $\delta_i = \text{DISK}_0(i) - \text{DISK}_1(i)$). By deleting X and/or Y partially, we increase the average latency by Δ. However, since deletion is unavoidable, the objective is to minimize $\Delta = \mu_1 - \mu_0$, while $\text{ABSENT}(Z) = \delta_X + \delta_Y$ (In computing the average latency, we ignore the latency of the other objects in the database as well as the repositioning time of the tertiary. This is because it only adds a constant to both μ_0 and μ_1 which will be eliminated by the computation of Δ).

$$\begin{aligned}\mu_0 &= \text{heat}(X) * (\text{size}(S_{X,1}) - \text{DISK}_0(X)) + \\ &\quad \text{heat}(Y) * (\text{size}(S_{Y,1}) - \text{DISK}_0(Y)) \\ \mu_1 &= \text{heat}(X) * (\text{size}(S_{X,1}) - \text{DISK}_1(X)) + \\ &\quad \text{heat}(Y) * (\text{size}(S_{Y,1}) - \text{DISK}_1(Y))\end{aligned}$$

Thus,

$$\begin{aligned}\Delta &= \text{heat}(X) * \delta_X + \text{heat}(Y) * \delta_Y \\ &= \text{heat}(X) * \delta_X + \text{heat}(Y) * (\text{ABSENT}(Z) - \delta_X) \\ &= \text{heat}(Y) * \text{ABSENT}(Z) + \\ &\quad \delta_X * (\text{heat}(X) - \text{heat}(Y))\end{aligned}$$

Since $\text{heat}(Y) * \text{ABSENT}(Z)$ and $(\text{heat}(X) - \text{heat}(Y))$ are constants, in order to minimize Δ, we can only vary δ_X (this impacts δ_Y because $\text{ABSENT}(Z) = \delta_X + \delta_Y$). If $\text{heat}(X) > \text{heat}(Y)$, then $(\text{heat}(X) - \text{heat}(Y))$ is a positive value; hence, in order to minimize Δ, the value of δ_X should be minimized (i.e., the object with higher $\text{heat}(X)$ should not be replaced). On the other hand, if $\text{heat}(X) < \text{heat}(Y)$, then $(\text{heat}(X) - \text{heat}(Y))$ is a negative value; hence, in order to minimize Δ, the value of δ_X should be maximized (i.e., the object with lower $\text{heat}(X)$ should be replaced). This demonstrates that the amount of data deleted from victim(s) (δ_i) in order to free up disk space depends only on heat (i) and not size (i).

Extended PIRATE. Extended PIRATE is a generalization of simple PIRATE that can be customized to strike a compromise between the two goals to minimize either the average latency time of the system or the variance in the latency time. The major difference between simple and extended PIRATE is as follows: Extended PIRATE (see Fig. 19) requires a minimum fraction (termed $\text{LEAST}(x)$) of the most frequently accessed objects to be the disk resident. Logically, extended PIRATE operates in three passes. Its first pass is identical to that of simple PIRATE. If this pass fails to provide sufficient disk space for the referenced object, then during the second pass, it deletes from objects until each of their disk resident portion corresponds to $\text{LEAST}(x)$. If pass two fails (i.e., provides insufficient space to materialize the referenced object), it enters pass 3. This pass is identical to pass 2 of simple PIRATE where objects are deleted in their entirety starting with the one that has the lowest heat.

With extended PIRATE, $\text{LEAST}(X)$ for each disk resident object is defined as follows:

$$\begin{aligned}&\text{LEAST}(X) = \\ &min(\lceil knob * \text{heat}(X) * \text{size}(S_{X,1})\rceil, \text{size}(S_{X,1}))\end{aligned} \tag{20}$$

```
define pfs: potential_free_space,
define rds: required_disk_space
rds ← ABSENT(Z) − free_disk_space
repeat
  victim ← object i from set 𝒯 with:
        1) the lowest heat, and
        2) DISK(i) > Size(S_i,1)
 if (victim is NOT null) then
   pfs ← DISK(victim) − size(S_victim,1)
 else
   victim ← object i from set 𝒯 with:
        1) the lowest heat, and
        2) DISK(i) > LEAST(i)
   if (victim is NOT null) then
     pfs ← DISK(victim) − LEAST(victim)
   else
     victim ← object i from set 𝒯
                with the lowest heat
     pfs ← DISK(victim)
 if (pfs > rds) then
   DISK(victim) ← DISK(victim) − rds
   rds ← 0
 else
   DISK(victim) ← DISK(victim) − pfs
   rds ← rds − pfs
until (rds = 0)
```

Figure 19. Extended PIRATE.

where *knob* is an integer whose lower bound is zero. The minimum function avoids the size of LEAST(x) to exceed the size of the first slice. When $knob = 0$, extended PIRATE is identical to simple PIRATE. As *knob* increases, a larger portion of each object becomes disk resident. Obviously, the ideal case is to increase the knob until the first slice of all the objects become disk resident. However, due to the limited storage capacity, this might be infeasible. By increasing the knob, we force a portion of some objects with lower heat to remain disk resident, at the expense of deleting from objects with a high heat. By providing each request referencing an object a latency time proportional to the heat of that object, extended PIRATE improves the variance while not increasing the average dramatically.

There is an optimal value for *knob* that minimizes σ^2. If the value of *knob* exceeds this value, then σ^2 starts to increase also.

Lemma 2: The optimal value for *knob* is C/Aug_Slice1.

Proof: Let U be the total number of unique objects that are referenced over a period of time. We define

$$Avg\_Slice1 = \sum_x \text{heat}(x) * \text{size}(S_{x,1})$$

$$Avg\_Heat = \frac{\sum_x \text{heat}(x)}{U} = \frac{1}{U}$$

$$Avg\_Least = knob * Avg\_Heat * Avg\_Slice1$$

The ideal case is when the LEAST of almost all the objects that constitute the database are disk resident (C is the total number of disk blocks):

$$C \approx U * Avg\_Least$$

$$\approx U * knob * \frac{1}{U} * Avg\_Slice1$$

Solving for *knob,* we obtain: $knob \approx C/Avg\_Slice1$.

Substituting the optimal value of *knob* in Eq. 20, we obtain $\text{LEAST}(X) = \text{heat}(X) \times \text{size}(S_{X,1})/\sum_i^m \text{heat}(i) \times \text{size}(S_{i,1}) \times C$. This is intuitively the amount of disk space an object X deserves. In (12), we employed a simulation study to confirm this analytical result. Note that because heat is considered in the computation of LEAST, with $knob = C/Avg\_Slice1$, the average latency time degrades proportional to the improvement in variance. In summary, when $knob = 0$, PIRATE replaces objects in a manner that minimizes average latency time. However, when $knob = C/Avg\_Slice1$, it minimizes the variance. To observe, consider the following discussion.

In the long run, with $knob = 0$, PIRATE maintains the first slice of all the objects with the highest heat disk resident, while the others compete with each other for a small portion of the disk space [see Fig. 20(a)]. To approximate the number of objects that become disk resident with $knob = 0$ ($\aleph_\mu$), we use the average size (Avg_Slice1) as follows:

$$\aleph_\mu \approx \frac{C}{Avg\_Slice1} \tag{21}$$

However, with $knob = C/Avg\_Slice1$, PIRATE maintains only a minimum portion of all these $\aleph_\mu$ objects disk resident.

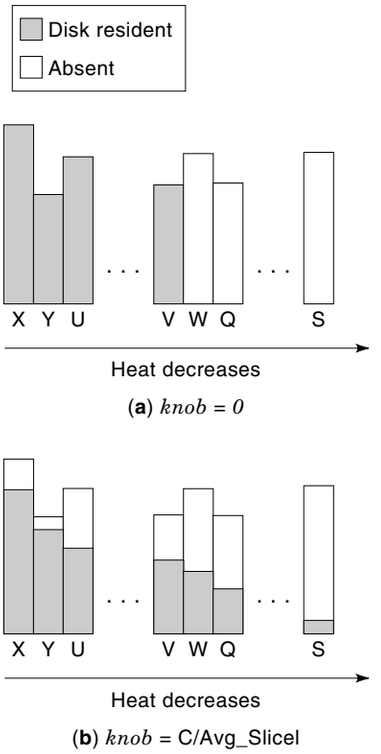

Figure 20. Status of the first slice of objects.

To achieve this, in an optimal case, it requires $\aleph_\mu * Avg\_Least = \aleph_\mu * C/U$ of disk space. This is optimistic, because the $\aleph_\mu$ objects have the highest heats, thus a large minimum portion. While it is not realistic to use *Avg_Least* in the equation, it is useful for approximation. The rest of the disk space, $C - (\aleph_\mu * C/U)$, can be used for the minimum portion of the other objects [see Fig. 20(b)]. Therefore, in the long run, the number of disk resident objects with $knob = C/Avg\_Slice1$ (i.e., $C/Max\_Size * Max\_Heat * knob < \aleph_\sigma < U$) is larger than $\aleph_\mu$. However, with $knob = 0$, the first slice of $\aleph_\mu$ objects are disk resident, while with $knob = C/Avg\_Slice1$, only LEAST of each of the $\aleph_\sigma$ objects are disk resident. This results in the following tradeoff. On one hand, a request referencing an object Z has a higher hit ratio with $knob = C/Avg\_Slice1$ as compared to $knob = 0$. On the other hand, a hit with $knob = 0$ translates into a fraction of a second latency time, while with $knob = C/Avg\_Slice1$, it results in a minimum latency time of $(\text{size}(S_{Z,1}) - \text{LEAST}(Z)) \times size(block)/B_{\text{Tertiary}}$. This explains why with $knob = C/Avg\_Slice1$, PIRATE improves the variance proportional to the degradation in average latency.

STREAM SCHEDULING AND SYNCHRONIZATION

In this section, we investigate a taxonomy of scheduling problems corresponding to three classes of multimedia applications. The application classes and the corresponding scheduling problems include

1. On-demand atomic object retrieval: With this class of applications, a system strives to display an object (audio

or video) as soon as a user request arrives referencing the object. The envisioned movie-on-demand and news-on-demand systems are examples of this application class. We formalize the scheduling problem that represents this class as the Atomic Retrieval Scheduling (ARS) problem.

2. Reservation-based atomic object retrieval: This class is similar to on-demand atomic object retrieval except that a user requests the display of an object at some point in the future. An example might be a movie-on-demand system where the customers call to request the display of a specific movie at a specific time; for example, Bob calls in the morning to request a movie at 8:00 p.m. Reservation-based retrieval is expected to be cheapter than on-demand retrieval because it enables the system to minimize the amount of resources required to service requests (using planning optimization techniques). The scheduling problem that represents this application class is termed Augmented ARS (ARS^+).
3. On-demand composite object retrieval: As compared to atomic objects, a composite object describes when two or more atomic objects should be displayed in a temporarily related manner (48). To illustrate the application of composite objects, consider the following environment. During the post-production of a movie, a sound editor accesses an archive of digitized audio clips to extend the movie with appropriate sound-effects. The editor might choose two clips from the archive: a gun-shot and a screaming sound effect. Subsequently, she authors a composite object by overlapping these two sound clips and synchronizing them with the different scenes of a presentation. During this process, she might try alternative gun-shot or screaming clips from the repository to evaluate which combination is appropriate. To enable her to evaluate her choices immediately, the system should be able to display the composition as soon as it is authored (on-demand). There are two scheduling problems that correspond to this application class: (1) Composite Retrieval Scheduling (CRS), and (2) Resolving Internal Contention (RIC). While CRS is the scheduling of multiple composite objects assuming a multi-user environment, RIC is the scheduling of a single composite object assuming a single request. RIC can also be considered as a preprocessing step of CRS when constructing a composite object for each user, in a multi-user environment.

CRS and RIC pose the most challenging problems and are supersets of ARS and ARS^+. Researchers are just starting to realize the importance of customized multimedia presentations targeted toward the individual's preferences (49). Some studies propose systems that generate the customized presentations automatically based on a user query (50). Independent of the method employed to generate a presentation, the retrieval engine must respect the time dependencies between the elements of a presentation when retrieving the data. To tackle CRS and RIC, we frist need to solve the simpler problems of ARS and ARS^+.

To put our work in perspective, we denote the retrieval of an object as a task. The distinctive characteristics of the scheduling problems (ARS, ARS^+, CRS, RIC) that distinguish them from the conventional scheduling problems a more detailed comparison can be found in (50) are (1) tasks are IO-bound and not CPU-bound, (2) each task utilizes multiple disks during its life time, (3) each task acquires and releases disks in a regular manner, (4) the pattern utilized by a task to employ the disks depends on the placement of its referenced object on the disks, and (5) there might be temporal relationships among multiple tasks constituting a composite task.

Independent of the application classes and due to the above characteristics, tasks compete for system resources, that is, disk bandwidth. The term retrieval contention is used in this study to specify this competition among the tasks for the disk bandwidth. This contention should be treated differently depending on alternative types of the system load. Furthermore, an admission control component, termed contention prediction (cop), is required by all the scheduling algorithms to activate tasks in such a manner that no contention occurs among the activated tasks. Indeed, the problem of retrieval contention and its prediction are shared by all the scheduling problems and should be studied separately.

In this section, we first extend the hardware architecture to support a mix of media types. Note that the terms time interval and time period are used synonymously in this section. Subsequently, we study the formally defined four scheduling problems. In (51), we proved that all of these scheduling problems are *NP*-hard in strong sense. In addition, scheduling algorithms based on heuristics are introduced per problem in (51).

Mix of Media Types

Until now, we assumed that all CM objects belong to a single media type. For example, all are MPEG-2 compressed video objects. In practice, however, objects might belong to different media types. Assuming m media types, each with a bandwidth requirement of c_i, the display time of each block of different objects must be identical in order to maintain the fixed-size time intervals (and a continuous display for a mix of displays referencing different objects). This is achieved as follows. First, objects are logically grouped based on their media types. Next, the system chooses media type i with block size Sub_i and bandwidth requirement c_i as a reference to compute the duration of a time interval ($interval = Sub_i/c_i$). The block size of a media type j is chosen to satisfy the following constraint: $interval = Sub_j/c_j = Sub_i/c_i$.

Defining Tasks

Let $\mathcal{T}$ be a set of tasks where each $t \in \mathcal{T}$ is a retrieval task that corresponds to the retrieval of a video object. Note that if an object X is referenced twice by two different requests, a different task is assigned to each request. The time to display a block is defined as a *time interval* (or time period) that is employed as the time unit.

For each task $t \in \mathcal{T}$, we define

- $r(t)$: Release time of t, $r{:}\mathcal{T} \rightarrow N$. The time that ts information (i.e., the information of its referenced object) becomes available to the scheduler. This is identical to the time that a request referencing the object is submitted to the system.

- $\ell(t)$: Length (size of the object referenced by t, $\ell: \mathcal{T} \rightarrow N$. The unit is in number of blocks.
- $c(t)$: Consumption rate of the object referenced by t, $0 < c(t) \leq 1$. This rate is normalized by R_D. Thus, $c(t) = 0.40$ means that the consumption rate of the object referenced by t is 40% of R_D, the cluster bandwidth.
- $p(t)$: The cluster that contains the first block of the object referenced by t, $1 \leq p(t) \leq D$. It determines the placement of the object referenced by t on the clusters.

We denote a task t_i as a quadruple: $\langle r(t_i), \ell(t_i), c(t_i), p(t_i) \rangle$.

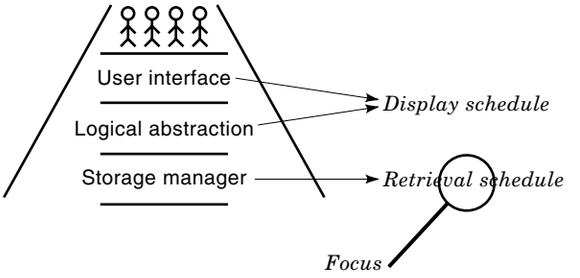

Figure 21. Three levels of abstraction.

Atomic Retrieval Scheduling

The ARS problem is to schedule retrieval tasks such that the total bandwidth requirement of the scheduled tasks on each cluster during each interval does not exceed the bandwidth of that cluster (i.e., no retrieval contention). Moreover, ARS should satisfy an optimization objective. Depending on the application, this objective could be minimizing either (1) the average startup latency of the tasks, or (2) the total duration of scheduling for a set of tasks (maximizing throughput). Movie-on-demand is one sample application.

Definition 1: The problem of ARS is to find a schedule σ (where $\sigma: \mathcal{T} \rightarrow N$) for a set $\mathcal{T}$, such that (1) it minimizes the finishing time (52) w, where w is the least time at which all tasks of $\mathcal{T}$ have been completed, and (2) satisfies the following constraints:

- $\forall t \in \mathcal{T}\ \sigma(t) \geq r(t)$.
- $\forall u \geq 0$, let $S(u)$ be the set of tasks which $\sigma(t) \leq u < \sigma(t) + \ell(t)$, then $\forall i, 1 \leq i \leq D\ \sum_{t \in S(u)} R_i(t) \leq 1.0$ where

$$R_i(t) = \begin{cases} c(t) & \text{if } (p(t) + u - \sigma(t)) \bmod D = i - 1 \\ 0.0 & \text{otherwise} \end{cases} \tag{22}$$

The first constraint ensures that no task is scheduled before its release time. The second constraint strives to avoid retrieval contention. It guarantees that at each time interval u and for each cluster i, the aggregate bandwidth requirement of the tasks that employ cluster i and are in progress (i.e., have been initiated at or before u but have not committed yet), does not exceed the bandwidth of cluster i. The mod-function handles the round-robin utilization of clusters per task.

Augmented ARS

ARS$^+$ is identical to ARS except that there is a delay between the time that a task is released and the time that it should start. A sample application of *ARS*$^+$ could be a movie-on-demand application where the customers reserve movies in advance. For example, at 7:00 p.m. Alice reserves *GodFather* to be displayed at 8:00 p.m. Hence, assuming t be the task corresponding to Alice retrieving *GodFather,* $r(t) = 7{:}00$, but its start time is one hour later. Due to this extra knowledge, more flexible scheduling can be performed.

The quadruple notation of a task t_i for ARS is augmented as t_i:$\langle r(t_i), \S(t_i), \ell(t_i), c(t_i), p(t_i) \rangle$ for *ARS*$^+$. The lag parameter, $\S(t)$, determines the start time of the task. That is, the display of a task that is released at $r(t)$ should not start sooner than $r(t) + \S(t)$. Other than this distinction in the definition of a task between *ARS* and *ARS*$^+$, the definition of *ARS*$^+$ is identical to that of *ARS*. Note that the first constraint of the definition remains as $\forall t \in \mathcal{T}$, $\sigma(t) \geq r(t)$.

Composite Objects

We conceptualize a system that supports composite objects as consisting of three components: a collection of user interfaces, logical abstraction, and a storage manager (see Fig. 21). User interfaces play an important role in providing a friendly interface to (1) access existing data to author composite objects and (2) display objects. The logical abstraction tailors the user interface to the storage manager and is described further in the following paragraphs. The focus of this report is on the storage manager.

The logical abstraction is defined to separate the storage manager issues from the user interface. This has two major advantages: simplicity and portability. It results in simplicity because different (and maybe inconsistent) representations of composite objects dictated by the user interface have no impact on the algorithms at the storage manager level. An intermediate interpreter is responsible for translating the user's representations and commands to a uniform, consistent notation at the logical level. The storage manager becomes portable because it is independent of the user interface. Hence, if future interfaces start to use goggles and head-sets, the storage manager engine does not need to be modified.

At the logical level of abstraction, a composite object is represented as a (X, Y, j), indicating that the composite object consists of atomic objects X and Y. The parameter j is the lag parameter. It indicates that the display of object Y should start j time intervals after the display of X has started. For example, to designate a complex object where the display of X and Y must start at the same time, we will use the notation $(X, Y, 0)$. Likewise, the composite object specification $(X, Y, 2)$ indicates that the display of Y is initiated two intervals after the display of X has started. This definition of a composite object supports the 13 alternative temporal relationships described in (48). Figure 22 lists these temporal relationships and their representation using our notation of a composite object. The first two columns of Fig. 22 demonstrates the basic 7 relationships between atomic objects X and Y. The rest of the relationships are the inverse of these 7 (note that an equal relation has no inverse). Our proposed techniques support all temporal constructs because they solve for (1) arbitrary j values, (2) arbitrary sizes for both X and Y, and (3) arbitrary clusters to start the placement of X and Y.

Allen relations			*Composite object construct*
X before Y	*XXX YYY*	(Y, X, j)	size(X) < j
X equals Y	*XXX YYY*	(Y, X, j)	size(X) = size(Y) & j =0
X meets Y	*XXXYYY*	(Y, X, j)	j = size(X)
X overlaps Y	*XXX YYY*	(Y, X, j)	0 < j < size(X)
X during Y	*XXX YYYYYY*	(Y, X, j)	j > 0 & size(X) < size (Y) – j
X during Y	*XXX YYYYY*	(Y, X, j)	j = 0 & size(X) < size(Y)
X finishes Y	*XXX YYYYY*	(Y, X, j)	j = size(Y) – size(X) < size(Y)

Figure 22. Allen temporal relationships and their representation using our notation of a composite object.

Our notation extends naturally to the specification of composite objects that contain more than two atomic objects. A composite object containing n atomic objects can be characterized by $(n - 1)$ lag parameter, for example, $(X^1, . . X^n, j^2, . . j^n)$, where j^i denotes the lag parameter of object X^i with respect to the beginning of the display of object X^1.

To simplify the discussion, we assume integer values for the lag parameter (i.e., the temporal relationships are in the granularity of a time interval). For more accurate synchronization such as lip-synching between a spoken voice with the movement of the speaker's lips, real values of the lag parameter should be considered. This extension is straightforward. To illustrate, suppose time dependency between objects X and Y is defined such that the display of Y should start 2.5 seconds after the display of X starts. Assuming the duration of a time interval is one second, this time dependency at the task scheduling level can be mapped to $(X, Y, 2)$. Hence, the system can retrieve Y after 2 s but employ memory to postpone Ys display for 0.5 s.

Composite Retrieval Scheduling

The objectives of the Composite Retrieval Scheduling (CRS) problem are identical to those of ARS. The distinction is that with CRS, each user submit a request referencing a composite object. A composite object is a combination of two or more atomic objects with temporal relationships among them. The scheduler assigns a composite task to each request referencing a composite object. A composite task is a combination of atomic tasks. The time dependencies among the atomic tasks of a composite task is defined by the lag parameter $\S(t)$ of the atomic tasks. A sample application of CRS is the digital editing environment. An editor composes a composite task on demand, and the result should be displayed to the editor immediately. This is essential for the editor in order to evaluate her composition and possibly modify it immediately.

With CRS, each composite task consists of a number of atomic tasks. We use t to represent an atomic task and θ for a composite task. Similarly, $\mathscr{T}$ represents a set of atomic tasks while Θ is a set of composite tasks. A composite task, itself, is a set of atomic tasks; for example, $\theta = \{t_1, t_2, . . ., t_n\{$. Each atomic task has the same parameters as defined earlier, except for the release time $r(t)$. Instead, each atomic tasks has a lag time denoted by $\S(t)$. Without loss of generality, we assume for a composite task θ, $\S(t_1) \leq \S(t_2) \leq \cdots \leq \S(t_n)$. Subsequently, we denote the first atomic task in the set θ as $car(\theta)$; that is, $car(\theta) = t_1$. Lag time of a task determines the start time of the task with respect to $\S(car(\theta))$. Trivially, $\S(car(\theta)) = 0$. Briefly, $\S(t)$ determines the temporal relationships among the atomic tasks of a composite task. Each composite task, on the other hand, has only a release time $r(\theta)$ which is the time that a request for the corresponding composite object is submitted.

Definition 2: An atomic task (of a composite task) t is *schedulable* at u if t can be started at u and completes at $u + \ell(t) - 1$ without resulting in retrieval contention as defined in Def. 1.

Definition 3: A composite task $\theta = \{t_1, t_2, . . ., t_n\}$ is said to be schedulable at u if $\forall t \in \theta$; t is schedulable at $u + \S(t)$.

Definition 4: The problem of CRS is to find a schedule σ (where σ:$\Theta \rightarrow N$) for a set Θ, such that (1) it minimizes the finishing time (52) w, where w is the least time at which all tasks of Θ have been completed, and (2) satisfies the following constraints

- $\forall \theta \in \Theta$; $\sigma(\theta) \geq r(\theta)$.
- θ be schedulable at $\sigma(\theta)$. (see Def. 3).

Resolving Internal Contention for CRS (RIC)

The CRS problem is involved with scheduling multiple composite tasks. RIC, however, focuses on the scheduling of a single composite task. The problem is that even scheduling a single composite task might not be possible due to retrieval contention among its constituting atomic tasks. RIC is very much like the clairvoyant *ARS* problem. The distinction is that a task can start sooner than its release time, employing upsliding. RIC can also be considered as a similar problem to ASR^+ where all the tasks have an identical release time but different start time. However, with ARS^+ in the worst case, a task (who cannot slide upward) can be postponed, while postponing a task with RIC will violate the defined temporal relationships. Furthermore, ARS^+ and RIC have different objectives. Due to the above reasons, we study RIC as a separate scheduling problem.

A composite object may have internal contention; that is, atomic tasks that constitute a composite task may compete with one another for the available cluster bandwidth. Hence, it is possible that due to such internal contention, a composite task is not schedulable even if there are no other active requests. In other words, it is not possible to start all the atomic tasks of θ at their start time.

Definition 5: Internal contention: Consider a composite task $\theta = \{t_1, t_2, . . ., t_n\{$, and $\forall u \geq 0$; let $S(u) \subseteq \theta$ be the set of atomic tasks which $\S(t) \leq u < \S(t) + \ell(t)$. The composite task

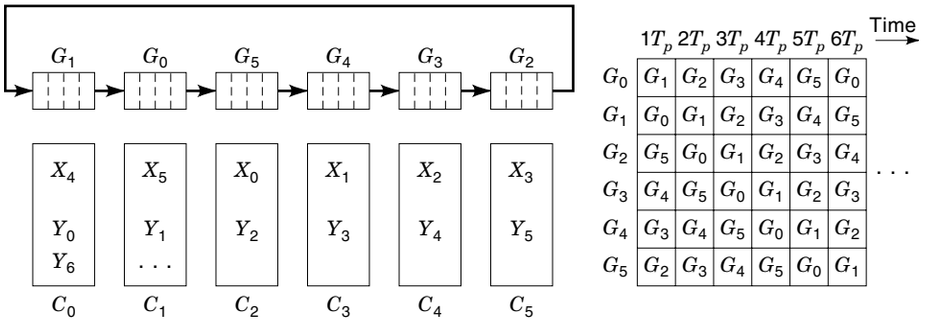

Figure 23. Rotating groups.

θ has internal contention if $\exists u$, i $1 \le i \le D$ such that $\sum_{t \in S(u)} R_i(t) > 1.0$ where

$$R_i(t) = \begin{cases} c(t) & \text{if } (p(t) + u - \S(t)) \bmod D = i - 1 \\ 0.0 & \text{otherwise} \end{cases} \quad (23)$$

The above definition intuitively means that /$\exists u$ such that θ be schedulable at u (see Def. 3). This problem is particular to composite objects because there is a dependency among start times of atomic tasks, and yet these atomic tasks can conflict with each other.

Definition 6: Resolving the internal contention for a composite task θ is to modify the start time of its consisting atomic tasks, such that Def. 5 does not hold true for θ.

Such a modification requires use of memory buffers. Ideally, we should minimize the amount of required buffer.

Definition 7: The problem of RIC is to resolve the internal contention for a composite task θ (as defined in Def. 6) while minimizing the amount of required memory.

OPTIMIZATION TECHNIQUES

In this section, we discuss some techniques to improve the utilization of a continuous media server. First, two techniques to reduce startup latency are explained. Next, three methods to improve the system throughput are described. Finally, we focus on retrieval optimization techniques for those applications where a request references multiple CM objects (i.e., composite objects are described earlier). We describe a taxonomy of optimization techniques which is applicable in certain applications with flexible presentation requirements.

Minimizing Startup Latency

Considering the hybrid striping approach described earlier, each request should wait until a time slot corresponding to the cluster containing the first block of its referenced object becomes available. This is true even when the system is not 100% utilized. To illustrate, conceptualize a set of slots supported by a cluster in a time period as a group. Each group has a unique identifier. To support a continuous display in a multi-cluster system, a request maps onto one group, and the individual groups visit the clusters in a round-robin manner (Fig. 23). If group G_5 accesses cluster C_2 during a time period, G_5 would access C_3 during the next time period. During a given time period, the requests occupying the slots of a group retrieve blocks that reside in the cluster that is being visited by that group.

Therefore, if there are C clusters (or groups) in the system, and each cluster (or group) can support $\mathcal{M}$ simultaneous displays, then the maximum throughput of the system is $m = \mathcal{M} \times C$ simultaneous displays. The maximum startup latency is $T_p \times C$ because (1) groups are rotating (i.e., playing musical chairs) with the C clusters using each for a T_p interval of time, and (2) at most, $C - 1$ failures might occur before a request can be activated (when the number of active displays is fewer than $\mathcal{M} \times C$). Thus, both the system throughput and the maximum startup latency scale linearly. Note that system parameters such as blocks size, time period, throughput, etc. for a cluster can be computed using the equations provided earlier, depending on the selected display technique. These display techniques are local optimizations that are orthogonal to the optimization techniques proposed by this section.

Even though the work load of a display is distributed across the clusters with a round-robin assignment of blocks, a group might experience a higher work load as compared to other groups. For example, in Fig. 24, if the system services a new request for object X using group G_4, then all servers in G_4 become busy, while several other groups have two idle servers. This imbalance might result in a higher startup latency for future requests. For example, if another request for Z arrives, then it would incur a two time period startup latency because it must be assigned to G_5 because G_4 is already full. This section describes request migration and replication (53) as two alternative techniques to minimize startup latency. These two techniques are or-

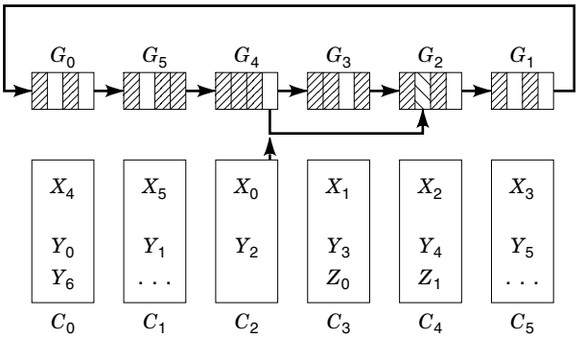

Figure 24. Load balancing.

thogonal to one another, enabling a system to employ both at the same time.

Request Migration. By migrating one or more requests from a group with zero idle slots to a group with many idle slots, the system can minimize the possible latency incurred by a future request. For example, in Fig. 24, if the system migrates a request for X from G_4 to G_2, then a request for Z is guaranteed to incur a maximum latency of one time period. Migrating a request from one group to another increases the memory requirements of a display because the retrieval of data falls ahead of its display. Migrating a request from G_4 to G_2 increases the memory requirement of this display by three buffers. This is because when a request migrates from G_4 to G_2 (see Fig. 24), G_4 reads X_0 and sends it to the display. During the same time period, G_3 reads X_1 into a buffer (say, B_0), and G_2 reads X_2 into a buffer (B_1). During the next time period, G_2 reads X_3 into a buffer (B_2), and X_1 is displayed from memory buffer B_0. (G_2 reads X_3 because the groups move one cluster to the right at the end of each time period to read the next block of active displays occupying its servers.) During the next time period, G_2 reads X_4 into a memory buffer (B_3), while X_2 is displayed from memory buffer B_1. This round-robin retrieval of data from clusters by G_2 continues until all blocks of X have been retrieved and displayed.

With this technique, if the distance from the original group to the destination group is B, then the system requires $B + 1$ buffers. However, because a request can migrate back to its original group once a request in the original group terminates and relinquishes its slot (i.e., a time slot becomes idle), the increase in total memory requirement could be reduced and become negligible.

When $k \leq C \cdot (\mathcal{N} - 1)$ (with the probability of $\sum_{k=0}^{C\cdot(\mathcal{N}-1)} p(k)$), request migration can be applied due to the availability of idle slots. This means that $Prob\{\text{a group is full}\} = 0$. Hence, $p_f(0, k) = 1$. If $k > C \cdot (\mathcal{N} - 1)$ (with the probability of $\sum_{k=C\cdot(\mathcal{N}-1)+1}^{m-1} p(k)$), no request migration can be applied because (1) no idle slot is available in some groups, and (2) the load is already evenly distributed. Hence, the probability of failures is:

$$p_f(i, k') = \frac{\binom{C-i}{k'-i} - \binom{C-(i+1)}{k'-(i+1)}}{\binom{C}{k'}} \tag{24}$$

where $k' = k - C \cdot (\mathcal{N} - 1)$. The expected latency with request migration is

$$\begin{aligned} E[L] = &\sum_{k=0}^{C\cdot(\mathcal{N}-1)} p(k) \cdot 0.5 \cdot T_p \\ &+ \sum_{k=c\cdot(\mathcal{N}-1)+1}^{m-1} p(k) \cdot p_f(0, k') \cdot 0.5 \cdot T_p \\ &+ \sum_{k=C\cdot(\mathcal{N}-1)+1}^{m-1} \sum_{i=1}^{k'} p(k) \cdot p_f(i, k') \cdot i \cdot T_p \end{aligned} \tag{25}$$

Object Replication. To reduce the startup latency of the system, one may replicate objects. We term the original copy of an object X as its primary copy. All other copies of X are termed its secondary copies. The system may construct r secondary copies for object X. Each of its copies is denoted as $R_{X,i}$, where $1 \leq i \leq r$. The number of instances of X is the number of copies of X, $r + 1$ (r secondary plus one primary). Assuming two instances of an object, by starting the assignment of $R_{X,1}$ with a cluster different than the one containing the first block of its primary copy (X), the maximum startup latency incurred by a display referencing X can be reduced by one half. This also reduces the expected startup latency. The assignment of the first block of each copy of X should be separated by a fixed number of clusters in order to maximize the benefits of replication. Assuming that the primary copy of X is assigned starting with an arbitrary clusters (say C_i contains X_0), the assignment of secondary copies of X is as follows. The assignment of the first block of copy $R_{X,j}$ should start with cluster $(C_i + jC/r + 1) \bmod C$. For example, if there are two secondary copies of object Y ($R_{Y,1}$, $R_{Y,2}$) assume its primary copy is assigned starting with cluster C_0. $R_{Y,1}$ is assigned starting with cluster C_2, while $R_{Y,2}$ is assigned starting with cluster C_4.

With two instances of an object, the expected startup latency for a request referencing this object can be computed as follows. To find an available server, the system simultaneously checks two groups corresponding to the two different clusters that contain the first blocks of these two instances. A failure happens only if both groups are full, reducing the number of failures for a request. The maximum number of failures before a success is reduced to $\lfloor k/2 \cdot \mathcal{N} \rfloor$ due to two simultaneous searching of groups in parallel. Therefore, the probability of i failures in a system with each object having two instances is identical to that of a system consisting of $C/2$ clusters with $2\mathcal{N}$ servers per cluster. A request would experience a lower number of failures with more instances of objects. For an arbitrary number of instances (say j) for an object in the system, the probability of a request referencing this object to observe i failures is

$$p_{f_j}(i, k) = \frac{\binom{m-j\cdot i\cdot \mathcal{N}}{k-j\cdot i\cdot \mathcal{N}} - \binom{m-j\cdot (i+1)\cdot \mathcal{N}}{k-j\cdot (i+1)\cdot \mathcal{N}}}{\binom{m}{k}} \tag{26}$$

where $0 \leq i \leq \lfloor k/j \cdot \mathcal{N} \rfloor$. Hence, the expected startup latency is

$$\begin{aligned} E[L] = &\sum_{k=0}^{m-1} p(k) \cdot p_{f_j}(0, k) \cdot 0.5 \cdot T_p \\ &+ \sum_{k=0}^{m-1} \sum_{i=1}^{\lfloor \frac{k}{j\cdot \mathcal{N}} \rfloor} p(k) \cdot p_{f_j}(i, k) \cdot i \cdot T_p \end{aligned} \tag{27}$$

Object replication increases the storage requirement of an application. One important observation in real applications is that objects may have different access frequencies. For example, in a Video-On-Demand system, more than half of the active requests might reference only a handful of recently released movies. Selective replication for frequently referenced (i.e., hot) objects could significantly reduce the latency without a dramatic increase in storage space requirement of an

application. The optimal number of secondary copies per object is based on its access frequency and the available storage capacity. The formal statement of the problem is as follows. Assuming n objects in the system, let S be the total amount of disk space for these objects and their replicas. Let R_j be the optimal number of instances for object j, S_j to denote the size of object j, and F_j to represent the access frequency (%) of object j. The problem is to determine R_j for each object j ($1 \leq j \leq n$) while satisfying $\sum_j^n = R_j \cdot S_j \leq S$.

There exist several algorithms to solve this problem (54). A simple one known as the Hamilton method computes the number of instances per object j based on its frequency (see (53). It rounds the remainder of the quota ($Q_j - \lfloor Q_j \rfloor$) to compute R_j. However, this method suffers from two paradoxes, namely, the Alabama and Population paradoxes (54). Generally speaking, with these paradoxes, the Hamilton method may reduce the value of R_j when either S or F_j increases in value. The divisor methods provide a solution free of these paradoxes. For further details and proofs of this method, see (15). Using a divisor method named Webster ($d(R_j) = R_j + 0.5$), we classify objects based on their instances. Therefore, objects in a class have the same instances. The expected startup latency in this system with n objects is

$$E[L] = \sum_{i=1}^{n} F_i \cdot E[L_{R_i}] \tag{28}$$

where $E[L_{R_i}]$ is the expected startup latency for object having R_i instances (computed using Eq. 27).

Maximizing Throughput

A trivial concept for increasing the throughput of a continuous media server is to support multiple displays (or users) by utilizing a single disk stream. This can be achieved when many requests reference an identical CM object. The problem is, however, when these requests arrive in different time instances. In this section, we explain three approaches to hide the time differences among multiple requests referencing a single object:

1. Batching of requests (55–58): In this method, requests are delayed until they can be merged with other requests for the same video. These merged streams then form one physical stream from the disk and consume only one set of buffers. Only on the network will the streams split at some point for delivery to the individual display stations.
2. Buffer sharing (59–64): The idea here is that if one stream for a video lags, another stream for the same video by only a short time interval; then, the system could retain the portion of the video between the two in buffers. The lagging stream would read from the buffers and not have to read from disk.
3. Adaptive piggy-backing (65): In this approach, streams for the same video are adjusted to go slower or faster by a few percent, such that it is imperceptible to the viewer, and the streams eventually merge and form one physical stream from the disks.

Batching and adaptive piggy-backing are orthogonal to buffer sharing.

Optimization Techniques for Applications with Flexible Presentation Requirements

In many multimedia applications, the result of a query is a set of CM objects that should be retrieved from a CM server and displayed to the user. This set of CM objects has to be presented to the user as a coherent presentation. Multimedia applications can be classified as either having Restricted Presentation Requirements (RPR) or Flexible Presentation Requirements (FPR). RPR applications require that the display of the objects conform to a very strict requirement set, such as temporal relationships, selection criteria, and display quality. Digital editing is an example of RPR (see CRS and RIC). In FPR applications, such as digital libraries, music-juke-boxes, and news-on-demand applications, users can tolerate some temporal, selection, and display quality variations. These flexibilities stem from the nature of multimedia data and user-queries.

RPR applications impose very strict display requirements. This is due to the type of queries imposed by users in such applications. It is usually the case that the user can specify what objects he/she is interested in and how to display these objects in concert. Multimedia systems have to guarantee that the CM server can retrieve all the objects in the set and can satisfy the precise time dependencies, as specified by the user. There has been a number of studies on scheduling continuous media retrievals for RPR applications, see (51,66,67,68). In (51,66,68) the time dependencies are guaranteed by using memory buffers, while in (67), they are guaranteed by using the in-advance knowledge at the time of data placement.

FPR applications provide some flexibilities in the presentation of the continuous media objects. It is usually the case that the user does not know exactly what he/she is looking for and is only interested in displaying the objects with some criteria (e.g., show me today's news). In general, almost all applications using a multimedia DBMS fall into this category. In this case, depending on the user query, user profile, and session profile, there are a number of flexibilities that can be exploited for retrieval optimization. We have identified the following flexibilities in (69):

- Delay flexibility which species the amount of delay the user/application can tolerate between the display of different continuous media clips (i.e., relaxed *meet, after,* and *before* relationship (48). In some applications, such delays are even desirable in order for the user (i.e., human perception) to distinguish between two objects).
- Selection flexibility which refers to whether the objects selected for display are a fixed set (e.g., two objects selected for display) or they are a suggestion set (e.g., display two objects out of four candidate objects.) This flexibility is identified; however, we do not use it in our formal definitions. It is part of our future research.
- Ordering flexibility which refers to the display order of the objects (i.e., to what degree that display order of the objects is important to the user).
- Presentation flexibility which refers to the degree of flexibility in the presentation length and presentation startup latency.

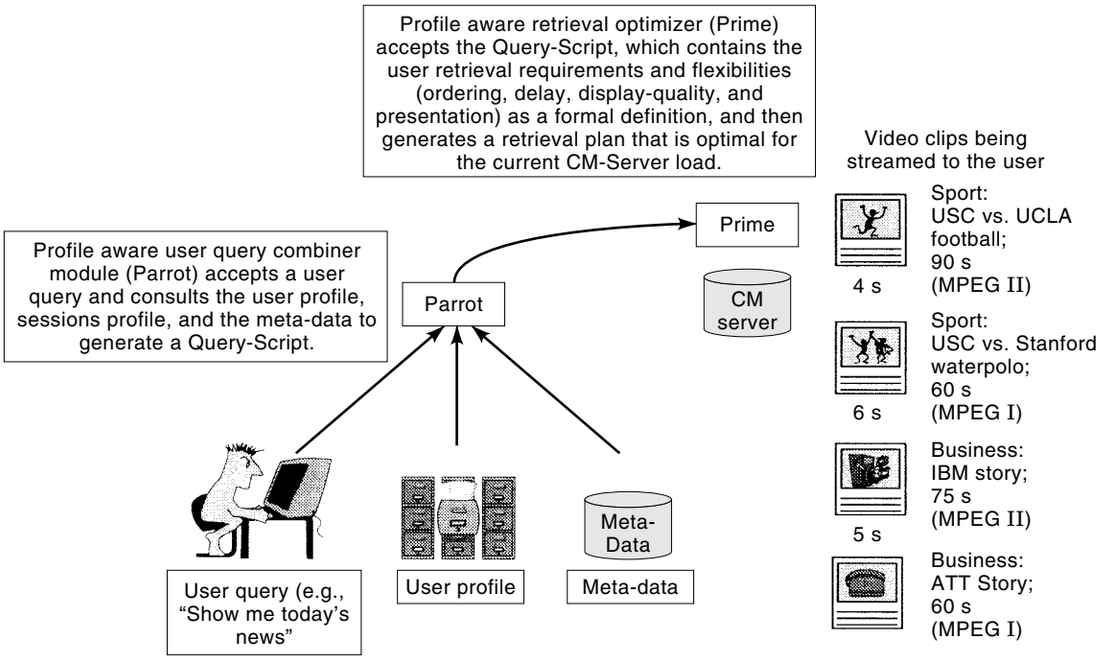

Figure 25. System architecture.

- Display-quality flexibility which specifies the display qualities acceptable by the user/application, when data is available in multiple formats (e.g., MPEG I, MPEG II, etc.) and/or in hierarchical or layered formats (based on layered compression algorithms) (70, 71).

With FPR applications, the flexibilities allow for the construction of multiple retrieval plans per presentation. Subsequently, the best plan is identified as the one which results in minimum contention at the CM server. To achieve this, three steps should be taken:

- Step 1: gathering flexibilities
- Step 2: capturing the flexibilities in a formal format and
- Step 3: using the flexibilities for optimization

In our system architecture (69), Fig. 25, the first two steps are carried out by the Profile Aware User Query Combiner (Parrot). It takes an input the user query, user profile, and session profile (e.g., type of monitor) to generate a query script (as output). We assume that there exist intelligent agents that would build user profiles either explicitly (i.e., by user interaction) and/or implicitly (i.e., by clandestine monitoring of the user actions, as in (72). This query script would capture all the flexibilities and requirements in a formal manner. The query script is then submitted to the Profile Aware Retrieval Optimizer (Prime) which, in turn, would use it to generate the best retrieval plan for the CM server.

Using the query script, Prime defines a search space that consists of all the correct retrieval plans. A retrieval plan is correct if and only if it is consistent with the defined flexibilities and requirements. Prime also defines a cost model to evaluate the different retrieval plans. The retrieval plans are then searched (either exhaustively or by employing heuristics) to find the best plan depending on the metrics defined by the application. In (69), we also describe a memory buffering mechanism that alleviates retrieval problems when the system bandwidth becomes fermented, namely the Simple Memory Buffering (SimB) mechanism. Our simulation studies show significant improvement when we compare the system performance for the best retrieval plan with that of the worst, or even average, plan of all the correct plans. For example, if latency time (i.e., time elapsed from when the retrieval plan is submitted until the onset of the display of its first object) is considered as a metric, the best plan found by Prime observes 41% to 92% improvement as compared with the worst plan, and 26% to 89% improvement as compared with the average plans when SimB is not applied (see (69)).

CASE STUDY

The design and implementation of many CM servers have been reported in the research literature (e.g., (30, 40, 73, 74, 75). Commercial implementations of CM servers are also in progress (e.g., Sun's MediaCenter Servers (76), Starlight Networks' StarWorks (77), and Storage Concepts' VIDEOPLEX (78), see Table 2). Many of the design issues that we discussed in this paper have been practiced in most of the above prototypes. In this section, we focus on the implementation of Mitra (Mitra is the name of a Persian/Indian god with thousands of eyes and ears) (40) developed at the USC database laboratory.

Mitra: A Scalable CM Server

Mitra employs GSS with $g = \mathcal{N}$, coarse-grain memory sharing, hybrid striping, a three level storage hierarchy with SDF data flow, no pipelining, and an Everest replacement policy. Multi-zone disk drive optimization (11) as well as replication

Table 2. A Selection of Commercially Available Continuous-Media Servers

Vendor	Product	Max. No. of Users	Max. Streaming Capacity
Starlight	StarWorks-200M	133 @ 1.5 Mb/s	200 Mb/s
Sun	MediaCenter 1000E	270 @ 1.5 Mb/s	400 Mb/s
Storage Concepts	VIDEOPLEX	320 @ 1.5 Mb/s	480 Mb/s[a]

[a] The VIDEOPLEX system does not transmit digital data over a network but uses analog VHS signals instead.

and migration optimizations have also been incorporated in Mitra.

Mitra employs a hierarchical organization of storage devices to minimize the cost of providing on-line access to a large volume of data. It is currently operational on a cluster of HP 9000/735 workstations. It employs a HP Magneto Optical Juke-box as its tertiary storage device. Each workstation consists of a 125 MHz PA-RISC CPU, 80 MByte of memory, and four Seagate ST31200W magnetic disks. Mitra employs the HP-UX operating system (version 9.07) and is portable to other hardware platforms. While 15 disks can be attached to the fast and wide SCSI-2 bus of each workstation, we attached four disks to this chain because additional disks would exhaust the bandwidth of this bus. It is undesirable to exhaust the bandwidth of the SCSI-2 bus for several reasons. First, it would cause the underlying hardware platform to not scale as a function of additional disks. Mitra is a software system, and if its underlying hardware platform does not scale, then the entire system would not scale. Second, it renders the service time of each disk unpredictable, resulting in hiccups.

Mitra consists of three software components:

1. Scheduler: This component schedules the retrieval of the blocks of a referenced object in support of a hiccup-free display at a PM. In addition, it manages the disk bandwidth and performs admission control. Currently, the scheduler includes an implementation of EVEREST, staggered striping, and techniques to manage the tertiary storage device. It also has a simple relational storage manager to insert and retrieve information from a catalog. For each media type, the catalog contains the bandwidth requirement of that media type and its block size. For each presentation, the catalog contains its name, whether it is disk resident (if so, the name of EVEREST files that represent this clip), the cluster and zone that contains its first block, and its media type.
2. Mass storage Device Manager (DM): Performs either disk or tertiary read/write operations.
3. Presentation Manager (PM): Displays either a video or an audio clip. It might interface with hardware components to minimize the CPU requirement of a display. For example, to display an MPEG-2 clip, the PM might employ either a program or a hardware-card to decode and display the clip. The PM implements the PM-driven scheduling policy (40) to control the flow of data from the scheduler.

Mitra uses UDP for communication between the process instantiation of these components. UDP is an unreliable trans-

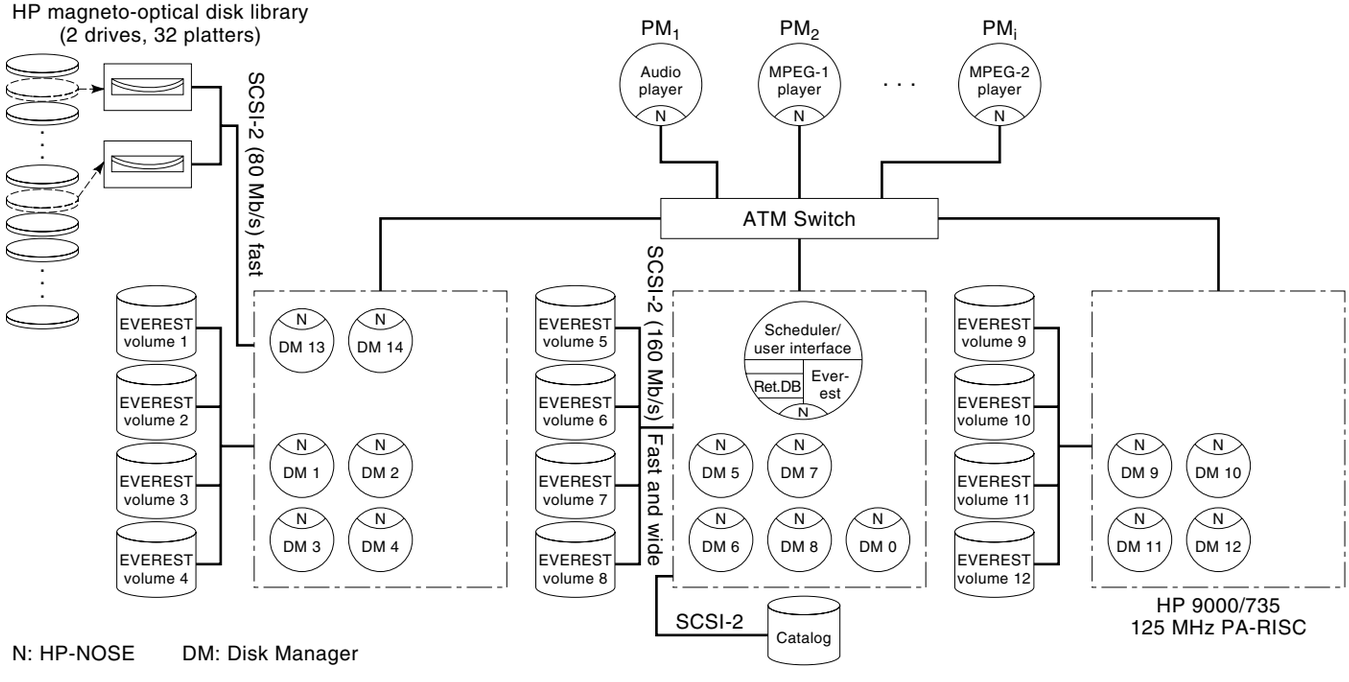

Figure 26. Hardware and software organization of Mitra.

mission protocol. Mitra implements a light-weight kernel named HP-NOSE. HP-NOSE supports a window-based protocol to facilitate reliable transmission of messages among processes. In addition, it implements the threads with shared memory, ports that multiplex messages using a single HP-UX socket, and semaphores for synchronizing multiple threads that share memory. An instantiation of this kernel is active per Mitra process.

For a given configuration, the following processes are active: one scheduler process, a DM process per mass storage read/write device, and one PM process per active client. For example, in our twelve disk configuration with a magneto optical juke box, there are sixteen active processes: fifteen DM processes and one Scheduler process (see Fig. 26). There are two active DM processes for the magneto juke-box because it consists of two read/write devices (and 32 optical platters that might be swapped in and out of these two devices).

The combination of the scheduler with DM processes implements asynchronous read/write operations on a mass storage device (which is otherwise unavailable with HP-UX 9.07). This is achieved as follows. When the scheduler intends to read a block from a device (say a disk), it sends a message to the DM that manages this disk to read the block. Moreover, it requests the DM to transmit its block to a destination port address (e.g., the destination might correspond to the PM process that displays this block) and issue a done message to the scheduler. There are several reasons for not routing data blocks to active PMs using the scheduler. First, it would waste the network bandwidth with multiple transmissions of a block. Second, it would cause the CPU of the workstation that supports the scheduler process to become a bottleneck with a large number of disks. This is because a transmitted data block would be copied many times by different layers of software that implement the scheduler process: HP-UX, HP-NOSE, and the scheduler.

ACKNOWLEDGMENTS

We would like to thank Ali Dashti, Doug Ierardi, Seon Ho Kim, Weifeng Shi, and Roger Zimmermann for contributing to the presented material.

BIBLIOGRAPHY

1. S. Ghandeharizadeh and L. Ramos, Continuous retrieval of multimedia data using parallelism, *IEEE Trans. Knowl. Data Eng.,* **1**: 658–669, 1993.
2. D. J. Gemmell et al., Multimedia storage servers: A tutorial, *IEEE Comput.,* **28** (5): 40–49, 1995.
3. D. Le Gall, MPEG: a video compression standard for multimedia applications, *Commun. ACM,* **34** (4): 46–58, 1991.
4. J. Dozier, Access to data in NASA's Earth observing system (Keynote Address), *Proc. ACM SIGMOD Int. Con. Manage. Data,* June 1992.
5. T. D. C. Little and D. Venkatesh, Prospects for interactive video-on-demand, *IEEE Multimedia,* **1** (3): 14–24, 1994.
6. D. P. Anderson, Metascheduling for continuous media, *ACM Trans. Comput. Syst.,* **11** (3): 226–252, 1993.
7. C. Ruemmler and J. Wilkes, An introduction to disk drive modeling, *IEEE Computer,* **27** (3): 1994.
8. D. Bitton and J. Gray, Disk shadowing, *Proc. Int. Conf. Very Large Databases,* September 1988.
9. J. Gray, B. Host, and M. Walker, Parity striping of disc arrays: Low-cost reliable storage with acceptable throughput, *Proc. Int. Conf. Very Large Databases,* August 1990.
10. S. Ghandeharizadeh, J. Stone, and R. Zimmermann, Techniques to quantify SCSI-2 disk subsystem specifications for multimedia, Technical Report USC-CS-TR95-610, Univ. Southern California, 1995.
11. S. Ghandeharizadeh et al., Placement of continuous media in multi-zone disks. In Soon M. Chung (ed.) *Multimedia Information Storage and Management,* chapter 2, Norwell, MA: Kluwer Academic, August 1996.
12. S. Ghandeharizadeh and C. Shahabi, On multimedia repositories, personal computers, and hierarchical storage systems. *Proc. ACM Multimedia,* 1994.
13. P. S. Yu, M. S. Chen, and D. D. Kandlur, Design and analysis of a grouped sweeping scheme for multimedia storage management. *Proc. Int. Workshop Network Oper. Sys. Support Digital Audio Video,* November 1992.
14. D. J. Gemmell and S. Christodoulakis, Principles of delay sensitive multimedia data storage and retrieval, *ACM Trans. Inf. Sys.,* **10**: 51–90, Jan. 1992.
15. D. J. Gemmell et al., Delay-sensitive multimedia on disks, *IEEE Multimedia,* **1** (3): 56–67, Fall 1994.
16. H. J. Chen and T. Little, Physical storage organizations for time-dependent multimedia data, *Proc. Foundations Data Organ. Algorithms FODO Conf.,* October 1993.
17. A. L. N. Reddy and J. C. Wyllie, I/O Issues in a Multimedia System, *IEEE Comput. Mag.,* **27** (3): March 1994.
18. A. Cohen, W. Burkhard, and P. V. Rangan, Pipelined disk arrays for digital movie retrieval. *Proceedings ICMCS '95,* 1995.
19. E. Chang and H. Garcia-Molina, Reducing initial latency in a multimedia storage system, *Proc. IEEE Int. Workshop Multimedia Database Manage. Syst.,* 1996.
20. B. Ozden, R. Rastogi, and A. Silberschatz, On the design of a low-cost video-on-demand storage system, *ACM Multimedia Syst.,* **4** (1): 40–54, February 1996.
21. P. Bocheck, H. Meadows, and S. Chang, Disk partitioning technique for reducing multimedia access delay, in *Proc. IASTED / ISMM Int. Conf. Distributed Multimedia Systems and Applications,* August 1994, pp. 27–30.
22. S. Ghandeharizadeh, S. H. Kim, and C. Shahabi, On configuring a single disk continuous media server, *Proc. 1995 ACM SIGMETRICS / PERFORMANCE,* May 1995.
23. S. Ghandeharizadeh, S. H. Kim, and C. Shahabi, *On disk scheduling and data placement for video servers,* USC Technical Report, Univ. Southern California, 1996.
24. T. J. Teory, Properties of disk scheduling policies in multiprogrammed computer systems. *Proc. AFIPS Fall Joint Comput. Conf.,* 1972, pp. 1–11.
25. Y. Birk, Track-pairing: A novel data layout for VOD servers with multi-zone-recording disks, *Proc. IEEE Int. Conf. Multimedia Comput. Syst.,* May 1995, pp. 248–255.
26. S. R. Heltzer, J. M. Menon, and M. F. Mitoma, *Logical data tracks extending among a plurality of zones of physical tracks of one or more disk devices,* U.S. Patent No. 5,202,799, April 1993.
27. R. Zimmermann and S. Ghandeharizadeh, Continuous display using heterogeneous disk-subsystems, *Proc. ACM Multimedia 97,* New York: ACM, 1997.
28. S. Ghandeharizadeh and C. Shahabi, Management of physical replicas in parallel multimedia information systems, *Proc. Foundations Data Organ. Algorithms FODO Conf.,* October 1993.

29. D. Patterson, G. Gibson, and R. Katz, A case for redundant arrays of inexpensive disks RAID, *Proc. ACM SIGMOD Int. Conf. Manage. Data,* May 1988.
30. F. A. Tobagi et al., Streaming RAID-A disk array management system for video files, *1st ACM Conf. Multimedia,* August 1993.
31. S. Ghandeharizadeh and S. H. Kim, Striping in multi-disk video servers, *High-Density Data Recording and Retrieval Technologies, Proc. SPIE,* **2604**, 1996, pp. 88–102.
32. S. Ghandeharizadeh, A. Dashti, and C. Shahabi, A pipelining mechanism to minimize the latency time in hierarchical multimedia storage managers, *Comput. Commun.,* **18** (3): 170–184, March 1995.
33. S. Berson et al., Staggered striping in multimedia information systems, *Proc. ACM SIGMOD Int. Conf. Manage. Data,* 1994.
34. S. Ghandeharizadeh et al., Object placement in parallel hypermedia systems, *Proc. Int. Conf. Very Large Databases,* 1991.
35. M. Carey, L. Haas, and M. Livny, Tapes hold data, too: Challenges of tuples on tertiary storage, *Proc. ACM SIGMOD Int. Conf. Manage. Data,* 1993, pp. 414–417.
36. P. J. Denning, The working set model for program behavior. *Commun. ACM,* **11** (5): 323–333, 1968.
37. M. M. Astrahan et al., System R: Relational approach to database management, *ACM Trans. Database Syst.,* **1** (2): 97–137, 1976.
38. H. T. Chou et al., Design and implementation of the Wisconsin Storage System, *Softw. Practice Experience,* **15** (10): 943–962, 1985.
39. J. Gray and A. Reuter, Chapter 13, *Transaction Processing: Concepts and Techniques.* San Mateo, CA: Morgan Kaufmann, 1993.
40. S. Ghandeharizadeh et al., A scalable continuous media server, *Kluwer Multimedia Tools and Appl.,* **5** (1): 79–108, July 1997.
41. S. Ghandeharizadeh and D. Ierardi, Management of disk space with REBATE, *Proc. 3rd Int. Conf. Inf. Knowl. Manage. CIKM,* November 1994.
42. P. J. Denning, Working sets past and present, *IEEE Trans. Softw. Engi.,* **SE-6**: 64–84, 1980.
43. S. Ghandeharizadeh et al., *Placement of data in multi-zone disk drives,* Technical Report USC-CS-TR96-625, Univ. Southern California, 1996.
44. K. C. Knowlton, A fast storage allocator, *Commun. ACM,* **8** (10): 623–625, 1965.
45. H. R. Lewis and L. Denenberg, Chapter 10, *Data Structures & Their Algorithms,* 367–372, New York: Harper Collins, 1991.
46. G. Copeland et al., Data placement in bubba, *Proc. ACM SIGMOD Int. Conf. Manage. Data,* 1988, pp. 100–110.
47. T. H. Cormen, C. E. Leiserson, and R. L. Rivest (eds.), *Introduction to Algorithms.* Cambridge, MA: MIT Press, and New York: McGraw-Hill, 1990.
48. J. F. Allen, Maintaining knowledge about temporal intervals, *Commun. ACM,* **26** (11): 832–843, 1983.
49. R. Reddy, Some research problems in very large multimedia databases, *Proc. IEEE 12th Int. Conf. Data Eng.,* 1996.
50. G. Ozsoyoglu, V. Hakkoymaz, and J. Kraft, Automating the assembly of presentations from multimedia databases, *Proc. IEEE 12th Int. Conf. Data Eng.,* 1996.
51. C. Shahabi, *Scheduling the Retrievals of Continuous Media Objects,* PhD thesis, Univ. Southern California, 1996.
52. M. Garey and R. Graham, Bounds for multiprocessor scheduling with resource constraints, *SIAM J. Comput.,* **4** (2): 187–200, 1975.
53. S. Ghandeharizadeh et al., On minimizing startup latency in scalable continuous media servers, *Proc. Multimedia Comput. Networking, Proc. SPIE* **3020**, Feb. 1997, pp. 144–155.
54. T. Ibaraki and N. Katoh, *Resource Allocation Problems—Algorithmic Approaches,* Cambridge, MA: The MIT Press, 1988.
55. A. Dan et al., *Channel Allocation under Batching and VCR Control in Movie-On-Demand Servers,* Technical Report RC19588, Yorktown Heights, NY: IBM Research Report, 1994.
56. A. Dan, D. Sitaram, and P. Shahabuddin, Scheduling policies for an on-demand video server with batching, *Proc. ACM Multimedia,* 1994, pp. 15–23.
57. B. Ozden et al., A low-cost storage server for movie on demand databases, *Proc. 20th Int. Conf. Very Large Data Bases,* Sept. 1994.
58. J. L. Wolf, P. S. Yu, and H. Shachnai, DASD dancing: A disk load balancing optimization scheme for video-on-demand computer systems, *Proc. 1995 ACM SIGMETRICS / PERFORMANCE,* May 1995, pp. 157–166.
59. M. Kamath, K. Ramamritham, and D. Towsley, Continuous media sharing in multimedia database systems, *Proc. 4th Int. Conf. Database Syst. Advanced Appl.,* 1995, pp. 79–86.
60. B. Özden, R. Rastogi, and A. Silberschatz, Buffer replacement algorithms for multimedia databases, *IEEE Int. Conf. Multimedia Comput. Syst.,* June 1996.
61. D. Rotem and J. L. Zhao, Buffer management for video database systems, *Proc. Int. Conf. Database Eng.,* March 1995, pp. 439–448.
62. A. Dan and D. Sitaram, *Buffer management policy for an on-demand video server,* U.S. Patent No. 5572645, November 1996.
63. A. Dan et al., Buffering and caching in large-scale video servers. *Proc. COMPCON,* 1995.
64. W. Shi and S. Ghandharizadeh, Data sharing in continuous media servers, submitted to VLDB '97, Athens, Greece, August 1997.
65. L. Golubchik, J. Lui, and R. Muntz, Reducing I/O demand in video-on-demand storage servers, *Proc. ACM SIGMRTRICS,* 1995, pp. 25–36.
66. C. Shahabi, S. Ghandeharizadeh, and S. Chaudhuri, *On scheduling atomic and composite multimedia objects,* USC Technical Report USC-CS-95-622, Univ. Southern California, 1995.
67. C. Shahabi and S. Ghandeharizadeh, Continuous display of presentations sharing clips, *ACM Multimedia Systems,* **3** (2): 76–90, 1995.
68. S. T. Cambell and S. M. Chung, Delivery scheduling of multimedia streams using query scripts. In S. M. Chung (ed.), *Multimedia Information Storage and Management,* Norwell, MA: Kluwer, August 1996, Chapter 5.
69. C. Shahabi, A. Dashti, and S. Ghandeharizadeh, Profile aware retrieval optimizer for continuous media, submitted to VLDB '97, Athens, Greece, August 1997.
70. K. Keeton and R. H. Katz, Evaluating video layout strategies for a high-performance storage server, *ACM Multimedia Syst.,* **3** (2): May 1995.
71. S. McCanne, *Scalable Compression and Transmission of Internet Multicast Video,* PhD thesis, Berkeley: University of California, 1996.
72. C. Shahabi et al., Knowledge discovery from users web-page navigation, *Proc. Res. Issues in Data Eng. RIDE Workshop,* 1997.
73. P. Lougher and D. Shepherd, The design of a storage server for continuous media, *Comput. J.,* **36** (1): 32–42, 1993.
74. J. Hsieh et al., Performance of a mass storage system for video-on-demand, *J. Parallel and Distributed Comput.,* **30**: 147–167, 1995.
75. C. Martin et al., The Fellini multimedia storage server, in S. M. Chung (ed.), *Multimedia Information Storage and Management,* Norwell, MA: Kluwer, August 1996, Chapter 5.
76. *Sun™ MediaCenter™ Series, Server Models 5, 20, and 1000E,* Sun Microsystems, Inc., 2550 Garcia Ave., Mtn. View, CA 94043-1100, 1996.
77. *Starlight™ StarWorks™ 2.0,* Starlight Networks, Inc., 205 Ravendale Drive, Mountain View, CA 94043, 1996.

78. *The New Standard in Modular Video Server Technology, VIDEOPLEX Video-on-Demand,* Storage Concepts, Inc., 2652 McGaw Avenue, Irvine, CA 92714, 1995.

SHAHRAM GHANDEHARIZADEH
CYRUS SHAHABI
University of Southern California