

Singapore Management University Institutional Knowledge at Singapore Management University

Research Collection School Of Information Systems

School of Information Systems

3-1999

Resource Scheduling in a High-Performance Multimedia Server

Hwee Hwa PANG

Singapore Management University, hhpang@smu.edu.sg


Bobby JOSE

M. S. KRISHNAN

Compaq Services, Singapore

DOI: <https://doi.org/10.1109/69.761665>

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

 Part of the [Databases and Information Systems Commons](#), and the [Numerical Analysis and Scientific Computing Commons](#)

Citation

PANG, Hwee Hwa; JOSE, Bobby; and KRISHNAN, M. S.. Resource Scheduling in a High-Performance Multimedia Server. (1999). *IEEE Transactions on Knowledge and Data Engineering*. 11, (2), 303-320. Research Collection School Of Information Systems.

Available at: https://ink.library.smu.edu.sg/sis_research/112

This Journal Article is brought to you for free and open access by the School of Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email libIR@smu.edu.sg.

Resource Scheduling In A High-Performance Multimedia Server

HweeHwa Pang, Bobby Jose, and M.S. Krishnan

Abstract—Supporting continuous media (CM) data—such as video and audio—imposes stringent demands on the retrieval performance of a multimedia server. In this paper, we propose and evaluate a set of data-placement and retrieval algorithms to exploit the full capacity of the disks in a multimedia server. The data-placement algorithm declusters every object over all of the disks in the server—using a time-based declustering unit—with the aim of balancing the disk load. As for runtime retrieval, the quintessence of the algorithm is to give each disk advance notification of the blocks that have to be fetched in the impending time periods, so that the disk can optimize its service schedule accordingly. Moreover, in processing a block request for a replicated object, the server will dynamically channel the retrieval operation to the most lightly loaded disk that holds a copy of the required block. We have implemented a multimedia server based on these algorithms. Performance tests reveal that the server achieves very high disk efficiency. Specifically, each disk is able to support up to 25 MPEG-1 streams. Moreover, experiments suggest that the aggregate retrieval capacity of the server scales almost linearly with the number of disks.

Index Terms—Multimedia server, time-based storage scheme, declustering/striping, replication, look-ahead data retrieval.



1 INTRODUCTION

IN recent years, demand for multimedia applications that are capable of manipulating both continuous media (CM) data, such as video and audio, and non-CM data (e.g., text and images), has been growing rapidly. Some examples are interactive multimedia education [1] and news on demand [28]. Such an application requires the underlying multimedia server to organize and retrieve its data in a way that would meet the performance objectives of the application. For non-CM data, the performance objective is typically to minimize the response time, i.e., the elapsed time between the issuance of a data request and the delivery of the target data. In the case of CM data, in addition to minimizing response time, there is also a constraint on the acceptable time interval between the delivery of successive portions of the data. For example, successive frames of an MPEG-1 [13] video may have to be retrieved within $\frac{1}{30}$ second of each other in order to achieve jitter-free display. These stringent performance objectives raise a number of issues in the construction of multimedia servers.

The first issue in building a multimedia server is data placement, both within individual disks and across multiple disks. In traditional storage servers that support only non-CM data, the order in which different portions of an object are retrieved is usually immaterial; what matters is the time that is needed to retrieve all of the portions. In

contrast, a multimedia server needs to respect the temporal relationship between the components of a CM stream. For instance, it does not make sense to retrieve the frames of an MPEG video randomly. A multimedia server, therefore, needs to take into account this requirement for ordered retrieval in placing CM data on disk. In addition, the intradisk data-placement strategy should maximize the access efficiency of individual disks, while the interdisk data-placement policy should lead to a balanced load across all of the disks in the system.

While good data-placement policies are crucial, the ultimate performance determinant of a multimedia server is its data-retrieval algorithm. This algorithm comprises a component that decides when to request for each portion of an object, and a disk scheduling component that governs the way that the disks service their requests. Note that the need for the first component is unique to multimedia servers featuring CM data support; traditional systems that store only non-CM data simply request for all of the blocks of an object at once and let the disk scheduler figure out the best way to fetch these blocks. A big challenge in designing the data-retrieval algorithm is to optimize its handling of variable bit rate streams, which could overload a disk during some periods while leaving the disk lightly utilized at other times. In order for the system to perform well under such loads, the data-retrieval algorithm needs to be able to take full advantage of lull periods to fetch data blocks that will be needed shortly.

The multimedia server project at the Institute of Systems Science was initiated in April 1994. The ultimate goal of the project is to build a distributed server that is capable of supporting both CM and non-CM data. As a first step toward this goal, we have implemented a centralized CM server. The primary contribution of this paper is a set of data-placement and retrieval algorithms that enable our server to efficiently manage its disks, the bottleneck resources in

-
- H.H. Pang is with Kent Ridge Digital Laboratories, 21 Heng Mui Keng Terrace, Singapore 119613, Republic of Singapore. E-mail: hhpang@krdl.org.sg.
 - B. Jose can be contacted at 8842 Winding Way, Suite 313, Fair Oaks, CA 95628. E-mail: bobbyjose@yahoo.com.
 - M.S. Krishnan is with Compaq Services, Compaq Center Tampines Plaza, 5 Tampines Central 1 #05-01, Singapore 529541, Republic of Singapore. E-mail: krishnan.ms@digital.com.

the server. We focus on how the algorithms handle *independent* multimedia objects; the issue of scheduling multiple dependent objects has been addressed elsewhere (e.g., [8]). The data-placement algorithm declusters every object over all of the disks, using a time-based declustering unit, with the aim of balancing the disk load. Within each disk, objects that are accessed more frequently are allotted pages in the middle cylinders, whereas objects that are less popular reside in the outer and inner cylinders. As for runtime retrieval, the quintessence of the algorithm is to give each disk advance notification of the blocks that have to be fetched in the impending time periods, so that the disk can optimize its service schedule accordingly. Moreover, in processing a block request for a replicated object, the server will dynamically channel the retrieval operation to the most lightly loaded disk that holds a copy of the required block. Extensive evaluations confirmed that the combination of these algorithms results in a very efficient multimedia server. Our implementation manages to achieve near jitter-free retrieval for up to 25 concurrent MPEG-1 streams, each averaging 1.2 Mbits/sec, out of a single Seagate Elite 9 drive that is capable of reaching only a maximum *sequential* retrieval speed of around 6 Mbytes/sec. Moreover, experiments suggest that the aggregate retrieval bandwidth of the server scales almost linearly with the number of disks.

The remainder of this paper is organized as follows: In Section 2, we present a framework for this study by introducing a taxonomy of the design decisions, and by classifying several related work according to this taxonomy. Section 3 presents an overview of our multimedia server. Following that, the intradisk and interdisk data-placement/retrieval algorithms are introduced in Sections 4 and 5, respectively. Section 6 describes the accompanying memory management algorithm. Following that, Section 7 gives the results of a series of experiments that highlights the performance of our implementation, while Section 8 uses a simulation model to show how the server might scale up. Finally, our conclusions are presented in Section 9.

2 A TAXONOMY OF MULTIMEDIA SERVERS

A multimedia server comprises many components, among which are storage space management, data-placement, disk scheduling, and playback. As a result, a multimedia server implementor faces a wide range of design decisions. This section introduces a taxonomy of the design options that encompasses the algorithms that are studied in this paper, as well as other related algorithms proposed in the literature.

There are many possible ways to slice the design space for multimedia servers. The approach that we have taken is to follow the sequence of events that are associated with the data in a multimedia server, i.e., from compression, storage, to retrieval. The levels of the taxonomy are described below.

- 1) *Object bit rate*: The first decision facing an implementor is whether to design the server for *constant bit rate* streams or *variable bit rate* streams. On one hand, many compression standards, including the popular MPEG and JPEG, formats, produce variable bit rate streams. Consequently, a server that supports variable bit rate streams will be more versatile. On the other hand, managing only constant bit rate streams leads to a more predictable resource requirement. This greatly simplifies admission control and resource scheduling, and is attractive from a quality of service standpoint.
- 2) *Object partitioning*: To facilitate storage space management and object retrieval, a server has to partition a multimedia stream into a sequence of smaller units. For a variable bit rate stream, there are two natural ways to carry out the partitioning—*fixed duration (FD)* vs. *fixed-size (FS)*. In FD partitioning, the stream is divided into blocks that each last the same playback duration. This makes the arrival time of block requests at playback more predictable, but introduces space fragmentation problems. In contrast, FS partitioning divides the stream into blocks of equal sizes, which simplifies space management at the expense of unpredictable block request arrival times. We will discuss in detail the trade-offs between FD vs. FS partitioning in Section 4, when we present our design decisions. Of course, there is no difference between the two partitioning schemes in the case of constant bit rate streams.
- 3) *Disk scheduling*: At retrieval time, each disk usually has to service multiple playback streams. The default disk scheduling algorithms are *SCAN* and *CSCAN* [40]. Using these algorithms, a disk services the playback streams as its read/write head sweeps across the disk surface. This minimizes seek overheads, leading to very efficient disk utilization. However, *SCAN* and *CSCAN* are inadequate for variable bit rate streams that are partitioned in a size-based manner. The reason is that some blocks comprising these streams may have to be retrieved much faster than others. Consequently, a server that stores variable bit rate streams in a size-based fashion has to adopt a *priority-cognizant* disk scheduling algorithm, even at the expense of reduced disk efficiency.
- 4) *Interdisk placement*: Given the large amount of data and the number of concurrent users that multimedia applications typically need to support, the underlying servers are likely to be equipped with an array of disks. In storing its data, a multimedia server can do *clustering*, *synchronous declustering*, or *asynchronous declustering*. In the case of clustering, each object is kept in one disk whenever space permits, whereas declustering (or striping) entails a deliberate effort to spread an object over multiple disks. With asynchronous declustering (or coarse-grained striping), the smallest unit of data deposited at a disk each time is one block, whereas synchronous declustering (or fine-grained striping) stripes each data block over the disks. Compared to clustering, declustering leads to a more balanced load distribution; the drawback is that declustering increases the likelihood that an object becomes unavailable due to disk failures. Between the two variants, asynchronous declustering incurs lower disk latency overheads and is more scalable, whereas disk scheduling is much simpler with synchronous declustering [15].

TABLE 1
A TAXONOMY OF MULTIMEDIA SERVERS

Bit Rate	Data Unit	Disk Service	Interdisk	Intradisk	VCR Op Copy	Validate	Example
Constant	FS	SCAN	Cluster	Cluster	Separate	Simulate	[16], [24]
				Interleave	Separate	Analyze	[4], [6]
			Sync	Cluster	Same	Analyze	[29]
						Implement	[43]
			Async	Cluster	Same	Simulate	[36], [10], [11], [3]
						Implement	[17], [18], [9]
			Interleave	Separate	Analyze	[31]	
Variable	FS	Priority	Async	Cluster	Same	Implement	[12]
		SCAN	Cluster	Cluster	Separate	Implement	[19]
	FD	Priority	Sync	Cluster	Same	Simulate	[23]
						Implement	[41]
		SCAN	Cluster	Cluster	Same	Simulate	[21]
						Implement	[7]
		Async	Cluster	Same	Simulate	[22]	
	Interleave	Same	Analyze	[32]			

- 5) *Intradisk placement*: Having chosen an interdisk placement scheme, the next decision is whether to *cluster* or to *interleave* objects within each disk. Clustering fragments of the same object together is more straightforward to implement, whereas interleaving fragments belonging to different objects may reduce disk seeks if two or more object requests can be synchronized according to their storage patterns.
- 6) *VCR operations*: Another design decision concerns the way in which VCR operations like fast-play and reverse-play are supported. Some servers are able to vary the playback rate of a stream by reading data blocks faster/slower from the disk, or by skipping blocks periodically. These servers allow VCR operations on the *same* object that is used for normal playback. Other servers can only support a different playback rate by switching to a *separate* copy that was previously encoded at that rate.
- 7) *Validation*: Finally, a multimedia server implementor has to decide on a way to validate his/her architecture and algorithms. This could be done through *analytical modeling*, *simulation*, or *system implementation*. Strictly speaking, the validation approach is not a server design decision, but we included it in the taxonomy nonetheless as a server design is not complete until it has been validated.

Having described the design choices, we now present Table 1. The table shows how several recent studies on multimedia servers fit into the taxonomy, and how they contrast with our own approach. We have encountered

some difficulty in the classification process, as a number of studies do not address all of the choices in the design space. However, we have tried our very best to be accurate. We would also like to clarify that we have labeled some proposed servers as “constant bit rate” because support for variable bit rate streams were not explicitly addressed; this does not mean that those servers cannot be generalized to cater for variable bit rate streams. In the interest of space, we shall not describe the related work in detail here.

3 SYSTEM OVERVIEW

The architecture of our multimedia server is depicted in Fig. 1. As the figure shows, a user interacts with the server via a terminal, which is connected by network to the server. Each of the user’s request for an object is forwarded by the *network manager*, and then the *client manager* to the *directory manager*. The directory manager translates the object request into physical block requests to the *disk manager*, which retrieves the data blocks into memory. Every block contains a predetermined number of data frames, as we will explain in the next section. Finally, the *client manager* periodically extracts data frames from the blocks and sends the frames back to the user terminal via the network manager. This section first describes the details of the various components, before presenting the performance objectives of the server.

3.1 Directory Manager

The directory manager keeps track of the storage locations of all of the objects. It maintains a root directory in the form of a B^+ -tree. Each entry in the root directory points to a

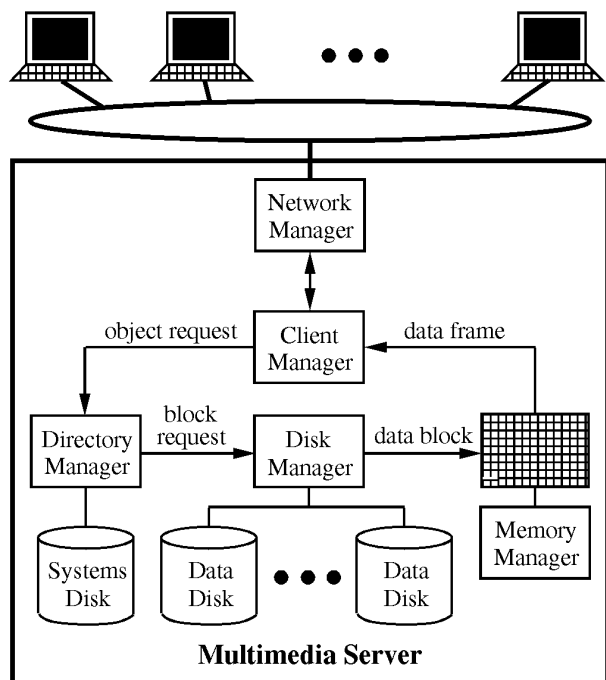


Fig. 1. System architecture.

subdirectory that records the physical block locations of an object. The subdirectory is also implemented as a B^+ -tree. Each record in the subdirectory corresponds to one data block, and is 9 bytes in size—2 bytes for the block id, 1 byte for the disk id, 4 bytes for the block's starting address on the disk, and 2 bytes for the number of disk pages that make up the block. All of the directories are stored on the system disk.

An object request includes two arguments: *ObjectId* and *BlockOffset*. Upon receiving such a request, the directory manager first identifies the subdirectory that corresponds to the required object by looking up *ObjectId* in the root directory. Next, using the storage location recorded in the subdirectory, a request for the *BlockOffset*th data block is generated and sent to the disk manager. After that, the directory manager periodically issues a request for the next block of data until the end of the object is reached, or until the user halts or terminates the retrieval. The length of a time period is determined by the data-placement algorithm to be described in Section 4.1. By suitably altering the block request pattern, the server can effect VCR-like operations other than just playback. For example, fast-forward is implemented by skipping data blocks, and reverse-play by decrementing *BlockOffset*.

3.2 Disk Manager

The disk manager controls the raw disks that hold the data blocks of all the objects. Each disk has its own block request queue, which is scheduled by the elevator algorithm [40]. In servicing a block request, the disk manager first checks whether the required block is in memory. If so, the request can be satisfied from memory and the disk manager proceeds to the next request. If not, a disk I/O is initiated to fetch the required data block into an I/O buffer that occupies physically contiguous locations. After that, the disk

manager acquires memory from the memory manager and transfers the data block to the acquired memory.

We adopt the above two-step approach because, due to memory fragmentation, the memory manager may have to allocate several smaller memory chunks, rather than a contiguous chunk of memory, for a block request. When this happens, the two-step approach is consistently faster than the simpler approach of fetching data directly into disjoint areas in memory. To further improve efficiency, each disk is assigned two I/O buffers so that it can fetch data for the next request into one I/O buffer even as the data block for the previous request is being transferred from the other I/O buffer to memory.

3.3 Memory Manager

The memory manager is responsible for allocating memory to satisfy block requests, and for reclaiming used memory. To limit fragmentation, the system memory is divided into fixed-sized pages, and each block request is given the minimum number of memory pages that will meet its requirement. A memory page can either be "New" if it contains fresh data, or "Used" if its data have been consumed and it is eligible to be reclaimed. The memory manager delays the reclamation process as long as possible, in the hope that future block requests can be satisfied from memory, thereby eliminating some disk I/Os.

3.4 Client Manager

As the description of the above components shows, an object request from a user will cause a steady stream of data blocks that make up the object to be fetched into main memory. The task of the client manager is to periodically send the data blocks of each object onto the network for decoding and display at the user terminal. The client manager also assists in memory management. Initially, all memory pages that are assigned for a block request are marked as "New." The client manager is responsible for changing the status of these memory pages to "Used" once all of the embedded data frames have been extracted.

3.5 Performance Objectives

One performance objective of the multimedia server is to minimize response time. For a non-CM object retrieval, this is defined as the time that the system takes to return the entire object. In the case of a CM object request, response time is the elapsed time before the first data block of the object is delivered. Another performance objective is to ensure the smooth delivery of CM objects. The server strives to minimize response times for both continuous and non-continuous media retrievals without jeopardizing the delivery of CM objects, i.e., smooth delivery of CM objects takes precedence over response time considerations.

To maximize the number of retrieval streams that a multimedia server can support while still meeting the above performance objectives, it is necessary to streamline the operation of the server, in particular, the bottleneck resources in the system. We expect the system disk to be relatively lightly loaded since the size of the directories are very much smaller than the size of the data: At 8 Kbytes per page and 9 bytes per directory entry, each directory page is capable of holding the physical locations of about

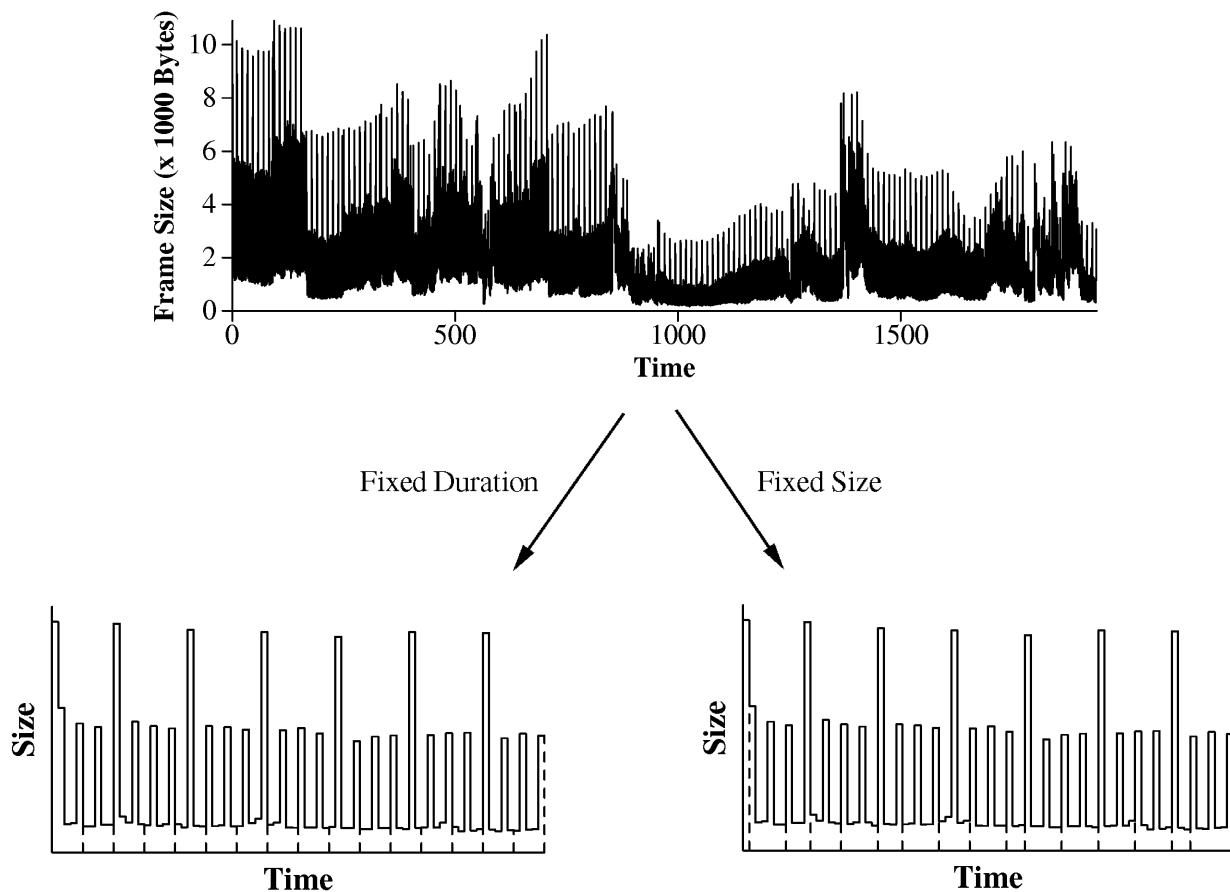


Fig. 2. Fixed size vs. fixed duration.

600 blocks. This means that the directory manager only fetches 1 page for every 600 data blocks, equivalent to 600 tracks since the average size of a block is chosen to be one track as described below, that the disk manager has to retrieve. Clearly, the data disks are the performance bottlenecks for which efficient data-placement and retrieval algorithms need to be developed, hence the focus of this paper.

4 INTRADISK DATA-PLACEMENT AND RETRIEVAL

Having presented an overview of the multimedia server, we now focus on how its components work in unison to produce a highly efficient server. This section presents the data-placement and retrieval algorithms for each individual disk; interdisk data-placement and retrieval will be introduced in the following section.

4.1 Data Placement

Depending on its compression rate and duration, the storage space requirement of a continuous media object may range from a few megabytes to a few gigabytes. The huge space requirement precludes a multimedia server from retrieving the object in its entirety all at once, as this is likely to result in very high buffer demands and unacceptably long waiting times for other retrieval requests. For this reason, a continuous media object typically has to be manipulated as a series of smaller-size blocks. There are two alternative formats for splitting the object: Fixed Size and Fixed

Duration. Fig. 2 gives an example showing how an MPEG video object is split under the two formats, with the dotted vertical lines in the figure indicating block boundaries.

The Fixed Size (FS) format splits a continuous media object into data blocks that are equal in size. An obvious advantage of this format is that the buffer requirement of each retrieval stream is constant, thus simplifying memory management. However, since the retrieval bandwidth requirement of an object is likely to fluctuate over time, the data contained in each block do not span the same time duration, as the irregular intervals between the dotted lines in Fig. 2 show. As a result, the time allowance for fetching a block of an object is not fixed, but depends on when the previous block gets consumed. This may require the disk manager to give priority to urgent block requests. As shown in [5], priority scheduling causes a disk to incur extra head movements and degrades its throughput. Consequently, the FS approach is conducive to efficient memory management but is likely to hinder productive disk utilization.

Using the Fixed Duration (FD) format, the system time is partitioned into equal time slices. Instead of insisting on a constant block size, an object is stored as a sequence of variable-size blocks that each contain enough data to span a time slice, as the constant interval between the dotted lines in Fig. 2 demonstrate. This fixes the time allowance for fetching each data block of an object to one time slice after its previous block's retrieval deadline. By batching block requests and by scheduling them together at the beginning

of each time slice, a disk is given an opportunity to optimize its service schedule within the time slice, e.g., according to the elevator algorithm [40]. The drawback of the FD approach is that, since data blocks may vary in size, the number of buffer pages that are needed to support a retrieval operation may fluctuate over time. These fluctuations in buffer requirement impose extra demands on the memory manager compared to the FS scheme.

We decided to base our multimedia server on the FD approach for two reasons. First, with current storage technologies, the disks, rather than CPU or memory, are the performance bottlenecks. The second reason is that extra buffers can be put to good use in other ways, e.g., for caching data that are frequently accessed, whereas inefficient disk usage leads to unproductive overheads that limit overall system performance. For these reasons, we deemed it important to adopt a scheme that utilizes the disks most efficiently, hence our choice of the FD format.

Having selected the Fixed Duration format, we next tackled the task of picking an appropriate time slice, which is equivalent to selecting a “good” average data block size. On one hand, we would like the time slice to be large so as to reduce the relative amount of time that the disks spend on seek and rotational delays, thereby improving disk access efficiency. On the other hand, the time slice should not be too large because the buffer demands of a retrieval operation and the waiting times that it experiences are both proportional to the length of the time slice. To strike a balance between disk access efficiency, buffer demand and waiting time, we chose a time slice that translates to an average block size of one track, i.e.,

$$\text{Time Slice} = \frac{\text{Average Track Size}}{\text{Average Object Retrieval Rate}}.$$

Using this time slice, the multimedia server can proceed to split each continuous media object into data blocks. Within each disk, those data blocks belonging to objects that are more frequently accessed are placed in the middle cylinders of the disks, in order to minimize seek delays for those objects. Objects that are not as commonly requested for are held in the outer and inner cylinders. All data blocks of the same object are assigned to physically contiguous sectors as far as possible. We chose to store objects in this manner, rather than to interleave data blocks of different objects as advocated in [35], [4], [31], for a number of reasons: First, when a disk is handling variable bit rate multimedia objects like MPEG videos, a contiguous placement allows the disk to combine those block requests that are leftover from an overloaded time slice with the block requests from a subsequent time slice. This would eliminate seek delays for the leftover block requests, thus helping the disk to clear its backlog of requests quickly after an overload. Second, we wish to avoid the extended start-up times that interleaving incurs as a new request waits in line for the region that holds its first required block to come under the disk head. A detailed comparison of the contiguous vs. interleaving schemes can be found in [33].

4.2 Data Retrieval

The quintessence of the data-retrieval algorithm that drives the server is to use any excess disk capacity in each time slice to carry out advance data fetching, so as to avert disk overloads during future time slices that involve heavy data retrievals. To simplify the description of the algorithm, in this section we will focus on a server with only one data disk, although the algorithm is fully capable of driving multiple disks.

The data-retrieval algorithm allows the disk manager to look ahead at (most of) the block requests that need to be satisfied in the next *LookAhead* + 1 time slices. There are two methods to achieve this. The first method is for the directory manager to request for the first *LookAhead* + 1 blocks all at once when a new CM stream is started. After this, the directory manager settles into a routine of requesting for one block per time slice until the end of the target object is reached, or until the retrieval operation gets interrupted. The disk manager is given only one time slice to deliver the first data block, and it has to follow up with another block every subsequent time slice. This method makes data available within one time slice after the start of a new stream, but does not allow the disk manager to look ahead at the initial block requests of any stream. Another method is to have the directory manager issue one block request per time slice right from the start of a new stream, while giving the disk manager up to *LookAhead* + 1 time slices to service its block requests. The second method has the advantage of enabling the disk manager to look ahead at *all* of its block requests, but incurs a *LookAhead*-time-slice delay in start-up time. We shall call the first method the *Immediate* variant, for immediate data delivery, and the second method the *Delay* variant.

At each disk, block requests are queued by the time slice when they are required for display, with requests belonging to the most imminent time slice being serviced first. Whenever the disk becomes free, the disk manager will send a batch of requests at the front of the queue for processing. The size of each batch varies, and is determined so that the requests are estimated to complete by the end of the current system time slice. The estimation is based on the disk model proposed in [38]. The reason for scheduling requests in batches is so that within each batch, requests can be processed according to the SCAN/elevator algorithm [40]. This allows block requests that are more urgently needed to be served ahead of other less critical requests, without resorting to a pure deadline-based disk scheduling approach that degrades throughput [5]. The batched SCAN approach is similar to the group sweeping scheme proposed in [45].

Since the disk model cannot be expected to capture the disk behavior perfectly, the disk could complete a batch of requests before or after the current system time slice. Consequently, the disk manager needs to compensate by inserting more or fewer requests in the next batch, so that they will occupy the disk until the end of the following time slice. In this way, the disk can be kept busy so long as there are pending requests, thus smoothing out some of the load fluctuations across time slices.

Fig. 3 illustrates how both variants might react to the introduction of a stream A in the midst of retrieving stream B.

Immediate Variant								
Time Period	i	i + 1	i + 2	i + 3	i + 4	i + 5	i + 6	i + 7
Dir Mgr Request	A ₁ , A ₂ , A ₃ , A ₄ , A ₅ , B ₂₀	A ₆ , B ₂₁	A ₇ , B ₂₂	A ₈ , B ₂₃	A ₉ , B ₂₄	A ₁₀ , B ₂₅	A ₁₁ , B ₂₆	A ₁₂ , B ₂₇
Disk Service	B ₁₉	A ₁	A ₂ , A ₃ , A ₄	A ₅ , A ₆ , B ₂₀	B ₂₁	B ₂₂ , B ₂₃ , B ₂₄	A ₇ , A ₈ , A ₉	A ₁₀ , A ₁₁ , B ₂₅
Client Display	B ₁₄	B ₁₅	A ₁ , B ₁₆	A ₂ , B ₁₇	A ₃ , B ₁₈	A ₄ , B ₁₉	A ₅ , B ₂₀	A ₆ , B ₂₁
Delay Variant								
Time Period	i	i + 1	i + 2	i + 3	i + 4	i + 5	i + 6	i + 7
Dir Mgr Request	A ₁ , B ₂₀	A ₂ , B ₂₁	A ₃ , B ₂₂	A ₄ , B ₂₃	A ₅ , B ₂₄	A ₆ , B ₂₅	A ₇ , B ₂₆	A ₈ , B ₂₇
Disk Service	B ₁₉	A ₁	A ₂ , B ₂₀	B ₂₁	B ₂₂ , B ₂₃ , A ₃	A ₄ , A ₅ , B ₂₄	B ₂₅ , A ₆	A ₇ , B ₂₆
Client Display	B ₁₄	B ₁₅	B ₁₆	B ₁₇	B ₁₈	B ₁₉	A ₁ , B ₂₀	A ₂ , B ₂₁

Fig. 3. Example of data-retrieval schedule, *LookAhead* = 4 time slices.

In the figure, A_i and B_i denote the i th data block of streams A and B, respectively. The schedule shows that, under the *Delay* variant, the disk manager elected to skip stream B in time slice $i + 1$ in order to service a large block A_1 . Also, for time slice $i + 4$, the disk manager squeezes in one more request B_{23} , as B_{22} and A_3 are expected to complete early. In such situations, the batched SCAN approach saves on disk seek operations as consecutive blocks of the same video are placed on contiguous disk sectors as explained previously. Thus, the retrieval algorithm gives the disk manager significant flexibility. Obviously, the higher *LookAhead* is set to, the better the retrieval algorithm performs as the data server has more leeway in scheduling its requests. In practice, though, the value of *LookAhead* is capped by response time and buffer space considerations.

Note that our retrieval algorithm simply enables the disk manager to *look ahead* at the block requests that it has to service. The decision to fetch which particular blocks is made dynamically during the lifetime of a retrieval operation, based on load conditions and the time slice when each block is needed. For example, the data server may elect to skip the block request of a stream in one time slice, and to make up in a later time slice by fetching two blocks for the stream, as happens to stream B in time slices $i + 1$ and $i + 4$ in Fig. 3. It, therefore, goes beyond prefetching algorithms, such as those proposed in [30], which only require a pre-determined number of blocks to be read into memory before a video playback begins. The algorithm is also more flexible than the scheme in [12], which mechanically prefetches data after reading every disk block.

5 INTERDISK DATA PLACEMENT AND RETRIEVAL

While the data-storage and retrieval algorithms described in the previous section are designed to maximize the efficiency of individual disks, they need to be complemented by interdisk data-placement and retrieval algorithms that ensure even load distribution among the disks. Our data-placement algorithm is based on a time-based declustering approach, while both the placement and retrieval algorithms support an object replication mechanism for better system reliability and performance. The details of the declustering and object replication mechanisms are presented below.

5.1 Declustering

To deliver good system performance, the interdisk data-placement algorithm must be designed to balance the load on all of the disks. In addition, the data-placement algorithm has to allow each disk to be utilized efficiently. This necessitates a trade-off between fine-grain declustering, which balances load by enabling parallel retrieval from all disks but produces high disk seek and synchronization overheads [34], and clustering, which leads to efficient utilization of individual disks but subjects the system to load imbalances as we will demonstrate later. A compromise between these two extremes is coarse-grain declustering. The aim of coarse-grain declustering is to let all the disks take turns to service every retrieval request. This avoids saddling a few disks with a disproportionately large number of retrieval requests for extended periods of time, and is expected to achieve statistically balanced load distributions [14].

The coarse-grain declustering strategy that we adopt divides an object into several data fragments. Each data fragment, in turn, comprises $DeclUnit \times LookAhead + 1$ data

blocks, where *DeclUnit* is a parameter of the interdisk data-placement algorithm, *LookAhead* is a parameter of the intra-disk data-retrieval algorithm, and each block contains enough data to last one time slice. The first fragment is assigned to a disk i , and subsequent fragments are assigned to disks $i + 1, i + 2, \dots, 1, 2, \dots, i - 1$, respectively. This assignment pattern is repeated for all of the fragments. The reason for choosing a declustering unit of $DeclUnit \times LookAhead + 1$ blocks is to maximize the efficiency of look-ahead retrieval by allowing data blocks that are fetched in the same retrieval cycle to reside in contiguous sectors of the same disk.

One important characteristic of our declustering strategy that distinguishes it from the schemes proposed in [17], [37] is that our declustering unit has a fixed duration, rather than a fixed size. This ensures that block requests are due only by the end of a time slice. As mentioned previously, this allows the disks to optimize their service schedules by batching block requests, and by scheduling them together at the beginning of each time slice. Our coarse-grain strategy is also different from the streaming RAID of StarWorks [41]. Although a time-based storage unit is used in streaming RAID, its fine-grain declustering approach stripes each data block across all of the disks. As explained above, we avoid this approach because the resulting disk seek and synchronization overheads can potentially make the approach counterproductive.

5.2 Object Replication

As mentioned earlier, our server offers an option to replicate objects that are more frequently accessed, so that block requests for a popular object can be channeled to a copy that resides on a relatively lightly loaded disk. Given that, in practice, a small number of objects (say, 20 percent of the objects) account for most of the accesses in the system (say, 80 percent of the accesses) at any time [24], this option allows a storage system to derive most of the load balancing benefits of replicating every object without incurring the same prohibitively large storage overheads.

When the object replication option is enabled, the server keeps a replica for every popular object. Like the primary copy, the replica is declustered across all of the disks. The disk assignments for the primary copy and the replica of a data fragment follow the *chained declustering* scheme proposed by Hsiao and DeWitt [20], where the replica is always placed one disk ahead of the primary copy. In addition, we require the replica of an object's data fragment to physically adjoin the primary copy of the object's subsequent data fragment. This requirement allows the retrieval of data blocks from two consecutive fragments to be combined by simply directing the block requests for the first fragment to its replica. Fig. 4 illustrates how the data fragments of a popular object are assigned to the disks. In the figure, F_j and F_j^r refer to the primary copy and the replica of fragment j , respectively.

In order to take advantage of the replicas, the directory manager described in Section 3.1 needs to be modified slightly. As before, the directory manager issues block requests on behalf of all active retrieval operations at the

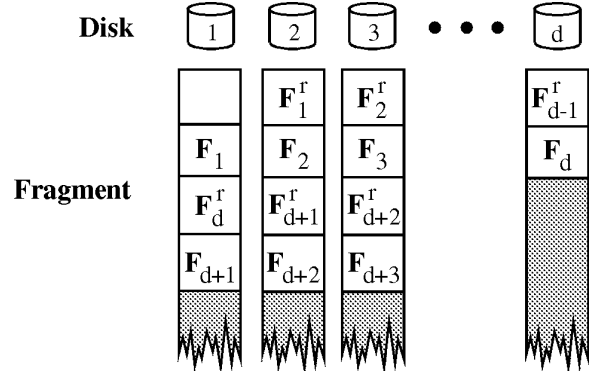


Fig. 4. Data placement example.

beginning of each time slice. However, the directory manager now has the added responsibility of deciding whether a block request for a popular object is to be satisfied by the disk that holds the primary copy, or by the disk with the replica. In making this decision, an obvious choice is to select the disk that is currently more lightly loaded in order to even out the load on the two disks. At the same time, however, the efficient operation of the disks should not be compromised. Specifically, the directory manager needs to avoid switching repeatedly between the primary copy and the replica for successive blocks in the same data fragment. The reason is because this hampers the disks' ability to combine the retrieval of physically contiguous blocks to lower overheads. Therefore, a retrieval operation is limited to only one switch from the primary copy to the replica within each data fragment. Once a switch is made, subsequent block requests will be directed to the replica until the operation requires a block from a different data fragment.

The above data-retrieval mechanism is implemented as follows: The directory manager keeps a running cumulative page count of the block requests that have been directed to each disk in the current time slice, and also *LastDisk_i*, the id of the disk where each retrieval operation i 's last block request was directed to. Whenever there is a block request with two alternative addresses, the directory manager checks the address of the replica to see if it belongs to the disk indicated by the issuing operation's *LastDisk_i*. If it does, there is a prior decision to supply the blocks in the current data fragment from its replica, so the block request is sent to disk *LastDisk_i*. If the replica's address does not belong to disk *LastDisk_i*, then the directory manager is free to choose between the primary address and the replica address. In this case, the directory manager simply channels the request to the disk that has a lower cumulative page count; if there is a tie, the primary address is chosen.

To illustrate the load balancing mechanism, let us trace the retrieval of the object depicted in Fig. 4. The first request, targeted at the first block in fragment 1, can be serviced by either disk 1 or disk 2. Assuming that the current cumulative page count of the two disks are 50 and 60, respectively, the block request is sent to disk 1. The second request, issued one time slice later, again offers a choice of disks. Suppose that the cumulative page counts stand at 45 and 30, the directory manager now sends the request to disk 2 in an attempt to balance the disk loads. After another

time slice, the directory manager issues a request for the third data block to disk 2 regardless of the current cumulative page count of disks 1 and 2, as there is a prior commitment to retrieve the remaining blocks in fragment 1 from disk 2. Taking the example one step further, suppose that the fourth block of the object is in fragment 2. The directory manager is again free to choose between the original copy and the replica since the retrieval operation has moved on to a different data fragment. The above process is repeated until all of the data blocks have been fetched, or until the retrieval operation gets interrupted.

Besides enhancing load balance, replicating frequently accessed objects also has the benefit of improving system robustness, one of our design goals (see Section 3): Since every data fragment of a replicated object has a replica that resides on a different data server from the primary copy, the entire object remains accessible in the event of a data-server crash or disk crash. Moreover, reconstruction of a replicated object after a crash can be done on-the-fly by copying the blocks constituting the object as they are retrieved to satisfy user requests. In view of the derivable benefits, object replication appears to be attractive despite the potentially large space overhead involved. This is especially so given the observation that most of the requests in a system are usually directed at a small fraction of popular objects. One of the objectives of this paper is to quantify the load balancing effect arising from object replication and dynamic copy selection.

6 MEMORY MANAGEMENT

While the data-storage and retrieval algorithms described in the previous sections are designed to maximize the efficiency of the disks, which are the system bottlenecks, the memory manager also plays an important role in determining the performance of the server. The memory manager has to strike a good compromise between two trade-offs: On one hand, the memory manager should keep enough free memory pages so it can satisfy the disk manager immediately when the latter requests for memory for a block request. On the other hand, the memory manager should delay reclaiming memory pages as long as possible in case those pages contain data that are needed by future block requests. This section presents the memory management algorithm in detail.

6.1 Memory Organization

The memory manager divides the system memory into fixed-sized pages. Each memory page either belongs to a *free list*, or it is part of a linked list of memory pages that together hold a data block in the *object cache*. The memory pages in the free list are assigned to hold new data blocks after they have been fetched from disk into an I/O buffer. The organization of the object cache is illustrated in Fig. 5. As the figure shows, the memory-resident data blocks are grouped by objects. Furthermore, data blocks belonging to the same object are chained in ascending order by their *BlockOffsets*.

To keep track of the status of data blocks, the client manager maintains an array of block pointers, one for each

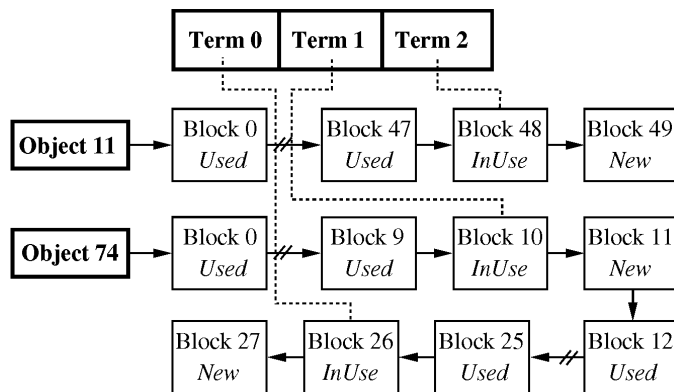


Fig. 5. Object cache.

active user terminal. Each block pointer leads to a data block that is marked as “InUse.” It is from this block that data are periodically extracted and sent to the corresponding user terminal. Once all of the data have been extracted from the “InUse” block, it will be marked as “Used,” and the block pointer will advance to the next data block in the chain. That block, which should have been previously marked as “New” to indicate that it contains fresh data, now becomes the “InUse” block. Since the directory manager issues block requests *LookAhead* time slices in advance, there could be up to $LookAhead + 1$ “New” blocks immediately trailing an “InUse” block.

The above memory organization is designed to facilitate quick identification of memory-resident data blocks that can satisfy block requests without necessitating disk I/Os. In serving a block request for a user terminal, the memory manager first scans through the chain of data blocks in the object cache that belong to the required object. If the required block is found in the object cache, the block request is satisfied immediately by changing the status of the block to “New.” A disk I/O is initiated for the block request only if the required data block is not in the object cache.

6.2 Memory Reclamation

While leaving “Used” data blocks in the object cache can improve the performance of the server by eliminating some disk I/Os, the free list will eventually become empty unless the memory pages occupied by “Used” data blocks are reclaimed. To reclaim memory, the memory manager first scans through every chain of data blocks in the object cache, recording every subchain of “Used” data blocks as it encounters them. Fig. 6, with the chains of “Used” data blocks shaded, illustrates this step.

Once this step is completed, memory reclamation begins. Starting with the longest chain, the memory manager snips off chain after chain of “Used” blocks from the object cache, returning the memory pages occupied by the “Used” data blocks to the free list. The reason for reclaiming the longer chains first is that these blocks are accessed less recently, hence they are likely to belong to a less frequently used object than those blocks in the shorter chains. For example, in Fig. 6 the shaded chain of object 11 is longer than any of the two shaded chains of object 74, suggesting that object 11 is retrieved less frequently, and that the benefit of keeping it in the object cache is lower. This reclamation process

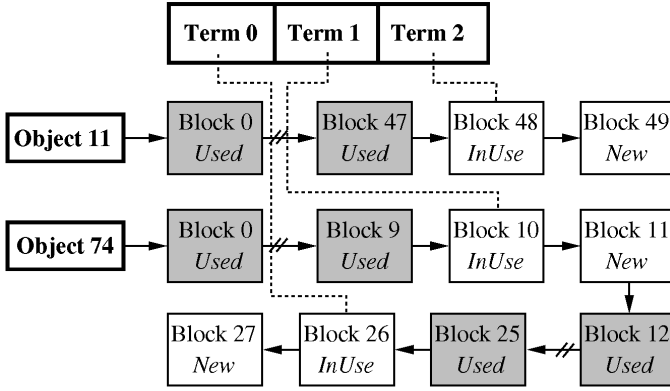


Fig. 6. Reclaimable memory.

continues until the free list has been replenished to a predetermined level, or until all of the “Used” data blocks have been reclaimed. If the reclamation process can be halted by recovering only the memory occupied by part of a chain, then the memory manager frees up the data blocks on the right side of the chain. Here the rationale is that data blocks that are near the start of an object are likely to be needed by a subsequent request for the same object before those data blocks toward the end. To illustrate this, consider the shaded chain comprising blocks 12 to 25 of object 74 in Fig. 6. Since block 12 will be required by terminal 1 before block 25, the reclamation process should start with block 25 and work toward block 12. Fig. 7 shows the status of the object cache after a memory reclamation.

The condition that triggers the memory reclamation process, and the condition that halts the process once it is started, are determined by two tunable parameters, *LoThres* and *HiThres*, respectively: Once the fraction of memory pages that remain in the free list falls below *LoThres*, the memory manager will begin to reclaim memory from the object cache, until the fraction of memory pages in the free list rises above *HiThres*. The *LoThres* parameter should not be set too high, which would deprive the object cache of memory and thus reduce its effectiveness, neither should it be set so low that the free list are allowed to become empty, thereby forcing the disks to come to a halt while the memory manager recovers memory for the data in the I/O buffers. As for the *HiThres* parameter, its value should

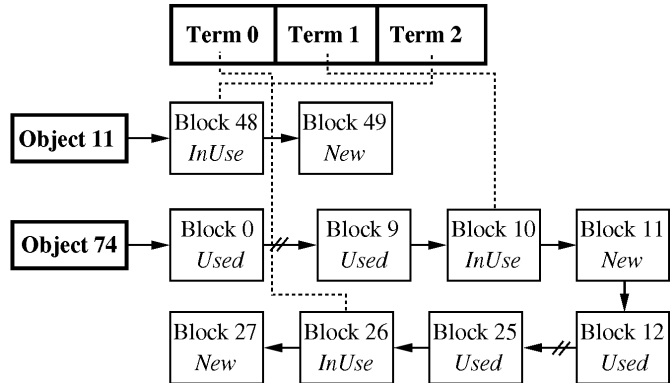


Fig. 7. After memory reclamation.

be high enough so that memory reclamation is not required too frequently, but not so high as to cause data blocks to be removed from the object cache unnecessarily. In our implementation, we have set *LoThres* and *HiThres* to 10 percent and 40 percent, respectively, and found these settings to work satisfactorily.

7 SERVER PERFORMANCE

In the previous sections, we have introduced the data-placement and retrieval algorithms that our multimedia server employs to manage individual disks, as well as an array of disks. We now present the implementation platform and the performance of the server.

7.1 Experiment Set-Up

We have implemented our multimedia server on a Sparc 20 workstation according to the system architecture depicted in Fig. 1. The Sparc 20, which runs Solaris 2.4, is equipped with a Seagate ST31200W system disk that we use to store the directories. Two external Seagate Elite 9 disks serve as data disks. These disks have an average seek time of 11 msec and a transfer rate of 6 Mbytes/sec. The Sparc 20 has a total of 96 Mbytes of memory, though the memory manager allocates only 1.25 Mbytes/terminal for the experiments reported in this paper.

As for the workload model for the experiments, each of the *TerminalsPerDisk* user terminals attached to the server repeatedly goes through a cycle comprising a thinking phase and an object retrieval phase. The thinking time, which models the time spent on picking a multimedia object, follows an exponential distribution with a mean of *ThinkTime* sec. *AccessSkew%* of the time, the chosen object is one of the popular objects that make up *HotSize%* of the total object population. The various experiment parameters, together with their default values, are summarized in Table 2.

Since variable bit rate video objects pose the toughest challenge to the multimedia server, we populate the two data disks with copies of the five MPEG-1 video clips described in Table 3. Different copies of the same clip are treated as distinct objects. Using the average bandwidths of the five videos and the formula in Section 4.1, the “ideal” time slice is 0.51 sec; we use a time slice of 0.5 sec in this study to simplify our discussion.

As the focus of this paper is on the multimedia server and not on the interconnection network between the server and the user terminals, we assume that there is a sufficiently fast network, together with an appropriate set of communication protocols. This network infrastructure would enable a user terminal to experience the same interframe waiting times as those observed by the client manager of the server. For this reason, we will measure the server performance at the client manager, rather than at the user terminals.

7.2 Performance Metrics

The primary performance metrics are the *% Late Frames* and the *Latency*. *% late frames* is defined as the percentage of data frames that are retrieved late, thereby causing jitters in the delivery of the video streams. *Latency* measures the average amount of time that a frame is late by, or the average

TABLE 2
EXPERIMENT PARAMETERS

Parameter	Meaning	Default
<i>LookAhead</i>	Number of time slices to look ahead	4
<i>DeclUnit</i>	Declustering unit = (DeclUnit × LookAhead + 1) time slices	1
<i>TerminalsPerDisk</i>	Number of terminals or retrieval streams per data disk	—
<i>ThinkTime</i>	Average thinking time	0 sec
<i>AccessSkew%</i>	Percentage of requests for popular objects	80%
<i>HotSize%</i>	Percentage of objects that are popular	20%

TABLE 3
CHARACTERISTICS OF VIDEO CLIPS

Trace	Duration (sec)	# Frames	Frame Size (Bytes)			Rate (Kbps)	
			Avg.	Min.	Max.	Avg.	Peak
1	64.7	1,940	1,503	132	8,743	45	1,679
2	68.8	2,065	6,502	2,501	18,233	181	3,501
3	44.8	1,344	3,845	1,556	7,814	116	1,498
4	66.4	1,992	3,656	1,149	12,546	110	2,409
5	22.2	667	9,945	3,646	16,982	298	3,261
Average	53.4					150	

duration of a jitter. Each experiment was run for 3 hours, allowing a minimum of 2,000 object retrievals. We also verified that the size of the 90 percent confidence intervals for % late frames, computed using the batch means approach [2], was within a few percent of the mean in almost all cases.

7.3 Baseline

The objective of the baseline experiment is to profile the performance of the intradisk data-placement and retrieval algorithms. For this experiment, we use only one of the data disks; the other data disk is left idle. As for the workload, *TerminalsPerDisk* ranges from 20 to 28, and *ThinkTime* is set to 0. Furthermore, 80 percent of the retrieval requests are for 20 percent of the objects, i.e., *AccessSkew%* and *HotSize%* are set to 80 percent and 20 percent, respectively. Finally, the *LookAhead* parameter of the data-retrieval algorithm is set to 4. The % late frames and latency values produced by the Immediate and Delay variants of the retrieval algorithm are plotted in Figs. 8 and 9. For comparison purposes, we also include in these figures the curves representing the system performance without look-ahead retrieval, i.e., with *LookAhead* = 0, which we label as “NoLookAhead.”

We first analyze the performance of the Immediate variant. Figs. 8 and 9 show that Immediate consistently produces lower % late frames and latency than NoLookAhead. The reason is that, with the exception of the initial few blocks of each object, all of the block requests arrive at the disk 4 time slices earlier. This allows the disk to cut seek costs by combining the retrievals of up to 5 consecutive blocks each time. Moreover, the look-ahead period allows the data disk to start servicing the next round of requests

right after the most immediate round of requests have been processed, thus averting potential overloads in the near future or at least reducing their impact.

Having examined Immediate, we now proceed to evaluate the Delay variant of the retrieval algorithm. The curves in Figs. 8 and 9 show that Delay performs better than NoLookAhead and Immediate. In fact, with Delay, the disk manages to limit the average duration of jitters to within one frame (indicated by the dotted line in Fig. 9) until the number of terminals exceeds 25. There are two reasons for Delay’s superior performance. The first reason is that every retrieval stream is given a 4-time-slice headstart. This headstart gives the retrieval streams a cushion against temporary overloads. Another reason is that every block request arrives early here. This contrasts with Immediate which demands that the first few blocks of each data stream be scheduled for fetching immediately upon arrival, thus hindering the disk’s ability to do look-ahead fetching whenever a new retrieval stream enters the system.

The results of this experiment show that the look-ahead feature of the intradisk retrieval algorithm can help to reduce disk access costs and to avert temporary overloads. In particular, if a 4-time-slice start-up delay can be tolerated, the Delay variant of the algorithm is very effective in ensuring smooth data delivery at the maximum number of concurrent streams.

7.4 Sensitivity of LookAhead Parameter

In the previous experiment, we have set the *LookAhead* parameter of the intradisk retrieval algorithm to 4. We now vary this parameter to study its performance impact. We will focus only on the Delay variant here since

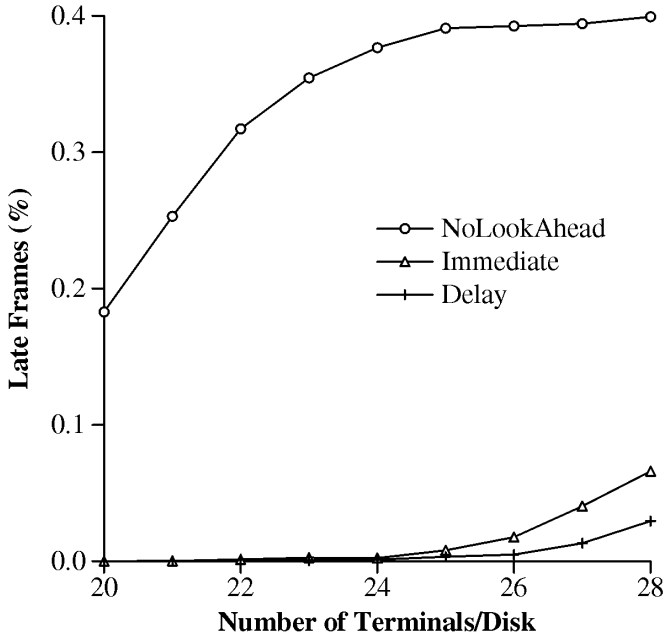


Fig. 8. Jitter ratio.

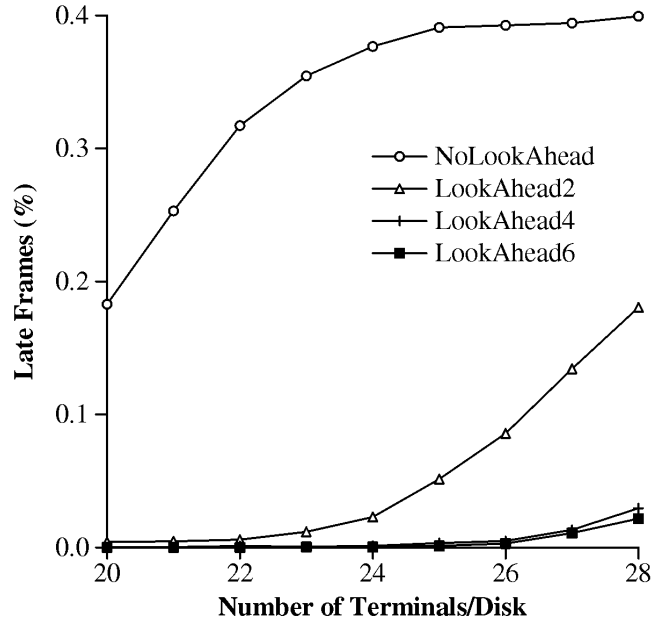


Fig. 10. Jitter ratio.

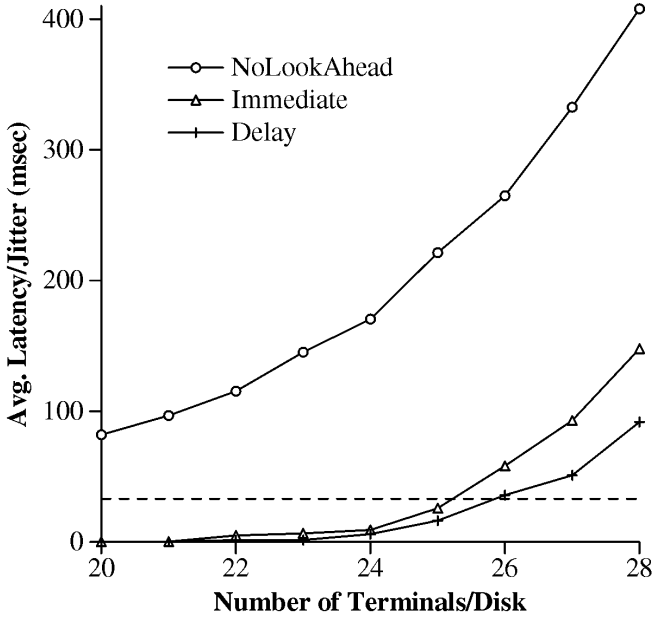


Fig. 9. Latency.

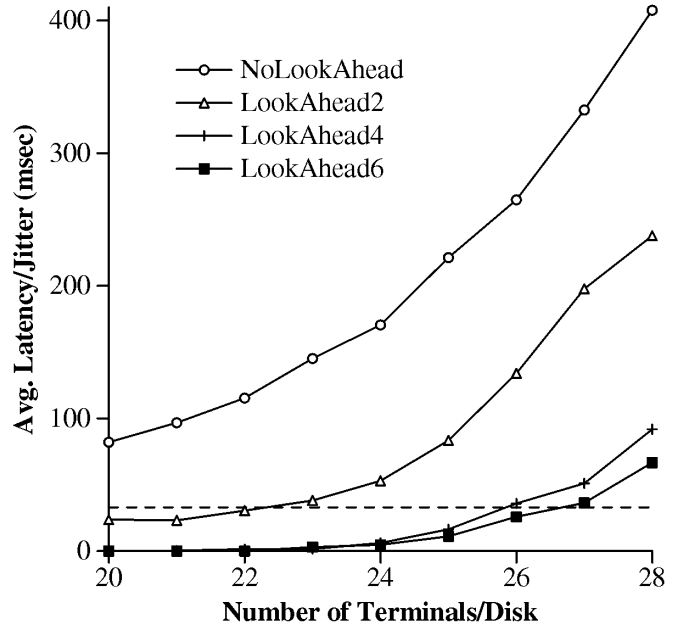


Fig. 11. Latency.

it outperforms the Immediate variant, as demonstrated in the base-line experiment.

Figs. 10 and 11 plot the % late frames and latency as a function of the number of terminals. As expected, a higher *LookAhead* value enables the system to achieve lower % late frames and latency

- 1) by providing a larger cushion against temporary overloads,
- 2) by combining the retrievals of a larger number of contiguous blocks to lower seek costs, and

- 3) by servicing block requests from future, overloaded time slices even earlier to prevent the overloads from taking place.

We observe that the two figures show a marked performance improvement from NoLookAhead to LookAhead = 2. The gain from LookAhead = 2 to LookAhead = 4 is smaller, but still sizeable. Beyond LookAhead = 4, further increases in the *LookAhead* parameter lead only to marginal reductions in % late frames and latency. We repeated this set of experiments with the Immediate variant (results not shown), and derived the same observations. In practice, the number

of time slices that a system should look ahead at is determined by the maximum tolerable response time and the amount of available memory, which in the Delay variant of the retrieval algorithm is proportional to *LookAhead*. The appropriate *LookAhead* setting also depends on the stability of the retrieval streams—a large *LookAhead* setting may actually be detrimental to performance if a substantial number of retrieval streams get terminated prematurely, rendering the prefetched data blocks useless.

7.5 Clustering vs. Declustering

The objective of the next experiment is to assess the relative merits of clustering vs. declustering in interdisk data placement and retrieval. To do this, we activate both data disks, and we set the *LookAhead* parameter of the intradisk retrieval algorithm to 4. As the previous experiment shows, this setting allows the multimedia server to reap most of the benefits of look ahead retrieval without delaying response time too much. Finally, the object replication option is disabled for this experiment.

Figs. 12 and 13 plot the % late frames and latency values for four *DeclUnit* settings—0, 1, 2, and ∞ . The behavior of these settings are denoted by *Decluster0*, *Decluster1*, *Decluster2*, and *Cluster* (since an infinite declustering unit leads to clustering) in the two figures. With *LookAhead* set to 4 and declustering unit = $DeclUnit \times LookAhead + 1$ (see Section 5.1 for the rationale), the actual declustering units are 1, 5, 9, and ∞ data blocks, respectively. The figures show that *Decluster1* consistently delivers the lowest % late frames and latency values for this experiment, followed by *Decluster2*. *Decluster0* performs satisfactorily initially. As the number of terminals per disk increases beyond 23, however, the performance of *Decluster0* deteriorates very rapidly. In fact, at 28 terminals/disk, *Decluster0* results in more than 2.5 times as many frames being delivered late, and 1.5 times as long an average latency compared to *Decluster1*. The last data-placement policy, *Cluster*, is also unsatisfactory. These observations clearly show that the choice of data-placement policies can have a very significant impact on the number of concurrent retrieval streams that a multimedia server is capable of supporting. We shall now analyze each placement policy in turn.

Let us first examine the *Decluster0* policy. With this policy, every retrieval operation fetches just one block from a disk, then moves on to another disk in the next time slice. Load imbalance situations where one disk is substantially more heavily loaded than others are, therefore, relatively rare and short lived. However, the balanced load is achieved at the expense of requiring each disk to incur a seek for every block request. This high-seek overhead quickly overwhelms the disks, causing *Decluster0* to produce the worst % late frames and latency across the entire range of *TerminalsPerDisk* settings.

In contrast to *Decluster0*, which spreads out the data blocks of an object over many disks, *Cluster* keeps all of the blocks within the same disk. This restricts each retrieval stream to being serviced by one disk throughout its lifetime. Such a situation causes load imbalance situations to arise readily. Moreover, the duration of these load imbalances are proportional to the length of the multimedia

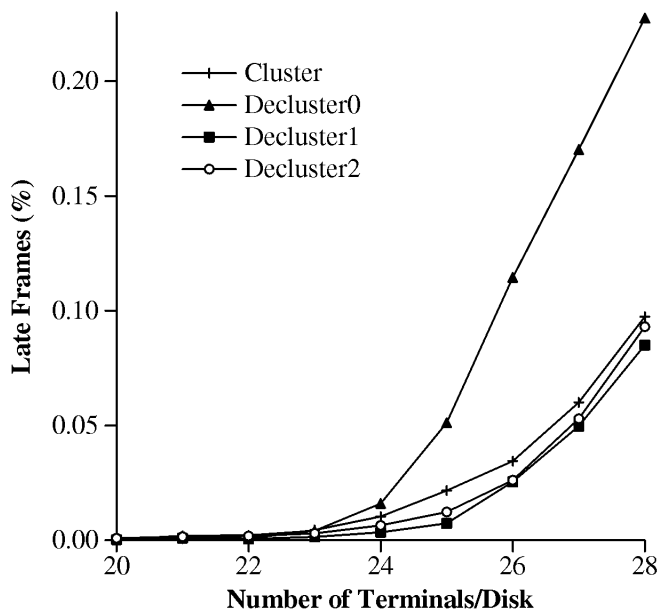


Fig. 12. Jitter ratio.

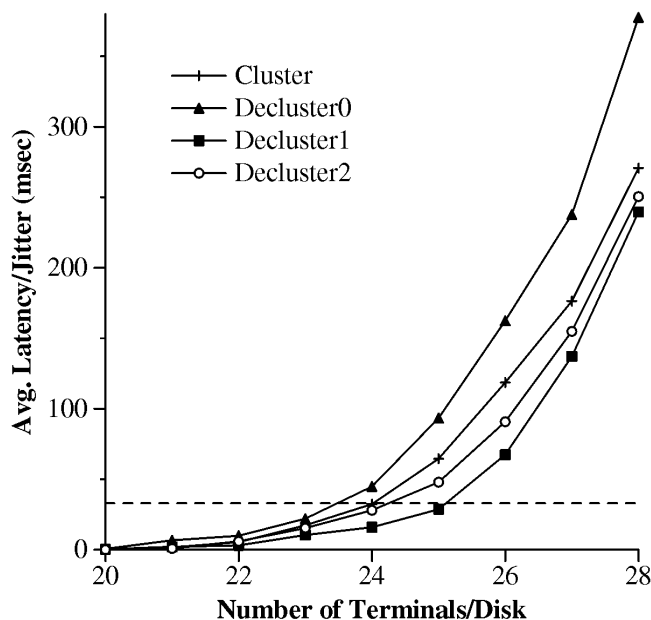


Fig. 13. Latency.

objects that are being retrieved. For these reasons, the disk utilizations that *Cluster* produces are the most skewed. However, *Cluster* does have the virtue of enabling the disks to exploit the physical contiguity between successive data blocks of the same retrieval stream to service a number of its requests at the same time. This eliminates disk seeks for some requests. As a result, *Cluster* is able to keep its % late frames and latency below those of *Decluster0*.

Turning our attention to *Decluster1*, we first note that it places $LookAhead + 1$ consecutive blocks of an object on each disk. This allows the disks to fetch all of the blocks in every look-ahead retrieval cycle very efficiently. Moreover, each

request stream moves on to another disk after one retrieval cycle, so the stream does not remain stationary at a disk for any extended period of time. As a result, *Decluster1* is able to outperform *Cluster*, especially at light to moderate loads. Even at heavy loads, *Decluster1* is no worse than *Cluster*.

Finally, we examine the behavior of the *Decluster2* policy. Compared to *Decluster1*, this policy should offer better retrieval efficiency at the individual disks as they have more leeway to combine requests from the same retrieval stream to reduce disk head movements. However, *Decluster2* also allows retrieval streams to remain longer at each disk, thus prolonging any load imbalances. The net effect is that *Decluster2* performs worse than *Decluster1*. Further experiments show that performance continues to deteriorate toward that of *Cluster* as we increase the declustering unit. We, therefore, do not present those results here.

To summarize the results of this experiment, we can derive the following conclusions about data placement across multiple disks: On one hand, clustering all the data blocks of each object on one disk leads to very efficient utilization of individual disks, but subjects the system to prolonged load imbalances. On the other hand, *Decluster0* forces retrieval streams to move to different disks after each time slice, resulting in very even load distributions. Unfortunately, the small decluster unit of *Decluster0* also degrades retrieval efficiency, causing *Decluster0* to perform much worse than *Cluster*. This finding agrees with Ghandeharizadeh and Ramos' conclusion in [16] that declustering can harm system performance more than clustering. However, our results also demonstrate that, with appropriately large declustering units, declustering *can* outperform clustering. In particular, *Decluster1* strikes a good compromise between efficient disk utilization and balanced load distribution, enabling a server to support the highest number of retrieval streams among the clustering and declustering policies that we tested.

7.6 Two Disks

Our last experiment is designed to study how well the number of supportable streams scales up with multiple disks. We will compare the system performance for 1 disk, as well as the performance for two disks with and without object replication. The *DeclUnit* parameter of the interdisk data-placement algorithm is set to 1, as *Decluster1* has been shown to produce the best performance. The rest of the experiment parameters remain at their settings in the last experiment. The results are plotted in Figs. 14 and 15.

Figs. 14 and 15 indicate that, up to 25 terminals/disk, declustering data over the disks alone is good enough to ensure that the server performs close to the single-disk configuration. The service degradation for the 2-disk configuration should not be noticeable, as the average latency is less than the duration of a frame (indicated by the dotted line in Fig. 15). However, beyond 25 terminals/disk, the % late frames and latency for two disks deteriorate significantly more rapidly than those for a single disk. This is because the disks become overloaded at more than 25 terminals/disk, so the effect of the reduced retrieval efficiency due to declustering, no matter how slight, tends to get magnified.

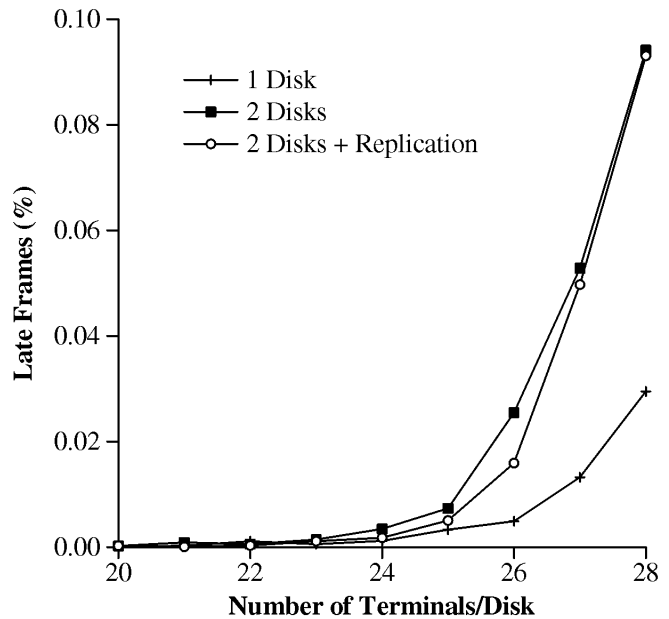


Fig. 14. Jitter ratio.

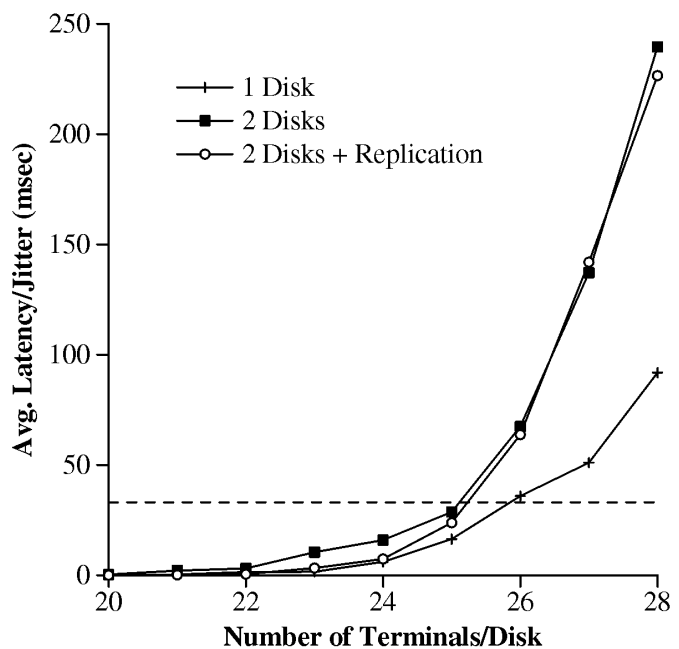


Fig. 15. Latency.

Turning our attention to the curves representing declustering with object replication, we note that the data-retrieval algorithm is very effective in exploiting object replicas to balance the disk load distribution. In fact, the % late frames and latency produced here are nearly the same as those for the single-disk configuration up to around 25 terminals/disk. Beyond this point, the influence of disk access efficiency again dominates over that of load balancing, so replication does not help to improve performance.

In summary, this experiment suggests that the coarse-grain declustering and object replication mechanisms of

TABLE 4
PHYSICAL RESOURCE MODEL

Parameter	Description	Default
<i>NumBoards</i>	Number of system boards	10
<i>PageSize</i>	Number of bytes per page	8 Kbytes
<i>XDBusSpeed</i>	Bandwidth of XDBus	625 Mbytes/sec
<i>SBusSpeed</i>	Bandwidth of SBus	66 Mbytes/sec
<i>CpuPerBoard</i>	Number of CPUs per system board	2
<i>CpuSpeed</i>	MIPS rating of the CPU	135.5 MIPS
<i>MemoryPerBoard</i>	Amount of memory on each system board	64 Mbytes
<i>NumSBusSlots</i>	Number of SBus slots per system board	4
<i>AtmPerBoard</i>	Number of ATM adaptors per system board	1
<i>AtmSpeed</i>	Bandwidth of ATM adaptor	155 Mbps
<i>ScsiPerBoard</i>	Number of SCSI adaptors per system board	3
<i>ScsiSpeed</i>	Bandwidth of Fast SCSI 2 adaptor	10 Mbytes/sec
<i>DiskPerScsi</i>	Number of disks per Fast SCSI 2 adaptor	1
<i>RotationTime</i>	Time for one disk rotation	11.1 msec
<i>TransferTime</i>	Rate of data transfer from disk	6.8 Mbytes/sec
<i>DiskSize</i>	Storage capacity per disk	9 GBytes
<i>NumCylinders</i>	Number of cylinders per disk	4,920
<i>DiskCache</i>	Size of disk cache	1 Mbyte

our proposed data-placement and retrieval algorithms are very effective. In the next section, we shall study how the performance of the algorithms scale up to a larger number of disks.

8 SCALABILITY: A SIMULATION STUDY

The last section shows that the server achieves good efficiency, and that the number of supportable streams scales linearly up to two disks. Unfortunately, due to resource availability we are not able to physically experiment with a larger number of disks. We shall, therefore, employ simulation to study the scalability of our server.

8.1 Simulation Model

In writing the simulator, we have taken great care to model the server implementation as accurately as possible, so that the simulator can be verified against the server. The simulator closely follows the system architecture depicted in Fig. 1, with operations accurately modeled down to the page level. The database and workload are exactly the same as described in Section 7.1. However, we cannot use a Sparc 20 workstation for the scaled up experiments as we did for the previous experiments. This is because the Sparc 20 has only four SBus slots, which means it can accommodate a maximum of three Fast SCSI-2 adaptors¹ after allocating one slot for the ATM adaptor. Since each Fast SCSI-2 adaptor has enough bandwidth for only one disk,² we will not be able to scale beyond three disks. For this

1. We could have increased the number of disk per SBus slot by using Fast Wide SCSI-2 adaptors, but we decided against that as it necessitates changes to the disks, the most critical resources in determining the server's performance.

2. While each Fast SCSI-2 adaptor can handle up to seven disks, its 10 Mbytes/sec bandwidth ceiling would prevent the number of supportable streams from scaling up. In fact, experiments with our implemented server showed a performance dip as soon as we attached more than one disk.

reason, the physical resources of the simulator are modeled after a SPARCcenter 2000E.

The parameters of the simulation model for the SPARCcenter 2000E are summarized in Table 4. The SPARCcenter 2000E holds up to *NumBoards* system boards, which are interfaced by 2 XDBus system buses with an aggregate bandwidth of *XDBusSpeed*. Each system board provides *CpuPerBoard* processors with a speed of *CpuSpeed*, *MemoryPerBoard* memory, and *NumSBusSlots* SBus slots. The SBus slots are occupied by *AtmPerBoard* ATM adaptors and *ScsiPerBoard* Fast SCSI 2 adaptors. Finally, we attach *DiskPerScsi* disks to each SCSI adaptor.

8.2 Model Verification

To verify the accuracy of the simulation model, we carry out an experiment using the same workload as the experiment in Section 7.6. The parameters for the physical resource model are set at their default values, except for the number of hard disks which we vary from one to two. The simulator's predictions for the "object replication" case are shown in Figs. 16 and 17. For comparison purposes, the figures also include the curves obtained from the server implementation.

Figs. 16 and 17 show that the simulator successfully captures the *qualitative* behavior of the server. However, the quantitative results are not entirely accurate. Specifically, the performance of the simulator degrades later, exceeding our threshold of 33-msec average latency per jitter only after 28 terminals/disk, whereas the server crossed the threshold after 25 terminals/disk. We suspect that the simulator's more optimistic predictions are primarily due to its not capturing delays introduced by the mechanical components of the disks, e.g., thermal recalibration. Another factor could be that the SPARCcenter 2000E modeled by the simulator has more CPU power than the SPARC 20 workstation that was used to run the server, though we expect the effect of this factor to be relatively insignificant

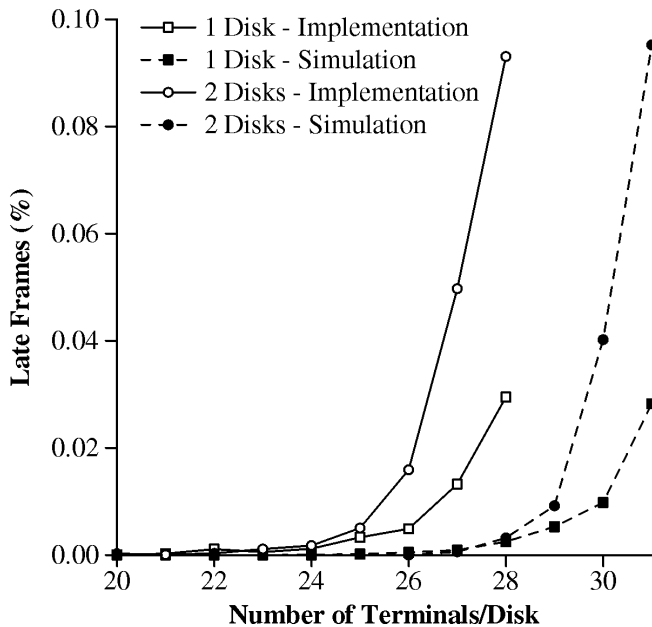


Fig. 16. Jitter ratio.

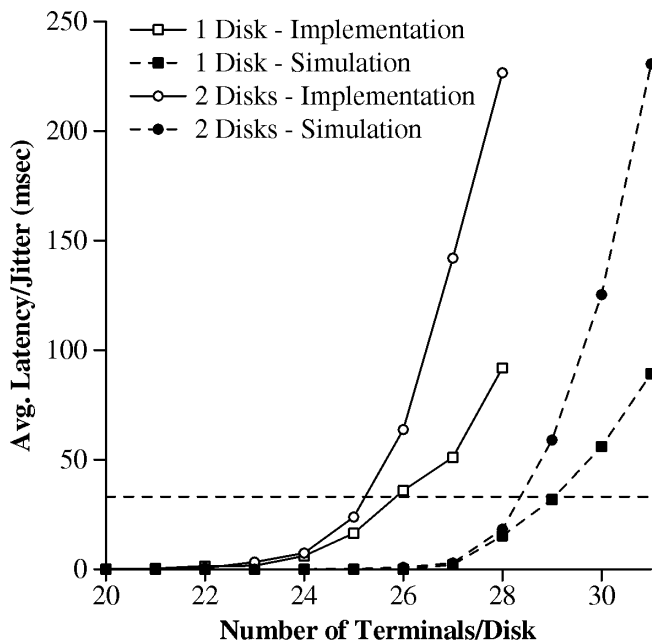


Fig. 17. Latency.

as the CPU utilizations on the SPARC 20 workstation were well below 20 percent.

Besides this experiment, we also repeated all of the experiments presented previously. In addition, we ran several more experiments with the server and the simulator. In one experiment, we varied the *ThinkTime* parameter and found that both the observed and predicted performance degraded later as *ThinkTime* increased, though there were no qualitative changes. In another experiment, we modified the setting of *AccessSkew%* and *HotSize%*. There, we observed that raising *AccessSkew%* and/or decreasing *HotSize%*

increased the number of supportable streams as disk retrievals became more localized, while lowering *AccessSkew%* and/or increasing *HotSize%* led to fewer supportable streams. In all of these experiments, we observed that the simulator consistently captured the qualitative behavior of the server. We will, therefore, use the simulator to study the scalability of the server.

8.3 Experiment

For the scale-up experiment, we retain all the workload and physical resource settings of the previous experiment, except for the total number of disks which we vary from 1 to 30. As for the server experiments, we ran each simulation experiment long enough to allow for 2,000 object retrievals. We also verified that the size of the 90 percent confidence intervals for the jitter ratio, computed using the batch means approach [2], were within a few percent of the mean in almost all cases.

The simulation results, presented in Figs. 18 and 19, show that the number of supportable terminals scales almost linearly with the number of disks, up to 28 terminals/disk. Beyond that, the disks get overloaded, and the effect of reduced retrieval efficiency due to declustering overwhelms that of load balancing. This causes performance to degrade rapidly as explained in Section 7.6. Due to quality of service considerations, in practice a multimedia server is not likely to be configured to operate in the overload region. The algorithms presented in this paper should, therefore, enable our server to scale up satisfactorily.

9 CONCLUSIONS

In this paper, we focus on the problem of implementing a multimedia server that supports continuous media (CM) data including video and audio. Our primary contribution in this work is a set of intra- and interdisk data-placement and retrieval algorithms that are designed to exploit the full capacity of the disks in a server, in order to maximize the number of retrieval streams that can be supported. The data-placement algorithm declusters every object over all of the disks, using a time-based declustering unit, with the aim of balancing the disk load distribution. Within each disk, objects that are accessed more frequently are allotted pages in the middle cylinders to minimize seek delays, whereas objects that are less popular reside in the outer and inner cylinders. As for runtime retrieval, the quintessence of the algorithm is to give each disk advance notification of the blocks that have to be fetched in the impending time periods, so that the disk can optimize its service schedule accordingly. Moreover, in processing a block request for a replicated object, the server will dynamically channel the retrieval operation to the most lightly loaded disk that holds a copy of the required block.

We have implemented a multimedia server based on the above algorithms. Performance tests on our implementation confirm that these algorithms lead to very efficient disk utilization, thus enabling the multimedia server to support a large number of concurrent users. We managed to achieve near jitter-free retrieval for up to 25 concurrent MPEG-1 streams, each averaging 1.2 Mbits/sec, out of

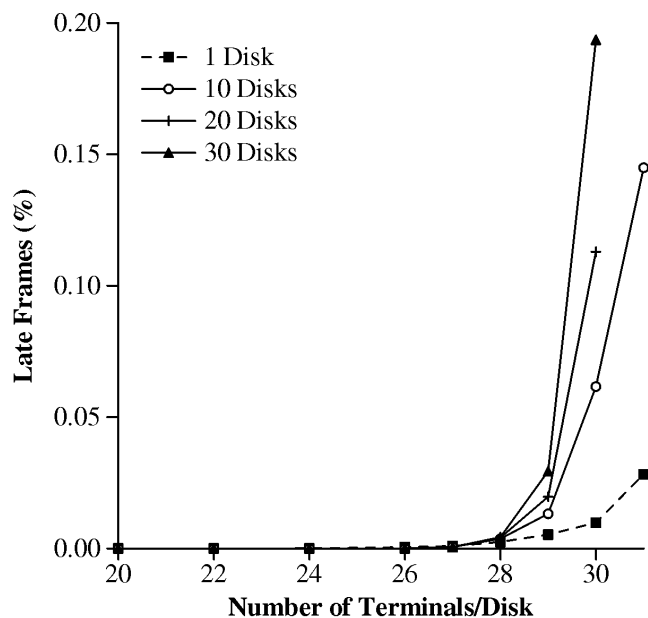


Fig. 18. Jitter ratio.

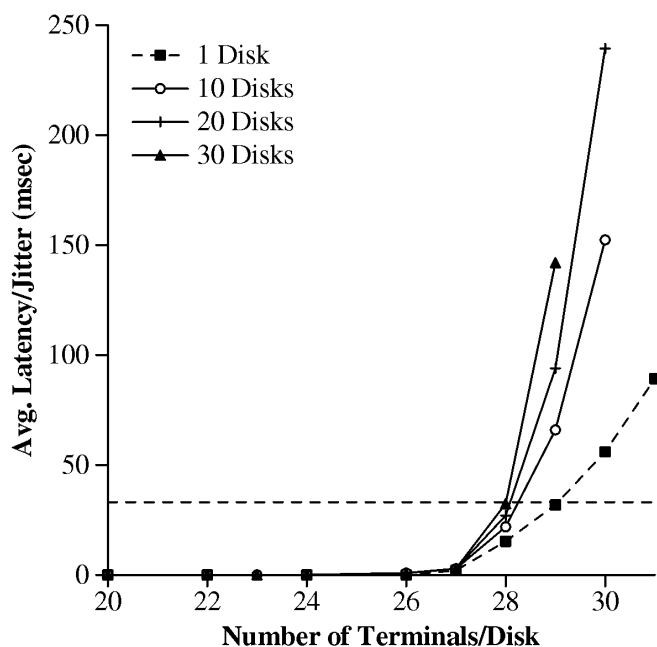


Fig. 19. Latency.

a single Seagate Elite 9 drive that is capable of reaching only a maximum *sequential* retrieval speed of around 6 Mbytes/sec. Moreover, experiments suggest that the total retrieval capacity of the server scales up almost linearly with the number of disks. Therefore, the server has a capacity equal to 30 Mb/s times the number of data disks, and new requests are admitted on a first-come-first-serve basis up to this capacity.

We are extending this work in several ways. Currently, we are in the process of adapting our implementation to a distributed system architecture. We are also partnering a

number of companies to deploy our servers to support multimedia-on-demand applications. Finally, we hope to extend the server into a full-fledged multimedia DBMS.

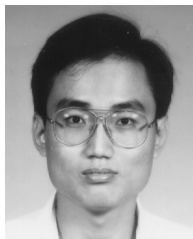
ACKNOWLEDGMENTS

The authors thank Desai Narasimhalu for his support and advice. We also thank Pingli Pang for contributing to the system implementation.

REFERENCES

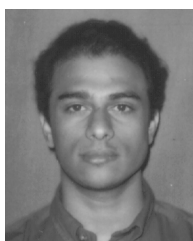
- [1] J.A. Adam, "Interactive Multimedia: Applications, Implications," *IEEE Spectrum*, vol. 30, no. 3, pp. 24-31, Mar. 1993.
- [2] J. Banks and J.S. Carson II, *Discrete-Event System Simulation*. Prentice Hall, 1984.
- [3] S.A. Barnett and G.J. Anido, "An Efficient Non-Hierarchical Storage System for Video Servers," *Proc. Multimedia Japan Conf.*, pp. 376-385, Mar. 1996.
- [4] P. Bocheck, H. Meadows, and S.-F. Chang, "Disk Partitioning Technique for Reducing Multimedia Access Delay," *Proc. IASTED/ISMM Int'l Conf. Distributed Multimedia Systems and Applications*, pp. 27-30, Aug. 1994.
- [5] M.J. Carey, R. Jauhari, and M. Livny, "Priority in DBMS Resource Scheduling," *Proc. 15th Int'l Conf. Very Large Data Bases*, pp. 397-410, Aug. 1989.
- [6] C.K. Chang, C.C. Shih, T. Nguyen, and P. Mongkolwat, "A Popularity-Based Data Allocation Scheme for a Cluster-Based VOD Server," *Proc. COMPSAC*, pp. 62-67, Aug. 1996.
- [7] E. Chang, "Storage and Retrieval of Compressed Video," PhD thesis, Univ. of California at Berkeley, 1996.
- [8] S. Chaudhuri, S. Ghandeharizadeh, and C. Shahabi, "Avoiding Retrieval Contention for Composite Multimedia Objects," *Proc. 21th Int'l Conf. Very Large Data Bases*, pp. 287-298, Sept. 1995.
- [9] M.S. Chen, D.D. Kandlur, and P.S. Yu, "Storage and Retrieval Methods to Support Fully Interactive Playback in a Disk-Array-Based Video Server," *ACM Multimedia Systems J.*, vol. 3, no. 3, pp. 126-135, July 1995.
- [10] A. Dan, M. Kienzle, and D. Sitaram, "A Dynamic Policy of Segment Replication for Load-Balancing in Video-On-Demand Servers," *ACM Multimedia Systems J.*, vol. 3, no. 3, pp. 93-103, July 1995.
- [11] A. Dan and D. Sitaram, "An Online Video Placement Policy Based on Bandwidth to Space Ratio (BSR)," *Proc. ACM SIGMOD Conf.*, pp. 376-385, May 1995.
- [12] C.S. Freedman and D.J. DeWitt, "The SPIFFI Scalable Video-On-Demand System," *Proc. ACM SIGMOD Conf.*, pp. 352-363, May 1995.
- [13] D. Gall, "MPEG: A Video Compression Standard for Multimedia Applications," *Comm. ACM*, vol. 34, no. 4, pp. 46-58, Apr. 1991.
- [14] G.R. Ganger, B.L. Worthington, R.Y. Hou, and Y.N. Patt, "Disk Subsystem Load Balancing: Disk Striping vs. Conventional Data Placement," *Proc. 26th Hawaii Int'l Conf. System Sciences*, pp. 40-49, Jan. 1993.
- [15] M.N. Garofalakis, B. Ozden, and A. Silberschatz, "Resource Scheduling in Enhanced Pay-Per-View Continuous Media Databases," *Proc. VLDB Conf.*, pp. 516-525, Aug. 1997.
- [16] S. Ghandeharizadeh and L. Ramos, "Continuous Retrieval of Multimedia Data Using Parallelism," *IEEE Trans. Knowledge and Data Eng.*, vol. 5, no. 4, pp. 658-669, Aug. 1993.
- [17] R.L. Haskin and F.L. Stein, "A System for the Delivery of Interactive Television Programming," *Proc. IEEE COMPCON*, pp. 209-215, Mar. 1995.
- [18] R.L. Haskin, personal communication, Mar. 1995.
- [19] A. Heybey, M. Sullivan, and P. England, "Calliope: A Distributed, Scalable Multimedia Server," *Proc. Usenix Technical Conf.*, pp. 75-86, Jan. 1996.
- [20] H. Hsiao and D. DeWitt, "Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines," *Proc. Sixth Int'l Conf. Data Eng.*, pp. 456-465, Feb. 1990.
- [21] D.R. Kanchammana-Hosekote and J. Srivastava, "I/O Scheduling for Digital Continuous Media," *ACM Multimedia Systems J.*, vol. 5, no. 4, pp. 213-237, July 1997.

- [22] K. Keeton and R.H. Katz, "Evaluating Video Layout Strategies for a High-Performance Storage Server," *ACM Multimedia Systems J.*, vol. 3, no. 2, pp. 43-52, May 1995.
- [23] T.G. Kwon, Y. Choi, and S. Lee, "Disk Placement for Arbitrary-Rate Playback in an Interactive Video Server," *ACM Multimedia Systems J.*, vol. 5, no. 4, pp. 271-281, July 1997.
- [24] T.D.C. Little and D. Venkatesh, "Popularity-Based Assignment of Movies to Storage Devices in a Video-On-Demand System," *Proc. Fourth Int'l Workshop Network and Operating System Support for Digital Audio and Video*, pp. 213-224, Nov. 1993.
- [25] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *J. ACM*, vol. 20, no. 1, pp. 46-61, Jan. 1973.
- [26] M. Livny, S. Khoshafian, and H. Boral, "Multi-Disk Management Algorithms," *Proc. ACM SIGMetrics Conf.*, pp. 69-77, May 1987.
- [27] M. Mehta and D.J. DeWitt, "Data Placement in Shared-Nothing Parallel Database Systems," technical report, Computer Sciences Dept., Univ. of Wisconsin—Madison, 1994.
- [28] G. Miller, G. Baber, and M. Gilliland, "News On Demand for Multimedia Networks," *Proc. ACM Multimedia Conf.*, pp. 383-392, Aug. 1993.
- [29] A.N. Mourad, "Issues in the Design of a Storage Server for Video-On-Demand," *ACM Multimedia Systems J.*, vol. 4, no. 2, pp. 70-86, Apr. 1996.
- [30] R.T. Ng and J. Yang, "Maximizing Buffer and Disk Utilization for News On-Demand," *Proc. 20th Int'l Conf. Very Large Data Bases*, pp. 451-462, Sept. 1994.
- [31] Y.-J. Oyang, M.-H. Lee, C.-H. Wen, and C.-Y. Cheng, "Design of Multimedia Storage Systems for On-Demand Playback," *Proc. 11th Int'l Conf. Data Eng.*, pp. 457-465, Mar. 1995.
- [32] B. Ozden, R. Rastogi, and A. Silberschatz, "On the Design of a Low-Cost Video-On-Demand Storage System," *ACM Multimedia Systems J.*, vol. 4, no. 1, pp. 40-54, Feb. 1996.
- [33] H. Pang, "Data Placement and Retrieval in a Disk-Based Multimedia Storage System," *ISS Technical Report*, 1995.
- [34] E. Rahm, "Parallel Query Processing in Shared-Disk Database System," *SIGMOD Record*, vol. 22, no. 4, pp. 32-37, Dec. 1993.
- [35] P.V. Rangan, H.M. Vin, and S. Ramanathan, "Designing an On-Demand Multimedia Service," *IEEE Comm.*, vol. 30, no. 7, pp. 56-64, July 1992.
- [36] A.L.N. Reddy and J.C. Wyllie, "I/O Issues in a Multimedia System," *Computer*, vol. 27, no. 3, pp. 69-74, Mar. 1994.
- [37] A.L.N. Reddy, "Scheduling and Data Distribution in a Multiprocessor Video Server," *Proc. Int'l Conf. Multimedia Computing and Systems*, pp. 256-263, May 1995.
- [38] C. Ruemmler and J. Wilkes, "An Introduction to Disk Drive Modeling," *Computer*, vol. 27, no. 3, pp. 17-28, Mar. 1994.
- [39] A. Srivastava, A. Kumar, and A. Singru, "Design and Analysis of a Video-On-Demand Server," *ACM Multimedia Systems J.*, vol. 5, no. 4, pp. 238-254, July 1997.
- [40] A.S. Tanenbaum, *Modern Operating Systems*. Prentice Hall, 1992.
- [41] F.A. Tobagi, J. Pang, R. Baird, and M. Gang, "Streaming RAID—A Disk Array Management System for Video Files," *Proc. ACM Multimedia Conf.*, pp. 393-400, Aug. 1993.
- [42] H.M. Vin and P.V. Rangan, "Designing a Multi-User HDTV Storage Server," *IEEE J. Selected Areas in Comm.*, vol. 11, no. 1, pp. 153-164, Jan. 1993.
- [43] Y. Wang, J.C.L. Liu, D.H.C. Du, and J. Hsieh, "Video File Allocation Over Disk Arrays for Video-On-Demand," Computer Science Technical Report TR95-067, Univ. of Minnesota, 1995.
- [44] J.L. Wolf, P.S. Yu, and H. Shachnai, "DASD Dancing: A Disk Load Balancing Optimization Scheme for Video-On-Demand Computer Systems," *Proc. ACM SIGMetrics Conf.*, pp. 157-166, May 1995.
- [45] P. Yu, M.S. Chen, and D.D. Kandlur, "Design and Analysis of a Grouped Sweeping Scheme for Multimedia Storage Management," *Proc. Third Int'l Workshop Network and Operating System Support for Digital Audio and Video*, pp. 38-49, Nov. 1992.

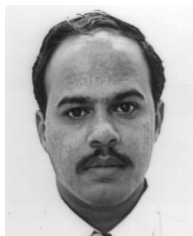


HweeHwa Pang received the BSc degree (with first-class honors) and the MS degree from the National University of Singapore in 1989 and 1991, respectively; and the PhD degree from the University of Wisconsin at Madison in 1994—all in computer science. He is now a research staff member at Kent Ridge Digital Laboratories (formerly the Institute of Systems Science) in Singapore. He heads a mobile computing project to develop software infrastructure and utilities to facilitate information access and computing from

mobile devices. His research interests include database management systems, multimedia servers, and real-time systems.



Bobby Jose received an ME degree in electrical engineering (with distinction) from the Indian Institute of Science in 1995. His thesis focused on issues in distributed collaborative systems. He joined the Kent Ridge Digital Labs in Singapore in 1995, and participated in the design and development of a distributed multimedia server and various applications. He also led a project to develop a key frame index stream-based retrieval scheme for networked video. He is currently developing the network architecture and services for the Wireless ATM Research Project at the Ubiquity Lab. He functions as technical lead for MPEG and transport-related issues, and investigation into applying open signaling concepts to location management and LANE services. His research interests include multimedia networking, wireless broadband networking, and application of open signaling concepts to build ATM-based mobile network architectures and services.



M.S. Krishnan received a bachelor of engineering degree in computer science from the R.V. College of Engineering at Bangalore University in 1990. He joined the Institute of Systems Science in Singapore in 1994, and participated in designing and developing a multimedia server. He is now with Compaq Services, Compaq Computer Asia Pte. Ltd., in Singapore.