Random RAIDs with Selective Exploitation of Redundancy for High Performance Video Servers*

Yitzhak Birk
birk@ee.technion.ac.il
+972 4 829 4637
Electrical Engineering Department
Technion – Israel Institute of Technology
Haifa 32000, ISRAEL

Abstract

This paper jointly addresses the issues of load balancing, fault tolerance, responsiveness, agility, streaming capacity and cost-effectiveness of high-performance storage servers for data-streaming applications such as video-on-demand. Striping the data of each movie across disks in a "random" order balances the load while breaking any correlation between user requests and the access pattern to disks. Parity groups are of fixed-size, comprising consecutive blocks of a movie and a derived parity block, and resulting in "random" disk-members of any given group. Consequently, the load of masking a faulty disk is shared by all disk drives, minimizing the degradation in streaming capacity. By using the redundant information to avoid accessing an overloaded disk drive, the occasional transient imbalance in disk load due to the randomization is partly prevented and, when occurring, can be circumvented. Finally and most important, making a distinction between data blocks and redundant blocks and using redundant blocks only when necessary is shown to substantially reduce required buffer sizes without giving up the benefits. The result is a simple, flexible and robust video-server architecture.

1 Introduction

A "video server" is a storage and communication system for data-streaming applications. These include the viewing of movies and video clips, and listening to audio. The primary measure of a video server's performance is the number of concurrent video streams that it can supply without glitches, subject to a sufficiently prompt response to user requests.

As depicted in Fig. 1, a video server usually stores movies on magnetic disk drives, from which they are read into RAM buffers and are subsequently "streamed" onto a distribution network. The RAM serves as an adapter between the disk drives and the network: it receives data from disk in bulk, thereby permitting efficient operation of the drives; the interleaving of data for different streams onto the distribution network is carried out with a much finer granularity, thereby avoiding undesirable burstiness over the network. The use of tertiary storage for video servers has been shown to be mostly ineffective [1].

The need for a video server to be fault-tolerant and highly available used to be questioned: the viewing of movies is not a critical application, and loss of data is not an issue since additional copies of movies are always available. Moreover, disk drives and electronic components are extremely reliable, so a mean time between failures of weeks if not months can be assumed even for fairly large servers. However, for reasons such as load-balancing that will be explained shortly, the *failure mode* of such servers is problematic: the failure of a single disk drive is likely to result in an interruption of all active streams. A "blackout" every three months is clearly much more visible than independent interruptions to individual streams at the same rate. For this reason, high availability is a must.

The design of a video server is not a matter of feasibility. Rather, the issue is cost-effectiveness. Also, despite the fact that it is often viewed primarily as a storage server, its bandwidth often affects cost at least as much as storage capacity.

In this paper, we will exploit the unique characteristics of video servers in jointly addressing cost, performance and fault-tolerance, and providing various

*This work was begun while at Hewlett Packard Company Labs in Palo Alto, and parts of it may be covered by U.S. Patents. The author is grateful to Scientific and Engineering Software, Inc., for providing the Workbench simulation package.

0-7803-3799-9/97$10.00 © 1997 IEEE.

IPR2025-01202
IBM v. Security First Innovations, LLC
IBM's Exhibit 1018

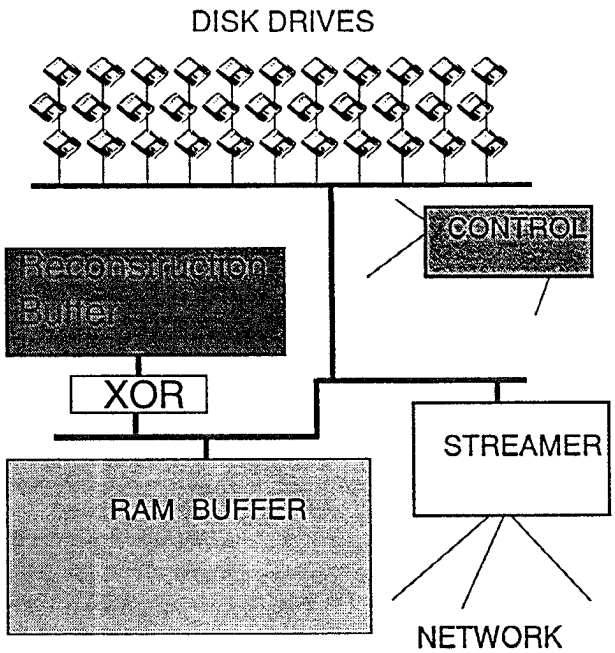

Figure 1: A typical video server. Data is stored on disk, read into RAM buffers, and then streamed onto a distribution network. Components also include intra-server communication, control and possible processing of the data for purposes such as fault-tolerance.

additional benefits. Our focus is on inter-disk data layout and scheduling. (Intra-disk issues, especially the accommodation of multi-zone recording, are addressed in [2][3][4].) We begin with a careful review of the characteristics of a video server and the applications for which it is used. The implications of those on the design are then examined, and current approaches to the design of servers are discussed in this context. We then present a new video-server architecture that jointly addresses all the problems that arise in the storage subsystem (including storage bandwidth and capacity, RAM buffers, intra-server communication overhead and fault-tolerance), and present simulation results that indicate the promise of our approach. The key idea in our approach is the use of randomization in data layout in order to solve certain problems, along with the selective exploitation of redundancy for the purpose of taming the randomization and sharply mitigating its undesirable side-effects. The result is a simple, cost-effective, agile and easily-testable server architecture.

The remainder of the paper is organized as follows. In section 2, we characterize a video server and the requirements imposed on it, and raise a number of important issues pertaining to the cost-effectiveness of a design. In section 3, we progress through a sequence of video-server designs, solving certain problems and identifying new ones as we go along. In section 4 we present our approach, and simulation results are presented in section 5 along with additional insights. Section 6 discusses our approach, and section 7 offers concluding remarks.

2 Characteristics of a video server

In most applications of storage systems, the semantics of requests are "provide all the requested data as soon as possible", and a server is measured on the time until the last byte is received. In contrast, the intent of a request from a video server is "begin providing data as soon as possible, then continue providing it at the prescribed *rate*". The primary measure of a video server's performance is the number of concurrent video streams that it can supply without glitches, subject to a sufficiently prompt response to user requests.

A video server is a "real-time" system in the sense that it must provide a given amount of data per unit time (with some tolerance in one direction, namely excess data, and none in the other) and in the sense that it must respond to user requests within a short period of time. At the "micro" level, however, it differs dramatically from a true "real-time" system since, once the viewing of a video stream has started, the sequence of requests for data is known and requests occur at times that can be determined in advance. Moreover, since all the data (movie) is available at the outset, the server may mask disk latency by prefetching data into its buffers. With the exception of response to new user requests, a video server is thus best viewed a "data pump" rather than as an interactive system.

The two primary resources of a video server are its storage capacity and its storage (communication) bandwidth. Unfortunately, the two are embodied in disk drives and are thus coupled. So, in order to claim that a server can supply a given set of video streams based on the server's aggregate storage bandwidth, one must balance the communication load among all disk drives at all times, independently of viewing choices.

In many cases, the system cost for supporting a disk drive is independent of the drive's storage capacity, and even the cost of a disk drive is the sum of a fixed component and a linear one across a large range of capacities. It is therefore desirable to use high-capacity disk drives. However, using fewer storage devices means that each one must have higher bandwidth in order to meet the aggregate bandwidth

requirement. Consequently, disk drives must be operated efficiently. This can be done by efficient seek algorithms in conjunction with early submission of requests (so as to effectively extend deadlines and permit more flexible disk-scheduling) [5] as well as by reading large chunks of data in each access.

"Video" is often associated with very high data rates. However, compressed digital video, which is being considered in this paper, requires very modest data rates, ranging from 150KB/s to some 600KB/s. The transfer rate of a modern disk drive is on the order of 5MB/s, i.e., much higher than the video rate. Throughout the paper, we will consider the ratio between a disk drive's effective transfer rate and the rate of a video stream, denoted *dvr*, and will show that it is a very important parameter.

RAM buffers in a video server are required for two purposes:

- *Starvation prevention.* Whenever there is a delay in the response of a disk drive to a read request, data is streamed from the buffer in order to mask the delay and prevent a glitch (starvation).

- *Overflow prevention.* Any data that is prefetched, e.g., to permit efficient operation of the disk drive, must be buffered. The large *dvr*, in conjunction with the desire to operate the disk drives efficiently, implies that data read from disk is held in RAM buffer for a long time (proportional to *dvr*), which increases the required amount of overflow buffer space.

Despite the fact that the two types of buffers are embodied in the same memory, it is important to observe an important difference between them: buffers for starvation prevention must be allocated in advance to every stream and filled with data, since when trouble strikes it is too late. In contrast, overflow buffers can be allocated dynamically on a need basis. We will return to this issue when simulation results are presented. For the above reasons and other ones, the required amount of RAM must not be overlooked when designing a video server.

3 Successive refinements of a basic architecture

The controllable components of server cost are RAM buffers and various overheads (storage capacity and bandwidth, for example). In this section, we discuss data layout and access scheduling in order to better understand the issues. We begin with data layout, and then move on to scheduling. We will use a sequence of design approaches in order to unveil fundamental problems. These will later be addressed in our architecture.

3.1 Data striping for load-balancing

The only way of avoiding communication hot spots is to distribute the communication load among all disk drives uniformly under all circumstances. This can only be done by splitting the data of each movie among the drives in sufficiently small pieces, as this "striping" guarantees that all drives are equally utilized at all times. Moreover, since a disk drive's transfer rate is much higher than the data rate of a single video stream, frequent seeks are unavoidable and are not caused by the striping. So, striping each movie across many or all disks is a widely accepted approach.

The amount of data read from disk in a single access should be sufficiently large so as to keep the seek overhead in check, and the striping granularity should be equal to this amount. Throughout most of the paper, we will ignore seek overhead.

3.2 Fault tolerance

A disk drive can be modeled as either providing correct data or declaring itself faulty. Consequently, parity (XOR) information can be used to correct a single disk failure. This is exploited in various ways in redundant arrays of inexpensive disks (RAID)[6] in order to provide fault tolerance. An alternative approach is duplication of data, also referred to as "mirroring". Throughout the remainder of the paper, we will use M to denote the total number of disk drives in the server, and the size of a parity group will be denoted by $k+1$. (In the case of mirroring, $k=1$.)

3.3 RAID 3: a buffer-explosion problem

In a RAID 3 organization [6], data is striped very thinly across all disk drives ($k = M-1$), and all are accessed simultaneously to service a single request. (One of the drives contains parity information which can be used to overcome a single disk failure.) The disk array thus has the appearance of a single high-capacity, high-bandwidth, highly-available disk drive.

Since, in a video server, the duration of a read is dictated by disk-efficiency considerations, accessing all disks concurrently on behalf of the same stream would result in a per-stream buffer size that is proportional to the number of disk drives. Since the server's streaming capacity is also proportional to the number of drives, total buffer size would increase quadratically with the number of drives. This approach, which has been employed in small video servers [7], is therefore indeed limited to very small servers.

3.4 Other regular layout and scheduling schemes

When viewed simplistically, the data-retrieval requirements imposed on a video server lend themselves to a round-robin layout of the data for any given movie across the disk drives, and an associated round-robin access schedule on behalf of the different video streams.

Approaches along this line include, for example, striping data across all drives but partitioning the drives into several parity groups. The access schedules to the different groups can be staggered relative to one another in order to prevent the buffer-explosion problem.

Unfortunately, and regardless of details, the placement of consecutive data chunks of any given movie in consecutive disks gives rise to several salient properties:

- There is a high correlation between the timing of user requests for streams from the same or different movies and the (temporal) load patterns on the disks. Moreover, patterns that arise tend to persist. This, combined with the limited scheduling flexibility, can create hot spots which must be masked by large buffers or restrict operation to low loads. Finally, even if one offers an algorithmic solution, it is extremely difficult to prove its potency in all cases, since it depends on user-generated scenarios.

- The round-robin data layout along with fixed-size parity groups results in a significant performance degradation upon disk failure. The group of $k+1$ drives that includes the failing drive loses $\frac{1}{k+1}$ of its bandwidth and is the weak link of a chain. Rebuilding the missing data onto a fresh replacement drive may even take up all the bandwidth.

- The apparent simplicity and efficiency of the "law and order" approach of highly regular layout and scheduling are not viable in practice: variability in disk performance, variable compression ratio, variable transfer rate (multi-zone recording) and other factors all break the regularity and mandate operation at light loads in order to provide sufficient slack. (The problem is akin to that of designing a synchronous pipeline from stages with variable execution times: the design must be based on the worst case.)

3.5 Randomized layout

Here, consecutive chunks of data for any given movie are stored on randomly-ordered disk drives. (In practice, we use some rules to govern the randomness so as to guarantee "short term" equal distribution of data across disks.) Moreover, fixed-size parity groups ($k << M$) are used to avoid the buffer-explosion problem. This randomization achieves two important goals:

1. It breaks the aforementioned correlations.

2. Any given disk drive equally participates in parity groups with all other disk drives. Consequently, when a drive fails, the load of covering for it is shared equally among all the remaining drives. Accordingly, performance degradation is minimal and remains small even during rebuild. A similar approach was proposed in a different context in [8] under the name "parity declustering". A much more restricted form of declustering was discussed in [9] in the context of fault-tolerant video servers, but ignored any variability in disk performance or video rate.

Unfortunately, randomization leaves problems such as the determination of performance by the slowest disk unsolved, and even creates a new problem, namely short-term hot spots. The former is due to the fact that, regardless of the details of the data layout, the load is distributed uniformly across all drives. The latter is due to the fact that, regardless of the exact arrival process of read-requests to the system, the process as seen by any given disk drive is similar to a Poisson process, and a disk can thus be viewed approximately as an M/D/1 queue. The "meaningfully thick" tail of the queue-length distribution of such a queue at heavy loads is long, and so is the delay that must be maskable with starvation buffers.

An alternative (or complementary) approach entails "scheduling into the future": if, when looking at the schedule, one observes that a presently-idle disk drive is soon to become overloaded, one reads from that drive in advance of the schedule in order to "smooth" the load [10]. This approach, however, cannot help when the load on the disks is unequal or their performance is not identical, since it is akin to taking a loan which must be repaid. One must also watch the impact on overflow buffer requirements very carefully.

3.6 Exploitation of redundancy for load balancing

Although redundancy was originally intended for fault-tolerance, its usefulness for performance enhancement has also been noted: in 1975, Maxemchuk proposed to partition a message into m packets, compute r redundant ones such that the original message

can be reconstructed from any sufficiently large subset of the $m+r$ packets, send those packets over different paths and use the earliest arrivers to reconstruct the message [11]. He showed that this substantially reduced delay and provided fault-tolerance. Similar observations were later made by Rabin [12] and dubbed "information dispersal algorithm". This approach, cast in terms of disk arrays, would entail submitting requests to all the disk drives holding a given parity group, and using the first ones that arrive to reconstruct the data. With parity groups of size $k+1$, this would entail a storage-bandwidth overhead of $1/k$, which is undesirable. This is alluded to by Bestavros [13] in a paper that focuses on the fault-tolerance advantages of IDA over multi-parity schemes.

Unlike in communication networks, in storage systems it is often possible to know the expected response time of different disk drives at any given time, based on the pending requests. Accordingly, one may select a subset of size k from among the $k+1$ disks holding the chunks comprising a parity group based on this information and submit requests only to those, thereby avoiding the bandwidth overhead. This approach, used here and developed independently by Berson, Muntz and Wong [14], eliminates the storage-bandwidth overhead. As shown in [14] and in a later section of this paper, this use of redundancy effectively "clips" the tail of the queue-length distribution, resulting in the ability to guarantee low response times with very high probability. This, in turn, sharply reduces the requirement for "starvation" buffers whose purpose is to mask the delays. Additional important observations are included in the next section, in which we present our new architecture.

4 Selective exploitation of redundancy

4.1 Motivation and the basic scheme

In the cited work on exploitation of redundancy for load balancing, no distinction was made between "data" chunks and "redundant" chunks. Thus, a simple policy for exploitation of redundancy for the purpose of delay (and buffering) reduction in a video server might entail, for every parity group, reading from the k (out of $k+1$) disk drives with the shortest queues.

In applications such as on-line transaction processing, wherein one typically accesses a single block that resides on a single disk drive, avoiding the reading of a particular data block would require the reading of the remainder of its parity group, namely k blocks. The resulting storage-bandwidth overhead is obviously unacceptable, so one would always access the data block unless the disk holding it is faulty.

At the other extreme, in applications that entail reading large amounts of contiguous data as fast as possible, the entire parity group is required as soon as possible, so (from a storage-bandwidth perspective) it doesn't mater which k of the $k+1$ drives are accessed.

A video server presents yet another situation: the chunks comprising a parity group are all required, since they are consecutive chunks of the same movie. However, they are not all required immediately: if only data chunks are read, they can be read one at a time; in contrast, if parity is read instead of one of them, k chunks must be read before the one that wasn't read can be reconstructed. Those that are read prematurely can be thrown away and read again when needed, but this would double the required storage bandwidth, which is unacceptable. The alternative is to buffer them in "overflow" RAM until needed for streaming.

The above observations have led us to realize the importance of a distinction between the mere inclusion of redundancy and its actual exploitation: storing a parity chunk for every k data chunks is the inclusion, whereas reading it instead of a data chunk constitutes exploitation. Inclusion is a largely "static" decision, whereas exploitation can be decided dynamically. With a parity group of size $k+1$, the required buffer size per stream is smaller by approximately a factor of $k/2$ if data chunks are read one at a time.

When no distinction is made between data chunks and redundant ones, the redundant chunk is exploited with probability $k/(k+1)$, so buffer requirements would be almost as if k chunks must be read concurrently.

Based on the above, one might jump to the conclusion that the redundant chunks should not be used unless absolutely necessary to prevent buffer starvation for the stream. However, the issue is more subtle: in addition to solving an immediate problem for the stream in question, refraining from accessing an overloaded disk drive also helps balance queue lengths, and thus has both a remedial and a preventive effect. Exploiting the redundancy only when absolutely necessary would thus increase the frequency of "emergencies". It would, in fact, increase the frequency of emergencies that cannot be treated in full (more than one overloaded disk among those holding a parity group).

Our approach is therefore to strike a balance between the desire for greedy, short-term minimization of required buffer space and the need for a preventive load-balancing action. For example, we exploit the redundant chunk if and only if the queue length to one of the disks holding a data chunk is longer than a cer-

tain value and the queue to the disk holding the parity chunk is sufficiently shorter. We refer to our approach as *selective exploitation of redundancy*, and apply it in conjunction with randomized layout.

4.2 Fine tuning

Fine tuning is possible in numerous ways, some of which we explain below along with the rationale.

Choice of k. The straightforward trade-off is due to the fact that smaller k increases the flexibility of avoiding congested disk drives, but increases storage overhead. However, as pointed out in [14], as k approaches the number of disk drives, and especially when $k+1$ equals the number of drives, load-balancing is guaranteed at all times. In the context of video servers, there are additional considerations that favor small k:

- The amount of data that must be buffered when reconstruction is employed is proportional to k. Taken to the extreme ($k = M - 1$), this becomes the buffer-explosion problem.
- In deciding whether to exploit the redundancy, we must consider the next k chunks of the movie. This, in turn, corresponds to a time window (into the future) whose size is proportional to k (the size of a data chunk is dictated by disk-efficiency considerations). As the window size increases, the snapshot of queue lengths becomes less and less relevant since many things can change by the time the data chunk of interest will have to be read. (Note that priorities are based on deadlines rather than FCFS.)
- The performance reduction during rebuilding increases as k increases, since more blocks must be read per single-block reconstruction.

In view of the above and due to the fact that the incremental storage-overhead savings with an increase in k diminish rather quickly, values of k between 3 and 5 appear the most sensible.

Criteria for redundancy-exploitation. Frequent exploitation of redundancy increases the required amount of "overflow" buffer space, since more chunks are read prematurely and must be buffered until streaming time. It also increases the memory-bandwidth overhead and the computation overhead. However, frequent (more aggressive) exploitation of redundancy improves load balancing, thereby helping clip the tails of the queue-length distributions. This, in turn, reduces the delay that must be maskable by buffers, thereby reducing the required amount of per-stream "starvation" buffers. The actual choice of parameters is best left to an implementation. Nonetheless, following is a brief description of the policies used in our simulations.

- The choice whether to exploit redundancy for a given sequence of k data chunks is based on the number of requests in the queue of the drive holding the first (earliest) data chunk and on that in the queue of the drive holding the parity chunk. This reflects the fact that the "normal" reading time of that data chunk is the soonest, and its queue length is the most meaningful. For redundancy to be exploited, we require that the "data" queue exceed a certain length, and that the "parity" queue be shorter by at least a certain amount. The choice of numbers is a simulation parameter; we used 8 and 4, respectively.
- Having decided to exploit parity, the $k-1$ data chunks that are read and the parity chunk are all read with the same urgency. Consequently, the original reason for giving preference to the earliest chunk is largely gone. So, despite the fact that the decision whether or not to reconstruct one of the data chunks was made based on the earliest chunk, the data chunk that is reconstructed is the one whose drive is the most congested. It should be noted that, unless there is an immediate problem for the earliest chunk, the choice of a chunk for reconstruction should be biased against this chunk, since reconstruction can only be completed when all other chunks have arrived (maximum of several i.i.d. random variables), and the earliest chunk is the most vulnerable to a missed deadline.
- Basing our decision whether or not to exploit redundancy solely on queue lengths appears strange. For example, if the queue to the disk holding the first (earliest streaming) chunk is long, this doesn't necessarily imply a long delay for that chunk, since its priority may be high. However, whenever a queue is of a certain length, at least one of the enqueued requests will have to wait for a time corresponding to the length of that queue. Recalling that the role of load-balancing as a preventive measure is as important to our scheme as its role as a quick fix, and that with proper tuning our exploitation of redundancy is normally done as a preventive measure, explains our decision.

Submission of requests. Having made the decision which of $k+1$ chunks (k data and one parity)

to read, requests must be submitted to the individual disk queues. This entails two decisions: time of submission and priority. We next discuss those.

- A chunk's priority is determined by its streaming time. If needed for reconstruction, it is determined based on the earlier of its streaming time and that of the chunk for whose reconstruction it is required. (In our simulation, we always used the streaming time of the earliest chunk whenever redundancy was exploited. This was done in part for simplicity and in part in order to counteract the fact that reconstruction can only be completed when all chunks have been read.)

- Our decision whether or not to exploit redundancy is based only on queue lengths, regardless of priority. Accordingly, the mere enqueuing of a request in a drive's queue may discourage other arrivals to that queue, even if their priority is higher than those of already-enqueued requests. Whenever reconstruction is not used, we therefore intentionally delay the submission of requests for "late" chunks until a fixed time prior to their streaming time.

Possible extensions and refinements. One possible extension is to simulate a future time window (excluding new user requests) and use that in refining the policies. This would be an extension of the approach suggested in [10] to our case. Numerous others are possible as well, but we prefer to refrain from those if a simple approach works sufficiently well and leaves little room for improvement.

5 Simulation results

In this section, we present simulation results. The intention is not an exhaustive search through the design space. Rather, it is a demonstration of the merits of our approach even with limited optimizations. Also, in order to focus on first-order effects and since randomization tends to mask details, we kept things simple whenever possible.

We simulated a 30-disk system. A sequence of random permutations was used as the disk-order in laying out the chunks of any given movie. (Whenever $k+1$ divides M, this guarantees that the same disk is not used twice by the same parity group. In other cases, alternate permutations are picked randomly, and a constrained random permutation is placed between successive independent ones. This guarantees that a disk is never used twice by any given parity group while preventing sequences of dependencies from forming.) Other approaches may be equally viable. Disk service time was assumed fixed and served as our unit of time. Seek and rotational latency were ignored (incorporated into the effective disk transfer rate). k-chunk requests for the different streams arrive in round-robin order, and the inter-arrival time is distributed uniformly between 0.5 and 1.5 times the mean value. Most of the studies were carried out with $k = 5$ and $dvr = 10$, but $k = 1, 4$ and $dvr = 5$ were used for comparison.

The system was simulated with no disk failures under a load of 0.95, and with a single disk failure under a load of 0.9 (this is actually a load of 0.93 because there are only 29 operational disk drives). Also, in both cases we introduced a small percentage ($\frac{2}{k}\%$) of "urgent" requests, corresponding to the need to respond promptly to viewer requests.

Requests for the first chunk in a group of k were submitted to the disk queues 20 time units prior to the streaming time; subsequent chunks were delayed by the inter-streaming times (dvr time units between consecutive chunks).

The figures below focus on the probability that the disk response time exceeds various values. This is a good indication of the ability to "clip" the tails of the queue-length distribution. However, the buffer sizes and response time to urgent chunks (representing viewer requests) are the true measures for comparison. We will discuss those as well.

In the figures, NER, FER and SER correspond to no-, full- and selective-exploitation of redundancy, respectively.

Fig. 2 depicts the probability that disk response time exceeds various values for the case of no faulty drives and no urgent requests. The load is 0.95, which is very high. The dramatic effect of exploitation of redundancy is clearly visible. (Note the logarithmic scale.) For example, the response time that is exceeded with probability of at most 0.0001 (corresponding to once per 100-minute movie) is reduced by a factor of two when redundancy is exploited with $k = 5$ and by another factor of two when mirroring is employed.

The difference between full exploitation of redundancy (picking the parity chunk whenever the queue to its disk is shorter than one of the queues to the data chunks) and selective exploitation is small. However, there is a big difference in the overhead due to exploitation of redundancy, as will be seen later.

Fig. 3 depicts similar plots for the case of a faulty drive and/or a small percentage of urgent chunks. The top curve depicts the case of a faulty drive (R=0.90)

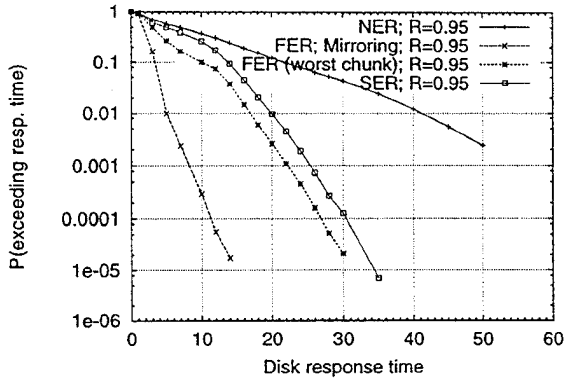

Figure 2: Probability of exceeding a specified disk response time vs. that response time: no exploitation of redundancy, selective exploitation, full exploitation and mirroring. No disk failures. $k = 5$; $dvr = 10$.

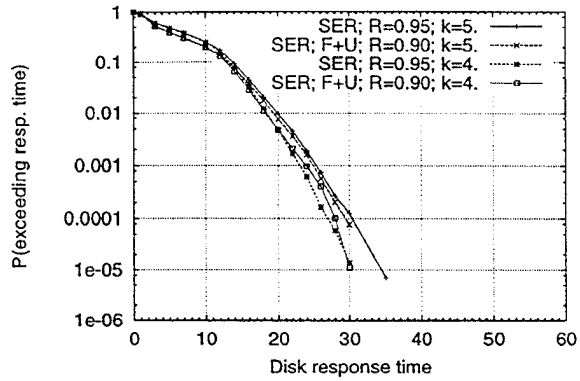

Figure 4: Probability of exceeding a specified response time vs. that response time: selective exploitation of redundancy. $k = 4, 5$; with and without faulty disk and urgent requests.

and exploitation of redundancy only when the faulty drive contains a requested data chunk. The cluster of curves includes the cases of selective exploitation for all the combinations of urgent chunks and/or a faulty drive, as well as the cases of no faulty drives (R=0.95). The leftmost plot depicts the response time to urgent requests. Again, the benefits of exploitation of redundancy are very clear, and the scheme works well even with a faulty drive at a very high load of 0.93 (on the remaining disks).

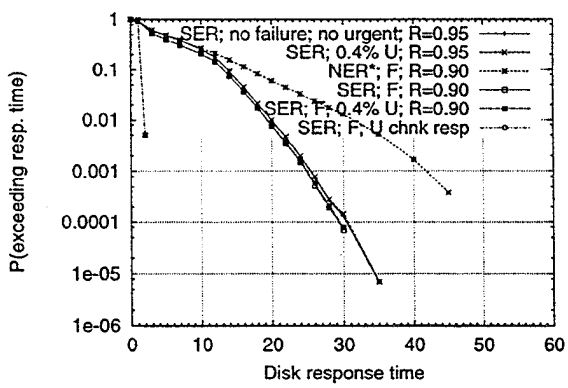

Figure 3: Probability of exceeding a specified response time vs. that response time: exploitation of redundancy only for fault tolerance; selective exploitation. With and without a faulty disk and urgent requests. Prompt response to urgent chunks is shown on the left. $k = 5$; $dvr = 10$.

Fig. 4 depicts similar plots for $k = 5$ and $k = 4$, focusing on selective exploitation. A small advantage for $k = 4$ is visible.

The simulation also tracked the total buffer occupancy and, for each chunk, recorded the difference between its streaming time and the time at which it emerged from the disks into the buffer. Dividing the buffer occupancy by the number of streams yields the required per-stream (amortized) overflow buffer size. (The number of streams equals the number of drives times dvr.) If chunks arrive post-deadline by t time units, a glitch could have been avoided if a starvation buffer of size $\frac{t}{dvr}$ were allocated to each stream. Consequently, by picking an acceptable glitch probability, one can compute the total amount of required buffer space as well as the amount of data that should be buffered for a stream before its streaming commences.

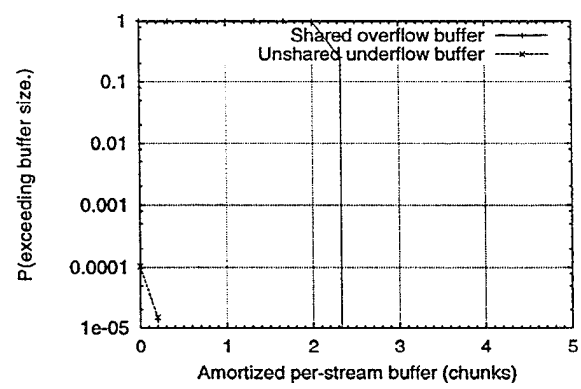

Figure 5: Probability of exceeding a specified buffer size (per stream). The overflow buffer is shown on the right and the underflow (starvation) — on the left. Selective exploitation of redundancy; faulty drive and urgent requests; $R = 0.90$; $k = 5$; $dvr = 10$.

Fig. 5 depicts the results for the case of selective exploitation of redundancy with a faulty disk and urgent chunks. One can see that a buffer of size 2.6 per chunk suffices. These results are the most meaningful ones,

Red. Exp.	Load	k	dvr	F	U	S.T.	Tot. Buff	Buff/ strm	pExp
NER	0.95	5	10	n	n	-35	675	5.75	0
FER	0.95	1	10	n	n	4.5	539	1.2	0.35
FERw	0.95	5	10	n	n	-05	991	3.8	0.75
FER1	0.95	5	10	n	n	-02	876	3.1	0.44
SER	0.95	5	10	n	n	-02	724	2.6	0.15
SER	0.95	5	10	n	y	-05	722	2.6	0.15
SER	0.90	5	10	y	n	-02	750	2.7	0.26
SER	0.90	5	10	y	y	-03	749	3.8	0.26
NER*	0.90	5	10	n	y	-20	718	4.4	0.16
SER	0.95	5	5	n	n	-07.3	450	4.5	0.13
SER	0.90	5	5	y	y	-13.6	460	5.7	0.26
SER	0.95	4	10	n	n	-01.3	685	2.4	0.13
SER	0.90	4	10	y	y	-01.0	686	2.4	0.22

Table 1: Summary of results for a permissible glitch probability of 10^{-4}. The first columns refers to redundancy exploitation (FERw and FER1 correspond, respectively, to full exploitation based on a comparison between the parity queue and the longest data queue or that of the first data chunk; NER* refers to exploitation of redundancy only when a requested data chunk resides in the faulty drive). F and U refer to faulty drive and urgent chunks, respectively. S.T. is the spare time remaining between the arrival of a chunk into the buffer and its streaming time (negative corresponds to a missed deadline). Tot. Buff. is the total overflow buffer used, and Buff/strm is the derived amortized per-stream total buffer requirement. Finally, pExp is the probability of exploiting redundancy.

since a change of policy can shift buffer requirement from one type to the other.

The most interesting results are depicted in Table 1. In the first batch of rows, we see that the required amount of RAM with selective exploitation is actually smaller than with full exploitation and has a much lower computation and memory-bandwidth overhead (lower probability of exploitation and the resulting need to reconstruct a data chunk).

The second batch shows that SER maintains a substantial advantage over NER even in the presence of a faulty drive. A lower value of *dvr* increases buffer size, since each chunk suffices for less streaming time.

Finally, $k = 4$ requires substantially smaller buffers in the presence of a faulty drive.

Simulations with different arrival patterns and variable disk service time yielded similar results. This also suggests that the variable transfer rate due to multi-zone recording can be accommodated automatically so long as placement is also randomized at the intra-disk level.

6 Discussion

We have seen that randomized (or other irregular) data layout in conjunction with selective exploitation of redundancy for load-balancing and avoiding occasional trouble permits operation at extremely heavy loads with very reasonable buffering and excellent response to urgent requests. We now shift our attention to other issues, such as communication overhead.

Storage-bandwidth overhead with our scheme is zero, since we read the same number of chunks regardless of whether redundancy is exploited. Similarly, communication overhead on the distribution network is zero, since reconstruction of missing chunks is carried out in the server.

Storage overhead is minimal ($1/k$) and is required for fault-tolerance. The same goes for computation of XOR and related memory traffic, since we exploit redundancy for load balancing with a relatively low probability.

Burstiness of traffic within the server is minimal, since we are operating at heavy load and the disks send data most of the time, regardless of what streams or parity groups the data belongs to. Also, the queueing delays were included in the simulation and starvation buffers were added as required. Finally, with the emergence of very-high-bandwidth disk interconnects, we expect burstiness of traffic within the server not to be very important.

We are presently in early stages of implementation of our architecture. We are constructing a Pentium-Pro based server with between 16 and 24 disk drives. The distribution network will include four 100base T Fast Ethernet lines and possibly a 155Mbps ATM line. The project is being carried out under Windows NT.

If one assumes the availability of some storage at the user end, a distributed implementation is also possible. One option would be a central server which delegates the XOR operations to the clients; another would distribute the server.

Design and operation become simpler when one employs replication instead of lower-overhead schemes such as parity. The main reason is that no operations need to be carried out among blocks. An interesting server that employs duplication is Microsoft's Tiger [15]. However, Tiger exploits the redundancy only when the primary copy is on a faulty disk or computer. Also, the operation of Tiger assumes streams with fixed data rates and identical disk drives. It may

be interesting to apply our approach to Tiger, thereby increasing its robustness and agility, and permitting it to operate at very heavy loads without simplifying assumptions. One must, of course, carefully consider control traffic and latencies in this case.

Finally, it is important to summarize some of the key observations that underly our approach:

- Exploitation of redundancy in video servers is costly, since data that is read prematurely must be buffered.
- The high ratio of disk rate to video rate sharply reduces the amount of buffer space that is required to mask queuing delays.
- Related to the above, it is permissible to encounter queues that are not empty or nearly so.
- Once the operating point does not assume nearly-empty queues, there is a possibility for carrying out some seek-related optimizations in the disk drives. These could improve the results.
- For the same reason, the sensitivity to variability in disk service time is sharply reduced.

7 Conclusions

The use of randomized layout in conjunction with selective exploitation of redundancy for load-balancing solves a large number of problems that arise in the design of a video server, paving the way to the construction of a simple, robust, flexible, high-performance video server. Any number of non-identical disk drives can be employed; streams needn't have fixed data rates; urgent user requests can be handled immediately without causing glitches in ongoing streams. Also, performance evaluation is simple since the randomized layout makes system performance depend only on the aggregate load. Finally, all this is attained with very small buffer requirements due to the selective exploitation of redundancy. Interestingly, this stochastic design enables us to give firmer (and higher) performance guarantees than would be possible with a more regular design. The reason, of course, is the underlying uncertainty regarding viewer requests and resulting load patterns.

The difference between architectures that do not exploit redundancy for load-balancing (e.g., use only smart scheduling, looking into the future, etc.) and the architecture described in this paper is fundamental: with the former schemes, the amount of work that must be done by any given disk cannot be changed, whereas the exploitation of redundancy permits it to be changed to some extent. While scheduling-only approaches can provide a some of the features claimed for our architecture, they are inherently incapable of providing others.

Finally, selective exploitation of redundancy appears to be useful in a variety of application domains, as demonstrated in [16] for distributed systems. We are continuing to explore its applicability and merits in various contexts.

Acknowledgment. The simulations were carried out using the Workbench simulation package by Scientific and Engineering Software, Inc.

References

[1] M.G. Kienzle, A. Dan, D. Sitaram and W. Tetzlaff, "Using tertiary storage in video-on-demand servers", *Proc. IEEE CompCon*, 1995, pp. 225–233.

[2] S.R. Heltzer, J.M. Menon and M.F. Mitoma, "Logical data tracks extending among a plurality of of zones of physical tracks of one or more disk devices", *U.S. Patent No. 5,202,799*, April 1993.

[3] Y. Birk, "Track-Pairing: a novel data layout for VOD servers with multi-zone-recording disks", *IEEE 1995 Int'l conf. on Multimedia Comp. and Sys.* (ICMCS95), Washington, D.C., May 15-18, 1995. Also, Hewlett Packard Technical Report HPL-95-24, March 1995.

[4] S. Chen and M. Thapar, "Zone-Bit-Recording Enhanced Video Data Layout Strategies", *Proc. of 4th Int'l Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'96)*, San Jose, CA, Feb. 1996, pp.29-35. Also, Hewlett Packard Technical Report HPL-TR-95-124.

[5] A.L. Narasimha Reddy and J. Wyllie, "Disk scheduling in a multimedia I/O system," *Proc. ACM Multimedia* 1993, pp. 225–233.

[6] D.A. Patterson, G. Gibson and R.H. Katz, "A case for redundant arrays of inexpensive disks (RAID)", *Proc. ACM SIGMOD*, pp. 109–116, June 1988.

[7] F.A. Tobagi, J. Pang, R. Baird and M. Gang, "Streaming RAID — A disk array management system for video files", *Proc. 1st ACM Int'l Conf. on Multimedia*, Aug. 1–6, 1993, Anaheim CA.

[8] M. Holland and G.A. Gibson, "Parity declustering for continuous operation in redundant disk

arrays," *Fifth Int'l Conf. on Architectural Support for Programming Lang. and Operating Sys. (ASPLOS-V)* SIGPLAN Not. (USA), SIGPLAN Notices, vol.27, no.9, p. 23-35 1992.

[9] S. Berson, L. Golubchik and R.R. Muntz, "Fault tolerant design of multimedia servers," *Proc. SIGMOD '95*, San Jose, CA, May 1995. (Also, Technical Report No. CSD-940040, UCLA, October 1994.)

[10] H.M. Vin, S.S. Rao and P. Goyal, "Optimizing the placement of multimedia objects on disk arrays", *Proc. Int'l Conf. on Multimedia Comp. and Sys.*, pp. 158-165, May 1995.

[11] N.F. Maxemchuk, "Dispersity Routing", *Proc. Int'l Commun. Conf.*, pp. 41.10-41.13, 1975.

[12] M.O. Rabin, "Efficient Dispersal of Information for Security, Load Balancing, and Fault tolerance", *J. ACM*, vol. 36, pp. 335-348, Apr. 1989.

[13] A. Bestavros, "IDA Disk Arrays", *Proc. PDIS'91, the First Int'l Conf. on Parallel and Distributed Information Systems, Miami Beach, Florida*, IEEE Computer Society Press, December, 1991.

[14] S. Berson, R.R. Muntz and W.R Wong, "Randomized Data Allocation for Real-time Disk I/O", *Proc. IEEE Compcon'96* vol. 11, no. 4, pp. 631-640, May. 1996.

[15] W. Bolosky, J.S. Barrera, III, R.P. Draves, R.P. Fitzgerald, G.A. Gibson, M.B. Jones, S.P. Levi, N.P. Myhrvold, R.F. Rashid, "The tiger video fileserver", *Proc. NOSSDAV96*, April 1996. Also Microsoft report MSR-TR-96-09.

[16] Y. Birk and N. Bloch, "Prioritized dispersal: a scheme for selective exploitation of redundancy in distributed systems", *Proc. 8th Israeli Conf. on Computer Sys. And Software Engr.* (ISySE'97), June 1997 (to appear).