Design of the RIO (Randomized I/O) Storage Server

Jose Renato Santos
Richard Muntz

Abstract

We describe the design and implementation of the RIO (Randomized I/O) multimedia object server which manages a set of parallel disks and supports real-time throughput and delay guarantees. This storage subsystem was implemented as part of a multimedia information server which supports multiple concurrent applications such as video on demand and 3D interactive virtual worlds. We discuss the principal issues and innovations involved in the design and implementation of the RIO storage system, and present experimental performance results measured on our prototype.

A multimedia data server must be ready to handle a variety of realtime object types (video, audio, 3D interactive virtual worlds, etc.) as well as non realtime workload. Achieving simultaneously (1) high utilization and (2) low latency with a high degree of certainty is the challenge. Our prototype system provides a statistical guarantee of quality of service. Our experimental results shows that it is possible to achieve a very small probability of missing a deadline (less than 10^{-6}*), with a relatively high disk utilization (70% to 99%, in terms of fraction of the maximum disk capacity) , together with relatively small deadlines (on the order of 0.5 sec. to 1.5 sec), using contemporary disks.*

To achieve guaranteed low delay with high utilization requires good short term load balancing to keep disk queue length distributions from having long tails. For workloads with predictable access patterns it is possible to try to exploit this predictability to gain the desired effect. However this is an approach with limited applicability and quickly becomes untenable as the workload becomes more diverse. Our approach is to randomize the physical allocation of disk blocks across the system's multiple disks. At the level of physical disk block access this turns all workloads into the same uniformly random access pattern and thus gives us one problem to deal with. Since each disk is equally likely to be the target of each disk access, this approach provides long range load balance. However short term statistical variations can result in short term imbalances in disk queues which in turn imply either (1) the guaranteed maximum latency must be increased or (2) a higher probability of missing the deadline. Our approach to counteracting this problem is to introduce limited redundancy by replicating (at random) some fraction of the data blocks. This data redundancy gives the disk scheduler some flexibility in scheduling disk block reads, allowing the system to reduce the short term load imbalance.

1 Introduction

1.1 Motivation

Advances in technology, have enabled the increasing usage of information systems for storing and retrieving multimedia data, such as images, video, audio, 3D graphics, etc. These new data types have different characteristics and require new approaches when designing systems. High bandwidth and large storage space requirements are among these characteristics. Moreover, continuous media data impose realtime restrictions on the retrieval and delivery of information which must be present at the client platform prior to the time it is needed for display. Failure to provide realtime service can result in disruptions and delays which compromise the quality of the presentation observed by the end user. To achieve high quality continuous media playout, such disruptions must be reduced to very low levels. The realtime characteristic of continuous media affect the design of multimedia systems at different levels, such as data storage, information management, processor and memory management, data communication, data display, etc. In this work we concentrate only on data storage issues for multimedia data. The other aspects of multimedia systems are equally important, but are not discussed here. Our assumption is that multimedia objects are to big to fit entirely in memory and need to be retrieved from disk on demand. Further, parallel disks are required for the bandwidth and storage capacities anticipated for multiple clients and for high performance applications.

Although video and audio require real time service their workloads have a certain degree of predictability since a typical playout accesses data sequentially. This predictability has been exploited in many multimedia data servers designs in which, based on the sequential access pattern, data is carefully layed out on disk drives such that contention for the drives is avoided and realtime guarantees can be made. This approach can work well when the workload is highly predictable. However, in practice several factors reduce the workload predictability and makes the problem of optimal data lay out very difficult, if possible at all. One of these factors is that video and audio are generally compressed by encoding techniques such as MPEG1 and MPEG2. These encoding techniques may generate variable bit rate (VBR) media streams for achieving a constant display quality, which introduces a temporal variability to the I/O pattern. In addition, providing VCR features such as *Pause*, *Fast Forward* and *Rewind*, also reduces the predictability of a stream I/O access pattern. Finally, multi resolution encoding schemes such as are found in MPEG2 complicate layout and scheduling techniques [7].

More important yet is the fact that new multimedia applications, such as 3D interactive virtual worlds, have I/O patterns much less predictable than video or audio. In a 3D interactive virtual world application the user navigates through large 3D graphic models at variable speed and along user controlled paths. In order for the display engine to show the current view of the world to the user, the graphical models of nearby 3D objects need to be continuously retrieved from disk as the user moves. The access pattern of this type of data thus depends on the speeds and paths selected by the user, which makes prediction imperfect at best. 3D virtual world models have been used for different applications such as architectural building design[8], urban city models[15], scientific visualization [16][9], etc.; and will be increasingly common in the future.

Because of the difficulties in predicting the I/O pattern of multimedia data access, we believe that multimedia data servers will move towards solutions that do not rely on careful data layout. Our approach to the problem is to use a random scheme for laying out data on disks that results in a uniformly random access pattern at the physical level regardless of any reference pattern at the logical level. This approach has the advantage of mapping the workload of different multimedia applications into the same workload at the disk access level. Thus we have a single problem to solve. Once we have solved it, the same storage system can be used for different multimedia applications and therefore we have a generic multimedia storage system capable of concurrently storing and retrieving many types of media objects.

RIO is the storage subsystem for a multimedia information system under development at UCLA called the Virtual World Data Server (VWDS). The VWDS addresses many different issues including data storage, memory management, network communication, admission control, traffic policing and shaping, and also specific issues of 3D model virtual worlds, such as spatial indexing, adaptive quality of service, user motion and perception modeling, etc. However, in this paper we focus only on the storage subsystem part of the data server, and provide a brief overview of the other system components to put the storage system in context. The prototype is implemented on a SUN E4000 machine, having 10 Ultrasparc processors, 1.25 Gbytes of shared memory, 14 disks with 4 Gbytes each dedicated to multimedia data storage, with ATM and Ethernet connections to client machines. The first prototype is operational and has been used to simultaneously support delivery of MPEG encoded videos, and 3D urban simulation city models. Other applications for realtime scientific visualization and medical VR are under development.

1.2 Paper organization

This paper is organized as follows. In section 2 we briefly describe the overall architecture of the VWDS. In section 3 , we discuss the novel aspects of the RIO system and compare its design with that of other approaches. In section 4 we describe some design issues of the RIO storage subsystem, namely the pseudo random data layout on disk, the allocation of blocks scheme and the system software architecture. In section 5 we present and discuss the principal experimental performance results obtained with our prototype. In section 6 we extend the performance study introduced in Section 5 using a validated simulation to analyze the system performance when using a larger number of disks than were available in our prototype. In section 7 we compare the performance of our system to a conventional streaming RAID scheme for the special case of a pure video-on-demand workload. The comparison is in some sense, a worst case for RIO since video is presumed to have purely sequential access and data layout can be optimized specifically for this case. Our results indicate that RIO is at least competitive and can sometimes outperform streaming RAID even in this case. Finally, in section 8 we present our conclusions.

2 The VWDS Data Server

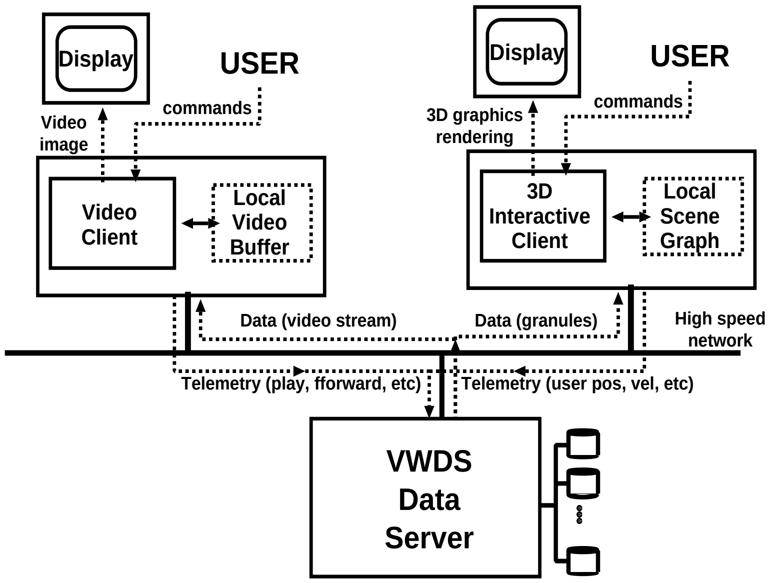


Figure 1: VWDS Data Server Architecture.

In this section we give a brief description of the principal components of the VWDS.

Figure 1 illustrates a typical configuration in which client machines access the data server through a high speed network, which in our prototype is implemented by an ATM network. Users run applications such as video players, 3D virtual world simulations, etc., on client machines that access data stored on the VWDS. Clients periodically send telemetry information to the server, which is used to compute the next logical set of data that needs to be sent to the client in the immediate future. For 3D virtual worlds, telemetry information consists of current eye position, direction of movement, speed, etc. The server uses this information, combined with a model of user mobility to generate a list of scene objects (logical subdivisions of data, representing 3D graphical objects geometry and texture) that need to be retrieved in the immediate future. In the case of video, telemetry information consists basically of user commands for VCR functionality such as *Pause*, *Fast Forward*, and *Rewind*.

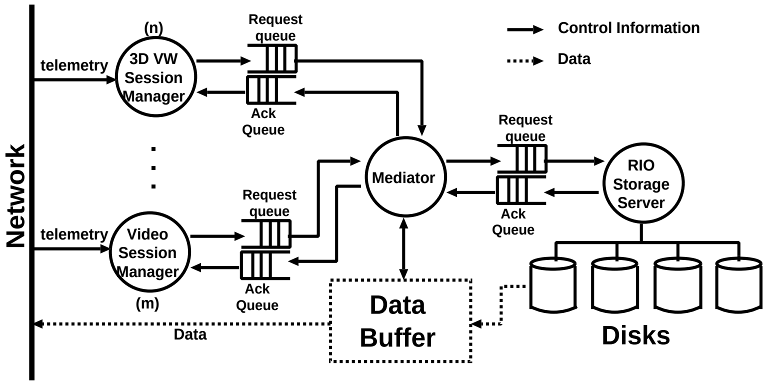


Figure 2: RIO Data Server Software Architecture.

Figure 2 gives a slightly expanded description of the VWDS storage architecture on the server side. Three major components on the system are identified; the RIO Storage Server, the Mediator and the Session Managers.

A session manager is specific to a type of application. Within a Session manager new threads are dynamically created as users start new sessions with the VWDS. The current prototype has session managers for two types of applications; video and 3D virtual worlds. The 3D Virtual World session manager is the most complex and is responsible for the logical organization of 3D models in terms of scene objects and it utilizes a spatial index for accessing the 3D virtual world data. The 3D session manager predicts the logical scene objects that will be needed on the client in the near future, based on the most recent telemetry information, a motion model of the user, and a description of the viewing frustum. A list of the needed scene objects, with deadlines, are then submitted to the Mediator. The required client deadline can not be greater than the sum of the maximum delay guaranteed by the RIO object server and the maximum delay to transmit the data through the network (In our current prototype, network guaranteed delays are still not provided, but will be added in future versions). The video session manager is much simpler and basically generates requests to sequential blocks of data on a video object, according to the current position of the data stream.

The Mediator is responsible for managing the available systems resources such as disk bandwidth, network bandwidth and buffer space across all active sessions. The resources are managed by the Mediator using a combination of techniques at two time scales. One of these techniques is the admission control scheme, which reserves a minimum amount of resources for a particular session when the session is created. If the system can not reserve the minimum resources required by the session the session is not accepted. This guarantees a minimum quality of service for each session. Any extra capacity available at the server is dynamically allocated to sessions based on the current demand and expected quality of service of sessions. Although still not implemented in our current prototype, in future versions the mediator will use data alternatives with different levels of detail provided by the session manager to adapt a session's current demand for system resources based on the impact this will have on the QoS for the session relative to other sessions. The Mediator is also responsible for mapping an access to a logical segment of data to a set of disk block requests to be submitted to the RIO Object Server and for managing the data buffer space.

The RIO storage subsystem is responsible for the allocation/deallocation of space for multimedia data objects on the multiple disks of the system and for scheduling disk block accesses to meet deadlines. This system component is the focus of this paper and is described in the following sections.

3 Realtime disk scheduling

3.1 Realtime scheduling of a single disk

The general problem of scheduling realtime requests to a single server is well studied and has been addressed in several papers. Liu & Layland [17] showed that for a set of preemptable tasks generating periodic requests, an optimal scheduling policy is the EDF (Earliest Deadline First) algorithm, that schedules requests in the order of their deadlines possibly preempting requests being serviced when a request with an earlier deadline arrives. The EDF algorithm is optimum in the sense that if there exists a feasible schedule where no request misses its deadline then scheduling the requests according to the EDF policy will guarantee that no request misses its deadline. Later, Jeffay et. all [14] extended this result and showed that the EDF policy is also optimal when the requests are not preemptable and when tasks are not periodic, but have a minimum interval between requests.

However, these results assume a work conserving server,which is clearly not the case for disk I/O. Serving requests ordered by their deadlines can lead to poor performance since this ignores disk seek overhead. The problem of optimizing disk I/O has been extensively studied in the case of non realtime I/O and several scheduling algorithms have been proposed. The most important and well known algorithms are variation of the SCAN algorithm in which requests are processed in the order of their cylinder position on the disk as the disk head moves in the direction from the outermost cylinder to the innermost cylinder or vice-versa. A realtime scheduling algorithm, must optimize the I/O cost and at the same time avoid requests missing their deadlines. Thus, a combination of disk I/O optimization techniques with real-time scheduling algorithms is required to achieve best results. We are not aware of any theoretical work that results in an optimal disk scheduling policy for realtime requests. However it is reasonable to assume that combining traditional disk I/O optimization techniques with real-time scheduling algorithms can lead to better disk utilization than simple realtime scheduling based only on the request deadlines. Combining disk SCAN algorithms with realtime scheduling is not new and has been proposed in other related works. [21][27]

For simplicity, in this paper we will only consider disk scheduling algorithms that attempt to minimize seek time. Although minimizing rotational latency is also possible, it is more complex and difficult to implement outside the disk controller. We adopted a modified bidirectional SCAN or elevator algorithm, which we call MSSCAN (maximum size SCAN), where the number of requests that can be serviced in one disk scan is limited to a maximum value n_{max}. Limiting the number of requests, allows the duration of a disk cycle (i.e one scan of the disk arm), to be bounded which, in turn, is important when deadlines must be guaranteed.

We assume that a disk cycle is non preemptable, and requests arriving during one disk cycle are deferred to the next disk cycle. (We will consider the case where a request could possibly be included in the current ongoing cycle, as future research.) At the beginning of each cycle requests are ordered according to their deadlines and the first n requests up to n_{max} are scheduled in the next cycle. The requests are then ordered according to their cylinder position on the disk surface and then served in the increasing (or decreasing) order of their cylinders position, depending on the direction of the next scan.

We call a request *successful* if it is served on the cycle following the one that is active when it arrives, and we call it a *delayed* request otherwise. If the number of requests waiting for service at the beginning of a cycle does not exceed n_{max} there will be no *delayed* requests on that cycle.

Since requests can be served in any order within a given cycle the worst case delay of a *successful* request happens when the request arrives just after a cycle starts and is the last request to be served in the following cycle, as is illustrated in figure 3. Thus, the maximum delay of a *successful* request is 2 times the maximum

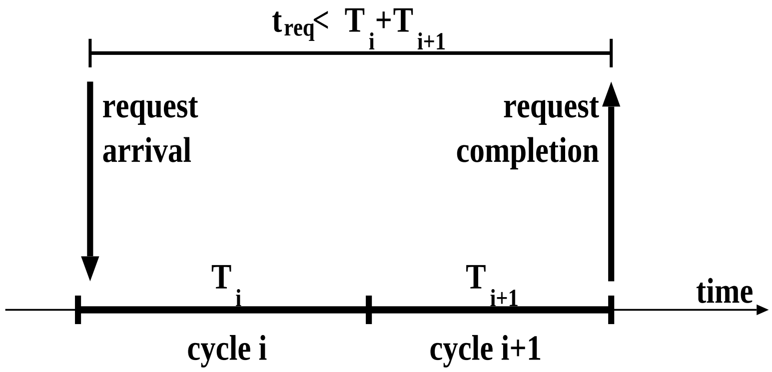


Figure 3: Successful request worst delay.

duration of a cycle T_{max} . The delay bound guaranteed by the RIO system will be related to $2T_{max}$.

3.2 Multiple Disks

Continuous media data typically requires large amounts of space and high bandwidth. Although current commercial disks, having an average data rate of 30 Mbps to 100 Mbps, can support a number of, say,, MPEG1 videos (typically 1.5 Mbps) or MPEG2 videos (typically 6-10 Mbps), in order to support a large number of concurrent users as well to support media objects with higher bandwidth requirements such as High Definition TV, parallel disk systems are required.

In order to effectively use the disk bandwidth available in a parallel disk system, we must ensure that the data requests are evenly distributed among all disks. We should avoid the situation where a significant fraction of requests are concentrated on a few disks even for a short time, and ideally we would like to have all the load evenly distributed among all disks at all times, such that we could use the full bandwidth of all disks and such that any arriving request will have a relatively short waiting time. This problem of load balancing is addressed in different ways in different systems.

Much of the work on continuous media servers has concentrated on video playback which has the nice property of being highly predictable. Once a video playback stream is started the video object is sequentially read from the storage system. This high predictability is exploited in some video servers to carefully layout data on disks to achieve a good load balance of the data requests to the systems disks.

A common data layout scheme is striping which is round robin allocation of blocks to disks, where logically consecutive blocks of an object are allocated on consecutive disks, cycling over all disks. Then, accessing the object sequentially, will generate accesses cycling through all disks. The great advantage of striping is that it decouples storage capacity from disk bandwidth, since each object uses the bandwidth of all disks and thus disk overloading due to skew in object popularity is not an issue.[6] [24] [11] [25] (Note that RIO has the same property)

There are basically two approaches to achieve load balancing on striped systems. One possibility is to use large reads requesting data from all disks at a single request for each active stream [24]. This guarantees that the disks will all have the same load, providing perfect load balancing. However this approach has some serious drawbacks. For efficient disk bandwidth utilization, each block allocated to a single disk must have a relative large size (in general 1 to a few tracks), to amortize seek and rotation latency. To achieve the same disk utilization a data striping system with large reads over D disks needs to have requests with sizes D times greater than the ones on a single disk system. For predictable workload like video streams, this has the effect of having larger buffers for streams. On less predictable workload, like 3D interactive virtual worlds, where data access is not sequential the situation is worse and the system may have to read a large amount of superfluous data.

Another way of accessing striped data is to use small requests accessing just one block on a single disk, with sequential requests cycling over the disks.[6][11]. This solves the problems associated with large reads but requires a new approach to ensure load balancing. A common approach, called Streaming RAID [24], is to process requests in cycles of constant duration synchronized on all disks. In each cycle, each active stream access a single block on a particular disk. On the next cycle each stream accesses the following block that is located on the next consecutive disk. Thus, the total load of each disk, moves to the next disk on consecutive cycles. Thus if there is bandwidth available in the system this bandwidth can be used by a new stream. The new stream only needs to wait until the available bandwidth cycles through the disks and reaches the disk that contains the first block of that object. After that the stream can start and be sure that it will have an available constant bandwidth of one block per cycle during the complete sequential reading of the object. Therefore, at the cost of a start up latency the bandwidth of the storage system can be fully utilized with a delay bound equal to the cycle length. However there are some drawbacks in this approach. First, this scheme requires that all streams have the same constant bandwidth. This limits the use of this type of systems in supporting different type of media objects like MPEG and MPEG2 video, sound, etc. Also, a single video may have a variable bit rate (VBR) when encoded using MPEG or other similar compression techniques. For the same reason it is hard to provide VCR functionality on this type of system. Also, in each cycle the same number of block accesses can have different durations due to disk seek times and rotation latency. Therefore the disk cycle that needs to be synchronized on all disks must be specified considering the worst case for the disk access. Thus most of the disks will be idle toward the end of a cycle while waiting for the beginning of the cycle and this enforced idle time reduces the effective bandwidth of the system. This system therefore saturates at some load which is less than the capacity of a system which does not impose such idle times. The implication is that, at high loads, the system that does not have the synchronization imposed idle times will perform better. We will show this explicitly when we compare the RIO approach to Streaming RAID in section 7.

Berson et all [2] propose a staggered striping scheme where instead of requests accessing a single disk on each cycle, they access multiple blocks on consecutive disks. On the next cycle the same number of disks are accessed, but the disks accessed are shifted by one or more disks. This allows a stream to access more than a disk block in each cycle, while still shifting the disk load every cycle.This scheme improves simple striping and allow multiple playout bandwidths, by having different number of blocks accessed on each cycle, depending on the object playout bandwidth. One problem in this approach is bandwidth fragmentation. This happens if a new stream requires access to a object with multiple block access per cycle, and although there is bandwidth available on the system, this bandwidth is not available on consecutive disks, and thus can not be allocated to the new stream. While staggered striping effectively deals with videos of different playout rates, it does not address any of the other types of access pattern variations that RIO does.

3.2.1 Random Data Allocation

In the last section we discussed current real-time data servers (mostly video servers) using disk striping over multiple disks and we pointed out some drawbacks of these approaches. We now propose a new data layout scheme. In this scheme a storage object is an array of fixed size blocks. The data blocks of an object are randomly mapped to disks blocks. Random data allocation has been proposed for non real-time system such as RAMA parallel file system proposed by Miller and Katz [18], but to our knowledge there has been no published work that uses random data allocation on real-time systems. Yitzahak Birk informed us through personal communication that his group has worked on video servers based on random data allocation at the Technion in Israel, but their work is still unpublished and unavailable at this time.

Random block allocation naturally provides long term load balancing; after a sufficient large number of requests uniformly distributed over the disks, by the law of large numbers the number of requests submitted to each disk will be approximately the same. However, short term statistical variations can still cause some short term load imbalance, which can lead to a relatively long tail to the response time distribution for individual requests. For a given desired maximum latency, this can force us to reduce the maximum average load that can be submitted to the system to provide slack time that can absorb the short term fluctuations. Our approach is different. Our solution is to use partial replication and an on-line, shortest-queue scheduling algorithm as discussed later in this section.

At this point we enumerate several claimed advantages of the random data allocation used in the RIO storage subsystem which will be substantiated in the following sections.

1. **Support for unpredictable access patterns**

   For applications with unpredictable access patterns like 3D interactive virtual worlds, carefully data layout is impracticable and random allocation seems the appropriate solution. Even the extreme case where access is completely random is properly handle by RIO, since this is completely equivalent to any other access pattern. They are all mapped to sequence of random access at the physical level.

2. **Robustness with respect to data hot spots**

   It is known that in most storage system a small fraction of the data is accessed by a large fraction of the I/O requests. These more popular data, usually called hot spots, are present in most systems with various degrees of intensity, depending on the particular data and application. In general this could lead to some load imbalance if the hot spots were concentrated on a few disks of the system. For example, if we distribute the data on disks based on logical objects and most of the I/O requests access data on few objects located on a few disks of the systems, these disks will have a higher load than the others. In our system, however, the hot spot data will be mapped to a set of hot physical blocks randomly distributed over the disks. The location of each hot physical block is independent of the location of any other hot physical block, even if they belong to the same popular object. Therefore, if the number of hot data blocks is sufficient large (more than a few blocks per disk), which is probably the case for most applications, they will be approximately equally distributed among the disks. In the extremely unprobable case in which the number of hot blocks is very small, that could lead to a skewed load, this skewed load could be easily removed by a simple disk cache with the size of only a few blocks per disk.

3. **One traffic pattern at the physical disks**

   Any logical data access pattern is mapped to a random physical access pattern. Thus, any data traffic will look the same for the system, in terms of its I/O pattern, independent of the application, giving us a single problem to to deal with. This uniformity on the traffic allows the system to be shared by many different applications, like video on demand, 3D interactive virtual worlds, interactive image systems, conventional non real time data access, etc.

4. **Asynchronous cycles**

   Since requests are not carefully layed out on disks, there is no need to have synchronized disk cycles on the system disks. This improves the performance of the system, since disks do not need to be idle, waiting for other disks to finish their cycles.

5. **Multiple playout rates and Variable bit rate**

   Since no careful layout or time synchronization among the disks is assumed, support for streams with different playout rates are naturally provided. Also, changing the access rate of any stream, which is required by VBR video encoding for example, depends only on the total available bandwidth of the system and not on the temporal pattern of available bandwidth of individual disks as is the case when for example staggered striping is used. For the same reason, providing VCR functionality is much easier.

6. **Adequate for dynamic quality of service**

   Streams can change the quality of the service at any time by changing the data they access, from a higher to a lower level of detail or vice-versa, since changing the logical pattern does not change the random nature of the physical access pattern. Changing the quality of service is an efficient way of dynamically changing streams rate requirement and adapt to the current system load.

7. **Request size**

   Most of the previous advantages are also present on data striping when using long reads, that forces all requests to use bandwidth of all disks. However as already pointed, random data allocation has the advantage of requiring much smaller data requests sizes for achieving the same disk utilization as data striping, which lets the system buffers be smaller, and reduces the probability of fetching unnecessary data.

Random data allocation, on the other hand can not provide absolute realtime guarantees, but only statistical guarantees. Although random allocation will statistically distribute data requests among all disks there is some possibility of short term skewed traffic patterns that could unequally load the disks. Therefore, to provide an absolute guarantee, a random allocation system would have to work on very low system utilization such that even if all load were mapped to a single disk the deadlines would still be satisfied. Instead, we adopt the approach of statistical guarantees, and guarantee that the probability of missing a deadline is smaller than ϵ, where ϵ can be made very small. Another limitation of random allocation is that the system can not be loaded at full disk utilization, i.e. at the average effective data rate that could be supported by the system disks, since some room for statistical variation should be provided. However, our experimental results, presented later, show that high disk utilization with small probabilities of missing deadlines are possible.

3.2.2 Load Balancing

Random mapping of data blocks to disks tends to equally distribute the load among all disks, in the long run. However, in the short term the load fluctuates on each disk around the ideal balanced load. The higher the variance of the disk load the smaller the total load that can be handled by the system, since we must provide space for these load fluctuations and still be able to meet the requests deadlines. Our main goal is thus to reduce the variance of the instantaneous load on each disk with respect to the ideal fraction of the total load that would give us a perfect balanced system. To reduce this variance we propose a partial data replication scheme where a fraction of the blocks stored on the system is replicated. Data replication improves load balancing since some of the load directed to disks with high load can be redirected to less loaded disks that contain a copy of the same data.

A classical problem studied in the probability and statistics literature is the problem of sequentially placing $\mathbf{n}$ balls in $\mathbf{m}$ boxes, choosing a random box for each ball. This problem is very similar to our problem of scheduling data requests to disks. The theoretical results for this problem is summarized as follows. It is known that as $n \rightarrow \infty$ and $m \rightarrow \infty$, after all balls are placed on the boxes, the fullest box will have with high probability $\mathbf{n/m} + ln\mathbf{n}/lnln\mathbf{n}$ balls. Azar et all [1] describe a variant of this problem where each ball comes with d possible destinations, chosen independently and uniformly at random and the ball is placed on the least full box among the d possible ones. They show that in this case, the fullest box will contain with high probability, $\mathbf{n/m} + lnln\mathbf{n}/ln\mathbf{d}$ balls. This result shows that the difference between the maximum number of balls in each box and the average size n/m is exponentially reduced when we have multiple choices of where to place the ball.

Mitzenmacher [19] extended Azar et all results to more realistic dynamic systems, where balls enter and leave the system over time, describing the problem as a queuing system model. They do a very detailed analysis of this dynamic model that they call the *supermarket model* in the limiting case of infinite servers (they also study the error bound of this assumption, when having a finite number of servers) and show an exponential improvement in the mean waiting time of customers in the system when multiple choices are provided as compared to the case where customers have only one choice, showing the effectiveness of what they call "*The Power of Two Choices in Randomized Load Balancing*".

These theoretical works support our assumption, which is also confirmed by our experimental results, that replicating data on multiple disks can significantly reduce the variation of individual disks loads, providing a more balanced and efficient system.

Data replication is not new and has been used in many data storage systems, both for providing dynamic load balancing and also for providing reliability under disk failure. However we believe that the use of partial data replication for load balancing on a system for realtime I/O with random data allocation has not been explored in any other published work.

In RIO a fraction α $(0 \leq \alpha \leq 1)$ of the total number of stored blocks is replicated while the others are not. We study the system for different values of α, varying from 0 (no replication) to 1 (full replication).

The load balancing algorithm we use in RIO is a simple on line algorithm. When a new request arrives at the system, if the requested block is replicated it can be placed on 2 possible queues and the one that has the smallest number of requests is selected (ties are broken at random). If however the block is not replicated it is placed on the only possible queue.

The cost, for having better load balancing and increased disk bandwidth utilization is the use of extra storage space. Current disk technology trends, shows that storage capacity is growing faster than bandwidth[12]. We thus believe that disk bandwidth will be the bottleneck for multimedia applications, specially if a large number of streams (users) needs to be supported.

If we replicate a percentage α of data, one may argue that by buying a α percentage of more disks the disk utilization (with same load) is decreased and deadlines will be met with higher probability due to lower utilization. One response could be that the replication may not require buying more disks; it depends on the relative requirement for storage versus bandwidth. We might have excess storage capacity because the disks were bought for their bandwidth. If however we do have to buy more disks to provide storage for replication, our scheme allows us to also use the additional disk bandwidth to support a higher number of active streams (users).

4 The RIO Storage Subsystem

4.1 RIO Data Object

In RIO a data object is a sequence of constant size data blocks. The size of a block is a system parameter and is specified at the time the storage system is created. Typical values for block size used on performance analysis of our prototype were in the range from 16Kbytes to 256Kbytes. Storage space is allocated to objects in units of one block.

An object, may contain references to other objects and be referenced by any number of objects. The creation of a new object on the system requires that one of the already existing objects add a reference to the new object. At the storage system creation a special object, named root object, is automatically created and is used as the starting point for populating the system with new objects. Except for the root object, all other objects must have at least one other object referring to it. When the last reference is removed the object is deleted from the system.

An object is identified by an unique object ID, which is selected by the system at the time of its creation. Each object is described by a control header structure stored on the storage system , named *objnode*. The *objnode* is very similar to the *inode* structure of UNIX^{TM} file systems. A *objnode* has 256 bytes and contains the following information: the object id; a user defined data type; access permission flags; object size; user and group id of object owner; times of creation, last access and last modification, number of referenced objects, and a reference counter that counts how many objects refers to it. The *objnode* also contains object references and pointers to the object data blocks, possibly with up 2 levels of indirection, to support large objects.

4.2 Data blocks indexing

In RIO there are 3 levels of data blocks indexing : object data block (*ODB*) index , system data block (*SDB*) index and disk data blocks (*DDB*) index.

A *DDB* index identifies a data block on a single disk. It is a an integer on the range [0,n_{DDB}-1], where

n_{DDB} is the total number of data blocks stored on a single disk. *DDB* blocks are stored continuously on the disk space and there is a simple mapping of a *DDB* index to the first disk sector containing that data block. Since in general disk tracks are not multiple of the data block size, some blocks may cross track boundaries, causing an additional overhead when accessing them. However, since the overhead of crossing cylinder boundaries are in general higher we decided to avoid this situation on our prototype, by skiping the last sectors of a cylinder if necessary.

A *SDB* index identifies a data block on the entire storage system hiding the existence of multiple disks. The range of a *SDB* index is [0,n_{SDB}-1], where n_{SDB} is the total number of data blocks stored on the entire storage system. A *SDB* block is mapped to a single *DDB* block on one of the system disks if the block is non replicated or to 2 *DDB* blocks on 2 different disks if the block is replicated.

A *ODB* index identifies a data block on a particular object. The range of a *ODB* index is [0,O_{size}-1], where O_{size} is the size of the given object. The mapping of *ODB* blocks to *SDB* blocks is done on the object *objnode* structure (for the first 18 data blocks of the object) and by indirect and double indirect block indexes (for the remaining data blocks).

4.3 Mapping System blocks to Disk Blocks

The mapping of *SDB* blocks to *DDB* blocks needs to satisfy 2 requirements. First, a fraction α of the *SDB* blocks are replicated, while the others are not. Second, the replicated blocks must be equally distributed among all possible combinations of disk pairs, while non replicated blocks are equally distributed among the disks. We assume here, that we have a uniform set of disks, with identical bandwidth and storage space. The extension of the system to handle heterogeneous disks is currently under investigation and is left as future work.

To meet the requirements we propose the following disk layout for a system with D disks. We subdivide the total space on each disk into multiple disk segments (*DS*) as illustrated in figure 4. We define a Storage Segment (*SS*)as the space of D *DS* segments composed by one *DS* segment at the same relative position on each disk.

The number of *DDB* blocks in each *DS* segment is defined as illustrated in figure 5. The first $D-1$ blocks are used for storing the primary copy of replicated *SDB* blocks. Each one of these *DDB* blocks is combined with a secondary copy on each of the remaining (D-1) disks to generate (D-1) *SDB* blocks. These

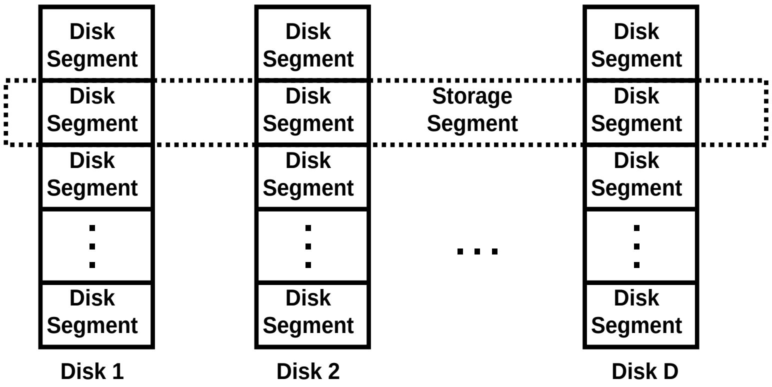


Figure 4: Storage Segments.

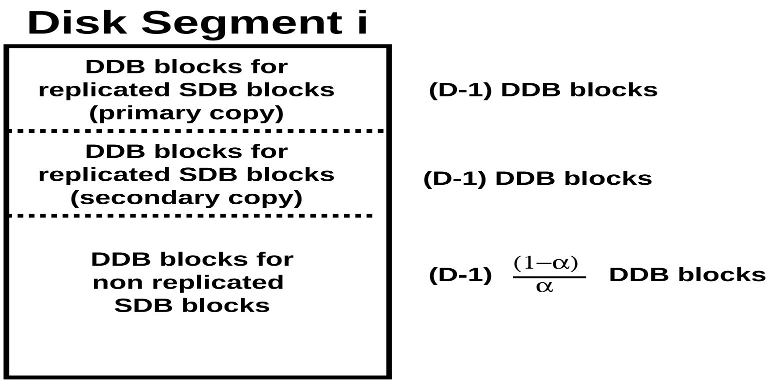


Figure 5: Disk Segments.

secondary copies are stored on the second set of $(D-1)$ DDB blocks of the corresponding disk DS segment. The remaining $(D-1)((1-\alpha)/\alpha)$ blocks on the DS segment are used to store non replicated blocks. This scheme ensures that each SS segment contains all possible $D(D-1)$ combinations of disks storing replicated blocks and that a fraction α of the total number of SDB blocks is replicated.

The total number of DDB blocks in each DS segment is given by:

$$n_{DS} = \frac{D-1}{\alpha} + (D-1)$$

Thus, since (D-1) of these blocks are used for replication, the total number of SDB on a SS segment is given by:

$$n_{SS} = D\frac{D-1}{\alpha}$$

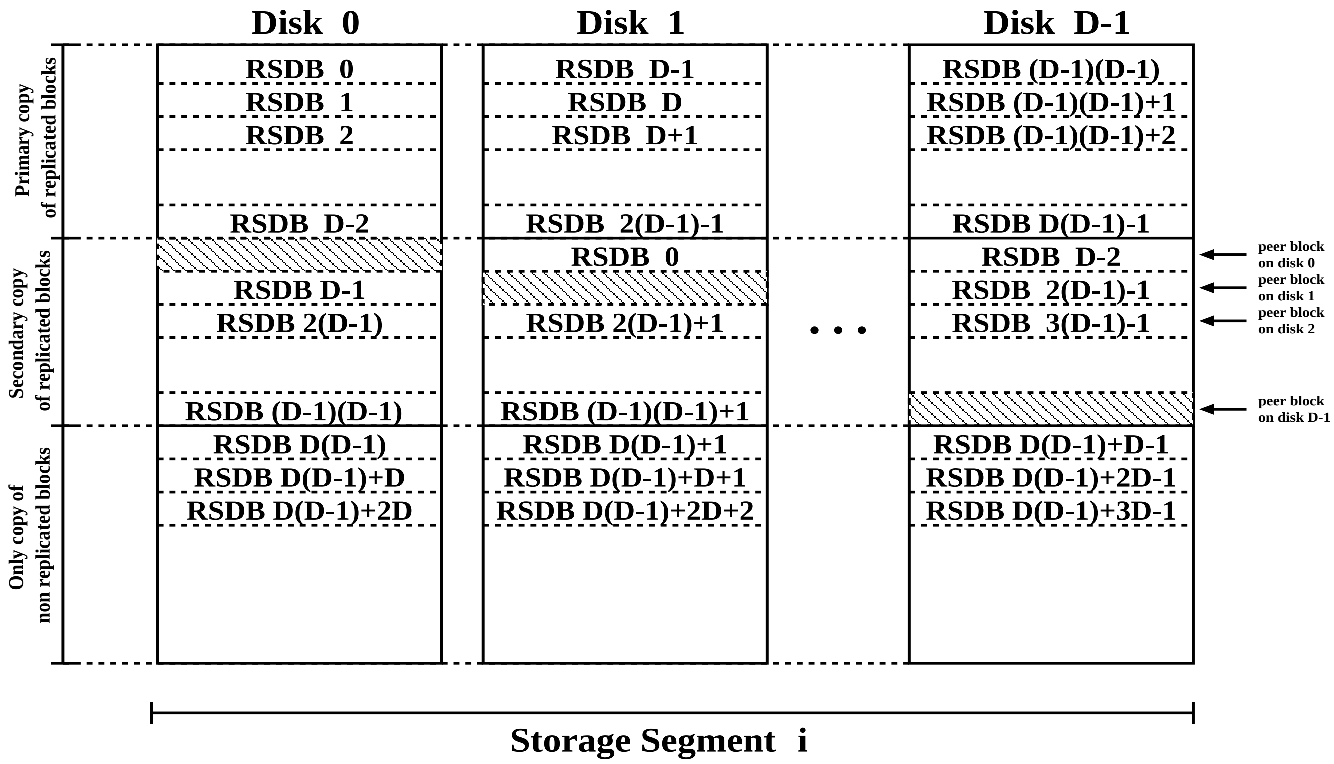


Figure 6: Mapping system blocks to disk blocks

With this disk layout, a given SDB block is mapped to a SS segment and a relative system data block ($RSDB$) inside a Storage Segment as follows:

$$SS = \left\lfloor \frac{SDB}{n_{SS}} \right\rfloor$$

$$RSDB = SDB \textbf{ mod } n_{SS}$$

Figure 6 shows the system block that is stored on each disk block. Note that replicated system blocks are stored on two disks and non replicated blocks on just one. The shaded areas on the figure do not correspond to real disk blocks and were introduced on the figure just to make it clearer.

If $RSDB < D(D-1)$ the block is replicated and its primary copy is stored on the relative disk data block $RDDB_p$ of disk d_p:

$$d_p = \left\lfloor \frac{RSDB}{D-1} \right\rfloor$$

$$RDDB_p = RSDB - d_p(D-1)$$

The secondary copy of the data block is stored on the relative $RDDB_s$ block of disk d_s:

$$d_s = \begin{cases} RDDB_p & \text{if } RDDB_p < d_p \\ RDDB_p + 1 & \text{if } RDDB_p \geq d_p \end{cases}$$

$$RDDB_s = \begin{cases} d_p + D - 1 & \text{if } d_s > d_p \\ d_p + D - 2 & \text{if } d_s < d_p \end{cases} .$$

If however $RSDB \geq D(D-1)$ the block is non replicated and is stored on the relative block $RDDB_n$ of disk d_n:

$$d_n = (RSDB - D(D-1)) \ \mathbf{mod} \ \ D$$

$$RDDB_n = 2(D-1) + \left\lfloor \frac{RSDB - D(D-1)}{D} \right\rfloor$$

Once the relative disk block $RDDB$ is known for any of the 3 cases, $RDDB_p$, $RDDB_s$, and $RDDB_n$ the absolute DDB on the disk can be easily computed as

$$DDB = RDDB + SS * n_{SS}$$

4.4 Block allocation

When an object block needs to be allocated to an object, a random SDB block is chosen. If this block is free it is allocated to the object. If not, consecutive blocks after it are checked until a free block is found and allocated to the object. This block allocation scheme guarantees that the physical location of any 2 blocks of an object are independent of each other, making the physical access pattern to object blocks random, independent of the logical access pattern.

4.5 Storage Subsystem Software Architecture

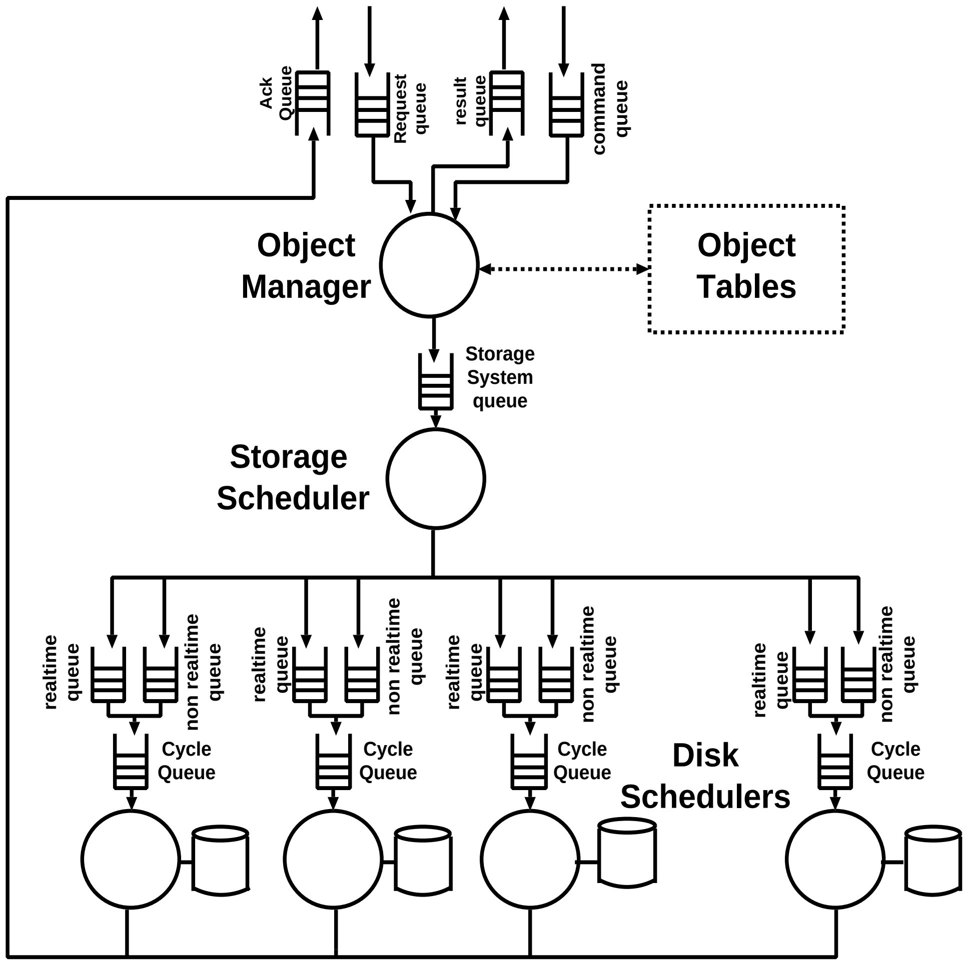


Figure 7: Storage Systems Software Architecture

Figure 7 shows a simplified description of the storage system architecture. The storage system is composed by 3 major components: object manager, storage scheduler and disk schedulers.

The object manager is responsible for managing data at the object level. It provides an interface to the upper layers of the system that allow them to manipulate data objects. The most important functions provided are: object creation and destruction , object open and close, realtime read and write, non realtime read and write, referencing and dereferencing objects from other objects, allocating and deallocating space to objects, etc. Functions are invoked through commands submitted to a command queue stored in shared memory. Command results are returned to the higher level on a result queue. Read and write commands are treated as special commands with higher priority and have an independent pair of queues (request and ack queues). When a data request arrives the object manager maps the logical data of the object to a system level data block (*SDB*) request, using the information stored on its internal tables, and send the *SDB* request to the storage scheduler. Note that due to indirect indexing of *SDB* blocks, to obtain the *SDB* index of a given object block (*ODB*) we may need to read 1 or 2 additional blocks. To avoid the overhead of multiple I/O requests to fetch data blocks containing block indexes, which would have a serious impact on the delay of realtime requests, we decided to keep all block indexes in memory if the object is open for realtime access. The block indexes are read when the object is open and are released when it is closed. This guarantees that each realtime block request will require just one I/O request. The amount of memory used to store the indexes is not very large since block sizes are relatively high (16Kbytes to 256Kbytes) and each index in memory uses only 4 bytes per block. If for example blocks of 128 Kbytes are used, we need 32 Kbytes of memory to store indexes for each Gbyte of data. In our prototype the total disk space is 56 Gbytes which would require only 1.75 Mbytes of memory for data indexes, in the extreme case where all the space were used (without replication) and all objects were opened for realtime access.

The storage scheduler basically maps a system *SDB* block to the correct disk and disk block (*DDB*) on that disk. If the request is a read to a replicated block the load balancing algorithm is applied to select the less loaded disk for the request. On the other hand if the request is a write to a replicated block the request is submitted to both disks storing the data block. Non replicated blocks are sent to the only disk that stores it. Depending if the request is realtime or non realtime the request is placed on the corresponding queue of the disk scheduler. The load balancing use the current size of the queues (sum of the realtime, non real time and cycle queues sizes) on each of the 2 possible disks holding the data block and sends the request to the one with smaller number of requests at the moment. In case the queues have the same size the selection is made at random.

The disk schedulers process I/O requests to a particular disk. Every time a cycle finishes, the disk scheduler grabs requests from its real time queue up to the maximum number of requests per cycle. If the number of realtime requests is smaller than the maximum cycle time, the non realtime queue is checked and additional requests are grabbed from there, up to the maximum cycle size. All selected requests are then ordered on the cycle queue according to their relative position on the disk surface, from the inner most cylinder to the outter most one or vice-versa, depending on the current position of the disk head. The requests are then serviced in that order. When the I/O for a given request is completed, the request is removed from the queue and an I/O completion notification is sent to the higher level through the ack queue.

5 Performance experiments

5.1 Cycle Time Distribution

The duration of a disk cycle is given by the sum of the I/O time of each request served in that cycle. It is well known that the time to read a data block is not simply a function of the block size, but depends also on other random components such as the relative position of the disk head with respect to the first byte of the requested block, the disk transfer rate (which on current commercial disks that implement disk zoning decreases from the outer most tracks to the inner most ones), as well as other minor factors. We analyze the cycle time as a random variable which is the sum of the random variables associated with the individual I/O time of requests in the cycle, using performance parameters for the disks used in our prototype (SEAGATE Barracuda ST15150W) (Our prototype is configured with two disks per SCSI string) .

The time to read one block is obtained as the sum of the following random variables: seek time, rotational latency, disk transfer time, track switch time, SCSI data transfer time , system call and SCSI command overhead, and SCSI bus contention among multiple disks. Most of this parameters were obtained experimentally or were computed from other experimental parameters since the vendor's manual does not provide all the information necessary. [22] [26]

The disk layout, including number of zones, number of heads, sector size, number of cylinders per zone, track size per zone, and track skew factor per zone, are obtained from information on the disk(s) using SCSI commands requests.

For simplicity, the seek time is obtained by assuming the worst case cycle in which the number of cylinders between the first block and the last block requested in the cycle is maximum and the other requested blocks in the cycle are equally spaced on the disk cylinders. Then the seek time is computed using the seek time function of the disk that was obtained experimentally. The rotational latency is computed as a uniform distributed random variable in the range of 0 to the disk revolution time. The disk transfer time is computed using the disk transfer rate for each zone, the block size and the size of each zone (which defines the probability of a block be located at a specific zone). The track switch time can be obtained by the distribution of the number of track boundaries crossed by a data block, the track skew factor and the disk rotation speed. System call and SCSI command overheads are assumed to be constant and are obtained by measuring the I/O time to read data stored in the disk cache memory for different sizes of data and eliminating the linear component that is due to SCSI data transfer time. The SCSI data transfer time is difficult to compute when data is read from the disk, without knowing the details of the disk controller operation, since SCSI data transfer overlaps disk transfers (when large blocks are accessed).We computed the additional overhead due to SCSI data transfer time, by measuring the I/O time for requests for a given block size and discounting the other components which had been previously computed. All the previous I/O times were obtained from experiments with only one disk on the SCSI bus being accessed. Since in our prototype there are two disks on each SCSI bus, there is an additional overhead due to contention on the SCSI bus. We estimated the distribution of the time overhead due to SCSI bus contention by measuring the request I/O time using just one and then using the two disks on the SCSI bus, and computing the difference of the observed means and variances in the two cases.

After computing the request time distribution the cycle time distribution was computed numerically as the sum of i.i.d (independent identically distributed) random variables, since the location of any two blocks are independent of each other, because of our random allocation scheme.

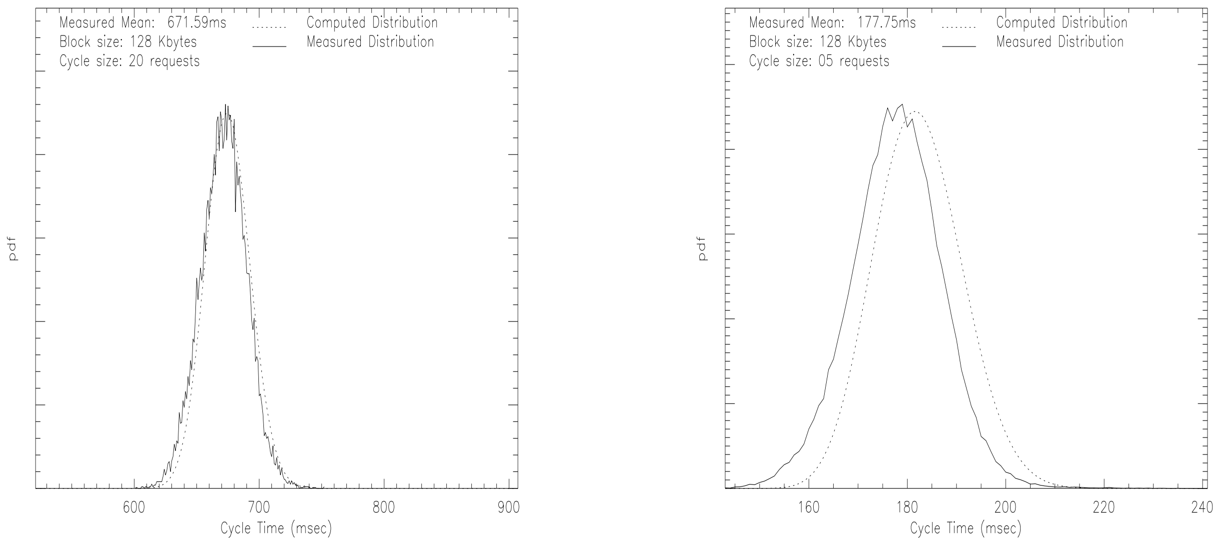


Figure 8: Cycle time distribution

Figure 8 shows the computed and measured cycle time distribution for 128 Kbytes requests and cycles with 20 and 5 requests, for the disks used in our prototype. The computed distribution is translated slightly to the right and represents a random variable which is stochastically greater than the actual cycle time. This occurs mostly because in our model we assumed the worst case seek time, i.e. the seek distance between the first and last request in a cycle was assumed to be the total number of cylinders in the disk. This error is relatively higher on smaller cycles, since in this case the difference of the expected seek distance and the maximum is higher. However the error due to this assumption is small and conservative which is what we need.

5.2 Minimum deadline "guaranteed" by the system

The probability of a request missing its deadline can be decomposed in two components: one due to *successful* requests and one due to *delayed* requests

$$Prob[delay > t] = Prob[successful]Prob[delay > t \mid successful] + Prob[delayed]Prob[delay > t \mid delayed]$$

We want to have small probabilities for these two components, i.e. $Prob[delay > t \mid successful] \leq P_{miss}$ and $Prob[delay > t \mid delayed] \leq P_{miss}$, for an arbitrary small probability P_{miss}, such that $Prob[delay > t] \leq P_{miss}$. In this section we discuss the case of *successful* requests an let the case for *delayed* requests to be discussed later when we present our experimental results.

We define the minimum deadline that the system can "guarantee" as the maximum delay that a *successful* request can have. As we mentioned before, the maximum delay of a *successful* request is given by the duration of two consecutive cycles ($T_i + T_{i+1}$). A simple way of computing this maximum delay is to consider two consecutive cycles and assume the absolute worst case I/O time for each request in these two cycles. This would give us a very conservative delay bound, since the probability of having the worst case for each request is very small and will seldom ever happen in practice. A tighter bound is desirable to reduce buffering in other parts of the system for example. Our approach is to offer a statistical guarantee of a much smaller delay bound with a probability close to one.

Given a small probability P_{miss} of missing the deadline we define the (stochastically) "maximum" delay of a successful request D_s as:

$$D_{s_{max}} = \{t \mid Prob(D_s > t) = P_{miss}\}$$

We then compute an upper bound for $D_{s_{max}}$ as follows.

The duration of two consecutive cycles T_i and T_{i+1} depends on the number of requests served in these cycles, but the sum is stochastically less than the sum when the two consecutive cycles have the maximum number of requests n_{max}:

$$T_i + T_{i+1} \quad \leq_{st} \quad \{T_i + T_{i+1} | n_i = n_{max}, n_{i+1} = n_{max}\}$$

where:

$x \leq_{st} y$: x is stochastically smaller than y, i.e. $Prob(x > a) \leq Prob(y > a)$, for all a.

n_i : Number of requests in cycle i.

Because of our random allocation scheme, the location of blocks requested in cycle $i + 1$ is independent of the location of blocks requested in cycle i, and thus the duration of two consecutive cycles, given the two cycles have the maximum number of requests, are independent random variables. Therefore the distribution of the total time of two consecutive cycles with the maximum number of requests $\{T_i + T_{i+1} | n_i = n_{max}, n_{i+1} = n_{max}\}$ can be obtained by convolving the distribution of the cycle time $\{T | n = n_{max}\}$ with itself, where the cycle time distribution is computed as described in section 5.1 and illustrated in figure 8.

We then define the minimum delay bound "guaranteed" by the system as

$$D_{min} = \{t | Prob(\{T_i + T_{i+1} > t | n_i = n_{max}, n_{i+1} = n_{max}\}) = P_{miss}\}$$

D_{min} is an upper bound for the maximum delay $D_{s_{max}}$ of a *successful* request for a given probability of missing the deadline P_{miss}, since:

$$Prob(D_s > D_{min}) \leq Prob(T_i + T_{i+1} > D_{min})\} \leq$$

$$Prob(\{T_i + T_{i+1} > D_{min} | n_i = n_{max}, n_{i+1} = n_{max}\})\} = P_{miss} = Prob(D_s > D_{max})$$

We used the above methodology to compute the minimum delay bound "guaranteed" by the system for the disks used in our prototype. Unless otherwise stated we consistently use $P_{miss} = 10^{-6}$.

When computing the cycle time distribution we ignored one delay component that may happen due to disk thermal re-calibration. Disk heads need to be re-calibrated periodically due to thermal variations. During thermal re-calibration data access is delayed until the calibration is done. The interval between 2 consecutive thermal calibrations for our disks is approximately 86 seconds and since request delay is of the order of hundreds of milliseconds to seconds, each request will be delayed by at most one thermal calibration. The fraction of requests affected by re-calibration is much larger than 10^{-6} and can not be ignored with respect to the delay bound; however, it does not have a significant effect on throughput.To account for possible thermal calibrations we added to D_{min} computed as described before an additional worst case thermal calibration time $Tcal_{max}$ as specified in the disk manual (for our disks $Tcal_{max} = 50msec.$).

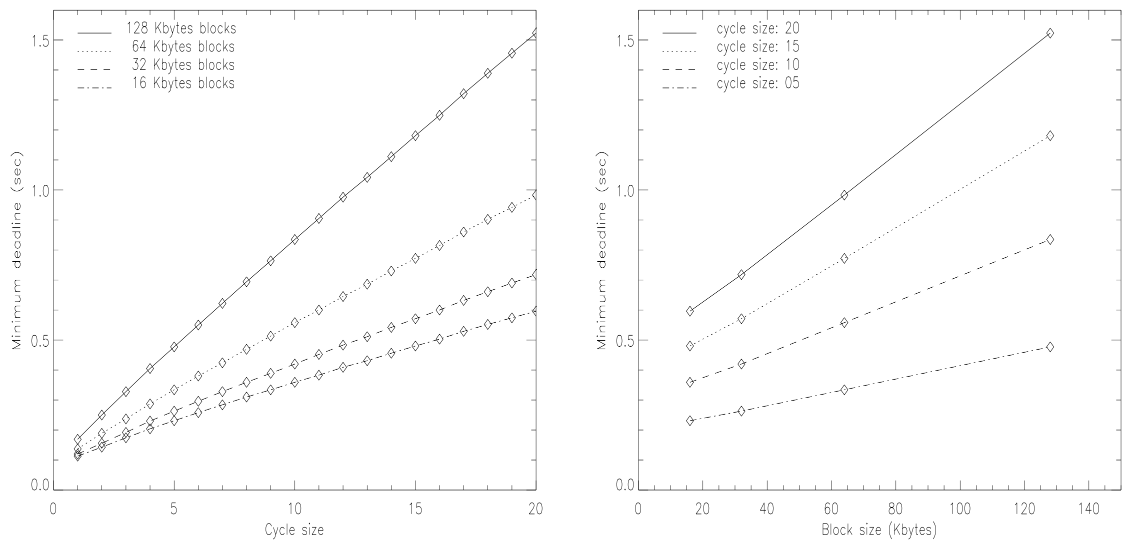


Figure 9: Minimum deadline guarantee

Figure 9 shows the minimum delay bound that can be "guaranteed" by the system, for different values of block and cycle sizes, for the disks used on our prototype. We observe that the minimum delay bound increases approximately linearly with the cycle and block size. This behavior is explained as follows.

a) Minimum delay bound as function of block size

The average I/O time for a single request increases linearly with the block size since it is given by the average disk transfer rate times the amount of data accessed plus an overhead time which is approximately constant. The variance of the request I/O time is due to two basic components: the variation of the disk transfer rate due to different disk zones and the variation of the I/O overhead time (rotational latency, seek time, etc). Since these components are independent random variables, the variance of the service time of a single request is given by the sum of two variances (one for each component). The variance of the data transfer time is proportional to the square of the block size (standard deviation is a linear function of the data transfer time) while the variance of the I/O overhead time is approximately constant for different block sizes. Therefore the standard deviation of the request I/O time increases linearly with the block size, and thus so does the cycle time standard deviation. Therefore, the minimum deadline, which we expect to be closely related to an offset from the mean cycle time proportional to the standard deviation, should increase linearly with the block size.

b) Minimum delay bound as function of cycle size

The average cycle time is proportional to the number of requests in a cycle. However the standard deviation increases proportionally to the square root of the number of requests in a cycle, since the service time of each request is an independent random variable. However the standard deviation is small when compared to the mean cycle time and the linear behavior of the mean cycle time dominates the deadline and this explains the almost linear results in figure 9 (e.g. for 128Kbytes blocks the standard deviation varies from 5.2% to 2.7 % of the mean when the cycle size varies from 5 to 20).

5.3 Effective disk bandwidth

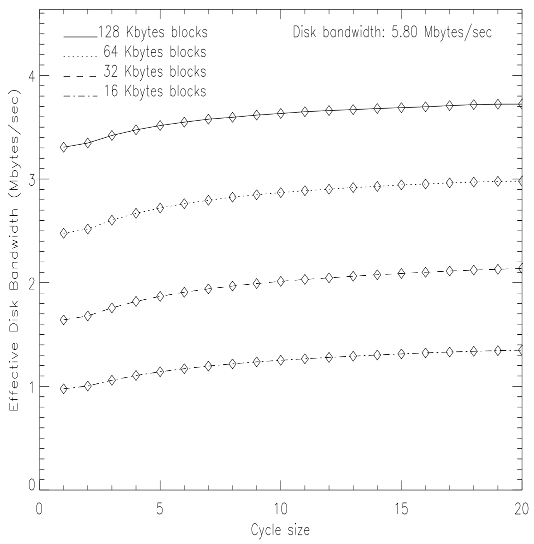


Figure 10: Disk effective bandwidth

We compare the effective bandwidth per disk for different block sizes and maximum number of requests per cycle (cycle size) . Clearly the effective disk bandwidth increases with the block size and the maximum number of requests per cycle. A higher block size amortizes the rotational latency and overhead times over a larger effective data transfer time and a higher number of requests per cycle amortizes the seek time over a larger number of requests. However increasing the block size increases the size of the buffers required for each stream which has an associated cost. Also, having larger block sizes increases the probability of reading superfluous data for certain applications and increasing the number of requests per cycle or the block size increases the minimum delay bound that can be "guaranteed" with a fixed probability. Therefore, there are multiple tradeoffs when selecting the block and cycle sizes, which have to be kept in mind. However in this section we concentrate only on how the bandwidth of the system varies as we vary these parameters.

The effective disk bandwidth for a particular block and cycle size is easily computed as the ratio of the total amount of data read and the average duration of a cycle. Figure 10 shows the effective disk bandwidth achieved by different combinations of block and cycle sizes as measured on our system prototype.

As expected, the performance improves when either the block size or cycle size increase. We observe in the figure that bandwidth is more sensitive to block size than cycle size. This is due to the fact that the seek time is a small fraction of the total I/O overhead.

Observe that these results apply more widely than our random data allocation approach. Any system that schedules data requests using disk cycles, will be bound by the same maximum performance as a function of the block and cycle sizes. These curves are the same for random data allocation or data stripping. In our case, where random data allocation is used, the maximum allowed average bandwidth per disk (which is enforced by admission control) will be a fraction of the maximum effective disk bandwidth shown in figure 10, since we have an additional constraint on the delay experienced by requests. To meet this requirement we have to constrain the traffic presented to the storage system to something less than the maximum values that appear in figure 10. As we discuss in section 7, data striping with streaming RAID [24] also has to operate at bandwidths smaller than those plotted in figure 10, but we will address this later when we compare the performance of the two approaches.

5.4 Probability of missing the deadline

As discussed before, the guaranteed delay bound is selected such that we can guarantee that the probability of any *successful* request missing its deadline is smaller than an arbitrary small probability P_{miss}. However, there is still the possibility of having a large fraction of the request not being *successful*, i.e., having more requests to serve on a given disk cycle than the maximum cycle size. These *delayed* requests are potential candidates for missing their deadlines. Our goal is then to reduce this probability.

Of course, as we increase the total load on the system, we expect the probability of having *delayed* requests to increase, since we are increasing the average load on each disk, and thus increasing the average disk queue size.

Another factor that determines the probability of having overflow cycles (i.e. cycles in which the disk queue can not be emptied) is the level of load balance among the system disks. Due to the randomness of the location of data blocks of requests, the instantaneous load on each disk will fluctuate around its mean value. The higher this fluctuation the higher the probability of having an instantaneous high load on a single disk and thus the higher the probability of having an overflow cycle (i.e.. having a *delayed* request), for the same average workload.

In order to analyze the performance of our system in detail we executed a series of experiments on our prototype. During each experiment the system was driven by a traffic simulator, generating a sequence of read requests to constant size blocks randomly selected, for a period of 30 minutes. The simulator generates a fixed number of requests in successive intervals of constant duration T, the mean time of a cycle with maximum number of requests, with the requests arrival times randomly distributed over the interval. This distribution of the request arrival process was chosen, because in the VWDS the mediator guarantees that in any interval T there is a maximum number of requests. Although not explicitly shown here, we did some experiments with different intervals T and other arrival process statistics (Poisson and constant rate) and the results obtained were indistinguishable from each other.

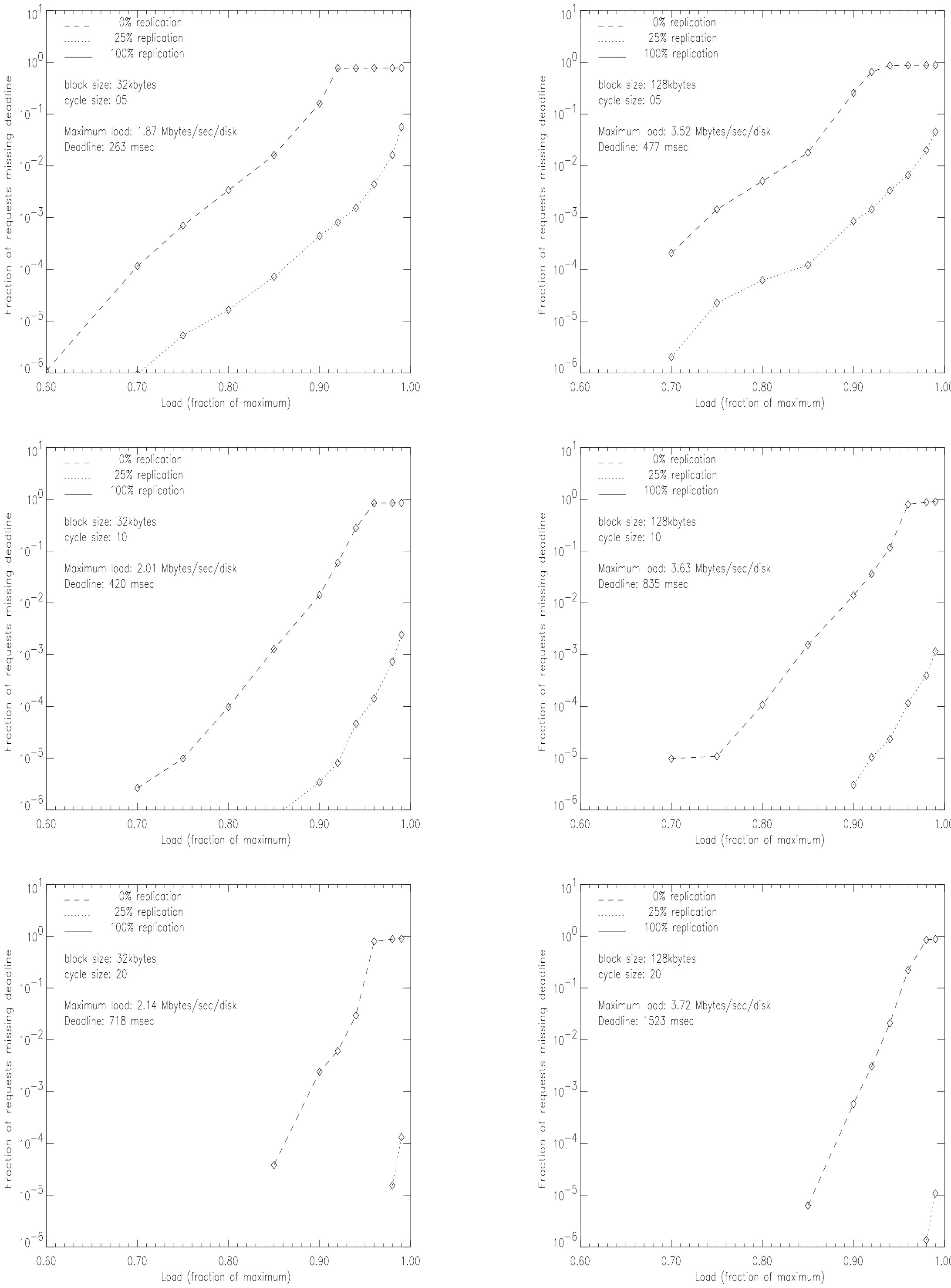


Figure 11: Probability of a request missing the deadline

In all experiments we assumed that the "guaranteed" delay bound was the same for all requests and equal to the minimum deadline that could be "guaranteed" by the system (This value is different for each block size and cycle size selected). The experiments were carried out on a configuration with 14 disks.

Figure 11 shows the measured frequency of requests missing their deadlines as function of the relative load. The relative load is defined as the fraction of the maximum effective disk bandwidth that can be achieved for the selected block and cycle sizes, and thus is different for each experiment. Results for the 100% replication case are not shown in the figures because in this case the fraction of requests missing the deadlines is smaller than 10^{-6} for the entire range of loads from 0% to 99% of the effective disk bandwidth and could not be measured. (The duration of experiments did not allow measurements of frequencies smaller than 10^{-6}). This demonstrates the startling effectiveness of short term load balancing via replication.

In all experiments, no successful request missed its deadline and the measured fraction of requests missing the deadline are all due to *delayed* requests, confirming our prediction that the probability of a successful request missing the deadline is less than 10^{-6}.

The results show that increasing the level of replication reduces the probability of missing the deadline, due to enhanced short term balancing of disk queues. We also observe that the relative performance for 32 Kbytes and 128 Kbytes are very similar, although the absolute performance is different due to different effective disk bandwidths for different block sizes. We also observe that increasing the cycle size, reduces the probability of missing the "guaranteed" deadline. The main reason is that increasing the cycle time causes a significant increase on the minimum "guaranteed" deadline used in the experiments. Regardless of n_{max} the number of requests served per cycle tends to be determined by the traffic rate (see discussion on distribution of cycle sizes below). Therefore the distribution of delay for a request is not so sensitive to n_{max}, but the delay bound does increase with n_{max}.

The results show that small probabilities of missing the deadline ($< 10^{-6}$) can be achieved with loads in the range of 70% to 95% of the maximum load supported by the disk with only 25% of the data replicated; or with loads in the order of 99% when all data is replicated. Even with no replication loads in the range of 60% to 80% can be achieved. (for deadlines in the range of 0.3 sec to 1.5 sec.).

The effect of replication can be observed in figure 12 that shows the distribution of cycle sizes for 128 Kbytes blocks, maximum cycle size 5 and 20 and relative load of 80% and 96% (Note that the scale in the graphs is logarithmic what explains why the areas under the curves do not look the same).

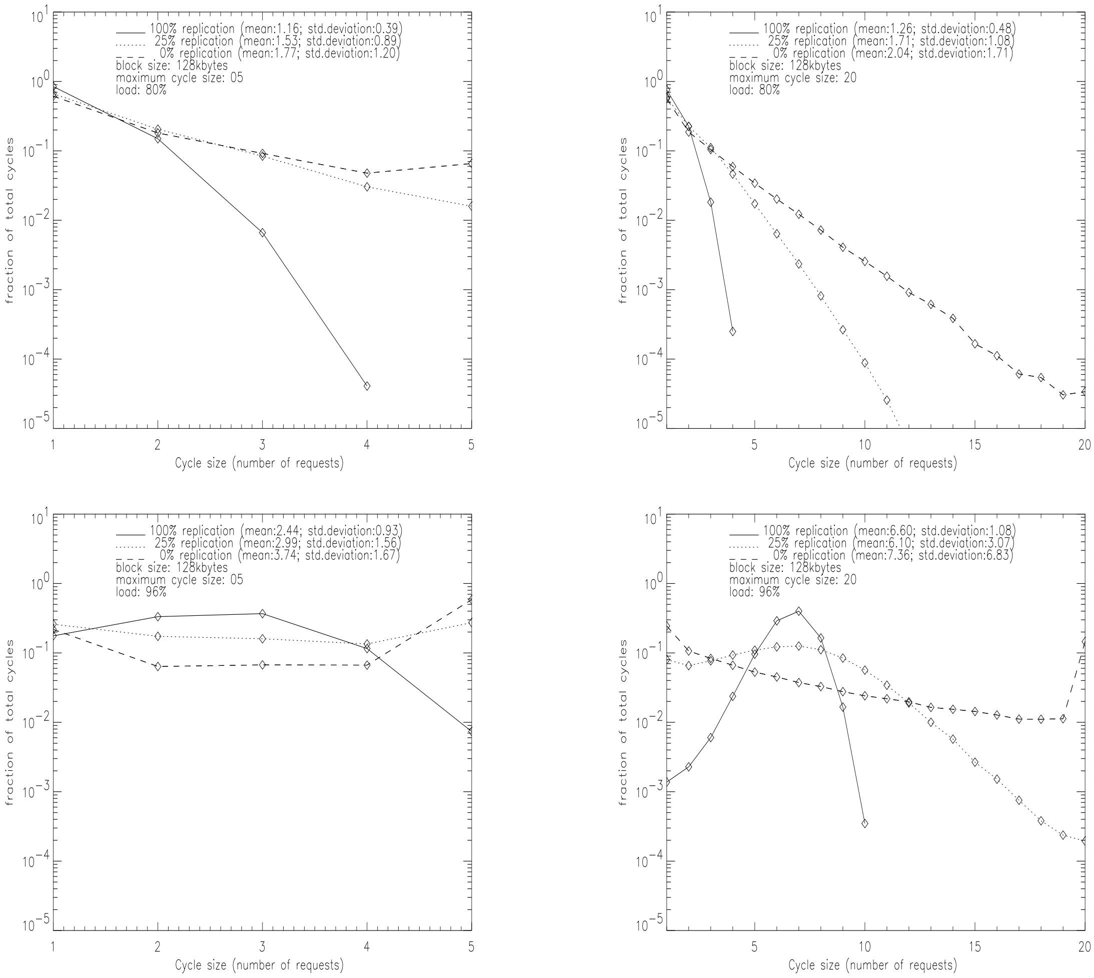


Figure 12: Cycle size distribution

There are several interesting points illustrated in figure 12. First, we observe that increasing the level of replication, reduces the variance of the cycle size distribution due to better load balancing. This reduces the probability of having cycles of the maximum size and the probability of having overflow cycles and thus reduce the probability of missing the deadline, as observed in our previous results.

Figure 12 also shows that the mean cycle time is often much smaller than the maximum cycle size. This occurs because of the asynchronous nature of cycles. The extra disk bandwidth available (which otherwise would generate idle time) is absorbed by having more frequent but smaller cycles (The disk tends to be as inefficient as it can for a given load, to absorb the extra bandwidth). This effect is more pronounced in the case of maximum cycle with 20 requests, where even the maximum observed cycle time is smaller than the nominal maximum cycle. Even a load slightly smaller than the maximum as the case of 96% load reduces the mean cycle size by approximately three times. This is due to the fact that reducing the cycle size just slightly reduces I/O performance as previously shown in figure 10.

This effect can be better illustrated with an example. Consider the case of 128 Kbytes blocks, maximum cycle with 20 requests and 96% load. From figure 10 the effective disk bandwidth in this case is 3.72 Mbytes/sec and thus a 96% load is equal to 3.57 Mbytes/sec/disk. Again from figure 10 we observe that an effective disk bandwidth approximately equal to 3.57 Mbytes/sec is achieved by a cycle with 7 requests, which not surprisingly is approximately the mean cycle time observed in figure 12. Thus the disk is operating with less efficient cycles and very little idle time in this case. This absorption of excess bandwidth has however a limit, since we can not decrease the cycle size to a value smaller than 1. When we reach this limit, most cycles will have size 1, and the disk will start having more frequent idle periods, as is observed in the case of 80% load in figure 12.

5.5 Maximum Load

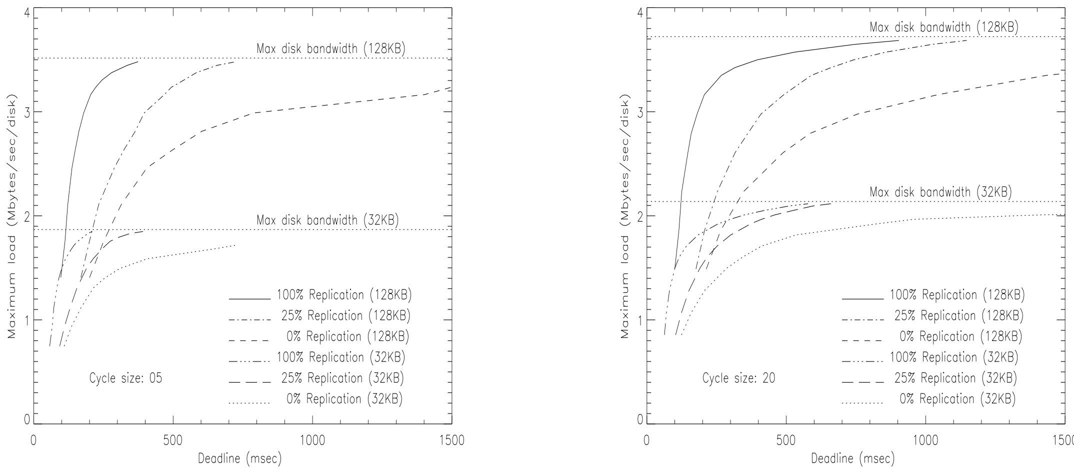


Figure 13: Maximum load as function of required deadline

The last section showed how the probability of missing the deadline varies as a function of the relative load, level of replication, block size and cycle size. However, when designing a system we would like to know what combination of block and cycle sizes would give us the best disk bandwidth utilization for a specified deadline. We measured the distribution of request delays and determined for each block size and cycle size what deadline guarantee can be made with a desired probability of missing the deadline, which we again set to be 10^{-6}. Using these results we then plotted in figure 13 the absolute load that the system could support for each desired deadline.

We observe that larger block sizes provide better disk bandwidth utilization simply because they generate more efficient disk accesses. As already mentioned, higher levels of replication improve the performance of the system due to better load balancing. We also observe that large and small maximum cycle size provide almost the same performance for small values of deadline, while large cycles allows the system to achieve slightly higher loads when longer deadlines are used, due to the slightly increased effective disk bandwidth.

6 Simulation Results

In the last section we showed measurement results obtained from our prototype. In order to evaluate the performance of our storage system for a larger number of disks than are currently available in our prototype we developed a simulator. In this section we show performance results obtained using the simulator.

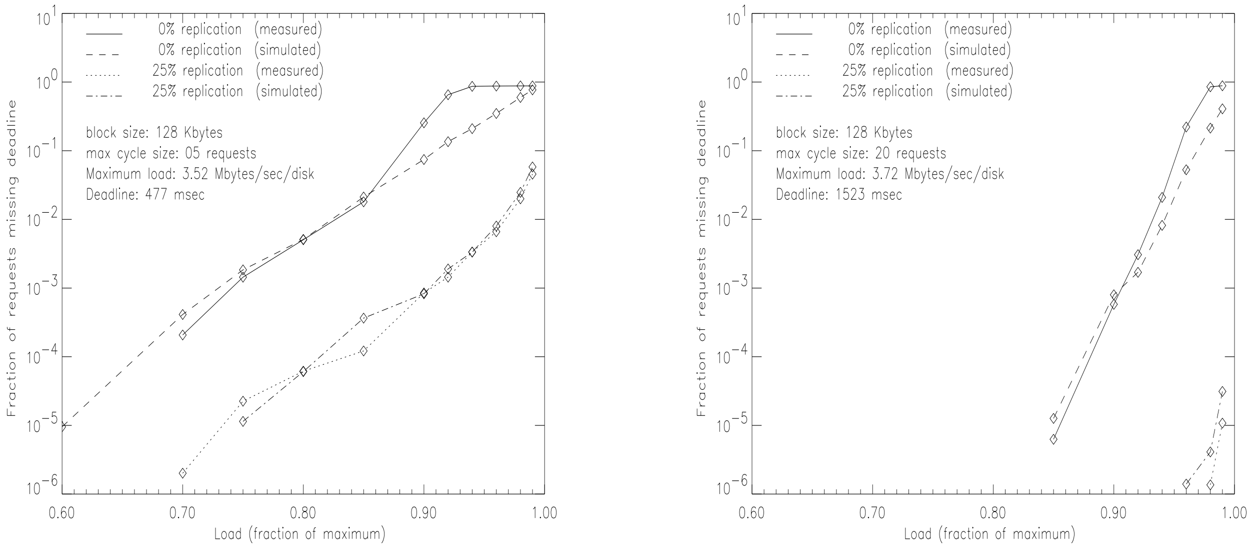
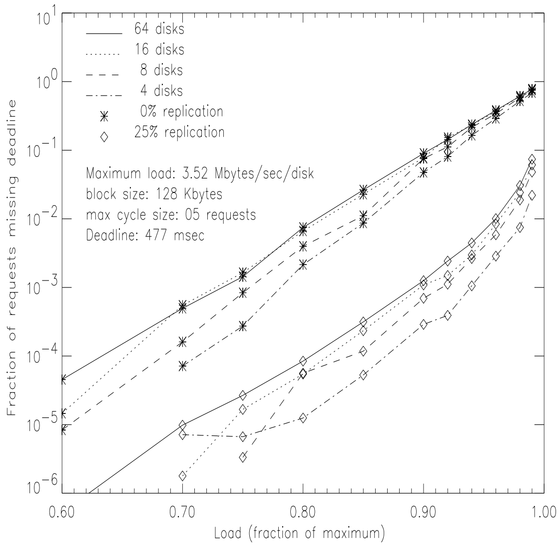


Figure 14: Simulation X real system

In order to validate our simulator we compare the results generated by simulation with the experimental results obtained in the last section. We observe that the results of the simulation are very close to the measured results as shown in figure 14, confirming that our simulation is modeling the system performance and disk behavior effectively.

6.1 Effect of the number of disks



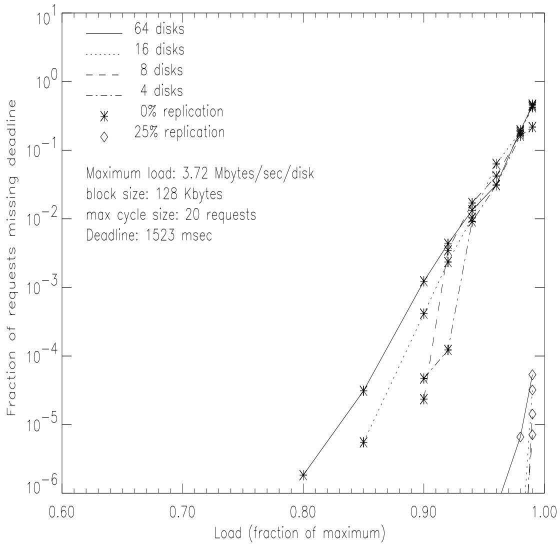


Figure 15: Effect of the number of disks

Figure 15 shows the fraction of requests that miss the deadline for different number of disks, with block size 128 Kbytes and various fractions of replication and cycle size. We observe that the system performance is approximately the same over a wide range of the number of disks given the same relative load and replication level. However, we observe that when the number of disks is relatively small, the probability of missing the deadline is slightly smaller than for larger number of disks. This can be explained as follows. The traffic generator generates a fixed number of requests in a specified time interval T, that is proportional to the number of disks of the system, thus generating more requests for systems with higher number of disks, for the same relative load. The total number of requests in this interval, for a system with D disks, is denoted by γD, where γ is a constant proportional to the relative load, effective disk bandwidth and interval duration. Consider a system with no replication. Then, the number of requests N_n in an interval that are directed to a particular disk is a random variable with a Bernoulli distribution with number of trials $n = \gamma D$, probability of success $p = 1/D$ and probability of failure $q = (D-1)/D$. The mean number of successes of a Bernoulli process with n trials is given by $E[N_n] = np$ and the variance is given by $Var(N_n) = npq$. Therefore the mean and variance of the number of requests directed to a particular disk in an interval is given by $E[N_n] = \gamma$ and $Var(N_n) = \gamma(D-1)/D$, respectively. We conclude that the variance on the number of requests in the disk queue is smaller when the number of disks is smaller, reducing the probability of having overflow cycles and thus reducing the probability of missing deadlines, explaining the results of figure 15. This also explains the observation that as the number of disks become larger, the variation of the system performance becomes less sensitive to the number of disks, since the variance of the number of requests directed to a disk approaches γ. These comments are also valid when replication is used, except that the variance in all cases is reduced due to load balancing.

6.2 Fault tolerance

The use of partial replication as described in the last sections is an effective method for providing short term load balance among the parallel disks. However in order to provide tolerance to disk failures we need to consider alternative schemes. There are basically two approaches; one is to use full replication and the other is to use parity groups as used on RAID schemes[20].

One possibility is to use RAID subsystems to provide fault tolerance and partial replication of data blocks to provide load balance. In this approach, we divide the total number D of disks in the system on N_A disk arrays with $D_A = D/N_A$ disks each. Each disk array is a RAID level 5 [20] with parity group declustering [?] that stores data on parity groups of size G distributed on G different disks ($G-1$ data blocks and 1 parity block; $G \leq D_A$). A fraction α of the data blocks is then randomly replicated across all disks of the system. Replicating data blocks across all disks, instead of replicating them only within the same disk array allows balancing the load among disks in the disk array as well as across disk arrays. Parity blocks are not replicated, since they are only used during disk failure (for data reading), while data blocks are used in all situations. The ratio of useful data space and total storage space for this scheme is given by:

$$U = \frac{1}{1 + \alpha + \frac{1}{G-1}}$$

Observe that this scheme has performance identical to the case that does not use RAID, when the system is operating in normal conditions (without failure) and with only reading operations, since parity groups are only used if a disk fails. Later, after we introduce alternative schemes we analyze performance when a disk fails.

Another scheme is to use 100% replication for both load balancing and fault tolerance purposes. Since all data blocks are replicated any single disk failure can be supported by this scheme. However, using random replication across all disks does not support simultaneous failures of any two disks, since every pair of disk contains some data that is stored only on that disk pair. Imposing constraints on where data can be replicated improves fault tolerance by allowing the system to survive multiple disk failures. However these same constraints can reduce the capacity for short term load balancing.

One such replication scheme is chained declustering [13], where data whose primary copy is stored in a given disk i is replicated only in the next logically consecutive disk $i+1$. This scheme allows multiple disk failure, provided only that no two failures occur on logically consecutive disks.

Chained declustering pays for increased fault tolerance with less flexible short term load balancing as our simulation results will show. A generalization of chained declustering, which we call overlapping clusters, allows the system designer to trade off fault tolerance and realtime performance. In the overlapping clusters scheme disks are divided into overlapping clusters of size C and offset by C/2 with respect to the neighboring clusters as illustrated on figure 16. The primary and secondary copy of a replicated data block is then constrained to be located in one cluster that is associated with the data block (note that each disk belongs to two clusters and data stored on it can be associated to any one of the two clusters) This scheme allows multiple disk failures provided the disks do not belong to a same cluster. This scheme lies between random replication and chained declustering since it supports more disk failures than random replication and less than chained declustering, while providing better capability of short term load balancing than chained declustering but not as good as random replication.

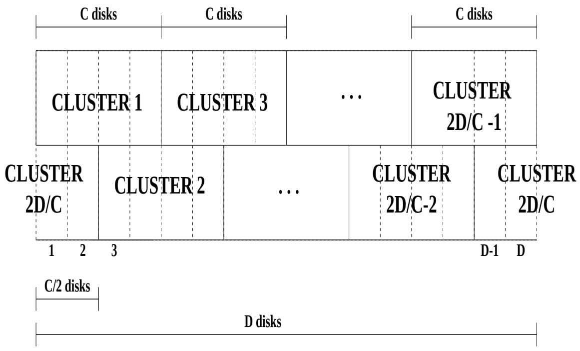


Figure 16: Replication using overlapping clusters

Simulation results for a system configuration of 64 disks are shown in figure 17. The fraction of requests that miss the deadline are shown for different replication schemes: RAID with partial random replication

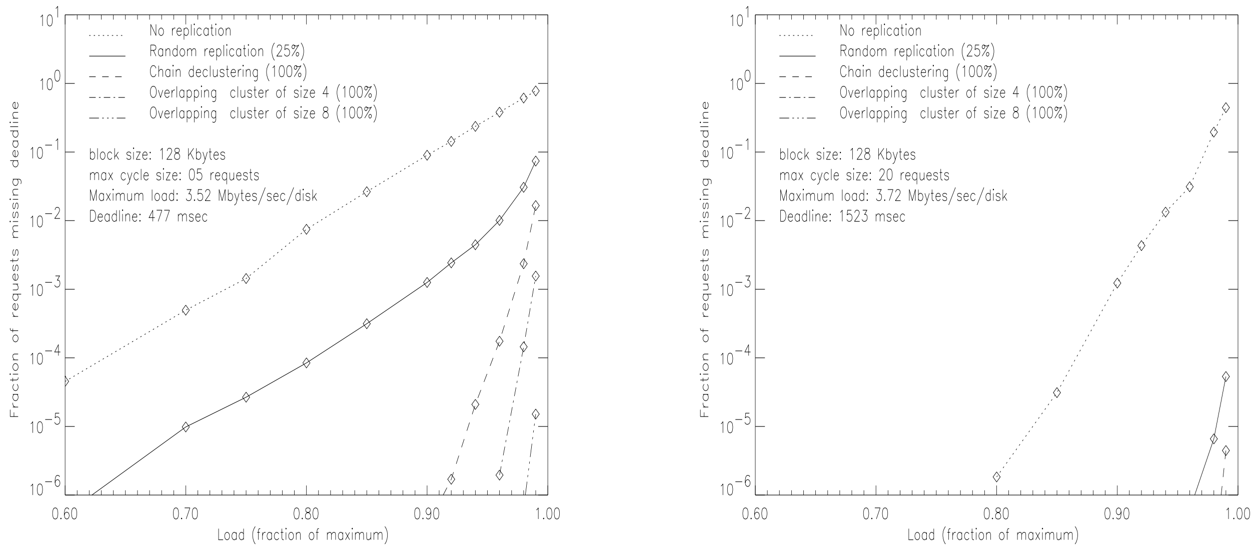


Figure 17: Performance without failure

(25%), chained declustering, and overlapping clusters. The curves not drawn in the figure, correspond to cases where the fraction of requests missing the deadline is less than 10^{-6} for all loads up to 99%. Clearly, full replication provides better results than partial replication (25%), even when full replication is subject to constrained data layout such as in chained declustering and overlapping clusters. We also observe that overlapping clusters provides better performance than chained declustering and increasing the size of the overlapping clusters improves the performance, as expected due to additional flexibility in load balancing.

It is also interesting to study the effectiveness of these fault tolerant schemes in the situation where there is a disk failure. Performance results for a system with 64 disks and one disk failure are shown in figure 18.

These results assume that, when disk arrays are used, data is requested in terms of individual data blocks, and not in terms of entire parity groups.[1] In this situation, a disk failure causes an increase in the system load, because unavailable data blocks need to be reconstructed by reading the other G-1 blocks of its parity group that, in general, would not be read if the disk had not failed. For all results in figure 18 we assumed a parity group of size $G = 5$ (4 data blocks and 1 parity block), and thus a overhead of storage capacity of 50% (25% due to parity and 25% due to replication). The additional load causes the system to saturate with a lower nominal load for the RAID based schemes than for the others. The excess load is directed

[1] For video applications it is more reasonable to read entire parity groups (or at least the data blocks in a parity group) since access is highly predictable and sequential. For other types of applications, e.g. 3D interactive virtual world, larger granularity of reads can imply large amounts of superfluous data being read.

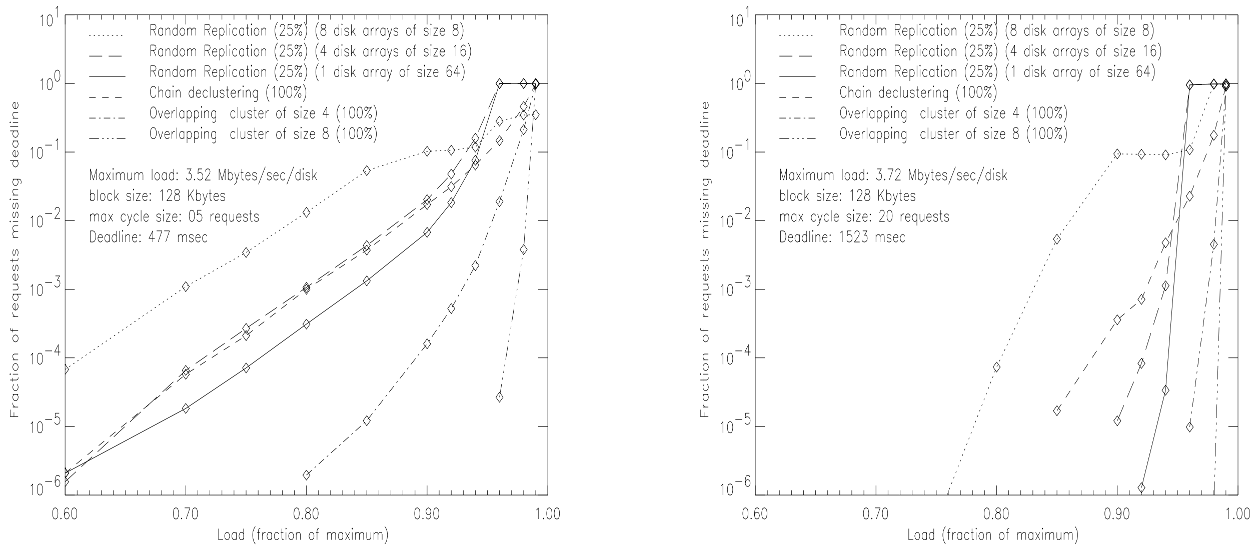


Figure 18: Performance with one disk failure

primarily to the disk array containing the failed disk, although a fraction of this load is redirected to other disk arrays by exploiting the partial replication. Thus, increasing the size of the disk array improves the system performance since the excess load is distributed among a larger number of disks. However increasing the size of the disk arrays reduces the reliability of the system since no multiple disk failures within a single disk array can be supported. The case of disk arrays with 8 disks shows a lower probability of missing the deadline than the cases of larger disk arrays on saturation, simply because the excess load saturates only the disk array that contains the failed disk, while the redirected load to the other disk arrays is not enough to saturate them, and thus a fraction of requests sent to disk arrays without failure are completed before the deadline. When the disk array size is larger, a higher load is redirected to the disk arrays without failure when the disk array with failure is saturated, saturating all the disks. The region of the curve for disk arrays of size 8 that is constant at approximately 10^{-1}, correspond to the situation where the disk array with the failed disk is saturated.

The performance of chained declustering under failure is only better than partial replication with parity groups when the disk array is small (8 disks in this example). When the size of the disk array is 16 disks the performance of chained declustering and RAIDs are equivalent for small deadlines (case with a maximum of 5 requests per cycle), but for larger deadlines (case with a maximum of 20 requests per cycle) chained declustering is worse. Disk arrays larger than 16 provide better performance than chained declustering

Scheme	Mean time between "data inaccessible" failures
Chained declustering	$\frac{K}{2}$
Random replication (100%) (System with D disks)	$\frac{K}{D-1}$
Disk array (size 8)	$\frac{K}{7}$
Disk array (size 16)	$\frac{K}{15}$
Disk array (size 64)	$\frac{K}{63}$
Overlapping clusters (size 4)	$\frac{K}{5}$
Overlapping clusters (size 8)	$\frac{K}{11}$

Table 1: Relative mean time between "data inaccessible" failures

under failure both with small and large deadlines, with a lower cost on data storage overhead, but with less tolerance to multiple disk failures.

In our experiments, overlapping clusters is the scheme that provided the best performance under disk failure (as well as without disk failure) and performed better for larger sizes of the overlapping clusters. However it has a cost of 100% storage capacity overhead and has less tolerance to multiple disk failures than chained declustering (but it has better or equal tolerance to multiple disk failures than RAID schemes).

Table 1 shows the relative mean time between "data inaccessible" failures (failures where a fraction of data is unaccessible) for all cases. We assume that the mean repair time of disks is much smaller than the mean time between disk failures in the system. If this is the case the mean time between "data inaccessible" failures are dominated by double failures, and it is proportional to the number of disks in which a second failure (after a failure in a specific disk) would cause unavailability of data. Note that if the number of disks is very large this assumption may not hold and we may have to consider multiple failures with more than 2 failed disks, but in general this is a reasonable assumption and is assumed here. Table 1 shows the approximate mean time between "data inaccessible" failures relatively to a constant K that can be computed as function of the mean time between failure of a disk and the mean repair time. This time is inversely proportional to the number of disks for which a second failure would cause unavailability of data.

We conclude that if there is enough data storage available to support 100% replication, the best scheme is overlapping clusters which has very good performance both with and without disk failures and also has relatively good tolerance to disk failures. If however data storage is scarce, then the use of disk arrays with parity groups and partial replication (eventually with no replication if data storage is too scarce) is the best choice. The selection of the size of the disk array needs to be done according to the relative importance of tolerance to "data inaccessible" failures against system performance under single disk failure, with larger disk arrays providing higher performance while smaller disk arrays provide better fault tolerance. Also the selection of the level of replication and size of parity groups is a tradeoff between the desired percentage of additional storage capacity and the benefit of increased performance with higher levels of replication and increased performance under disk failure for smaller sizes of parity groups.

7 Performance Comparison of Random Allocation and Streaming RAID

In this section we compare the performance of random allocation with replication and the performance of streaming RAID [24]. Streaming RAID is specifically designed for a workload in which objects are read sequentially at a constant rate (for example, playing video with constant bit rate encoding). RIO is applicable to much more general workloads. So the comparison we do here favors streaming RAID significantly. Let's assume that each media stream requires a constant bit rate of 1.5 Mbits/sec (approximately the rate of a MPEG1 video) and that a buffer of 256 Kbytes is allocated to each data stream. We then compute the maximum number of streams that can be supported by both schemes as described below.

In streaming RAID, requests are processed in synchronized cycles of constant duration. In each cycle, each stream accesses data blocks from a particular disk array and in the following cycle the next block for each stream is accessed in the following logical disk array, making all streams cycle through all disks arrays that compose the system until the last block of the each stream is read. After a given block is read it is then consumed at a constant rate by the user until the next block is read in the next cycle. The block size is set to half the size of the stream buffer, since while the last read block stored is being consumed from one half of the buffer the next block is being read in the other half. Since we are assuming a buffer of size 256 Kbytes per stream the block size used is 128 Kbytes. The duration of a cycle should be equal to the time necessary for the user to consume one data block. For a required rate of 1.5 Mbits/sec and a block of 128 Kbytes, the cycle time is given by

$$T = \frac{128}{\frac{1.5}{8}1024} = \frac{2}{3}sec. = 0.667sec.$$

To compute the number of requests that can be processed in each disk in each cycle, we have to consider the cycle time distributions for different numbers of requests as previously shown in figure 8 and select the maximum possible number of requests in a cycle such that the maximum time to read the requests does not exceed the required cycle time T computed above. To have a fair comparison with random allocation we select the number of requests in a cycle such that the probability of having a cycle longer than the required value T is less or equal to the desired probability of missing the deadline, instead of using the worst case cycle time. We are assuming that streams are served always in the same order within a cycle and that a long cycle will cause all requests to miss their deadlines. The real situation is a little more complex than this, since the distribution of times of consecutive I/O requests in a cycle vary, and thus the time elapsed from the retrieval of one block to the retrieval of the next one of the same stream in the following cycle is different for each stream ; but for the purpose of our comparison this simplification is a good approximation. To avoid being too conservative for the case of streaming RAID, we ignored possible extra delays due to thermal calibration in this case, although they were included in our scheme (So, all approximations are done in a manner to favor streaming RAID).

We assume a probability of missing the deadline equal to 10^{-6} and then compute the maximum cycle time T_{max} as the time such that the probability of having a cycle longer than T_{max} is equal to 10^{-6}. Figure 19 shows this maximum cycle time T_{max} as function of the number of requests in a cycle. We observe that to have maximum cycle time equal to 0.667 , the maximum number of requests per disk in a cycle is 17.

For RIO, we use the same block size of 128 Kbytes and deadline of 0.667 sec. For these values we use the graphs of figure 13 to compute the maximum bandwidth per disk that the system can support for a probability of missing the deadline of 10^{-6} and then divide this value by the stream data rate. From figure 13 the maximum supported bandwidth per disk is 2.88 Mbytes/sec, 3.43 Mbytes/sec and 3.62 Mbytes/sec for 0%, 25% and 100% of replication respectively.

Using these values we show in table 2 the maximum number of streams that can be supported for both schemes assuming a system with 14 disks. Although the example is based on a 14 disk system, the relative

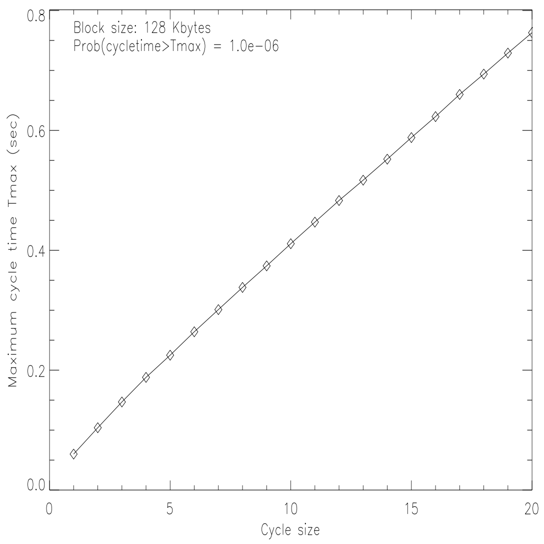


Figure 19: Maximum cycle time

Scheme	Maximum number of streams	relative performance (w.r.t. streaming RAID)
streaming RAID	238	100%
Random allocation with 100% replication	270	113%
Random allocation with 25% replication	255	107%
Random allocation with 0% replication	215	90%

Table 2: Comparison of Random replication with streaming RAID

performance of the system is approximately the same for any number of disks, since we have shown that the performance of random allocation is relatively insensitive to the number of disks, as well as the same is true for streaming RAID.

The results of this example show that the performance of the RIO random allocation scheme is equivalent to the performance of streaming RAID, in the ideal case for streaming RAID where the data access pattern is highly predictable. In this example the performance of random allocation is on the order of 10% better than streaming RAID when replication is used (even with partial replication) and 10% worse when no replication is used. Although random replication has to operate at a load lower than the effective disk bandwidth, so does streaming RAID, since the maximum number of requests in a cycle has to be selected based on the maximum cycle time, instead of the mean cycle time that defines the effective disk bandwidth. This effect reduces disk utilization for streaming RAID, since at the tail end of cycles many disks are idle waiting for the beginning of the next cycle.

This is only a particular example (bit rate, block size, buffer size, etc.) and to have a more extensive comparison, we need to evaluate the performance of the system under different sets of parameters, but since our results of sections 5 and 6 show that the performance of random allocation is not far from the the maximum disk effective bandwidth, we can conclude that random allocation will always be equivalent or better than streaming RAID. We, however, expect the performance of random allocation to be better than streaming RAID in most cases. Moreover, random replication has many other advantages over streaming RAID such as flexibility to support streams with different playout rates without bandwidth fragmentation (as is the case on staggered striping), flexibility to support VCR functionality for videos with simpler schemes, flexibility to switch among multiple data representations for dynamic quality of service, etc.

8 Conclusion

We have presented the principal issues involved on the design of a parallel storage system for realtime multimedia applications. The realtime storage system described in this paper was designed and implemented as part of the VWDS Project at the UCLA Computer Science Department.

The principal characteristics of the proposed storage system is random allocation of blocks to disks and use of partial replication for improving load balancing and thus achieving smaller probabilities of missing deadlines for realtime requests.

Our experimental results showed the validity of our scheme, and demonstrated that is possible to achieve small probabilities of missing deadlines (less than 10^{-6}) with high disk utilization (70% to 95% of the disk effective bandwidth), for relatively small request deadlines (0.5 to 1.5 seconds) using partial replication and 60% to 80% disk utilization when no replication is used. We also showed that full replication allow almost full disk utilization (up to 99% of the effective disk bandwidth).

We also discussed different approaches for fault tolerance schemes and analyzed their performance both when there is a disk failure and under normal operational conditions. We proposed a fault tolerance scheme which we called overlapping clusters that has both good performance characteristics and reliability under disk failure that would be the preferred scheme when storage space is not scarce and 100% replication is acceptable. In systems where storage space is scarce relative to bandwidth the use of RAID schemes combined with partial replication would be more adequate.

Finally we compared the performance of our scheme with streaming RAID, under the ideal workload for streaming RAID; videos in which streams require the same constant playout rate. Even in this ideal situation we showed that our scheme has superior performance to streaming RAID when full or partial replication is used, while its performance is only 10% bellow streaming RAID when no replication is used.

References

[1] Y. Azar, A.Z. Broder, A.R. Karlin, E.Upfal, "Balanced Allocations", *Proc. 26th Annual ACM Symposium on the Theory of Computing (STOC 94)*, pp. 593-602, 1994

[2] S. Berson, R. Muntz, S. Ghandeharizadeh, X. Ju, "Staggered Striping in Multimedia Information Systems", *SIGMOD 94*, pp.79-90, 1994.

[3] S. Berson, R. Muntz, W. Wong, "Randomized Data Allocation for Real-time Disk I/O", *Compcon 96*, pp.286-90, 1996.

[4] Birk, Y., "Track-Pairing: A Novel Data Layout for VOD Servers with Multi-Zone-Recording Disks", *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, pp. 248-255, May 1995.

[5] Chen, L.T., Rotem,D., "Declustering Objects for Visualization", *VLDB 1993*, Dublin, Ireland, pp. 85-96, 1993.

[6] A.L. Chervenak, A.A. Patterson, R.H. Katz, "Choosing the Best Storage System for Video Service", *ACM Multimedia 95*, pp.109-119, 1995.

[7] T. Chiueh, R.H. Katz "Multi-Resolution Video Representation for Parallel Disk Arrays", *ACM Multimedia 93*, pp. 401-9, 1993.

[8] Funkhauser, T., Sequin, C., Teller, S., "Management of Large Amounts of Data in Interactive Building Walkthroughs", *ACM SIGGRAPH Proc. of the 1992 Symposium on Interactive 3D Graphics*, 1992.

[9] W. Gekelman, D. Leneman, J. Maggs, "Experimental Observation of Alfven Wave Cones", *Physics of Plasmas*, 1, pp.3775-3783, 1994.

[10] Gemmell, D., Vin, H., Kandlur, D., Rangan, V., Rowe, L., "Multimedia Storage Servers: A Tutorial", *IEEE Computer*, pp.40-49, May 1995.

[11] J Hsieh, M. Lin,J.C.L. Liu, D.H.C Du, "Performance of a Mass Storage System for Video-On-Demand". *Journal of Parallel and Distributed Computing*, Vol. 30, No. 2, pp.147-67, Nov. 1995.

[12] E. Grochowski, R.F. Hoyt "Future Trends in Hard Disk Drives", *IEEE Transactions on Magnetics*, Vol. 32, No. 3, May 1996.

[13] H. Hsiao, D.J. Dewitt, "Chained Declustering: A New Availability Strategy for Multiprocessor Database Machines", *Proc. of Data Engineering*, pp 456-65, 1990.

[14] K. Jeffay, D.F. Stanat, C.U. Martel, "On Non-preemptive Scheduling of Periodic and Sporadic Tasks", *Proc. of Real-time Systems Symp.*, pp.129-139, Dec. 1991.

[15] Jepson, W., Liggett, R., Friedman, S., "Virtual Modeling of Urban Environments", *Presence: Teleoperators and Virtual Environments*, Vol. 5, No.1, MIT Press, 1996.

[16] W. Karplus, M.R. Harreld, "The Role of Virtual Environments in Clinical Medicine: Scientific Visualisation", *Proc. First Jint Conference of International Simulation Societies (CISS)*, Zurich, Switzerland, pp. 13-17, September 1994.

[17] C.L. Liu, J.W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-time Environment" *journal of ACM*, pp 46-61, 1973.

[18] Miller, E.L., Katz, R.H., "RAMA: Easy Access to a High-Bandwidth Massively Parallel File System", *USENIX 95*, pp. 59-70, 1995.

[19] M.D. Mitzenmacher, "The Power of Two Choices in Randomized Load Balancing", *PhD Dissertation*, University of California at Berkeley, Computer Science Department, 1996.

[20] D.A. Patterson, G. Gibson, R.H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)", *SIGMOD 88*, pp.109-116, 1988.

[21] Reddy, A.L.N., Wyllie, J., "Disk Scheduling in a Multimedia I/O System", *ACM Multimedia 93*, pp. 225-233, 1993.

[22] Ruemmler, C., Wilkes, J., "An Introduction to Disk Driving Modeling", *IEEE Computer*, pp17-28,March 1994.

[23] M. Seltzer, P. Chen, J. Ousterhout, "Disk Scheduling Revisited" *USENIX Winter 90*, pp.313-324, 1990.

[24] F.A. Tobagi, J. Pang, R. Baird, M. Gang, "Streaming RAID - A Disk Management System For Video Files" *ACM Multimedia 93*, pp.393-400, 1993.

[25] J.L. Wolf, P.S. Yu, H. Shachnai, "DASD Dancing: A Disk Load Balancing Optimization Scheme for Video-on-Demand Computer SYSTEMS". *SIGMETRICS 1995*,pp.157-166, 1995.

[26] B.L. Worthington, G.R. Ganger, Y.N. Patt, J. Wilkes, "On-Line Extraction of SCSI Disk Drive Parameters" *SIGMETRICS 95*, pp.146-156, 1995.

[27] P. Yu, M. Chen and D. Kandlur, "Design and Analysis of a Grouped Sweeping Scheme for Multimedia Storage Management", *NOSSDAV'92*, pp.44-55, 1992.