A Fault Tolerant Design of a Multimedia Server

Steven Berson Leana Golubchik
Richard R. Muntz

UCLA Computer Science Department

Abstract

Recent technological advances have made building of multimedia on-demand servers feasible. Two challenging tasks in such systems are: a) satisfying the real-time requirement of continuously providing an object at a specified bandwidth and b) servicing multiple clients simultaneously; to accomplish these tasks, the multimedia server would require a large disk subsystem. Although a single disk is fairly reliable, such a large disk farm would have a high probability of disk failure. Due to the real-time constraint, the reliability and availability requirements of multimedia systems are even more stringent than those of traditional database systems. In this report we study techniques for providing high reliability and availability, at low storage and bandwidth costs, for on-demand multimedia servers.

1 Introduction

Technological advances over the past decade have made on-demand multimedia servers feasible. A challenging task in such systems is the real-time requirement of continuously providing an object at a constant bandwidth, e.g., if this object is a movie, then, once it begins, it must be transmitted continuously for the duration of the film at a specified bandwidth. Another challenging task in multimedia systems is to service multiple clients simultaneously. Both of these tasks are accomplished in [1] through a proper layout of the data on the disk drives, using a technique termed *staggered striping*, which is discussed in more detail in Section 2.

An example multimedia system is illustrated in Figure 1; it includes a multimedia server, a communication network, and a set of display stations. The multimedia server consists of a tertiary storage library, a collection of disks, and a set of processors. The entire database permanently resides on tertiary storage, where objects are retrieved and placed on disk drives on demand. The size of the objects precludes them from being stored in main memory. The high latency of tertiary devices, largely due to their sequentiality, combined with the relatively high cost of tape drives[1] precludes objects from being transmitted directly from

[1]The seek latency for a 1.3GB tape on a $1000 tape drive can be on the order of 20 seconds [4], whereas

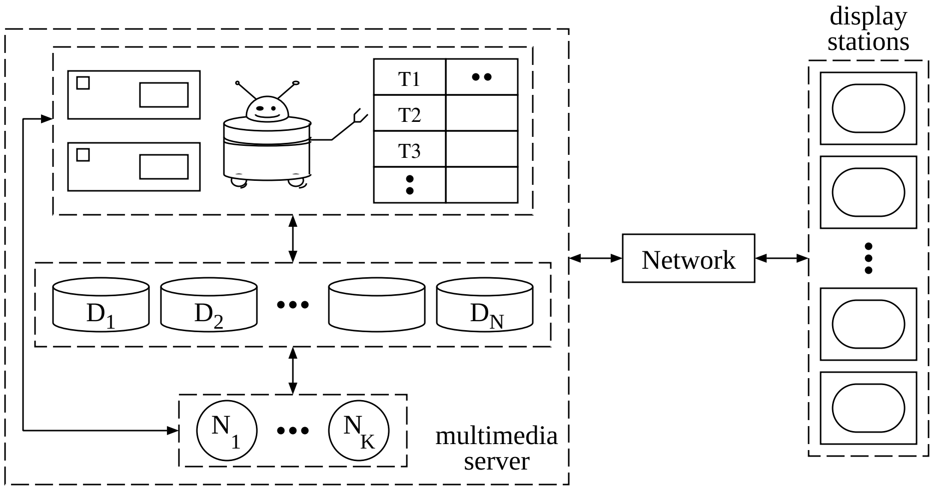


Figure 1: Multimedia System

the tertiary store. If the secondary storage capacity is exhausted when an object is requested which is not on the disks, then one or more objects must be purged. It is important to note that retrieval of objects from the tertiary store results in a significant delay and should be avoided as much as possible. The more objects that can be stored on disks, the less is the probability that any one request will result in tertiary storage access.

To exhibit reasonable performance, the multimedia server should contain a large number of disks; something on the order of 1000 drives would not be uncommon. A large number of disks would provide the bandwidth required to service many requests simultaneously and the storage space needed to insure that the probability of fetching an object from tertiary store is small. Although a single disk can be fairly reliable, given such a large number of disks, the probability that one of them fails is quite high. For example, if the mean time to failure (MTTF) of a single disk is $100,000$ hours; this means that the MTTF of some disk in a 1000 disk system is on the order of 100 hours. This can be unacceptable for traditional database systems, which require a high degree of reliability and availability of data. Due to the real-time constraint, the reliability and availability requirements of a multimedia system are even more stringent. Given the system of Figure 1, a disk failure does not result in data loss, since a copy of all objects is stored on tertiary storage. However, it can result in interruption of requests in progress. If some of the data for a currently displayed object is on a disk that fails, a significant degradation in performance is incurred, since the portion of objects lost due to the failure must be retrieved from tertiary storage. Therefore, without some form of fault tolerance, such a system would not be acceptable.

To improve the reliability and availability of the system we must use some fraction of the disk space to store redundant information. For instance, consider the storage subsystem

a similarly priced disk, of a similar capacity, has a maximum seek time on the order of 35 milliseconds and more than 16 times the transfer rate. Tape systems with significantly higher transfer rates and tape capacities (although not with much lower seek latency) as compared to the disk technology, do exist, but at a cost $\$40,000$-$\$300,000$.

illustrated in Figure 2, where two out of every three disks are used to store "real" data and the third one is used to store parity information, e.g., $X0.p = X0.0 \oplus X0.1$. In this example, only two thirds of the total storage space (disks 0 and 1) is occupied by actual objects. Similarly, only two disks at a time can be used to transmit data; when all three disks are operational, disks 0 and 1 are used to transmit data, but when one of these disks fails, the parity disk is read in its place and the missing data is reconstructed on-the-fly. Therefore, in this example, one third of the storage space and one third of the available bandwidth are used for reliability.

How much redundant information we store and how we place it on the disks will determine: a) the fraction of storage available for "real" data, b) the fraction of disk bandwidth available for delivering this data, and c) the resiliency of the system to disk failure. In general, the more redundant information is stored, the lower is the probability that a failure results in data loss[2] and a subsequent access of tertiary storage; although not a massive one (say on the order of 1GB), this access would be scattered[3], which is (usually) expensive for tertiary storage. On the other hand, the less redundant information is stored, the more space is available for storing objects and possibly the more bandwidth is available for displaying them. Therefore improvements in reliability must be balanced against degradation in performance.

In this report we present several *parity* based fault tolerance schemes for a multimedia server. For each scheme, we determine: a) how much storage overhead is incurred, b) how much bandwidth is wasted, c) how system behavior is altered due to a disk failure, and d) how many failures can the system withstand before degradation of service occurs or data is lost. *Degradation of service* denotes the situation in which the system is unable to continue servicing all requests that are active when the failure occurs. This may be necessary due to a lack of available bandwidth (as opposed to data loss). In this situation, one or more requests have to be rescheduled at a later time. We define a failure to be *catastrophic* if it results either in data loss or in degradation of service, since both are considered unacceptable. Our goal in this report is to determine schemes for providing high reliability and availability for multimedia servers and to study the design tradeoffs associated with each.

The rest of the report is organized as follows. Section 2 presents background information in the areas of multimedia servers and fault tolerant disk subsystems. Sections 3 and 4 present our parity schemes, and Section 5 compares these schemes on the merits of performance and reliability. Section 6 presents concluding remarks.

[2]The data is not really lost, since there is a copy on tertiary storage, but part of an object that was in disk storage is no longer there.

[3]Such accesses would be scattered since the disk subsystem reflects only a small fraction of the whole database, and the objects are fetched into disks in a "random" fashion.

2 Background

The concepts of striping and declustering are used in general purpose I/O subsystems, as in RAID technology [9], as well as in application specific systems, such as database management systems, e.g., [3] and [10], to name a few. In [5] and [6], these concepts are extended to a parallel multimedia system. In [7] an architecture including a tertiary store is introduced and a mechanism for supporting multiple users, termed *virtual replication*, is described; the authors suggest partitioning the drives in the system into clusters and replicating frequently accessed objects on several clusters to reduce contention for a cluster containing a popular object. The *staggered striping* method, introduced in [1], is a major improvement over the virtual replication method because it avoids bottleneck formation by distributing each object across all the disks in the system, thus balancing the system's load without resorting to replication of data. In a multimedia system, such as an on-demand video server, disk space is limited, in the sense that only a fairly small fraction of the objects can reside on secondary storage. Eliminating the need for replication increases the number of disk-resident objects and therefore reduces the probability of having to satisfy a request by fetching an object from tertiary storage; this results in a significant performance improvement.

The basic idea behind staggered striping is that objects that are brought into disks from tertiary storage are striped across all the disks in the system. Each object is divided into subobjects, and each subobject is divided into fragments. The size of a fragment is chosen to be sufficiently large to amortize the seek and rotational latency over a large enough number of bytes to obtain most of the available disk bandwidth. The number of fragments in a subobject corresponds to the number of disks required to satisfy the bandwidth requirement, k, of the object, and is termed the *degree of declustering*; we define the set of k disks storing a subobject as a *space slot*. A fragment corresponds to the amount of data to be read from a single disk in a single *time slot* or *time interval*. A time interval is the amount of time a disk needs to position its head and read the fragment; a more thorough definition appears in [1]. (For the remainder of the report, we will refer to "time slot t" as "time t".) We place a subobject on a set of k disks; to place the next subobject we move onto the next set of k disks (the two sets could overlap). The *stride* determines the number of disks we skip before placing the next subobject; in other words, it determines the distance (in number of disks) between the first fragment of subobject i and the first fragment of subobject $i + 1$[4]. An example of such a system is illustrated in Figure 10, where subobjects of X are declustered among three disks and subobjects of Y are declustered among two disks. The first subobject of object X, $X0$, resides on disks $0, 1$ and 2, where the first fragment $X0.0$ resides on disk 0, the second fragment $X0.1$ resides on disk 1, etc. The stride in this example is equal to 1, and hence the first fragment of subobject $X1$ is placed on disk 1. Object Y is placed in a similar manner. The scheduling of X for transmission is as follows. We first find a time slot in which the space slot corresponding to X's first subobject, namely disks $0, 1$ and 2, is idle. This is the time slot in which we can begin transmitting X; call this time slot t. At time t we transmit $X0$, then at time $t + 1$ we (move over by 1 disk) and transmit $X1$ from disks $1, 2$ and 3, and so on. Note that while the degree of declustering can be different for

[4] All arithmetic is done modulo number of disks.

different objects, the stride is the same for all objects.

A special case of staggered striping occurs when all objects have the same bandwidth requirement and hence the same degree of declustering, with the stride equal to the degree of declustering; this is termed *simple* striping. In this case, the disks are divided into clusters, with each cluster containing k disks with the stride also equal to k. Figure 2 illustrates such as system, with 3 clusters and degree of declustering equal to 2 (for the moment, we ignore the parity disks in that figure). Scheduling in this system is similar to the staggered striping schedule described above. For instance, if in Figure 2 at time t cluster 0 is idle, then transmission of object X can begin. Hence, at time t subobject $X0$ can be transmitted from cluster 0; at time $t+1$ subobject $X1$ can be transmitted from cluster 1, and so on. A more detailed description of the physical data allocation, request scheduling, etc., in the context of simple and staggered striping, can be found in [1].

As mentioned in Section 1, reliability and availability of data are serious concerns in information systems, which are solved by storing redundant information. One way to introduce redundancy into the system is to use parity based schemes which construct a parity block for every d data blocks, where mirroring [2] is a special case with $d = 1$. (An example, using Figure 2, is given in Section 1.) The parity block plus the d data blocks constitute a parity group; we define the number of data blocks in a parity group to be the parity group size. The overhead for parity based schemes is $\frac{1}{d+1}$ of the total storage space, which allows control over the amount of storage (and bandwidth) overhead due to replication of information, i.e., the more data blocks are in each parity group, the smaller is the fraction of storage used for redundant information.

There are three modes of operation for a disk subsystem [8], originally defined in the context of disk arrays: 1) *normal mode*, where all disks are operational, 2) *degraded mode*, where one (or more) disks have failed, and 3) *rebuild mode*, where the disks are still down, but the process of rebuilding the missing information on spare disks is in progress. Due to the lack of space, we only discuss the system's behavior under normal and degraded modes of operation. In the following sections we extend the idea of parity based schemes to multimedia servers, first in the context of simple striping and then in the context of staggered striping.

3 Simple Striping

In this section we discuss possible parity schemes in the context of simple striping.

3.1 Naive Scheme

A simple scheme for providing parity information is to add a parity disk to each cluster, which amounts to every cluster acting as a separate disk array. The example of Figure 2 illustrates this *naive* scheme, where every cluster consists of two data disks with an added parity disk[5];

[5]We use examples with small number for ease of exposition.

for instance, fragments $X0.0$ and $X0.1$ are protected by parity stored in fragment $X0.p$. During the normal mode of operation, only the data disks are used for displaying objects;

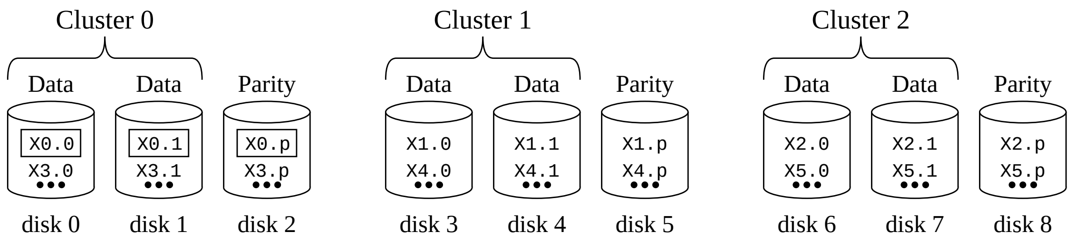


Figure 2: Naive Parity Scheme

however, we should *read* both the data and the parity disks at all times. If we do not read the parity disk during normal operation and a disk fails in the middle of a time slot, then there will not be sufficient time to mask the failure in the same time slot; this would result in interruption in service. During the degraded mode of operation, the surviving data disks plus the corresponding parity disk are read and used to reconstruct the data that resides on the failed disk. For instance, to deliver $X0$ in Figure 2 under normal operation, we only need the data from disks 0 and 1. Under failure, for example of disk 0, $X0.1$ and $X0.p$ are read from disks 1 and 2, respectively, and $X0.0$ is reconstructed through a parity computation[6].

The naive scheme can withstand up to one failure per cluster before a catastrophic failure occurs. In this scheme there can never be a degradation of service without data loss, since enough bandwidth is reserved in a cluster to make up for a single disk failure. Note that in this naive scheme reliability is gained at the cost of both storage and bandwidth. For instance, in the example of Figure 2 about 33% of the available storage is not used to store "real" data, and about 33% of the available bandwidth is not being utilized.

To reduce the storage overhead and the amount of bandwidth wasted due to parity, we could increase the number of clusters per parity group. In other words, we can use one parity disk for every j clusters, and hence construct parity groups using j subobjects, as opposed to one. (The first and the last parity group of every object might, of course, have less than j subobjects.) This *multi-clusters-per-group* scheme is illustrated in Figure 3, where all three clusters share a single parity disk. In this example, each parity group, except for the first and last group of every object, is constructed using 3 subobjects. For instance, object X has 4 subobjects and is split into two parity groups, one of size 3 and one of size 1; therefore,

$$P0 = X0.0 \oplus X0.1 \oplus X0.2 \oplus X1.0 \oplus X1.1 \oplus X1.2 \oplus X2.0 \oplus X2.1 \oplus X2.2$$

$$P1 = X3.0 \oplus X3.1 \oplus X3.2.$$

When a disk fails, we can mask the fault by reading an appropriate fragment from the parity disk whenever the failed disk is scheduled to read a fragment; however, note that this might result in a "hiccup" in data delivery. In general, to avoid interruptions in data delivery,

[6]At the time of failure, time slots would have to be increased slightly to account for parity computation.

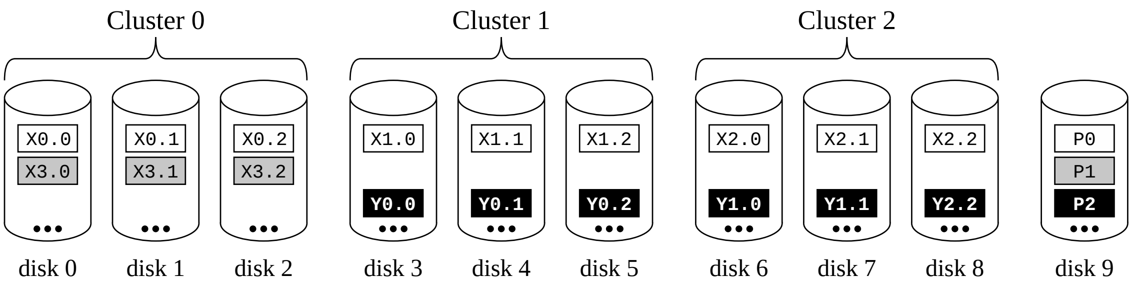


Figure 3: Multi-clusters-per-group Scheme

during the degraded mode of operation, it is necessary to read an entire parity group before starting delivery (to the network or equivalently to the display station), since the parity computation can not be completed until that time, i.e., if the disk failure effects a subobject other than the last one in a parity group, then that subobject can not be reconstructed until the *last* subobject in a parity group is read.

In this scheme, there are several subobjects in each parity group, and therefore, after a failure has been detected, there might be a "hiccup" in data delivery which results in a need for buffer space (for collecting enough information to compute the parity). For instance, if delivery of subobject $X1$ is scheduled for time t in Figure 3 and disk 4 fails, then we are faced with the following problems. If we do not buffer $X0$, at time $t-1$ (for future parity computation), then there will be a hiccup in the delivery of X, i.e., we will be forced to "freeze" the delivery of X until the next time slot when X is rescheduled on cluster 0 and we are able to re-read that information. Moreover, even if we do buffer $X0$ at $t-1$, we must in addition buffer $X1.0, X1.2$, and $P0$ until $t+1$, when we can complete the parity group by reading $X2$. Figure 4(a) illustrates delivery of X with buffering of parity groups, after the failure of disk 4 has been detected[7].

As mentioned above (and illustrated in Figure 4), there will still be a small[8] hiccup in delivery of X due to the lack of information at time t; moreover, this hiccup will persist throughout the remainder of delivery of X (until disk 4 is rebuilt), i.e., it will occur every time X is scheduled on cluster 1. We can remedy this problem by offsetting the start of an object's transmission for $j-1$ time intervals, i.e., until the first parity group is read in its entirety. The offset insures smooth data delivery under failure by providing sufficient time to read an entire parity group before having to deliver its first subobject. This modification to the delivery schedule of Figure 4(a) is illustrated in Figure 4(b). The penalty for avoiding periodic hiccups is a small delay in transmission starting time plus the buffer space required to hold $j-1$ subobjects for each request in service. Since this buffer space must be reserved in case of failure, we could use it to offset the delivery of each request under normal operation;

[7]Buffering of each fragment of a parity group is not necessary; we can reduce the buffer space requirements by computing the parity on-the-fly and buffering only the result; however, as will be shown shortly, there is still a need for buffering $j-1$ subobjects for each request in service, which makes this optimization unnecessary.

[8]Something on the order of a third of a second, in this example.

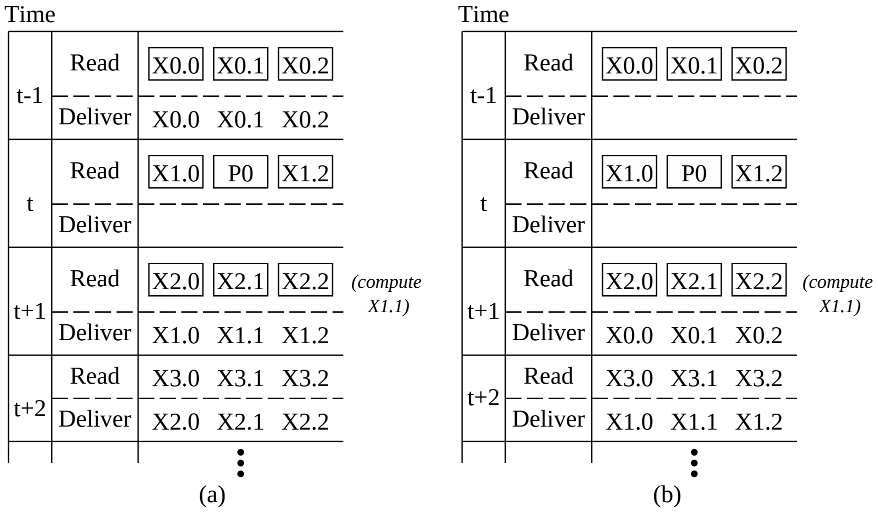


Figure 4: Delivery under Failure

at the time of failure, this will eliminate the initial hiccup for all requests.

Note that the disk storage and bandwidth costs in this scheme can be much lower than in the basic naive scheme. For instance, if in the previous example all objects are divided evenly into parity groups of j subobjects each, then only 10% of storage and bandwidth would be given up for parity. However, larger parity groups increase the probability of multiple failures occurring within the same parity group, which results in data loss; in addition, in this scheme, larger parity groups also result in increased buffer space requirements. Hence, the increased efficiency in storage and bandwidth must be balanced against the decreased reliability and availability as well as the increased buffer space requirements.

Another approach is to try to associate parity with physical locations of the data blocks (as is done in disk array systems [9]), rather than with a particular object. This scheme is illustrated in Figure 5, where again there is only one parity disk corresponding to all three clusters. Subobjects $X0, X1$, and $Y0$ are protected by parity $P0$; subobjects $Y1, Y2$, and

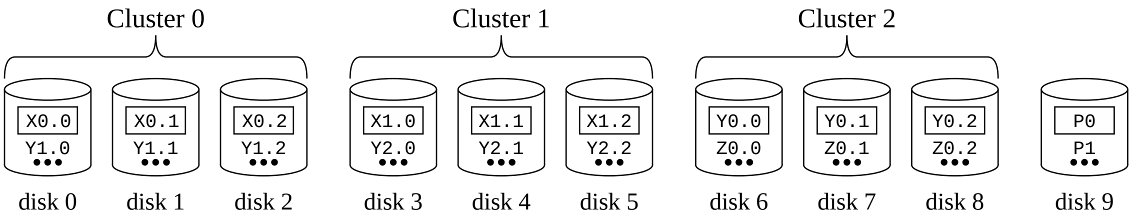


Figure 5: Another Improvement to the Naive Parity Scheme

$Z0$ are protected by parity $P1$, etc. Constructing parity groups in this manner, however, will not allow delivery of data in real-time under failure; this is best illustrated through the example of Figure 5. Suppose that disk 0 fails and consequently $X0.0$ must be reconstructed. To compute the missing data, $X0.1$, $X0.2$, $X1$, $Y0$, and $P0$ must be read. If clusters 1 and

2 are busy and are scheduled to read $X1$ and $Z0$, respectively, then $X0.1, X0.2$ and $X1$ must be buffered until $Y0$ can be read from cluster 2. Clearly, this does not work, since we can not guarantee when an appropriate block will be scheduled to be read or when an appropriate cluster will become idle. In order to make this scheme attractive, we must be able to predict which combinations of objects will be scheduled together, e.g., that X and Z in the above example, not X and Y, will be scheduled together. Therefore, in our system we must associate parity with *objects* rather than with *physical* disk blocks.

3.2 Improved Bandwidth Schemes

One problem with the schemes described thus far is that the parity disks are not used during normal operation and thus significant bandwidth can be wasted in addition to the storage overhead of the parity data. Instead of having dedicated parity disks, which are only used for transmission in case of failure, we can place both data and parity information on a disk, which would allow us to use the bandwidth of all disks during both normal and degraded modes of operation. The simplest way of accomplishing this is to place the parity information associated with cluster i on some disk, let's say the first one, of cluster $i+1$. This *improved-bw* scheme is illustrated in Figure 6, where all the data residing on cluster 0 is protected by the parity on the first disk of cluster 1, etc. Note that this scheme improves the bandwidth

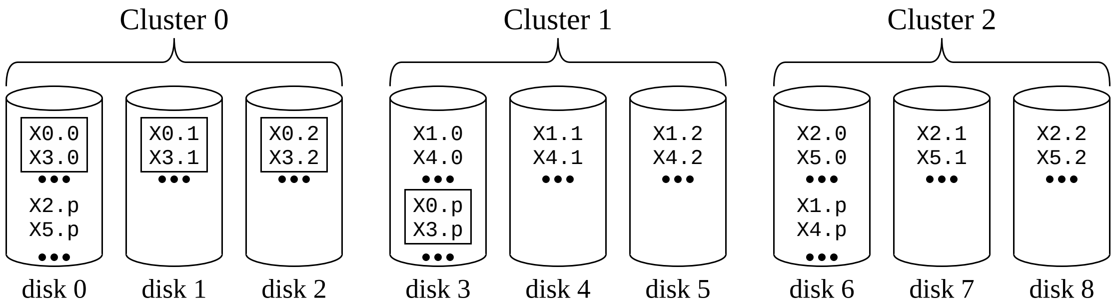


Figure 6: Improved BW Scheme

efficiency, but significantly reduces the storage efficiency, as compared to the naive scheme. For example, in Figure 6, 100% of the bandwidth can be utilized during normal operation, but, since the first disk of every clusters contains both data and parity, only half of that disk, and therefore only half of any other disk, can be used to store actual data; this results in storage efficiency of only 50%, which is by no means a good tradeoff. However, we still use this scheme in the following discussion for ease of illustration and then show how to improve upon it so as not to incur any more storage overhead than in the naive scheme.

When a failure occurs at time t in cluster i, it and its adjacent clusters have to perform a "shift to the right" as follows. Cluster i delivers data from all its operational disks plus it reconstructs the missing data by reading the appropriate parity blocks residing on the first disk of cluster $i+1$. If cluster $i+1$ is busy at time t, then it has to behave as if its first disk also failed, since a disk can read only one fragment in each time slot. Therefore, cluster $i+1$ has to perform a similar shift to the right, i.e., read the data from all its available disks plus use the first disk of cluster $i+2$ to reconstruct the data residing on its first disk. This shift has to propagate to the right until an idle cluster is found. If all clusters are busy, one request must be dropped (we discuss this in more detail below). As an example, consider again Figure 6. If disk 0 fails, then in order to deliver subobject $X0$, $X0.0$ must be reconstructed by reading $X0.1, X0.2$ and $X0.p$ from disks 1, 2, and 3, respectively. If cluster 1 is not idle, then it must perform a similar shift to the right. Of course, if cluster 3 is not idle, then one of the three requests must be dropped.

The major advantage that this scheme has over the naive and multi-clusters-per-group schemes is the additional bandwidth available during normal operation. However, it is not as reliable as the naive[9] scheme. Firstly, if a failure occurs in the middle of a time slot, then it might not possible to mask it for the object scheduled on the failed disk during that time slot, since the entire bandwidth of the system can be utilized during normal operation (i.e., there are no "reserved" parity disks in this scheme that can be read at all times as it is done in the naive scheme). For example, if the failure of disk 0 occurs while we are reading $X0$, then there is not enough time to mask the failure by reading the appropriate parity fragment; in this case we are forced to "freeze" delivery of X until the next time slot when it is possible to reschedule X on disk 0 (delivery of all other objects can continue). If the system is not heavily loaded, it should be possible to find an empty slot fairly quickly; at worst, this would require a full revolution[10], i.e., the time for the slot originally assigned to X to "return" to disk 0. Since at the time of failure it is not possible to avoid an interruption in service of the object using the failed disk, in the remainder of the report we discuss the fate of other objects in the system; in other words, we discuss how to proceed when the failure of a disk is detected sometime before (e.g., during the previous time slot) we start reading the subobject scheduled to use that disk.

In addition, the number of possible scenarios where the second failure turns out to be catastrophic is greater in this system than with the naive parity scheme. In general, a system with K clusters, can withstand up to K failures, such that there is no more than one failure per cluster and no more than one failure per parity group, before data is lost. If the system has no idle slots, then (in general) even a single failure results in degradation of service; this is true because in the normal mode all the available bandwidth is already used. Thus, it is desirable not to use the *full* bandwidth capacity of the system during normal operation, i.e., for a small price, namely, the bandwidth of a single disk, we can insure that the system can recover from a single failure without degradation of service. (This should result in a high degree of availability, since a second failure, before the first failed disk is rebuilt, is very unlikely.) If the system is not full, then it can possibly withstand multiple failures, but no more than the number of clusters. If the second failure occurs in the same run, then it results in degradation of service, where a *run* is a string of busy disks with an idle disk on each side. Consider the example of Figure 7, where $X0, Y0$, and $Z0$ are scheduled to be read at time t (and cluster 3 is idle). If disk 0 fails before time t, then $X0, Y0$, and $Z0$ can still be delivered by doing a shift to the right. Suppose that in addition a second failure occurs by time t; if

[9]We make a comparison with the naive (rather than the multi-clusters-per-group) scheme because the naive and the improved-bw schemes use the same size parity groups.

[10]In a system of 90 disks divided into 30 clusters of 3 disks, this would be on the order of 9 seconds.

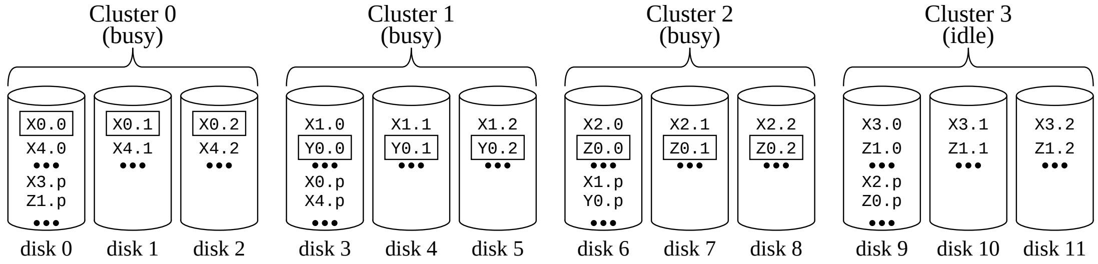


Figure 7: A Single Run

it turns out to be disk 3, then data is lost. If on the other hand[11], it turns out to be disk 7, then no data is lost, but it is not possible to deliver $Z0$, since cluster 2 would have effectively lost two disks: 1) disk 6 due to the shift to the right resulting from the failure of disk 0 and 2) disk 7 due to the second failure. Hence our choices are: 1) drop $Z0$, 2) drop $X0$, which will make the shift unnecessary, or 3) drop $Y0$, which will prevent the shift from propagating to cluster 2. In this system, it might be desirable to drop $Y0$, which will result in two runs of one cluster each, as oppose to one run of two clusters. In general, long runs are undesirable; as the length of a run containing the first failure increases so does the probability that the second failure will occur in that same run and result in degradation of service. Of course, other factors enter into such a decision, e.g., what fraction of a particular object has already been delivered. The increased sensitivity to a second failure is due to the fact that there are dependencies between parity groups which do not exist in the naive scheme, i.e., certain disks belong to two parity groups; for instance, disk 3 in Figure 7 belongs to two different parity groups because it acts as the parity disk for cluster 0 and as a data disk for cluster 1.

As mentioned above, placing all the parity associated with cluster i on a single disk of cluster $i+1$ is wasteful of storage. We can easily improve on this scheme by spreading the parity information among all the disks of cluster $i+1$. For instance, one way of doing this is by placing the parity for the first subobject of cluster i on the first disk of cluster $i+1$; the parity for the second subobject of cluster i can go on the second disk of cluster $i+1$, etc.; in this case, the storage efficiency would be equal to that of the naive scheme. A system using this *improved-bw-storage* scheme is illustrated in Figure 8. In this example subobjects $X0$ and $X3$ reside on cluster 0. The parity corresponding to subobject $X0$ is placed on disk 3, the first disk of cluster 1; the parity corresponding to subobject $X3$ is placed on disk 4, the second disk of cluster 1, etc. In case of disk failure the system behaves essentially as it did with the previous scheme, i.e., it performs the same shift to the right discussed above, except, depending on which subobject is being delivered at the time, a different parity disk is used to compute the missing information. Note however, that this system has a slightly higher probability of data loss than the one using the "improved-bw" scheme. Spreading the parity among all the disks in a cluster, results in even more dependencies between disks in adjacent clusters; hence, any two failures in adjacent clusters result in data loss.

[11]Note that any disk failure in clusters 1 or 2 will result in the same scenario.

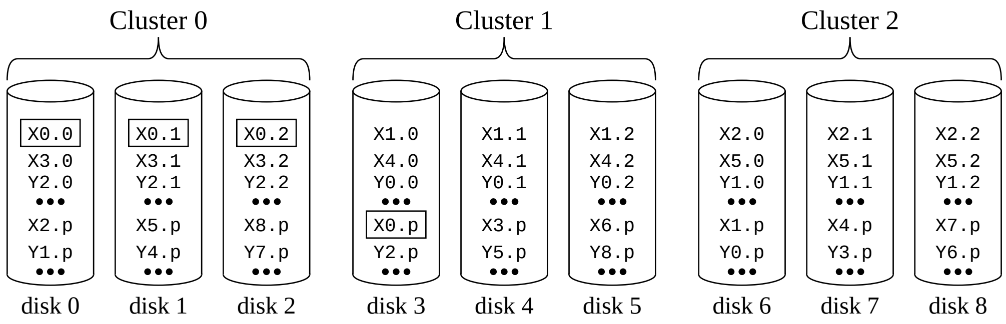


Figure 8: Improved BW-Storage Scheme

4 Staggered Striping

In this section we continue the discussion of parity schemes but in the context of staggered striping, which supports variable bandwidth requirements, i.e., different types of objects.

4.1 Parity per Subobject

If at first we maintain the constant bandwidth requirement, then the improved-bw-storage scheme (see Section 3.2), used with simple striping, can be easily adapted to a system using staggered striping by associating a parity fragment with each subobject. This *parity-per-subobject* scheme is illustrated in Figure 9. In this example there are three objects, X, Y,

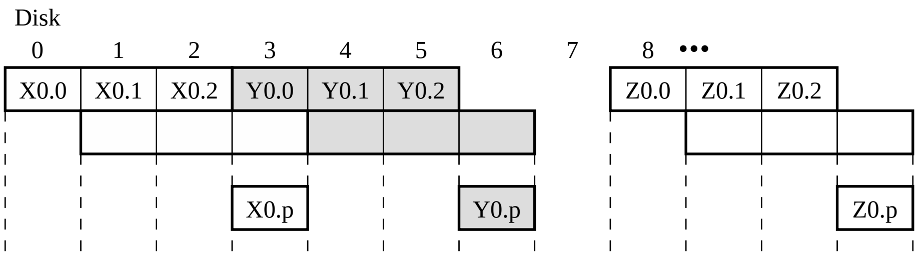


Figure 9: Parity per Each Subobject

and Z; each of the subobjects is declustered among three disks, e.g., subobject $X0$ resides on disks $0, 1$, and 2. The parity for each subobject is stored on the disk following the subobject, e.g., the parity for subobject $X0$ is stored on disk 3. After a disk failure has been detected, all active requests, starting with the one on the failed disk, have to perform a shift to the right, until an idle slot is found. For instance, suppose disk 1 fails in Figure 9 before we read $X0, Y0$, and $Z0$, which are scheduled to be delivered simultaneously. Then, in order to deliver $X0$, we must read $X0.0$ and $X0.2$ as we would under normal operation, plus we must read $X0.p$ from disk 3. In this case, we can not simultaneously deliver $Y0.0$ because disk 3 is busy reading $X0.p$; hence, we must read $Y0.p$ from disk 6 to reconstruct $Y0.0$ stored on disk 3. Assuming disk 6 is not scheduled to read anything, i.e., we found an idle slot, the shift stops here; $Z0$ is not affected by the failure and can be delivered from disks $8, 9$, and 10. Since this is very similar to the improved-bw-storage scheme, we do not discuss it any further, but we note that in this scheme a single idle disk is sufficient to break up a run (see discussion in Section 3.2) as opposed to a whole cluster of disks, as in the improved-bw-storage scheme. We also note, that under variable bandwidth requirements, this scheme remains unchanged, but due to lack of space, we do not elaborate on this further. (In the following sections we assume variable bandwidth requirements.)

As was already mentioned in Section 3, one problem with the parity-per-subobject scheme is that it potentially has too much storage overhead due to parity. In the above example, only three out of every four fragments are object fragments, i.e., only 75% of the storage is used for actual data. The storage penalty for this example is high because the degree of declustering is relatively small. Hence, this scheme should exhibit poor performance in systems with small bandwidth requirements. An extreme example would be a system with a constant bandwidth requirement of 1; in that case we would end up duplicating the data and wasting 50% of the disk space. If the disks are fairly reliable, such a high cost for parity storage is unnecessary, and the system should be able to use much larger parity groups without suffering a significant reduction in reliability. This is also true for low bandwidth objects, i.e., objects that require less than one disk worth of bandwidth. An approach to placing such objects on disks and scheduling them for transmission is discussed in [1]; the parity schemes discussed in this report remain essentially unchanged in the context of low bandwidth objects, except for additional buffer requirements (the details are not given due to the lack of space).

4.2 Multiple Subobjects per Parity Group

The scheme of the previous section has a large disk storage cost, which is a result of small parity groups; to remedy this problem, we can increase the parity group size. A first attempt at this would be to place several consecutive subobjects in one parity group[12]. This scheme is illustrated in Figure 10, where every two subobjects are protected by one parity fragment. For instance, subobjects $X0$ and $X1$, stored on disks $0, 1, 2,$ and 3, are protected by $X01.p$

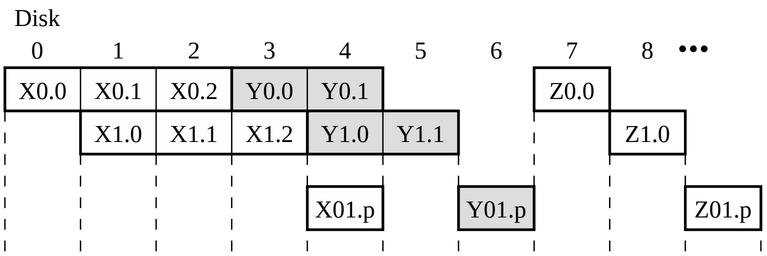


Figure 10: Multiple Subobjects per Parity Group

stored on disk 4.

When a disk fails we again have to perform a shift to the right, which is more complicated

[12]We would like these to be consecutive due to buffering problems, discussed later.

due to having several subobjects in a parity group. We do not discuss the shift in detail in the context of this scheme because it is often not able to recover even from a single failure. For example, consider what happens in Figure 10 when disk 1 fails at time t and the system must still deliver objects X, Y, and Z. In this case there are two fragments lost from one parity group, namely $X0.1$ and $X1.0$, and therefore there is not sufficient data to deliver subobjects $X0$ and $X1$. The problem is that each parity group of object X consists of 7 fragments, e.g., $X0.0, X0.1, X0.2, X1.0, X1.1, X1.2$ and $X01.p$, which reside on only 5 disks; hence, in each parity group there are two pairs of fragments which share a disk, e.g., fragments $X0.1$ and $X1.0$ which share disk 1 and fragments $X0.2$ and $X1.1$ which share disk 2. A failure of either one of these disks effectively results in *multiple dependent failures*. In other words, a single disk failure results in the loss of several fragments and therefore has the same effect as multiple (simultaneous) disk failures. From here on we refer to this condition as "multiple dependent failures".

One way to remedy this problem is to increase the stride such that fragments in the same parity group do not share disks, e.g., as is shown in Figure 11, where the stride is increased to 3. In general, a system can avoid multiple dependent failures by having a stride that is

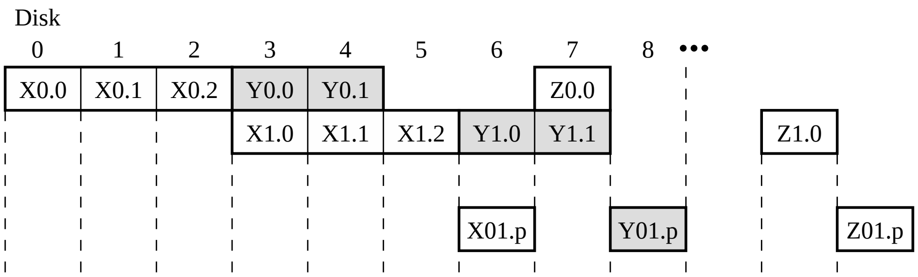


Figure 11: Increased Stride Size

greater than or equal to the largest bandwidth requirement.

Under failure, it is again necessary to shift to the right, and it is also necessary to satisfy the real time constraint by maintaining the same transmission rate as under normal operation. Suppose that the bandwidth requirement can be satisfied by k disks; this means that k fragments must be read during each time slot, whether there is a failure or not. Under normal conditions this is accomplished by reading one subobject in each time slot. But when a failure occurs, it might not be possible to read some fragment of a subobject, either due to the failure or due to the resulting shift to the right; to make up for it we should read some other fragment. With one subobject per parity group, it should clearly be the parity fragment. With multiple subobjects per parity group, some other fragment in the same parity group should be read "out of turn", i.e., some fragment that would normally be read in a later time slot. The problem is that there are usually several choices; for instance, if disk 2 fails in the example of Figure 11, then we could compensate for $X0.2$ by reading any one of the fragments in $X1$ or by reading the parity fragment $X01.p$. The general rules of compensating for a missing fragment are as follows:

- *rule* 1: the fragment to be read out of turn should be the next fragment to the right that is not in the same subobject but is part of the same parity group; sometimes, this turns out to be a parity fragment rather than a data fragment
- *rule* 2*:* never read any fragment twice; therefore, if at time t we read fragment Xi which would normally be read at time $t+n$, then at time $t+n$ we should "pretend" that we are unable to read Xi and employ rule 1 for reading out of turn

For example, if disk 1 fails in the system of Figure 11 before time t, then X and Y can still be delivered as follows (reading and delivery of Z are not shown since it is not affected by the failure):

time t:	read $X0.0, X0.2$, and $X1.0$	*rule* 1
	read $Y0.1$ and $Y1.0$	*rule* 1
time $t+1$:	read $X1.1$ and $X1.2$	
	do *not* read $X1.0$	*rule* 2
	read $X01.p$	*rule* 1
	read $Y1.1$	
	do *not* read $Y1.0$	*rule* 2
	read $Y01.p$	*rule* 1

As with the *multi-clusters-per-group* scheme, this *large-stride* scheme can suffer from hiccups when a failure occurs and requires buffering of data to eliminate this possibility (refer to Section 3.1). Again, the problem occurs because an entire parity group is not read in a single time slot. The discussion on buffering and the "offset-of-delivery" method used to solve the hiccup problem in the context of the multi-clusters-per-group scheme, apply to this scheme, and the rest of the schemes in this report, and hence will not be discussed any further. Note also that it is not necessary to have a constant number of subobjects per parity group. In fact, it is desirable for all objects to have *nearly* equal parity group sizes, which would allow us to decide how much we are willing to pay for reliability in storage, buffer space, etc., and then choose the parity group size(s) accordingly. We come back to this discussion in Section 5.

It might be difficult to predict the largest bandwidth requirement that a system will ever be asked to satisfy. In this case, we could try a different approach to avoiding multiple dependent failures. We could still use multiple subobjects per parity group, but not necessarily consecutive ones. We must still satisfy the condition that no two fragments of a parity group are stored on the same disk, and we can do that by grouping non-overlapping subobjects into parity groups (as opposed to increasing the stride size). This *non-overlapping* scheme is illustrated in Figure 12. In this example, subobjects $X0$ and $X4$ are paired in a parity group, and are protected by fragment $X04.p$; similarly $Y0$ through $Y3$ belong to one parity group and are protected by $Y0-3.p$, etc. In this case, if disk 4 fails before time t, we can still deliver X, Y, and Z as illustrated in Figure 13. The same shift to the right, as was described in the previous scheme, works here. Every time there is a fragment missing from

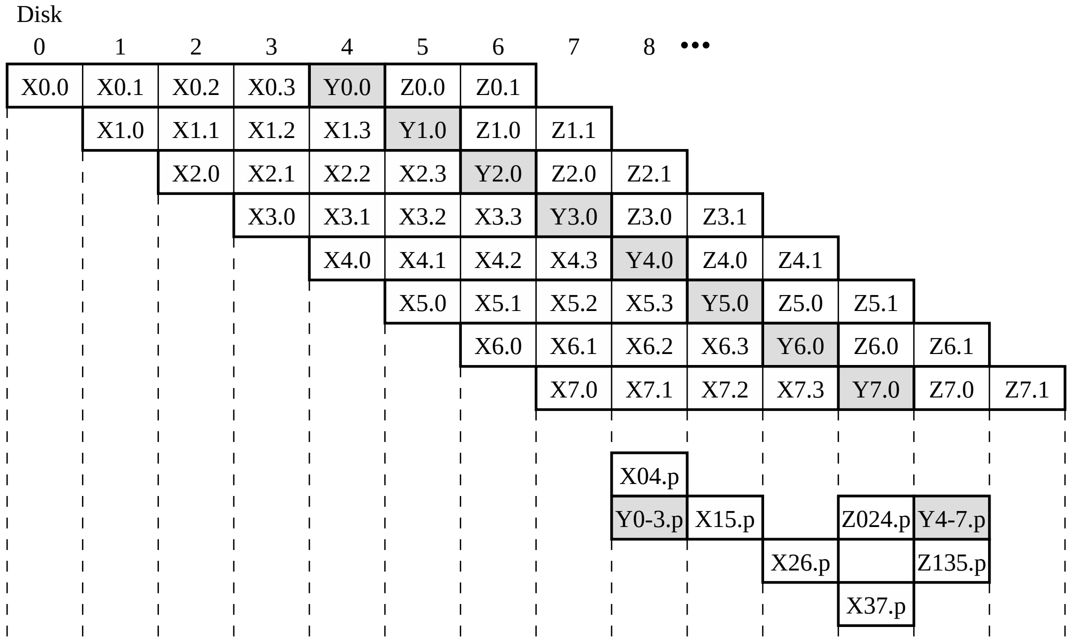


Figure 12: Non-overlapping Subobjects in a Parity Group

a subobject, either due to a failed disk or due to a shift by an object on the left, we make up for it by reading the next available fragment to the right that is in the same parity group, but not in the same subobject; in addition we never read anything twice.

The probability of a catastrophic failure increases as we use more and more fragments per parity group. Specifically, this increases the probability of data loss, which occurs when two disks fail in the same parity group; therefore, the wider the parity group, the greater is the probability of a second failure resulting in data loss. In the example of Figure 12, the largest parity group is 9 disks wide; hence, if two failures occur within 9 disks of each other, data might be lost. Note also, that in this example by time $t+3$ we are able to deliver $Y0-Y3$, Y's first parity group; by $t+7$ we are able to deliver $X0-X7$, X's first four parity groups, and by $t+5$ we are able to deliver $Z0-Z5$, Z's first two parity groups. Again, in order to prevent hiccups we would have to sufficiently offset the delivery of each object and provide buffer space, as we did in previous schemes. In this example the system continues to operate in the shifted mode even after the failed disk is no longer required to deliver objects. This is due to the fact that some data was read out of turn and we have to continue operating in shifted mode until we get back "in sync".

5 Comparison of Schemes

In this section we compare the schemes presented in Sections 3 and 4, using the following metrics as the basis of comparison. As defined earlier, the reliability of a system includes: a) its susceptibility to *degradation of service* and b) the probability of *data loss*. The costs

Time							
t	X0.0	X0.1	X0.2	X0.3	Y1.0	Z2.0	Z0.1
t+1	X1.0	X1.1	X1.2	X5.0	Y2.0	Z3.0	Z1.1
t+2	X2.0	X2.1	X6.0	X2.3	Y3.0	Z4.0	Z2.1
t+3	X3.0	X7.0	X3.2	X3.3	Y0-3.p	Z5.0	Z3.1
t+4	X04.p	X4.1	X4.2	X4.3	Y5.0	Z024.p	Z4.1
t+5	X15.p	X5.1	X5.2	X5.3	Y6.0	Z135.p	Z5.1
t+6	X26.p	X6.1	X6.2	X6.3	Y7.0	Z8.0	Z6.1
t+7	X37.p	X7.1	X7.2	X7.3	Y4-7.p	Z9.0	Z7.1
					⋮		

Figure 13: Delivery of Objects

of providing reliability fall into to one of three categories (we consider the additional cost for providing reliability, i.e., in addition to the cost of a system without redundant information):

- *disk storage*: the amount of disk storage that must be dedicated to redundancy, e.g., parity information, which can not be used to store actual data
- *bandwidth*: the amount of bandwidth that must be dedicated to redundancy, e.g., for transmitting parity, which can not be used for transmitting actual data
- *buffer space*: the amount of memory needed to provide buffering for support of a redundancy scheme, e.g., for storing some portion of a parity group until parity computation is possible

In general, the costs are not independent of each other. For instance, in certain parity placement schemes (see Section 3.1), a penalty in storage is accompanied by a penalty in bandwidth. Or, as will become evident later in this section, it is often possible to tradeoff disk storage cost for buffering cost, and vice versa. (In addition to the quantitative metrics, one should also consider more qualitative factors, such as: a) complexity of scheduling retrieval and delivery of objects during the normal and degraded modes of operation, b) complexity of data layout, and c) complexity of the rebuild process. However, due to lack of space we do not discuss these in detail.)

In order to make comparisons between the various schemes, let us first review our goals, and the steps we took to achieve them. We would like to use parity information to improve the reliability of our system. The price for storing parity is measured in terms of disk bandwidth and/or disk storage and a requirement for additional buffer space. The schemes that managed to take advantage of the full bandwidth capability of the system did so at the price of reliability and availability, as well as the complexity of the system, under both the normal and degraded modes of operation. Of course, none of the schemes managed to take advantage of the full storage capability of the system, but in general, the mechanism for improving storage efficiency was increasing the parity group size. This led to a need for buffer space and, in general, more main memory buffer space was required to obtain higher levels of disk storage efficiency. As we increase the parity group size, the performance of the system improves, since the less space is used for parity, the more can be used for storing objects. However the reliability of the system degrades, since larger parity groups are more susceptible to a second, catastrophic, failure. Hence, the tradeoff we must consider is between performance and reliability. We should also note that a need for buffer space affects the cost of our system; as we improve the storage efficiency, and hence the cost of the disk subsystem, we might increase the buffering cost. For the class of *buffering* schemes, which are schemes that deal with large parity groups by reading and buffering small pieces until an entire parity group is collected[13], increasing the parity group size, decreases the cost of storing the parity information, but it also increases the amount and hence the cost of buffer space needed to support such large parity groups. We will show that after a careful consideration of both disk and buffer space costs it is possible to approximate a configuration that will yield the best combination of reliability and performance for a given total cost constraint.

First, consider simple striping and three (of the four) schemes presented in that context, namely the naive, the improved-bw, and the improved-bw-storage schemes. The storage and bandwidth efficiency of the naive scheme depend on the size of the cluster, the larger the cluster, the greater the storage and bandwidth efficiency. The improved-bw scheme utilizes the full bandwidth of a system, but it always uses only half of the available storage for actual data, independent of the cluster size. The improved-bw-storage scheme also utilizes the entire bandwidth of a system, and its storage efficiency depends on the cluster size. Since neither of these three schemes requires buffer space, it is more informative to consider how disk space and bandwidth efficiency change as a function of cluster size. These graphs are illustrated in Figures 14(a) and 14(b), which indicate that the improved-bw-storage scheme is best in both metrics; however, as mentioned in Section 3, the naive scheme has the highest degree of reliability and availability. It is also important to keep in mind that as the cluster size increases, the reliability of a system (under any of these schemes) decreases. Hence, performance gains must be balanced against reliability losses when considering a large parity group size.

The tradeoff between performance and reliability is similar in the context of staggered striping. Thus, we move on to the discussion of buffering costs, where the following discussion applies to buffering schemes which group consecutive subobjects into parity groups, i.e., in our case the multi-clusters-per-group and the large-stride schemes (using simple and staggered striping, respectively). As the parity group size increases, the amount of storage

[13]Non-buffering schemes, other than the ones that are limited to small parity groups, e.g., parity-per-subobject scheme, are not presented in this report due to lack of space.

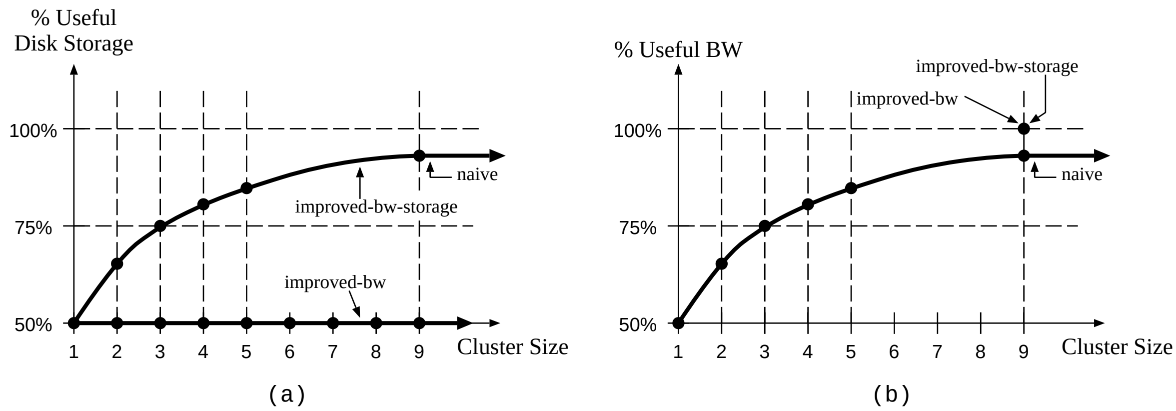


Figure 14: Costs for Simple Striping Schemes

needed for parity information decreases and the size of buffers needed to support such large parity groups increases. Assuming that cost is a constraint in our system it is important to consider the size and cost of the buffers and compare these against the savings in disk space. For the purpose of illustration we consider a simple system with a constant bandwidth requirement. The following simple equations can be used to compute the cost of disk and buffer storage:

$$C_M = C_b * d * D * f$$

$$C_D = C_f * \frac{D * S}{d * k + 1}$$

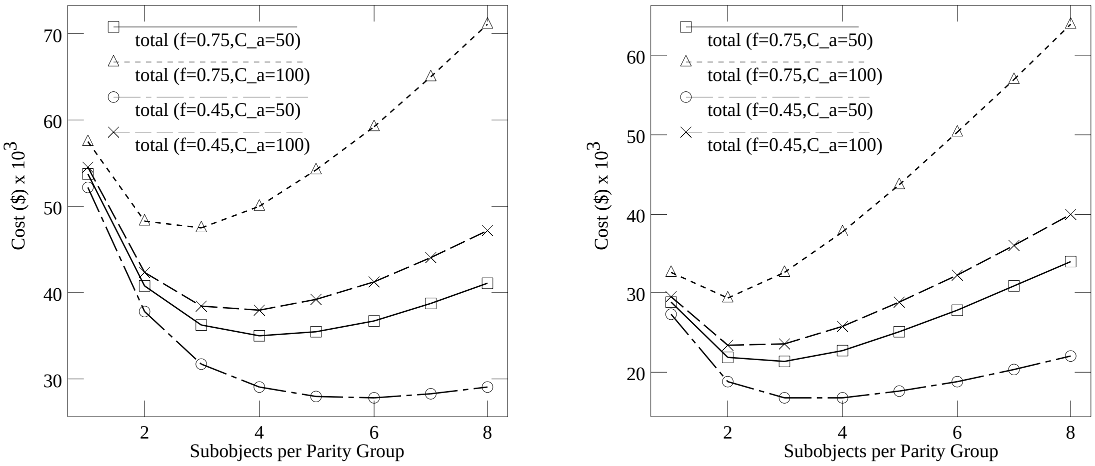
where all the parameters are illustrated in Figure 15. Note that the parity group size is

D = # of disks	C_b = cost of memory (\$/MB) (use 100, 50)
S = storage space per disk (use 1GB)	C_f = cost of disk space (\$/MB) = 1
f = fragment size (use 0.75MB, 0.45MB)	C_a= difference in costs = C_b / C_f
k = # fragments / subobject	C_M = total memory cost
d = # of subobjects in a parity group	C_D = total disk space cost

Figure 15: Parameters

a function of the number of subobjects per parity group and the number of fragments in a subobject. The buffer cost is a function of the number of subobjects in a parity group, fragment size, and the number of disks in the system; the disk cost is a function of the total storage space and the parity group size. Hence, the buffer cost increases with parity group size, and the disk cost decreases. At first the disk cost dominates, and so the total cost decreases as the parity group size increases. However, after a certain group size, the buffer cost begins to dominate, and the total cost goes back up as the number of subobjects per parity group increases further.

Consider first a system with degree of declustering equal to 1, which should need large parity groups in order to achieve good storage efficiency. The total cost for this system, as a function of the number of subobjects in a parity group, is illustrated in Figure 16(a), where the price of memory and the size of fragments are varied. Note that only with cheap memory



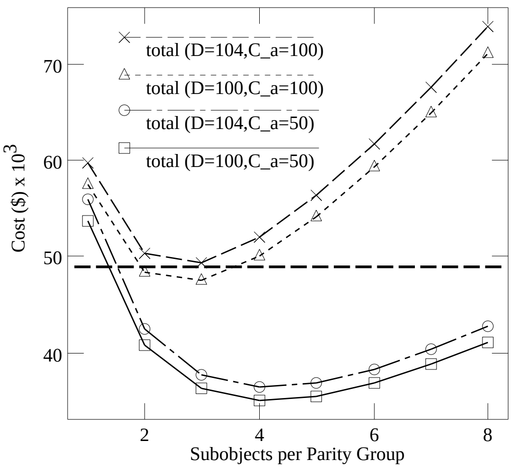
(a) Total Storage Cost (D=100, k=1)

(b) Total Storage Cost (D=100, k=3)

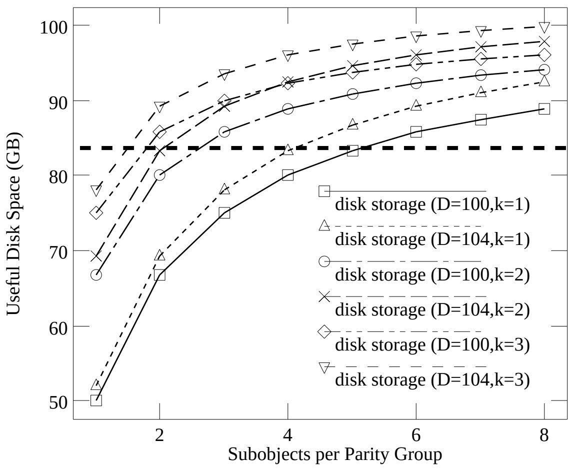
Figure 16: Total Storage Cost

and small fragments would it be cost effective to use a very large number of subobjects per parity groups. For instance, the minimum cost, for a system with 1:100 disk to buffer cost ratio and 0.75MB fragments, is achieved at 3 subobjects per parity group; this indicates that it might not be desirable to construct large parity groups for objects with low bandwidth requirements, since the savings in disk storage can be more than offset by the cost of buffer space. A similar cost graph, but for a system with the degree of declustering equal to 3, is illustrated in Figures 16(b). We should note that, in Figure 16(b), a system with 1:100 disk to buffer cost ratio and 0.45MB fragments requires 2 subobjects per parity group to achieve minimum cost, as opposed to the same system in Figure 16(a) which requires 4 subobjects per parity group. As the bandwidth requirement is increased, the cost the storage overhead and hence the total cost are decreased, and therefore, the minimum cost is achieved at a smaller number of subobjects per parity group.

In addition to being costly, parity groups with many subobjects result in poor reliability. Given a total cost constraint, the money might be better spent on more disks rather than more memory. For instance, Figure 17(a) depicts a cost comparison between a system with 100 disks and a system with 104 disks, all other things being equal. Note that a system with 104 disks at a parity group of 4 subobjects has a lower cost than a system with 100 disks at a parity group of 5 subobjects, and in addition a higher reliability. Therefore, in this case spending money on disks rather than memory pays off in reliability. But, we should also consider the performance aspect of these two systems, which depends on available bandwidth and disk storage. Of course, the system with 104 disks has more bandwidth, but, as illustrated in Figure 17(b), which depicts the amount of storage available for actual data, it also has more useful disk storage, i.e., it can store more objects. Therefore, the system with 104 disks (and 4 subobjects per parity group) should have better reliability and better performance, than the system with 100 disks (and 5 subobjects per parity group), at a *lower* cost.



(a) Storage Cost Comparison (k=1,f=0.75)



(b) Amount of Useful Storage

Figure 17: Storage Comparison

The last scheme presented in Section 4, the non-overlapping scheme, requires even more buffer space than the schemes discussed so far. The total memory cost equation for this scheme would have to be modified as follows:

$$C_M = C_b * d * D * f * \lceil \frac{k}{s} \rceil$$

where s is the stride. Since this scheme does not read subobjects of the same parity group in consecutive time slots, each request holds on to its buffers longer than in previous schemes, which is equivalent to requiring more buffer space. Due to lack of space, we do not show graphs similar to Figures 16 and 17, but the basic conclusion is that this scheme requires more buffer space (than previous schemes), which means that the memory cost will begin to dominate (see discussion above) at a smaller number of subobjects per parity group; hence, this scheme results in smaller parity groups.

6 Conclusions

In summary, we presented several parity schemes, both in the context of simple and staggered striping, for providing reliability and availability in a multimedia on-demand server, at the cost of storage overhead, wasted bandwidth and buffer space (not necessarily all three at once). We have shown that improvements in reliability, which depend on the amount of redundant information stored and on how this information is placed on disks, must be balanced against degradation in performance, due to storage overhead and loss of bandwidth. In addition, we have shown that the cost of buffer space must also be taken into consideration when designing a redundancy scheme, since savings in disk storage, resulting from the use of large parity groups, might be more than offset by the cost of buffer space necessary to support them. Work on further analysis of these schemes as well as on more sophisticated parity schemes is currently in progress.

References

[1] Steven Berson, Shahram Ghandeharizadeh, Richard R. Muntz, and Xiangyu Ju. Staggered striping in multimedia information systems. *Accepted for 1994 SIGMOD*, 1994.

[2] D. Bitton and J. Gray. Disk shadowing. *VLDB*, pages 331–338, 1988.

[3] D. DeWitt, S. Ghandeharizadeh, D. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[4] A. L. Drapeau and R. Katz. Striped Tape Arrays. In *Proc. of the 12th IEEE Symposium on Mass Storage Systems*, pages 257–265, Monterey, California, April 1993.

[5] S. Ghandeharizadeh and L. Ramos. Continuous retrieval of multimedia data using parallelism. *IEEE Transactions on Knowledge and Data Engineering*, 1(2), August 1993.

[6] S. Ghandeharizadeh, L. Ramos, Z. Asad, and W. Qureshi. Object placement in parallel hypermedia systems. *In Proc. of VLDB*, 1991.

[7] S. Ghandeharizadeh and C. Shahabi. Management of physical replicas in parallel multimedia information systems. *In Proc. of the 1993 Foundations of Data Organization and Algorithms (FODO) Conference*, October 1993.

[8] Richard R. Muntz and John C.S. Lui. Performance analysis of disk arrays under failure. *VLDB Conference*, pages 162–173, 1990.

[9] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). *ACM SIGMOD Conference*, pages 109–116, 1988.

[10] Teradata. *DBC/1012 database computer system manual release 2.0*, 1985.