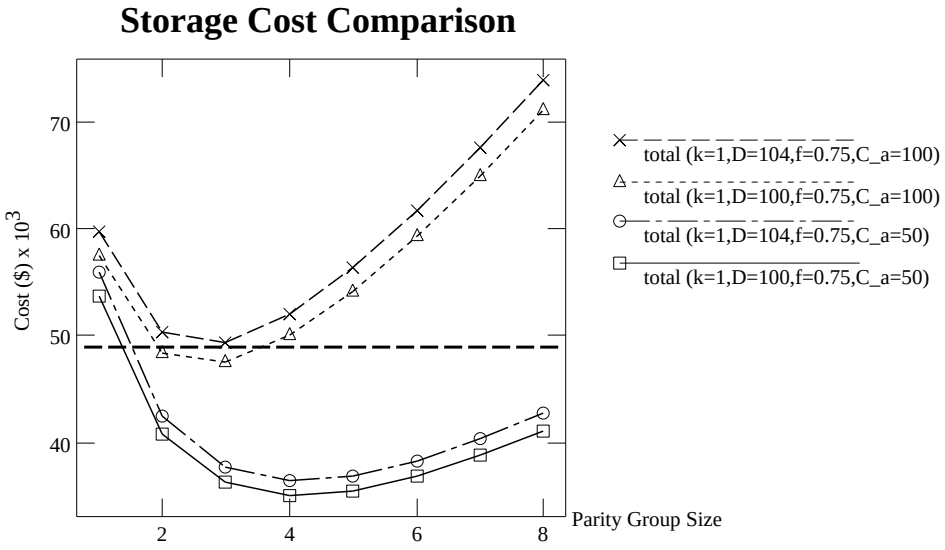

Figure 36: Storage Cost Comparison

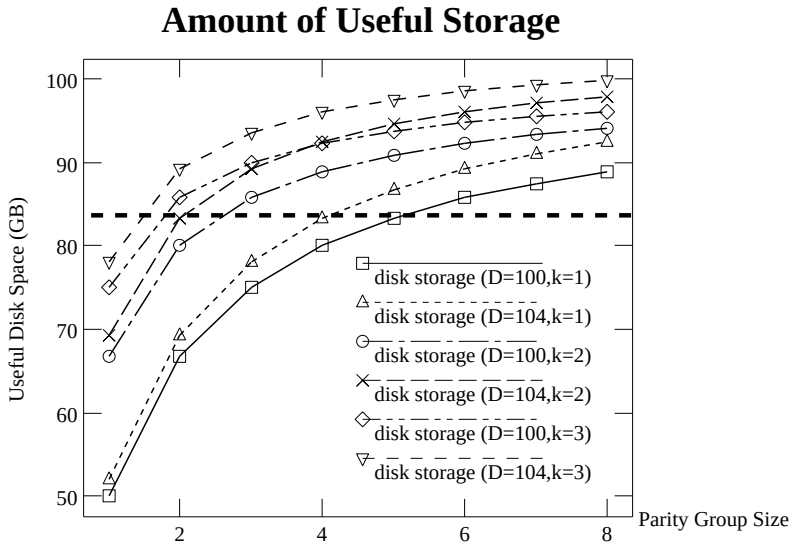

Figure 37: Comparison of Useful Storage

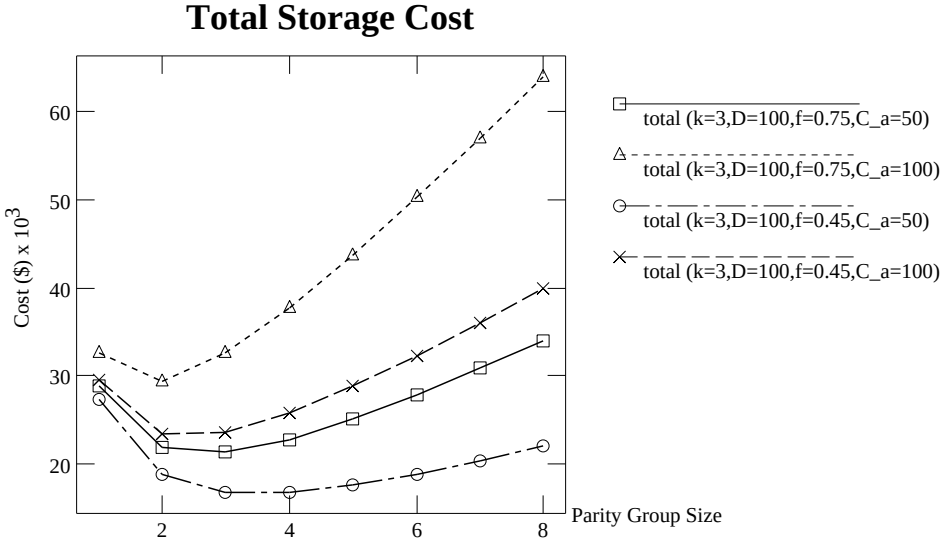

Figure 35: Total Storage Cost 3

[8] 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.

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

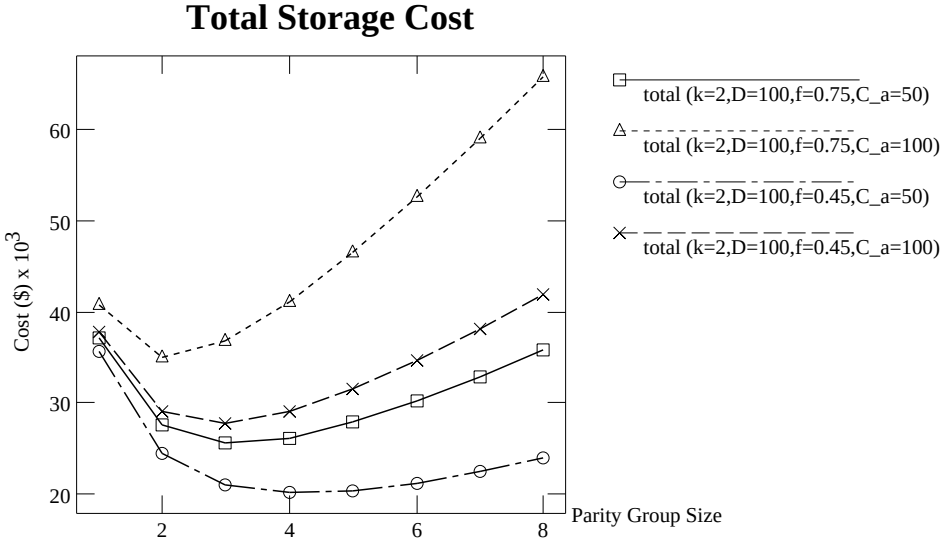

Figure 34: Total Storage Cost 2

References

[1] Steven Berson, Shahram Ghandeharizadeh, Richard R. Muntz, and Xiangyu Ju. Staggered striping in multimedia information systems. *Submitted to 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] S. Ghandeharizadeh and L. Ramos. Continuous retrieval of multimedia data using parallelism. *IEEE Transactions on Knowledge and Data Engineering*, 1(2), August 1993.

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

[6] 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.

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

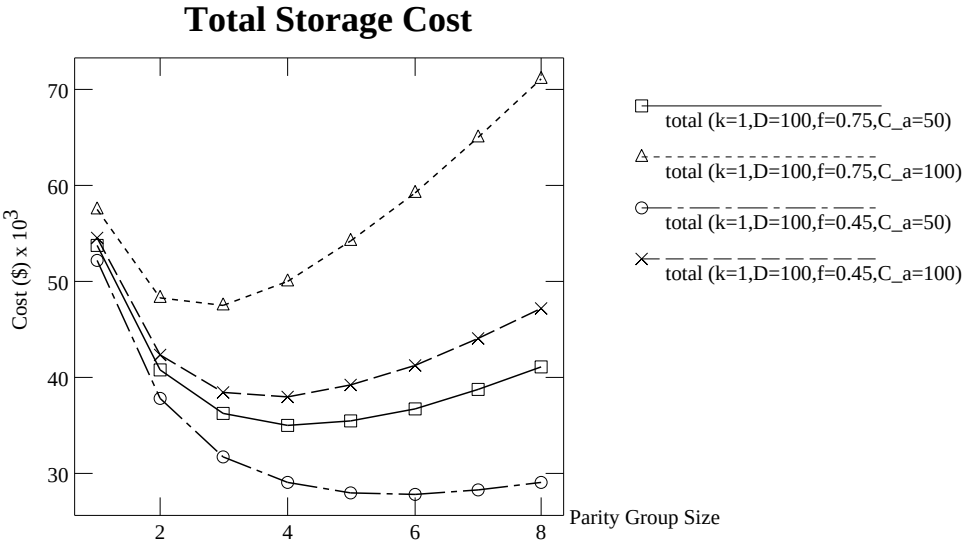

Figure 33: Total Storage Cost 1

group of 2.

In addition to being costly, large parity groups 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 36 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 has a lower cost than a system with 100 disks at a parity group of 5, 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 37, 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 should have better reliability and better performance, than the system with 100 disks, at a lower cost.

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

where all the parameters are illustrated in Figure 32. Note that the parity group size

D = # of disks
S = storage space per disk (use 1GB)
f = fragment size (use 0.75MB, 0.45MB)

k = # fragments / subobject
d = # of subobjects in a parity group
s = # of subobjects on disks
B = total buffer space requirement
F = total disk space requirement
d* = crossover point

C_b = cost of memory ($/MB) (use 100,50)
C_f = cost of disk space ($/MB) = 1
C_a= difference in costs = C_b/C_f
C_M = total memory cost
C_D = total disk space cost
C_total = total cost = C_M + C_D

Figure 32: Parameters

is defined as the number of subobjects per parity group. The buffer cost is a function of parity group size, 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 parity group size increases further.

Consider first a system with a constant bandwidth requirement of 1, which should need large parity groups in order to achieve good storage efficiency. The total cost for this system is illustrated in Figure 33, where the price of memory and the size of fragments are varied. Note that only with cheap memory and small fragments would it be cost effective to use very large 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 a parity group of 3. Similar cost graphs are illustrated in Figures 34 and 35 for systems with bandwidth requirement of 2 and 3, respectively. Of course, as the bandwidth requirement is increased, the cost is decreased; furthermore, the minimum cost is achieved at smaller parity group sizes. For instance, in Figure 33, a system with $1:100$ disk to buffer cost ratio and 0.45MB fragments requires a parity group of 4 to achieve minimum cost, but the same system in Figure 35 only requires a parity

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 31(a) and 31(b). It is important to keep in mind that as the cluster size increases, the

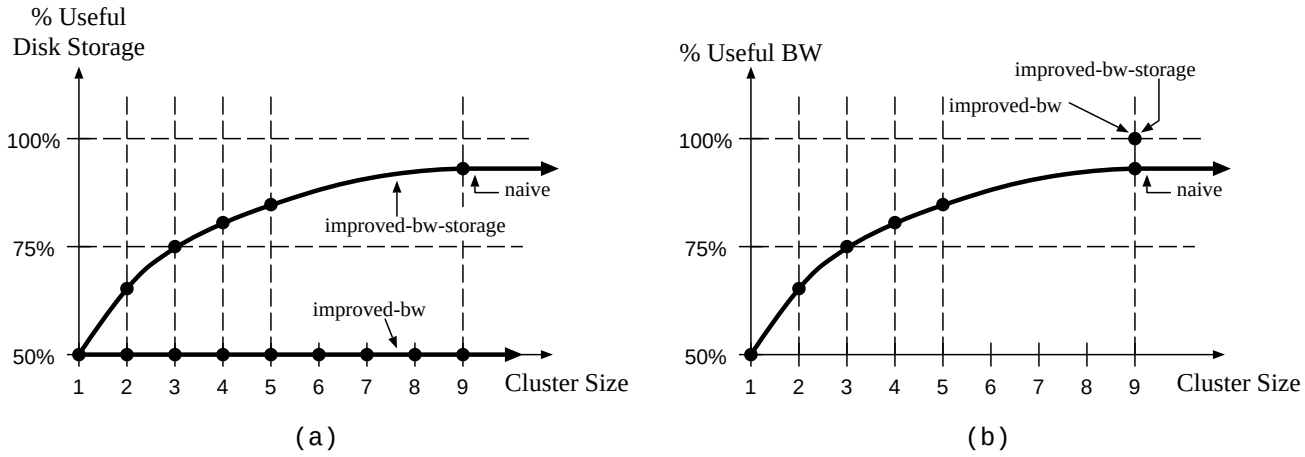

Figure 31: Costs for Simple Striping Schemes

reliability of a system 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, where we developed two basic types of schemes: 1) buffering schemes and 2) non-buffering schemes. Buffering schemes deal with large parity groups by reading and buffering small pieces until an entire parity group is collected. The non-buffering schemes read the entire parity group at once, which can result in fragmentation problems (see Section 4). We first concentrate on buffering schemes, and return to discussing non-buffering schemes later in this section.

The following discussion applies to all buffering schemes in general. As the parity group size increases, the amount of storage 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[5], 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 (for a class of schemes that buffer data and use several subobjects per parity group):

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

[5]Otherwise we could just buy more disks.

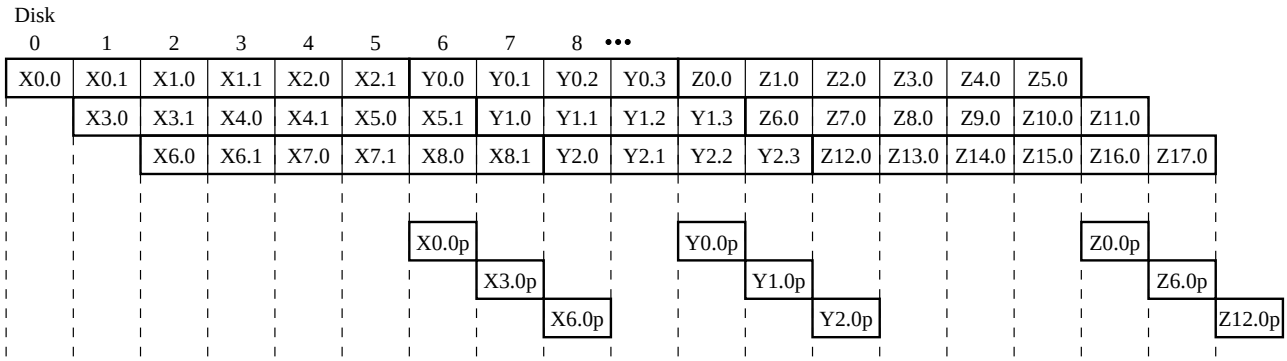

Figure 30: Variable Group Sizes

additional buffer space. Most of the schemes presented in this paper managed to take advantage of the full bandwidth capability of the system at the price of: a) reliability, i.e., degradation of service and b) 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 the general mechanism for improving storage efficiency was increasing the parity group size. For most schemes, this led to a need for buffer space, and in general, the greater was the disk storage efficiency the more main memory buffer space was required to maintain that level of efficiency. As we increase the parity group size, the performance of our system improves, since less space is used for parity, the more can be used for storing objects. However the reliability of our 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 are improving the storage efficiency, and hence the cost of the disk subsystem, we might be increasing the buffering cost. For a certain class of schemes presented in Section 4, namely, the "buffering" schemes, 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 construct a configuration that will give one the best combination of reliability and performance for a given total cost constraint.

First, consider simple striping and the three basic 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 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 it's storage

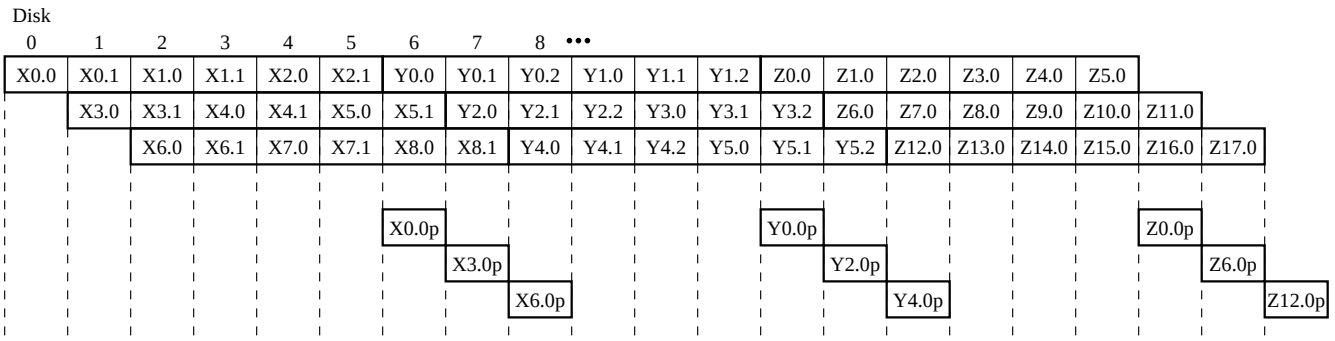

Figure 28: Constant Group Size – All Fits

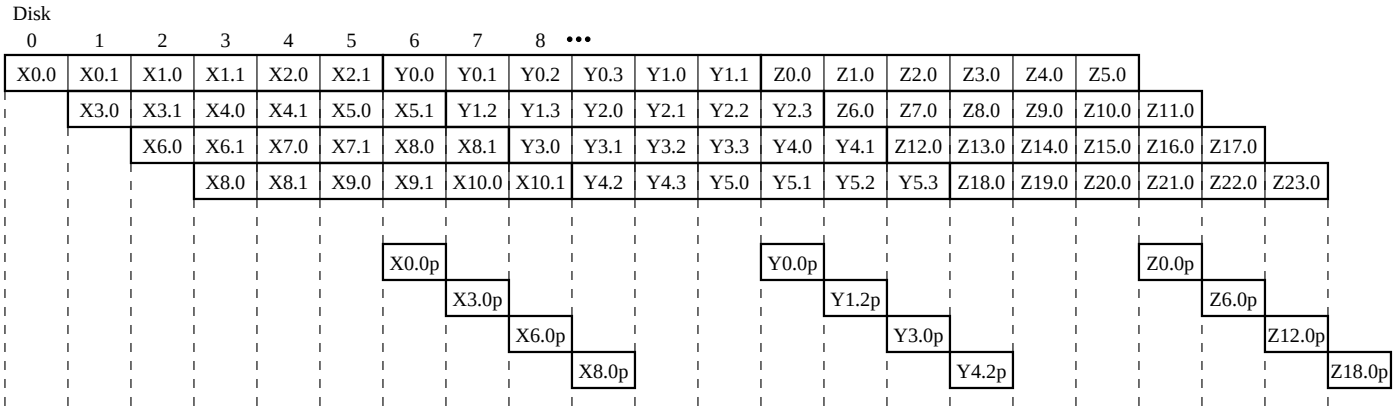

Figure 29: Constant Group Size – Split Subobjects

- complexity of scheduling retrieval and delivery of objects during the normal and degraded modes of operation
- complexity of data layout
- complexity of the rebuild process

Note that 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 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 information overhead is measured in disk bandwidth and/or disk storage and a requirement for

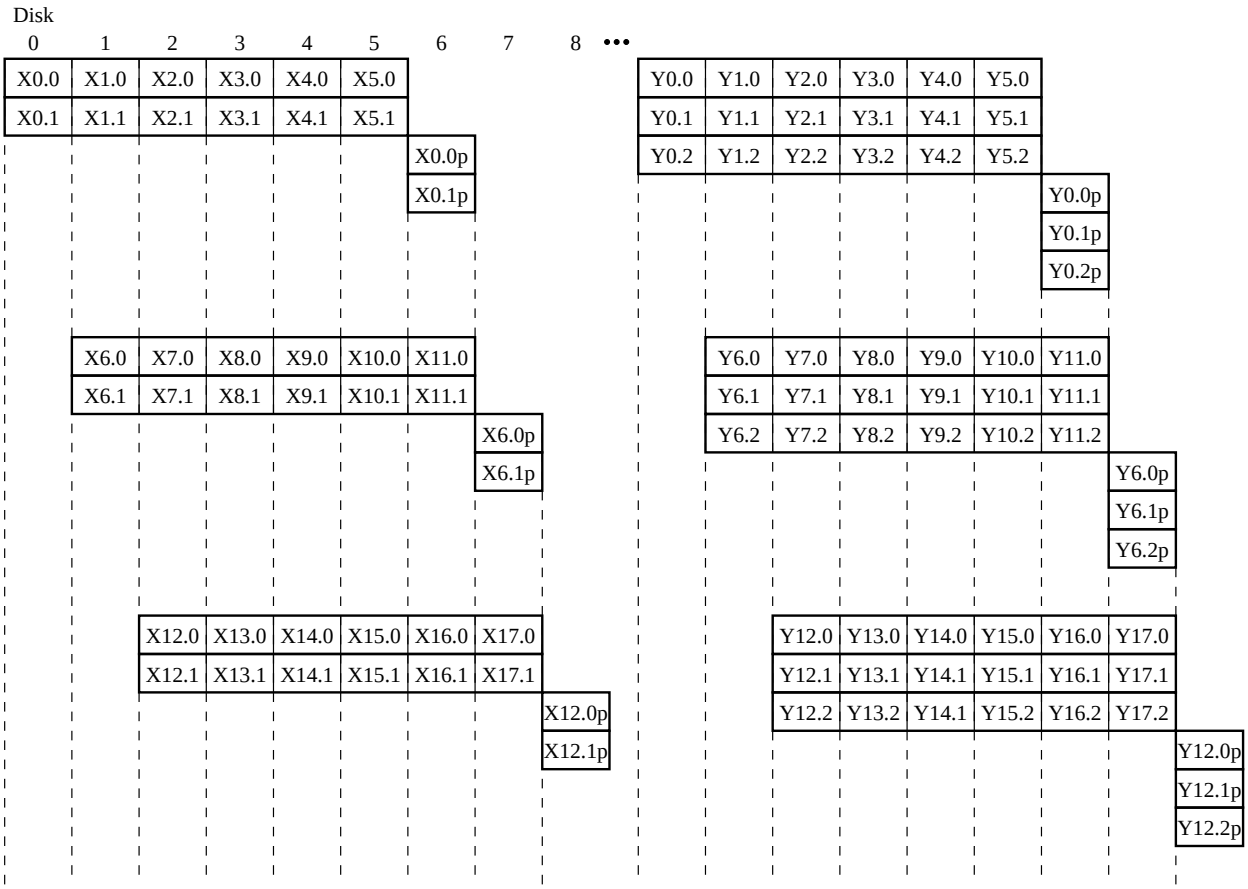

Figure 27: Constant Group Size

redundant information):

- *disk storage*: an amount of disk storage that must be dedicated to redundancy, e.g., parity information, which can not be used to store actual data
- *bandwidth*: an 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*: an 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 addition to the quantitative metrics, we must also consider more qualitative factors, such as:

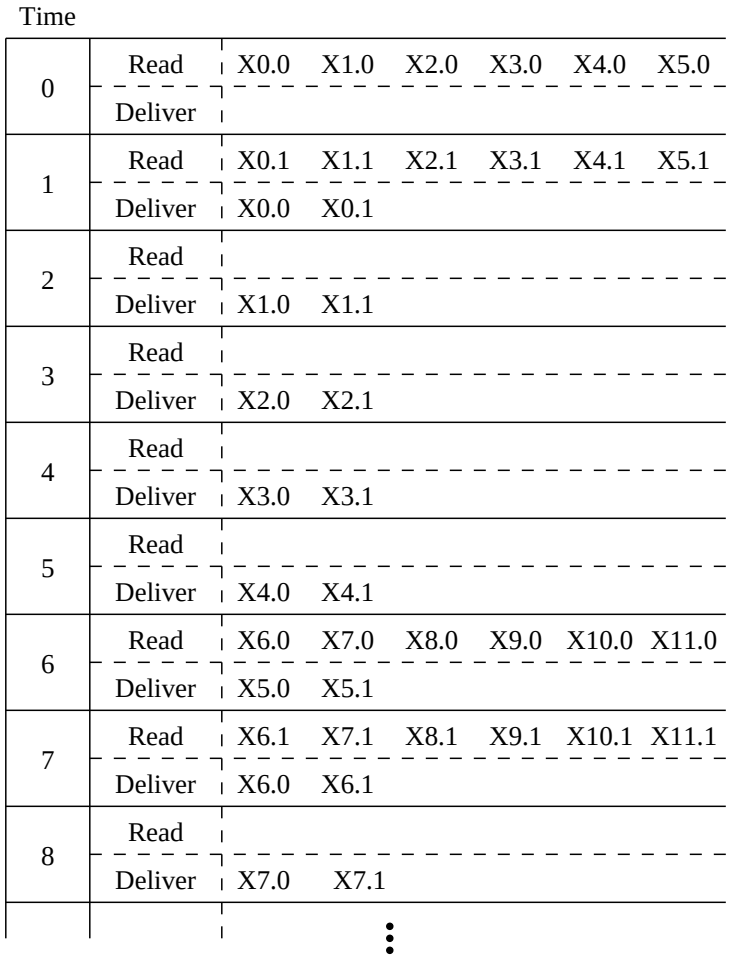

Figure 26: Schedule and Delivery

5 Comparison of Schemes

In this section we compare the multitude of schemes presented in Sections 3 and 4, using the following metrics as the basis of comparison, and make recommendation as to which are useful and under what conditions and constraints. As defined earlier, the reliability of a system includes:

- its susceptibility to *degradation of service*
- the probability of *data loss*

The costs of providing reliability fall into to one of three categories (we consider the additional cost for providing reliability, in addition to the cost of a system without

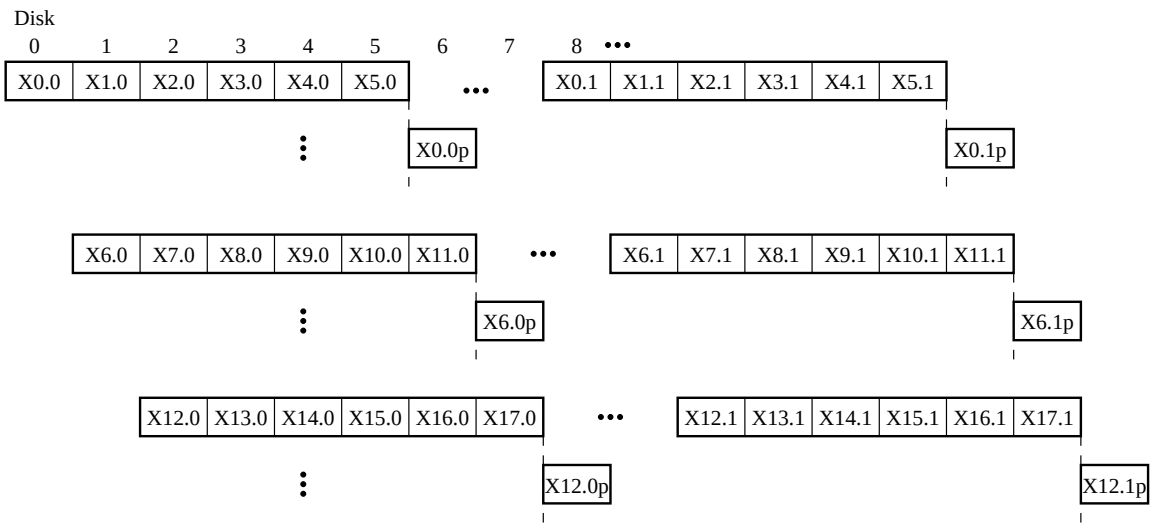

Figure 25: New Data and Parity Placement

Instead of "reshuffling" the data, as we did in the previous scheme, we could still lay it out according to subobjects by packing several subobjects into one parity group and laying out the entire parity group on consecutive disks. The data and parity placement for this scheme, with a constant parity group size of 6, is illustrated in the example of Figure 28. In this example we have three objects, X with subobjects of size 2, Y with subobjects of size 3, and Z with subobjects of size 1. Each parity group consists of 6 object fragments plus a parity fragment. Therefore, each parity group in this example can have either three X subobject, or two Y subobjects, or six Z subobjects. To schedule any one of the objects, we would need a slot of six disks. When delivering objects, even in normal mode of operation, we could deliver the first subobject and buffer the rest. As in the previous schemes, we need to insert idle periods in order to control buffer growth; this increasing the complexity of scheduling. Under failure, we would have to perform the same shift to the right as in a system that uses the "parity-per-subobject" scheme with a constant bandwidth requirement.

Note that in Figure 28 the subobjects just happened to fit nicely into parity groups of size 6; this of course does not have to happen. We can deal with this problem in one of two ways. Either we can insist on a fixed parity group size, in which case we would have to split subobjects between different parity groups, which is illustrated in Figure 29, or we can insist on having whole subobjects in each parity group, in which case we would have to deal with variable size parity groups, which is illustrated in Figure 30. Either of the two results in fragmentation problems, although perhaps of different types, and complications in scheduling.

Time													
t	Read	X0.0	X1.0	X2.0	X3.0	X4.0	X5.0	X0.1	X1.1	X2.1	X3.1	X4.1	X5.1
	Deliver	X0.0						X0.1					
t+1	Read												
	Deliver		X1.0						X1.1				
t+2	Read												
	Deliver			X2.0						X2.1			
t+3	Read												
	Deliver				X3.0						X3.1		
t+4	Read												
	Deliver					X4.0						X4.1	
t+5	Read												
	Deliver						X5.0						X5.1
t+6	Read	X6.0	X7.0	X8.0	X9.0	X10.0	X11.0	X6.1	X7.1	X8.1	X9.1	X10.1	X11.1
	Deliver	X6.0						X6.1					
t+7	Read												
	Deliver		X7.0						X7.1				
					⋮								

Figure 24: Schedule and Delivery

described in [1]. But, even though it has a very regular "space schedule", it would not have a fixed "time schedule" in a system with variable bandwidth requirements; the only requirement is to read during k out of every d time intervals, which makes the scheduling more complex and could result in another type of a fragmentation problem, namely a time fragmentation problem.

An alternative to the above scheme is to place data on disks in such a way that we do not have to stagger over to the next set of disks for d time slots during which we could read d objects. Again, for an object with bandwidth requirement of k, we only read during k of the d time slots. Then we stagger over to the next set of disks, and read the next d objects while delivering the previous d. This is illustrated in Figure 27. In this example, we can read $X0 - X5$ during the first d time slots, which takes 2 time slots to read and still has 4 time slots idle, and we can also read $Y0 - Y5$ which takes 3 time slots to read and still has 3 time slots idle. The difficulty, of course, is in trying to schedule something during those idle time slots, i.e., in this example finding objects that reside on the same set of disks as X and Y and require 4 and 3 busy slots, respectively.

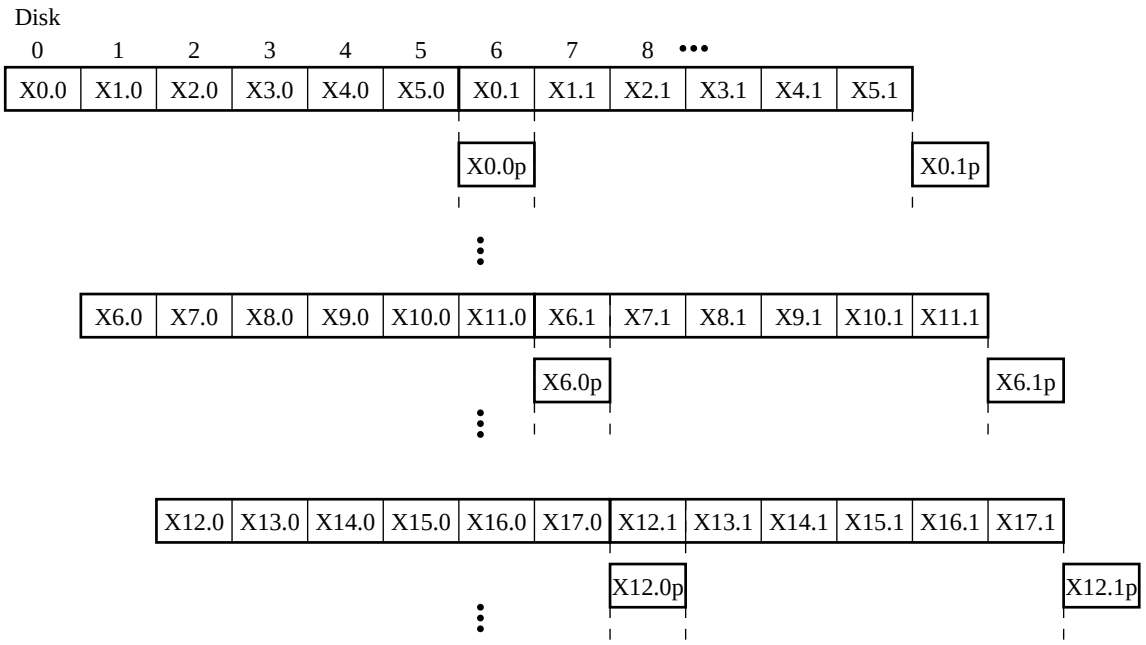

Figure 23: New Data and Parity Placement

i.e., every object reads only during one out of every d time intervals, there is no real additional complexity.

The scheme in Figure 23 has a fragmentation problem, due to requiring a large number of disks per request. Another way to lay out the data for a similar scheme is illustrated in Figure 25. Here we still maintain a constant parity group size d and a constant bandwidth requirement k, but we do not place every k parity groups on consecutive disks, and hence do not need to schedule $k * d$ disks to be read simultaneously. We only read d disks at a time, i.e., one parity group at a time. In other words, out of every d time intervals, we only read k times; the rest of the $d - k$ time intervals are idle. Hence, in the example of Figure 25, we must read during 2 out of every 6 time intervals and the other 4 are free to be used by other objects. Scheduling and delivery for this example, under normal operation, is illustrated in Figure 26. During the first 6 time intervals, we read and buffer the first 2 parity groups, which amounts to reading the first 6 subobjects. In the next 6 time intervals we deliver the six subobjects read in the previous time interval (one per interval), and in addition read the next 2 parity groups, and so on.

The advantage of this scheme over the previous one is that there is no need to schedule $k*d$ disks at a time, but rather only d disks at a time. Hence, not only does a request need smaller slots, but also, all the slots are of the same width (since the parity group size is constant); this scheme eliminates the space fragmentation problem,

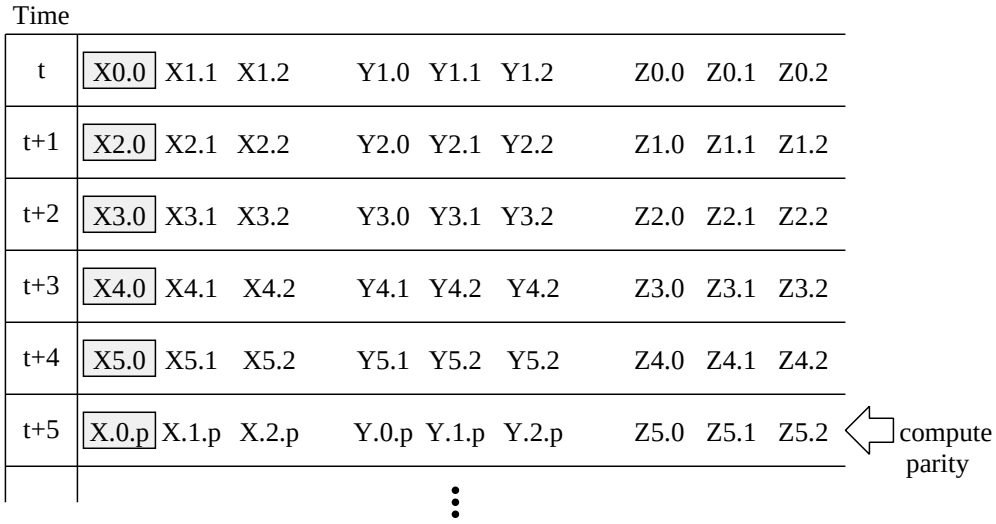

Figure 22: Reading under Failure in Orthogonal Scheme

as follows:

- schedule a slot which is $k * d$ disks wide
- during a single time slot read d subobjects; deliver one of them and keep the other $d-1$ in the buffers
- in the next $d-1$ time slots do not read anything but deliver the $d-1$ subobjects from the buffers, one per time interval
- in the next time slot start all over, i.e., read the next d subobjects

Consider again the example of Figure 23 and suppose that disks $0-11$ are idle at time t. The reading and delivery of X, in normal mode of operation, is illustrated in Figure 24, where at time t we read subobjects $X0-X5$, but only deliver subobject $X0$. In the next 5 time intervals we deliver the rest of the subobjects, $X1-X5$, one per interval; at time 6 we read subobjects $X6-X11$, etc. Since in this scheme an entire parity group is read in a single time slot, the same shift to the right as in the "parity-per-subobject" scheme will work under failure. Therefore, an advantage of this scheme is its simpler behavior under failure. A disadvantage is an increased potential for fragmentation, since it requires much wider space slots than all the previous schemes.

The idle time intervals can be used for servicing other requests, which adds another dimension to the scheduling problem. However, since the scheduling is very regular,

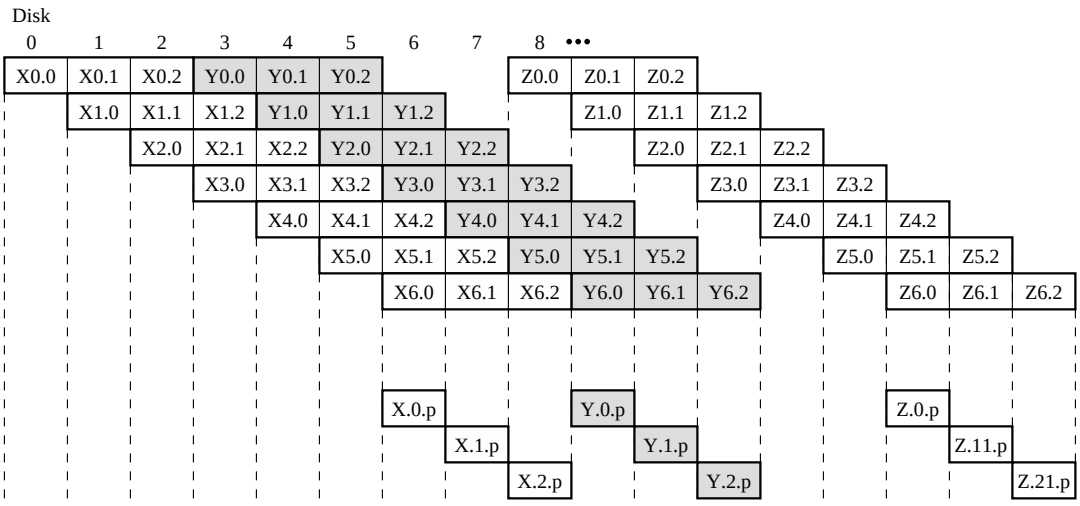

Figure 21: Orthogonal Parity Grouping

and were all read in a *single* time interval. The problem we had with that scheme was that it possibly wasted too much storage for parity when it came to objects with a small degree of declustering, e.g., objects that only required the bandwidth of one disk. In those cases we preferred to have several subobjects in the same parity group. We can take advantage of the simplicity of the parity-per-subobject scheme without paying the storage overhead by changing the way data is placed on disks and the way objects are scheduled for transmission. Instead of using a subobject as a striping unit for data placement, we use the parity group as a striping unit. The scheduling scheme is modified accordingly; instead of scheduling subobjects, we schedule parity groups. We discuss several such such schemes in this section, where a constant parity group size of d and a constant bandwidth requirement of k are assumed.

Consider a scheme where the first k parity groups, which also make up the first d subobjects, are placed on consecutive disks. The starting place of the next k parity groups is determined by the stride size and the parity group size. An example of this *parity-striping* scheme, with $d = 6$ and $k = 2$, is illustrate in Figure 23, where $X0.0, X1.0, X2.0, X3.0, X4.0$ and $X5.0$ make up the first parity group and are placed on the first 6 disks; $X0.1, X1.1, X2.1, X3.1, X4.1$ and $X5.1$ make up the second parity group, and are placed on the next set of 6 disks, and so on; these first 2 parity groups constitute the first 6 subobjects of X. The scheduling of reads and delivery of objects are modified accordingly. At each time interval we read d subobjects, but are only able to deliver one. Therefore, we must buffer the other $d - 1$ subobject, and in addition not read any more subobjects until we are able to clear this backlog, i.e., we must wait for the next $d - 1$ time intervals before reading the next set of parity groups. Therefore, the basic rules for delivery of objects and scheduling of reads are

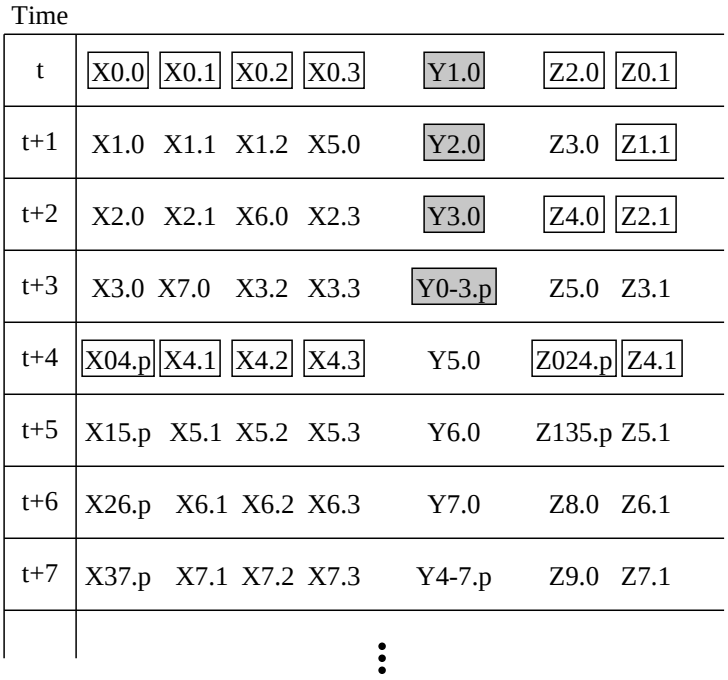

Figure 20: Reading of Objects

first subobject has a cascading effect; the following occurs in the very first time slot. Due to the failure of disk 1, we read $X1.1$ instead of $X0.1$, which makes it impossible to read $X0.2$. Therefore, we make up for $X0.2$ by reading $X1.2$, which makes it impossible to read $Y0.0$. We make up for $Y0.0$ by reading $Y1.0$, which results in reading of $Y1.1$ instead of $Y0.1$, which finally results in reading of $Y1.2$ instead of $Y0.2$. In other words, the entire row that was suppose to be read at time t, starting at the column of failure, is replaced by the row that was suppose to be read at time $t + 1$.

The advantage of this scheme is that we no longer have to worry about overlaps and a proper stride size, i.e., even with a stride size of 1 these fragments can not overlap. In addition, packing objects into parity groups of some constant size is simple, i.e., costs and benefits of reliability are more manageable.

4.3 Alternative Data Placement Schemes

The simplest scheme (so far) has been the parity-per-subobject scheme, discussed at the beginning of this section; in that case we didn't have to worry about multiple dependent failures, the "shift to the right" was simple, etc. The reason things were simpler is because all fragments in the same parity group resided on different disks

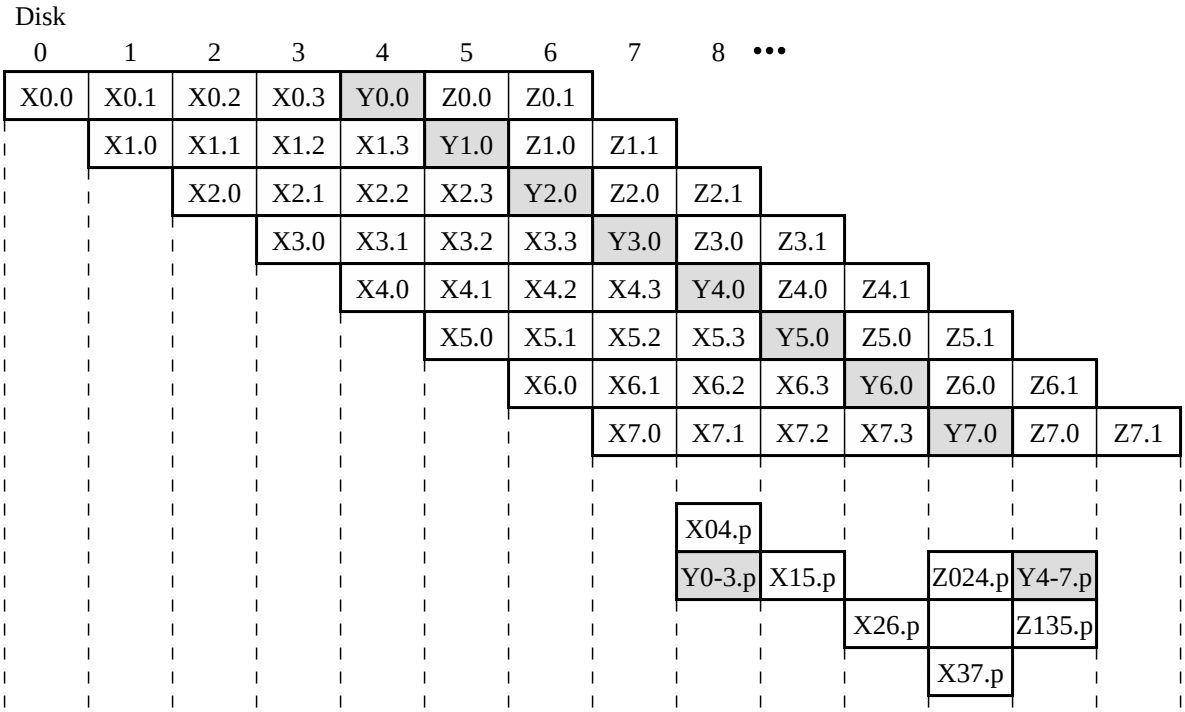

Figure 19: Different Parity Group Sizes

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 synch".

There is a third way to remedy the problem of multiple dependent failures, exhibited by the system of Figure 13. Instead of grouping fragments into parity groups according to subobject layout, as we have done until now, we can group them into parity groups orthogonally to the subobject layout, i.e., we can decouple the bandwidth requirement, which dictates subobject size, from the reliability requirement, which dictates parity group size. This alternative is illustrated in Figure 21. In the scheme of Figure 15 each parity group consisted of 6 fragments belonging to 2 different subobjects. In this new scheme, there are still 6 fragments per parity group, but they are not grouped by subobjects; instead, all 6 fragments come from different subobjects. For instance, in Figure 15 we grouped subobjects $X0$ and $X1$ into the first parity group; hence, fragments $X0.0, X0.1, X0.2, X1.0, X1.1$, and $X1.2$ belonged to one parity group. In the new scheme fragments $X0.0$, $X1.0$, $X2.0$, $X3.0$, $X4.0$, and $X5.0$ belong to one parity group. Under failure, we can use the same rules for shifting to the right as we did in previous schemes. Given the failure of disk 1 in Figure 21 before time t, the schedule for reading of objects X, Y, and Z starting at time t is illustrated in Figure 22. Notice that in this scheme a shift to the right of the

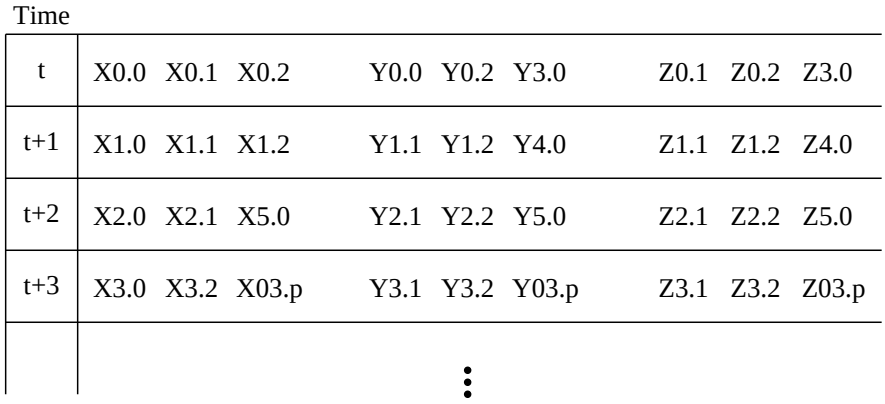

Figure 18: Delivery of Objects

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 17, each parity group is 7 disks wide; hence if two failures occur within 7 disks of each other, data will be lost.

This scheme remains unchanged in the case of variable bandwidth requirements. Figure 19 depicts an example of a system with variable bandwidth requirements and variable number of subobjects per parity group. In this example, there are three objects, X, Y, and Z, with bandwidth requirements of $4, 1$, and 2, respectively. Subobjects from X are packed into parity groups of size 8, i.e., with two subobjects per parity group. Subobjects from Y are packed into parity groups of size 4, i.e., with 4 subobjects per parity group. And, finally, subobjects from Z are packed into parity groups of 6, i.e., with 3 subobjects per parity group.

In case of failure a system with variable bandwidth requirements behaves identically to the one with a constant bandwidth requirement. The schedule, starting at time t, for reading objects in the example of Figure 19, is illustrated in Figure 20; an assumption in this figure is that disk 4 fails before time t. 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. In order to prevent hiccups we would have to sufficiently offset the delivery of each object, as we did in previous schemes. Note that in this example the system continues to operate in the "shifted" mode even after the failed disk is no

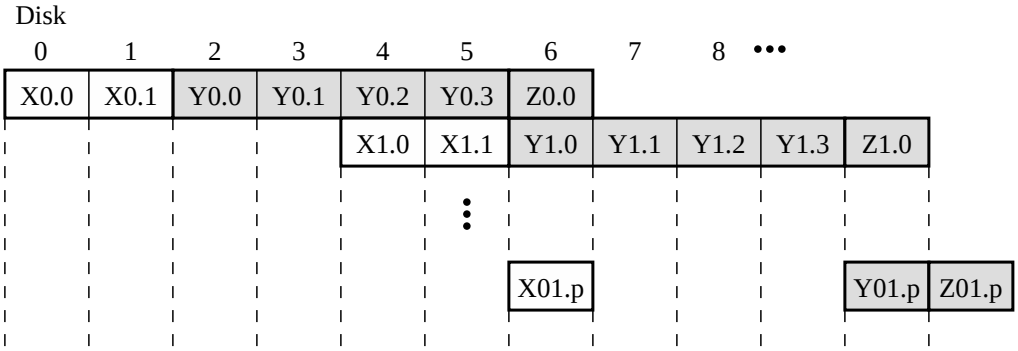

Figure 16: Variable BW – Multi-Subobject Parity Groups

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, but we can do that by grouping non-overlapping subobjects into parity groups (as opposed to increasing the stride size). This scheme is illustrated in Figure 17. In this example, subobjects $X0$

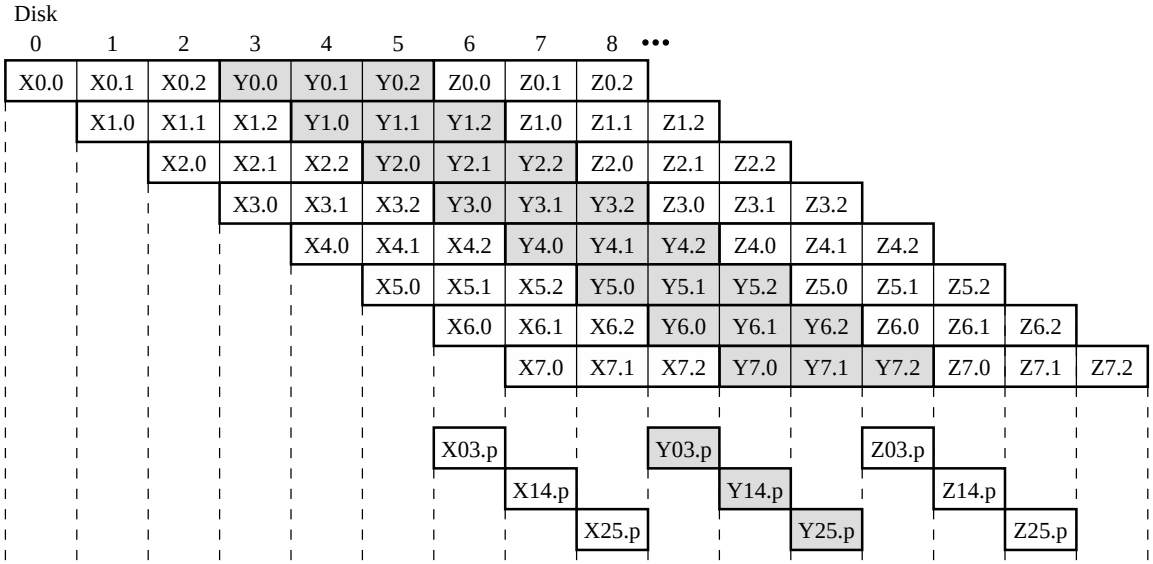

Figure 17: Non-overlapping Subobjects in a Parity Group

and $X3$ are paired in a parity group, and are protected by fragment $X03.p$; similarly $Y0$ and $Y3$ are paired up in a parity group and are protected by $Y03.p$, etc. In this case, if disk 4 fails at time t, we can still deliver X, Y, and Z as illustrated in Figure 18. The same shift to the right, as was described in the previous scheme, works here. Every time there is a fragment missing from 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

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 $t + i$, then at time $t + i$ 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 15 at time t, then X and Y can still be delivered as follows:

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

Given a system of objects with variable bandwidth requirements, we'll be forced to use a stride of size greater than or equal to the largest bandwidth requirement. The example of Figure 16 illustrates this point using three types of objects with bandwidth requirements of $2, 4$, and 1. In this example, Z is a 1-disk-wide object; there are two Z subobjects per parity group, and it is safe to place them on consecutive disks, e.g., if $Z0.0$ resides on disk 6, then $Z1.0$, which belongs to the same parity group, can be safely placed on disk 7. However, Y is a 4-disk-wide object, and hence imposes a condition that the stride be at least equal to 4. Note 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,

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 for 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 15, where the stride is increased to 3. In general, a system can avoid multiple dependent failures

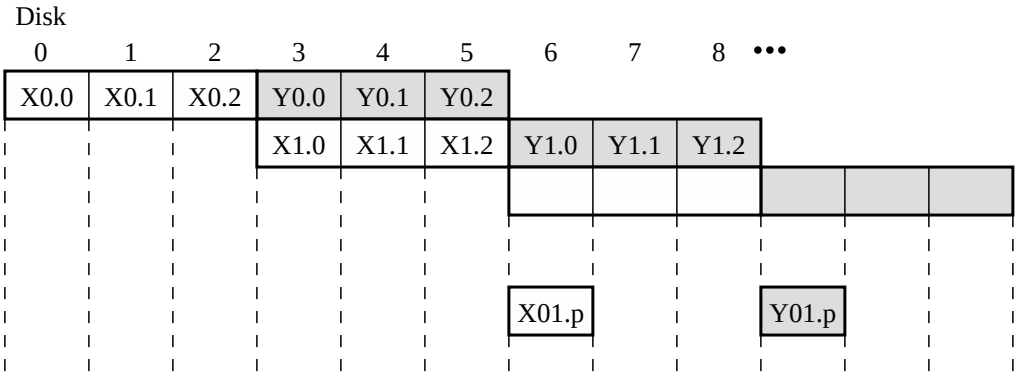

Figure 15: Increased Stride Size

by having a stride that is greater than or equal to the 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 "out of turn", i.e., something that would normally be read in a later time slot. With one subobject per parity group, it should clearly be the parity fragment, but with multiple subobjects per parity group there are usually several choices. For instance, if disk 2 fails in the example of Figure 15, then we could compensate for $X0.2$ by reading any one of

operation, since a failure does not have to occur before we start reading a particular parity group. For instance, if disk 1 in Figure 13 fails at time $t + 1$, we would not be able to deliver $X1$ if we do not buffer $X0$ at time t. Hence, we must buffer each fragment until the entire parity group, to which it belongs, is delivered. As mentioned earlier, we could fix the "hiccup" problem by not starting the delivery of an object until we've read an entire parity group. In case of failure, this would allow us to provide data to the network, from the buffers, while collecting enough information to perform a parity computation. The price we pay for this offset is a small delay in the delivery start time; there is no buffering penalty associated with the offset, since we must buffer in anticipation of failure anyway. An example delivery schedule, for the system of Figure 13 where disk 2 fails at time $t+2$, is illustrated in Figure 14. At time

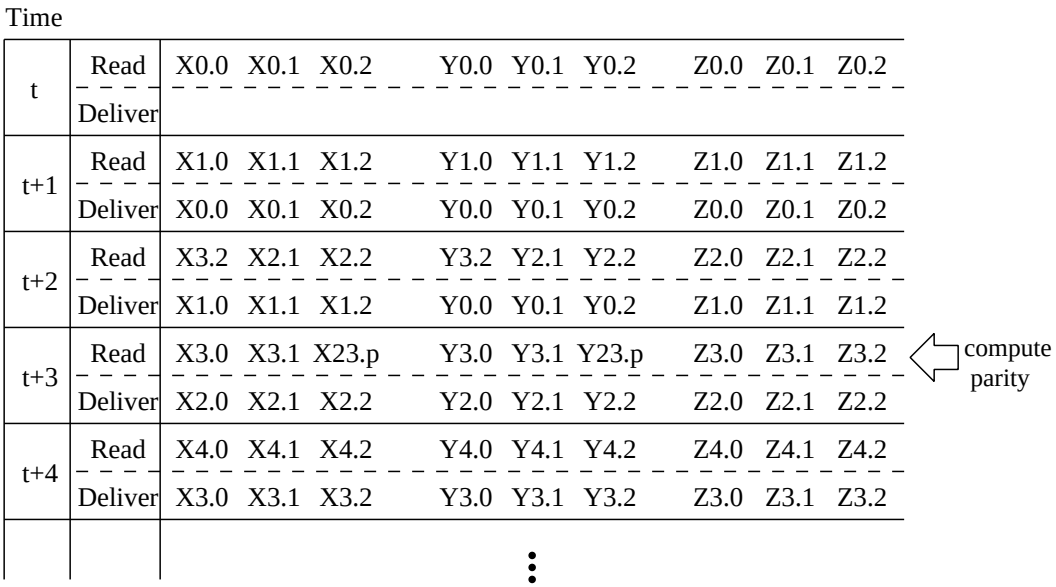

Figure 14: Delivery when Buffering Ahead

$t+2$ we read $X2$, the first subobject affected by the disk failure, but due to the offset, we only deliver $X1$. At time $t+3$ we read $X3$, which completes the parity group and makes delivery of $X2$ possible, and so on. The offset enables us to continue a smooth delivery of object X by providing a sufficient time slack for reading an entire parity group before having to deliver its first subobject. Since in this example there are two subobjects per parity group, the delivery only needs to be offset from reading by a single time slot. In general, an offset of $j - 1$ time slots is sufficient for objects with j subobjects per parity group.

The parity placement scheme illustrated above will not always be able to recover from a single failure. Consider what happens in Figure 13 when, for instance, disk 1 fails at time t and the system must still deliver objects X, Y, and Z. In

size. A first attempt at this would be to place several consecutive subobjects in one parity group[4]. This scheme is illustrated in Figure 13, where every two subobjects are protected by one parity fragment. For instance, subobjects $X0$ and $X1$, stored

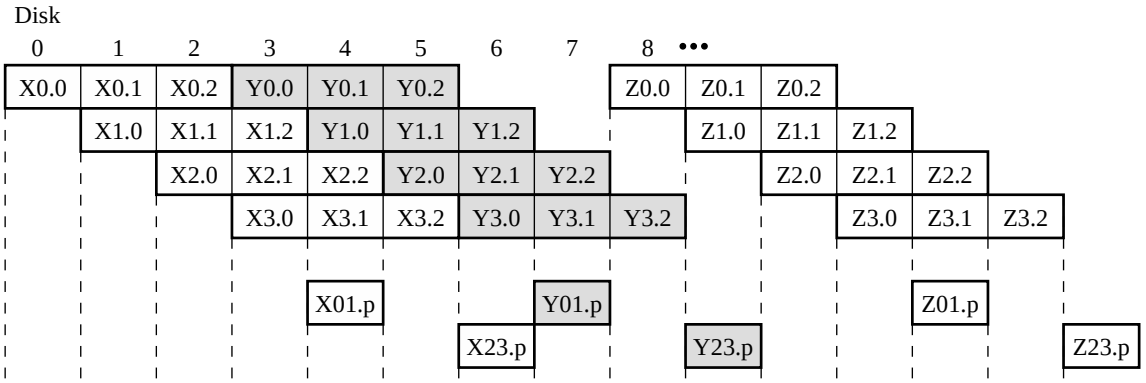

Figure 13: Multiple Subobjects per Parity Group

on disks $0, 1, 2$, and 3, are protected by $X01.p$ stored on disk 4.

When a disk fails we again have to perform a shift to the right. The shift is more complicated, due to having several subobject in a parity group, and is best illustrated through the example of Figure 13. Suppose disk 0 fails at time t, when we must read subobjects $X0$, $Y0$, and $Z0$. As a result of failure, we shift to the right and read fragments $X0.1, X0.2$, and $X1.2$ for object X, fragments $Y0.1, Y0.2$, and $Y1.2$ for object Y, and subobject $Z0$; since there is an idle slot between Y and Z, there is no need to shift Z. At this time, we are not able to deliver $X0$ because we do not have enough information to reconstruct $X0.0$. Since there are two subobjects per parity group, it takes two time slots to read an entire parity group and reconstruct the missing data. For the same reason we are unable to deliver $Y0$ at time t; we can, however, deliver $Z0$ since it is not affected by the failure. $X0.1, X0.2, X1.2, Y0.1, Y0.2$, and $Y1.2$ must be buffered until we are able to read the remainder of the parity groups. At time $t+1$ we read fragments $X1.0, X1.1$, and $X01.p$ and compute the missing data from object X, i.e., $X0.0$. Similarly, we read $Y1.0, Y1.1$, and $Y01.p$ and reconstruct $Y0.0$; of course, we also read $Z1$, but there is no need to do parity computation there. At this time, we are able to deliver $X0, X1, Y0, Y1$, and $Z1$.

The problem we experienced in the previous example was that each subobject was not "self-sufficient" and depended on other subobjects for parity information. This not only created a "hiccup" in the delivery of objects, but also required buffer space for each object affected by the failure. These buffers are also necessary during normal

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

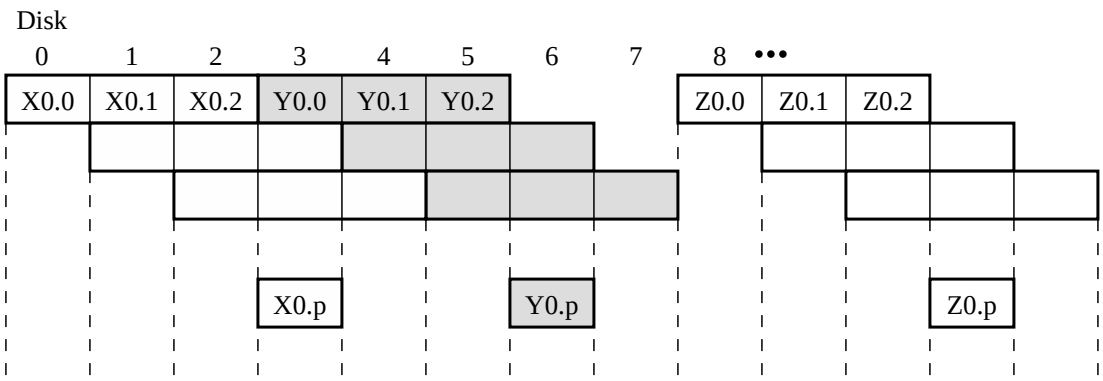

Figure 12: Parity per Each Subobject

fails, all the 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 12 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 $Y0$ because disk 3 is busy; 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, the discussions on reliability, reconstruction, etc., presented in that context, apply here.

As was already mentioned in Section 3.2, one problem with the parity-per-subobject scheme is that it potentially wastes too much storage 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.

4.2 Multiple Subobjects per Parity Group

The scheme of the previous section had a large storage cost, which was a result of small parity groups; hence, to remedy this problem, we must increase the parity group

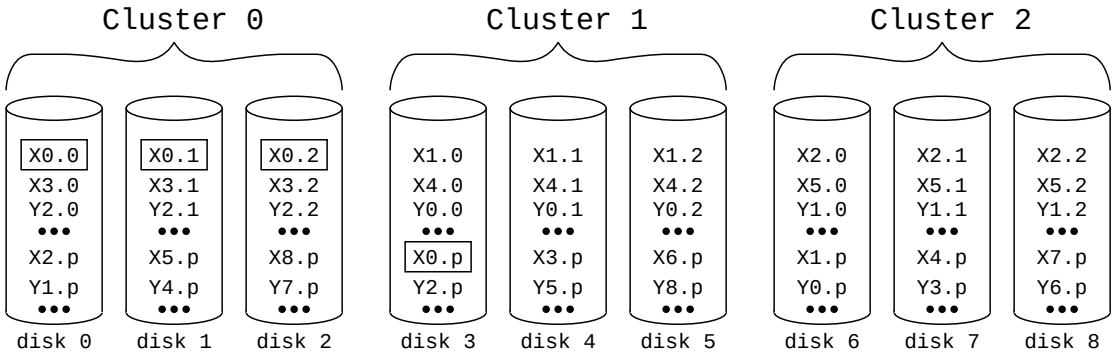

Figure 11: Improved BW-Storage Scheme

disks $4, 5$, and 6 and reconstruct $Y0.0$ through a parity computation. If, on the other hand, we are suppose to deliver $X3$ and $Y0$ at this time, then we can accomplish that by reading disks $1, 2$, and 4, to reconstruct $X3.0$ and disks $3, 5$ and 6 to reconstruct $Y0.1$.

Note however, that a system using this "improved-bw-storage" scheme is even more prone to catastrophic failures than a system using just the "improved-bw" scheme. By spreading the parity among all the disks in a cluster, we've created even more dependencies between disks in adjacent clusters. Any two failures in adjacent clusters result in data loss. As in the improved-bw scheme, any two failures in the same run result in degradation of service.

4 Staggered Striping

In this section we continue the discussion of parity schemes, but in the context of staggered striping. First, we maintain the constraint of a constant bandwidth requirement, and then we generalize this discussion to the case of variable bandwidth requirements.

4.1 Parity per Subobject

In the case of constant bandwidth requirement, we can adapt the improved-bw-storage scheme (see Section 3.2), used with simple striping, to a system using staggered striping by associating a parity fragment with each subobject. This *parity-per-subobject* scheme is illustrated in Figure 12. In this example there are three objects, X, Y, 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. When a disk

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 we have created 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 10 belongs to two different parity groups because it acts as the parity disk for cluster 0 and as a data disk for cluster 2. To illustrate this point, consider the example of Figure 9 again. If there is only one active request in the system, then it can withstand up to 3 failures, before a catastrophic one occurs, such that there is no more than one failure in each cluster and there is no more than one failure in each parity group. With 2 active requests in the system, it can withstand only a single failure, in any parity group. As was mentioned before, with 3 requests in the system (which is a full system), it can not withstand even a single failure.

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 11. 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. For example, suppose that in Figure 11 $X0$ and $Y0$ must be delivered at time t, and disk 0 is down. Then, we can read $X0.1, X0.2$ and $X0.p$ from disks $1, 2$ and 3, and reconstruct $X0.0$ through a parity computation; in order to deliver $Y0$ at the same time, we must read $Y0.1, Y0.2$ and $Y0.p$ from

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 it can read only one fragment in each time slot. Therefore, it 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 9. 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 scheme is that reliability is not provided at the cost of bandwidth. However, it is not as reliable as the naive scheme; 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 even a single failure results in degradation in service; this is true because in the normal mode all the available bandwidth is already used. 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. Consider the example in Figure 10, where $X0, Y0$, and $Z0$ are scheduled to be read at time t, and nothing is scheduled on cluster 3. If disks 0 fails before time t, then $X0, Y0$, and $Z0$ can still

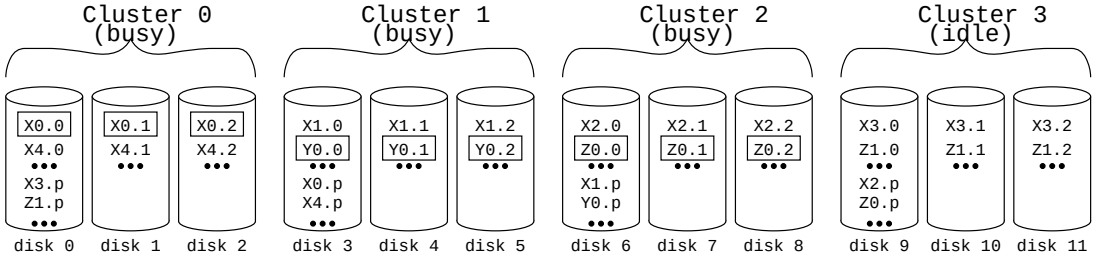

Figure 10: A Single Run

be delivered by doing a shift to the right. Suppose a second failure occurs at time t; if it turns out to be disk 3, then data is lost. If on the other hand[3], it turns out to be disk 7, then no data will be lost, but it will not be possible to deliver $Z0$, since

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

read simultaneously. Therefore, in our system, we must attach semantics to parity computation, i.e., we must associate parity with objects rather than with disk blocks.

3.2 Improved Bandwidth Schemes

In this section we continue using simple striping for data placement, but we use a more sophisticated parity placement scheme. As mentioned earlier, one problem with the naive scheme is that it potentially gives up a significant amount of bandwidth in order to provide reliability. We can improve on the naive scheme as follows. 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; hence, we'd be able to use it 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 $i+1$[2]. This *improved-bw* scheme is illustrated in Figure 9, 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 efficiency, but

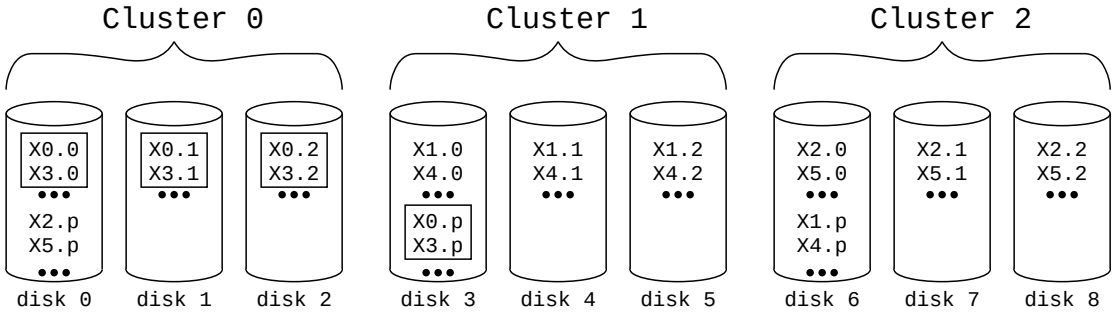

Figure 9: Improved BW Scheme

significantly reduces the storage efficiency, as compared to the naive scheme. In the example of Figure 9, under the normal mode of operation, data can be transmitted at the maximum possible 12 MB/sec. But, only 4.5 out of the 9 GB of disk space can be used to store objects. Because the first disk of every cluster contains both data and parity, at most half of that disk can be used by either one; therefore, only half of each disk in the system can be occupied by actual data. This results in only 50% efficiency of storage use, which by no means is a good tradeoff. However, we will still use this scheme in the following discussion for the purposes of illustration and then show how to improve upon it so as not to waste any more storage than is wasted by the naive scheme.

When a failure occurs at time t in cluster i, it and its adjacent clusters have to

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

recovery whether there is a delivery offset or not.

Note that the 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 divided evenly into parity groups of j subobjects each, then only 10% of storage and bandwidth would be given up for parity. Another approach is to try to associate parity with physical locations of the data blocks (as it is done in disk array systems [8]), rather than with a particular object. This scheme is illustrated in Figure 8, where again there is only one parity disk corresponding to all three clusters. Subobjects

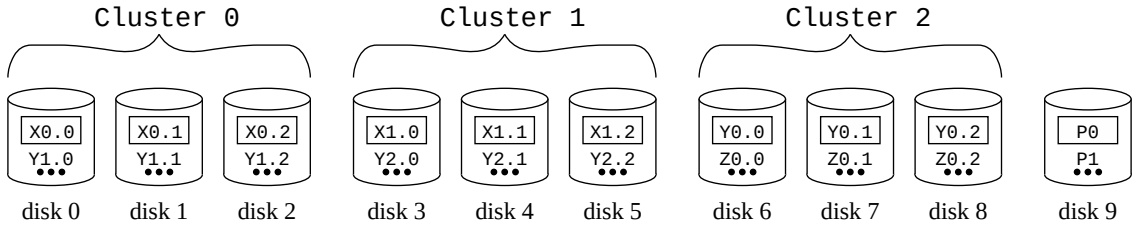

Figure 8: Another Improvement to the Naive Parity Scheme

$X0, X1$, and $Y0$ are protected by parity $P0$; subobjects $Y1, Y2$, and $Z0$ are protected by parity $P1$, etc.

Constructing parity groups in this manner solves the "object fragmentation" problem and also improves the efficiency of storage use; however this scheme does not perform well under failure. When a failure occurs, *appropriate* blocks from other data disks plus the parity disk must be read to reconstruct a missing block. Scheduling the parity disk to read an appropriate block is not difficult; however, scheduling the data disks to read the correct blocks might be. An effort to reconstruct the missing data could result in degradation of service and unreasonable buffering requirements. This is best illustrated through an example. Suppose that disk 0 fails in the example of Figure 8, and as the result of this failure $X0.0$ must be reconstructed. In order to compute the missing data, fragments $X0.1$, $X0.2$, subobjects $X1$ and $Y0$, and fragment $P0$ must be read. Suppose also that clusters 1 and 2 are not idle, and that cluster 1 is scheduled to read subobject $X1$, but cluster 2 is scheduled to read subobject $Z0$. In this case, $X0.1, X0.2$ and $X1$ must be buffered until $Y0$ is read. 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; in that case, we could construct the parity fragments accordingly. In other words, if in the above example we knew that X and Z would be scheduled at the same time, then we could compute $P0$ using $X0, X1$, and $Z0$ instead of $Y0$. But, we can not make such prediction. However, given the data layout, we are able to predict which fragments of a particular object are going to be

Note that in this scheme, it would usually not be possible to recover the missing information at the time of failure, which results in: a) a need for buffers and b) a "hick-up" in data delivery until enough information is collected. For instance, if in Figure 6 disk 4 fails before time t and delivery of objects X and Y is scheduled to start at time t, then these objects can still be delivered as follows: 1) at time t, read, deliver, and buffer $X0.0, X0.1$, and $X0.2$, and also read and buffer, but not deliver $Y0.0, Y0.2$, and $P1$, 2) at time $t+1$ read and buffer $X1.0, X1.2$, and $P0$ and also read $Y1.0, Y1.1, Y1.2$, reconstruct $Y0.1$ through parity computation, and deliver $Y0$ and $Y1$, and 3) at time $t+2$ read $X2.0, X2.1$, and $X2.2$, reconstruct $X1.1$ through parity computation, and deliver $X1$ and $X2$. This is illustrated in Figure 7(a).

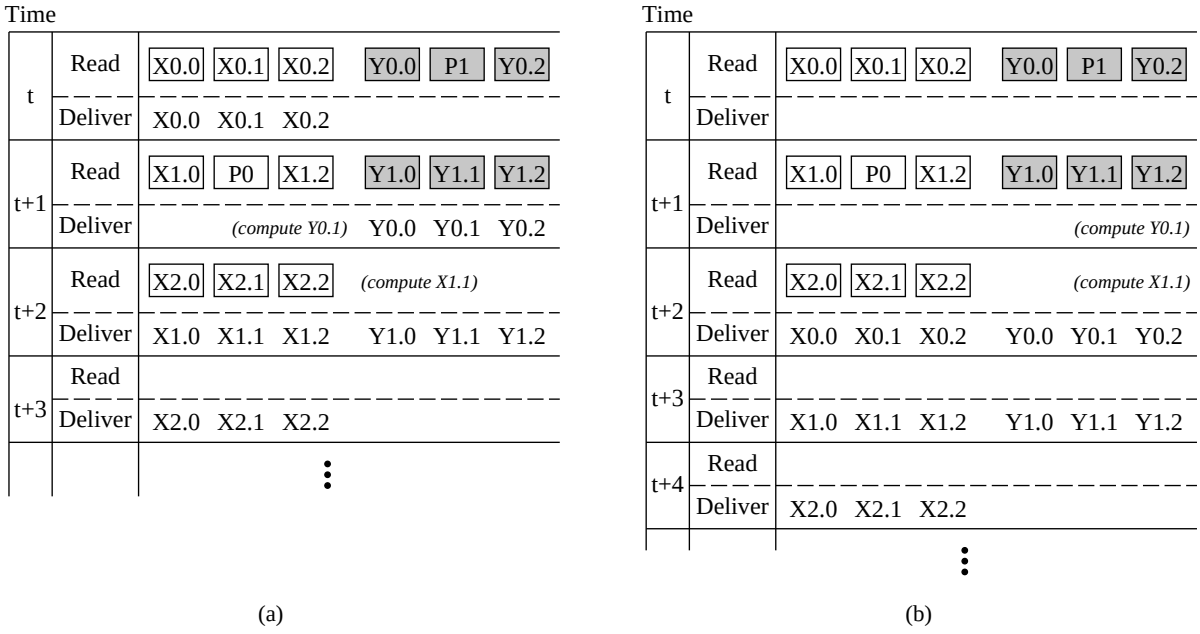

Figure 7: Delivery under Failure

Buffering of data is necessary even under normal operation. Since we never know when a failure might occur, each subobject must be buffered until its entire parity group is read; otherwise there might not be sufficient information to compute parity at the time of failure. Since we must pay a buffering penalty in order to insure recovery under failure, we could also use these buffers to avoid hick-ups. This can be done by offsetting the start of an object 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 7(a) is illustrated in Figure 7(b). The penalty for avoiding hick-ups is a small delay in transmission starting time; as mentioned already, the buffers are necessary for failure

group has been contained within one cluster, with each subobject having its own parity. Instead, we could use one parity disk for every j clusters, and hence construct parity groups using j subobjects, as opposed to one. Of course, we can not expect each object to have a "proper" number of subobjects, i.e., a number that can be divided evenly by j; therefore, the first and last parity group of every object might have less than j subobjects. Such a scheme is illustrated in Figure 6, where all three clusters share a single parity disk. In this example, each parity group, except for the

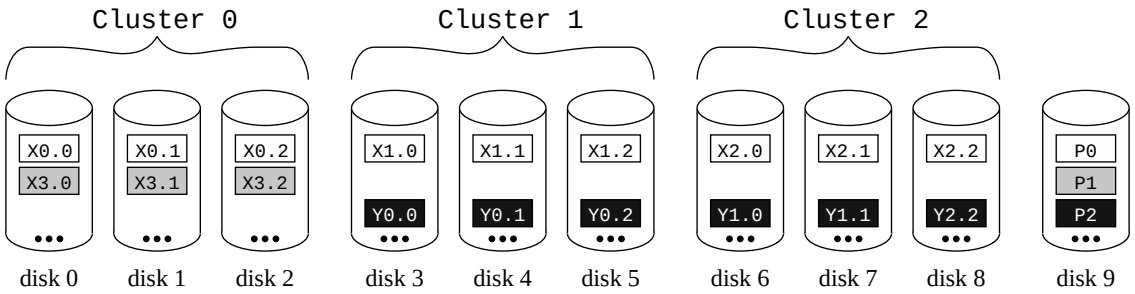

Figure 6: "Improved" Naive Parity Scheme

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,

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

and

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

On the other hand, object Y's first parity group is constructed out of only 2 subobjects, and therefore

$$P2 = Y0.0 \oplus Y0.1 \oplus Y0.2 \oplus Y1.0 \oplus Y1.1 \oplus Y1.2.$$

When a disk fails, the following must be done to recover the missing information:

- whenever the failed disk is scheduled to read a fragment, read an appropriate fragment from the parity disk instead
- collect the necessary fragments in buffers, until an entire parity group is available for parity computation
- when an entire parity group is available, compute the parity and deliver the subobjects affected by the failure

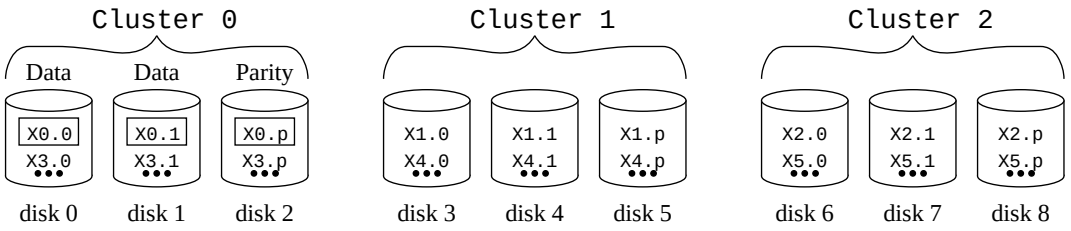

Figure 5: Naive Parity Scheme

are read from disks 1 and 2, respectively, and $X0.0$ is reconstructed through a parity computation, i.e., $X0.0 = X0.1 \oplus X0.p$.

The naive scheme can withstand up to one failure per cluster, before a catastrophic failure occurs. If we are unable to rebuild the failed disk before a second failure occurs in the same cluster, then that second failure will result in data loss. 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. Consider again the example of Figure 5 and suppose that each disk can hold 1 GB of data which can be delivered at 4 MB/sec. (We will assume this storage capacity and transmission rate for the remainder of the paper, unless stated otherwise.) In that case, the system has 9 GB of storage, of which only 6 GB are used for storing objects, i.e., about about 33% of the available storage is not used to store data. In addition, during the normal mode of operation, only two (data) disks in each cluster are used for object transmission; although each cluster is capable of delivering 12 MB/sec, it only delivers 8 MB/sec. Hence, even during the normal mode of operation about 33% of the available bandwidth is not being utilized. (Of course, in the degraded mode each cluster can only deliver 8 MB/sec.)

In general, storage and bandwidth costs could be reduced by using larger parity groups. For instance, if in the previous example the cluster size is increased to 3 data disks (plus a parity disk), then storage and bandwidth loss would be reduced to 25%, i.e., 9 out of 12 GB could be used for data and transmission could be done at 12 out of the possible 16 MB/sec. However, this "artificial" increase in cluster size could result in wasted bandwidth, if there is no need for such high transmission rates. In addition, larger clusters increase the probability of multiple failure occurring within the same cluster, which results in data loss. Hence, the increased efficiency in storage and (possibly) bandwidth must be balanced against the decreased reliability.

Instead of increasing the parity group size by increasing the cluster size, we could try to do it by increasing the number of clusters per parity group. So far, each parity

and $Z0$ are protected by parity stored in $P0$, i.e., $P0 = X0 \otimes Y0 \otimes Z0$. If disk 1 fails, then the information stored on it can be reconstructed by reading the appropriate data from disk 4, i.e., by reading $P0$ and then reconstructing $X0$ through a parity computation, where $X0 = Y0 \otimes Z0 \otimes P0$.

There are three modes of operation for a disk subsystem [7], 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. It is important to rebuild the failed disk(s) as fast as possible, since a second failure might result in degradation of service and/or loss of data; it is also important to do so without a significant interference with the system's workload. Several rebuild schemes are described in [7]; these include: a) *basic* rebuild, where the data is read from the surviving disks, reconstructed through a parity computation, and then written to the spare disk, b) rebuild with *read-redirect*, where, in addition, requests, for the portion of the data on the missing disk that has already been reconstructed on a spare, are redirected to the spare disk, and c) *piggy-backing* rebuild, which takes advantage of requests for data on surviving disks and uses the retrieved information to reconstruct some portion of the failed disk. 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, under the assumption of constant bandwidth requirement and clustered disk scheduling.

3.1 Naive Scheme

A simple scheme for providing parity information is to designate one of the disks in a cluster as the parity disk, which amounts to every cluster acting as a separate disk array. The example of Figure 5 illustrates this *naive* scheme, where every cluster consists of two data disks and one parity disk. 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 in each cluster are used for displaying an object. During the degraded mode of operation, the surviving data disks plus the parity disk are read and used to reconstruct the data that resides on the failed disk. For instance, to deliver $X0$ in Figure 5 under normal operation, $X0.0$ and $X0.1$ are read from disks 0 and 1, respectively. Under failure, for example of disk 0, $X0.1$ and $X0.p$

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 (stagger over by 2 disks) and transmit $X1$ from disks 2, 3 and 4, and so on.

A special case of staggered striping, where all objects have the same bandwidth requirement and hence the same degree of declustering, we refer to this as a constant bandwidth requirement, 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 4 illustrates such as system, with 2 clusters and degree of declustering equal to 3. Scheduling in this system is similar to the staggered striping schedule described

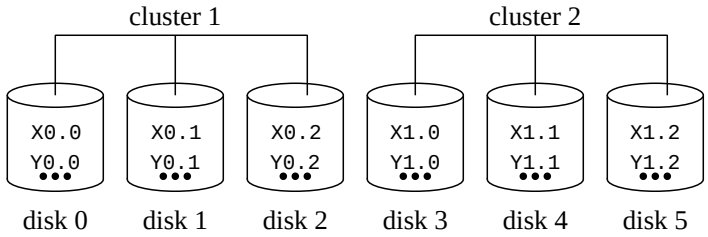

Figure 4: Simple Striping

above. For instance, in Figure 4, if at time t cluster 1 is idle, then transmission of object X can begin. Hence, at time t subobject $X0$ can be transmitted from cluster 1; at time $t+1$ subobject $X1$ can be transmitted from cluster 2, and so on. We refer to simple striping as having a *constant bandwidth* requirement and *clustered disk scheduling*.

As mentioned in Section 1, reliability and availability of data are serious concerns in information systems in general, which are solved by storing redundant information. Two basic schemes for introducing redundancy are: a) mirroring and b) parity based schemes. Mirroring [2] refers to fully replicating each disk on another disk; whenever a disk fails, its mirror can be used to retrieve the missing data. Mirroring is not a good replication technique for multimedia servers because it sacrifices too much of the system's storage and bandwidth for reliability (see Section 1). Parity based schemes construct a parity block for every d data blocks. The parity block plus the d data blocks constitute a parity group; we define d to be the parity group size, i.e., the number of data blocks in a parity group. The overhead for parity based schemes is $\frac{1}{d+1}$ of the total storage space, as opposed to $\frac{1}{2}$ of the storage space as in mirrored schemes. Parity schemes are a better choice of introducing redundancy into multimedia systems since they allows control over how much storage and bandwidth is used for 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. For example, Figure 2 illustrates a system using a parity scheme, where objects $X0$, $Y0$,

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 striping 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 a 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 in to disks from tertiary storage are striped across all the disks in the system. Each object is divided into subobject, and each subobject is divided into fragments. 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]). The number of fragments in a subobjects 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*. We place a subobject on a set of k disks and then stagger over to the next set of k disks to place the next subobject; 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 $i+1$. An example of such a system is illustrated in Figure 3, where object X is declustered among three disks and object Y is declustered among two disks. The first subobject of object X,

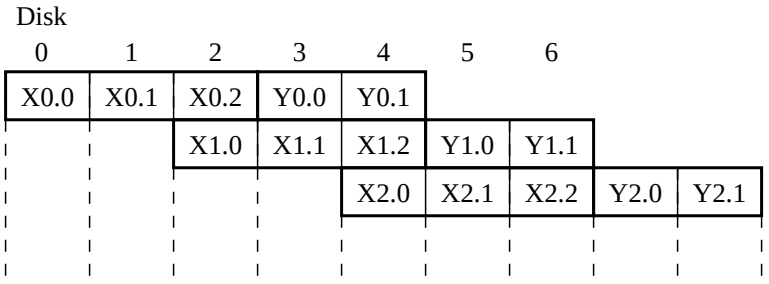

Figure 3: Staggered Striping

$X0$, 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 2, and hence the first fragment of subobject $X1$ is placed on disk 2. 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

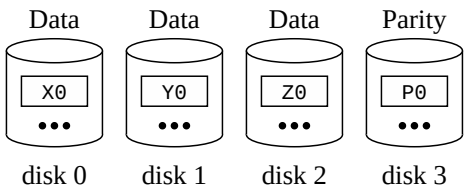

Figure 2: Disk Subsystem with Parity Information

in tertiary storage access; the greater is the available bandwidth of a system, the more requests can be serviced simultaneously. 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 is wasted due to parity, b) how much bandwidth is wasted, c) how system behavior is altered due to a disk failure, d) how many failures can the system withstand before degradation of service occurs or data is lost[1], and e) (in some cases) how to rebuild the failed disk on a spare one. *Degradation of service* denotes the situation in which the system is unable to continue servicing all requests that are active when the failure occurs, 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. Our goal in this report is to determine the schemes for providing high reliability and availability in the multimedia server and to consider the 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.

2 Background

The concepts of striping and declustering are used in general purpose I/O subsystems, as in RAID technology [8], as well as in application specific systems, such as database management systems, e.g., [9] and [3], to name a few. In [5] and [4], these concepts are extended to a parallel multimedia system. In [6] 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

[1]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.

stored on disks, the less is the probability that any one request will result in tertiary storage access. The size of the objects precludes them from being stored in main memory; the (usually) low bandwidth and high latency of tertiary devices precludes objects from being transmitted directly from the tertiary store.

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. However, disks are not very reliable; hence, given such a large number of disks, the probability that one of them fails is quite high. According to [8], the mean time to failure (MTTF) of a single disk is on the order of $30,000$ hours; this means that the MTTF of some disk in a 1000 disk system is on the order of 30 hours. This is 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 (and possibly rescheduling) of request in progress, if a the data for a currently displayed object is on a failed disk, and it does result in a significant degradation in performance, since the objects lost due to a failure must be retrieved from tertiary storage. Therefore, without some form of fault tolerance, such a system would not be usable.

To improve the reliability and availability of the system we must use some fraction of the disk space to store redundant information. How much redundant information we store and how we place it on the disks will determine: a) the amount of storage available for "real" data, b) the amount of bandwidth available for displaying this data, and c) the resiliency of the system to disk failure. The more redundant information is stored, the less is the probability that a failure results in data loss and a subsequent (massive) access of tertiary storage. However, the less redundant information is stored, the more space is available for storing objects and possibly the less bandwidth is available for displaying them. For instance, consider the four-disk storage subsystem illustrated in Figure 2, where the first three disks are used to store "real" data and the last disk is used to store parity information, e.g., $P0 = X0 \otimes Y0 \otimes Z0$. In this example, only three fourth of the total storage space (disks $0, 1$ and 2) is occupied by actual objects. Similarly, only three disks at a time can be used to transmit data; when all four disks are operational, disks $0, 1$ and 2 are used to transmit data, but when one of these disks fails, the data stored on it is reconstructed by reading disk 3. Therefore, in this example, one fourth of the storage space and one fourth of the available bandwidth are sacrificed for reliability. As mentioned earlier, the more objects can be stored on disks, the less is the probability that any one request results

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 bandwidth specified by the display station. Another challenging task in multimedia systems is to 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

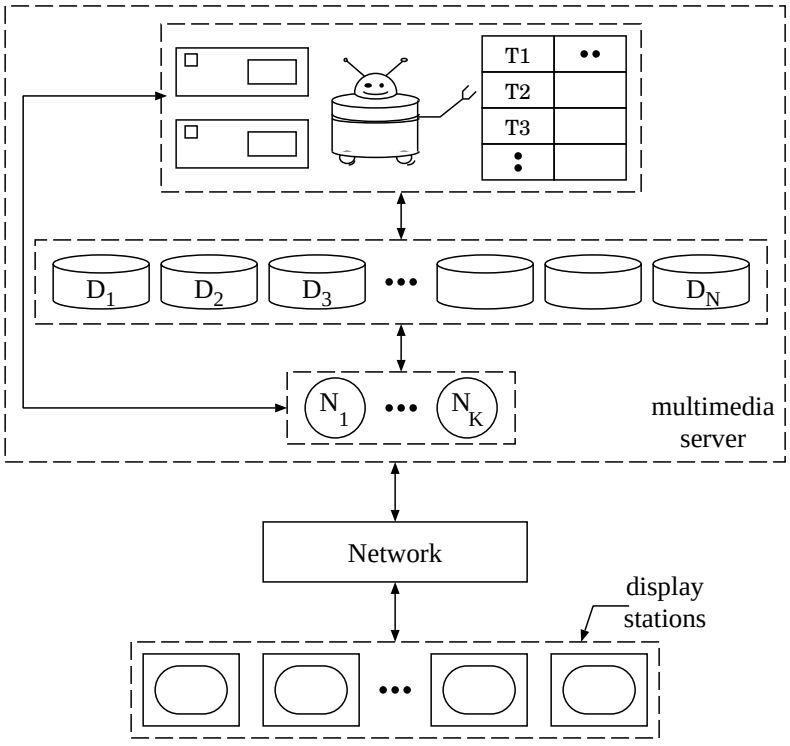

Figure 1: Multimedia System

consists of a tertiary storage library, a collection of disks, and a set of processors. The entire database permanently resides on tertiary storage. The objects are retrieved, from the tertiary device, and placed on disk drives on demand; if the secondary storage capacity is exhausted 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 performance degradation and should be avoided as much as possible. The more objects can be

A Fault Tolerant Design of a Multimedia Server

WORK IN PROGRESS: please do not distribute

Steven Berson Leana Golubchik
Richard R. Muntz

UCLA Computer Science Department

February 10, 1994