Software and Performance Issues in the Implementation of a RAID Prototype

Edward K. Lee

March 13, 1992

Contents

List of Figures

List of Tables

1 Introduction

This report is the result of knowledge gained in designing and implementing software for RAID-I (*RAID the First*), a prototype RAID level 5 system designed and built at the University of California at Berkeley. RAID-I is currently fully operational and should soon be available as a network file server. The main purpose of RAID-I is to prototype RAID specific software, gain experience with SCSI, and foresee performance bottlenecks in the implementation of more advanced RAID systems currently being designed at Berkeley. The following assumes that the reader is familiar with RAID [3, 7, 8] systems.

The earlier sections of this document are concerned with describing the RAID-I software. Specifically, they describe how *logical RAID block addresses* (block number) are mapped to *physical disk addresses* (disk, sector), how IO requests are serviced, and how the recovery mechanism reconstructs the contents of failed disks concurrently with user request servicing. The latter sections investigate the performance consequences of various parity placement schemes. In particular, we find that at relatively large request sizes, on the order of a few hundred kilobytes, the choice of parity placement can cause significant differences in performance.

1.1 Hardware Overview

RAID-I is configured from off-the-shelf components consisting of a Sun4/280 running the Sprite operating system, 4 dual-string Interphase Jaguar Controllers and 32 5.25 inch Imprimis Wren IV SCSI disks. The fully configured system consists of 8 SCSI strings with 4 disks per string. The Sun4/280 is generally rated at 8 to 10 VAX MIPS and is configured with 128 MB of memory. A single Interphase HBA has a sustained transfer rate of 4 MB/s. A Wren IV has a formatted capacity of 344 MB with 512 byte sectors, average rotation time of 8.33 ms, average seek of 17.5 ms, a 32 KB track buffer and a sustained transfer rate of 1.3 MB/s. Four Wren IV drives on a single string of the Interphase HBA supports SCSI bus transfers of 4 MB/s peak and 3 MB/s sustained.[1]

1.2 Software Overview

The software for RAID-I is implemented as a device driver in the Sprite operating system. This makes RAID-I logically indistinguishable from the other block devices supported by Sprite, and appears to Sprite as a single disk of very large capacity. The RAID device driver communicates with disks through SCSI device drivers supported by Sprite. The RAID device driver is best pictured as a layer of software that runs on top of Sprite's SCSI device drivers.

2 Logical to Physical Mapping

Formally, we define a logical to physical RAID mapping as a pair of functions: the *data-mapping-function* and the *parity-placement-function.* The data-mapping-function is a one-to-one function which maps a logical block address (blockNumber) to a physical disk addresses (disk, sector). Frequently, we will find it convenient to picture a disk array as a two dimensional array of disks and thus to identify a disk by its row and column numbers. Thus, the physical disk address can be

[1] I am indebted to Ann Chervanak, a member of the Berkeley RAID group, for providing me with the information presented here concerning the Wren IV disks and Interphase HBA's.

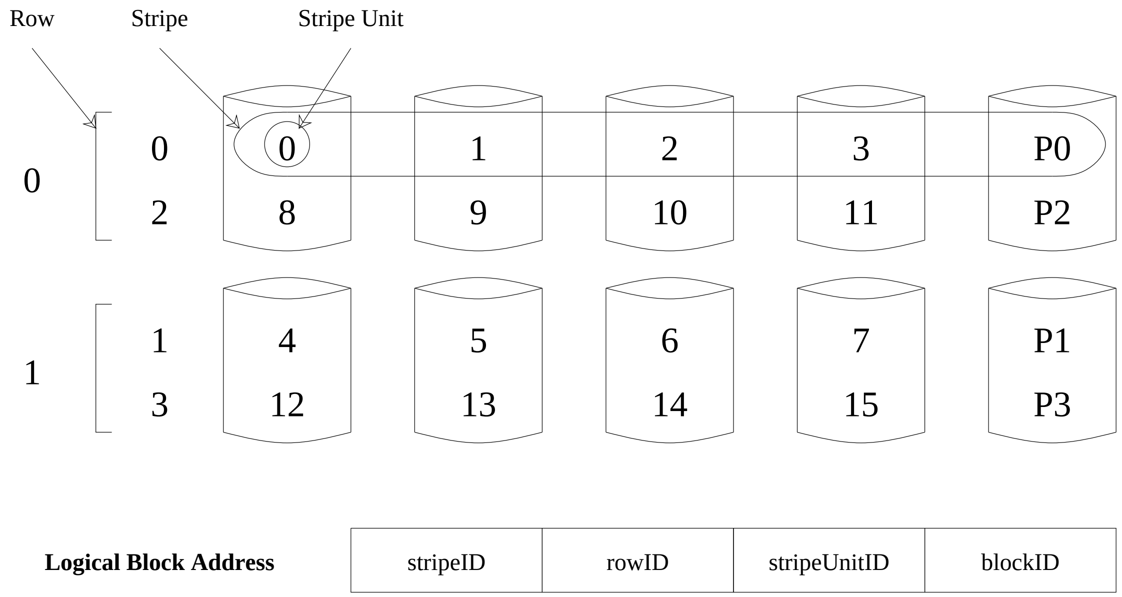

Figure 1: Data Mapping Entities. This figure illustrates the data mapping entities, *stripe unit stripe* and *row* in the context of a RAID level 4 disk array. This figure also illustrates how the logical RAID block address is decomposed into the ordered tuple (stripeID, rowID, stripeUnitID, blockID).

represented as the ordered tuple (row, column, sector). The parity-placement-function is a many-to-one function which maps a logical block address (blockNumber) to a physical disk address (row, column, sector). The parity-placement-function specifies for each logical block its corresponding parity sector. Parity is computed block-wise over all logical blocks which map to the same parity sector.

Strictly speaking, it is not possible to specify the data mapping independently of the parity placement. It is, however, useful to discuss certain aspects of the two problems as if they were independent. The following sections decompose the logical to physical RAID mapping problem into two somewhat independent subproblems:

- The mapping of data across disks.
- The placement of parity within the data mapping.

2.1 Data Mapping

Ignoring for now the placement of parity, the following data mapping entities are considered. Figure 1 illustrates each mapping entity except the block.

block The minimum unit of data transfer to or from a RAID device.

stripe unit The unit of data interleaving; i.e. the group of logically contiguous blocks that are placed consecutively on a single disk before placing blocks on a different disk.

parity stripe The group of stripe units over which parity is computed. The term *stripe* used unqualified refers to a parity stripe. In contrast, the term *data stripe* refers to a collection

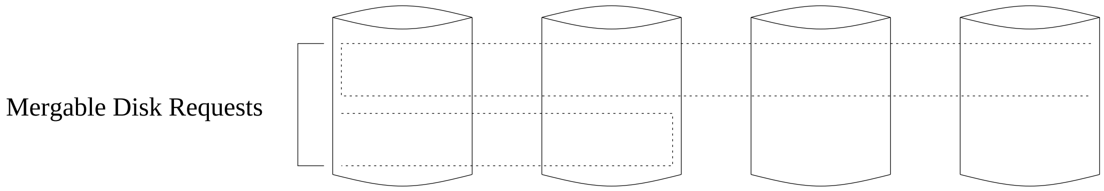

Figure 2: Request Merging. Disk requests which are physically contiguous on the same disk may be merged into a single disk request.

of logically contiguous stripe units over which data is striped. The distinction arises because parity does not have to be computed across logically contiguous stripe units. For example, in Figure 1, if $P0$ computed parity over stripe units 0, 5, 10 and 15, then those stripe units would comprise a parity stripe even though data would not be striped in that order. Henceforth, we will assume that the number of stripe units in a parity stripe is equal to the number of columns in the disk array.

row The minimal group of disks over which a parity stripe can be placed.

Thus, a logical RAID block address can be interpreted as the following ordered tuple: (stripe number within row, row number within array, stripe unit number within stripe, block number within stripe unit). We will abbreviate the above tuple as (stripeID, rowID, stripeUnitID, blockID). Using the most significant part of the logical RAID block address as the stripeID interleaves the stripes over the rows.

The following sections discuss the stripe unit and parity stripe in more detail. The following definitions will prove useful for this purpose:

small request A request that fits entirely within a stripe unit (a single disk); i.e. requests that are smaller than or equal in size to a stripe unit and do not cross stripe unit boundaries.

moderate request A request that spans multiple stripe units but does not use each disk more than once; e.g. in Figure 1, a request to stripe units zero through five.

large request A request that is large enough to use several disks more than once; e.g. in Figure 1, a request for stripe units zero through fourteen.

Thus, whether a particular request is considered small, moderate or large is dependent upon the size of the disk array.

2.1.1 Stripe Unit

The stripe unit size is one of the most important parameters in configuring a RAID level 5 device and especially affects the performance of moderate requests. Small requests are not affected since they fit entirely within a stripe unit. Large requests are less sensitive to stripe unit size because they tend to wrap to the same disk. This allows accesses to the physically contiguous stripe units to be merged, effectively forming a larger stripe unit. Figure 2 illustrates requests that may be merged.

Small stripe units increase the amount of parallelism available for servicing moderate requests but are less efficient at higher loads since a greater number of disks are utilized per request. Large stripe units result in lower disk utilization but also lower performance for moderate sized requests at low loads[9]. If the disks in the array are not rotationally synchronized, the increase in parallelism achieved with smaller stripe units will be offset by the synchronization overhead incurred by spreading the request over more disks. Because data striping does not reduce the initial seek and rotational latencies but only the data transfer time, performance increases due to parallelism are often much less than expected. For example, with current technology, striping a 32 kilobyte request over two synchronized disks can result in only a 25 percent speedup. Chen [1] has studied the performance implications of varying stripe units sizes in non-redundant disk arrays.

2.1.2 Parity Stripe

The parity stripe size affects the performance of moderate write requests. Writes in RAID systems are most efficient when they are the same size as the stripe size. This is because parity can be calculated for exact stripe writes by xoring the data to be written without having to read additional information from the disk. Thus, it is frequently desirable to choose smaller stripe sizes, so that a majority of writes can be executed as exact stripe writes.

2.2 Parity Placement

The number of different ways parity can be placed relative to data is astronomical. It is, moreover, very likely that the choice of parity placement will significantly affect performance under certain workloads. We will begin our investigation of parity placements by requiring that that they satisfy the following properties:

- Stripe units belonging to the same parity stripe should not map to the same column. (In many RAID systems, the disks within a column have a common failure mode; e.g. the string interface.) This is referred to as the "orthogonal RAID" property and guarantees that the failure of a single column does not result in data unavailability.

- In a RAID with n stripe units per parity stripe, the ith parity stripe unit should correspond to logical stripe unit j such that $j \operatorname{div} n = i$. This guarantees that the parity for any write request that is stripe aligned and a stripe in size can be computed by using only the data being written: i.e. without having to read old data from disk.

Dibble [2] has investigated parity placements under a different set of requirements[2]. Figure 3 illustrates eight parity placements that are supported by our prototype RAID driver and which satisfy the above properties. The RAID level 0 and RAID level 4 placements were proposed in earlier RAID papers [7,8]. Of the RAID level 5 placements, the right-asymmetric, left-asymmetric, right-symmetric and left-symmetric placements were initially created ad-hoc to determine if different placements actually impacted performance. The other two RAID level 5 placements, the extended-left-symmetric and flat-left-symmetric, are modifications of the left-symmetric placement that were developed in order to improve read performance as a result of performance studies that will be presented in later sections of this report. The following lists the placements illustrated in Figure 3 for the reader's convenience:

[2] He requires all writes to be small. A large write is therefore broken into many independent small writes.

- RAID level 0
- RAID level 4
- RAID level 5 Placements
  - Right-Asymmetric
  - Left-Asymmetric
  - Right-Symmetric
  - Left-Symmetric
  - Extended-Left-Symmetric
  - Flat-Left-Symmetric

The following describes each placement and specifies for each placement the data-mapping-function and the parity-placement-function. Both functions map logical block addresses, (stripeID, rowID, stripeUnitID, blockID), to physical disk addresses, (row, column, sector). Unfortunately, the interpretation of the logical block address as (stripeID, rowID, stripeUnitID, blockID) is not applicable to RAID level 0 placements which does not have parity stripes. So that we may treat the RAID level 0 placement in the same uniform framework as the other placements, we arbitrarily define a parity stripe for a RAID level 0 placement as n consecutive and aligned stripe units where n is the number of columns in the disk array. Thus, in Figure 3, stripe units 5, 6, 7, 8 and 9 constitute a parity stripe. Note that a RAID level 0 placement with n columns has n data stripe units per parity stripe whereas the redundant placements with n columns have only $n-1$ data stripe units per parity stripe. This affects the conversion of the logical block address to the ordered tuple (stripeID, rowID, stripeUnitID, blockID). In the interests of simplifying the following equations, we will assume that the stripe unit is the same size as a block which is the size as a sector. Thus, our functions will map (stripeID, rowID, stripeUnitID) to (row, column, sector). The following assumes that any number modulo a positive number is also a possible number. Unfortunately, on some machines, a negative number modulo a positive number is a negative number. The following definitions and clarification of terms will prove useful:

$$
\begin{aligned}
n &= \text{Number of columns in array.} \\
  & \text{(Number of stripe units per parity stripe.)} \\
m &= \text{Number of rows in array.} \\
ndata &= \text{Number of data stripe units per parity stripe.} \\
  &= n \text{ (RAID level 0) or } n-1 \text{ (all others)} \\
blockNumber &= \text{Logical RAID block address.} \\
stripeUnitID &= blockNumber \bmod ndata \\
rowID &= (blockNumber \operatorname{div} ndata) \bmod m \\
stripeID &= (blockNumber \operatorname{div} ndata) \operatorname{div} m \\
logicalStripeID &= blockNumber \operatorname{div} ndata
\end{aligned}
$$

2.2.1 RAID level 0

The RAID level 0 placement results in the conventional modulo n data striping scheme [4–6, 10]. The RAID level 0 placement is defined here for comparison purposes only.

RAID level 0

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
20	21	22	23	24

RAID level 4

0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3
16	17	18	19	P4

Flat-Left-Symmetric

0	1	2	3	4
5	6	7	8	9
10	11	12	13	14
15	16	17	18	19
P4	P3	P2	P1	P0

Asymmetric — Right

P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4

Symmetric — Right

P0	0	1	2	3
7	P1	4	5	6
10	11	P2	8	9
13	14	15	P3	12
16	17	18	19	P4

Asymmetric — Left

0	1	2	3	P0
4	5	6	P1	7
8	9	P2	10	11
12	P3	13	14	15
P4	16	17	18	19

Symmetric — Left

0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

Extended-Left-Symmetric

0	1	2	3	P0
10	11	P2	13	4
P4	21	12	23	14
20	31	22	P6	24
30	P8	32	33	34

5	6	7	P1	9
15	P3	17	8	19
25	16	27	18	P5
35	26	P7	28	29
P9	36	37	38	39

Figure 3: Parity Placements. Each square corresponds to a stripe unit. Each column of squares corresponds to a disk. Each matrix corresponds to a single row of disks. In all cases except the RAID level 0 and RAID level 4 placements, the minimum repeating pattern is shown.

Data-Mapping-Function

$$\begin{aligned} row &= rowID \\ col &= stripeUnitID \\ sector &= stripeID \end{aligned}$$

Parity-Placement-Function

Not applicable.

2.2.2 RAID level 4

The RAID level 4 placement is derived from the RAID level 0 placement by adding a parity disk to each row. The read performance of a RAID level 4 placement with n disks per row is identical to the read performance of a RAID level 0 placement with $n-1$ disks per row. Because the RAID level 0 placement is widely used and better understood than the other placements, this can be a desirable property. A disadvantage of the RAID level 0 placement is that the parity disks will become a bottleneck for small writes [8] since every write must update a parity disk of which there is only one per row. Also, only $n-1$ disks per row, instead of n disks per row are available for servicing reads since the nth disks do not contain data.

Data-Mapping-Function

$$\begin{aligned} row &= rowID \\ col &= stripeUnitID \\ sector &= stripeID \end{aligned}$$

Parity-Placement-Function

$$\begin{aligned} row &= rowID \\ col &= n-1 \\ sector &= stripeID \end{aligned}$$

2.2.3 Right-Asymmetric

The right-asymmetric placement is derived from the RAID level 0 placement by pushing out data stripe units horizontally as parity stripe units are inserted. For each successive parity stripe, the point at which the parity stripe unit is inserted is rotated one stripe unit towards the right.

Data-Mapping-Function

$$\begin{aligned} row &= rowID \\ col &= stripeUnitID && \text{if } stripeID \bmod n > stripeUnitID \\ &= stripeUnitID + 1 && \text{otherwise} \\ sector &= stripeID \end{aligned}$$

Parity-Placement-Function

$$\begin{aligned} row &= rowID \\ col &= stripeID \bmod n \\ sector &= stripeID \end{aligned}$$

2.2.4 Left-Asymmetric

The left-asymmetric placement is derived from the RAID level 0 placement by pushing data stripe unit out horizontally as parity stripe unit are inserted. For each successive parity stripe, the point at which the parity stripe unit is inserted is rotated one stripe unit towards the left.

Data-Mapping-Function

$$\begin{aligned} row &= rowID \\ col &= stripeUnitID \quad &&\text{if } (-stripeID - 1) \bmod n > stripeUnitID \\ &= stripeUnitID + 1 &&\text{otherwise} \\ sector &= stripeID \end{aligned}$$

Parity-Placement-Function

$$\begin{aligned} row &= rowID \\ col &= (-stripeID - 1) \bmod n \\ sector &= stripeID \end{aligned}$$

2.2.5 Right-Symmetric

The right-symmetric placement is derived by right rotations of entire parity stripes from the RAID level 4 placement.

Data-Mapping-Function

$$\begin{aligned} row &= rowID \\ col &= (stripeUnitID + stripeID + 1) \bmod n \\ sector &= stripeID \end{aligned}$$

Parity-Placement-Function

$$\begin{aligned} row &= rowID \\ col &= stripeID \bmod n \\ sector &= stripeID \end{aligned}$$

2.2.6 Left-Symmetric

The left-symmetric placement is derived by left rotations of entire parity stripes from the RAID level 4 placement.

Data-Mapping-Function

$$\begin{aligned} row &= rowID \\ col &= (stripeUnitID - stripeID) \bmod n \\ sector &= stripeID \end{aligned}$$

Parity-Placement-Function

$$\begin{aligned} row &= rowID \\ col &= (-stripeID - 1) \bmod n \\ sector &= stripeID \end{aligned}$$

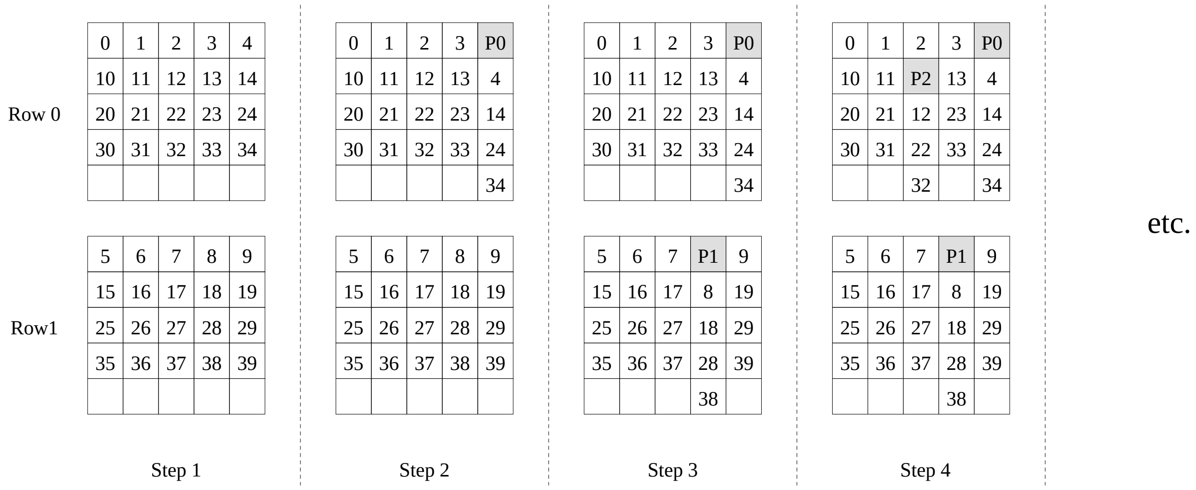

Figure 4: Derivation of the Extended-Left-Symmetric Placement

2.2.7 Extended-Left-Symmetric

The extended-left-symmetric placement is derived from the RAID level 0 placement by pushing out data stripe units vertically as parity stripe units are inserted as shown in Figure 4. For each successive parity stripe, the point at which the parity stripe unit is inserted is rotated one stripe unit towards the left. Figure 4 illustrates the conceptual steps in deriving the extended-left-symmetric placement from the corresponding RAID level 0 placement. In arrays with only one row of disks, the extended-left-symmetric placement is identical to the left-symmetric placement. In arrays with multiple rows of disks, traversing the logical blocks of the extended-left-symmetric placement in sequence results in a traversal of all the disks whereas the left-symmetric placement will skip every nth disk. Note that in Figure 3, where each five element column represents a distinct disk, $P1$ is the parity corresponding to stripe units 4, 5, 6 and 7 and *not* 5, 6, 7 and 9. Likewise, $P2$ corresponds to stripe units 8, 9, 10 and 11. This is to ensure that writes which are a parity stripe in size and parity stripe aligned can always be written without having to read additional information from the disks.

We find it convenient to specify the data-mapping-function for the extended-left-symmetric placement in terms or the mapping that would have been obtained with the RAID level 0 placement. Thus,

$$\begin{aligned} row' &= (blockNumber \operatorname{div} n) \bmod m \\ col' &= blockNumber \bmod n \\ sector' &= (blockNumber \operatorname{div} n) \operatorname{div} m \end{aligned}$$

Data-Mapping-Function

$$\begin{aligned} row &= row' \\ col &= col' \\ coladj &= 1 \;\text{ if } \exists i \text{ s.t. } 0 \leq i \leq sector' \bmod (n-1) \text{ and } (-m \times i - row' - 1) \bmod n = col' \\ &= 0 \;\text{ otherwise} \\ sector &= sector' + sector' \operatorname{div} ndata + coladj \end{aligned}$$

Parity-Placement-Function

$$
\begin{aligned}
row &= rowID \\
col &= (-logicalStripeID - 1) \bmod n \\
sector &= stripeID
\end{aligned}
$$

2.2.8 Flat-Left-Symmetric

The flat-left-symmetric placement is derived from the *extended*-left-symmetric placement by grouping all of the parity together and placing them at identical offsets within each disk. When reading large amounts of data, all disk heads within the same row skip over parity at the same time; thus, reducing disk synchronization time. When writing data, however, performance is likely to be worse than the extended-left-symmetric placement since the parity stripe unit is located at a different offset within the disk relative to its corresponding data stripe units. Note that in Figure 3, $P0$ is the parity corresponding to stripe units 0, 1, 2, and 3. Likewise, $P1$ corresponds to 4, 5, 6 and 7.

$$
\begin{aligned}
row' &= (blockNumber \operatorname{div} n) \bmod m \\
col' &= blockNumber \bmod n \\
sector' &= (blockNumber \operatorname{div} n) \operatorname{div} m
\end{aligned}
$$

Data-Mapping-Function

$$
\begin{aligned}
row &= row' \\
col &= col' \\
sector &= sector' + sector' \operatorname{div} ndata
\end{aligned}
$$

Parity-Placement-Function

$$
\begin{aligned}
row &= rowID \\
col &= (-logicalStripeID - 1) \bmod n \\
sector &= (stripeID \operatorname{div} n) \times n + n - 1
\end{aligned}
$$

2.2.9 Physical Versus Logical Placements

When rotating the placement of parity, the question arises whether to rotate relative to physical stripe addresses (stripeID) or logical stripe addresses (logicalStripeID). Physical stripe addresses are sequential on each disk whereas the logical stripe addresses follow the same sequence as the data striping. Figure 5 illustrates the physical and logical variations of the left-symmetric placement when applied to an array with two rows. Note that for a single row of disks, there is no difference between a physical placement and a logical placement (stripeID = logicalStripeID).

Specifying parity stripe unit relative to logical addresses can have surprising undesirable consequences. For example, *rotating parity stripe unit relative to logical addresses under certain array configurations will cause all of the parity stripe unit to be concentrated on a small subset of the disks!* Figure 5 illustrates this problem. To guarantee that this does not occur with a logical placement, one must ensure that the number of rows per array and the number of disks per row do not have a common integer factor other than one. For example, a two-by-five configuration is acceptable but a two-by-four configuration is not since two and four are both divisible by two. Physical placements, by contrast, always guarantee a uniform distribution of parity over all disks. Thus, physical placements are generally preferable over logical placements. All of the parity placements presented above, with the exception of the extended-left-symmetric and flat-left-symmetric placements, which suffer from the problem just described, are physical placements.

2 rows by 5 columns

Physical Left-Symmetric

Physical Stripe No.					
0	0	1	2	3	P0
1	9	10	11	P2	8
2	18	19	P4	16	17
3	27	P6	24	25	26
4	P8	32	33	34	35
0	4	5	6	7	P1
1	13	14	15	P3	12
2	22	23	P5	20	21
3	31	P7	28	29	30
4	P9	36	37	38	39

Logical Left-Symmetric

Logical Stripe No.					
0	0	1	2	3	P0
2	10	11	P2	8	9
4	P4	16	17	18	19
6	25	26	27	P6	24
8	35	P8	32	33	34
1	5	6	7	P1	4
3	15	P3	12	13	14
5	20	21	22	23	P5
7	30	31	P7	28	29
9	P9	36	37	38	39

2 rows by 4 columns

Physical Left-Symmetric

Physical Stripe No.				
0	0	1	2	P0
1	7	8	P2	6
2	14	P4	12	13
3	P6	18	19	20
4	24	25	26	P8
0	3	4	5	P1
1	10	11	P3	9
2	17	P5	15	16
3	P7	21	22	23
4	27	28	29	P9

Logical Left-Symmetric

Logical Stripe No.				
0	0	1	2	P0
2	8	P2	6	7
4	12	13	14	P4
6	20	P6	18	19
8	24	25	26	P8
1	4	5	P1	3
3	P3	9	10	11
5	16	17	P5	15
7	P7	21	22	23
9	28	29	P9	27

Figure 5: Physical Versus Logical Placement. This figure illustrates the physical and logical variations of the left-symmetric placement for a two-by-five array and a two-by-four array.

3 IO Request Servicing

The RAID driver recognizes two basic types of IO requests from Sprite: reads and writes. In addition, the RAID driver recognizes various IO controls for configuring/initializing the RAID device and for performing recovery of failed disks. The basic steps followed in servicing IO requests are as follows.

1. Break up IO requests into stripe requests. From here on, each stripe request is processed independently. The IO request completes successfully if and only if all of the corresponding stripe requests complete successfully. Note that breaking up IO requests into stripe requests at this level and handling each stripe request independently means that physically contiguous requests, generated by a very large logical request which wraps to the same disk, are not merged into a single request but are processed via separate physical requests to the disk. Figure 2 illustrates requests that may be merged.

   The main reason for not merging requests was simplicity of implementation. Merging logically discontiguous requests implies recopying data to make it logically contiguous, playing games with the virtual memory mapping, or using disks which support scatter-gather operations.

2. Lock required stripes. This is necessary to guarantee the consistency of the parity information associated with each stripe.

3. For each stripe request, choose and execute an appropriate *IO method*.

4. Unlock stripes.

3.1 IO Methods

The RAID driver supports the following five IO methods which are illustrated by Figure 6.

- Read
- Read-Modify-Write
- Reconstruct-Write
- Reconstruct-Read
- Nonredundant-Write

An appropriate IO method is selected to service a stripe request by examining the request type (read or write), request size, and RAID device specific state information. The selected IO method then breaks up the stripe request into physical disk requests to be processed in parallel.

The primary RAID state information concerns the validity of disk blocks. A block is marked invalid when the disk containing the block fails. The block remains invalid until the content of the invalid block is recovered to a functional disk. It is necessary to maintain the above information for two reasons. First, a disk failure can be intermittent, in which case a read to a failed disk, for which a previous write failed, may succeed, returning stale (not the most recently written) data. Second, when the failed disk is replaced, one needs a mechanism for determining whether a particular block has been recovered. Currently, a single counter per disk is used as a "high water mark" of valid blocks. Thus, when a disk fails, blocks must be recovered sequentially starting from zero. A bitmap will eventually be used to support a wider range of recovery strategies.

3.1.1 Read

Read is the simplest IO method. The read method is selected if all of the blocks read by a stripe read request are valid. The read method simply reads those blocks and only those blocks that are requested. If one of the physical requests needed to service the stripe request fails, the read method invokes the reconstruct-read method. If more than one physical request fails, the stripe request fails.

3.1.2 Read-Modify-Write

The read-modify-write method is invoked if a relatively small portion of the stripe is written. In such a case, it is more efficient to update the parity incrementally by xoring the new data, old data and old parity than by xoring the new data with the rest of the stripe. Currently, the read-modify-write method is used in preference to the reconstruct-write method if less than half of the stripe is being written. The read-modify-write method is executed in three distinct non-overlapping stages:

1. Read old data and old parity.
2. Compute new parity.
3. Write new data and new parity.

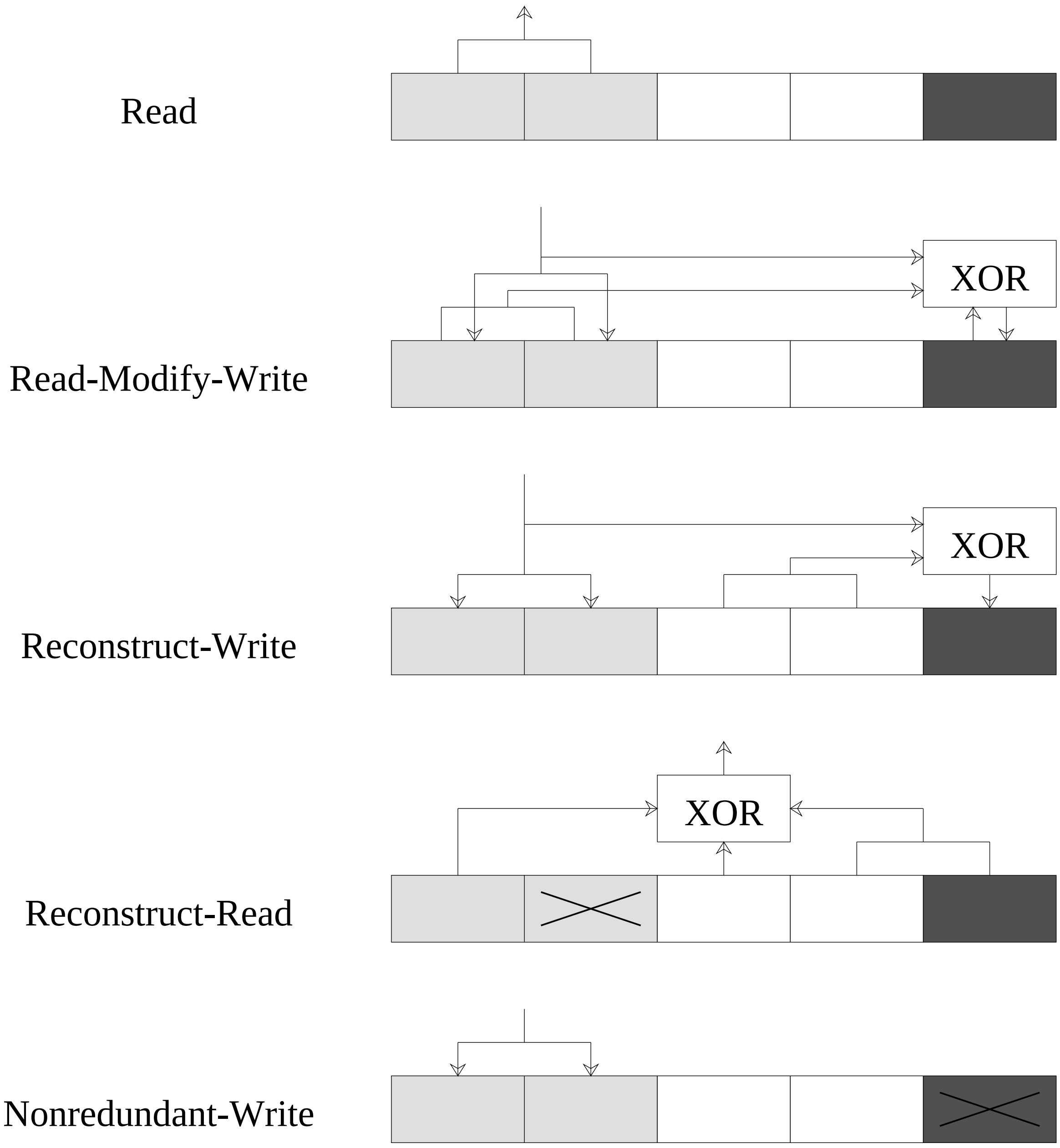

Figure 6: IO Methods. The lightly shaded regions correspond to data that is to be read or written. The dark regions correspond to parity. The 'X' denotes a failed disk.

Executing the read-modify-write method in non-overlapping stages simplifies the implementation but results in a slight penalty in response time (I estimate this penalty at less than 10 percent for very small writes and up to 25 percent for larger writes.). The performance penalty can be eliminated by overlapping the stages:

1. Write the new data as soon as the old data is read without waiting for the read of the old parity to complete.

2. Acknowledge the request as soon as the new data is written without waiting for the write of the new parity.

The key to recovering from physical request failures during the execution of the read-modify-write and reconstruct-write methods is to recognize that the two methods can be used to recover from each others.

A single physical request failure during the read stage causes the reconstruct-write method to be invoked unless this method was called to recover from the reconstruct-write method in which case the stripe request fails. More than one physical request failure causes the stripe request to fail. A single physical request failure during the write stage causes the blocks of the disk corresponding to the failed component IO to be invalidated; i.e. the disk is assumed to have failed, but the stripe request succeeds. More than one physical request failure causes the IO request to fail.

3.1.3 Reconstruct-Write

The reconstruct-write method is invoked if a relatively large portion of the stripe is written. Currently, the reconstruct-write method is used in preference to the read-modify-write method if half or more of the stripe is being written. The reconstruct-write method is executed in three distinct non-overlapping stages.

1. Read data from rest of stripe.

2. Compute parity.

3. Write new data and new parity.

If an exact stripe is being written, step one is trivial and does not require physical IO.

At first, it may seem possible to read the data from the rest of the stripe in parallel with writing the new data and therefore reduce the time it takes to do a reconstruct-write by assuming that the parity update, which is redundant, can be done in the "background" after the stripe request has been acknowledged. This causes a problem, however, if after initiating the read and write physical requests simultaneously, one of the read requests fail. In such a case, the data on the failed block can never be reconstructed since we have just overwritten a part of the stripe without reading it and hence the parity is inconsistent.

A single physical request failure during the read stage causes the read-modify-write method to be invoked unless this method was called by the read-modify-write method in which case the stripe request fails. More than one physical request failure causes the stripe request to fail. A single physical request failure during the write stage causes the blocks of the disk corresponding to the failed component IO to be invalidated; i.e. the disk is assumed to have failed, but the stripe request succeeds. More than one physical request failure causes the IO request to fail.

3.1.4 Reconstruct-Read

The reconstruct-read method is invoked if one of the blocks covered by a stripe read request is invalid. The contents of the invalid blocks are computed with the aid of parity. If called by the read-method, those blocks which have already been read are not reread. Any physical request failure causes the stripe request to fail.

3.1.5 Nonredundant-Write

The nonredundant-write method is invoked if the blocks containing the parity are invalid. In such a case, there is no need to update the the parity, so a simple write of the data is sufficient. Any physical request failure causes the stripe request to fail.

4 Recovery and Data Consistency

In the event of disk failure, the RAID driver satisfies user requests by reconstructing, on demand, lost data from redundant information. In addition, the RAID driver allows recovery of failed disks and reconstruction of parity on system startup to proceed concurrently with user request servicing in order to reduce periods of data unavailability. The above implies that care must be taken to ensure data consistency; it is not permissible to return stale (not the most recently written) data or incorrectly reconstructed data as valid. Data consistency must be maintained in the face of power and system failures where the contents of main memory can be lost. In the following, the term *reconstruction* refers to the computation of data or parity via XOR's while the term *recovery* refers to the reconstruction of the contents of a failed disk onto a replacement disk.

4.1 Consistency States

The following state information is used by the RAID driver to maintain data consistency:

- stripe-unit-state
  (VALID/INVALID)
- disk-state
  (READY/FAILED)
- stripe-state
  (CONSISTENT/INCONSISTENT)
  (LOCKED/UNLOCKED)

The most important state is the stripe-unit-state which describes the logical state of a stripe unit. Any changes in the stripe-unit-state *must* be logged to stable storage before user requests involving the stripe-unit can be serviced. The disk-state is used during data recovery and parity reconstruction to prevent the inadvertent recovery/reconstruction of information onto failed disks.

To guarantee the consistency of the parity, stripes are locked before any part of the stripe is read or written. A stripe is consistent if the parity is up to date; otherwise, it is inconsistent. During normal operation, a stripe is considered consistent if and only if it is unlocked and the corresponding parity stripe unit is valid. Immediately after a system crash, all stripes that were locked at the time of the crash are considered inconsistent. If it is not possible to determine which stripes were locked at the time of the crash, all stripes must be considered inconsistent; i.e. all parity stripe units must be reconstructed.

4.2 System Startup

The following steps must be taken when starting up the RAID driver after a system crash.

1. Recover disk-state from log. (Not absolutely necessary but highly recommended.)
2. Recover stripe-unit-state from log. This includes invalidating parity stripe units corresponding to inconsistent stripes.
3. Accept user requests and reconstruct invalid parity and data.

5 Investigation of Parity Placement

This section evaluates the performance of the RAID level 0, RAID level 4, right-asymmetric, left-asymmetric, right-symmetric, left-symmetric, extended-left-symmetric and flat-left-symmetric parity placement schemes (all the placements illustrated in Figure 3) via simulation.

5.1 The RAID Simulator

The simulator was constructed by interfacing the RAID driver with a model of disk behavior and a program to generate synthetic IO requests[3]. In all, the simulator consists of approximately 10,000 lines of C. The only hardware resources modeled are disks. The disks are rotationally synchronized. The simulator merges physically contiguous disk requests. The request types used are random reads and random writes of various sizes. Sequential requests are not used because the sequentiality is much less meaningful when there is more than one process generating requests to the same disk (the disk will thrash between the two sequential request streams). The load on the system is controlled by specifying the number of concurrent processes which issue requests. The number of processes, request type and request size are fixed for each simulation run.

5.2 The Disk

Table 1 tabulates the parameters of the simulated disk. Seek times are calculated with the following equation:

$$\begin{aligned} \mathit{seekTime} &= 0 && \text{if } x = 0, \\ &= a(x-1)^{0.5} + b(x-1) + c && \text{if } x > 0. \end{aligned}$$

Where x is the seek distance in cylinders and a, b and c are constants chosen to satisfy the single cylinder, max stroke and average seek times. For the simulated disk, $a = 0.4623, b = 0.0092$ and $c = 2$. Figure 7 plots the seek time modeled by the above function versus the seek distance.

[3] I am indebted to Garth Gibson and Peter Chen for supplying the disk model and program to generate synthetic IO requests respectively.

cylinders per disk	949
tracks per cylinder	14
sectors per track	48
bytes per sector	512
track skew	4
revolution time	13.9 ms
single cylinder seek time	2.0 ms
average seek time	12.5 ms
max stroke seek time	25.0 ms
max sustained transfer rate	1.7 MB/s

Table 1: Disk Parameters

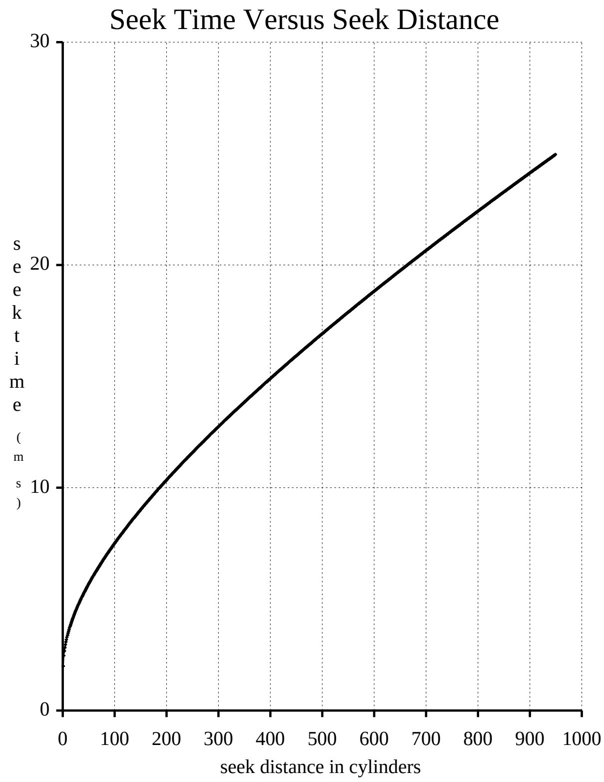

Figure 7: Seek Profile. A plot of the seek time in milliseconds versus the seek distance in cylinders for the simulated disk. The plot was obtained using the following equation:

$$\begin{aligned} seekTime &= 0 && \text{if } x = 0, \\ &= 0.4623(x-1)^{0.5} + 0.0092(x-1) + 2 && \text{if } x > 0. \end{aligned}$$

5.3 Simulation Parameters

The following is a list of the input and output variables used in the simulation and their corresponding ranges.

- Input Variables.
  - Parity placement scheme (one of the eight listed above).
  - Number of rows of disks in array (1 or 2).
  - Number of disks per row (5 or 9).
  - Size of the stripe unit (4KB, 8KB, 16KB, 32KB, or 64KB).
  - Request type (read or write).
  - The size of the request (2KB to 1MB).
  - Request alignment (aligned or unaligned). Aligned requests are aligned on their own size boundaries. Unaligned requests are aligned on sector boundaries (512 byte).
  - The degree of concurrency; i.e. the number of processes generating requests (1, 2 or 16). When the concurrency equals one, the average response time is proportional to the inverse of throughput. The term *low load* refers to a degree of concurrency of one. The term *high load* refers to a degree of concurrency of sixteen.
- Output Variable.
  - Throughput, measured in megabytes per second.

Note, in the interests of reducing the parameter space, all workloads are homogeneous; i.e. mixtures of reads and writes as well as non-constant distributions of request sizes are not used.

The cross product of all input variables was considered too large to simulate and analyze properly, as well as being unnecessary. We also did not have reasonable ranges for the input variables when we started. Thus, only specific subsets of the cross product were investigated. The selection of the subsets and the investigation of the results proceeded in an iterative manner to reduce the possibility of missing interesting points in the cross product. In all, approximately 20,000 points in the cross product were investigated.

5.4 Simulation Results

Figures 8, 9, 10 and 11 illustrate the simulated performance of each parity placement under a set of particular inputs and is characteristic of other input combinations which are not illustrated here.

At low load, the graphs display a sawtooth pattern with a period approximately equal to 300KB[4]. The first dip occurs when requests become large enough to wrap around the array. At this point, some of the disks must read/write two stripe units while other disks read/write only one stripe unit. Some of the disks end up waiting for the other disks, resulting in inefficient resource utilization. This behavior is repeated each time the request wraps around the array, resulting in a periodic behavior. Note that at low load and relatively large requests sizes, the choice of parity placement results in up to a 20 to 30 percent difference in performance for the array configuration simulated.

[4] The length of the period is a function of the logical to physical mapping and the size of the array.

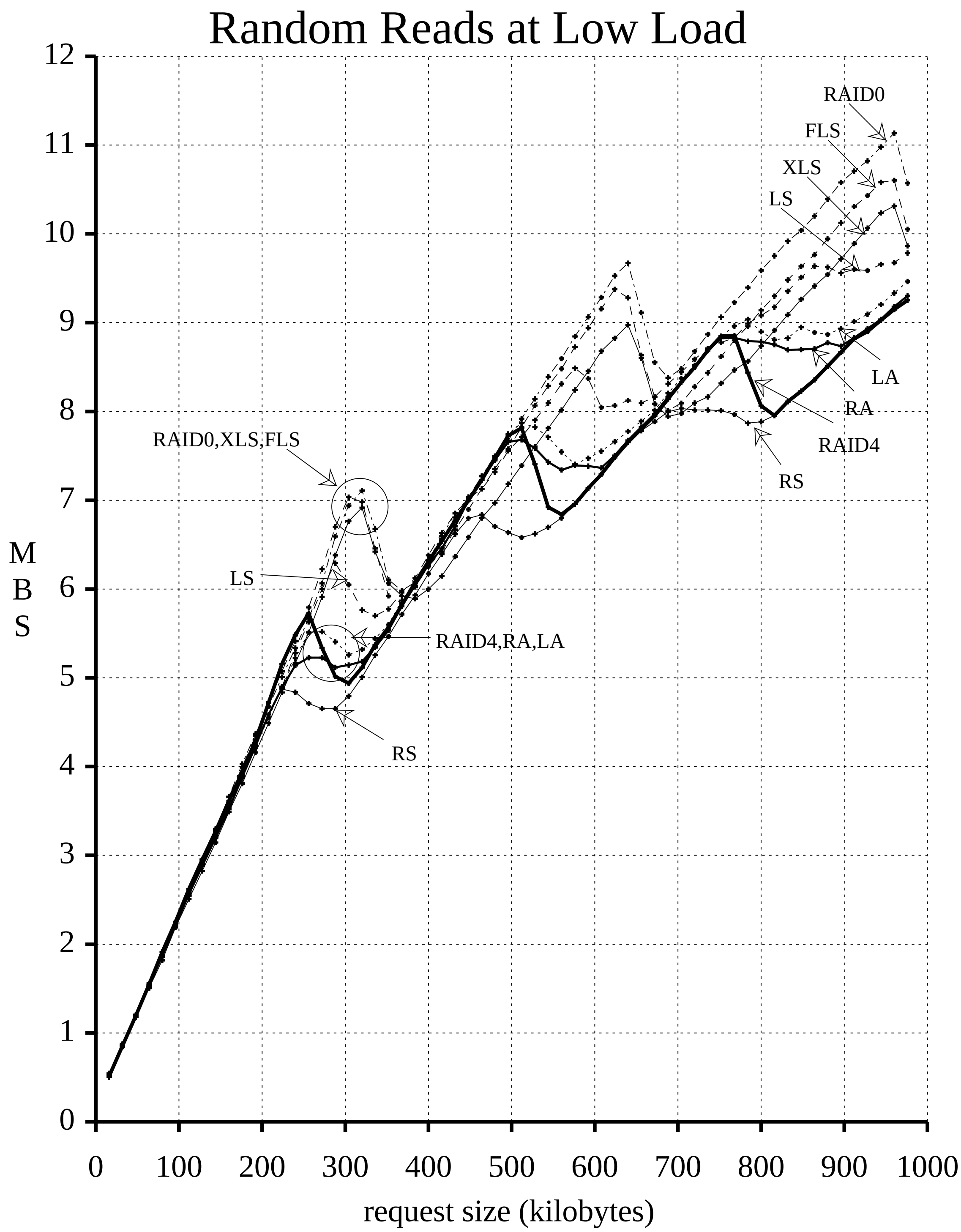

Figure 8: Reads at Low Load. The simulation parameters are two rows, five disks per row, 32KB stripe units, and unaligned requests. The degree of concurrency for low load is one and the degree of concurrency for high load is sixteen.

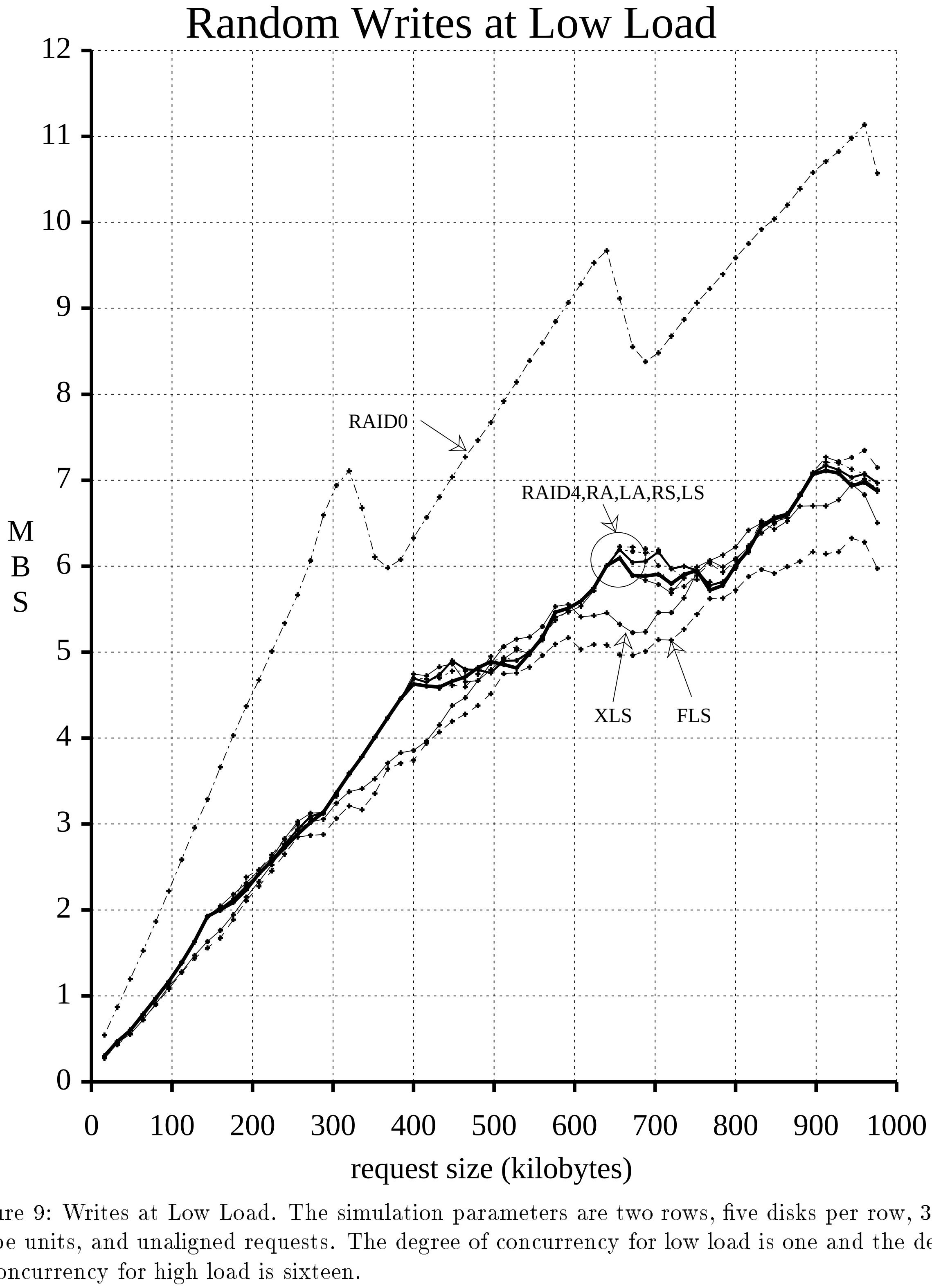

Figure 9: Writes at Low Load. The simulation parameters are two rows, five disks per row, 32KB stripe units, and unaligned requests. The degree of concurrency for low load is one and the degree of concurrency for high load is sixteen.

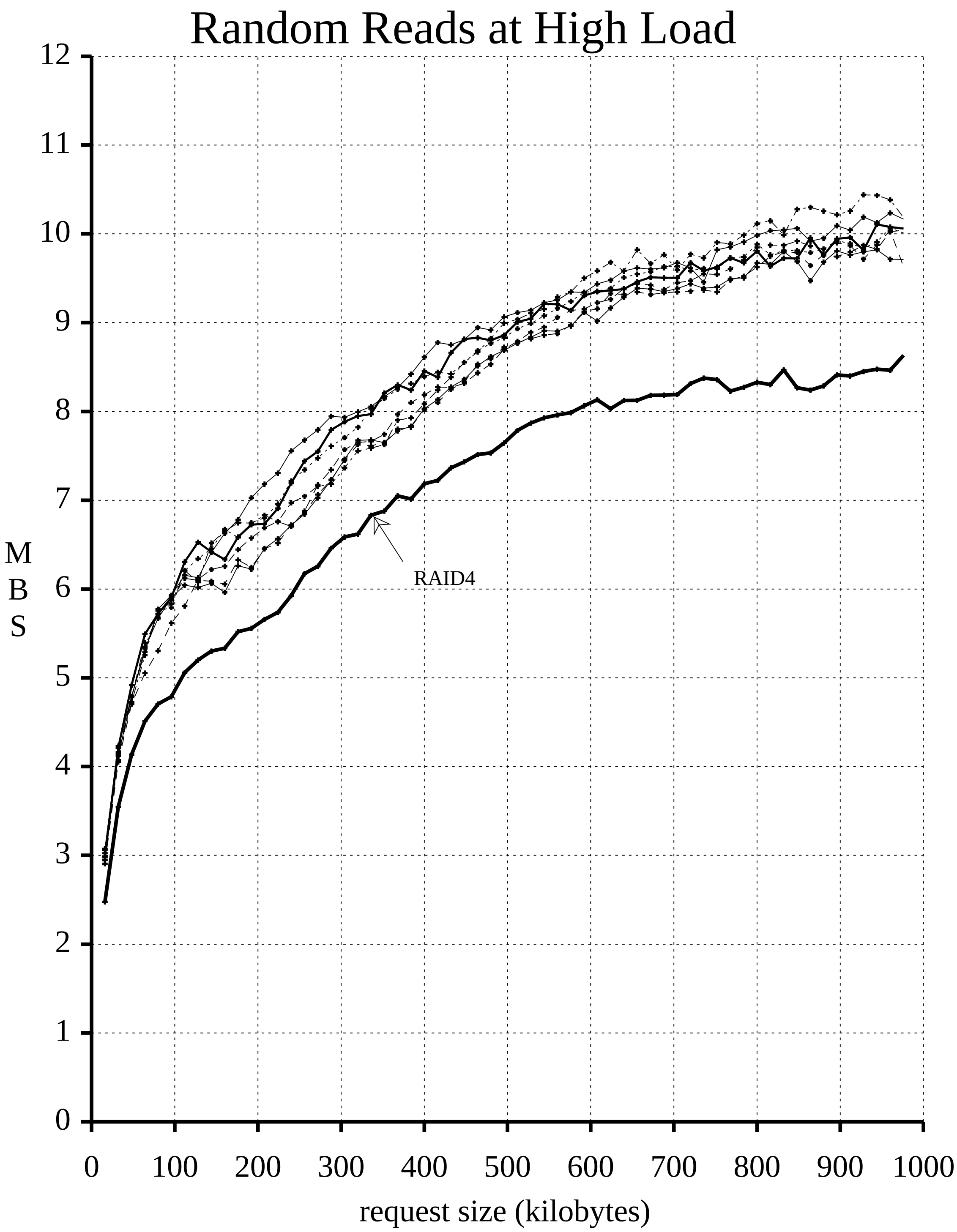

Figure 10: Reads at High Load. The simulation parameters are two rows, five disks per row, 32KB stripe units, and unaligned requests. The degree of concurrency for low load is one and the degree of concurrency for high load is sixteen.

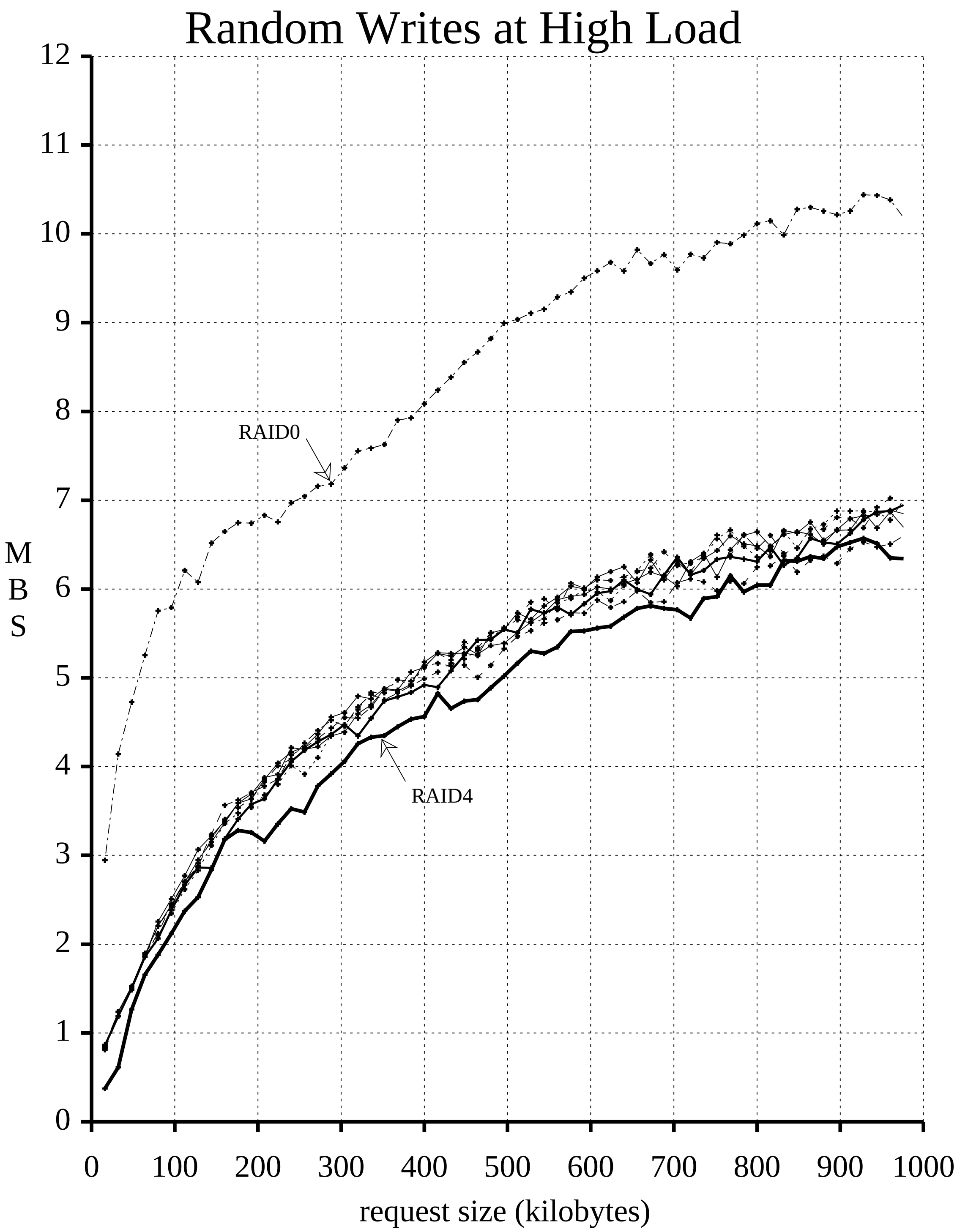

Figure 11: Writes at High Load. The simulation parameters are two rows, five disks per row, 32KB stripe units, and unaligned requests. The degree of concurrency for low load is one and the degree of concurrency for high load is sixteen.

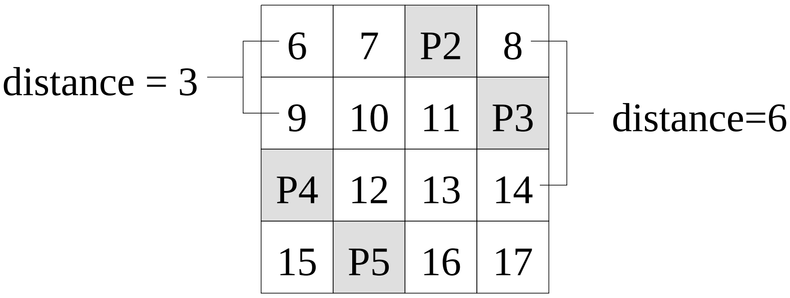

Figure 12: Placement Distance

5.4.1 Reads at Low Load

When performing reads at low load, roughly four groups of placements can be distinguished based on performance. From highest performance to lowest performance they are grouped as follows:

1. RAID level 0, extended-left-symmetric and flat-left-symmetric.
2. Left-symmetric.
3. RAID level 4, right-asymmetric and left-asymmetric.
4. Right-symmetric.

We define the *minimum placement distance* of a placement, which gives a good prediction of read performance at low load, as the minimum arithmetic difference between logical data stripe unit addresses of stripe units that are logically sequential on the same disk. Figure 12 illustrates the concept of distance. Parity stripe units are ignored when determining minimum placement distances. The minimum placement distance gives an indication of the number of disks a parity placement can stripe data over before reusing a disk and hence the amount of parallelism available for servicing a single request. The larger the minimum placement distance, the better the read performance at low load. We will refer to a placement with a minimum placement distance of N as a *distance N placement*. The maximum minimum placement distance for an array with N disks is N. Table 2 tabulates the minimum placement distances for the parity placement so far considered.

The RAID level 0 placement outperforms the extended-left-symmetric placement even though they have the same minimum placement distance because the RAID level 0 placement never has to skip over parity stripe units. The flat-left-symmetric placement outperforms the extended-left-symmetric placement even though they have the same minimum placement distance because the flat-left-symmetric placement aligns the parity stripe units together so that on large reads, all parity stripe units can be skipped simultaneously.

5.4.2 Writes at Low Load

When performing writes at low load, roughly four groups of placements can be distinguished based on performance. From highest performance to lowest performance they are grouped as follows:

1. RAID level 0.

	minimum placement distance
RAID level 0	mn
Flat-Left-Symmetric	mn
X-Left-Symmetric	mn
Left-Symmetric	$m(n-1)+1$
Left-Asymmetric	$m(n-1)$
RAID level 4	$m(n-1)$
Right-Asymmetric	$m(n-1)-1$
Right-Symmetric	$m(n-1)-1$

Table 2: Parity Placement Distances. The symbol m denotes the number of rows in the array and the symbol n denotes the number of disks per row. Note that the left-symmetric and extended-left-symmetric placements are identical placements when the number of rows is one.

2. RAID level 4, right-asymmetric, left-asymmetric, right-symmetric and left-symmetric.
3. Extended-left-symmetric.
4. Flat-left-symmetric.

As expected, the RAID level 0 placement – no redundancy – displays the highest performance since it does not maintain parity. The extended-left-symmetric placement performs worse than the placements in the second group because it places the parity of the previous stripe on the same disk as the data of the current stripe; e.g. note that in the extended-left-symmetric placement illustrated in Figure 3, $P0$ is on the same disk as stripe unit 4. Thus, if two sequential parity stripes are written, a single disk must service both a parity access and a data access. The extended-left-symmetric placement performs better than the flat-left-symmetric placement because it aligns parity with its corresponding data stripe units, whereas the flat-left-symmetric placement aligns the parity together. Note that this was the same reason the flat-left-symmetric-placement outperformed the extended left-symmetric-placement on reads at low load.

5.4.3 Reads at High Load

The RAID level 4 placement has the worst read performance at high load because it does not distribute parity and data over all disks. Thus, only $n-1$ rather than n disks are available for servicing read requests. The performance of the other placements, which distribute parity and data, are comparable. At high load, the most important criterion for high read performance is uniform disk utilization; i.e. placements which distribute the load uniformly are desirable.

5.4.4 Writes at High Load

As expected, the RAID level 0 placement displays the highest write performance at high load since it does not maintain parity. Note that because the redundant placements must write parity, their write performance asymptotically approachs $(n-1)/n$, $4/5$ in the case of Figure 11, of the maximum

RAID level 0 performance at large request sizes. The RAID level 4 placement displays the worst write performance at high load due to contention for the parity disks. For small request sizes (< 32KB) and these particular simulation parameters, the difference in performance between the RAID level 4 placement and the other placements is over a factor of two. With more disks per row and at higher load, we would expect the performance difference to be even larger. Thus, for high write performance at high load, placements which distribute the parity uniformly are desirable.

5.4.5 Summary

In this section, we have examined the performance characteristics of eight parity placements. The RAID level 0 placement is unsuitable since it does not support parity. The left-symmetric placement outperforms the RAID level 4, right-asymmetric, left-asymmetric and right-symmetric placements on reads at low load over almost all request sizes and has comparable or better performance in all of the other cases. Thus, the choice to be made is between the left-symmetric, extended-left-symmetric and flat-left-symmetric placements. Which placement is used for a particular system depends on the importance of read performance at low load versus write performance at low load. For the reader's benefit, the performance of the above three placements at low load is redisplayed in Figure 13 with the other placements omitted. To make the differences easier to evaluate, Table 3 tabulates the absolute throughput and throughput relative to the left-symmetric placement for a few key requests sizes ($2\ rows \times 5\ columns \times 32KB\ stripeunits = 320KB$) from Figure 13.

Note that the relative performance of the left-symmetric, extended-left-symmetric and flat-left-symmetric placements converge as the request size increases for reads but remains relatively unchanged for writes. Note also that the flat-left-symmetric placement achieves only 90% of the performance of the left-symmetric placement for 16KB request sizes.

We propose the following list of desirable placement properties, roughly in the order of their importance:

1. Stripe units belonging to the same parity stripe should not occupy the same column. (In many RAID systems, the disks within a column have a common failure mode; e.g. the string interface.) This is referred to as the "orthogonal RAID" property. All of the proposed placements have this property.

2. In a RAID with n disks per row, the ith parity stripe unit should correspond to stripe unit j such that $j \operatorname{div} n = i$. This guarantees that any write request that is stripe aligned and a stripe in size can be written without reading old data. All of the proposed placements have this property.

3. Parity and data should be distributed over all disks. All of the proposed placements except the RAID level 4 placement have this property.

4. The minimum placement distance should be maximized. Only the extended-left-symmetric and flat-left-symmetric placements are maximum distance placements. The left-symmetric placement is a maximum distance placement for a single row of disks but not for multiple rows of disks.

5. If the number of rows is n then stripe units belonging to any n consecutive parity stripes should not occupy the same disk. This avoids the write contention experienced by the extended-left-

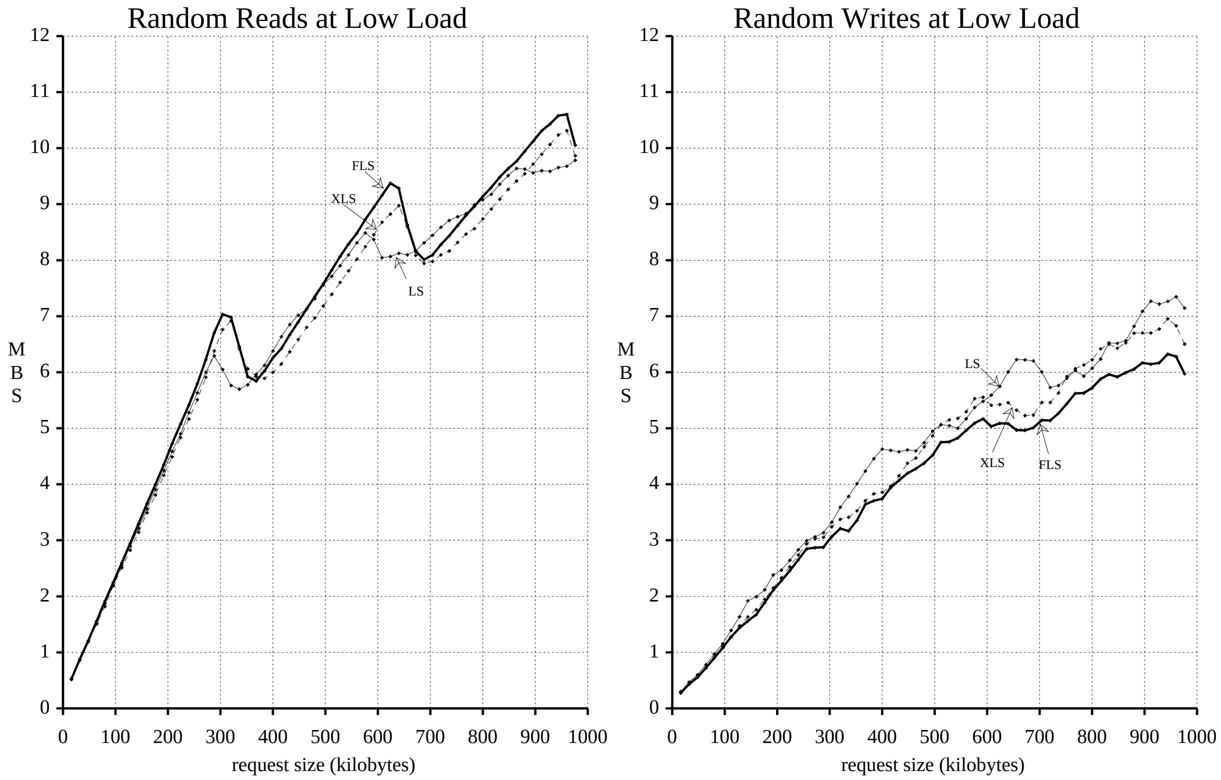

Figure 13: Parity Placement Performance Summary. The simulation parameters are two rows, five disks per row, 32KB stripe units, unaligned requests, and the degree of concurrency for low load is one.

Random Reads at Low Load

	16KB		320KB		640KB		960K	
	MBS	% of LS	MBS	% of LS	MBS	% of LS	MBS	% of LS
Left-Symmetric	0.52	100%	5.77	100%	8.12	100%	9.68	100%
X-Left-Symmetric	0.53	102%	6.92	120%	8.98	111%	10.31	107%
Flat-Left-Symmetric	0.52	100%	6.98	121%	9.28	114%	10.60	110%

Random Writes at Low Load

	16KB		320KB		640KB		960K	
	MBS	% of LS	MBS	% of LS	MBS	% of LS	MBS	% of LS
Left-Symmetric	0.30	100%	3.59	100%	6.01	100%	7.35	100%
X-Left-Symmetric	0.29	97%	3.38	94%	5.46	91%	6.83	93%
Flat-Left-Symmetric	0.27	90%	3.21	89%	5.08	85%	6.28	85%

Table 3: Parity Placement Performance Summary. *MBS* denotes throughput in megabytes per second. *Percentage of LS* denotes throughput relative to the left-symmetric parity placement.

symmetric and flat-left-symmetric placements which are the only proposed placements which do not satisfy this property.

Unfortunately, properties 4 and 5 seem to be at odds with one another. That is, given an array with N disks, it does not seem possible to devise a placement which is both distance N and satisfies Property 5 for a general m-by-n disk array. With only a single row of disks, the left-symmetric, extended-left-symmetric and flat-left-symmetric placements are optimal with respect to both properties. This would imply that when selecting a parity placement for a general disk array, one must sacrifice either read performance or write performance. A key question to be answered in a later section is whether there exists a placement which is distance N and satisfies Property 5 for disk arrays with multiple rows of disks.

5.5 Computational Overhead

The previous section has investigated the performance of parity placements without considering the amount to CPU time needed to compute each placement. This section quantifies the computational overhead incurred in computing each parity placement. Table 4 summarizes the measured CPU overheads.

	relative mapping overhead	% of RAID level 0 overhead
RAID level 0	1.00	100%
RAID level 4	1.00	100%
Right-Asymmetric	1.16	103%
Left-Asymmetric	1.17	103%
Right-Symmetric	1.13	103%
Left-Symmetric	1.15	103%
X-Left-Symmetric	1.49	110%
Flat-Left-Symmetric	1.09	102%

Table 4: Computational Overhead. *Relative mapping overhead* denotes the relative amount of CPU time needed to map a single stripe unit using the given mapping. *Percentage of RAID level 0 overhead* denotes the relative RAID driver time (software overhead excluding SCSI drivers) of the specified placement algorithm relative to a RAID driver using the RAID level 0 placement. For all of the placement algorithms, approximately 20 percent of the total RAID driver time is spent in the mapping code. The time needed to map a single stripe unit with the RAID level 0 placement on a DECstation 3100 (12-15 MIPS) is approximately 16μS.

Of all the placements, the extended-left-symmetric placement incurs the highest CPU overhead, and requires almost 50 percent more CPU time to compute than the RAID level 0 placement; however, when the rest of the RAID driver software is included, a RAID driver configured with the extended-left-symmetric placement uses only 10 percent more CPU time than a RAID driver configured with the RAID level 0 placement. We feel that once other factors such as the operating system and SCSI drivers are factored in, it is unlikely that the 50 percent relative mapping overhead will significantly affect performance. Of course, the above measures the performance of only a *specific implementation* of each mapping algorithm, still, the computational overhead does not seem to be an important factor in the choice of parity placements.

5.6 A Systematic Look at Parity Placements

The above sections have compared the performance of eight different parity placements and derived properties that are desirable of parity placements. The question remains, however, whether there exists a placement which is superior to the eight placements so far considered. This section attempts to answer this question by systematically generating and examining parity placements[5]. We first look at placements for a single row of disks and then generalize our method to look at placements for multiple rows of disks.

5.6.1 Single Row Placements

This section takes a systematic look at parity placements for arrays with a single row of disks by considering all three-by-three parity placements. Consider a three-by-three checkerboard where the columns correspond to distinct disks and the squares to stripe units. We define a *parity placement* for a single row of disks as any column indistinct permutation of distinct stripe units; i.e. shuffling the columns does not change the parity placement. Since the columns are indistinct, the number of possible ways in which nine distinct stripe units, six of which are data and three of which are parity, can be placed on the checkerboard is $9!/3! = 60480$. This is clearly too many to examine by hand. In order to reduce the number of distinct cases to consider, we will initially ignore the ordering of stripe units within each column and look at the number of ways in which the nine distinct stripe units can be partitioned over the three indistinct columns. Thus, each column forms a set of size three which we will refer to as a *placement group*. The three placement groups taken together will be referred to as a *placement class*.

There exist $9!/(3!3!3!3!) = 280$ distinct placement classes and each placement class corresponds to $3!3!3! = 216$ distinct parity placements. If we restrict placement classes to those which place parity and its corresponding data stripe units on different disks and further specify that parity, and hence data, must be uniformly distributed over all disks, the number of placement classes drops down to eight (the eight placement classes were derived by exhaustively generating placement classes and eliminating those that did not satisfy the above requirements). Figure 14 illustrates the eight placement classes which are labeled with instances of their corresponding parity placements. Note that each asymmetric class gives rise to a "family" of placements differing only on the column offset at which the rotation is started.

Now that we have identified eight feasible placement classes, it is time to consider which parity placements derivable from each class are the most useful. Since the data stripe units within each column should obviously be placed in ascending order, the question becomes one of where to place the parity. For all placement classes except the left-symmetric placement class, the parity should almost certainly be aligned with its corresponding data stripe units; i.e. there is no advantage to not aligning the data and parity. This ensures that the data and corresponding parity stripe units will be rotationally synchronized, and when writing only exact parity stripes, all of the disk heads will travel equal distances. Placing the data stripe units in ascending order and aligning the

[5] I am indebted to Garth Gibson for suggesting that I take a more systematic approach to identifying parity placements.

	Symmetric			Asymmetric(0)			Asymmetric(1)			Asymmetric(2)		
	P0	0	1	P0	0	1	0	P0	1	0	1	P0
Right	3	P1	2	2	P1	3	2	3	P1	P1	2	3
	4	5	P2	4	5	P2	P2	4	5	4	P2	5
	0	1	P0	P0	0	1	0	P0	1	0	1	P0
Left	3	P1	2	2	3	P1	P1	2	3	2	P1	3
	P2	4	5	4	P2	5	4	5	P2	P2	4	5

Figure 14: Three By Three Parity Placement Classes

parity and its corresponding data results in the family of right-asymmetric placements, the family of left-asymmetric placements, the right-symmetric placement and the left-symmetric placement. For the extended-left-symmetric placement class, there are two reasonable ways to place the parity. The first method, as already described, is to align the parity with its corresponding data. The second method is to align all of the parity with each other. This ensures that on large reads, all disk heads will skip over the parity sectors simultaneously. The two methods of placing the parity within each row give rise to the left-symmetric and flat-left-symmetric placements.

Thus, we conclude that the most useful parity placements derivable for a single row of disks from a three-by-three grid are the family of right-asymmetric placements, the family of left-asymmetric placements, the right-symmetric placement, the left-symmetric placement and the flat-left-symmetric placement. It would be useful to characterize the placements derivable for general m-by-n disk arrays. Due to the lack of time and other resources, we leave these as possibilities for future work.

5.6.2 Multi Row Placements

This section takes a systematic look at parity placements for arrays with multiple rows of disks by considering all three-by-three parity placements for two rows of disks. Consider two three-by-three checkerboards where the columns of each checkerboard correspond to distinct disks and the squares to stripe units. Each column within each checker board forms a set of size three which we will refer to as a *placement group*. A placement group represents the stripe units placed together on a single disk. The set of placement groups within the same column is referred to as a *column group*. With two rows of disks, there are two placement groups per column group. The set of all column groups will be referred to as a *placement class*. Figure 15 illustrates the above concepts. A parity placement is derived from a placement class by imposing an ordering on the stripe units within each placement group. Thus, in our example, there are $18!/(3!3!^{6}2^{3}) = 2,858,856$ placement classes

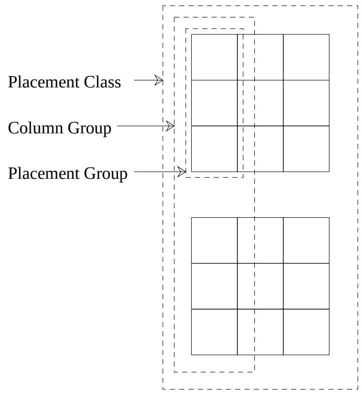

Figure 15: Parity Placement Definitions

and each placement class corresponds to $3!^6 = 46,656$ parity placements (columns are unordered $\rightarrow 3!$, stripe units within each of six placement groups unordered $\rightarrow 3!^6$, placement groups within column groups unordered $\rightarrow 2^3$).

In what follows, we implicitly restrict placement classes to those which place parity and its corresponding data stripe units on different disks and uniformly distributes parity and data over all disks. We will refer to such a placement as a *RAID level 5 placement.* Note that the results which follow, unless otherwise indicated, have been derived by brute force; i.e by enumeration of all possibilities.

A key question to be answered in this section is whether there exists a RAID level 5 placement which is distance six and satisfies Property 5, i.e. places stripe units belonging to any two consecutive parity stripes on different disks. An enumeration of all possible placement classes indicates that there does not exist such a placement.

The number of RAID level 5 placements which are distance six but do not satisfy Property 5 is sixty-four. The extended-left-symmetric and flat-left-symmetric placements are derived from one of these sixty-four classes. The other sixty-three classes can be generated from the extended-left-symmetric class by swapping the parity stripe units within column groups, and by swapping the set of data stripe units within a placement group with another such set containing data stripe units of the same parity stripe. Figure 16 illustrates each type of swap. There are eight ways to perform the former swap and eight ways to perform the latter swap, resulting in a total of $8 \times 8 = 64$ different placement classes. The performance of placements derived from the other sixty-three placement classes is difficult to predict with certainty although we would not expect them to be significantly different from the extended-left-symmetric and flat-left-symmetric placements.

The number of RAID level 5 placement classes which are distance five and satisfy Property 5, as illustrated by Figure 17, is six. The placement classes differ only in the grouping of placement groups to form column groups. The six placement classes would produce identical performance in our simulator.

5.7 Summary

We have examined the performance and computational overhead of eight different parity placements, derived desirable properties for parity placements, and analytically examined a large space

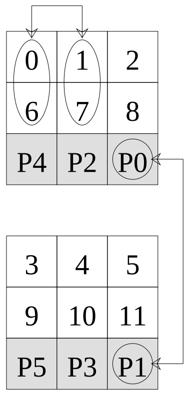

Figure 16: Distance Six Multi-Row Parity Placement Classes

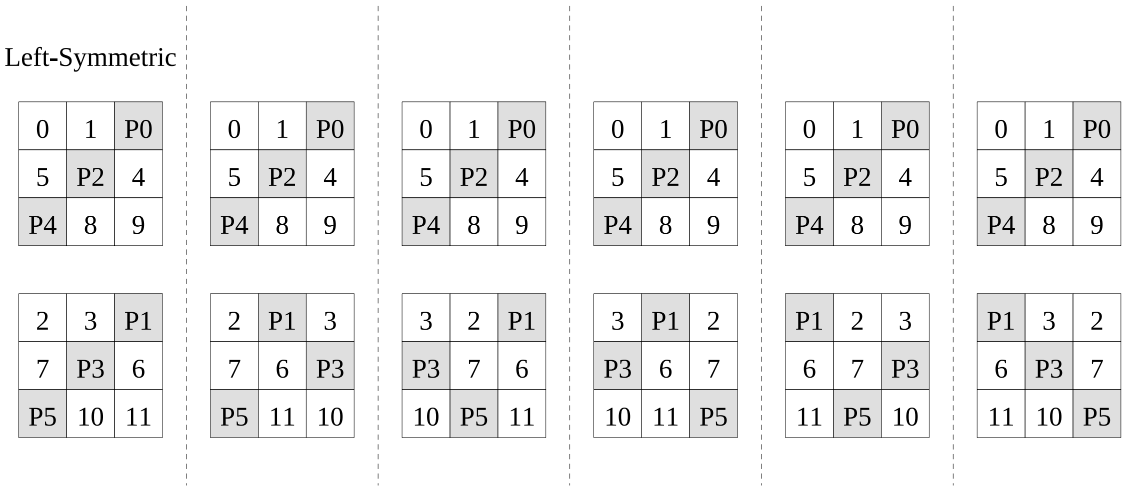

Figure 17: Distance Five Multi-Row Parity Placement Classes

of possible placements. The latter shows that there does not seem to exist a placement which is capable of significantly higher performance than those that we have already examined (In truth, the analytical analysis initially yielded placements of superior performance which we added to the list we had at that time to derive our current list!).

Given what we now know of parity placements, the following "recommendations" can be made:

- The performance of small reads and writes is insensitive to the parity placement.
- If the performance of large reads at low loads is of primary importance, than the flat-left-symmetric placement is the best.
- If the performance of large writes at low loads is of primary importance, than the left-symmetric placement is the best.
- At high loads, all of the redundant placements with the exception of the RAID level 4 placement, which does not distribute the parity, perform equally well.
- In general, if the workload of a system is not well known, the left-symmetric placement is an all-around good placement.

6 Conclusion

We have described the software of RAID-I, including the data mapping, parity placement, how user requests are serviced and how data consistency is maintained. In addition, we have classified and investigated the performance of a variety of parity placement schemes in detail via simulation and analytical methods. We have shown that the left-symmetric, extended-left-symmetric and flat-left-symmetric are the best parity placements. We have also shown that, of the three placements mentioned above, the placement with the highest read performance (flat-left-symmetric) has the lowest write performance and the placement with the lowest read performance (left-symmetric) has the highest write performance. Additional work remains to be done in investigating other RAID parameters such as the stripe unit [1] and row sizes and their impact on performance.

7 Acknowledgements

I would like to thank my advisor, Randy Katz, members of the RAID group (especially Peter Chen, Ann Chervenak and Garth Gibson) members of the Sprite group (especially Mendel Rosenblum and Mary Baker) and our government and industrial affiliates (Array Technologies, DARPA, DEC, Eastman Kodak, Hewlett-Packard Labs, IBM, Intel Scientific Computers, NASA, NSF, Seagate, Sun Microsystems and Thinking Machines Corporation) for making this work possible. I am also greatly indebted to David Patterson for reviewing this report and making numerous helpful suggestions. I would also like to thank John Ousterhout for supervising the initial implementation of the RAID software under Sprite.

References

[1] Peter M. Chen and David A. Patterson. Maximizing performance in a striped disk array. In *Proc. International Symposium on Computer Architecture*, pages 322–331, May 1990.

[2] Peter C. Dibble. A parallel interleaved file system. Technical Report CS TR 334, University of Rochester, March 1990.

[3] R. H. Katz, G. A. Gibson, and D. A. Patterson. Disk system architectures for high performance computing. In *Proc. IEEE*, volume 77, pages 1842–1858, December 1989.

[4] Michelle Y. Kim. Synchronized disk interleaving. *IEEE Trans. on Computers*, C-35:978–988, November 1986.

[5] Michelle Y. Kim and Asser N. Tantawi. Asynchronous disk interleaving. Technical Report RC12497, IBM, January 1987.

[6] Miron Livny, S. Khoshafian, and H. Boral. Multi-disk management algorithms. In *Proc. SIGMETRICS*, pages 69–77, May 1987.

[7] David A. Patterson, Peter M. Chen, Garth Gibson, and Randy H. Katz. Introduction to redundant arrays of inexpensive disks (RAID). In *Proc. IEEE COMPCON*, pages 112–117, Spring 1989.

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

[9] A. L. Narasimha Reddy and Prithviraj Banerjee. An evaluation of multiple-disk I/O systems. *IEEE Trans. on Computers*, 38:1680–1690, December 1989.

[10] K. Salem and H. Garcia-Molina. Disk striping. In *Proc. IEEE Data Engineering*, pages 336–342, February 1986.