A Redundant Disk Array Architecture for Efficient Small Writes (CMU-CS-94-170)

Daniel Stodolsky
Carnegie Mellon University

Mark Holland
Carnegie Mellon University

William V. Courtright II
Carnegie Mellon University

Garth A. Gibson
Carnegie Mellon University

Follow this and additional works at: http://repository.cmu.edu/pdl
A Redundant Disk Array Architecture for Efficient Small Writes
Daniel Stodolsky, Mark Holland, William V. Courtright II, and Garth A. Gibson

July 29, 1994
CMU-CS-94-170

School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213-3890


Abstract

Parity encoded redundant disk arrays provide highly reliable, cost effective secondary storage with high performance for reads and large writes. Their performance on small writes, however, is much worse than mirrored disks - the traditional, highly reliable, but expensive organization for secondary storage. Unfortunately, small writes are a substantial portion of the I/O workload of many important, demanding applications such as on-line transaction processing. This paper presents parity logging, a novel solution to the small write problem for redundant disk arrays. Parity logging applies journalling techniques to substantially reduce the cost of small writes. We provide detailed models of parity logging and competing schemes - mirroring, floating storage, and RAID level 5 - and verify these models by simulation. Parity logging provides performance competitive with mirroring, but with capacity overhead close to the minimum offered by RAID level 5. Finally, parity logging can exploit data caching more effectively than all three alternative approaches.

This research was supported by the ARPA, Information Science and Technology Office, under the title “Research on Parallel Computing”, ARPA Order No. 7330, the National Science Foundation under contract NSF ECD-8907068, and by IBM and AT&T graduate fellowships. Work furnished in connection with this research is provided under prime contract MDA972-90-C-0035 issued by ARPA/CMO to CMU. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of ARPA, NSF, IBM, AT&T, or the U.S. government.
Keywords: Redundant disk arrays, RAID level 5, Parity logging.
1. INTRODUCTION

The market for disk arrays, collections of independent magnetic disks linked together to form a data store, is undergoing rapid growth and has been predicted to exceed 13 billion dollars [DiskTrend94]. This growth has been driven by three factors. First, the growth in processor speed has outstripped the growth in disk data rate. This imbalance transforms traditionally compute-bound applications to I/O-bound applications. To achieve application speedup, I/O system bandwidth must be increased by increasing the number of disks. Second, arrays of small diameter disks offer substantial cost, performance advantages over larger drives. Third, such systems can be made highly reliable by storing a small amount of redundant information. Without this redundancy large disk arrays have unacceptably low data reliability because of their large component disks. For these three reasons, redundant disk arrays, also known as Redundant Array of Inexpensive Disks (RAID), are strong candidates for nearly all on-line secondary storage [Patterson88, Gibson92].

Figure 1 presents an overview of the RAID systems considered in this paper. The most promising variant, RAID level 5, employs distributed parity with data striped on a unit that is one sector.

RAID level 5 arrays exploit the low cost of parity encoding to provide high data reliability. Data is striped over all disks so that large files can be fetched with high bandwidth. By distributing parity many small random blocks can also be accessed in parallel without hot spots on any disk.

While RAID level 5 disk arrays offer performance and reliability advantages for a wide

Fig. Data Layout in RAID Levels 0, 1, 4 and 5. This figure shows the first few units on each disk in each level. “D” represents a block of user data (of unspecified size, but some multiple of one sector) computed over user data units x through y. The numbers on the left indicate the offset into the raw disk units. Shaded blocks represent redundant information, and non-shaded blocks represent user data. Examples are shown for striped data. RAID Level 0 (Nonredundant) and RAID Level 1 (Mirroring) are simple mirror copies of each data unit. RAID Level 4 (Block-Interleaved Parity) uses a single block of parity for the entire disk. RAID Level 5 (Rotated Block-Interleaved Parity) uses a single block of parity for each disk. Levels 4 and 5 differ in that in level 5, the parity blocks rotate through the array rather than being concentrated on a single sector.

1. In current industry usage, the “I” in RAID denotes “independent”.
applications, they possess at least one critical limitation: their throughput is penalized four over nonredundant arrays for workloads of mostly small writes [Patterson88]. This penalty because a small write request may require the old value of the user's targeted data be read (we call this a preread), overwriting this with new user data, prereading the old value of the corresponding disk, then overwriting this second disk block with the updated parity. In contrast, systems based on mirrored disks simply write the user's data on two separate disks and, therefore, are only penalized by a factor of two. This disparity, four accesses per small write instead of two, has been termed the small write problem.

Unfortunately, small write performance is important. The performance of on-line transaction processing (OLTP) systems, a substantial segment of the secondary storage market, is determined by small write performance. The workload described by Figure 2 is typical of TP benchmarks and nearly the worst possible for RAID level 5, where a single read-modify-write of an account can require five disk accesses. The same operation would require three accesses on mirrored disks and two on a nonredundant array. Because of this limitation, many OLTP systems continue to employ the much more expensive option of mirrored disks.

This paper describes and evaluates a powerful mechanism, parity logging, for eliminating this small write penalty. Parity logging exploits well understood techniques for logging or journaling small random accesses into large sequential accesses. Section 2 of this paper describes the parity logging mechanism. Section 3 introduces a simple model of its performance and cost. Section 4 describes alternative disk system organizations, develops comparable performance models, and contrasts them to parity logging. Section 5 provides an analysis of small-write overheads in parity logging with respect to configuration and workload parameters, and analyzes potential load imbalances in a parity logging system, Section 6 introduces our simulation system, describes implementation of parity logging and alternative organizations, and contrasts their performance on workloads of random writes and an OLTP workload. Section 7 analyzes extensions to multiple-failure tolerant arrays. Section 8 discusses how the large write optimization can be accommodated in a parity logging disk array. Section 9 reviews related work. Section 10 closes with a few comments on future work.

2. PARITY LOGGING

This section develops parity logging as a modification to RAID level 5. Our approach is motivated by the fact that disks deliver much higher bandwidth on large accesses than they do on small. Logging disk array batches small changes to parity into large accesses that are much more efficient. Our model is introduced in terms of a simple, but impractical RAID level 4 scheme, then refined to a realistic implementation used in our simulations.

The duration of a disk access can be broken down into three components: seek time, positioning time, and data transfer time. Small disk writes make inefficient use of disk bandwidth because the sum of the seek and positioning times is small compared to the data transfer time.
because the data transfer component is much smaller than the seek and rotational position components. Thus, a disk servicing a small-access-dominated workload spends the majority of its time positioning instead of transferring data. Figure 3 illustrates the relative bandwidths of random block, track, and cylinder accesses for a modern small-diameter disk [IBM0661]. This figure largely bears out the lore of disk bandwidth: random cylinder accesses move data twice as fast as random track accesses, which, in turn, move data ten times faster than random block accesses.

<table>
<thead>
<tr>
<th>Disk 0</th>
<th>Disk 1</th>
<th>Disk 2</th>
<th>Disk 3</th>
<th>Disk 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data</td>
<td>Data</td>
<td>Data</td>
<td>Parity</td>
<td>Log</td>
</tr>
</tbody>
</table>

Fig. 4: Basic Parity Logging Model

<table>
<thead>
<tr>
<th>D</th>
<th>Data units per track</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Cylinders per disk</td>
<td>949</td>
</tr>
<tr>
<td>S</td>
<td>Average seek time</td>
<td>12.5 ms</td>
</tr>
<tr>
<td>R</td>
<td>Average rotational delay</td>
<td>95 ms</td>
</tr>
<tr>
<td>H</td>
<td>Head switch time</td>
<td>1.16 ms</td>
</tr>
<tr>
<td>B</td>
<td>Number of regions per disk</td>
<td>100</td>
</tr>
<tr>
<td>C_L</td>
<td>Cylinders of log per region</td>
<td>~20</td>
</tr>
<tr>
<td>C_D</td>
<td>Cylinders of parity per region</td>
<td>~20</td>
</tr>
<tr>
<td>K</td>
<td>Tracks buffered per region</td>
<td>1</td>
</tr>
<tr>
<td>L</td>
<td>Log striping factor</td>
<td>1</td>
</tr>
</tbody>
</table>

Fig. 5: Model Parameters. The bandwidth utilization models of Section 2, 3, and 4 are presented in the parameters list above. The first table presents common disk parameters and the second, parameters used for sequential seeks. The first and fourth columns in each table show the symbol used in the text; the second and fifth columns show the parameter as used in the models; and the third and last column show the default value of the parameter as used in the models. The first and fourth columns in each table show the symbol used in the text; the second and fifth columns show the parameter as used in the models; and the third and last column show the default value of the parameter as used in the models.

Logically, we develop our scheme beginning with Figure 4 in which a RAID level 4 disk is augmented with one additional disk, the log disk. Initially, this disk is considered empty. When a RAID level 4 disk receives a write that needs to be performed on an ordinary disk, the new write overwrites the old user data, then overwrites the parity. Instead of following this, we allow the parity update image (the result of an XOR operation between the old user data and new user data) to be held in a dedicated block of memory called the log buffer. When the log buffer becomes full, the out-of-date parity and the log of parity update records are removed from memory using sequential cylinder accesses. The logged parity update images are applied to the memory image of the stale parity and the resulting updated parity is written with large writes. When this completes, the log disk is marked empty and the logging cycle begins again.

Fig. 3: Peak I/O Bandwidth. This figure shows the sustained data rate in kilobytes per second that was written to an IBM 0661 drive using random one block (2KB), one track (24 KB), and one cylinder (336 KB) for disk parameters.
Because only parity updates (not data changes) are deferred, this scheme preserves single-disk tolerančće. If a data disk fails, the log disk (and any buffered parity updates) are first recorded and then used to reconstruct the lost data in the same manner as is done for RAID level 5. If the log or parity disk fails, the system can simply recover by reconstructing parity onto the surviving parity or log disk. The failed drive is then replaced with a new empty disk. If the controller fails, its buffered parity updates are lost, but, after the controller has been replaced, parity can be updated in the same way as if a log disk had been lost.

The addition of a log disk allows substantially less disk time to be devoted to parity updates than in a comparable RAID level 4 or 5 array. Assume there are B data units per track, T tracks per cylinder, and C cylinders per disk (refer to the glossary in Figure 5). Each user write requires a corresponding data unit, which introduces an overhead of one block (data unit) access per memory transfer. In addition, each user write to a data unit consumes buffer memory equal to the size of the user data unit. The read of a data unit's worth of small (unit-sized) writes issued to the array causes one track write to occur. Next, a disk's worth of small writes causes the log disk to fill up, which must then be emptied by updating the parity disk. This update involves reading the entire contents of the parity disk (2V cylinders), and then writing the entire parity disk (2V cylinders) at cylinder transfer rates. On average, then, for every small user write, there are D block accesses, sequential track accesses, and cylinder accesses for maintenance of the parity information. Recall track access is D times larger than random small writes but about 10 times more efficient and cylinder access is twice as efficient as track access. Thus, parity maintenance for a disk's worth of small user writes consumes about as much disk time as:

\[ TVD + TV(D/10) + 3V(T/2 \times D/10) = TVD/4 \]

random small accesses. In a standard RAID level 4 or parity maintenance, the ratio of parity maintenance work performed by parity logging to RAID level 4 or 5 is therefore

\[
\frac{5TVD/4}{3TVD} = \frac{5}{12}
\]

Thus, by logging parity updates, we have reduced the disk time consumed by parity maintenance by about a factor of two.

In many cases, it may be possible to avoid the preread of the user data. For example, in a benchmark (Figure 2), the update of a customer account record is a read-modify-write operation. The account record is read, modified in memory, and then written back to disk. In these cases, the old data block is usually known (cached) at the time of the write and the preread of the data may be avoided [Menon93]. Under these conditions, the overhead for RAID levels 4 or 5 is just two random block accesses per small write, whereas for parity logging is

\[ TV(D/10) + 3V(T/2 \times D/10) = TVD/4 \]

random small accesses. Therefore, in these cases, parity logging reduces disk time consumed by parity maintenance by about a factor of eight.

2.1. Partitioning the Log Into Regions

As stated, however, this scheme is completely impractical: an entire disk's worth of random access memory is required to hold the parity during the application of the parity updates. Figure 5 shows that this limitation can be overcome by dividing the array into manageable-sized regions. Each region is a miniature replica of the array proposed above. Small user writes for a particular
journalled into that region. When a region's log fills up, only that region's log is required to update that region's parity. This reduces the size of the controller memory buffer needed during reintegration from the size of a disk to a manageable fraction of a disk. Section 2.4 shows the number of regions is dependent on disk capacity, but is about 100 in our example 22-disk array.

Each region requires its own log and parity buffer. Each log buffer holds a few (typically less than three) of parity update images. When one of these buffers fills up, the corresponding region's log is appended with an efficient track (or multi-track) write. Thus, the sequential track writes of the single-region layout are replaced by random track (or multi-track) writes in the multiple-region layout. While write operations are less efficient than sequential track writes, Section 3 will show that this implementation still has dramatically lower parity maintenance overhead than RAID level 4 or 5.

2.2. Striping Log and Parity for Parallelism

As in the RAID level 4 case, the log and parity disks may become performance bottlenecks in many disks in the array. The maximum aggregate bandwidth for log accesses is just the bandwidth of single disk. This limitation can be overcome by distributing parity and log across all the disks in the array, as indicated in Figure 6(b). This distribution boosts the aggregate bandwidth to the bandwidth of the array, but remains that of a single disk.

Following the example of RAID level 5, Figure 7 shows a layout in which the parity for each region is distributed across the array to increase bandwidth. This distribution decreases the time required for re-reading parity updates for a particular region by accessing all disks to effect the parity read and write. So that these operations are also efficient, the granularity of distribution is one contiguous set of parity units per disk per region. The log, however, remains a potential bottleneck.

The log bottleneck may also be eliminated by distributing the log for each region over multiple disks. Figure 8 shows a parity logging array with the log for each region striped across two disks. Data update records in the log are logically placed on the same disks as the data they protect. If they were, the failure of that disk would cause both data and parity to be lost, which is an unrecoverable failure in a disk array using a parity-based code. To avoid such data loss, data and log for each region are restricted to disjoint sets of disks. Thus, log striping reduces the number of data cylinders per region, thereby reducing the disk space overhead.

This reduction in data striping in a region increases the number of disks over which each log is striped and the number of cylinders of parity per region. The number of data cylinders per region, related to the size of the parity region, is given by the standard RAID level 4 and 5 rule for data striped disk.
where $N$ is the number of disks in the array. Because the log is equal in size to the parity, $C_P$, the number of cylinders of log per region, equals $C_L$. Hence, the disk space overhead (the fraction of the array containing log and parity) equals

$$(C_P + C_L)/(C_L + C_P + C_D) = 2/(N - L + 1)$$

and rises as the degree of log striping increases. Figure 9 shows the disk space overhead for different degrees of log striping for an array of 22 disks. Section 6 will show that the performance advantages of log striping are substantial.

### 2.3. The Impact of Varying Log Length

The previous subsection assumes that the same amount of disk space (cylinders) and parity update logs are preserved for all regions. However, the number of parity update logs required is equal to the number of parity regions. Therefore, the number of parity update logs required is $2^R$. Figure 8 shows the distributed parity logs.

**Fig. Distributed Parity Logs**
parity \(C_p\) cylinders) is allocated in each region because our introduction adds exactly one parity array. Given the more flexible striped log and parity model of Figure 8, the efficiency overheads of parity logging can be altered by increasing or decreasing the amount of log space per region.

Let \(A\) be the ratio of total log space to total parity space \(C_p\) per region. The disk space overhead then becomes

\[
\frac{C_L + C_p}{C_L + C_p + C_D} = \frac{AC_p + C_p}{AC_p + C_p + (N-L-1)C_p} = \frac{1+A}{N-L+A}
\]

Now the log for each region fills \(A\) times after small user writes into that region. Updating the parity still requires prereading old data on each small user write (assuming the old data is not cached), writing the log buffers \(A\) times, plus, every time the log fills, reading \(C_p\) parity cylinders, reading the log \(C_p\) cylinders), and writing the updated parity \(C_p\) cylinders). Thus the parity maintenance work for uncached small user writes is

\[
ATC_pD + ATC_p\left(\frac{D}{10} + (2+A)C_p\left(\frac{T \times D}{10}\right)\right) = \left(\frac{23}{20} + \frac{1}{10A}\right)ATC_pD
\]

random small accesses, or an overhead of \(10(23/20 + 1/10A)\) random small accesses per uncached small user write. Performance can therefore by traded for space, as shown in Figure 10. Applying the example 22 disk array with logs striped over two disks (\(A = 2\)) increases the space overhead from 9.5% to 13.6% of the total capacity, the parity maintenance overhead from 41.7% to 40% of that of RAID level 5, where three related parity images occur for each small user write. Halving the amount of parity logging does reduce parity maintenance work from 12.5% to 10% of RAID level 5 while halving the size reduces parity maintenance work from 12.5% to 10% of RAID level 5 while halving increases the work to 17.5% of RAID level 5.

2.4. Accounting for and Managing Buffers

The primary benefit of parity logging, that parity maintenance operations access disks under efficient transfers, requires expensive controller memory buffers. This buffer memory is allocated in ways. First, each region delays the most recent parity update images until efficient log-apply operations can be performed. If \(D\) tracks are transferred in a log-append operation, then, conservatively, \(D\) tracks of buffer memory are required to delay log append. Second, whenever the log for a region fills, the parity for that region is read from the disk by buffers.
to it, and the updated parity is written back. This parity reintegration operation requires buffer memory, where \( C_P \) is the number of cylinders of parity per region and \( N T \) is the number of tracks per cylinder. Since the number of cylinders of parity per region is the same as the total number of tracks on a disk, \( N \), divided by the number of regions, \( N_R \), total buffer memory space is \( \frac{C_P N T}{N_R} \) tracks.

By selecting \( C_P = \sqrt{TV/K} \), the memory buffer space is minimized. If the ratio of the cost of a byte of memory and a byte of disk is \( \frac{C_M}{C_D} \), then the buffer memory space cost, relative to the cost of an array of disks \( 2X \cdot TVK/(NTV) = 2X/N \cdot TV/K \). If memory costs 30 times as much as disk [Feigel94], then an array of 22 IBM 0661 (Figure 12) disks buffering a single log track \((K = 1)\) requires about 5.6 MB of buffer, the equivalent of about 2% of the array's space.

In practice, parameters such as the number of regions must be discrete. If we further require the size per region of the log appends, sublogs (the portion of a region's log on one disk), parity, and data, per region, to be an integral number of tracks, then a significant fraction of the space may be wasted. We have found that if the number of regions is allowed to vary from the optimum by \( \pm 10\% \), then a set of integral parameters can be found such that the wasted disk space is less than 1% of the array's space.

If, however, the size per region of the sublogs, parity, and data, per region, are only required to be integral number of disk sectors (rather than tracks), substantially less disk space is wasted, as \( N = 1 \). Relaxing this discrete-tracks condition will cause additional head switches and single cylinder seeks to occur during log and parity operations, but because these positioning overheads are small relative to track access times, parity logging performance is only slightly affected (3% for our experiments).

A more significant performance degradation results if small user writes are blocked during reintegration of a region's log into its parity. This blocking should be minimized by managing the per-region buffers as a single global buffer pool. Using this approach, user writes are only delayed until the entire buffer pool is full of parity update images that have not yet been appended to the logs.

### 2.5. Summary

In summary, parity logging buffers parity updates until they can be written to a log efficiently. It further delays their reintegration into a redundant parity array until there are enough parity updates in the log to make a complete revision of the disk array's parity and data. With limited memory for reintegration of parity records, the disk array is partitioned into regions with per-region parity logging on. Then, to avoid bandwidth bottlenecks, parity and log information is striped over multiple disks. The parity logging scheme reduces the extra work done by RAID level 5 arrays for small random writes, as little more than is done in the much more expensive, traditional mirrored approach even with RAID 6 arrays.

![Parity Logging vs. RAID level 5](image1.png)

![Parity Logging vs. RAID level 5](image2.png)

Fig. 1 Log Length and Efficiency
caching is ineffective.

3. MODELING PARITY LOGGING

In this section we present a utilization-based analytical model of a parity logging array. This model predicts saturated array performance in terms of achieved disk utilization, geometry, and access size. The variables used in this model are defined in Figure 5.

Consider a single small user write in a parity logging array. In a parity logging array, each data must be preread, then overwritten. This is done in an access which seeks to the cylinder on which the data to rotate under the head, reads the data, waits for the disk to rotate around once, then overwrites the data. Defining $S$ as the average seek time, $R$ as the time for one-half of a disk rotation, and recalling that $D$ is the number of data units per track, the time to perform this operation,

$$t_{\text{w}} = (S + R) + 2R/D + (2R - 2R/D) + 2R/D = S + (3 + 2/D)R$$  \hspace{1cm} (1)

disk seconds, on average.

As mentioned earlier, many cases it may be possible to predictably avoid the preread of data. Without prereading, the disk busy time needed for a small write access,

$$t_{\text{w}} = S + (1 + 2/D)R$$  \hspace{1cm} (2)

disk seconds.

Each region has $K$ tracks worth of log buffers. On average, if a single writer writes, one region's buffers will fill and be written to the region's single track write. Defining the disk's head-switch time, the number of disk seconds required to do this,

$$t_{\text{K,track}} = (S + R) + 2RK + (K - 1)H = S + (2K + 1)R + (K - 1)H$$  \hspace{1cm} (3)

assuming all $K$ tracks are on the same cylinder.

Finally recall that each region consists of cylinders, each of which contains $N$ data units. Therefore, on average, for every small user write, one region of logged parity must be reintegrated. Consider the case of an array that does not stripe its log (Figure 7). The region consists of three steps: a sequential cylinder seek to the log, a striped read of the parity disks, and a striped write of the parity back onto disks.

$$t_{\text{L}} = (S + R) + C_L(2RT + (T - 1)H) + M(C_L - 1)$$  \hspace{1cm} (4)

4. This single access could be separated into two accesses of $S + 2R/D$ disk seconds for a total of $2S + (4 + 4/D)R$.

5. Disks that support zero-latency writes [Salem86] can eliminate the initial rotational positioning delay the I/O time by up to 26% in drives such as the IBM 0661 (which does not support this feature), if only a single track is used. However the impact of zero-latency write support on parity logging is small (under 3%), because the track-seek overhead is relatively small (under 3%).
disk seconds, and may be rewritten as

$$t_{CL} = S + (2TC_L + 1)R + (T - 1)HC_L + (C_L - 1)M$$

(5)

The striped parity accesses each consist of sequential transfers of cylinders. Each of these sequential transfers takes

$$t_{CP} = (S + R + C_L/N - 1)(2RT + (T - 1)H) + (C_L/N - 1)M$$

(6)

disk seconds. The total striped parity access, $t_{CP}$, takes

$$t_{amw} = t_{rmw} + \frac{1}{KD} [t_{track}] + \frac{t_{CL}}{DTC_L} + \frac{2t_{CP}}{DTC_L}$$

(7)

Figure 11 shows the contributions to disk busy time of the three terms in equation 7 for the example disk array given in Figure 12.

The analysis for a parity logging disk array with a striped log and similar log buffer fills, it will be written to one of the regions. The cost of this operation is the same as in the unstriped case. Log reintegration still takes every writes, but now consists of three striped I/Os: a log read of the log, and a striped (over $N$ disks) read and write of the parity. The accesses in the striped log read costs

$$t_{CL}(L) = L(S + R + C_L(2RT + (T - 1)H) + (C_L - L)M$$

(8)

disk seconds. Similarly, the striped parity reads and writes will consume...
Thus, striping introduces an additional overhead of disk seconds to the log reintegration. This increases the parity maintenance overhead per small write work. As Section 6 will show, this increase in parity maintenance work is worthwhile because it reduces long reintegration periods during which disk queues grow, the system becomes underutilized, and maximum performance falls far short of expectations.

4. MODELING ALTERNATIVE SCHEMES

Only a few array designs have addressed the problem of high performance, parity-based, disk arrays for small write workloads. The most notable of these is floating data and parity [Menon92]. Reviews and estimates the performance of four designs: nonredundant disk arrays (RAID 0), mirrored disks (RAID level 1), distributed N+1 parity (RAID level 5), and floating data and parity. The notation and analysis methodology are the same as used in Section 3.

In nonredundant disk arrays (RAID level 0), a small write requires a single disk access which consumes disk-arm seconds. No long-term storage is required in the controller. In mirrored systems, every data unit is stored on two disks, and all write requests are copies. Each access takes as much time as a small write in a nonredundant disk array. Hence, each small user write utilizes disks for disk seconds. While mirrored disks’ write operations are more efficient than RAID 5, their capacity is devoted to...
redundant data. As in the RAID level 0 case, controllers for mirrored disk arrays do not require long-term buffer memory.

Small writes in RAID level 5 disk arrays require four accesses: data preread, data write, and parity write. These can be combined into two read-rotate-write accesses, each of which takes disk seconds for a total disk busy time of . Again, no long-term controller storage is required.

The floating data and parity modification to RAID level 5 was proposed by Menon and Kasson [Menon92]. In its most aggressive variant, this technique organizes data and parity into cylinders containing either only data or parity, as illustrated in Figure 13, by maintaining a single track of space per cylinder. Data and parity effectively eliminates the extra rotational delay that RAID level 5 read-rotate-write accesses. Instead of updating the data or parity in place, a floating data and parity array will write modified information into the rotationally nearest free block. With floating data and parity, the rotational term in the RAID level 5 disk arm busy time expression above is replaced with a head switch and a short rotational delay. Using disks similar to those in our sample array, Menon and Kasson report an average delay of 0.76 data units. So, the expected disk busy time for each access in a floating data and parity array is

\[
(\frac{S+R}{D} + H + 0.76(\frac{2R}{D}) + 2R) \cdot T
\]

which may be rewritten as \((1 + 5.52D)R + H\). Hence, the total disk busy time for a small random user write in a floating data and parity array is \((2 + 11.04D)R + 2H\). Note that if the number of data units per track, \(C\), is large and the head-switch time is small, this is close to the performance of mirroring.

Even with a spare track in every cylinder, data and parity arrays still have excellent storage overheads. For a disk array with \(N\) tracks per cylinder, data and parity has a storage overhead of \((T + N - 1)/(TN)\). Floating data and parity arrays require substantial fault-tolerant storage in the array controller to keep track of the current location of data and parity. 

---

6. Each disk gives \(T\) units capacity for free space and the array gives \(T\) units extra free space. The storage efficiency is \(NT-1\) and the array storage overhead is \(T+N-1\).
each cylinder, an allocation bitmask is maintained. This requires \( \log(DT) \) bits per cylinder. Thus, with cylinders per disk, a total of \( (1+\log(DT)) \) bits of fault-tolerant controller storage are required. For the disks in Figure 12, this is \( 3,608 \) bytes per disk. The total controller storage in a 22 disk array is about 3,608KB, roughly comparable to parity logging. Note, however, that controller memory in parity logging need not be fault-tolerant.

While floating data and parity substantially improves the performance of small writes relative to RAID level 5, its performance for other types of accesses is degraded. Within a cylinder, logically contiguous user data units are not likely to be physically contiguous. In the worse case, two different tracks may end up at the same rotational position, requiring disk rotation to read both. In addition, the average seek time of roughly data units. Thus, even on disks with zero-latency reads, the maximum sequential read bandwidth is reduced.

5. ANALYSIS

Figure 14 compares these models’ estimates for maximum throughput of the example array based on Figure 12. Throughput at lower utilizations may be calculated by scaling the maximum throughput numbers by the disk utilization. Figure 14 predicts that parity logging and floating data arrays both substantially improve on RAID level 5, approaching the performance of mirroring. Varying the model’s parameters from our example 22 disk array does not substantially change the relative performance of parity logging and its alternatives except for the effects of the number of data units per track and the ratio of average seek time to rotational latency. This section describes the effects of these parameters and the effects of log striping degree on array load balance.

Of the model parameters, the number of data units per track has the greatest impact on performance. Parity logging transfers each data unit two more times than RAID level 5 and three more times than mirroring. If the transfer time of a unit is small, parity logging will be ineffective. Figure 14 shows the relative performance when data caching is ineffective (i.e., a preread is required) for parity logging, mirroring, and RAID level 5 for different values of numbers of data unit per track. The performance of mirroring exceeds that of parity logging with 13 or fewer data units per track and RAID level 5 performance exceeds that of parity logging with the unlikeliness of 1 or 2 data units per track. Industry estimates, however, show that track capacity within a given form factor is increasing at over 20% a year. Consequently, it is reasonable to assume that the number of data units per track may not decrease even as database account record sizes grow.

The ratio of average seek time to rotational latency has a substantial impact on the performance of parity logging disk arrays. Figure 16 plots the performance of parity logging relative to RAID level 5 and mirroring relative to RAID level 5 as this ratio changes. The performance of parity logging is reduced by a factor of 1.5 when the seek time is reduced to one-third of the rotation time. This trend is expected since parity logging relies on user-level caching. The nature of fault tolerance in a storage controller depends on the underlying failure model. If only power failure is of concern, then nonvolatile storage will suffice, while other failure models require redundant controllers.
achieves as much benefit from decreased seek time as nonredundant arrays because its two disks are each equivalent to the single nonredundant access. RAID level 5 and parity logging, however, do more rotational work for each seek so decreasing seek time relative to rotational latency increases performance relative to nonredundant arrays. Consequently, advantage of parity logging over RAID level 5 decreases as the seek time to rotational latency ratio decreases. This ratio is nearly unity for all modern drives, and shows no particular trend in any direction.

Figure 14 assumes the user requests access data uniformly. This assumption is reasonable for huge OLTP databases, other workloads may exhibit substantial locality. In the worse case, all user I/O
is concentrated within one region. Choosing an appropriate data stripe unit [Chen90] will partition user I/O across the actuators that contain data for this "hot" region. Data traffic are partitioned over non-overlapping disks. If this traffic is not balanced, parity logging performance will fall short of Figure 14.

The log, parity and data traffic can be balanced by determining the appropriate degree of log stripe width L. Recall that ewTC_L small user writes (er = t_2 and c_L are the number of data units per track, tracks per cylinder, and cylinders of log per region, respectively) to disks of a particular region will cause writes of tracks to the striped log, and then a full log read and full write of the parity for that region to effect parity reintegration. Writes are spread out over all disks, so a uniform load is maintained if the work per data disk is equal to the work per sublog disk. That is,

$$\frac{DTC_{LZ}}{N-L} = \frac{(TC_{L}/K)t_{\text{track}} + t_{c_L}}{L}$$

(10)

where t_{\text{track}} (Equation 3), and t_{c_L} (Equation 8) are the service times for a write and a full log read striped over K disks, respectively, and the number of disks in the array. The disk service time for a small user write. When data caching is effective (Equation 2). When caching is ineffective, t_{\text{rmw}} (Equation 11). Expanding in Equation 10 yields a quadratic equation in whose solution is omitted here because it is unnecessarily complex. Denoted by transfer t_{\text{TFC}}(), we approximate this balance equation as a linear equation in whose solution is

$$L = \frac{N}{1 + (KD t_z)/(t_{\text{track}} + 2KR)}$$

(11)

Using this approximation and the disk array parameters from Figure 12, one obtains 6 drones when t_{\text{rmw}} and L = 0.11N when caching is effective. Therefore, to balance the load over all disks in a single region, the example 22 disk array must have 6 sublogs per region.

6. SIMULATION

To validate the analytic models presented in Sections 3 and 4 and to explore response times of various arrays, we simulated the example array described in Figure 12 under five different configurations: nonredundant, mirroring, RAID level 5, floating data, and parity. Simulation was also implemented with a single track of log buffer per region and different degrees of log striping. The simulations were performed using the RAIDSIM package, a disk array subsystem derived from the Sprite operating system disk array driver [Ousterhout88], which was extended to include implementations of parity logging and floating data and parity.

In each simulation, a request stream was generated by 66 user processes, an average of 3 processes per disk. Each process requests a 2KB write from a disk selected at random, waits for acknowledgment from the disk array, then "thinks" for some time before issuing another request. Process think time is exponentially distributed, but the mean is dynamically adjusted until the desired system load is achieved. If the disk array is unable to sustain the offered load, think time is increased. Simulations were run until the 95% confidence interval of the response time became less than the system mean. Because this makes all confidence intervals directly computable, the subsequent plots do not show them.

6.1. The Need for Log Striping

Figure 17 shows peak throughput, response time, and response time variance as the degree of log stripe width L.

Page 15 of 25
striping is varied from one (unstriped) to thirteen. As predicted in Section 5, when striped over a small number of disks, performance is substantially lower than in configurations with more widely striped logs. This behavior results from a "convoy effect" in which process blocking writes queue behind very long sublog read accesses. Figure 18 shows sublog read times for different degrees of log striping. While these long accesses are efficient, they completely tie up a caching array to become idle until the log read completes, reducing peak throughput and utilization. The convoy effect also has a substantial impact on response time; requests that block behind the convoy will have very long response times, leading to an increase in both average response time and response time variance. Fortunately, a modest degree of striping eliminates the convoy effect. Figure 19 shows that striping the log over six disks achieves most of the available throughput while increasing disk space overhead.

With convoys avoided by a log striped over six disks, Figure 19 compares the performance of a parity logging array with one track buffered per region against alternative organizations: nonredundant, mirroring, RAID level 5, and floating data and parity. The graphs of this figure present performance in terms of response time as a function of throughput. Figures 19(a)-(b) assume user data must be preread (data cache miss), and Figure 19(c) presents the corresponding case of no preread (data cache hit) case.

These simulation response time results may be summarized as follows. Nonredundant disk logging performs a single disk access per user write, so they have the lowest and most slowly growing response time. RAID level 5 has a standard deviation of the response time under peak load for various degrees of striping. The difference in performance between striping over 4 to 13 disks is slight, indicating the robustness of the technique.

8. The simulations reported herein consider a user write in a parity logging array complete when the user data in a parity update record has been buffered. The alternatives (nonredundant, mirroring, floating data and parity) consider a user write complete when data and parity are on disk.
time. Mirroring shows a similar behavior, driven into saturation with half as much load. In contrast, each small user write in RAID level 5, when user data must be preread, sequentially two slow read-rotate-write accesses. The unload system response time is thus quite high and queuing effects cause it to grow quite rapidly with load. While the response time for parity logging and mirroring is approximately 14 ms (one revolution) higher than mirroring because of the full rotation delay incurred by a data preread. In addition, the parity update can be issued concurrently, further improving the response time and array utilization. Floating data and parity achieves a lesser benefit from elimination of the preread because its preread overhead is less. Response time does drop, however, due to the ability to issue user write and parity update accesses simultaneously. The response time of parity logging improves by a full rotation and array utilization.

9. In a highly aggressive implementation, it is possible to initiate the parity read-rotate-write access after the user data completes, but we assume that no status is returned until the entire read-rotate-write access completes.

Figure 19(a): Response times

Figure 19(b) presents the average response time standard deviations as a function of the number of small random writes achieved per second. Figures 19(a) and (b) present the results when the user data must be preread, while the results in Figure 18 was cached, making the preread of the user data unnecessary. Reducing the amount of I/O required for data allows the user write and parity update to occur simultaneously, reducing response time for RAID floating data and parity. The reported times are in milliseconds. The response time standard deviation is essentially identical to Figure 19(b).
nonredundant array. This also reduces the actuator time per access by nearly one third, thereby improving throughput and response time to improve proportionately.

The variance in user response time, however, is larger with parity logging than with mirroring floating data and parity, although it is not as large as with RAID level 5. This results from the structure of parity logging: most accesses are fast because inefficient work is delayed. However, some accesses see long response times as delayed work is (efficiently) completed. With this higher variance in mind, we conclude that the response time estimates in Figure 19 show that parity logging and much lower cost, alternative to mirroring for small-write intensive workloads.

<table>
<thead>
<tr>
<th>RAID level</th>
<th>Floating D/P</th>
<th>Mirroring</th>
<th>Parity Logging</th>
<th>Nonredundant</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preread Required</td>
<td>83.7</td>
<td>82.8</td>
<td>89.7</td>
<td>83.5</td>
</tr>
<tr>
<td>No Preread</td>
<td>86.7</td>
<td>87.0</td>
<td>89.7</td>
<td>81.2</td>
</tr>
</tbody>
</table>

Fig. 20: Disk Utilization at Peak Load
6.2. Analytic Model Agreement with Simulation

The analytical model estimates in Figure 14 predict the vertical asymptotes (saturation time) of Figure 19(a) and (c). A direct comparison will display significant discrepancies because of the relatively small number of simulated processes, the deep queue of one overloaded disk can periodically go idle. Figure 20 shows the disk utilizations for the configurations simulated. These peak-load disk utilizations differ according to the number of concurrent disk accesses issued by a user write in each configuration. RAID level 5 and parity, when user data is not cached, and parity logging and nonredundant disk arrays, require present only one disk access request at a time per process. Mirroring and the other array configurations, parity logging, mirroring, and RAID level 5, keep the array busier because each user access is beyond the scope of this paper and this technique is omitted from this section. Floating data and parity keep the array busier because each user access is handled as two concurrent disk accesses. Figure 21 shows that simulation tracks the model predictions of Figure 14 by the disk utilizations of Figure 20, simulation tracks the model predictions to within 5%.

6.3. Performance in More General Workloads

Up to this point, all of the analysis has been specialized for workloads whose accesses are on (2KB) random writes. This section examines a mixed workload, defined in Figure 22, modeled using statistics taken from an airline reservation system [Ramakrishnan92]. With this more general workload, the results of the earlier sections are modified by two important effects: reads are large writes. The issues encountered in extending floating data and parity to handle variable read accesses are beyond the scope of this paper and this technique is omitted from this section. Floating data and parity keep the array busier because each user access is handled as two concurrent disk accesses. Writes that are not small, however, hurt the performance of parity logging as discussed in Section 5.

![Fig. 2Model errors](image)

This figure shows the percent error between the models of sections 3 and 4 Section 6. The model predictions have been scaled by the achieved disk utilizations of Figure 20. Disagreement between the simulation and the models is less than 5 percent. Note that the 95% simulation response time is the mean.

<table>
<thead>
<tr>
<th>Type</th>
<th>% of workload</th>
<th>Size (KB)</th>
<th>Type</th>
<th>% of workload</th>
<th>Size (KB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read</td>
<td>20</td>
<td>1</td>
<td>Read</td>
<td>20</td>
<td>2</td>
</tr>
<tr>
<td>Read</td>
<td>33</td>
<td>4</td>
<td>Read</td>
<td>9</td>
<td>24</td>
</tr>
<tr>
<td>Write</td>
<td>9</td>
<td>1</td>
<td>Write</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>Write</td>
<td>2</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fig. 2Airline reservation workload. The I/O distribution shown above was selected to agree with general airline reservation system [Ramakrishnan92]. This workload is reported as approximately 82% reads, 4.61 KB, and a median read size of 3 KB. The mean write size was larger than the median write size was KB. Locality of reference and overwrite percentages were not reported. All accesses are assumed to have the same boundaries.
Figure 23 presents the results of simulations of four of the array configurations — no mirroring, RAID level 5, and parity logging — on this more realistic OLTP workload. With FIFO disk scheduling, used throughout the rest of this paper, parity logging is always superior to RAID level and is equivalent to mirroring when data caching of writes is effective. With CVSCAN [Geist87], all configurations deliver higher throughput with lower average response times, but mirroring and nonredundant arrays benefit most. Nonetheless, parity logging remains superior to RAID level comparable to mirroring when data caching of writes is effective.

7. MULTIPLE FAILURE TOLERATING ARRAYS

A significant advantage of parity logging is its efficient extension to multiple failure arrays. Multiple failure tolerance provides much longer mean time to data loss and greater for bad blocks discovered during reconstruction [Gibson92]. Using codes more powerful than RAID level 5 and its variants can all be extended to tolerate concurrent failures. Figure 24 gives an example of one of the more easily-understood double failure tolerant disk array organizations: dimensional parity and the more familiar one dimensional parity used in the rest of this call-inary code because a particular bit of the parity depends on exactly one bit from subset of the data disks. If, instead, generalized parity (check information) is computed...

---

10. Our simulations do not explicitly model a disk or file cache. We consider accesses satisfied in such a cache to not contribute to the disk array workload. Cache write hits are special-cased because the disk access is modified by the average prior data values.
bit symbol, dependent on a multiple-bit symbol from each of a subset of the data disks, this non-binary code [Macwilliams77, Gibson92]. Non-binary codes can achieve much lower check information space overhead in a multiple failure tolerant variant of a Reed-Solomon code called “Parity” has been used in disk array products to provide double failure tolerance with only two check information disks [A

This paper is not concerned with the choice of codes that tolerate failure, except to note that the best of these codes all have one property important to small random write [Gibson89]: each small write updates disks containing check information (generalized parity) and the disk containing the data check maintenance work, which scales up with the number of failures tolerated, is exactly the work that parity logging handle more efficiently.

Multiple failure tolerating parity logging disk arrays arise as a natural extension of multiple failure tolerating variants of RAID 5. As with single failure tolerating parity logging, the unique way multiple failure tolerance, the log itself must be (failure tolerant. One way to achieve failure tolerance is to replicate the log \(2\) times. Figure 25 shows one region of a double-fault tolerant parity logging disk array based on a nonbinary “P+Q Parity.”

The log management cycle is quite similar to that of a single fault tolerant parity logging. When a region's log buffers fill up, the corresponding parity update records are written once into the log. As with a single fault tolerant, the log itself is read on a copy shadowing [Bitton88]. Floating data and parity becomes floated data and check, requiring read-rotate-write accesses per blind write.

The overhead associated with maintaining check information can be divided into two components: preread bandwidth overhead and nonpreread bandwidth overhead. The bandwidth needed to pre-read the old copy of the data is independent of the number of failures to be tolerated. Nonpreread bandwidth, the disk work done to update the check information given a data change, grows with the number of failures to be tolerated. Parity logging has the smallest size for this growing component of check maintenance overhead because all check information accesses (generalized parity) are done efficiently.

Figure 26 shows the maximum rate that small random writes can be completed in zero,
double, and triple failure tolerating arrays using mirroring, RAID level 5, floating data and parity logging. This data is derived from the models of sections 3 and 4 and applied to the array of figure 12.

The maximum I/O rate of the parity logging array declines more slowly than the other configurations because parity logging has a substantially lower nonpreread overhead. For example, triple failure tolerating parity logging arrays should sustain about 35% of the I/O rate of nonredundant storage for random small writes, quadruplicated storage (triple failure tolerating mirroring) will sustain only 25%.

8. ACCOMMODATING THE RAID LEVEL 5 LARGE WRITE OPTIMIZATION

In parity-based disk arrays, a large write operation, which is defined as a write that updates data units associated with a particular parity unit, can easily be serviced more efficiently than a small write operation. Since all data units in the stripe are updated, the new parity can be computed in memory from the new data and written directly to the parity unit. This “large write operation” avoids the preread of data and parity associated with small writes, improving write performance.
about a factor of four [Patterson88].

This optimization cannot be applied directly to parity logging disk arrays as we have done so far because there may exist outstanding (not yet reintegrated) logged updates for a parity unit at the time when a large write overwrites that parity unit. If these logged updates and a parity overwrite were done, the parity could be erroneously updated with the stale updates when reintegration occurs. This problem can be corrected by placing the new parity instead of writing it directly in the log by a large write operation is marked as a special “overwrite” record, and the reintegration process, which normally XORs each log record corresponding to a parity unit, now distinguishes between a normal “update” log record and an overwrite record. Update records are XORed into the accumulating parity unit, while overwrite records are simply copied in.

This approach has the disadvantage of forcing the log to be processed sequentially or concurrently if the log were guaranteed to contain only update records, the log records could be written directly to the parity image in any increasing parallelism. The existence of overwrite records forces the reintegration process to determine the sequence in which the log updates occurred and to apply update records accordingly.

This new sequentiality constraint potentially lengthens the reintegration time, which, will show can substantially degrade performance at high loads. In the simplest case, a record must be in the log in the order they were written and merged to produce a update/overwrite image of any of the parity is processed. Given sufficient buffer memory, parity, and log, full parallelism could be achieved during the log and parity reads, but the application of log reintegration would still have to be deferred until these reads complete. At this point, a sequential process for reintegration could be performed. However, as log buffers are written to sublogs in a round-robin fashion, it is reasonable to assume that parallel sublog reads will return parity records in any sequential order. Based on this observation and because overwrite records eliminate all information, the following highly parallel algorithm can be used. Each block in the reintegration is initially zeroed and marked “non-overwrite”. Parity and log for the target region are processed in parallel. A parity block is applied if the corresponding buffer is marked “non-overwrite,” if the buffer is marked “overwrite.” If a logged record is an update and the block is not an overwrite, the target block is overwritten and marked as “overwritten by record X.” All update records that have already been applied should occur after this overwrite are reapplied. Overwrite records preceding X are not applied to a block marked “overwritten by X.” As long as log builds on different sublogs proceed at nearly the same rate, this algorithm will not consume extra buffer space. If buffer exhaustion occurs, the algorithm can simply serialize.

9. RELATED WORK

Bhide and Dias [Bhide92] have independently developed a scheme similar to parity logging. LRAID-X4 organization maintains separate parity and parity-update log disks, and periodically reintegration, they duplicate both the parity and the parity log for a total of four parity integrity regions to reduce the required storage in the log. In order to allow writes from the user to occur in parity by reading from one parity disk and writing The reintegration of a full log disk uses an external sorting algorithm to collect subsequences applying to one area of parity from the log disk. If this area is large, all log reads and parity reads and writes will be efficient. LRAID-X4 reaches its performance maximum of 34.5 writes per disk per second with 20 disks, 2 parity, 2 log) for a 100% write workload with 5% of disk memory [Stodolsky93]. Additional disks do not increase performance. In comparison, the parity logging disk array of Section 6, whose controller requires about 2% of disk memory, is predicted to achieve 36.7 I/Os per disk per second in Section 3 on the same workload, and its performance continues to...
with increasing numbers of disks.

Less closely related research efforts can be characterized by their use of three techniques frequently exploited to improve throughput in disk arrays: write buffering, write-twice, and floating location.

Write buffering delays users’ write requests in a large disk or file cache to achieve deeper write buffering delays users’ write requests in a large disk or file cache to achieve deeper storage. When a write is issued, the data is immediately written (with a self-identifying manner) to a rotationally close empty location in a reserved track, making the write durable. The write is then acknowledged, but the data is retained in the host or controller until it is eventually written to its fixed location. When the data has been written the second time, the corresponding bit in the allocation bitmap is cleared. While significant memory may be required for allocation bitmaps, mapping tables, and write buffers, this storage is not required to be limited controller cost. Write-twice is typically combined with one of the write buffering techniques to improve the efficiency of the second write. This technique has been pursued most fully for fault-tolerant systems [Solworth91, Orji93].

The write-twice approach attempts to reduce the latency of writes without relying on fault-tolerant caches. Similar to floating data and parity tracks in every disk cylinder, several tracks in every disk cylinder are reserved, and an allocation bitmap is maintained. When a write is issued, the data is immediately written (with a self-identifying manner) to a rotationally close empty location in a reserved track, making the write durable. The write is then acknowledged, but the data is retained in the host or controller until it is eventually written to its fixed location. When the data has been written the second time, the corresponding bit in the allocation bitmap is cleared. While significant memory may be required for allocation bitmaps, mapping tables, and write buffers, this storage is not required to be limited controller cost. Write-twice is typically combined with one of the write buffering techniques to improve the efficiency of the second write. This technique has been pursued most fully for fault-tolerant systems [Solworth91, Orji93].

The floating location technique improves the efficiency of writes by eliminating the stationary locations in logical disk blocks and fixed locations in the distributed log. A new location is chosen in a manner that minimizes the disk arm time devoted to the write, and a new logical mapping is established one such scheme, floating data and parity [Menon92], in this paper. An extreme example of this approach is the log structure filesystem (LFS), in which all data is written in a segmented log, and segments are periodically reclaimed. Using fault-tolerant caches to delay data writes, this approach writes into long sequential transfers, greatly enhancing write throughput. However, nearby blocks may not be physically nearby, and performance of LFS in read-intensive workloads may be degraded if the read and write access patterns differ widely. The distorted mirror approach [Solworth91] uses the 100% storage overhead of mirroring to avoid this problem: one copy of the data is stored in fixed location, while the other copy is maintained in floating storage, achieving high write throughput while maintaining data sequentiality [Orji93]. All floating location techniques require substantial host or controller storage for mapping information and buffered data.

10. CONCLUDING REMARKS

This paper presents a novel solution to the small write problem in redundant disk arrays and distributed log. Analytical models of the peak bandwidth of this scheme and alternative proposals were derived and validated by simulation. The proposed technique achieves substantially better performance than RAID level 5 disk arrays on workloads emphasizing small random access writes. When data must be preread before being overwritten (writes miss in the cache), parity logging performance is comparable to floating parity and data without compromising sequential access performance or application control of data placement. When the data to be overwritten has high performance is superior to floating parity and data and mirroring array configurations, and performance is obtained without the 100% disk storage space overhead of mirroring. The technique scales to multiple failure tolerating arrays and can be adapted to accommodate the limited controller optimization.

While the parity logging scheme presented in this paper is effective, several optimizations were explored. More dynamic assignment of controller memory should allow higher performance to be achieved or a substantial reduction in the amount of memory required. Application of data to the parity log should be very profitable. The interaction of parity logging and parity declustering [Holland92] merits exploration. Parity declustering provides high performance during degenerate disk and reconstruction while parity logging provides high performance during fault-free operation. A combination of the two should provide a cost-effective solution.
II. ACKNOWLEDGEMENTS

We would like to thank Ed Lee for the original version of Raidsim. Brian Bershad, Peter Chen, Hugo Patterson, Jody Prival, and Scott Nettles who reviewed earlier copies of this work.

REFERENCES

