# Improving RAID Performance Using a Multibuffer Technique

*Kien A. Hua* *Khanh Vu*

School of Computer Science
University of Central Florida
Orlando, FL 32816-2362
U. S. A.
{kienhua, khanh}@cs.ucf.edu

*Ta-Hsiung Hu*
Syscom Computer Engineering Co.
R & D Division
11F, No. 260, Pa Teh Road, Sec. 2
Taipei, Taiwan
R.O.C.
thhu@oodb.syscom.com.tw

## Abstract

*RAID (redundant array of inexpensive disks) offers high performance for read accesses and large writes to many consecutive blocks. On small writes, however, it entails large penalties. Two approaches have been proposed to address this problem:*

- *The first approach records the update information on a separate log disk, and only brings the affected parity blocks to the consistent state when the system is idle. This strategy increases the chance of disk failure due to the additional log disks. Furthermore, heavy system loads for an extended period of time can overflow the log disks and cause sudden disastrous performance.*
- *The second approach avoids the above problems by grouping the updated blocks into new stripes and writing them as large writes. Unfortunately, this strategy improves write performance on the expense of read operations. After many updates, a set of logically consecutive data blocks can migrate to only a few disks making fetching them more expensive.*

*In this paper, we improve on the second approach by eliminating its negative side effects. Our simulation results indicate that the existing scheme sometime performs worse than the standard RAID5 design. Our method is consistently better than either of these techniques.*

## 1. Introduction

RAID is the acronym for *Redundant Array of Inexpensive Disks* [6]. This technology uses an array of disks where data is spread across the drives so that different sections of an I/O operation (read or write) can be served in parallel by multiple disks. These disk systems have been widely adopted as the mass storage solution for many applications. This trend is attributable to three factors [10]:

1. The performance of microprocessors has outrun the performance of secondary storage justifying the need for using multiple disks to take advantage of their aggregate bandwidth.
2. Arrays of small diameter disks offer a low cost, yet higher performance solution compared to larger drives.
3. Arrays of small disks provide high data reliability by employing low-cost encoding schemes.

There are eight discrete levels of RAID functionally defined so far. Among them, Levels 5 and 7 are more suitable for database applications since they support block interleaving with parity distribution. Group of blocks, one per disk, protected by a parity block is called a *stripe*. Parity information is generated for each stripe by applying XOR to the blocks in the stripe[1]. The parity blocks are distributed across various drives to avoid the need for a dedicated parity drive, thereby removing a potential bottleneck. This strategy allows updates to different data blocks to occur simultaneously. We will refer to an update to a single block as a *small write* in this paper.

RAID5 [6, 2, 9] greatly increases the speed of reads. However, write accesses suffer from having to update the parity data at each write occurrence. The cost of performing a small write is very expensive [4]. It requires reading the old value of the data block (may already be cached in main memory), reading the corresponding parity block, overwriting the old data block with the new data value, and over-

[1] RAID 6 organizes the disks in a matrix format and the parity is generated for rows and for columns of disks in the matrix to allow recovery of lost data in the event of two disks failing simultaneously. The additional overhead in this dual-level redundancy compared to RAID 5 is substantial.

writing the old parity block with the newly computed parity. The excessive overhead makes it a lot more expensive than that of nonredundant arrays. In fact, the I/O throughput increases relatively little throughout RAID 2-5.

To address the small-write problem, a scheme, called *Parity Logging* [10], uses a log disk to store changes to parity blocks. It computes the change to the parity block by computing the XOR of the old data and the new one. These changes are accumulated in a fault tolerant buffer until a more efficient disk transfer can be used to append them to the end of the log file. When the log disk fills up, the out-of-date parity and the log of parity update information are read into memory with large sequential accesses. The logged parity updates images are applied to an in-memory image of the out-of-date parity, and the resulting updated parity is rewritten with large sequential writes. Hence, this strategy replaces many random small parity update accesses with a few large update accesses to log and parity blocks. In order to reduce the memory required for updating the parity from the log, [10] suggests dividing the disk array into several regions, with a separate log area for each. Furthermore, the log and parity areas may be distributed across all disks to balance the load on the disks.

A drawback of Parity Logging is that the hot regions can overflow quickly, a condition that must invoke the relatively long log cleaning operation during which all updates to the data must be suspended. To address these problems, *Data Logging* was proposed [3]. Instead of logging the changes to the parity blocks, this scheme logs the old and the most recent values of the data blocks. Updating a single disk block that is not mirrored in the log requires three I/O operations: reading old data value, writing new data block in home location, and writing old and new block values to the log. Updating a single block that is mirrored in the log, however, requires only two concurrent write operations: write data block in home location, and write new block value in the corresponding log entry. We note that when the log fills, updates to blocks that already have log entries are unaffected. Update to blocks without log entries can perform the regular RAID update operation, incurring the small-write overhead. This graceful handling of log overflow is its advantage over Parity Logging.

Although Data Logging has some advantages over Parity Logging (e.g. graceful degradation), both have similar drawbacks. First, they require additional disks to provide the extra space for the log entries. This increases the chance of disk failure, and hence the downtime to rebuild the data. Another drawback is due to access conflicts. Concurrent updates can happen only both the log disk and the data disk are available for each update. Even if this is possible, the number of small writes can occur simultaneously is limited to $\lfloor \frac{n+1}{2} \rfloor$, where $n$ is the number of disks in the array.

To avoid the aforementioned problems, a different approach, called *Dynamic Parity Stripe Reorganization* (DPSR)[5], can be used. This scheme avoids the need for updating the parity blocks by creating a new stripe for every $n-1$ small writes. When a block is written, its old value remains in place on disk while the new values goes elsewhere in the disk array. The old data block is then marked as dirty. Its old value is still needed in order to protect the containing stripe. Dirty blocks are eventually recycled by a garbage collector to create new free stripes. This approach is twice more efficient than the log-based approach because it can perform $n-1$ small writes simultaneously. Unfortunately, this is achieved on the expense of read operations. A side effect of DPSR is the loss of physical locality of data. Block updates are queued up and written to disks in a new free stripe without respecting their logical order in the data file. After some number of updates, a set of consecutive data blocks which was originally designed to occupy a stripe can migrate to only a few disks making fetching these blocks more expensive. Another drawback of this scheme is as follows. When the data file is initially loaded onto the disk array, all the data blocks contain valid data. If most of the updates are for a few disks. Many of their data blocks become dirty, whereas most of the blocks in the other disks remain valid. The lack of dirty blocks on some of the disks will eventually prevent the garbage collector from producing free stripes even though plenty of dirty blocks are available in the disk array, a phenomenon known as external fragmentation. It is not discussed in [5] how this situation can be handled. It seems that the only solution is to apply the standard RAID5 algorithm, and accept the small-write overhead.

In this paper, we investigate a *Multibuffering* scheme to address the drawbacks in DPSR. We are able to reduce the high cost of small writes while preserving all the advantages offered by the standard RAID designs including large scans. To ensure that contiguous data blocks are always spread across the disks, we maintain a separate fault tolerant buffer[2] for each disk, although these buffers can share the memory space. Data blocks are cached in the buffer corresponding to their home disk. A new stripe is formed by using one block from each of these buffers. The parity is then computed, and the new stripe is written to disk with its parity as a *large write* (LW). This approach prevents a block from migrating to different disks over time. The desirable data striping property of the original RAID design, therefore, can be preserved to ensure efficient data fetching. During the relocation, frequently referenced blocks can be moved to the *hot* region to minimize free-stripe generation overhead and to reduce seek time [1, 7]. Another advantage of Multibuffering is due to the fact that the buffer space also

[2] a non-volatile storage which is used in delayed-writing schemes including DPSR to rebuild data in case of crash

serves as a cache to reduce the I/O activities. We will discuss the buffer management policy in details later.

Although Multibuffering can be used for RAID5, it is ideal for RAID7 which employs a technique called *Dynamic Mapping*. In traditional storage design, a block of data once created is written to a fixed location on disk. Any update to that block must be rewritten to that same physical location. Dynamic Mapping permits updates to be written to new locations within the array. With this facility, we need only add the buffers in order to implement the multibuffering feature. The task is straighforward since RAID7 offers the flexibility to modify the system code. The new technique can be adapted for the operating system embedded in the RAID7 array, which controls the parity generation, check control logic, microprocessor control logic, and the resource drive control logic of the array.

The remainder of the paper is organized as follows. We describe Multibuffering in more detail in Section 2. In Section 3, we present our simulation model. The simulation results are discussed in Section 4. Finally, we give our concluding remarks in Section 5.

## 2. MultiBuffering

DPSR performs worse than standard RAID5 if the application involves many more read than write operations, or when skew is present in the access pattern. In contrast, Multibuffering is robust, and is good for a wide range of applications. It outperforms standard RAID5 under all conditions. We present this scheme in this section.

### 2.1. Handling Small Writes

Similar to DPSR, Multibuffering caches small writes in a fault tolerant buffer in order to write them to disks in parallel at a later time. However, unlike DPSR we use a distinct buffer for each disk although these buffers physically share one memory space. The name Multibuffering alludes to the fact that multiple buffers are used in the proposed technique. This scheme performs a small write as follows:

- An update to a block of a particular disk is inserted into the buffer for that disk, and the old data block is marked as dirty.
- When a new update encounters a buffer-full condition, a block is selected from each buffer according to some replacement policy to form a LW, and these data blocks are written to a free stripe in the array.

This strategy is illustrated in Figure 1. It shows the write buffers at some instance in time, each holds updated blocks for the respective disk. The first disk has two pending small writes, the second disk has three, and so forth. To free up

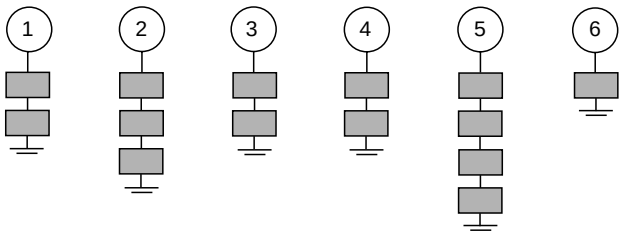


**Figure 1. Each disk has its own write buffer.**

some buffer space, a LW can be performed by flushing one block from each of the six write buffers simultaneously. We note that the original clustering property is preserved since data blocks are confined to their original disk. Furthermore, we do not cast out the data blocks to disks unless it is necessary to make room for a new update. We will examine two replacement policies later.

It is desirable that we can always convert writes to LWs to avoid the parity maintenance overhead. We will give simulation results to show that indeed this can be achieved most of the time even with a small buffer space. Nevertheless, the buffer size is finite. We must have provision to handle bursts of updates which are destined for only a small subset of the write buffers causing the other buffers to become empty after a few large writes. A straightforward solution is to allow graceful degradation by executing traditional RAID5 algorithm. This approach is beneficial only when most buffers are empty. In a better approach, Multibuffering selects one block from each non-empty buffer, and write them to a partial free stripe (PFS) which has a free block aligned with each of the non-empty write buffers. We refer to such a write as a *partial write* (PW) in this paper. To perform a PW, we need to retrieve only the parity block in order to compute the new parity. The new parity block is then written to the disk as part of the PW.

In standard RAID5, a stripe and its member blocks are identified by their physical locations. Which one of these blocks is used to store the parity is determined by the stripe ID. In Multibuffering, the mapping is done through a *stripe directory* which records the physical location of the member blocks in each stripe. To help locate the parity blocks, an entry is maintained for each stripe to identify the parity block. Parity blocks are assigned to the disks in a round-robin fashion. That is, they take turn to hold the parity block for each new LW. We note that a stripe directory was also used in DPSR. This idea allows a data block to migrate to various locations on one (in the case of Multibuffering) or different (in the case of DPSR) disks. This strategy is essential for the free stripe generator to move the data blocks around in order to generate free and PFSs. We discuss this topic in the next subsection.

## 2.2. Free Stripe Generation

Disk activity analyses [7] have shown that a small portion of data blocks receive a majority of the updates under a general file system workload. To take advantage of this pattern, our stripe directory is partitioned into one *hot* and one *cold* regions. The hot region holds blocks under heavy-update activities. The remaining blocks are kept in the cold region. The relocation of data blocks occurs during the free stripe generation process.

To create free and PFSs, the free stripe generator must be able to recycle *dirty blocks* which contain invalid data. This process proceeds as follows.

1. The system selects a *victim stripe* which has the least number of *active blocks* which contain valid data. We note that the victim stripe is likely in the hot region.

2. For each active block in the victim stripe, the system finds a *partner stripe* which has a dirty block on the same disk as the active block. The active blocks and the corresponding dirty blocks are then swapped, and the parities of the partner stripes are recalculated to reflect the exchange of data.

3. If the victim stripe contains only dirty blocks, the system marks the stripe as free; otherwise, the dirty blocks are marked as free, the parity is recalculated, and the stripe is labeled as partially free.

An active block and its corresponding dirty block are physically swapped if at least one of them is in the cold region. Otherwise, the swap of the data blocks in the above procedure does not have to incur any disk activity. As in DPSR, a *stripe directory* can be maintained to record the physical location of the data blocks in each stripe. In other words, the mapping of each stripe and its member blocks is not determined by their physical position but through a directory. With this facility, swap of data blocks can be realized simply by manipulating the stripe directory. For this reason, a partner stripe in the cold region is used only when one in the hot region can not be found. With this approach, the average search space (and time) for the victim and the partner stripes is expected to be small.

We illustrate the free stripe generation process in Figure 2. The parity disk is not shown in this example. In Figure 2(a), the stripe with the least number of active blocks is selected as the victim stripe. Since there is no dirty block on disk 4, we can only produce a PFS in this example. It is shown in Figure 2(a) that the second stripe is selected as the partner stripe since it has a dirty block aligned with an active block in the victim stripe. These two blocks are then swapped to create a new PFS as depicted in Figure 2(b). This stripe is now ready for a PW. We note that DPSR would fail to create a free stripe under this circumstance, and would have to resort to the inferior standard RAID5 update algorithm. As we have discussed in Section 1, this scenario can happen if most of the updates are for a few disks. The lack of dirty blocks on some of the disks will eventually prevent the garbage collector from producing free stripes. Multibuffering circumvents this problem by allowing PWs.

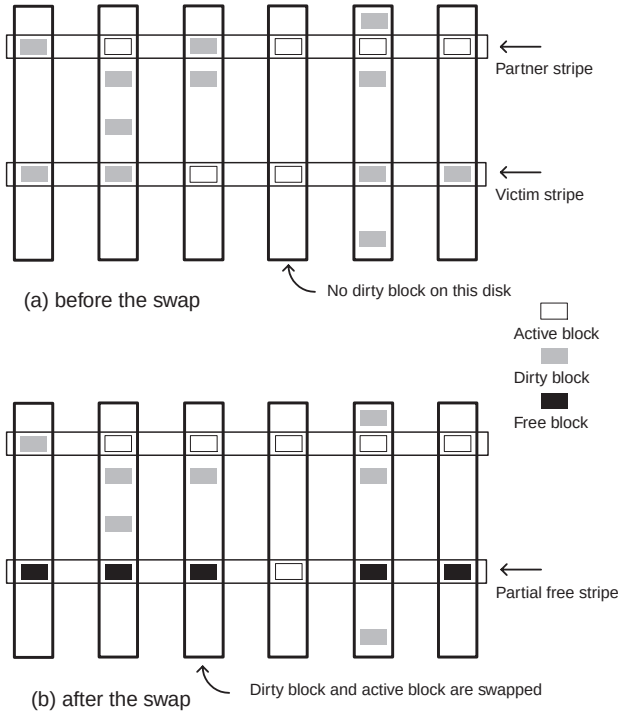


**Figure 2. Generation of a partial free stripe**

One important point we note here is that the garbage collector can always find a free stripe or a PFS to accommodate a LW or a PW, respectively. The reason is as follows. An update to a particular block of a disk is first cached in the buffer of that disk. The corresponding block on disk is then marked as dirty. As a consequence, the number of updated blocks in a buffer is exactly the same as the number of dirty blocks on the respective disk. Likewise, if no update has been performed to a disk, its buffer must be empty and the disk has no dirty block. Due to this nature, when there is a need to perform a LW or a PW, the garbage collector can always find at least one dirty block in the same disk as that of each updated block of the pending write operation. Fragmentation, therefore, can be prevented.

## 2.3. Buffer Management

An update can involve one or many logically contiguous data blocks. These data blocks are referred to as *update blocks* in this paper. Each update block belongs to a specific

disk, and can only be buffered in the corresponding write buffer. This write buffer is said to be appropriate for the update block.

An easy way to manage the write buffers is to employ the FIFO replacement policy. In this approach, the data blocks in each of the write buffers are chained together in a queue as illustrated in Figure 1. When an update results in a hit in a particular buffer, no change is made to the respective queue. A write miss typically incurs a replacement of some blocks in order to make room for the pending update. When this happens, the buffer manager forms a LW (if possible, or a PW otherwise) by dequeueing one block from each of the appropriate buffers which share the same disk with one of the update blocks. After the large (or partial) write is completed, the dequeued blocks are returned to the free pool. The pending update can then use as many of these free blocks as necessary to enqueue the update blocks, each onto the aligned buffer. If the number of update blocks is greater than $n-1$, where $n$ is the number of disks, this large update can be treated as a sequence of smaller updates, each involves $n-1$ blocks with the exception of the last one which can have less than $n-1$ blocks. We note that only up to $n-1$ blocks can be flushed to the disk array at a time since $n$ is the size of a stripe including the parity.

The advantage of FIFO is its simplicity. In this paper, we also consider another policy called LRW (*Least Recently Write*). As in FIFO, a separate LRW buffer is maintained for each disk. The formation of LWs and PWs is as in the FIFO approach. LRW is similar to the standard LRU replacement policy except that read operations have no effect on the replacement decisions. When a free block is needed, LRW selects the block which has not been updated for the longest time to replace. Recent read operations on this block does not change its LRW status. The rationale of this strategy is as follows. First, it is simpler than LRU. Secondly, the LRW buffers can be seen as an extension of the buffer pool in the memory. They help to keep the pages cast out by the memory buffer to stay in another level of semiconductor memory for an extended period of time. If these data are referenced again in the near future, no disk access will be necessary.

We recall that Multibuffering employs multiple logical buffers, one for each disk. These logical buffers, however, share one physical space. This approach can better handle the skew in the access pattern since disks with more updates are automatically allocated more buffer space. This property actually allows Multibuffering to benefit from the skew condition. A more severe skew would improve the hit ratio, and many reads and PWs can be avoided. We will give simulation results to show that a small buffer space can achieve a hit ratio from 8% to 86% depending on the skew condition. We note that even an 8% reduction in the number of disk accesses is quite significant.

## 3. Simulation Model

To assess the performance of MB-RAID7, we compare it with DPSR and the standard RAID5. We assumed an array of nine disks. The database contains 10,000 blocks. They were initially stored on the disks in accordance with RAID5/RAID7 standard. That is, the the striping is block interleaving with the parity blocks distributed across all nine disks. Each simulation run consists of 120,000 random I/O requests. Each read operation accesses up to 100 blocks, and each write operation updates up to 10 blocks. In each simulation run, we waited for 70,000 disk operations before we started to take the statistics.

We choose the *number of requests* as the performance metric. We note that each concurrent access can perform operations on up to nine disks simultaneously. A scheme is superior if it requires less number of concurrent accesses in processing a given workload. To compare the techniques under various real-life conditions, we performed sensitivity analyses with respect to the following workload parameters:

**Write-read ratio** : This is the ratio between the number of write and the number of read operations. For instance a write-read ratio of 0.3 signifies that the disk system receives, on the average, three write requests for every 10 I/O operations.

**Access skew** : This is the measure of skew in the access frequencies of the data blocks. A large access skew results in a few data blocks which are accessed substantially more frequently than the others. On the other hand, all data blocks are accessed with the same frequency if the access skew is zero.

**Access-size skew** : This is the skew condition of the data sizes requested by the I/O operations. A larger access-size skew indicates that most of the I/O operations involve a large number of data blocks.

We model the various skew conditions using a Zipf-like distribution [8, 11]. For instance, to simulate an access skew of $z$, we used the following distribution function to determine the access probability $P_i$ of block $i$:

$$P_i = \frac{1}{i^Z \sum_{j=1}^{n} \frac{1}{j^Z}},$$

where $n$ is the number of blocks in the database, i.e., 10,000. Note that when $z = 0$, the distribution becomes a uniform distribution. In this paper, $z$ is refered to as the *skew factor*.

The request generator employs three random number generators. Each I/O request is generated in three steps:

1. The first random number generator, guided by the Zipf-like distribution, creates the ID of the first data block of the I/O request.

2. The second random number generator, guided by the write-read ratio, determine the type of the operation (i.e., read or write).

3. The third random number generator, guided by the Zipf-like distribution, creates the access size (between 1 and 10 for a write operation, and between 1 and 100 for a read operation).

We discuss the simulation results in the next section.

## 4. Simulation Results

We discuss the simulation results in this section. We first compare the replacement policies: FIFO versus LRW. The better replacement policy is then used in the subsequent studies to compare Multibuffering with DPSR and RAID5.

### 4.1. Replacement Policy

The size of the buffer is the most influential factor in the success of our scheme. The larger the buffer size, the greater the chance for LWs to be formed and the higher the hit rate. However, a large amount of NV-RAM can be expensive. To show that Multibuffering is feasible, we fixed the buffer size at 100 data blocks. That is, it is approximately 1% of the size of the database in our study. We note that RAID5 and DPSR both use some amount of write buffer (more than 2MB in DPSR [5]), but no attempt was made to manage it as a cache to reduce disk I/Os.

The hit ratios of FIFO and LRW under various access-skew conditions are plotted in Figure 3. It shows that LRW is consistently better than FIFO unless the access pattern is uniform. The saving is most significant at the middle range displaying a 50% improvement. Although the savings are only about 10% for near-uniform access patterns, a reduction of 10% in disk accesses is still very significant. Due to these results, we will use LRW for the remaining studies.

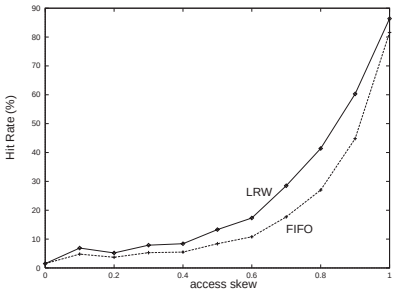


**Figure 3. The hit rates (%) vs. access skews at 0.3 W/R ratio.**

### 4.2. Effect of Write/Read Ratio

In this study, we analyze the effect of write/read ratio on Multibuffering, DPSR and RAID5. We fixed the access-size skew at 0.2. The rationale is as follows. 0.2 is a moderate skew which seems to represent many real-life applications. Furthermore, our simulation studies indicated that this factor had little impact on the performance of the various techniques. The performance of the three RAID techniques under the access-skew conditions of 0.3 and 0.7 are plotted in Figures 4 and 5, respectively. They represent a mild and a severe skew conditions, respectively. The effect of various skew conditions will be discussed in Section 4.3.

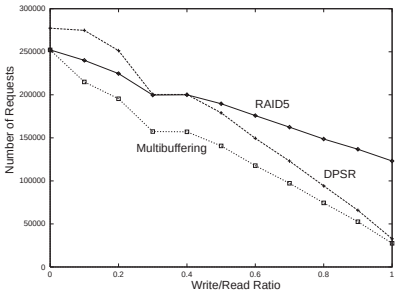


**Figure 4. The number of accesses vs. W/R ratios, at 0.3 access skew.**

We observe that DPSR performs poorly when the W/R ratio is low. For instance, Figure 5 indicates that DPSR is actually worse than the standard RAID5 by 14% when the W/R ratio is 0.1. They perform at the same level for W/R ratios in the neighborhood of 0.4. DPSR does not offer a tangible benefit until the W/R ratio is greater than 0.5. These results confirm that DPSR is only suitable for applications with many write operations, say six or more writes for every 10 I/O's. In contrast, Multibuffering is shown to be very robust. It consistently offers the best performance. The savings are as high as 40% compared to DPSR, and 84% compared to RAID5. We note that the number of requests are reducing for higher W/R ratios due to the fact that the fixed maximum size of writes is only 10% that of reads

### 4.3. Effect of Access Skew

In this study, we investigated the effect of access skew on the performance of the three RAID designs. The experiments were performed under three different W/R ratios: 0.1, 0.5 and 0.9. The corresponding plots are shown in Figures 6, 7 and 8, respectively. Again, we observe that DPSR does not perform as well as the standard RAID5 unless the W/R ratio is significantly large.

When W/R ratio is 0.1 (Figure 6), DPSR performs poorly because most of the operations are reads. This condition

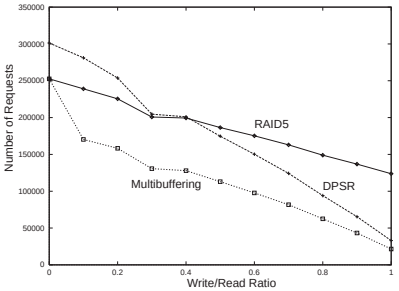


**Figure 5. The number of accesses vs. W/R ratios, at 0.7 access skew.**

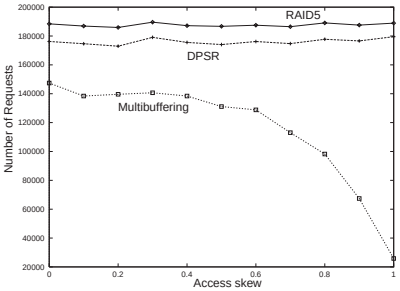


**Figure 7. The number of accesses vs. access skew, at 0.5 W/R ratio.**

does not affect Multibuffering because its write operations preserve the physical locality of the data.

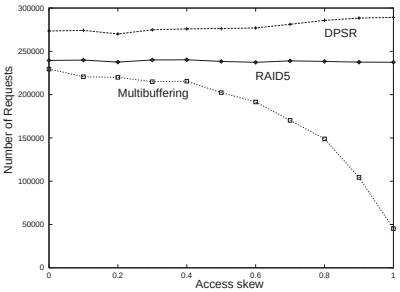


**Figure 6. The number of accesses vs. access skew, at 0.1 W/R ratio.**

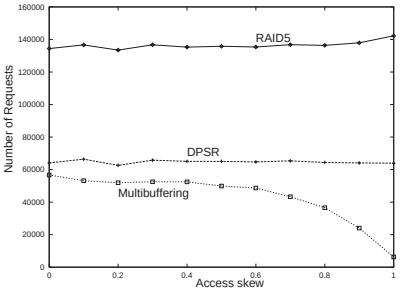


**Figure 8. The number of accesses vs. access skew, at 0.9 W/R ratio.**

As the W/R ratio becomes bigger, i.e., 0.5 (Figure 7), there are many write operations, and the performance of DPSR improves to match that of RAID5. Multibuffering, however, is still the best performer. In particular, when the skew is very severe (i.e., access skew is 1.0), Multibuffering is as much as 86% better than the other techniques. This is due to the fact that a more severe access skew results in a better hit rate for the LRW buffers rendering many disk I/Os unnecessary under Multibuffering. On the other extreme, When the access skew is zero, although the hit rate is low, the combined effect of some cache hits and more efficient read operations still gives Multibuffering a significant 17% improvement over DPSR.

When the W/R ratio is very large, i.e., 0.9 (Figure 8), the percentage of read operations decreases substantially. As a result, DPSR displays a significant performance gain over RAID5. Nevertheless, Multibuffering still leads DPSR primarily due to the fact that the LRW buffers are also managed as a cache to reduce disk activities. This is evident as the performance of Multibuffering improves with the increases in the access skew. When the skew is 1, the saving over DPSR is a huge 89%.

### 4.4. Percentage of LWs

In summary, although DPSR is a good technique for applications with many write operations, the simulation results indicate that Multibuffering is more robust. It consistently outperforms the other two schemes regardless of the access skew and the W/R ratio. The above studies also demonstrate that only a small buffer space is necessary to exploit the benefit of Multibuffering.

The high performance of Multibuffering can be attributed to the very high percentage of LWs. To substantiate this claim, we run a simulation with various combinations of W/R ratios and access skew. The results of this study are summarized in Table 1. Since there are eight data blocks for each stripe in our study, the write size, S, ranges from one up to eight blocks (excluding the parity block). The entries in the second row of the table are the counts, #W, observed for each category of the write size S during the simulation run. The last row gives the percentage, %W, of each category. We note that 98.3% of the write accesses to the disk array are LWs. Another 1.2% are near-large writes

(i.e., 6 and 7) while writes of the smaller sizes are mostly the results of the final buffer flush. These simulation results confirm that Multibuffering is very effective in avoiding the small-write problem.

| S | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** |
|---|---|---|---|---|---|---|---|---|
| #W | 6 | 10 | 13 | 13 | 14 | 16 | 106 | 10130 |
| %W | 0.0 | 0.0 | 0.1 | 0.1 | 0.1 | 0.2 | 1.0 | 98.3 |

**Table 1. Average percentage of write sizes.**

## 5. Concluding Remarks

In this paper, we surveyed recent techniques for addressing the small-write problem in RAID, and discuss their drawbacks. To address these problems, we investigate a new scheme called *Multibuffering*. Compared to Data Logging and Parity Logging, it has the following advantages:

- Performing large writes in Multibuffering is more efficient than doing multiple small writes simultaneously in Data Logging and Parity Logging. The former can write $n-1$ blocks at a time whereas the latter can do no more than $\lfloor \frac{n+1}{2} \rfloor$ blocks simultaneously, where $n$ is the number of disks in the array.
- The logging approach requires extra disks, which increases the chance of disk failure, and hence the downtime to rebuild the data.
- The log cleaning operation in Parity Logging is relatively long during which all updates to the data must be suspended. Data Logging allows graceful degradation when the log fills. The solution, however, is to resort to the standard RAID update algorithm for non-logged data. Multibuffering does not have these problems.

Compared to Dynamic Parity Stripe Reorganization (DPSR), Multibuffering has the following advantages:

- Unlike DPSR, Multibuffering preserves the physical locality of data, thus enhances the performance of large reads.
- In the presence of high access skew, the lack of dirty blocks on some disks will eventually prevent DPSR to produce free stripes. Multibuffering addresses this problem by performing partial writes.
- In addition to avoiding the small-write overhead, the LRW (Least Recently Write) policy employed in Multibuffering also takes advantage of the write buffers to reduce the number of disk accesses. Data reorganization is also considered to minimize garbage collection overhead and to reduce seek time.

To assess the benefit of the proposed technique, we performed simulation studies to compare its performance with those of DPSR and standard RAID5. Our simulation results convincingly indicate that Multibuffering is very robust. It consistently outperforms the other two techniques. Depending on the workload, the savings can be up to 89% compared to DPSR, and 95% compared to RAID5. Our study also confirmed that DPSR is not suitable for applications with many read operations. It offers no advantage compared to RAID5 if the write-read ratio is in the neighborhood of 0.4 (i.e., four writes for every 10 I/O operations).

In conclusion, we introduced in this paper an efficient technique to address the small-write problem in disk arrays. While prior solutions achieve some of the benefits, they also suffer many side effects. Multibuffering is unique in avoiding the small-write overhead without any of the undesirable behavior.

## References


[1] S. AKyuret and K. Salem. Adaptive block rearrangement. In *Proc. of the Nineth ICDE*, pages 182–189, April 1993.

[2] P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. Raid: High-performance, reliable secondary storage. In *ACM Computing Surveys*, pages 26(2):145–185, June 1994.

[3] E. Gabber and H. F. Korth. Data logging: A method for efficient data updates in constantly active raids. In *Proc. of the* $14^{th}$ *ICDE*, pages 144–153, February 1998.

[4] M. Holland and G. A. Gibson. Parity declustering for continuous operation in redundant disk arrays. In *Proc. of the Fifth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 22–34, Oct. 1992.

[5] K. MOGI and M. KITSUREGAWA. Dynamic parity reorganizations for raid5 disk arrays. In *Proc. of IEEE*, pages 17–26, February 1994.

[6] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proc. of ACM SIGMOD*, pages 109–116, June 1988.

[7] C. Ruemmler and J. Wilkes. Unix disk access paterns. In *Proc. of the 1993 winter USENIX Conference*, pages 405–420, San Diego, CA, Jan. 25-29 1993.

[8] G. M. Sacco. Fragmentation: A technique for efficient query processing. In *ACM Trans. Database Systems*, pages 113–133, February 1986.

[9] A. Silberschartz, H. F. Korth, , and S. Sudarshan. *Database System Concepts*. McGraw-Hill, New York, third edition edition, 1997.

[10] D. Stodolsky and G. A. Gibson. Parity logging: Overcoming the small write problem in redundant disk arrays. In *Proc. of ISCA*, pages 64–75, May 1993.

[11] G. K. Zipf. *Human Behavior and the Principle of Least Effort: An Introduction to Human Ecology*. Addison-Wesley, Reading, MA, 1949.