ARC: An Approach to Flexible and Robust RAID Systems

Ba-Quy Vuong and Yiying Zhang
Computer Sciences Department, University of Wisconsin-Madison

Abstract

*RAID systems increase data storage reliability by employing one or more data integrity techniques such as parity or checksum. However, the current implementations of software RAID systems suffer both inflexibility and unrobustness. First, different software RAID levels require different RAID systems and the system written for one level cannot be reused for others. In addition, writing a new RAID system is known to be difficult and it requires a lot of debugging effort. In this paper, we propose Automatic RAID Construction (**ARC**), an approach to automatically construct RAID systems with different parity and checksum schemes. In this approach, we first provide various data structures and parameters for specifying parity and checksum schemes. We then automatically translate each scheme into a RAID system with the desired features. With ARC, only one system is required to handle many different RAID levels. Thus, the effort of writing new RAID systems is eliminated and both flexibility and robustness are achieved. We also implemented the ARC system on Linux with simulated virtual disks. The system consists of slightly more than one thousand lines of code which is not much different from current RAID systems such as RAID 1 and RAID 5. The performance evaluation shows that ARC can be configured to act as different RAID systems with reasonably good performance.*

1 Introduction

1.1 Motivation

Many applications today such as databases and Web servers are requiring more reliable storage since media or disk failures may cause unmanageable cost. Until recently, one of the most common ways to improve storage reliability is to store data in one of the RAID systems. RAID systems usually use an array of disks together with one or more data integrity techniques such as parity or checksum to detect and recover from disk failures. RAID systems are currently built either at the hardware level (hardware RAID) or at the software level (software RAID).

In hardware RAID, the RAID controller is built as a dedicated hardware to control the disk array. The hardware RAID has the advantage of good performance. However, is it not flexible as the RAID level is fixed in the hardware. In addition, hardware RAID is expensive and difficult to maintain as the firmware is often very complex and changing firmware code is known to be painful.

Software RAID is another alternative to build RAID systems. In this approach, a software layer (RAID driver) is developed to sit above the disk drivers to control the disk array. The RAID driver is responsible for distributing requests and controlling the disk array. Software RAID has several advantages over hardware RAID. First, it is more flexible. A new RAID level on the same disk array can be constructed by just replacing the RAID driver with a new one. Maintaining software RAID is also more straightforward. One just needs to change the code of the RAID driver at the software level. Finally, software RAID is cheap and it is distributed with most versions of Linux. In the scope of this paper, we only focus on software RAID.

Although software RAID has lots of advantages over hardware RAID, it is still considered as not flexible and robust enough for many applications' needs. First, each level of RAID needs a RAID driver that is specifically designed for it. The driver developed for one level cannot be reused for other levels. If one application requires a new RAID level then a new RAID driver needs to be built from scratch. Thus, the current software RAID is not flexible. Second, although developing a software RAID driver is easier than writing a hardware firmware, it is still a challenging task, especially for inexperienced programmers. Every newly written RAID driver requires numerous amount of time, effort and experience for testing to make sure that it is bug free. Because of this factor, many new RAID drivers are not robust and may contain a lot of bugs.

1.2 Objectives and Contributions

In this work, we aim to automate the process of constructing RAID systems and thus improve both flexibility and robustness. We do not focus on building a specific RAID level system. Instead, we propose a generic system that can be configured to become any RAID system by setting appropriate data structures and parameters. In particular, we design two matrices for configuring parity schemes and a list of parameters for configuring checksum schemes. The two matrices for parity include:

- A layout matrix for setting the roles of different blocks on a RAID stripe.
- A parity matrix for indicating which data blocks in one stripe contribute to a particular parity block.

The parameters for checksum include:

- The checksum function.
- The location of each checksum.
- The size of each checksum.
- Additional information for each checksum such as the block physical location.

We also implemented the proposed design as an actual RAID driver in Linux with simulated virtual disks. The code for this driver consists of slightly more than one thousand lines, which is not much different from the current implementations of RAID 1 and RAID 5. The performance evaluation shows that our driver can be configured as drivers for different RAID levels with all the basic features. Although we do not focus on performance in this design, our driver is shown to result in reasonably good performance for both read and write operations.

1.3 Paper Outline

This paper is organized as follows. We investigate and summarize some related work in Section 2. In Section 3, we describe the architecture of ARC in detail. We then discuss the designs of automatic parity and automatic checksum in Section 5 and Section 4 respectively. Section 6 discusses some implementation issues of ARC in Linux. We present and analyze some performance results in Section 7. Finally, Section 8 concludes the paper.

2 Related Work

RAID has been an active research direction for years since the five basic RAID levels were introduced in [10]. However, most of the work focus on proposing new RAID levels [3, 4], estimating RAID reliability [5] or modeling disk failures [9].

In [8], Krioukov discussed a formal approach for analyzing the design of different data protection strategies. This paper mentions a list of strategies such as different checksum schemes and version mirroring. However, it provide a way of building an automatic system that exploit these strategies.

RAIDframe [6] is one of the efforts to provide a firmware environment for prototyping and evaluating different RAID architectures. It localizes modification and provides libraries of existing architecture to extend. The disk array operations are modeled as directed acyclic graphs (DAGs) and a simple state machine. Although this approach can prototype different RAID codes, it is not clear how an automatic RAID construction system is built.

Until recently, there have been several system-on-chip (SOC) products [2, 1]. Typically, these systems contain an embedded processor, DMA/XOR engines and host and disk interface logic. While these system are considered to support variety of RAID codes by using the programmable capability of processors, they are built at the hardware level which has no relation with software RAID.

One of the work that is closest to the problem we are attacking is REO [7]. This paper proposes a generic RAID engine and optimizer that is able to optimize I/O operations of different RAID codes. The RAID code input to REO is in the form of a generator matrix, which specifies how parity blocks are laid out and calculated. However, this work focuses on using the generator matrix to optimize the RAID operations instead of building an automatic RAID construction system.

3 Architecture

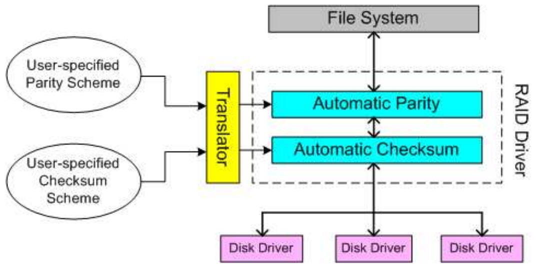


Figure 1: ARC architecture.

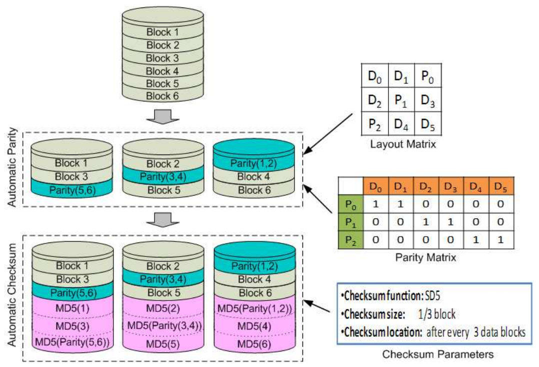
Like other software RAID systems, ARC is designed as a software layer sitting between the disk drivers and the file system. Figure 1 shows the overall design of ARC. The system consists of three main components. The roles of these components are as follows.

- **The Translator:** This module allows users to specify different schemes of parity and checksum. It then converts these schemes into appropriate data structures and parameters that are ready to be used by the Automatic Parity and the Automatic Checksum modules.

- **The Automatic Checksum:** This module takes in a list of parameters from the Translator and uses it to control the operation of checksums. This module works on an array of disks. For each disk, it builds appropriate checksum blocks for a set of data blocks using the parameters provided. It then hides the checksum blocks and only exports the data blocks to the Automatic Parity module. Thus, the checksum blocks are transparent to the Automatic Parity module. What this module sees is an array of disks, each disk consists of consecutive data blocks.

- **The Automatic Parity:** This module takes in a layout matrix and a parity matrix from the Translator. These two matrices are used to control the operation of the parity. For each read or write operation from the file system, one or more read and write operations will be issued from this module to the underlying disk array exported by the Automatic Checksum module. The Automatic Parity module also hides the parity blocks and only exports the data blocks from the disk array as consecutive blocks of a single disk. The detailed design and operation of this module are discussed in Section 5.

To better understand the operation of ARC, let us consider the following example:



	D_0	D_1	D_2	D_3	D_4	D_5
P_0	1	1	0	0	0	0
P_1	0	0	1	1	0	0
P_2	0	0	0	0	1	1

- •Checksum function: SD5
- •Checksum size: 1/3 block
- •Checksum location: after every 3 data blocks

Figure 2: An example of disk block layout at different modules.

Example 1: *Assume that one user specifies RAID 5 configuration with three disks. He also wants to have a MD5 checksum for each data block. The reserved size for each checksum is one third of a block and the checksum region is placed after every three data blocks. First, the Translator translates the requirements from the user into two matrices for parity and a list of configuration parameters from checksum. These matrices and parameters are then fed into the Automatic Parity and the Automatic Checksum modules respectively. Figure 2 shows the overall process and shows how different blocks of data, parity, checksum are placed by different modules. At the file system level, the entire disk array is seen as a single disk with six data blocks. At the Automatic Parity level, the six data blocks are distributed over three disks. Three parity blocks are also constructed and distributed according to the values of the matrices. At the Automatic Checksum level, one checksum block are added for every two* *data blocks and one parity block. As the size for each checksum is one third of a block, each checksum region occupies exactly one block. According to the Checksum Location parameter, the checksum block is place after every three blocks.*

One may notice that the order of the Automatic Parity and Automatic Checksum is important in our design as it may change the locations of blocks on the physical disks. For example, if the Automatic Checksum is built on top of Automatic Parity instead of the reversed order as in the current design, then the same configuration will result in different locations of data, parity and checksum blocks. We realize that either design can produce fairly good system. However, we stick to the current design because we implemented the Automatic Parity module first and it was easier for us to implement the checksum module below the parity module.

4 Automatic Checksum

Checksuming is a common technique to detect data corruption, which is often used in current RAID systems. There are various ways of checksuming in terms of checksum functions, checksum unit, checksum size, checksum location, and additional information stored with checksum. Common checksum functions include hash-based functions such as SHA1 and MD5. Each checksum can be provided for one or more data blocks. The size of the checksum often depends on the checksum function. However, a reserved size can be set for storing each checksum. In general, the checksum can be placed anywhere on the disk as long as there exists a mechanism to determine the checksum for a requested data block. Additional information, such as the block physical location, can be also added to the checksum.

All current RAID implementations have a fixed design of checksums (checksum functions, checksum positions, and so on). Once a RAID system is built, changing the checksum scheme is often very hard. For software RAID, this means rewriting the RAID driver. For hardware RAID, it means changing the firmware code.

4.1 Design of Automatic Checksum

In ARC, we put the Automatic Checksum module below the Automatic Parity module for the reason mention in Section 3. With this design, we provide checksum for both data and parity blocks. In fact, the checksum module cannot distinguish between data blocks and parity blocks. It performs checksuming regardless of the content of the block. Thus, unless specifically noticed, in this subsection we use the term “data block” for both data blocks and parity blocks.

A major concern of our design is to provide flexibility in defining checksum schemes while ensuring that the defined scheme works properly. The degree of flexibility includes: (1) flexible checksum unit, (2) flexible checksum location, (3) built-in and user-provided checksum functions, and (4) time of verification. In our design, we support the first three flexibilities while leaving the last one for future work. Specifically, we allow users to change the following parameters:

- The checksum function.
- The size reserved for the checksum of one data block. This size must be at least the actual size needed to accommodate the checksum.
- The number of data blocks whose checksums are put together in one checksum region. This parameter is used to control the location of the checksum regions on disk.

Checksum Mapping: In the current ARC design, a checksum region is placed after every n consecutive data blocks. This region contains a checksum for each of the n data blocks. The n consecutive data blocks are called a checksum unit. The size n of the checksum unit is adjustable. The smallest value is one and the largest value is the number of blocks in the disk (except a few blocks are reserved for the checksum region). One should note that the checksum region may span more than one disk block.

In each checksum region, checksums for the n data blocks are placed sequentially, in the order of data blocks on disk. Each checksum occupies the size reserved for it. Figure 3 shows an example of checksum scheme. In this example, each checksum unit consists of three data blocks ($n = 3$) and each checksum is reserved one third of a block. Thus, each checksum region occupies exactly one block. The first chunk in the first checksum region stores the checksum for block 0, the second chunk stores the checksum for block 1, and the third chunk stores the checksum for block 2.

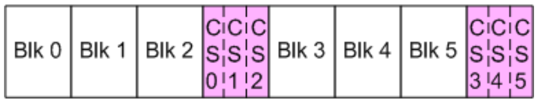


Figure 3: An example of checksum scheme.

In order to export only data blocks to the Automatic Parity module while hiding the checksum blocks, data blocks need to be re-mapped. For each request of the i^{th} block from the Automatic Parity module, Equation 1 is used to calculate the actual data block to be accessed.

$$i' = \lfloor i/n \rfloor * (n+k) + i\%n \quad (1)$$

Where i is the requested block, i' is the actual block on disk, n is the number of data blocks in each checksum unit, k is the number of blocks that a checksum region occupies.

For each block read or block write request, the corresponding checksum needs to be accessed for data integrity verification or checksum update. The starting block of the checksum region for the requested data block i is given in Equation 2.

$$c = (\lfloor i/n \rfloor) * (n+k) + n \quad (2)$$

Where c is the starting block of the checksum region. i, c and k are the same as in Equation 1.

For the example in Figure 3, we map the data block 4 into $\lfloor 4/3 \rfloor * 4 + 4\%3 = 5$. The checksum region starting block is $(\lfloor 4/3 \rfloor) * 4 + 3 = 7$.

5 Automatic Parity

In this section, we describe the design and data structures of the Automatic Parity module in detail. Our parity module uses the XOR-based code to calculate parities and to reconstruct data blocks. To make the design simple, we set the unit of parity as a disk block. Each parity block is a XOR result of one or more data blocks. Although XOR is binary operator, we make the convention that if it is applied to only one input then the result is the input itself (i.e. $XOR(A) = A$). A *stripe* is a set of data blocks and all related parity blocks. Within a stripe, a parity block is a XOR of a subset of data blocks in that stripe. A *strip* is a set of blocks that are arranged contiguously on one disk. The number of disks is also the number of strips in one stripe. For simplicity, we assume that the block size, stripe size and strip size are uniform across all disks.

As noted earlier, there are two data structures used by the Automatic Parity module: the Layout Matrix and the Parity Matrix. In the following subsections, we discuss the format of these two matrices and how they are used to calculate parities and to reconstruct data blocks.

5.1 The Layout Matrix

The layout matrix has the size of $n\ rows \times m\ columns$ where m is the number of disks (number of strips per stripe) and n is the number of blocks per strip. The size of the layout matrix is also the number of blocks in one stripe. The cell at row i, column j in the matrix corresponds to the i^{th} block on the j^{th} strip of each stripe. For each cell in the matrix, value D_x indicates the corresponding block is a data block and it is the x^{th} data block within the stripe. Value P_y indicates the corresponding block is a parity block and it is the y^{th} parity block in the stripe. Both x and y are counted from the beginning of the stripe, starting from zero. Figure 4 shows some examples of the the layout matrix for different RAID levels. In Figure 4(a), the cell containing D_4 indicates that the the first block of the first strip is a data block, and that is the fourth data block in the stripe [1].

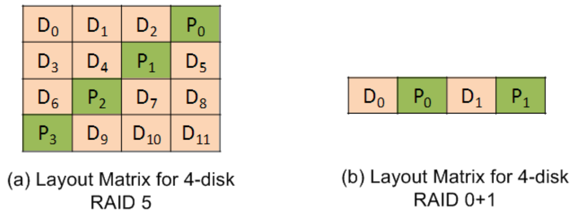

D0 D1 D2 P0
D3 D4 P1 D5
D6 P2 D7 D8
P3 D9 D10 D11
(a) Layout Matrix for 4-disk RAID 5
D0 P0 D1 P1
(b) Layout Matrix for 4-disk RAID 0+1

Figure 4: Examples of the Layout Matrix.

[1]Note that all the indices start from zero.

Now let us discuss how read and write requests from the file system are translated into appropriate read and write requests on actual disks.

Read Operations: Since the whole disk array is seen as a virtual single disk from the file system perspective, a read request issued by the file system is in the form of reading a particular block on the virtual disk. The Automatic Parity module first identifies the actual disk and the actual block for the request by applying Algorithm 1. This algorithm works as follows. First, it determines the position of the requested block in the stripe by taking the *mod* of the requested block number with the number of data blocks in each stripe. It then iterates through the Layout Matrix to find the row and column of the cell that corresponds to that block. The column number of that cell is the actual disk number. The row number of that cell, is then used to calculate the actual sector number. For example, assume that there is a request at block number 15. This corresponds to the cell D_3. This means the actual disk number is 0 and the actual sector number is 4.

```
Algorithm 1. Determine the Actual Disk and block
Input:     I_Block – The requested block
           M – The Layout Matrix
Output:    Disk – The actual disk
           O_Block – The actual block on Disk
Process:
  1.       offset ← I_Block mod (#data blocks per stripe)
  2.       cell ← cell $(0, 0)$ in M
  3.       while (offset>0)
  4.           $cell \leftarrow$ the next data cell in M
  5.           offset ← offset-1
  6.       end while
  7.       $Disk \leftarrow$ the column of $cell$
  8.       $s \leftarrow$ I_Block div (#data blocks per stripe)
  9.       $O\_Block \leftarrow s \times strip\_size +$ (the row of $cell$)
```

Figure 5: Algorithm to determine the actual disk and actual block of a request.

After the actual disk and block have been identified, the request is redirected to the those disk and block. If the read fails (e.g. an error code is returned from the disk driver or the checksum inconsistent), the Parity Matrix is used to determine a list of candidate parities for reconstruction. The process of determining candidate parities for reconstruction is discussed in Section 5.2. For each candidate, multiple read requests may be issued by the parity module to read blocks for reconstruction. Notice that these reconstruction reads may also fail. If the module finds at least one candidate that results in a successful reconstruction, it stops and returns the reconstructed block to the file system. Otherwise, an error code is returned to indicated that the system is unable to read the requested block.

Write Operations: To respond to a write request from the file system, the Automatic Parity module executes in two steps:

1. Writes the requested block to the actual disk.
2. Calculates the new parity values and updates all the parity blocks that are affected by the new data block.

The first step is straightforward. Using Algorithm 1, the Automatic Parity module can identify the actual disk and the actual block to redirect the write request to. To execute the second step, the module first uses the Parity Matrix to determine a list of parity blocks to be updated. It then calculates the new parity for each block. Notice that in order to calculate a new parity value, some data blocks and the old parity block may be read. If this process fails due to failed reads, an appropriate error code is returned to indicate the writing of the parity block(s) fails. In our design, we use a simple approach to calculate the new parity that requires only two reads: one read on the old data block and one read on the old parity block. The new parity is then calculated using Equation 3.

$$\textit{new parity}=\textit{old parity} \text{ XOR } \textit{old data} \text{ XOR } \textit{new data} \qquad (3)$$

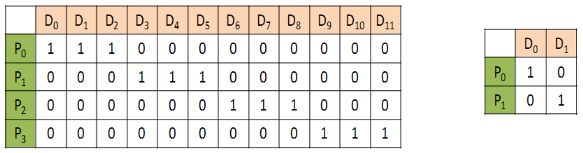
5.2 The Parity Matrix

The Parity Matrix specifies what data blocks in one stripe are XORed to make a certain parity block. This matrix serves two purposes:

- **Read Reconstruction:** The matrix helps to determine which parity blocks can be used for reconstructing a data block, and what are other data blocks that need to be read for the reconstruction process.
- **Writing Parity Blocks:** The matrix helps to determine what are the parity blocks that need to be updated in response to a write request.

We design the Parity Matrix as a matrix with p rows and q columns. p is the number of parity blocks in one stripe and q is the number of data blocks in one stripe.

Each row corresponds to a parity block and each column corresponds to a data block. For each cell at row i and column j in the matrix, value 1 indicates the j^{th} data block in the stripe is used to calculate the i^{th} parity block. Value 0 means the i^{th} data block is not used. Figure 6 shows some examples of the Parity Matrix for the two RAID levels mentioned in Figure 4. In the following, we discuss how this matrix is used to serve the two purposes mentioned above.



	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7	D_8	D_9	D_{10}	D_{11}
P_0	1	1	1	0	0	0	0	0	0	0	0	0
P_1	0	0	0	1	1	1	0	0	0	0	0	0
P_2	0	0	0	0	0	0	1	1	1	0	0	0
P_3	0	0	0	0	0	0	0	0	0	1	1	1

(a) Parity Matrix for 4-disk RAID 5

	D_0	D_1
P_0	1	0
P_1	0	1

(b) Parity Matrix for 4-disk RAID 0+1

Figure 6: Examples of the Parity Matrix.

Read Reconstruction: If a read on the j^{th} data block of a stripe fails, we first scan through the column j in the Parity Matrix. If the cell at row i is one then the i^{th} parity block is a candidate for the reconstruction. For each parity candidate at position i, we scan the row i in the Parity Matrix. If the cell at column k ($k \neq i$) is one then the k^{th} data block in the stripe needs to be read to reconstruct the i^{th} data block. Note that all the indices start from zero. For example, for the 4-disk RAID 5 in Figure 4 and Figure 6, assume that a read on D_5 fails. The column D_5 in the parity matrix indicates that there is only one parity candidate which is P_1. The row P_1 in the same matrix tells us that D_3 and D_4 need to be read for the construction. The formal algorithm is given in Algorithm 2.

Writing Parity Blocks: The process of getting parity blocks for update is simple. We just scan through the corresponding column of the newly written block. If a value one is encountered then the parity block corresponds to that row needs to be updated. For example, in 4-disk RAID 5, if the written block is D_9 then using the D_9 column in the Parity Matrix, we see that the parity P_3 needs to be updated.

Algorithm 2. Determine candidates for reconstruction

```
Input:    $i$ – position of the failed block in the Layout Matrix
          $M$ – The Parity Matrix
Output:   $P = \{k_1, ..., k_u\}$ – positions of candidate parity blocks in
          the Layout Matrix
          $D_k = \{j_{k1}, ..., j_{kv}\}$ – positions of data blocks for each
          candidate parity block $k$ in the Layout Matrix
Process:
  1.      $P \leftarrow \emptyset$
  2.      for each cell $c1$ in $M$ at column $i$
  3.          if $c1$ contains 1
  4.              $k \leftarrow$ the row number of $c1$
  5.              add $k$ to $P$
  6.              $D_k \leftarrow \emptyset$
  7.              for each cell $c2$ in $M$ at row $k$
  8.                  $j \leftarrow$ the column number of $c2$
  9.                  if $c2$ contains 1 and $j \neq i$
  10.                     add j to $D_k$
  11.                 end if
  12.             end for
  13.         end if
  14.     end for
```

Figure 7: Algorithm to determine candidates for reconstruction.

6 Implementation

This section provides some discussions regarding the issues of ARC implementation. We first describe the implementation of ARC in Linux. We then discuss and compare the memory-based and disk-based versions.

6.1 Device Driver

The ARC system is implemented as a device driver in Linux. This is the typical way of implementing RAID systems which has been adopted by RAID 0, RAID 1, RAID 5 and RAID 6. The benefit of this approach is kernel independence, meaning that the kernel does not need to be recompiled in order to accommodate ARC system. In addition, ARC can be deployed at run-time. No restart is required on the host machine. One should note that for current RAID system, although they are kernel independent, they still require some certain RAID support from the kernel. These support are embedded into kernel at compilation. Without these support, current RAID systems are unable to run. To avoid this problem, we built ARC from scratch, without requiring any specific kernel support. Thus, the ARC system can be deployed on any Linux system, with or without RAID support.

For the current implementation of ARC, we ignore the Translator module. Instead, we allow users to change the code of ARC directly to specify data structures and parameters for the desired RAID system. This approach is risky and inconvenient as users need to know which parts of the code to change. Users may also modify incorrect parts of the code and therefore cause the system to behave unexpectedly. In the next version of ARC, we plan to build the Translator module to separate users from modifying the code directly. Thus, it helps reducing both risks and inconveniences.

In the current ARC implementation, we provide two built-in checksum functions: sum and hash-based. The sum function just sums up all the bytes in one data block to make the checksum. The hash-based function makes use of function *csum_partial* in *checksum.h* to calculate the checksum. We also allow users to write their own checksum functions.

6.2 Memory-based v.s. Disk-based

The current ARC implementation works on simulated virtual disks. It means we split the memory into different chunks with uniform sizes. We then use each memory chunk as a virtual disk by implementing the read and write operations on memory with appropriate delays. This approach has several advantages:

- Easy to build and debug as reading and writing from and to memory is quite straightforward. This is essentially useful in the first stage of building the ARC system.
- There are no synchronization issues as memory is always synchronous.
- The code used for memory-based version can be used for disk-based version without significant changes.

However, the memory-based version does not work on real disks which is our ultimate goal. To solve this problem, we are currently working on a disk-based version that uses real disks instead of memory. This turns out to be hard due to the following reasons:

- In kernel programming, reading and writing to disks require communication with disk drivers through the *bio* structure. This structure is somewhat complex and making everything work smoothly is a challenging task.
- Most operations on disks are asynchronous. This is particularly difficult for our system as we need some specific order among read and write operations. For example, when updating a new parity block we have to make sure that the old data block for parity calculation has been read successfully before writing the new data block. Otherwise, the old data value will be lost.

Towards a disk-based version, we have designed an approach to work with the *bio* structure and to handle asynchronous operations. Specifically, we split the request function in the ARC system into smaller functions. Each small function is guaranteed to be executed completely (i.e. execute a complete read or write) before the next small function is called. By doing this, the execution order is guaranteed. However, this approach requires a precise split of the request function and that is time-consuming. Due to lack of time, we do not implement this approach in the current ARC system.

7 Performance Evaluation

7.1 Experiment Settings

To evaluate the performance of ARC, we performed several experiments in the VMWare environment with the Fedora 8 operating system. The machine configuration was Intel Core 2 Duo 2.2GHz CPU and 1GB RAM. All the experiments were performed using the memory-based version of ARC. We simulated the disk delay by adding a 15ms delay to each low-level disk read and a 17ms delay to each low-level disk write. The workload comprised reading, writing 30KB files and *mkfs* command. To compare the performance of a normal ARC I/O operations and the ones with reconstruction, we ran each workload twice. One was without failure. One was with simulated failures.

To compare the effect of different RAID levels with different checksum scheme, we used the following four systems in the experiments.

- **System 1:** 4-disk Raid 0+1 (striping and mirroring) with hash-based checksum.

- **System 2:** 4-disk Raid 0+1 (striping and mirroring) with sum checksum.
- **System 3:** 4-disk Raid 5 (striping with distributed parity) with hash-based checksum.
- **System 4:** 4-disk Raid 5 (striping with distributed parity) with sum checksum.

7.2 Results

Figure 8 shows the performance of normal reads and reads with reconstructions. For all the settings, normal reads take similar amount of time. We can see that different RAID levels or checksum schemes do not affect the performance of normal reads much because without failures, the system just needs two read twice: one for the data block and one for the checksum block. Reads with reconstructions usually take at least as twice as normal reads. For mirroring RAID (system 1 and 2), reconstructional reads take about twice the time of normal reads. This is because in a reconstruction for read, the system needs to read the mirrored block from the alternative disk, which takes an extra unit of read time. For RAID 5 (system 3 and 4), reconstructional reads take about four times as long as normal reads. This is because in a reconstruction in 4-disk RAID 5, we need to read three blocks (two data blocks, one parity block) on all the other three disks, which takes three extra units of read time. Generally, for RAID 4 and RAID 5, reconstructional read takes n times as long as normal reads, where n is the number of disks in the system. More generally, we can estimate the reconstructional read time using the Parity Matrix. To reconstruct a block, we need to read the parity block and all other data blocks that contribute to the parity block.

Note that the effect of different checksum schemes is not significant. This is because the CPU time needed to calculate checksum is almost negligible in comparison with the disk overhead (1 millisecond in our experiments).

Figure 9 shows the performance of normal writes and writes with reconstructions.

Comparing figure 8 with figure 9, we can see that for all the settings, normal writes takes a little bit longer than four times of normal reads. This is because for each write, we have five operations: reading the old data block, reading the parity block, writing the new data block, writing the checksum block, and writing the parity block. Since each write is slightly longer than read (2 milliseconds in our setting), the total time of a normal write is a little bit longer than four times of a normal read.

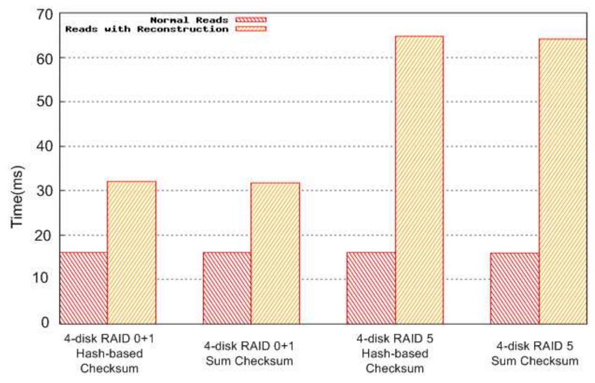


Figure 8: Performance of normal reads and reads with reconstructions.

There are two read operations in a write, either one or both of these reads can fail. When one read fails and needs reconstruction, the extra cost is the same as the extra cost of a reconstructional read (it is four times longer than a normal read). When both reads fail and need to be reconstructed, the extra cost is as twice as the extra cost of one reconstructional read.

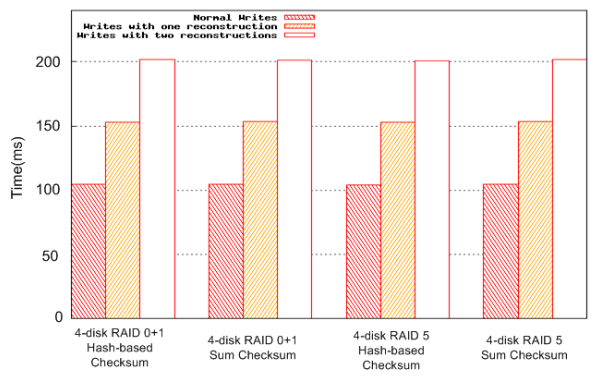


Figure 9: Performance of normal writes and writes with reconstructions.

Figure 10 shows the time line of the *mkfs* command on the 4-disk RAID 5. In this experiment, we injected random read failures. The probability of read failure is 20% and the probability of checksum inconsistency is 20%.

Thus, the probability that a read requires reconstruction is 40%. The lower points of the first part of the graph represent normal reads. The higher points of the first part represents reads with reconstructions. Comparing these two sets of points, we can see that on average, reconstructional reads takes four times longer than normal reads, which follows the argument we gave earlier. The lowest points of the second part of the graph represents normal writes. We can see that normal writes take a little bit more than four times more than normal reads. The middle points of the second part represents writes with one read failure and thus one reconstruction is needed. This failure can happen when either reading the old data block fails or when reading the parity block fails. The extra cost for one reconstruction is the same as the extra cost of a reconstructional read. The highest points of the second part in the graph represent writes with two failures and thus two reconstructions. This happens when both readings of the old data block and the parity block fail. The extra cost is as twice as the cost of a reconstructional read.

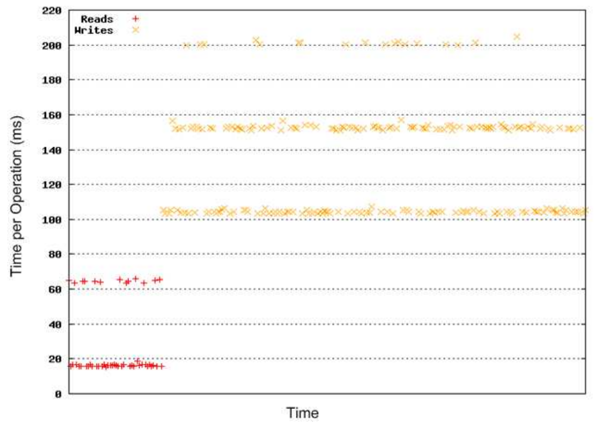


Figure 10: Time line of the *mkfs* command on 4-disk RAID 5. The x-axis is the actual time and the y-axis is the time per I/O operation. The first part (on the left) consists only of reads operations. The second part (on the right) consists only of write operations.

8 Conclusions and Future Work

ARC provides a way to specify various RAID settings, which gives much more flexibility and robustness than traditional software RAID systems. Using two matrices for parity parity parameters for checksum, ARC can be configured to act as any RAID system. Experimentally, ARC is shown to have reasonably good performance with expected behaviors. We see that reconstructions are costly and writes have much higher cost than reads. While the memory-based version of ARC works well, a lot of efforts need to be spent to make a working disk-based version. This suggests that kernel programming is much harder than user-level programming, especially when working with asynchronous systems such as disk drivers.

Arc is just the first step to approach the problem of automatic RAID construction. There are many issues that need to be solved. Among them, the more important ones are: (1) finishing the disk-based version of our automatic RAID system, (2) improving the current ARC system: adding more features, handling more error situations and adding user input correctness checking, (3) improving the performance, especially for writes, (4) designing a better language to describe the parity and checksum schemes.

References

[1] Aristos logic. *http://www.aristoslogic.com*.

[2] ivivity. *http://www.ivivity.com*.

[3] M. Blaum, J. Brady, J. Bruck, and J. Menon. Evenodd: An efficient scheme for tolerating double disk failures in raid architectures. *IEEE Trans. Computers*, 44(2):192–202, 1995.

[4] M. Blaum and J. Bruck. Mds array codes for correcting a single criss-cross error. *IEEE Transactions on Information Theory*, 46(3):1068–1077, 2000.

[5] W. A. Burkhard and J. Menon. Disk array storage system reliability. In *Symposium on Fault-Tolerant Computing*, pages 432–441, 1993.

[6] G. A. Gibson, W. V. Courtright II, M. Holland, and J. Zelenka. RAIDframe: Rapid prototyping for disk arrays. Technical Report CMU-CS-95-200, 1995.

[7] D. Kenchammana-Hosekote, D. He, and J. L. Hafner. Reo: a generic raid engine and optimizer. In *FAST '07: Proceedings of the 5th USENIX conference on File and Storage Technologies*, pages 31–31, Berkeley, CA, USA, 2007. USENIX Association.

[8] A. Krioukov, L. N. Bairavasundaram, G. R. Goodson, K. Srinivasan, R. Thelen, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Parity Lost and Parity Regained. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, CA, February 2008.

[9] J. Menon and D. Mattson. Comparison of sparing alternatives for disk arrays. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 318–329, New York, NY, USA, 1992. ACM.

[10] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks. In *ACM SIGMOD Conference on the Management of Data (SIGMOD '88)*, pages 109–116, Berkeley, CA, USA, 1988. USENIX Association.