RAID-X: RAID eXtended for Heterogeneous Arrays

András Fekete, Elizabeth Varki

Computer Science, University of New Hampshire
Durham, NH, 03824, USA
afekete@wildcats.unh.edu, varki@cs.unh.edu

Abstract

RAID has been around since the late 80s. It is well understood and has been implemented in many systems throughout the world. Technology has much improved since the 80s and disks are increasing in size and reliability at a rapid rate. When a RAID system has been set up with a certain sized drives and a disk breaks after a couple years of use it does not make sense to purchase a drive with the same capacity when a drive with twice the size exists. RAID-X solves the problem of heterogeneous disks in an array based on RAID. This paper introduces the implementation of such an array with heterogeneous disks. Additionally, with modern file-systems the authors demonstrate how to support growing the file-system while it is on-line.

1 Introduction

Today's big buzzword is big data. What is frequently forgotten is the underlying storage problem on the ever expanding data. There are many problems to solve with big data; accessibility, storage, searching. There are many discussions on the way the database is architected to solve this issue, but hardly any research in the low level storage mechanism. In the late 80s Patterson *et al*. [1] came out with the idea for RAID. Today it is ubiquitous and well understood. However, all the latest technology is based on 30 year old ideas. This paper gives a brief overview of RAID and what it means. There have been several tweaks proposed and tested. In a world where there are Solid State drives as well as magnetic drives and even experiments with RAM-based storage much has been developed in terms of storage technology. Algorithmically RAID has not changed since it was first introduced. The latest research has some very clever ideas but alas it is far too complicated to be worthwhile to implement in hardware.

Consider HERA [2], it is a step in the direction of heterogeneous disks in a RAID configuration, where they assemble parts of each disk into a parity group. The inherent problem is that once the system is initialized, it cannot handle addition or removal of different sized disks. It concentrates on the MTTF of the RAID, but omits the consideration of rebuilding with a possibly larger disk.

The algorithm presented in this paper eliminates this restriction and affords several other luxuries that aren't available. One of these is the ability to have the total size of the RAID increase with the addition of another disk. Another is the graceful degradation of total space in the event of a disk failure.

Conceptually, RAID-X removes the predictability of where a block of data is stored on the physical disk. This comes at the cost of having to store a lookup table, but in relation to the size of storage that is available this cost is negligible. The benefits of this include having a variable sized partition which is similar in concept to that of a Linux Volume Manager.

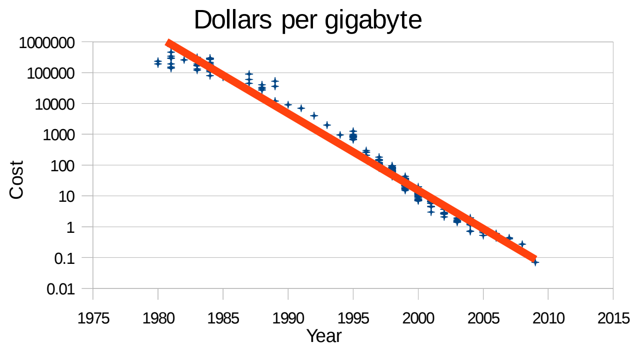

Figure 1: Cost of drive capacity over time

The graceful degradation feature of this system allows RAID-X to exit from a degraded state by reorganizing the way data is stored on the disks as long as it is possible to shrink the file-system on top of the RAID. With this feature, it is possible to lose another disk without catastrophic effects. This is not possible with traditional RAID. Traditional RAID can only rebuild what has been lost on the missing disk once a new disk has been put in place.

2 Background

Hard drives are notorious for being sensitive. Since they are mechanical devices, great deal of care must be taken for their use and keeping it operating. Throughout the years, the storage density of drives has been increasing at a rate of 40% per year. The cost per gigabyte is decreasing at an exponential rate (see Figure 1).

Today more and more data is stored, but the reliability of the drives stays the same. The Bathtub Curve (see Figure 2) has been used as the general plot of the failure rate of a mechanical system over time. In the beginning, the drive is being broken in (blue line), so it can have a malfunction due to the way it was manufactured. As time goes on, it has a constant likelihood for failures due to natural causes (green line). As we get to its life expectancy, the likelihood of a failure starts increasing due to parts wearing out (red line). Summing these curves gives us the bathtub curve simply because it looks like the cross section view of a bathtub.

The shape of this curve hasn't changed since the first day hard drives were introduced. What has changed is the time scale, which is measured by the Mean Time To Failure (MTTF) of a drive. Nowadays having a million hours before failure is rather normal for a drive, while 20 years ago it was maybe 10K hours. So not only are drives getting more dense, but they are failing less. This is a good thing, but the failure rate isn't keeping up with the rate the capacity increases.

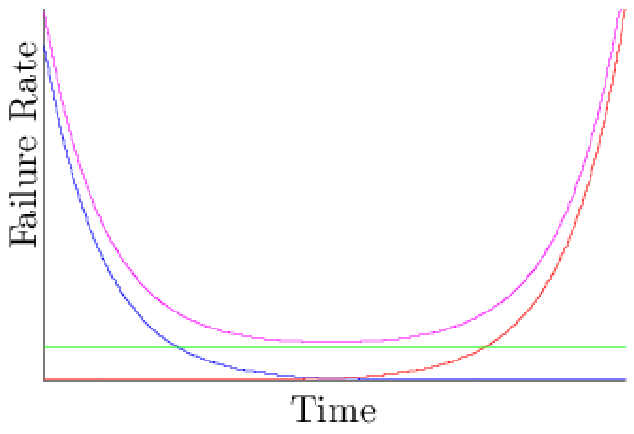

Figure 2: The bathtub curve: Blue – breakin failure, Red – wearout failure, Green – random failure, Magenta – sum of all failures

Hard drive manufacturers have also put in a lot of effort into making each drive withstand harsher environments. Drives in laptops are especially vulnerable due to the simple fact that they are mobile devices and must withstand shock and vibration from multiple angles. Systems like Apple's Sudden Motion Sensor, Dell's Free Fall Sensor, or Lenovo's Hard Drive Active Protection System are all created to pro-actively defend against these outside events which don't normally occur in desktop or server drives.

Another angle to consider is that mechanical drives are really slow at reading and writing data. This is because the head has to move to the correct track, and the platter has to spin to the proper sector. This all takes time and there's a need to have it be reduced so that programs don't spend their time waiting for I/O operations to finish.

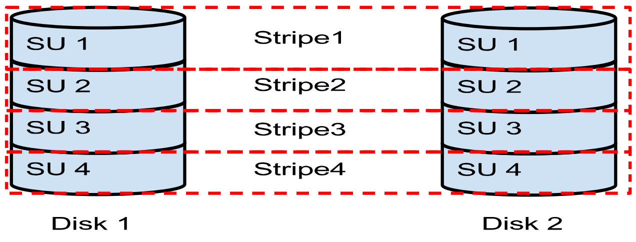

Figure 3: Stripe vs Stripe Unit

Patterson *et al.* [1] came up with methods to combine the MTTF of multiple disks to decrease the likelihood of data loss as well as speed up I/O operations by combining the speed of drives together. They introduced the concept of Redundant Arrays of Inexpensive Disks (RAID).

This research takes a look at the most popular types of RAID, their benefits and drawbacks. Most RAID concepts and techniques rely on the fact that all disks are identical.

In RAID, each disk is split up into stripe units. They are generally based on the block size of a disk. Stripe units that share the same logical location across different disks are grouped into a stripe (see Figure 3).

Depending on the number of disks in the array, it is possible to achieve certain so-called RAID levels. The simplest level is RAID0 and RAID1 also known as striping and mirroring respectively. This is the simplest kind of RAID that can be implemented with as little as two drives. More advanced RAID levels are ones like RAID5 (single parity) and RAID6 (double parity). Parity refers to how redundant the algorithm is which determines how many disks can die before data loss occurs. There have been numerous enhancements [3][4][5][6][7] to these parity calculations that have either made the calculations simpler or allow for more disks to fail.

In recent years, solid state storage has come down in cost and is rapidly being incorporated into RAID technologies to speed up access. Research [8] is being done to make use of this new technology which requires wear leveling across the disks.

3 RAID-X

Up until recently, it has been understood that all disks in an array are of the same size, or the space available is of the size of the smallest disk. This causes inefficient disk usage and causes problems when a disk in an array needs to be replaced, as the same model or size disk may not be available.

Liu *et al [9]* have demonstrated a different layout scheme on heterogeneous disks where it will use a RAID5 topology until it fills up a disk, then it will use the remaining disks to continue allocating stripes of a RAID until all disks are full. This creates a complicated mechanism where it must store a pseudo stripe pattern where each stripe unit on the disk is located using a complex computation.

RAID-X is a novel solution to using heterogeneous disks where it removes any logical restriction on where a stripe unit is, but still keeping the logical block mapping an O(1) problem.

4 Chunking

In RAID-X the concept of chunking is introduced. To move forward, one has to move away from the idea of stripe units. With more RAM and faster CPUs one can afford a loss in strict logical placement of data.

RAID-X works by creating groupings of sequential logical blocks on a disk called chunks instead of stripe units. A stripe is then made up of chunks, but now, instead of stripes being at a predictable location on each disk, a stripe will be scattered throughout the array. Each chunk in a stripe will have related chunks stored on another disk in an arbitrary location to ensure that if a disk breaks it does not take out more than one chunk in one stripe. The distinction here is important, because chunks can be paired anywhere on the disk, whereas stripe units are determined based on physical location.

With RAID-X it will be necessary to keep a lookup table in RAM to have a map from the physical to the logical mapping of chunks on the disk. Since this will be an array whose size is bounded by the number of chunks on the drive, it will be an O(1) lookup. The size of a chunk will be 4096 bytes or some multiple of that number as per the standard put forth by IDEMA (International Disk Drive Equipment and Materials Association) makes that be the most efficient size to transfer on today's drives. The size of the chunk also determines the amount of RAM that will be used, but having a large chunk size affects the efficiency at which small fragmented data can be accessed. This is a topic of much debate in traditional RAID, and the same reasoning applies to RAID-X. The smaller chunk size is good for quick access to small files, but a larger chunk size allows for better throughput.

Procedure: initRAID(*allDisks*, *CS, startCID*)

```
1: CID = startCID
2: DFC = getDisksWithFreeChunks(allDisks)
3: while(DFC > CS)
4:    curStripe = { }
5:    validDisks = allDisks
6:    for i = 1:CS
7:       curDisk = mostChunksFree(validDisks)
8:       curStripe = curStripe U curDisk
9:       validDisks = validDisks – curDisk
10:      if(chunksFree(curDisk) == 1)
11:         DFC = DFC – 1
12:      endif
13:   endfor
14:   allocStripe(curStripe, CID)
15:   CID = CID + 1
16: endwhile
```

Algorithm 1: This is how to initialize a RAID-X array. *DFC* is disks with free chunks; CS is number of chunks in a stripe; allDisks is the set containing the disks in the array

5 Initialization

To initialize the array, more work has to be done compared to other RAIDs. A total number of free chunks must be calculated for each disk that is being added. Then take a chunk from the disk with the largest number of free chunks, and put it in the same stripe as the disk with the next largest number of free chunks. Once the stripe has been allocated, mark all the chunks used. Continue this process until the number of disks with chunks available in the system is zero or one less than the total number of chunks in a stripe (call this *CS*). It is guaranteed to have at most *CS* - 1 free chunks because if *CS* disks have chunks available, then they can be added and used to store chunks. The initialization function takes in the *startCID* value (which is the starting chunk ID) because the function itsself is generic enough to be used at times when the RAID is being reshuffled. This is discussed further in later sections.

It must be pointed out that each disk may not necessarily contain a chunk of every stripe. The only time that each disk will contain a chunk of each stripe is when CS is equivalent to the number of disks in the array. This property becomes useful when trying to access multiple stripes at the same time, the access time can be decreased by spreading the load across all the disks.

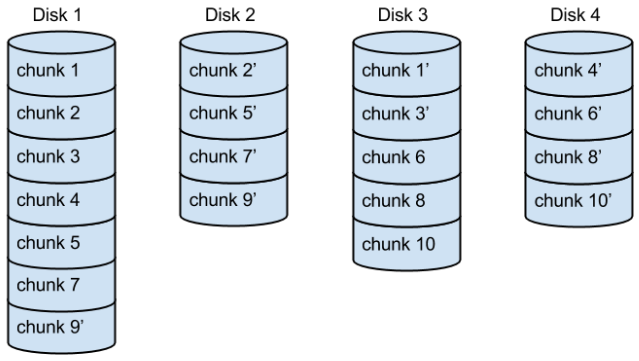

Figure 4: RAID-X - 2 disk redundancy

To look at an example, one can take Figure 4 and assume all disks are empty and that an underlying RAID which only requires 2 disks is requested. For the sake of simplicity, a RAID1 level (which is mirroring) is chosen. Following Algorithm 1, we calculate that Disk 1 has the most chunks followed by Disk 3, then Disk 3 and 4. In the first iteration, Disk 1 will get its first free chunk allocated to CID=1 with a mirror on Disk 3. This will become our first stripe. Next we have Disk 1 with the most free chunks and the remaining 3 disks have the same amount. Thus CID=2 can be allocated onto Disk 1 and Disk 2 is arbitrarily chosen as its mirror. Now Disk 1 still has the most free chunks, so it is chosen, and Disk 3 is arbitrarily chosen from the set containing Disk 3 and 4. The process continues until there aren't any available disks to mirror across which in this case is at most one.

It is also important to realize, that with homogeneous disks, RAID-X is the same as the underlying RAID in its layout given that the underlying RAID uses the same number of disks as there are disks in total.

The main advantage of this system is to make use of the extra space left over on larger disks in a redundant fashion. Another advantage is that when a disk dies, if the amount of data stored on the array would fit in the now smaller array, the system could reorganize the chunks so that the data is all redundant and the system is able to suffer the loss of another drive. Since the probability of having 2 drives die simultaneously is very small (in the case of a 1 disk redundancy), this becomes a viable option similar to RAID1 and RAID5.

Having this type of redundancy allows for the MTTF to be rather large, much larger than any other kind of RAID. This is because without the need for spare disks, the RAID can repair its self into a redundant state as long as the array is not completely full with data. The worst case MTTF is same as for RAID1, but in practice what is likely to happen is that smaller disks will have more hours on them, and thus the likelihood of them failing is larger than the larger disks.

Like in traditional RAID, RAID-X also keeps track of which disks are members of a particular RAID. This is useful for two things:

1. Detection of RAID groups from a pool of disks
2. Detection of disks missing from the RAID

Both points are important in the case of system initialization. Point 1 is useful to group the disks into their respective RAID sets from a collection of disks. Point 2 is necessary when a drive is removed and the RAID is being rebuilt.

5.1 Stripes with 3 or more chunks

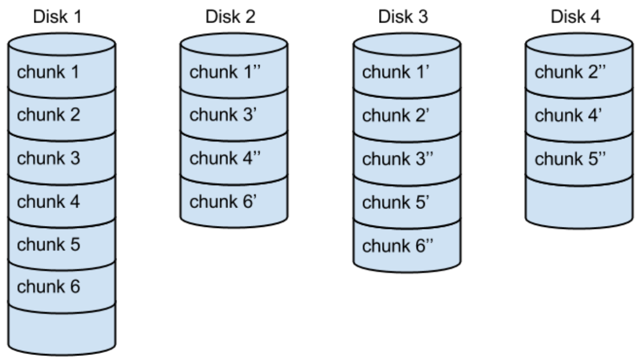

Figure 5: RAID-X - 3 disk redundancy

RAID-X is also well suited for having more than 2 blocks in one chunk. For example, if the underlying RAID is RAID5, then one can see in Figure 5 that with the same set of disks as before in Figure 4 we can realize the different data layout. Of course, since there are 20 chunks available, which is not evenly divisible by 3, so the remainder chunks will be unused. Even still, 2 extra stripes are gained out of the system. Adding in an additional disk with a single chunk would allow for full utilization of all disks. Similarly, any disk with size calculated using Equation 1 (where $n \geq 0$) would be able to expand the array to full utilization.

$$diskSize = missingChunkCount + n * CS \quad (1)$$

Looking at Figure 5, it can observed again, that Disk 1 has the most free chunks, then Disk 3, then Disk 2 and 4. When the first stripe is created, it starts with Disk 1, then Disk 3, then arbitrarily picks Disk 2. For the second stripe, Disk 1 still has the most free chunks, then it arbitrarily picks Disk 3, then Disk 4.

5.2 Data access

Another interesting advantage of RAID-X is that the data access algorithms would be much different that of traditional RAID. Assume that we want to read the data in stripe 4 from the disk array in Figure 5. We can play tricks with optimization based on whether it is faster to compute the XOR of the last chunk or to wait for the disk to read it. It is important to point out that the heads of each disk are in a relatively random configuration (compared to that of traditional RAID), therefore we can also do a simple shortest path optimization of which disk is best suited to read which chunk since it is possible to keep track of what chunk each disk last read.

This can as a byproduct reduce wear on the disk as well as energy usage. Many others [10][11] have tried to reduce the power consumption of RAID.

For writes to the array, there is no such optimization that can be applied since each chunk must be written to. As with all other RAIDs, RAID-X is also prone to the “Write hole” phenomenon, where if there is a power loss during a write, the RAID will not be able to correct itsself. Generally, this problem is left to be solved by the file-system to replay its last journal entry or run a consistency check. Since the probability of this occurring is very small, not much has been done to prevent it.

Procedure: addDisk(*raidDisks, maxCID*, *CS*, *newDisk*)

```
1: curCID = maxCID
2: for i = 1:floor(chunksFree(newDisk) / CS)*CS
3:     validDisks = containsCID(allDisks, curCID)
4:     curDisk = leastChunksFree(validDisks)
5:     reallocChunk(newDisk, curDisk, curCID)
6:     curCID = curCID - 1
7: endfor
8: initRAID(raidDisks ∪ newDisk, CS, maxCID + 1)
```

Algorithm 2: Showing how to add a disk to the RAID-X array. RaidDisks is the set containing the existing disks in the array with maxCID as the largest Chunk ID.

6 Array degradation

As with any other RAID, the array can get in a degraded state when a disk has gone missing for any number of reasons. There will be degradation events such as when a disk dies or when there is a new disk inserted. In RAID-X, this is the opportunity to reshuffle chunks. Generally, it is desired to have chunks in long sequences stored on the disks to optimize large reads. It is not vital though to have it that way on all disks as non-sequential reads would be sped up by having the chunks scattered on several disks.

When a disk is removed from the array for any reason, as it was just discussed, it is in a degraded state. RAID-X has the ability to discard some unused stripes and reallocate the newly freed chunks to those stripes that have lost a chunk because of the missing disk. Once the process is completed, the array is once again fully redundant and can suffer the loss of an additional disk. This technique is called graceful degradation. It is not required to do graceful degradation as it may be unnecessary work performed on restructuring the array when a new disk is standing by to be added.

While the array is in a degraded state, it will be less than optimal in performance similar to a traditional RAID because of having to calculate XORs of all the chunks in the stripe.

When a removed disk is added back in, it has to be checked for consistency. This consistency check is similar to rebuilding a traditional RAID except that only those stripes which are on the newly connected disk have to be checked.

Fixing a degraded array has a smaller amount of time needed for rebuilding than for a traditional array, because the original disks in the array will be also written to while the chunks are being reorganized. Therefore it is more of a parallel process compared to when a traditional RAID is being rebuilt and a single disk is being written to.

6.1 Adding new disks

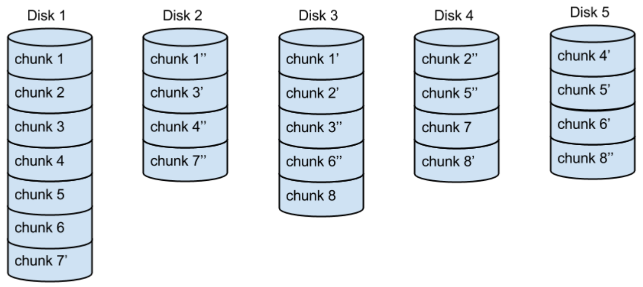

Figure 6: RAID-X - adding a new disk

When there is a new disk added to the array, we will determine the number of chunks available on the disk, take only whole divisors of *CS* number of chunks and copy chunks from other disks in the array to the new disk. This will free up chunks on the other disks which we'll allocate into new stripes. We start by taking the chunk with largest ID from the disk which is the most full and going down in chunk IDs until all the chunks have been transferred. Obviously, on the new disk, we'll put these in increasing order to help with sequential reads. This method best preserves sequences on disks.

Next, it might be necessary to shift down any chunks to fill in the gaps. This might not be as important of a step and will be a topic of future research. Seeing that this area on the disks will likely have non-sequential chunks it might not make sense to do the effort. In fact it may be beneficial to have chunks with larger IDs be closer to the front of the disk as access times are generally better.

The last step is to fill in the now empty chunks using the same algorithm as in the initialization. Figure 6 shows the final result of such a restructuring.

When a new disk is added to the RAID, the amount of space available is obviously increased. With new advances in file systems, it is possible to increase the available space on the file system in-situ. An example of such a file-system is the widely used EXT3/4.

6.2 Graceful degradation

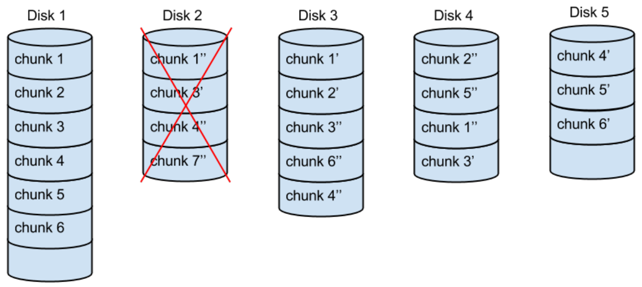

Figure 7: RAID-X - Removing a disk

In the event that a disk is removed from the array due to hardware failure or otherwise, the array enters a degraded state. The first task is to resize the file-system similar to what was mentioned in the previous section. The size of the reduction depends on the resulting size of the array after the RAID is rebuilt. This can be calculated by simply taking the number of chunks that were lost and dividing by the number of chunks in the stripe rounding up. This will tell how many stripes have been lost.

When the data is condensed onto the remaining chunks thanks to the file-system resizing, the higher numbered chunks can be eliminated. As mentioned in section 6.1, it may be wise to shift down the chunks on the disk. One can

```
Procedure: delDisk(raidDisks, maxCID, CS, selDisk)

1: usedStripes = ceil(numUsedChunks(selDisk) / CS)*CS
2: shrinkByStripe(raidDisks, usedStripes)
3: maxCID =  maxCID - usedStripes
4: for i = 1:numUsedChunks(selDisk)
5:     curCID = getCID(selDisk, i)
6:     validDisks = raidDisks
                          - containsCID(raidDisks, curCID)
7:     destDisk = mostChunksFree(validDisks)
8:     reallocChunk(destDisk, selDisk, curCID)
9: endfor
```

Algorithm 3: Removing a disk from RAID-X

see in Figure 7 the shift-down step was omitted and that with the removal of Disk 2, there were two stripes that had to be removed. Any disk that stored chunks from those stripes has those particular chunks back on the free chunks pool. The data lost in Disk 2 was recreated by using the techniques of the underlying RAID which in this case was taking the XOR of the remaining chunks in each stripe.

7 Conclusions

In this paper, a different approach to RAID is discussed using chunks instead of stripe units. This new approach loosens the restrictions of traditional RAID systems to allow for simple algorithmic implementation of a RAID on a heterogeneous disk array.

The system could be used as an intermediary system between the storage and the underlying RAID level. This would then incorporate all the research and development that has been done with RAID and be able to expand it to a heterogeneous disk system.

8 Future Work

The authors intend on further exploring the potential of this algorithm and comparing it to current standards for RAID. It will be interesting to compare the overhead of keeping track of the chunk locations in memory as opposed to having it be calculated. Given this baseline, the optimality of this algorithm over the current RAID implementations can be shown.

Determining the speed of a disk where the chunks are not shifted around after a change in the number of disks will also be a big concern of this research. In addition to that, when arbitrarily selecting a disk to place a chunk on might not be the best solution. Instead it may be necessary to devise an algorithm that picks the optimal disk to spread the chunk on to which depends on the correlation of disks in the stripe. Simply put, if the selected disk has a high number of stripes shared with the disks in the array, it may be wise to put the chunk on another disk to even out the workload.

Benchmarking the rebuild speed of RAID-X versus a traditional RAID will yield additional results on the overhead costs of using RAID-X. There will be an extra angle where the time to perform a graceful degradation will be studied. Since this is not available in traditional RAID, it can still be compared to a rebuild time.

References

[1] D. A. Patterson, G. Gibson, and R. H. Katz, “A case for redundant arrays of inexpensive disks (RAID),” *ACM SIGMOD Record*, vol. 17, no. 3. pp. 109–116, 1988.

[2] R. Zimmermann and S. Ghandeharizadeh, “HERA: Heterogeneous Extension of RAID.,” in *PDPTA*, 2000, no. Pdpta.

[3] H. Anvin, “The mathematics of RAID-6,” *online Pap.*, no. January 2004, pp. 1–9, 2007.

[4] C. Jin, H. Jiang, D. Feng, and L. Tian, “P-Code: A new RAID-6 code with optimal properties,” … *23rd Int. Conf. …*, 2009.

[5] G. a. Alvarez, W. a. Burkhard, and F. Cristian, “Tolerating multiple failures in RAID architectures with optimal storage and uniform declustering,” *ACM SIGARCH Comput. …*, 1997.

[6] P. Xie, J. Huang, Q. Cao, and C. Xie, “Balanced P-Code: A RAID-6 Code to Support Highly Balanced I/Os for Disk Arrays,” *Networking, Archit. …*, pp. 133–137, Aug. 2014.

[7] M. Chen, B. Yang, and C. Cheng, “RAIDq : A software-friendly , multiple-parity RAID,” *(USENIX) HotStorage*, pp. 1–5, 2013.

[8] M. Balakrishnan, A. Kadav, V. Prabhakaran, and D. Malkhi, “Differential RAID,” *ACM Transactions on Storage*, vol. 6, no. 2. pp. 1–22, 2010.

[9] Y. Liu, X. Xie, and H. Li, “A Data Layout Method in Heterogeneous RAID,” *Scalable Comput. Commun. …*, pp. 218–225, 2009.

[10] D. Colarelli and D. Grunwald, “Massive arrays of idle disks for storage archives,” *Proc. 2002 ACM/IEEE …*, vol. 00, no. c, 2002.

[11] T. Chen, T. Yeh, H. Wei, and Y. Fang, “CacheRAID: An Efficient Adaptive Write Cache Policy to Conserve RAID Disk Array Energy,” *Proc. 2012 …*, pp. 117–124, Nov. 2012.