RAID on a Heterogeneous Disk Array

András Fekete and Elizabeth Varki
University of New Hampshire
afekete@wildcats.unh.edu, varki@cs.unh.edu

Abstract

RAID assumes homegeneous disks. When a disk in RAID fails, it may be replaced by a larger disk, but the extra space in the new disk is wasted. To address this problem, this paper proposes RAIDX, RAID eXtended for heterogenous disks. Similar to RAID, RAIDX bundles data across its disks; the stripes of RAID and RAIDX are constructed differently. In RAID, a stripe is a row of stripe units, one per disk; the stripe units are at identical locations on each disk. In RAIDX, a bundles is a row of chunks, with at most one chunk per disk; the chunks in a bundle need not be on the same location on each disk. Chunks of a bundle may be relocated dynamically when new disks are added to or removed from RAIDX. RAIDX requires a lookup table to map block numbers to chunk locations. This table is at worst only 0.3% of the total array size. The lookup tables add overhead to RAIDX, however, experiments demonstrate that RAID and RAIDX have comparable speeds when the array consists of similar disks. RAIDX supports arrays that contain a mix of hard disks and solid state disks. This combination can be used to increase the access speed of the array by directing traffic to the faster disks. RAIDX is compared to RAID by experiments that attempt to exercise similar functionality on the same hardware. We show that the proposed lookup table design adds only a little computational overhead and RAIDX approaches the speed of a traditional RAID array.

1 Introduction

Redundant Array of Inexpensive Disks (RAID) consists of several disks logically bound together to form a single storage unit, capable of higher performance than each individual disk. Hardware RAID uses RAID cards between the storage devices and the motherboard while software RAID uses the system CPU and memory to achieve the same function. Additionally, RAID offers redundancy to prevent data loss in the event of a drive failure. While the redundancy causes computational overhead, there is a net gain to this organization of storage.

There are several RAID configurations, of which RAID5, RAID10, and RAID6 are the most popular. All configurations of RAID assume that the disks are identical. When disks fail, they are usually replaced by identical disks. Disks have an increasing Mean Time to Failure (MTTF), and they are growing in size. The combination of these two facts makes it plausible that when a disk fails in an array, it is likely replaced by a larger disk. Over time, a homogeneous array gets replaced by a completely new array with the old hardware discarded, a costly solution. A second solution is to replace the failed disks by new large disks, but the additional space in the new disks are not utilized. A third option is to extend the second solution by using the additional space as a separate storage unit, which interferes with the operation of RAID. All these solutions are ad-hoc, wasteful, and expensive. This paper addresses this issue by evaluating a RAID configuration called RAIDX [**?**], for heterogeneous disks.

RAIDX - RAID eXtended for heterogeneous disks - configures different types of disks into a single array. For example, SSD and HDD could be combined into a single RAIDX array. The configuration of RAIDX ensures that the speed of the array approaches that of the faster disk in the array. Moreover, when disks of several sizes are placed in a RAIDX array, the additional space in the larger disks becomes available. RAIDX also supports all the traditional RAID levels (striping, mirroring, and parity). RAIDX, designed for heterogeneity of disk sizes and speeds, provides optimal performance regardless of homogeneous or heterogeneous disks.

This paper describes the RAIDX system architecture. The traditional, homogeneous RAID organizes storage data into stripes that span the disks uniformly. Each stripe (row) consists of stripe units, one per disk; the stripe units corresponding to a stripe at the same location on each disk. RAIDX has a completely different organization since disks vary in size. The bundles in RAIDX consist of chunks (not stripe units), which are not necessarily at the same location of each disk. Moreover, bundles need not span all the disks;

larger disks participate in more bundles than smaller disks. In traditional RAID, stripes are just rows of storage data that logically follow each other. In RAIDX, bundles are organized for maximizing storage utilization of different sized disks.

Like in traditional RAID, the chunk size is selectable at initialization. With a smaller chunk size, the number of bundles increases thus increasing the space requirements for the lookup table both in memory as well as on disk. Nevertheless, the size of the lookup table in the worst case is less than 0.3% percentage of the array size. If the chunk size increases, the lookup table size decreases. The lookup table still offers an O(1) retrieval of chunk addresses. This is because once the array is assembled, the lookup table no longer changes.

In this paper we compare RAIDX performance to that of traditional RAID. We take into account one of the simpler RAID types, namely mirroring as a comparison. While there is a difference between the speeds achieved by the two implementations, there are many things that can still improve the speed of RAIDX. We have shown a 10 times speed increase over an individual drive in our 7 disk array. Mirroring RAIDX outperformed traditional RAID in terms of writes, but fell short on transactions with mostly reads. The crossover point occurs in transactions where at least 60% of the access is a read. A simple remedy for this would be to add a cache layer which will keep most frequently accessed bundles in memory.

2 Related work

RAID [8] calls a piece of data or parity a stripe unit. Multiple stripe units are organized into stripes. Stripe units at the same logical location are in the same stripe. Stripe units are the smallest blocks that a RAID system can access.

Although other RAID algorithms exist, typically, RAID10, RAID5 and RAID6 is most commonly used. RAID10 is a combination of mirroring and striping across the SUs where RAID5 and RAID6 are single and double parity algorithms respectively. A parity is calculated by taking the XOR of the data SUs in the parity. For RAID5, these are the SUs in the stripe other than the parity stripe unit. In RAID6, the first parity can be the same that of RAID5, where the second is a myriad of possibilities [5, 7, 13, 14].

The basic rule in making an algorithm redundant is that each piece of the data and parity must be kept on a separate disk. The only way this is possible on differently sized disks is to treat all disks equal to the size of the smallest disk in the array. One trick that has been done is to place a second array on top of the remaining space on the disks. This process repeats until there are only two disks with free space which is then combined with a RAID1 array. The trouble this causes is that if both RAID arrays are accessed simultaneously there is a steep performance degradation. In any disk access, it is well known that the throughput is heavily influenced by the seek time required to access the data [9]. Therefore, any rotational disk that has multiple RAIDs on it will suffer from this.

Many solutions to this have revolved around obfuscating the underlying storage devices such as Logical Volume Management [12] (LVM). This divides up a physical volume (disk) to physical extents which can be allocated to logical extents in a logical volume. LVM leads to waste by adding a layer of abstraction in addition to sub-optimal utilization of hardware by treating all disks as identical. Others have tried to create virtual arrays [11] and combine them together into a single system.

RAIDX combines the layer which creates an abstraction of the underlying heterogeneous storage with the layer that provides redundancy and speed improvements over a single drive. Being aware of the low level storage, RAIDX can make optimizations on the ordering of how transactions get executed.

3 RAIDX

Our current implementation is done entirely in software, but there is no restriction that would prevent a hardware implementation. Using the Linux network block device module, it is possible to create a block-level user space environment called BUSE [3]. This creates a block-level device that can represent the RAID array. By avoiding kernel modules, a much simpler implementation can be accomplished using object oriented languages. Using direct file access, we can force the kernel to retrieve data directly to the userspace memory and avoid unnecessary memory copies.

3.1 RAIDX Instantiation

In RAIDX there are some additional steps to assembling the RAID. When creating a new RAIDX set, one must define the number of chunks per bundle. The smaller the number, the closer to full utilization of the disks there is, but the larger the lookup table will be. This is because for a smaller chunks per bundle, there are more possible ways of spreading the chunks across the disks to optimize the layout. Each disk in the array can only store one chunk in a bundle to maintain redundancy. A disk with more than one

Table 1: RAID level and minimum chunks per bundle

RAID Level	0	1	5	6	10
chunk per bundle	1	2	3	4	4

chunk out of a bundle would be a weak link in the array where only a single disk failure is tolerated. If the chunks per bundle is equal to the number of disks in the array, then we have a traditional RAID layout and the number of stripes is determined by the smallest disk in the array. RAIDX is similar in that once the RAID is assembled, the chunks per bundle cannot be easily changed. The difference exists in the number of bundles. RAIDX allows for bundles to be added as well as removed based on the number of available chunks in the array. This means that as the array ages and transforms, it is possible to further increase the storage capacity. With modern filesystems, it is possible to resize the filesystem to follow the size of the underlying device. Many filesystems also allow to do this change on a live system.

The minimum necessary chunks per bundle for traditional RAIDs are shown in Table 1. The largest traditional RAID requires 4 chunks. In our experiments, where we have several different sized disks, we can see that this still produces a RAID with nearly the full size of the physical array. With a different set of disk sizes this same phenomenon occurs. The larger the number of disks in the array, the better the ability to make use of the full disk space.

To initialize the array, the number of chunks on each disk is calculated and these become the number of free chunks. It is possible to estimate the maximum number of bundles (see equation 1), but not every configuration of physical disk sizes allows to have all the disks used. A simple example is taking an array with a 1TB disk and two 250GB drives. There is no possible way to make full use of the 1TB array in a redundant array. If there were two chunks per bundle, the size of the array is at most 750GB using equation 1, but because of the large disparity between disk sizes and the small number of disks in the array, the array size is going to be only 500GB. If we added two more 250GB (or one 500GB) disks then we would reach the maximum size of the array.

$$numStripes = \frac{totalArraySize}{chunksPerStripe * chunkSize} \quad (1)$$

In traditional RAID, a hot spare is a drive that is put into the array as a backup drive that is normally unused, but in the event of a disk failure it becomes activated and the array fails over to that drive. In RAIDX, the concept of hot spare becomes unnecessary, because it is better to have the added performance of an additional disk in parallel. It is possible to put a smaller filesystem on the RAIDX array, so that if there is a disk failure, the RAIDX array can reorganize the chunk locations without needing to modify the filesystem.

3.2 RAIDX lookup table

As all chunks in a bundle can be arbitrarily allocated on the disk, we must keep a lookup table to later determine the selected locations. The table on the disk is much different than what is kept in memory. The disks store only information about those chunks that are stored on it. The size of the table on the disk is constant, because the number of chunks a disk can store is based on the size of the disk and the size of a chunk.

The size of the lookup table depends linearly on the size of the chunks in the array. The smaller the chunks, the more that fit on the disks thus larger the table. This is true for both on-disk and in-memory tables. The lookup table only changes when there is a disk added or removed. Once a table is read from disk, there is no need to make any updates. The in-memory lookup table is intended to provide an O(1) lookup for the location of each chunk. Loss of the in-memory table is not a problem, because it can always be reconstructed from the tables stored on the disk.

One of the many advantages of RAIDX is that it is possible for the array to return to a fully redundant state after a disk failure but before that disk is replaced. This is because with a lookup table, the bundles can get redistributed across the remaining disks. With most modern filesystems, it is possible to reduce the filesystem size as long as the new size is greater than the size of the data stored on the filesystem. Once the filesystem has been resized, the RAIDX array can be restructured to remove the additional bundles and update the lookup table with the new locations of the chunks of a bundle. After the restructure, the RAIDX array is now fully redundant and can handle an additional disk failure. Conversely, when the data storage on the RAIDX array has become full, it is possible to add an additional disk to increase the physical storage capacity. This additional disk can be incorporated into the array by expanding the lookup table and reshuffling the chunks across the array.

3.3 RAIDX1

Using RAIDX, we looked at mirroring RAID. With chunks scattered across the disks, mirroring does not occur like it does on a traditional RAID. It is more akin to a RAID10 implementation. Figure 1 shows an example with two chunks per bundle that would be used

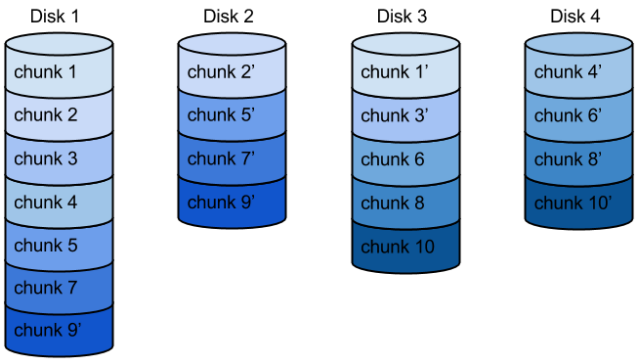

Figure 1: RAIDX array showing a 2 chunk per bundle setup

for a mirrored RAID setup. Notice that if bundles 1-4 are requested, all the disks could be utilized. This does not necessarily mean that they should. Depending on the speed of the disks, a better solution might be to use only one disk to access the data. Recall that the time that takes the longest on any magnetic disk is the seek time. So if all four disks must seek to the same location to retrieve a small chunk, then in essence, each of the slower disks in the set are trying to keep up with the fastest disk. Disk scheduling methods are nothing new [10], what is new is how to select the best disk among a set of different disks. For this, we tested 4 different algorithms. The first (alg1), iterates through all disks, and only considers disks that are either twice as fast as the selected, or is idle while the selected is busy, or the distance the disk is estimated to seek is less than the currently selected one. If any of the considerations is true, then the disk under consideration is set as the selected one. The second algorithm (alg2) considers all disks independent of the speed. The third algorithm (alg3) is like the second, except it omits the busy consideration. The fourth algorithm (alg4) only looks at the seek distances.

Data access is treated as an individual chunk being the smallest block of data accessible. If a transaction requires multiple chunks, the best disk is decided on a per-chunk basis starting with the chunk on the first logical bundle. In our tests, we always had the bundles organized in ascending order. When disks have been added and removed from the array, this may not always be the case. It depends on how the algorithm that adds in new disks to a previously degraded array places the chunks. For the present experiments, we need not concern about this scenario.

4 Experimental setup

To test the speed of the array implementations, the standard testing tool called flexible I/O tester (fio) [1] was used. Fio is a tool for any arbitrary I/O throughput testing. It is commonly used as a comparison tool by many researchers [2, 6] for arbitrary I/O traffic generation. It is possible to generate random reads and writes with a desired percentage being reads. The main drawback is that it takes a long time to get the results as the transactions have to be executed on the given hardware.

In our tests we use fio to issue random reads and writes, starting with all writes and incrementing by 10% reads to 100% reads. This can show how the RAID performs with different types of loads. In initial tests, the size of each read or write is a constant 10MB. Subsequent tests were done with a random size ranging from 1KB to 10MB. Each test iteration was run for approximately 60 seconds. An iteration consisted of creating constant transactions at a set reads to writes ratio from 4 separate thread sources. This produced a very large number of transactions to exhaust any kind of system buffer or cache device and produce a reliable average transaction rate. The speed of the disk was recorded as the amount of data transferred in the actual amount of time.

A decision regarding the chunk size needs to be made when allocating a RAID storage device. A multiple of the disk block size is used for an optimal disk interface. In 2011, sector sizes have been defined to be 4096 bytes on all commercial drives [4]. In prior years, the sector size has been 512 bytes, but the increase in drive capacity allowed for a block size increase to make transactions more efficient.

In the RAIDX instance, the size of a chunk was varied across tests. The tested sizes included: 16K, 32K, 64K, 128K, 256K and 512K bytes. These are all the typical stripe unit sizes for traditional RAID.

Our control in this experiment was using the standard Linux MD RAID implementation on the same drives. For both the control as well as RAIDX1, we used the same 7 drives that were described in section 3.1. All the drives were magnetic platter disks. The speeds of the disks ranged from 10.7MB/s to 21MB/s for reads and 2.2MB/s to 21.6MB/s for writes. The average speed was 16.6MB/s for reads and 10.25MB/s for writes.

Our experiments had four 148.5GB, one 139.2GB, one 297.5GB and one 74GB drives for a total of 1104GB. When the chunks per bundle equals the number of disks, this forces the total available space to be a multiple of the smallest disk in the array. In our case, that makes seven 74GB drives for a total of 518GB.

Had there been a traditional RAID on this set of drives, over half of the physical storage would be unattainable to the array. Using a chunk size of one, two or three yields a total of 1104GB storage space for this set of drives. Therefore, a RAID1 or RAID5 can be easily placed on these disks and have the total capacity of the array while also being redundant. Using a double disk redundancy technique where there are four chunks per bundle like RAID6 would allow 1076GB of storage space. In RAIDX, a double disk redundancy becomes unnecessary. The reason for this is the possibility of dynamic array resizing discussed in section 3.2.

5 Results

This section describes the results of the experiments run with both RAIDX and traditional RAID. First, an analysis of the base structure of the RAID is examined. Many attributes that apply to traditional RAID also applies to RAIDX. Subsequent sections show how RAIDX compares to traditional RAID.

It is important to examine the trade-offs in chunks per bundle and chunk size selection. We can easily determine that the chunk size has a linear correlation with the amount of memory the lookup table will consume and the allocation time of the lookup table. This is because the smaller the chunk size, the more bundles that fit on a disk. Smaller chunk sizes result in more efficient transfers of small files, but would cause slowdowns in larger files.

To understand the how RAIDX performs, it is important to have a baseline comparison. This test used the Linux multi-disk module to create the RAID1 array with the parameters all set to the default. The RAID1 was constructed to use all the available space on the disks (see Figure 2). The total space available on the RAID was equal to the smallest disk. After the RAID was assembled, we ran the test routine to determine the baseline speed. This scenario would never be realized in a system as it would be too cost prohibitive. The reason for this test was to provide a baseline for what kinds of results we should be expecting with existing methods.

5.1 RAIDX performance

RAIDX also provides a set of write buffers to increase the bandwidth of the disks by keeping the disks continuously active during burst writes. These write buffers have been observed to increase the write intensive workload speeds by up to 40% in our experiments. The concept is that in a hardware implementation, these write buffers would be placed in a non-volatile RAM so even in the case of power loss, the data will have been written to the RAID device and can resume writing when started up again.

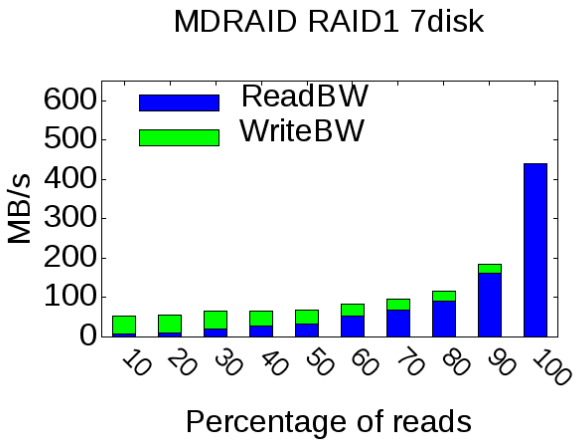

Figure 2: Using traditional RAID1 on a 7 disk array

Without the write buffers, the disk speeds suffer. Larger chunk size arrays have the largest impact. In these cases, what happens is that when a write transaction is taking place, it is blocking any future write transactions. The larger the chunk size, the larger the minimum size that a transaction must be to be split across multiple disks.

With smaller sequential transactions (1KB to 10MB versus a constant 10MB size), it is natural that the throughput shrinks on magnetic disks. Even still, the RAID array performs better than an individual disk due to the parallelism that is available. The transactions are likely to be carried out by only a single disk which leaves the rest of the disks idle.

When the same experiment was run with write buffers, the transaction speeds gained a 30% boost for 512KB chunk sizes, but only a 7.7% gain for 16KB chunk sizes. The buffers allowed more disks to be processing writes at the same time, but since the 16KB chunk size already split the transactions across more disks, it didn't see as much of an improvement. Consider that if a transaction is 64KB and chunk size is 16KB, then the transaction will likely be split across 4 separate disks, whereas if the chunk size was 512KB, then the transaction will fall on a single disk unless it is on a chunk boundary where it will be split across two disks. For this reason is why write buffers help the array with larger chunk sizes.

RAIDX1 on a set of heterogeneous disks is able to store more data than a traditional RAID1 because of the different layout. Traditionally, in RAID1, each two disks form a mirrored set. Thus, if the data being requested is largely in a certain logical location, then

only two drives will have that data. The other disks in the array would be sitting idle. With RAIDX1, there wouldn't be any mirrored sets, as the bundles would be distributed evenly across all of the disks. Therefore, there is greater parallelism in a RAIDX1 set.

We find that the best algorithm is alg4 where we strictly look at the distance between where the head of the disk was last and where the next transaction needs to be.

6 Conclusions

RAIDX, a new type of heterogeneous RAID was developed and tested on a simple striping and mirroring RAID. RAIDX is different in that it uses bundles which are arranged on the disks in a fashion that is determined by the sizes of the disks. While this requires the use of lookup tables to keep track of where the bundles are, it does perform on par with traditional RAID and allows for additional features (such as RAID size extension) that can't be done with traditional RAID. We have shown that write speeds 10 times the speed of an individual drive in the array are attainable and sustainable. While read speeds are not as fast, it is possible to add a cache and prefetch layer to improve it, like it is done on most systems. To also help enhance reads on disks, we looked into how RAID1 will perform on a simplified implementation treating unequal sized disks as equal, but different speeds. We then took this and created a RAIDX1 implementation using a subset of these algorithms and compared it to the traditional RAID1.

In this work, the main concentration was to ensure fast writes to an array of heterogeneous disks. In the current implementation of the algorithm, when several bundles are requested, each chunk is requested on the disk individually. Combining physically sequential chunk requests to disks have been shown to give significant improvements in our simplified tests.

In the future we will also consider RAIDX5, with the added advantage, that it may be possible to use the parity blocks on faster disks rather than data blocks to optimize the throughput. For example, given a bundles with chunks on various speed disks, it may be better to retrieve only those chunks that are on the faster disks and calculate the parity than to always retrieve the data blocks.

References

[1] Jens Axboe. fio - Flexible I/O Tester Synthetic Benchmark.

[2] Deepavali Bhagwat. A Practical Implementation of Clustered Fault Tolerant Write Acceleration in a Virtualized Environment. *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, pages 287–300, 2015.

[3] Andras Fekete. BUSE-CPP.

[4] IDEMA. The Advent of Advanced Format, 2013.

[5] Chao Jin, Hong Jiang, Dan Feng, and Lei Tian. P-Code: A new RAID-6 code with optimal properties. *... of the 23rd international conference on ...*, 2009.

[6] Hyun Ku Lee. Minimizing Consistency-Control Overhead with Rollback-Recovery for Storage Class Memory. 2015.

[7] Xianghong Luo and Jiwu Shu. Generalized X-code. *ACM Transactions on Storage*, 8(3):1–16, 9 2012.

[8] David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID), 1988.

[9] Steven Schlosser, Jiri Schindler, Stratos Papadomanolakis, Minglong Shao, Anastassia Ailamaki, Christos Faloutsos, and Gregory Ganger. On multidimensional data and modern disks. *on File and Storage*, pages 225–238, 2005.

[10] Margo Seltzer, Peter Chen, and John Ousterhout. Disk scheduling revisited, 1990.

[11] Alexander Thomasian. Disk arrays with multiple RAID levels. *ACM SIGARCH Computer Architecture News*, 41(5):6–24, 2014.

[12] Laurent Vanel, Ronald Van Der Knaap, Dugald Foreman, Keigo Matsubara, and Antony Steel. sg245432-AIX Logical Volume Manager from A to Z-Introduction and Concepts-20130226.pdf. M.

[13] Ping Xie, Jianzhong Huang, Qiang Cao, Xiao Qin, and Changsheng Xie. A New Non-MDS RAID-6 Code to Support Fast Reconstruction and Balanced I/Os. *Computer Journal*, 58:1811–1825, 2015.

[14] Ping Xie, Jianzhong Huang, Qiang Cao, and Changsheng Xie. Balanced P-Code: A RAID-6 Code to Support Highly Balanced I/Os for Disk Arrays. *Networking, Architecture, and ...*, pages 133–137, 8 2014.