RAIDX: RAID without striping

András Fekete
University of New Hampshire
afekete@wildcats.unh.edu

Elizabeth Varki
University of New Hampshire
varki@cs.unh.edu

Abstract*—Each disk of a RAID is logically divided into stripe units, and stripe units at the same location on each disk form a stripe. Thus, RAID striping forces homogeneity on its disks. Now, consider a heterogeneous array of hard disks, solid state disks, and RAM storage devices, with various access speeds and sizes. If this array is organized as a RAID system, then larger disks have wasted space and faster disks are under utilized. This paper proposes RAIDX, a new organization for an heterogeneous array. RAIDX disks are divided into chunks; larger disks have more chunks. Chunks from one or more disks are grouped into bundles, and RAIDX bundles data across its disks. The heterogeneity of disks causes unbalanced load distribution with some under-utilized disks and some bottleneck disks. To balance load across disks, RAIDX unbalances its workload. RAIDX moves most frequently access chunks to under-utilized, faster disks and least frequently used chunks to larger, slower disks. Chunk remapping is done at the RAIDX level and does not impact file system to to storage addressing. Experiments show that RAIDX performs more than 2.5x in a heterogeneous configuration after remapping when compared to MD-RAID. RAIDX also shows 1.6x speed improvements when trace files were executed across a network compared to local MD-RAID. We also show a 30% speed boost when the RAIDX array is accessed across a network. Using a large array with 7 disks of assorted speeds, we also are able to show a 7.7x improvement in bandwidth.

I. Introduction

RAID technology transforms an array of small, cheap disks into a storage system with large capacity, low failure rate, and high performance. The acronym, RAID, stands for Redundant Array of Independent/Inexpensive Disks, but it could also stand for Redundant Array of Identical Disks since RAID configuration assumes that the disks have identical capacity and speed. The performance of a parallel system is dependent on the slowest component, so disk homogeneity allows for the superior performance of RAID. However, the homogeneous disk constraint limits the applicability of RAID. With the establishment of Solid State Disks (SSDs), it would be useful to put together a RAID consisting of expensive, fast SSDs and the cheaper, slower disks. Moreover, when an old failed disk is replaced by a larger disk, it would be advantageous to incorporate the greater capacity and speed of the new disk. This paper addresses the problem of integrating heterogeneous disks into a RAID array.

In RAID, storage data are striped (distributed row-wise) across all the disks in the array. A stripe unit is the smallest block of addressable data. A stripe refers to all the stripe units at the same location on each disk. When a file is stored sequentially, the file's data are written on consecutive stripes spanning all disks. There are several RAID levels, which differ in how data and redundancy are striped. Two common RAID levels are RAID5 and RAID10. In RAID5, every stripe has a parity stripe unit; in case a stripe unit is corrupted, the corresponding parity stripe unit is used to regenerate the data. In RAID10, data are striped across all the disks, and every disk has a copy to protect against single disk failure.

A common feature in all RAID levels is that stripe units from a stripe are at identical locations on each disk. This feature facilitates quick mapping from logical to physical space and ensures load balancing across the disks. However, when a disk in an array is replaced by a larger disk, the extra space in the new disk is wasted since the number of stripe units on this new disk cannot exceed the number of stripe units on the smallest disk in the set. Another disadvantage of RAID striping is that faster disks in the array are under-utilized, so it is wasteful to add fast disks to a RAID array. Thus, RAID striping is inefficient for an array of heterogeneous disks.

This paper proposes RAIDX, a new RAID technology for disks with varying capacities and speeds. A design objective of RAIDX is to ensure that capacity of larger disks be included in the array. RAID striping does not satisfy this objective since each stripe includes a stripe unit from every disk in the array; thus, every disk has the same number of stripe units. Another design objective of RAIDX is to ensure that slower disks are not the bottleneck. RAID striping does not satisfy this objective either since stripes are distributed across every disk and each disk has the same number of stripe units. Consequently, faster disks would be under-utilized and slow disks would become bottlenecks. RAIDX configuration must allow larger disks to store more data while ensuring that the speed of the array approaches the speed of faster disks. Striping works when disks are homogeneous, so RAIDX does not stripe.

RAIDX is fundamentally different from RAID: RAIDX has no stripe units, and it does not stripe data across its disks. RAIDX bundles data across its disks. A bundle need not span all disks of the array. Bundles are composed of storage blocks called chunks, with at most one chunk per disk. Chunks from a bundle need not be at identical locations on each disk. The assignment of chunks to bundles may be dynamically remapped. This feature of RAIDX is used to balance the inherent imbalance of RAIDX disks. When a disk becomes a bottleneck, most frequently used chunks from the disk are dynamically remapped to an under-utilized, faster disk. The remapping of bundles does not impact the file system to storage system address mapping. As such, RAIDX bundling is designed for heterogeneous disks.

The contribution of this paper is the development of

RAIDX, a new organization designed for dissimilar disks. The bundles in RAIDX efficiently incorporate larger disks and faster disks. Prior papers on heterogeneous arrays [1] used striping; we believe that we are the first to develop a non-striping organization for heterogeneous arrays. Our experimental evaluation shows that RAIDX outperforms MD-RAID in a heterogeneous disk environment and performs on par in with homogeneous disks. The overhead of RAIDX is minimal, and is eclipsed by the benefit of a dynamic RAID where combining different storage device types is a simple possibility.

The rest of the paper is organized as follows: Section II presents RAIDX architecture. Section IV explains the algorithm for relocation of chunks. Section V-D discusses our tools for testing heterogeneous arrays. Section V-E presents our experimental results. Conclusions and plans for future work are presented in Section VII.

II. RAIDX Architecture

RAIDX disks are divided into equal sized **chunks** - the smallest physical storage unit. A RAIDX chunk is equivalent to a RAID stripe unit. The number of chunks on a disk depends on its capacity. Figure 1 shows a RAIDX array with 4 disks of unequal sizes; disk 1 has 7 chunks, disks 2 and 4 have 4 chunks each, and disk 3 has 5 chunks. Chunks are grouped into **bundles**. Each bundle consists of at most 1 chunk from each disk of the array, and bundles of an array have the same number of chunks. In Figure 1, each bundle consists of 3 chunks; bundle 1 consists of the first chunk (labeled B1 : B1-1, B1-2, B1-3) from disk 1 (B1-1), disk 3 (B1-2), disk 2 (B1-3); bundle 2 consists of chunks (labeled B2) from disk 1 (B2-1), disk 3 (B2-2), and disk 4 (B2-3), and so on. The label Bx-y represents the y^{th} chunk from bundle number x. A bundle is equivalent to a stripe, but bundles and stripes are fundamentally different. The number of chunks in a bundle may be less than the number of disks in the array, chunks of a bundle need not be at identical locations on each disk, and two bundles may contain chunks located on different set of disks within the array. Figure 2 shows a RAIDX array with 2 chunks per bundle.

RAIDX allows two types of redundancy, namely, **parity** and **copy**. In RAIDX-parity, each bundle contains a parity chunk; thus, RAIDX-parity is similar to RAID5. Assume that the RAIDX array of Figure 1 is RAIDX-parity with 3 chunks in a bundle: the third chunk of each bundle could be the parity chunk. In RAIDX-copy, each bundle consists of a chunk and its mirror; thus, RAIDX-copy is similar to RAID10. Assume that the RAIDX array of Figure 2 is RAIDX-copy: each bundle consists of 2 identical chunks - a chunk and its mirror.

For a RAIDX array, two input parameters are: 1) the size of a chunk and 2) the number of chunks in a bundle. The factors that determine the size of a stripe unit are relevant to determining the size of a chunk. A chunk size could be 4KB, 8KB, 16KB, 64KB, 256KB, or greater (similar to selection of stripe unit sizes).

The number of chunks in a bundle, **bundle length** L, could be a minimum of 1 and a maximum of D, the number of disks in the array. The allocation of chunks to bundles has more options when the value of L is small. The smallest L value depends on the redundancy level: for RAIDX-parity, L=3; for RAIDX-copy, L=2; for RAIDX-zero, L=1. There are three cases: L=1, L=D, $1 < L < D$. Below, we discuss the three cases.

The special case is 1 chunk per bundle (L=1). Here, the number of bundles on each disk is equal to the number of chunks on the disk (given by the disk's size divided by chunk size). This level of RAIDX has maximum capacity and no redundancy; we call it RAIDX-zero after RAID0 (striping, no redundancy). The number of bundles, B, in the array is equal to the sum of number of chunks on all the disks. The bundles may be numbered as follows: bundle 1 (B1-1) consists of first chunk on disk 1, B2-1 consists of first chunk on disk 2, B3-1 consists of first chunk on disk 3, and so on.

The next case is D chunks per bundle (L=D) where D is the number of disks in the array. Here, the number of bundles in the array, B, is equal to the number of chunks on the smallest disk. Each bundle consists of 1 chunk from each disk. If each bundle consists of chunks at an identical location on each disk, then RAIDX is similar to RAID.

The final but most common case is $1 < L < D$: when the bundle length is greater than 1 but less than the number of disks in the array. There may be different strategies for assigning chunks to bundles. A simple allocation strategy that maximizes utilization of all disks is as follows: calculate the maximum number of chunks for each disk (disk size divided by chunk size minus the number of chunks the lookup table occupies). Initially, all chunks are not assigned to any bundle. Bundle to chunk assignment: B1 is assigned chunks from the L disks with greatest number of non-assigned chunks; next B2 is assigned chunks from the L disks with the greatest number of remaining unassigned chunks, and so on, until there is no space for further complete bundle assignment. For example, consider Figure 1 where L=3, and Figure 2 where L=2. In Figure 1: B1 consists of chunks from the three disks (D1, D2, D3) with maximum unassigned chunks; B2 consists of chunks from the three disks (D1, D3, D4) with the maximum remaining unassigned chunks, and so on. In Figure 2: B1 consists of chunks from the disks with the maximum number of unassigned chunks (D1, D3); B2 consists of chunks from the disks with the maximum number of remaining unassigned chunks (D1, D2), and so on.

The number of bundles in an array, B, is inversely proportional to the bundle length L. In Figure 1, B=6; in Figure 2, B=10. Intuitively, the degree of parallelism seems directly proportional to the bundle length, and L=D results in maximum parallelism. Reconsider Figures 1 and 2: L=4 with chunks bundled across all disks has parallelism of 4. However, L=2, can also result in maximum parallelism depending on how bundles are assigned to chunks. For example, suppose B1 is assigned to chunks from D1, D2, and B2 is assigned to chunks from D3, D4, and assignment continues in this manner. This

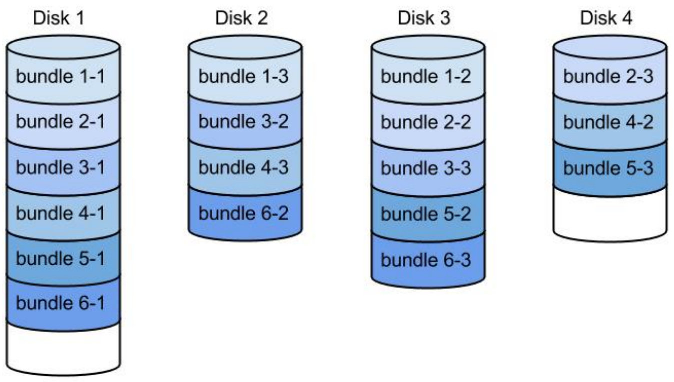

Fig. 1: Sample allocation of a 3 chunk per bundle array

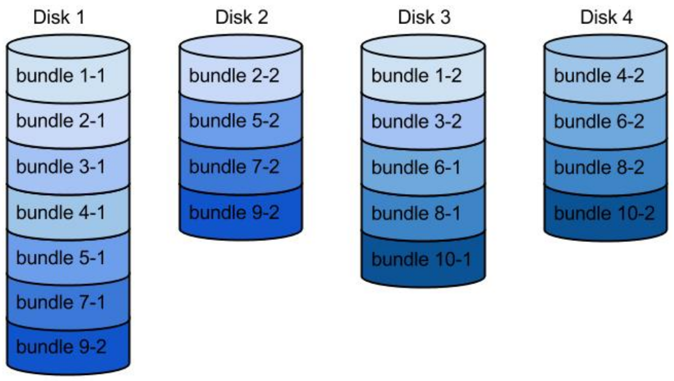

Fig. 2: Sample allocation of 2 chunk per bundle array

degree of parallelism is determined both by the bundle length and the bundle allocation policy.

The redundancy level of RAIDX-copy depends on the bundle length L. L=2 gives a single disk redundancy whereas L=3 gives dual disk redundancy, and so on. For RAIDX-parity, the larger the bundle length L, the lesser the disk space wasted by parity and the longer the rebuild process upon single disk failure. This can be illustrated by 10 disks of identical size. If L=3 (the minimum necessary for RAIDX-parity), then 1/3rd of the total storage space is used for parity but only L-1=2 disks need to be read during the rebuild process. Now consider L=10: only 1/10th of the storage space is used by parity, but L-1=9 disks have to be read during the rebuild process.

III. RAIDX LOOKUP TABLE

In RAID, stripe numbers are assigned to stripe units implicitly: stripe 1 consists of the first stripe unit from each disk, stripe 2 consists of the second stripe unit from each disk, and stripe i consists of the i^{th} stripe unit from each disk. In RAIDX, bundle numbers are assigned to chunks by a bundle allocation strategy, so a data structure that explicitly maps bundle numbers to chunks is required. The chunks in bundle i (Bi) are found by using a lookup table that maps bundle numbers to disk numbers and to the chunk number within the disk.

Each disk has an associated array that maps chunk numbers to bundle numbers. In Figure 1, disk 1's lookup array, D1-lookup is as follows: c[1]=1, c[2]=2, c[3]=3, c[4]=4, c[5]=5, c[6]=6, c[7]=0. The last chunk of D1 is not assigned to any bundle, so c[7] is set to 0. Disk 2's lookup array, D2-lookup, is as follows: c[1]=1, c[2]=3, c[3]=4, c[4]=6. Disk 3's lookup array, D3-lookup, is as follows: c[1]=1, c[2]=2, c[3]=3, c[4]=5, C[5]=6. Disk 4's lookup array, D4-lookup, is as follows: c[1]=2, c[2]=4, c[3]=5, c[4]=0.

At boot time, a bundle-lookup table (or RAIDX-lookup table) is constructed from the disk-lookup arrays; a RAIDX-lookup maps bundle numbers to chunk addresses. The RAIDX-lookup associated with Figure 1 is explained here. Bundle 1 is mapped to the first chunk in disks 1, 3, 2, so it may be written as: b[1]=[(1,1),(3,1),(2,1)] where (1,1) refers to disk 1, chunk 1; (2,1) refers to disk 2, chunk 1; (3,1) refers to disk 3 chunk 1. Similarly, for bundle 2, b[2]=[(1,2),(3,2),(4,1)], bundle 3, b[3]=[(1,3),(2,1),(3,3)], and so on.

Every request arriving at a RAIDX array is mapped to the correct disk location via the lookup table. Mapping request addresses to disk locations can be done directly in RAID; address mapping in RAIDX has a lookup overhead.

IV. PERFORMANCE TUNER

A RAIDX array is inherently unbalanced; the disks differ in sizes and speeds. In RAIDX, data are bundled across the heterogeneous disks of the array. A request that is bundled across several disks is completed only when all its chunks are read/written. A goal of RAIDX is to maximize performance by balancing load across the disks.

The objective is to minimize the average response time of a RAIDX array. A strategy to lower response time of a multiple server system is the following: an arriving job should be directed to a faster server if the system is idle; on the other hand, an arriving job should be directed to an idle slower server if the faster servers are busy. The problem with storage is that requests can only be serviced from the disk on which the data resides. A larger disk has more chunks and it is natural that it will get a proportionally larger number of requests. Similarly, slower disks will have more outstanding requests waiting in the queue than faster disks.

The inherent variance in sizes and speeds of RAIDX disks would result in unbalanced load with some disks being under utilized while other disks are bottlenecks. The performance of RAIDX would be closer to the performance of the bottleneck disks. To prevent disk bottlenecks, the workload to the disks must be unbalanced too - faster disks should get proportionally more read/write requests. In RAIDX, we balance disk load by unbalancing the RAIDX workload. Next, we explain how RAIDX unbalances the workload.

Several papers [2]–[4] have shown that only a small percentage of storage space is accessed. Miranda and Cortes [3] showed that in various popular traces, 90% of I/O requests access less than 40% of the storage device. For some traces, less than 0.05% of the storage device is accessed. EMC states that up to 50% of allocated storage is unused [5]. This implies that only a small percentage of storage data are accessed frequently. If frequently accessed data are moved to under utilized disks, then the performance would improve.

It should be noted that under utilized disks are not necessarily the fastest disks. If all frequently accessed data are moved to the faster disks, then they would become the bottleneck when arrival rates increase. The workload is dynamic - the arrival rates change during the day and the data access patterns change during the day. The goal is to ensure that load is balanced dynamically across disks. The RAIDX performance tuner must address three issues: 1) tracking disk utilization, 2) shuffling (moving) storage data between disks, and 3) tracking frequently accessed chunks.

A. Disk utilization

Disk utilization is a measure of a disk's busy time - the proportion of busy time over total running time. If all the disks of RAIDX contained identical data, then an arriving request should be directed to the disk with no outstanding requests or an SSD with a short queue. (The SSD is an order of magnitude faster than the HDD, so an SSD with a short queue of outstanding requests may service an arriving request faster than an idle HDD.) To balance load, we require a measure of the load at each disk when a request arrives at the array. We estimate arrival instant disk utilization from arrival instant queue lengths. Each time a request is submitted to RAIDX, we record the number of outstanding requests at each disk and compute a moving average of arrival instant queue lengths.

The disk with the greatest queue length is a bottleneck and frequently accessed chunks from this disk should be moved to disks with smaller queue lengths. If the average arrival instant queue length of all disks is close to 0, then it is best to move frequently accessed data to the fastest disk; if not, it is best to move frequently accessed data to under utilized disks.

It may be that the arrival queue length is equal among all disks. This happens when requests are small and don't require the use of all the disks. In this case, we use the estimated disk speeds by observing the size of the request divided by the amount of time the request takes to finish. We look at reads and writes separately as SSDs have a much faster read, but may not exhibit a significantly faster write because of the large memory caches placed on HDDs.

B. Shuffling chunks between disks

In RAIDX, data are bundled across disks. Each bundle spans L disks of the array. It is possible that an entire bundle is frequently accessed; it is also possible that a chunk in a bundle is frequently accessed. When all the chunks of a bundle are frequently accessed, then the entire bundle must be moved to under-utilized disks. When a chunk is frequently accessed, only the chunk in the bundle needs to be moved.

In RAID, storage blocks cannot be moved. It must be handled at the file system level. Assume that this is not the case. If data from one stripe is moved to another stripe, then the new data address must be transmitted upward to the file system, which updates the logical block numbers. Without this update, the requests from file system would map to incorrect disk blocks. Therefore, in RAID, data may not be moved from one stripe to another stripe by the RAID controller. (Note that a disk may remap a physical blocks to another disk location, since this movement does not affect the logical block numbering.)

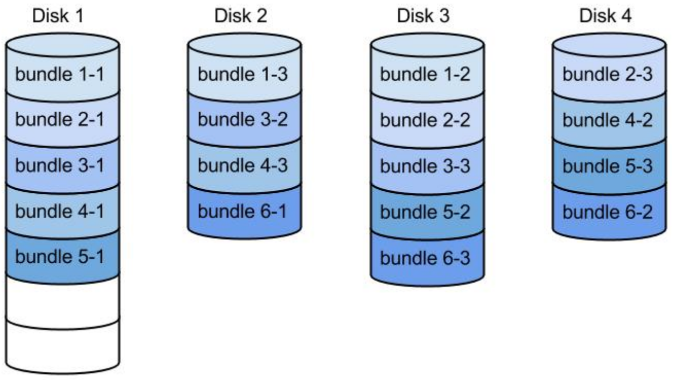

Fig. 3: Example movement of chunks on the 3 disk array

In RAIDX, however, it is possible to move storage data without updating logical block numbers. RAIDX moves data by moving the corresponding "chunks" and "bundles" where the data are stored. When chunks and bundles are moved, the only data structure that has to be updated is the RAIDX-lookup table (and the associated disk-lookup arrays). Changing the address of chunks and bundles does not impact the file system's logical block numbers since the file system maps to the same bundle.

For example, Figure 3 is the RAIDX array of Figure 1 with chunk B6-2 of bundle 6 (b[6]=[(1,6), (2,4),(3,5)]) moved to a faster disk 4. This change would be updated in the RAIDX lookup table as: b[6]=[(1,6),(4,4),(3,5)]. Suppose disk 1 is the bottleneck and chunk B6-1 is frequently accessed; if disk 2 is under utilized, the chunk B6-1 on disk1 may be moved onto disk 2, so that b[6]=[(2,4),(4,4),(3,5)]. The corresponding changes are also made in the disk-lookup arrays.

Thus, RAIDX moves data between disks by shuffling chunk addresses. The shuffling of chunks and bundles is an atomic (all-or-nothing) operation. Initially, all disks in RAIDX will have free (unallocated) chunks used as temporary chunks for shuffling. A chunk may only be moved to an unused chunk. This ensures atomicity as the chunk is not considered moved until the lookup table is updated. The shuffling occurs during RAIDX's idle time. If a request arrives during shuffling, the shuffling stops and the request is serviced.

C. Tracking chunk access frequency

The objective is to balance the imbalance of RAIDX hardware. We achieve this by unbalancing the workload so that more requests are serviced by faster disks. To do so, frequently accessed chunks must be moved to faster disks, and less frequently accessed chunks must be moved to larger, slower disks. Thus, RAIDX must track the rates at which chunks are accessed.

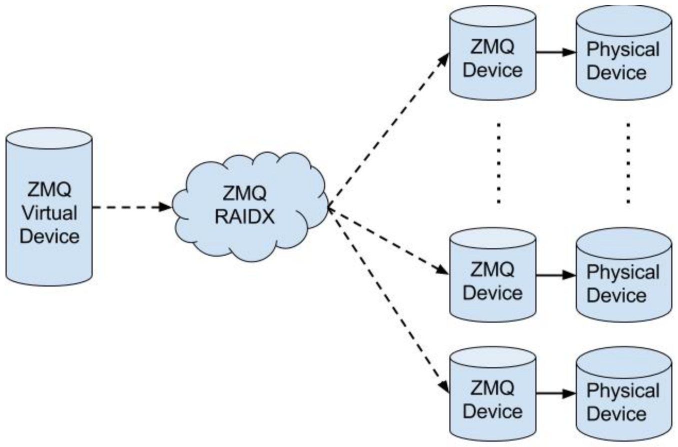

Fig. 4: ZeroMQ RAIDX architecture: dashed arrows show connections that can be either local or networked, solid arrows are local connections

RAIDX uses the frequency of access of a chunk to determine the chunk's placement on disk. This is similar to how cache replacement algorithms determine what data should remain in cache and what data should be evicted. Cache replacement policies such as LRU (Least Recently Used) and LFU (Least Frequently Used) evict blocks from the cache. RAIDX uses similar algorithms to determine chunk placement on disk.

RAIDX tracks the Most Frequently Used (MFU) chunks of the array. When there is a lull in array traffic, RAIDX starts chunk shuffling by moving the most frequently accessed chunk from a bottleneck disk to an under-utilized faster disk. If the fast disk is full, then the least frequently used chunk on the fast disk is swapped out. During chunk shuffling, the algorithm must ensure that there is at most 1 chunk from a bundle on each disk.

V. Experimental evaluation

Our implementation of RAIDX, shown in Figure 4, uses the ZeroMQ framework [6], a high speed bus/network transmission protocol. ZeroMQ reduces the number of memory copies, thereby lowering transmission overhead. ZeroMQ has been successfully used for forecasting stock prices [7] and distributed data storage systems [8]. In Figure 4, ZMQ-Device is our application to connect a raw block device. We created ZMQ-RAIDX to combine multiple ZMQ-Devices together to form a RAID. Our ZMQ-ToNBD application creates a virtual device that can be mounted on a Linux filesystem. For testing bandwidth, we connected a ZMQ Device to a ZMQ Virtual Device directly; Table I presents our results. When testing the ZMQRAIDX framework, using a single device we found that in the worst case there was a 76% utilization of a gigabit network using a RAM drive.

Each of the ZMQ devices connects to a disk. The ZMQ RAIDX stores the RAIDX-lookup table and the performance tuning data structures. We use MD-RAID, the Linux software RAID, as the RAID platform to compare against.

TABLE I: ZeroMQ bandwidths

Connection	Buffered	Unbuffered
IPC	65.56MB/s	60.03MB/s
Net-Local	64.56MB/s	60.15MB/s
Net-Remote	57.17MB/s	52.82MB/s

A. Performance tuner implementation

The data structures and outline of the performance tuner, which balances workload across the unbalanced disks, is presented in the previous section. Here, we present implementation details. For this first version, we implemented an approximate LFU algorithm for estimating the most frequently used chunks and the least frequently used chunks on a disk. There are two separate lookup tables in RAIDX: one on disk and one in RAM. Both tables are relatively small compared to the size of the storage. The performance tuner is only called when the system is idle. To help the system get up and running, it will create additional reads to better estimate disk speeds. Note, this only happens if the system is idle. In our experiments, we find that a more accurate the measurement of the disk speeds gives a significant improvement to initial movement of chunks. This eliminates many of the chunk moves that may have been moved to disks that had a performance boost due to caching or moved from disks that had a performance loss due to disk spinup.

The performance tuner periodically runs a disk scrub, since deterministic disk scrubbing is the most efficient and reliable method to avoid unrecoverable sector errors [9]. In addition to disk scrubbing, the performance tuner flushes dirty caches to disk.

Most of the workload in the performance tuner is in the movement of chunks based on average access interval. Each section of the performance tuner is timed, and if it has been running for more than 50 milliseconds, we check to make sure there aren't any outstanding system requests for data. If there is an outstanding request, then it is executed and the performance tuner is called 1 second after the last request has completed. Otherwise if there are no outstanding requests, the performance tuner is called again.

The disks are grouped by speed and average queue length into disk groups. Each disk group contains L disks. The shuffling algorithm, which moves chunks between disks, is activated during lull periods. The shuffling algorithm pops an arbitrary element from the MFU set. It recalculates what the average access interval would be if it were accessed. If this interval is appropriate for the disk group that it is in, then the next element on the MFU set is popped. When a MFU element is in a slower disk group, then it is moved onto the appropriate faster disk group. If there is no free chunk in a faster disk group, then the least frequently accessed chunk is selected from a subset of all chunks on a disk in the group to be moved to a slower disk group. There is no need to find the absolute least frequently accessed chunk because it may vary as the system is being used. An approximate chunk selection yields a fast routine while the chunk layout approaches the optimal layout.

B. Lookup table implementation

RAIDX has the stripes dynamically assigned, and those locations need to be stored, thus compared to traditional RAID, RAIDX requires a lookup table. A concern may be that this gives unpredictability in the system requirements for the array. However, the memory required to store the table is a simple equation:

$$memoryRequired = \frac{15 * totalPhysicalStorage}{chunkSize}$$

Chunks need to store 4 things: 1) the disk it is on; 2) the offset it's located at on the disk; 3) the last access time; and 4) the average access interval. The number 15 is the size of a chunk data structure. Each chunk entry comprises of an 8byte chunk offset on the disk, 4byte previous access time, 2byte average access interval, and 1byte disk ID. A 64bit chunk offset will be sufficient to support a 2048 petabyte array. The previous access time is stored as a C++ std::chrono::timepoint using a 32bit container, which is sufficient to store time at a resolution of minutes. The average access interval is a simple 16 bit integer which stores the average time between updates to the last access time. The larger the interval, the less often the chunk is accessed. When the average access interval changes by a significant amount, the chunk ID is placed in an unsorted set so that during the next cycle of the performance tuner will move the chunk to a faster disk. At boot time, this unsorted set will be empty. As chunks are accessed, the set is updated.

Upon initialization for each chunk entry, the offset field is populated based on the location of the chunk and the last access time is set to the current time. The other two fields are left as the maximum value for the field.

The selection of a chunk size is similar to that of traditional RAID. The smaller chunk size creates more chunks which is better for systems with smaller files but slower for larger files. Larger chunk sizes also require less RAM to store the lookup table. The smallest chunk size is 4KB. Many filesystems use this as the minimum as well because the sector size of a drive has been set at 4KB since 2011 [10]. In the worst case with 4KB chunk sizes, using the equation above, the largest lookup table will be only 0.36% of the total physical storage space.

The lookup table will be stored on the physical storage and loaded in to RAM upon startup. Each drive stores only the information about those chunks that are stored on the device. There is no advantage to the knowledge of chunks on other disks. In the event of a drive failure, or disks missing on initialization, the knowledge of which disks had a particular chunk offer no utility in reconstructing the array. The lookup table stored in RAM will have the chunks and bundles in an array so lookup is an O(1) operation. Once the array is assembled in RAM, there is no addition or removal of bundles in the course of normal operation.

We store an array of free chunk locations for each disk. This helps determine which disks can store chunks. The system overall will only have as many free chunks as there are disks in the system. Over time, the free chunks tend to be on the slowest disks in the array. Since chunks are only a couple kilobytes in size, this is also not a major waste of space.

C. Disk caching

Using ZeroMQ allows the connection to be across a network that is transparent to the running program. The remote address gives a clue as to the type of connection to be used. We use inter-process and network connections. When a ZMQ-Device is across a network, we needed a cache to reduce the amount of data traveling over the ethernet. For this, we implemented a simple LRU cache mechanism. When there's a cache miss for a particular chunk, the least recently used chunk is flushed and is replaced with the missing chunk.

All the previous knowledge about LRU caching applies and is very much load dependent whether the cache enhances performance or detracts from it. In the case of a load with random unique requests, the cache will be a hindrance. However, as discussed in prior sections, it has been shown that most disk accesses occur to a small subsection of the physical storage space. Therefore, having a cache is likely to improve the system speed.

D. Device Testing

The goal of our experimental evaluation is to compare the performance of RAIDX against the performance of RAID. In our first experiment, we use two Samsung 850 EVO 250GB SSD homogeneous disks, and compare RAID against RAIDX. In this experiment, we do not perform performance tuning (shuffling of chunks) since the disks are homogeneous. The goal of this experiment is to estimate the overhead of lookup tables in RAIDX. We expect RAID to perform better than RAIDX.

The workloads used for our experimental evaluations are financial and websearch trace files [11]–[14]. In order to understand the workloads, we generated statistics shown in table II. We also looked at the sizes of the transactions in the traces (see figure 5). This shows what kind of performance can be expected with varying chunk sizes. On average, the transactions were 5-30Kib in size. The WebSearch traces showed an interesting pattern to the reads. There were some outliers in each of the plots which we omitted for the sake of clarity in the figures.

To replay these trace files, we created our own replayer [15]. This collects the statistics of the trace as well as can replay the trace as it was recorded or at maximum speed where the delay between transactions is ignored. We use the maximum speed to see what the bandwidth of the device is. The replayer can also run a limited range of transactions to be able to add custom delays. Since RAIDX has a performance tuning that only runs at idle times, the intention was that custom delays would be added, but we found no difference between custom delays versus rerunning the same trace file multiple times.

In addition to trace files, we developed diskSpotcheck [16], a tool that generates random I/O on the disks; a random workload allows us to evaluate RAID and RAIDX without

TABLE II: Statistics of SPC trace files

Trace Name	Run time	Read %	Transactions	TX/sec	Read size	Write size	Min. disk size	ASUs
Financial1	12h 8m	23.16	5334987	122.05	2715.67MB	14918.69MB	29.319GB	23
Financial2	11h 23m	82.345	3699195	90.24	6778.685MB	1860.74MB	0.02GB	18
WebSearch1	0h 52m	99.98	1055448	334.92	15608.88MB	1.78MB	0.163GB	5
WebSearch2	4h 16m	99.978	4579809	297.47	67394.82MB	7.83MB	0.163GB	5

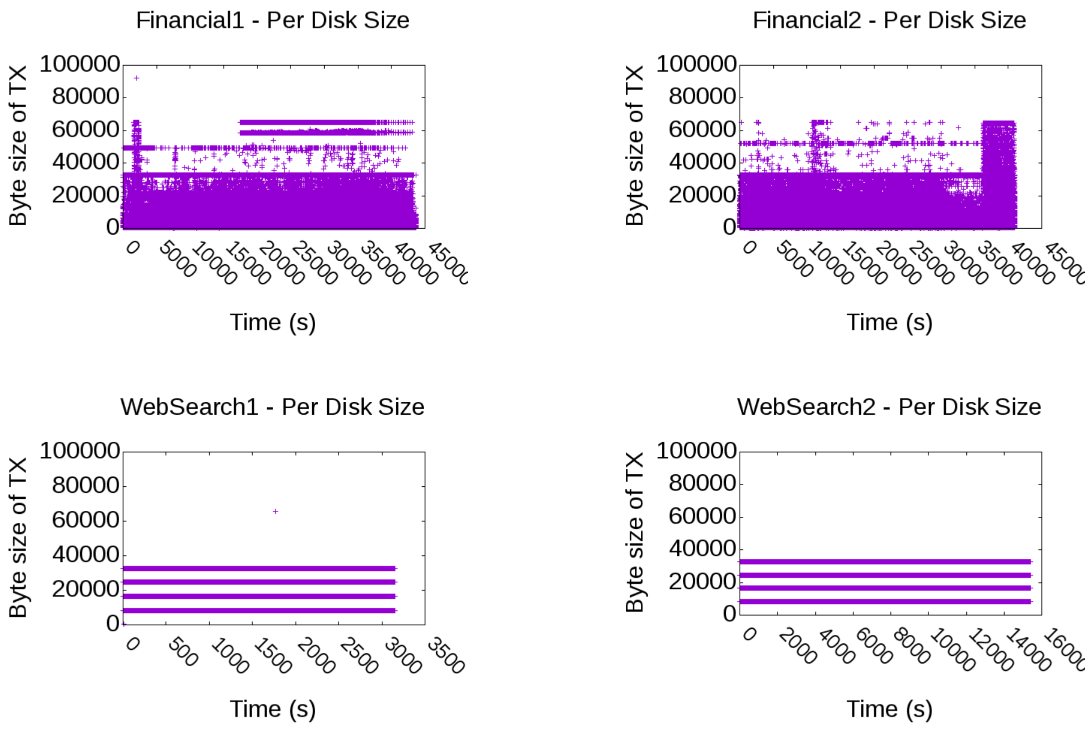

Fig. 5: Size of transactions in SPC trace files

effects of caching. The tool uses a predetermined random seed to generate pseudo-random values for the offsets and data to write. The benefit of using a pseudo-random number generator is to create arbitrarily large transactions that can be verified as correctly written. This tool is device agnostic, meaning that it can be used on any kind of block device. Other tools such as fio [17] are very advanced and can execute a variety of workloads, but it doesn't verify that the contents are correctly written. In particular, we want to ensure that different offsets are not overlapping in our RAIDX.

This program is intended to mimic the access to a multi-user system as each user has their own files which will be in different locations on the disk. DiskSpotcheck also attempts to minimize the utility of caching by purposely making disk transactions that are highly random. This tool also forces the system to drop caches and performs a flush of the buffers after all writes have been issued. The total time to execute all the writes and readback verification is measured. The tests run 3 different times with a different random seed each time. The average of the 3 times is what is used to compare different block devices and configurations.

E. Results

Figure 6 shows the results of our first experiment with similar disks. The goal of this experiment is to evaluate the peformance slowdown caused by lokup tables in RAIDX. Each bar graph plots RAID/RAIDX speedup: the time to run a workload on RAID divided by the time to run the same workload on RAIDX. A speedup of 1 is obtained when RAID and RAIDX have similar times; a value less than 1 is obtained when RAID is faster than RAIDX; a value greater than 1 is obtained when RAID is slower than RAIDX. The performances of RAID and RAIDX are comparable when the disks are identical. Our results show that the overhead of lookup tables is not significant.

Using MD-RAID on a local set of disks, we ran the same experiments using RAIDX on local and remote disks. Remote disks were achieved using ZeroMQ over a gigabit LAN to a separate server using a Dell Vostro 420 system. The system with the disks was a Dell PowerEdge 2900. The cache refers to whether the LRU cache of the ZMQ virtual device was enabled. The speedups in figures 6 and 7 are all in relation to the MD-RAID.

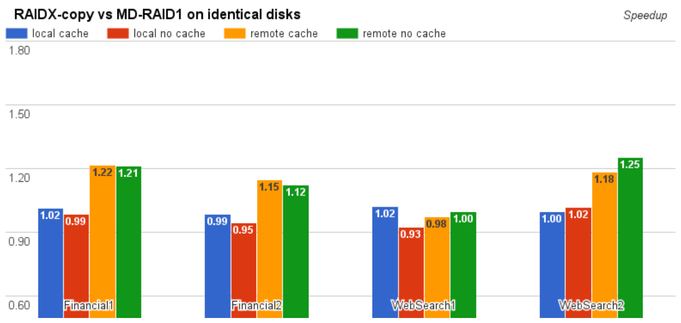

Fig. 6: Speedup comparison between traditional RAID and RAIDX on identical disks

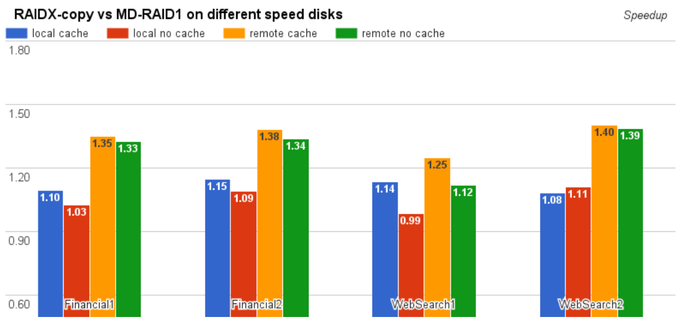

Fig. 7: Speedup comparison between traditional RAID and RAIDX on different speed disks

1) General RAID performance: In figure 6 the experiment was to compare the performance of RAIDX to MD-RAID on a homogeneous system using mirroring. MD-RAID was configured for a two disk mirror using two SSDs. RAIDX was set for RAIDX-copy using the same two SSDs. This experiment gives us the baseline performance of the two types of RAID. In the figure, we see that when the disks are local RAIDX is nearly identical in performance to MD-RAIDX. An unexpected result is that when the disks were accessed remotely, there was a significant speedup in performance. Our hypothesis is that the load of executing the trace file and retrieving the data is akin to having a separate hardware take care of the RAID.

The next experiment involved similar size disks, but different architectures. We used a 300GB HDD and a 250GB SSD. Figure 7 shows the results. Again, RAIDX performed on par with MD-RAID if not a slight bit better. This experiment is concentrating on the queuing systems for deciding which disk in the mirror the transaction should be scheduled onto. MD-RAID just uses the disk that is idle. RAIDX does this as well, but if there are no idle disks, it will queue all the transactions so that as soon as a disk is ready, it will have a task to execute. The key difference is that RAIDX takes into consideration the disk speeds as well as the queue lengths when deciding which mirror to put the transaction on. MD-RAID on the other hand keeps these queue lengths equal. Remote RAID showed an improvement again, but caching actually worsened performance. Perhaps a cache size of 200MB is not large enough to keep the most accessed blocks in the cache.

2) Performance tuner testing: We tested RAIDX-zero to determine the efficacy of the performance tuner algorithm. Figure 8 shows that using the same 250GB SSD and 300GB HDD as before, after about the 2nd iteration of the diskSpotcheck utility, the most frequently used chunks have been optimized across the disks. We were able to achieve over 2.5x speed gains by this kind of reorganization.

Figure 9 shows our performance tuner in a situation involving more than just two disks. We assembled a large array consisting of four 250GB SSDs, an assorted set of HDDs which ranged in size from 74GB to 1TB, and lastly a 600MB RAM disk. The RAM disk was simply a piece of system RAM

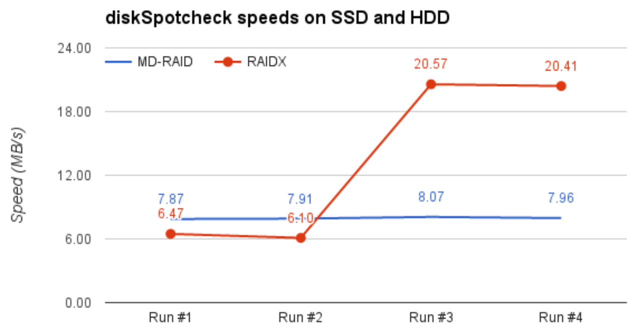

Fig. 8: Comparing MD-RAID to RAIDX-zero

mounted as a block device. We can see that already after the first iteration, the chunks have moved to the optimal location giving a 7.7x improvement in disk performance.

3) Filesystem testing: Using a single server with 7 disks and a RAM drive, we tested the speedup of a RAIDX array. Using buffered I/O, we were able to achieve a significant speedup after the first pass of performance tuning. Subsequent tuning only gave marginal speed improvements. Testing was done on the array by measuring the time it took to create a

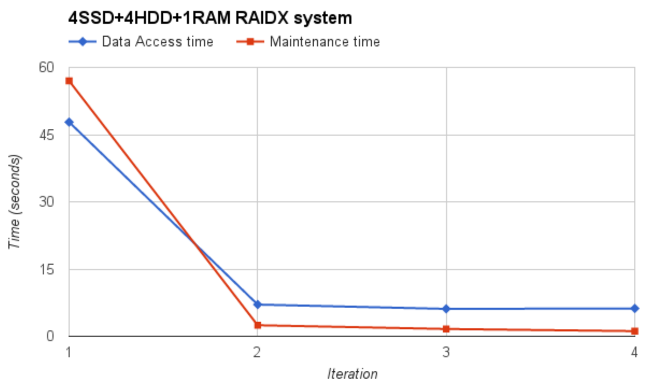

Fig. 9: Speed of data access and performance tuning time of a 7 disk RAIDX array

filesystem and copying 1GB of data to the drive. This took on average 35 seconds. After the tuning completed, the same process took only 21 seconds on average.

To test the read performance, we took the test procedure above and followed it by measuring the time it took to copy from the RAIDX array all the files. The reads took on average 20 seconds to complete.

Even in a HDD-only system we were able to get a speed boost by distributing the frequently accessed chunks on the faster disks. We then recorded the amount of time it took to retrieve 3GB of data on the array before (87 seconds) and after tuning (61 seconds). 30% performance improvement was achieved after a single performance tuning. We find that the larger the diversity in speed, the greater the speedup after tuning.

VI. Related Work

Research into heterogenous RAID has focused on multi-level RAID [18]–[20], RAID reconstruction [11], parity coding [21], [22], and striping on different sized disks [1]. Recently, research has focused on RAID for SSDs [23], [24]. When storage consists of SSDs and HDDs, the RAID is constructed on HDDs and SSDs are treated as caches [25]–[27].

CRAID [28] is an implementation of a RAID array which moves blocks that are more frequently accessed onto a small cache partition on the disks in the array. When a new disk is added to the array, the RAID needs to be reorganized and all accesses are restricted to the cache partition.

Others have used chunk migration strategies to reduce the number of spinning disks in the array to increase energy efficiency. An example is ThinRAID [29], which concentrates the RAID on a subset of disks and then spins down the rest. When there is a heavy load predicted, the chunks are moved to the unused disks to speed up I/O with added parallelism. It is shown to have a 15-27% energy savings. EERAID [30] is another method which spins down one disk and uses redundancy to reproduce it during disk access. In RAID5, this means calculating the parity based on the other blocks. In EERAID, there is up to a 30% energy savings while using mirroring and 11% for a single parity RAID.

Prior work treats SSDs as caches. We are the first to move frequently accessed areas of the array to faster disks without the use of a cache device. Dynamic bundling is a new concept that allows a RAIDX array to match the system hardware with the data access patterns of the device.

This paper extends prior work on heterogeneous disks. Similar to prior work [31], [32], we try to balance the load across the disks in the array. Unlike the prior work which is constrained to a single system using a single program, we split up the system into smaller pieces so that each piece only handles one particular task, but it handles it well. We also improve on this work by introducing the performance tuning algorithm. The work in [32] concentrates on determining the best write scheduling policy for a heterogeneous disk system.

VII. Conclusions

This paper develops RAIDX, a new RAID for heterogeneous arrays of disks with varying speeds, sizes, and technologies. RAIDX is unique in that it does not stripe data across disks; RAIDX bundles data across disks. Bundles are composed of chunks, with at most one chunk per disk. The number of chunks per disk varies according to size. Therefore, RAIDX incorporates the capacities of larger disks.

A novel feature of bundles is that chunks are dynamically assigned to bundles. At boot time, a bundle may be assigned chunks from disks, say 1, 2, 3, 4; at a later time, the same bundle may be assigned chunks from disks, say 2, 4, 5, 6. This dynamic assignment of bundles allows RAIDX to balance load across disks and maximize performance. Frequently accessed chunks are moved to faster disks on-the-fly, thereby ensuring that there are no under-utilized disks nor bottleneck disks.

RAIDX provides two types of redundancy, namely RAIDX-parity, which is similar to RAID5, and RAIDX-copy, which is similar to RAID10. The goals of RAIDX are to provide redundancy and parallelism of traditional RAID, and to provide the access speeds of the faster disks in the array. Our experiments suggest that RAIDX satisfies these goals. RAIDX is a suitable technology for arrays composed of slower hard disks and faster solid state disks.

RAIDX has a performance tuner that tracks chunk access frequency and determines what chunks should be moved, when they should be moved, and where they should be moved. Thus, RAIDX has computational overhead and traffic overhead. The movement of chunks between disks causes traffic at the disks. We plan to study the overhead of RAIDX performance tuner. Our preliminary implementation of the performance tuner uses an approximate LFU algorithm. We plan to analyze alternative algorithms for the performance tuner.

In the future, we hope that we can propose a solution to the write hole problem in RAID by having RAIDX perform an automatic copy on write of a chunk. For this, we anticipate having to create a journaling mechanism to ensure that all writes are correctly performed.

References

[1] T. Cortes and J. Labarta, “A Case for Heterogenenous Disk Arrays,” *Proceedings of the IEEE International Conference on Cluster Computing (Cluster’2000)*, pp. 319–325, 2000.

[2] T. Gibson, E. L. Miller, and D. D. E. Long, “Long-term File Activity and Inter-Reference Patterns University of California,” *International Business*, no. December, pp. 976–987, 1998.

[3] a. Miranda and T. Cortes, “Analyzing Long-Term Access Locality to Find Ways to Improve Distributed Storage Systems,” *Parallel, Distributed and Network-Based Processing (PDP), 2012 20th Euromicro International Conference on*, pp. 544–553, 2012.

[4] D. Roselli, J. Lorch, and T. Anderson, “A Comparison of File System Workloads,” *In Proceedings of the 2000 USENIX Annual Technical Conference*, pp. 41–54, 2000. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.195

[5] EMC, “EMC VNX Virtual Provisioning,” Tech. Rep. December, 2013.

[6] “ZeroMQ,” 2015. [Online]. Available: http://www.zeromq.org

[7] P. Arce, C. Maureira, R. Bonvallet, and C. Fernandez, “Forecasting High Frequency Financial Time Series Using Parallel FFN with CUDA and ZeroMQ,” *Information and Telecommunication Technologies (APSITT), 2012 9th Asia-Pacific Symposium on*, no. February, pp. 1–5, 2012.

[8] P. Fengping and C. Jianzheng, “Distributed System Based on ZeroMQ,” 2012.

[9] I. Iliadis, R. Haas, X. Hu, and E. Eleftheriou, “Disk scrubbing versus intradisk redundancy for RAID storage systems,” *ACM transactions on storage (. . .*, vol. 7, no. 2, pp. 1–42, 7 2011. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1970348.1970350 http://dl.acm.org/citation.cfm?id=1970350

[10] IDEMA, “The Advent of Advanced Format,” 2013. [Online]. Available: http://www.idema.org/?page_id=2369

[11] L. Tian, D. Feng, H. Jiang, and K. Zhou, “PRO : A Popularity-based Multi-threaded Reconstruction Optimization for RAID-Structured Storage Systems,” *Fast07*, pp. 277–290, 2007.

[12] J. Wan, J. Wang, Q. Yang, and C. Xie, “S2-RAID: A new RAID architecture for fast data recovery,” *Mass Storage Systems and . . .*, vol. c, 2010. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=5496980

[13] S. Wu, D. Feng, H. Jiang, B. Mao, L. Zeng, and J. Chen, “JOR: A journal-guided reconstruction optimization for RAID-structured storage systems,” *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS*, pp. 609–616, 2009.

[14] F. Ye, J. Chen, X. Fang, J. Li, and D. Feng, “A Regional Popularity-Aware Cache Replacement Algorithm to Improve the Performance and Lifetime of SSD-based Disk Cache,” pp. 45–53, 2015.

[15] A. Fekete, “UMASS Trace Replayer,” 2016. [Online]. Available: https://github.com/bandi13/UMASS_Trace_Replayer

[16] ——, “diskSpotcheck,” 2016. [Online]. Available: https://github.com/bandi13/diskSpotcheck

[17] J. Axboe, “fio - Flexible I/O Tester Synthetic Benchmark.” [Online]. Available: https://github.com/axboe/fio

[18] J. L. Gonzalez and T. Cortes, “Adaptive Data Block Placement Based on Deterministic Zones (AdaptiveZ),” pp. 1214–1232, 2007.

[19] N. Jeremic, H. Parzyjegla, and G. Mühl, “Improving Random Write Performance in Heterogeneous Erasure-Coded Drive Arrays by Offloading Code Block Requests,” pp. 2007–2014, 2015.

[20] A. Thomasian, “Disk arrays with multiple RAID levels,” *ACM SIGARCH Computer Architecture News*, vol. 41, no. 5, pp. 6–24, 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2641364

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

[22] 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, Architecture, and . . .*, pp. 133–137, 8 2014. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6923172 http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6923172

[23] C. C. Chung and H. H. Hsu, “Partial parity cache and data cache management method to improve the performance of an SSD-based RAID,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 22, no. 7, pp. 1470–1480, 2014.

[24] Y. Li, P. P. C. Lee, and J. C. S. Lui, “Stochastic analysis on RAID reliability for solid-state drives,” *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pp. 71–80, 2013.

[25] S. Im and D. Shin, “Flash-aware RAID techniques for dependable and high-performance flash memory SSD,” *IEEE Transactions on Computers*, vol. 60, no. 1, pp. 80–92, 2011.

[26] N. Jeremic, G. Mühl, A. Busse, and J. Richling, “The pitfalls of deploying solid-state drive RAIDs,” in *Proceedings of the 4th Annual International Conference on Systems and Storage - SYSTOR ’11*. ACM Press, 2011, p. 1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1987816.1987835

[27] J. Ren and Q. Yang, “I-CASH: Intelligently coupled array of SSD and HDD,” *Proceedings - International Symposium on High-Performance Computer Architecture*, vol. 02881, pp. 278–289, 2011.

[28] A. Miranda and T. Cortes, “CRAID: Online RAID Upgrades Using Dynamic Hot Data Reorganization,” *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, pp. 133–146, 2014. [Online]. Available: http://blogs.usenix.org/conference/fast14/technical-sessions/presentation/miranda

[29] J. Wan, X. Qu, N. Zhao, J. Wang, and C. Xie, “ThinRAID: Thinning Down RAID Array for Energy Conservation,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 10, pp. 2903–2915, 2015. [Online]. Available: http://www.computer.org/csdl/trans/td/2015/10/06912991.pdf

[30] D. Li and J. Wang, “EERAID: energy efficient redundant and inexpensive disk array,” *Proceedings of the 11th workshop on ACM SIGOPS European workshop*, vol. 6, p. 29, 2004. [Online]. Available: http://portal.acm.org/citation.cfm?id=1133572.1133577

[31] A. Fekete and E. Varki, “RAID-X : RAID eXtended for Heterogeneous Arrays,” in *30th International Conference on Computers and Their Applications*, 2015, pp. 157–162. [Online]. Available: http://www.bandilabs.com/2015/03/12/cata-2015-presentation-of-raid-x-raid-extended-for-heterogeneous-disks/

[32] ——, “RAID on Heterogeneous Arrays,” 2016.