Cider: A Case for Block Level Variable Redundancy on a Distributed Flash Array

Sharath Chandrashekhara, Madhusudhan R. Kumar*, Mahesh Venkataramaiah*, and Vipin Chaudhary
Department of Computer Science and Engineering, University at Buffalo, SUNY
sc296@buffalo.edu, madhura@microsoft.com*, maheshv@nutanix.com*, vipin@buffalo.edu
**Work done while studying at University at Buffalo, SUNY*

***Abstract*—With the increase in data volumes, it is prudent to classify data depending on its criticality. One might prefer a cheap storage for a year old system logs but a highly fault tolerant storage for personal photos. The existing solutions include storing these two sets of data in two different systems or choosing a system with a fault tolerance level required by the most critical data. This means a higher storage footprint for the lesser critical data. In today's petabyte scale data centers, this will result in a significant increase in the overall storage footprint making the system unattractive. In addition, data storage solutions designed for traditional data centers have to be re-engineered to work at cloud scale. For instance, reliable storage using traditional RAID like systems are inflexible and do not provide a mechanism to easily change the level of redundancy once the system is set up. As a result, changes to the reliability requirements of data over time cannot be serviced efficiently.**

To address these problems, we propose *Cider*, which aims at providing an extremely flexible, reliable, and distributed block store using erasure codes. *Cider* takes a new approach to reduce the storage overhead by offering a variable degree of fault tolerance at a granularity of a single block. We achieve this by using a thin block translation layer and a block level metadata system. *Cider* can be readily used by many high performance and enterprise applications which require a block level interface. Through a case study of a novel flash based clustered storage system, we make a strong case for the adoption of *Cider* in future systems. We discuss various design trade-offs and implementation challenges associated with designing our system. Lastly, we discuss the preliminary implementation of our system and results from our initial experiments.

I. Introduction

With the data volumes growing rapidly, enterprise data is being migrated from traditional data centers to cloud data centers. This is driven by a promise of providing higher reliability at a lower cost by cloud vendors. To fulfill these promises, the cloud vendors have to provide storage services at a lower cost without compromising on the degree of reliability. This is particularly challenging because, often, the cloud data centers have different requirements compared to the traditional data centers. For instance, the storage capacity of cloud service providers is magnitudes higher than the private data centers. Additionally, a single cloud platform is often shared by many clients whose needs are not identical. In order to appeal to a wide range of clients, the service providers have to provide a flexible platform which can cater to the specific needs of each customer. To address these challenges, various traditional computing solutions are redesigned and engineered to fit the cloud. Storage systems are no different and are redesigned to efficiently meet the cloud requirements. Often, cloud specific optimizations are built on top of traditional storage solutions like RAID [1], and sometimes, they are designed from scratch, specifically for the cloud.

Traditionally, storage systems are dominated by RAID based monolithic arrays connected over SAN. These systems not only tolerate single disk failures (two disk failures in case of higher RAIDs) but are also faster than regular disks due to parallel I/Os. While they are suitable in medium and small environments, they have significant limitations in cloud-scale data centers. First, RAID features are mostly implemented in hardware, which is expensive and offers limited flexibility. While the hardware prices have dropped, it has not kept up with the increase in the amount of data. Further, once the system is set up, it is difficult to switch between different RAID levels. Second, after a disk failure in a RAID system, it is critical to replace the failed disk and start a rebuild process immediately. This can take hours to complete, during which the system is either completely down or running in degraded performance mode. While declustered RAID systems offer faster reconstruction times with all nodes participating in the process of reconstruction, these optimizations offer only a marginal improvement in large data centers. Additionally, there is also a possibility of a second disk failure during reconstruction. Such failures are very likely in today's world of multi-terabyte disks [2], [3], [4]. Lastly, a typical RAID based system can protect only against local disk failures which is a severe limitations for cloud scale systems which require the protection to span across multiple servers and sometimes across multiple data centers. There exists many implementations of RAID at the software level and are collectively called soft RAID. While these systems offer better flexibility, they too suffer from many of the limitations associated with the hardware RAID, like high reconstruction times.

These limitations of traditional storage architectures have given rise to a myriad of scale-out solutions. The most popular cloud storage services like Amazon S3 [5] are based on object storage and provides additional reliability guarantees on top of traditional storage. Systems like Swift [6] and Microsoft Azure [7] have been using object replication to support high performance and reliability. But replication remains extremely expensive due to high storage footprint. A popularly used alternative to replication to provide reliable data storage is erasure coding [8], [9]. Many implementations with such a scheme

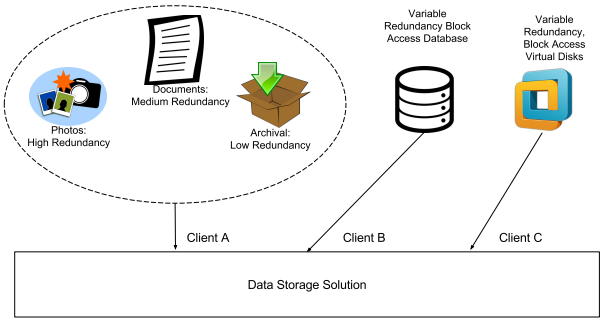

Fig. 1. Heterogeneous Data

are already available for object based stores in the cloud [10], [11], [12]. While these solutions are centered around object storage and are efficient on some of the workloads, they are less than ideal for virtual machine or database workloads that require block level access to data. The virtual disks, which are the building blocks of virtual environments are nothing but large object blobs for an object store. The I/O operations on these objects are mainly random and involve an update to only a small part of the entire object/file. They are also not particularly suitable for traditional applications which are crafted with a POSIX based interface and semantics in mind. Such applications require fine grain control over their data and tend to perform large amounts of small but random I/O. Hence, a much more fine grained solution is needed.

While adapting to the scale of the cloud is one side of the problem, adapting to the nature of data is another. Data by nature is heterogeneous and different types of data requires different degrees of reliability. Figure 1 shows multiple clients with different needs using the same storage solution. While *Client-B and Client-C* require block access and a variable redundancy for their data, *Client-A* has different types of files, each requiring different levels of redundancy. For instance, based on legal requirements, financial data would mandate a high degree of reliability for the first seven years while medical records would be expected to last forever. Additionally, the reliability requirement for a piece of data might change over time. Most of the available storage solutions use a globally fixed redundancy for the data and do not allow this kind of flexibility. Thus, a system that allows changes to the degree of fault tolerance at a finer level is highly desirable.

To address these challenges we present our system *Cider*[1]—a highly flexible and scalable virtual block device with a temporally variable redundancy model. In *Cider*, we perform erasure coding at block level and stripe data blocks across a number of remote disks. *Cider* provides a high degree of flexibility in which the degree of fault tolerance can be set on a per block basis. Such fine control over the degree of redundancy would result in the reduction of storage footprint. We argue that the flexibility and benefits we get from our system far outweigh the small performance penalty paid for the calculation of the erasure codes.

Lastly, recognizing the importance of flash based SSDs in the future of large scale data centers, we make a compelling case for using *Cider*—first, by discussing how *Cider* can be integrated with flash based SSD, and then, through a case study of a large scale clustered flash array. We also analyze and discuss how *Cider* can be easily combined with some of the previously developed techniques such as Blizzard [14] and CAFTL [15] to create an extremely flexible and scalable system.

To demonstrate these concepts and to prove the feasibility of such a system, we have implemented a prototype of *Cider* as block device on Linux. Our current implementation is done at user level using NBD [16] to mock block device driver in user space. We evaluate our prototype for correctness and measure the overhead using micro-benchmarks. Our evaluation shows the practicality of our system.

The remainder of this paper is organized as follows. We introduce erasure coding and flash storage in section II. In section III, we describe the variable redundancy model and in section IV, we introduce a flash SSD based storage array as an example of a system which can benefit from *Cider*. Section V describes our system design and in section VI, we discuss the implementation of our prototype and its evaluation. We then discuss various optimizations and open issues of *Cider* and its adoptions in section VII, related work in section VIII and end with a conclusion in section IX.

II. Background

A. Erasure Coding

Erasure coding is a well known technique of adding encoded redundant information to a piece of data such that the data can be recovered when a part of the original data is lost. We can interpret this as expanding 'k' units of data to 'm' units, where $(m-k)$ units of encoded data is added to 'k' units of original data, enabling reconstruction of the data with a combination of any 'k' units. Throughout the paper, we use the terms 'k' and 'm', where 'k' is the minimum number of blocks required to reconstruct data and 'm' is the total number of data blocks.

B. Flash based SSDs

Recent developments in storage technologies have made flash based SSDs a predominant player in the storage industry and they are predicted to replace the traditional hard drives sooner or later. At the core of an SSD is a NAND flash memory chip used for storing raw data. Each NAND chip is divided into pages of fixed size—typically 4KB or 8KB. Multiple pages are grouped into blocks of a fixed size, typically 128KB or 256KB. One or more NAND chips are present in a die and is managed by an SSD controller. The controller has some computation resources, including a small RAM.

One of the distinguishing hardware characteristics of flash memory is the 'erase before write' architecture. This means a block has to be erased before data can be written into it.

[1]The core of our system uses a variable redundancy model as described in our patent [13]

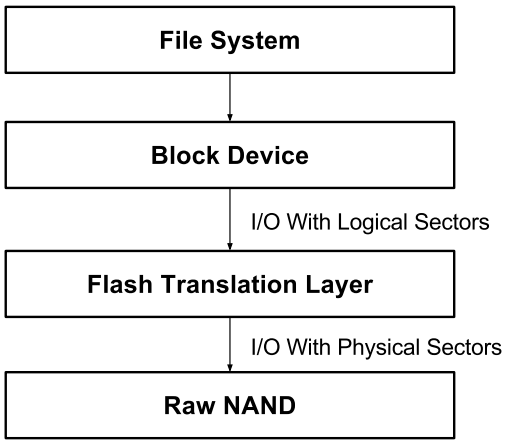

Fig. 2. Flash Storage

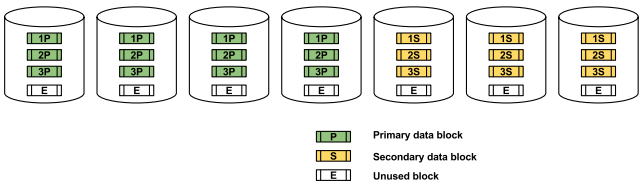

Fig. 3. Blocks distribution with constant redundancy

The cost of erase is much more than both read and write and this leads to performance degradation while writing data to flash. Storage controllers address these issues by providing intelligent data placement algorithms, the central of which is the '*flash translation layer* (FTL)'. The SSD, through the FTL, exposes an array of '*logical block addresses* (LBAs)' to the application. The FTL maps these LBAs into physical addresses. This is shown in figure 2. The address translation and mapping are done dynamically to reduce the previously mentioned constraints of the flash memory. The mapping is done in a very efficient, often hardware assisted way. The FTL also has other functionalities like garbage collection, wear leveling, and abstracting the hardware parallelisms of the SSD. Interested readers can refer to a detailed survey of the FTLs and flash internal characteristics that have been compiled previously [17], [18].

III. Variable Redundancy

In this section, we discuss the challenges of providing variable redundancy at a block level. Consider a virtual block device backed by seven physical disks. The virtual block device exposes an array of logical block addresses (LBAs) to its host. Each block has a fixed size, called 'block size'. The virtual block devices stores the data on the physical disks. Consider an erasure coding scheme applied to store logical blocks on the physical disks. Each logical block is striped in 'k' sectors and erasure coded blocks are added to it to create 'm' sectors of data. These 'm' sectors are

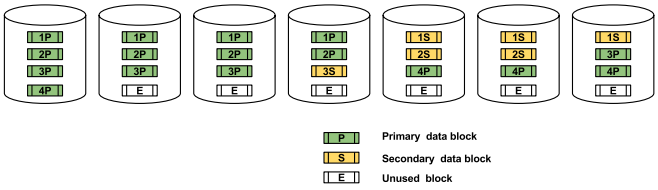

Fig. 4. Blocks distribution with variable redundancy

written to the physical disks. When the chosen redundancy is same for all the blocks in the system, we can deploy a static redundancy distribution similar to traditional RAID, with the only difference being that the number of primary and secondary blocks can be any arbitrary number. This scheme is shown in figure 3, with 7 disks (disk-0 to disk-6). Primary blocks are named as 1P, 2P, 3P, etc., and secondary blocks are named as 1S, 2S, 3S, etc. Even with other schemes like spreading the secondary blocks across all disks to avoid hot-spots in the array, it would be possible to mathematically determine the set of sectors that a virtual block represents.

However, this becomes impossible when a variable redundancy scheme is applied for each block. This scheme is illustrated in figure 4. We can see that, the first block is written with a policy of (4 + 3) and starts from disk-0. The second block is written with a policy of (4 + 2) and hence occupies only 2 secondary blocks. The 3^{rd} block starts with disk-6 and is written with a policy of (4 + 1). Finally, the 4^{th} block starts at disk-5 and is written with a policy of (4 + 0). We can see that the last block on disk-1 to 7 is not used.

This scheme requires maintaining a table which can translate a virtual block number into a set of physical sectors and the disk *ids* in which they are stored. With a variable redundancy system it is not just sufficient to provide a way to store data with different redundancy settings, but also to allow modifying the redundancy levels of the previously written data. When the redundancy of a block is changed to reflect the change in the reliability requirement of the data, the blocks have to be re-encoded according to the new redundancy parameters, and this requires an update in the translation tables. A more sophisticated algorithm may be used to determine the placement of the sectors while striping them across the disk. Various schemes have been studied in the past for such placements [19], [10], [20]. The translation mechanism is further discussed in section V.

IV. Distributed Flash Array

In this section, we examine how a system with block level variable redundancy can work with flash arrays. As discussed in section II, the SSDs include an address translation system in their FTL. A similar translation is required to enable variable redundancy and this leads to duplication of effort. Moreover, in the current flash SSDs, the FTL is implemented inside the controllers and works as a black box as the SSD manufacturers do not reveal their inner workings. This architecture makes it hard to integrate the address translation layers to avoid

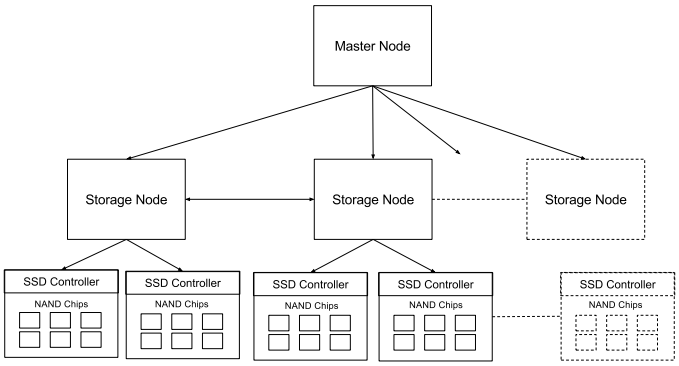

Fig. 5. Multi-Tier Flash Storage Array

duplication. The problems of duplication and other overhead due to FTL has been recognized and discussed in some of the previous works. Some solutions have been proposed to move the management of the blocks to the application layer [21], [22]. The limitations of SSDs in classical RAID like systems [23] also adds to the advantages of managing the data blocks at a higher level.

In the rest of this section, we present a case study of a flash SSD based storage array, where most of the block management is done outside of the SSD. Then, we discuss how *Cider* can be integrated into such a system.

A. Case study: Flash SSD based Clustered Storage

We give a high level description of a highly flexible clustered storage based on flash SSD, as shown in figure 5. Such a cluster will have a multitude of servers, each having a multitude of SSDs. Each SSD has a number of raw NAND chips and a controller. All the servers in the cluster are connected to a 'Master Node'—a special node which acts as a gateway to the entire cluster. In a system like this, in the most basic configuration, the master node handles an I/O and sends the request to the appropriate storage node. The storage node forwards the request to the required SSDs. Finally, the SSD controller chooses the physical NAND page to use and store or fetch the data. The flash wear-leveling, garbage collection (GC), address translation etc., are all controlled locally at the SSD level and the master node has no information about the state of individual SSDs. This limits the ability of the master node to optimize the I/O as the global state of the system is missing. For instance, if one of the SSDs is undergoing garbage collection and the master node issues an I/O to it, it can experience a higher than expected latency. When some of this functionality is moved from the SSD controllers to the master node, we have the benefit of having a finer control over the system while potentially bringing down the cost of SSD controllers.

Such global state gives more flexibility to the entire system and manages block allocation, address translation, wear-leveling, I/O scheduling, and garbage collection through a 'Global FTL'. For instance, given the erase count of all flash blocks, block allocation algorithms can be designed to wear out one SSD at a time rather than a uniform degradation. Another example is the ability to have a better I/O scheduling. If one of the controllers has initiated a garbage collection, the I/O can be issued to a different controller reducing the latency. These optimizations can make the system performance more predictable. Further, the master node can either use the global state to extract the internal parallelism of the SSDs or leave the individual SSDs to handle it.

B. Simulation of Global FTL

To further understand the impact of having global FTL we simulated the working of an SSD array with global FTL. We modified FlashSim [24], [25] to create a global FTL to manage an array of SSDs. We used the `RaidSsd` layer to simulate multiple SSDs. In the SSDs, we bypassed the translation layer and disabled garbage collection and block allocation. Our global FTL controlled block allocation, address translation, and garbage collection. We tested our system for correctness, details of which will be published separately.

C. Cider in Flash

Unlike many difficulties we faced while integrating *Cider* with the existing architecture of flash arrays, the integration becomes straight forward with the system we just described. *Cider's* block allocation and address re-mapping can be easily integrated into the global FTL with no extra overhead. Thus, the system we described above would be able to support storing data at different redundancies as required by the client. With the ability to change the redundancy, the storage becomes elastic, growing and shrinking with the increase and decrease of redundancy respectively.

V. Cider System Architecture

In this section, we discuss the architecture of *Cider*. First, we give an overview of the system, then, we explain the block translation layer and I/O processing in *Cider*.

A. Overview

Figure 6 shows the software components of *Cider*. As previously explained, *Cider* is a virtual block device exposing an array of logical block addresses to the host system. We call these as '*Cider* virtual block numbers'. Typically the block device is accessed through a file system in the kernel or as a raw block device using the POSIX calls from the user space. Since *Cider* itself does not have any physical storage it relies on a storage backend consisting of a set of block storage devices to persistently store data. Each *Cider* virtual block is striped into 'k' segments, over which '$(m-k)$' erasure coded segments are calculated. Finally, 'm' segments are written to 'm' backend storage devices. We call the block numbers on these storage devices as 'physical block numbers'. Usually, for a set of data, the value of k is fixed and the value of m is varied as per the required reliability. But our system handles modifying both k and m. The details of the 'block translation layer' and the way redundancy can be initially set, and then modified, is described in section V-B and the 'I/O engine' is described in section V-C.

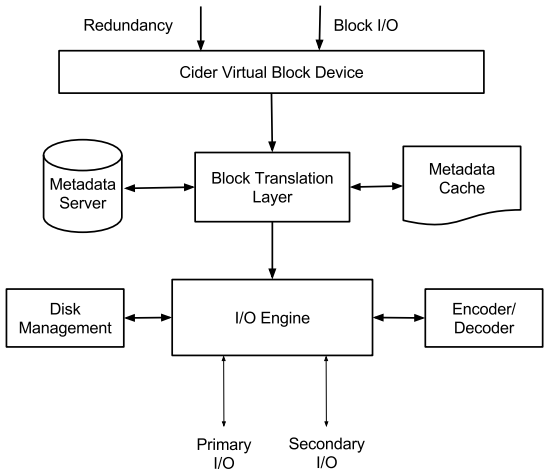

Fig. 6. High Level System Architecture

Cider can be divided into the following components:

1) **Storage Nodes:** These are remote machines or storage controllers where the data will be physically stored.
2) **Master Node:** *Cider* runs on the master node. It is connected to a set of the remote storage nodes, presented as block devices (Ex. as LUNs).
3) **Metadata Server:** The Metadata server stores *Cider's* metadata. This can either run on the master node or on a dedicated server and is typically a highly fault tolerant and fast server.

As an example configuration, consider a network of 7 storage nodes with a capacity of 1TB each and 1 master node. Let the erasure coding scheme be 4 + 3, where 3 is the maximum number of failures the system can tolerate. Let the block size be 4KB.

1) When a write request is made to the *Cider* block device (typically through a file system) each block is striped into 4 sectors of 1KB each and written to 4 remote disks as primary blocks.
2) 3 erasure coded blocks are calculated and written to 3 remote disks as secondary blocks.
3) When a read request is made, primary blocks are read in parallel from the 4 disks and rearranged by *Cider*. In the case of failures, data is reconstructed by reading secondary blocks.

The implementation of *Cider* would depend on the targeted platform. On Linux, we propose to implement *Cider* as a Linux virtual block device as a kernel module using the Device Mapper framework [26].

B. Cider Block Translation Layer

Every virtual block in *Cider* has an associated metadata and the block translation layer has the job of converting any given virtual block number to a set of physical block numbers. As explained previously, this is similar to the address translation in the FTL of SSDs. For writes, the new physical blocks have to be allocated according to the specified k and m values and for reads, these values have to be retrieved. For this purpose, it uses the following two sub-components.

Block Metadata – BlockMap: This is a large map holding the fields required for the translation of a virtual block to a set of physical blocks. Each virtual block in use will have an entry in the BlockMap. The size of the block map increases with the size of the system and has to be persistent. This can be stored either on the master node or on a dedicated metadata server. The block translation layer maintains a metadata cache for faster access. As we show in our prototype, discussed in section VI, when 8 nodes are used, each block is associated with a metadata of 68 bytes. For a block size of 4KB, this would amount to an overhead of about 1.6%. The overhead is significantly lower for larger blocks.

Block allocator – Bitmap: The block translation layer also has to manage the physical block allocations and deallocations. Whenever there is a write request to a new virtual block, we have to assign a set of unused physical blocks to this virtual block. To do this, we maintain a bitmap representing the blocks for each device. With the increase in the number of blocks, searching a bitmap for an available free block becomes increasingly expensive. This is a common problem with file systems and previous approaches such as using hierarchical bitmaps and B-Tree data structures can improve the efficiency [27]. Given a block size of 4KB, the size of all the bitmaps would be about 0.2% of the system capacity. Therefore, the storage overhead due to metadata would be about 1.8-2.0% of the system's capacity.

On the Linux platform, when implemented as block device driver, *Cider* provides a unique ioctl called `CIDER_SET_REDUNDANCY` to set the required redundancy. The clients must set the required redundancy before each write operation, without which the system uses a pre-configured redundancy level. For instance, when using this feature to set the redundancy for files, the 'Extended Attributes' of the filesystem can be used to determine the required reliability for a given file. When the 'reliability degree' extended attribute of a file is set, the corresponding parameters (k and m) can be passed to *Cider* using the aforementioned ioctl and *Cider* will write the data accordingly. This requires only minor modifications to a filesystem like ext4. To dynamically adopt to the temporal change in the reliability of data, *Cider* allows modifying the redundancy of the previously written data blocks. To modify the redundancy of an existing block, a new redundancy value has to be set and a write call with no data has be to issued. *Cider* handles such a write call as a special case and re-encodes the previous data with the new parameters. We will discuss more on the mechanisms of the I/O processing later in this section.

Lastly, the placement of the blocks is crucial to maintain high reliability. For instance, we have to make sure no two sectors of a virtual block reside in the same failure domain. However, this is an orthogonal problem to the one we are trying to solve and we can use many of the previously suggested techniques [10], [19], [20] to ensure reliability.

C. Read/Write Processing

All the I/O requests and the redundancy modification requests of the data of the *Cider* block device are handled by the 'I/O Engine'.

Write: On each write request, the caller sets the values of k and m as described in the previous section. Using these values, a virtual block number is converted to a set of physical block numbers, as discussed in section V-B. Each block is striped and primary data along with encoded secondary data is written to remote nodes. As previously described, a write with no data is considered as a special write issued to modify the redundancy of a previously written block. A group of blocks requested by the call is re-encoded as per the new values of k and m and written to disk. Usually, a change in reliability of a block would result in changing only the number of secondary blocks. Therefore only m will be modified while k remains the same, and results in rewriting of $(m-k)$ secondary blocks.

Read: For reads, the requested virtual block is converted to a set of physical blocks with the help of the block translation layer as explained earlier. Then reads are issued in parallel to the corresponding primary devices at the corresponding offsets. If one or more primary sectors cannot be read due to a devices failure, the data is constructed on the fly by reading the required secondary sectors.

Encoder/Decoder: The encoder calculates the erasure coded sectors for every write while the decoder reconstructs the original blocks during failures. This is a pluggable module to the system and the most suitable erasure coding algorithm can be chosen from several options available [9]. The trade-off would generally be between performance and storage cost. For instance, if the user is willing to bear higher storage costs (use 2x secondary disks for the same level of protection) or opt to get protection only against $k/2$ disks, she can choose faster algorithms which give a better performance.

Disk Management: The disk management module does all the maintenance activities of the system like full disk reconstructions and detecting intermittent failures. The use of software controlled erasure codes makes the system more flexible and resilient under failures. We discuss more strategies for disk management and handling failures in section VII.

VI. Prototype Implementation and Evaluation

In this section, we discuss the implementation of *Cider* prototype. We evaluate our prototype for correctness and use micro-benchmarks and throughput tests to demonstrate the practicality of *Cider*.

Figure 7 shows the implementation of our prototype. We use Network Block Device (NBD) [16] to simulate a virtual block device. On the master node, we start the NBD server and set up the NBD client as a loopback device. This redirects all the I/O calls of NBD client to the NBD server running in user space. We intercept these requests in the server and implement all the functionality of *Cider* discussed in section V in user space. The applications access our virtual device through the NBD client.

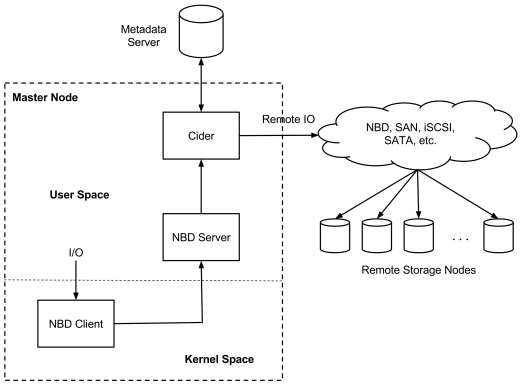

Fig. 7. Cider Implementation

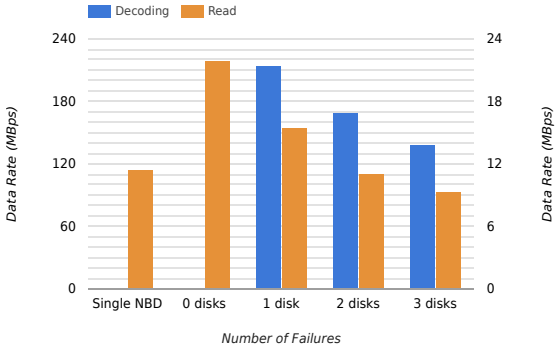

Fig. 8. Read benchmarks

In our prototype, we have implemented block translation and block management layers to support the variable redundancy. To support redundancy, we have used erasure code based on the open source library of zfec-Tahoe [28], which is a Reed-Solomon coding library based on previous works [29], [30]. Our metadata server is also implemented on the master node.

A. Microbenchmarks

We initially deployed our system on off-the-shelf machines in our lab for preliminary benchmarks. Our testbed configuration used a server with Intel Pentium-D 3.20GHz and 4GB RAM. We connected 7 Remote disks to the server over the NBD protocol. We performed the experiments to measure the throughputs for large sequential I/O. We used the same configuration discussed in the earlier example of V-A. We used 4 primary disks and 3 secondary disks and a block size of 4KB. Each block was striped to sectors of size 1KB. Although block sizes are getting larger in newer systems, we chose 4KB as it is the most commonly used block size in today's file systems like ext4. Using larger block size will provide better performance due to reduced metadata size and reduced fragmentation. The write tests synced all the data to the disks and the read tests were conducted on a cold system after clearing the kernel level cache.

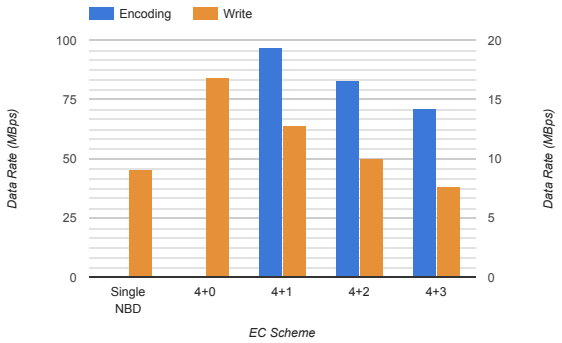

Fig. 9. Write benchmarks

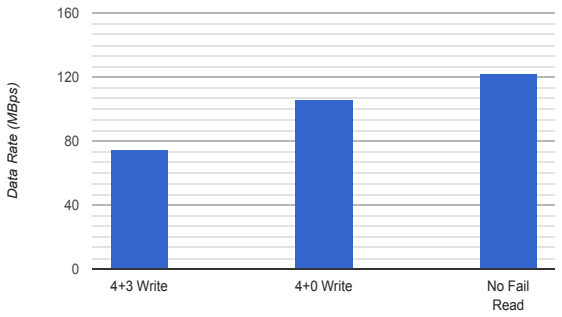

Fig. 10. Throughputs on high performance hardware

Figure 8 shows the read throughputs of our system for (4 + 3) scheme under 0, 1, 2 and 3 disk failures (y-axis on the right). These can be contrasted against the decoding throughputs due to erasure coding (y-axis on the left). We have also included a raw NBD read performance from a single device to compare the performance of our system. We are able to achieve higher throughputs when there are no failures. The number of failed disks is indicated on x-axis.

Figure 9 shows the write throughputs of our system for different redundancy schemes (y-axis on the right) along with the encoding throughputs (y-axis on the left). The erasure coding schemes is indicated on the x-axis.

B. Scalability on High-End Machines

To test our system on a high performance hardware, we collaborated with a leading storage vendor[2] and deployed our system on their highly scalable hardware. We conducted limited experiments on this setup and we leave publishing of a comprehensive test report for our future work. This configuration included 8 disks configured in direct attached storage mode (used as remote disks), a master node with Intel Xeon Processor X5690 and 48GB RAM, running Linux kernel—3.14.22 and a dedicated SSD for metadata. The figure 10 shows the throughputs of (i) write with an erasure coding scheme of (4 + 3) applied, (ii) write with no erasure coding and (iii) reads under no-failure conditions. The block size and the erasure coding configuration was similar to the previous tests. The results were very promising as our system was able to achieve reasonably high throughputs even on this setup. Both read and writes were done with kernel cache disabled.

[2]We thank Scalable Informatics Inc., Detroit, for letting us use their systems for performance benchmarking

VII. Implementation Challenges and Discussion

In this section, we discuss some of the engineering challenges and suggest some improvements that can be built on top of *Cider* to make it more robust and reliable. Many of the enhancements and features we discuss below requires an 'out-of-band' connection system management, which is typically present in large storage arrays.

A. Disk Recovery

The disk recovery framework provides a mechanism to reconstruct an entire disk and copy the contents to a new disk when it fails. Because of the flexibility that comes with erasure coding, the recoveries can be triggered by the administrator through a tool or when the system is light on load. When the reconstruction of a disk is in progress *Cider* can make that disk unavailable and continue to serve the reads to that disk by reconstructing data. To keep the availability high, the writes can be processed by directing the I/O to the new disk using some on the previously studied online rebuilding techniques [31], [32].

B. Handling failures

In this section, we discuss some of the implementation techniques to handle failures.

Temporary Node failures: Full disk reconstruction is used for permanent failures but intermittent failures have to be handled differently. Again, the use of erasure codes gives us more flexibility to handle this. During writes, a virtual block can be remapped to a set of nodes/devices which are available without affecting the performance. The reads of affected blocks would still have to be served by the process of reconstruction.

Heartbeat: Using a common technique used in distributed system we can detect node or device failures continuously. This enables us to pro-actively reconstruct the affected blocks and remap them to the available nodes whenever a particular nodes goes down.

Master Node Failure: In our design, the master node serves as a gateway to the entire system. The failure of the master node will make the system inaccessible. To increase the availability, we can have a high-availability (HA) pair master node where the secondary node can take over in an event of failure. This problem is also applicable for metadata server, which acts as a single point of failure. With the loss of metadata, it would be impossible to recover the data. Again, in this case, we can use a HA pair or a highly reliable server.

Data Scrubbing: We can use data-scrubbing techniques to periodically verify the integrity of the data blocks. In an event of data loss, we can remap the affected blocks to a different device. This will be helpful in preventing undetected and silent data corruption.

C. Disk Defragmentation

Because of the remapping of blocks, another unique challenge our system has to deal with is fragmentation. With the usage of the system, the performance can degrade due to sparse distribution of the blocks. This is especially true if the system is initially used with high redundancy and later changed to a lower redundancy model. For example, if the system has written the first 1000 blocks with a scheme of 4+4, and later the first 100 blocks are converted to a 4+0 scheme, the associated secondary blocks are freed. When these freed blocks are reallocated to another virtual block they will be highly fragmented. We can have an automatic de-fragmentation system which periodically scans through the list of blocks and rearranges them in a contiguous fashion.

D. Increasing Parallelism:

Currently, the *Cider* virtual block device appears like one device to the filesystem. This limits the amount of parallelism to the value of k. For instance, if a system has 64 disks, but the chosen value of k is 4, we can achieve only 4 way parallelism. The reads have to be sequentially served because of the way the I/O queue is implemented in the kernel. To get around this, we are working on a mechanism in which we extract the request in the I/O queue and issue reads to all remote nodes to extract maximum parallelism. Achieving parallelism during writes is more challenging because the sync calls acts as barriers. Microsoft's Blizzard [14] addresses a similar problem and is able to achieve high parallelism without trading consistency. *Cider's* block translation layer helps to do out-of-place writes to maintain consistency during parallel writes. Once a high degree of parallelism is achieved in the block layer, even regular filesystems can benefit from faster I/O rather than requiring dedicated parallel filesystems.

VIII. Related Work

While there is no previous work we are aware of that directly competes with *Cider*, previous works have tackled various challenges of designing distributed storage systems for cloud scale. These range from creating new systems to increase parallelism to systems which can save storage space. There is also an active research in the field of erasure codes where new techniques try to optimize for storage space and computation time. Optimizing the storage stack for flash based storage is another active area of research. *Cider* can work in conjunction and benefit from many of these systems rather than competing with them.

Distributed and Block Storage: Microsoft Blizzard implements a highly parallel, virtual block store for cloud-scale applications [14]. While *Cider* shares the goals of creating a system with a POSIX based block interface, its emphasis is to provide higher flexibility and reliability. Microsoft has also proposed erasure coding at a block level [33]. This system applies a pre-configured erasure coding scheme to the data and lacks the flexibility and fine granularity provided by *Cider*. IBM's work on network RAID protocols [34], and systems like Starfish [35] have explored creating a RAID like system over the network. Distributed Replicated Block Device [36] creates a replicated block storage over the network. Sheepdog [37] is a distributed block storage system tuned for virtual machines. While these systems cater to specific use cases, *Cider* can complement them by adding flexibility through its variable redundancy scheme.

Erasure Coding: Hydra [38], like *Cider*, builds a flexible storage system that utilizes erasure coding. But *Cider* works at a much lower level in the software stack and has a lower storage overhead compared to Hydra. Many other major players like OpenStack Swift [11] and Ceph [39] are also incorporating erasure coding in their cloud stores. SafeStore [40] uses erasure coding and distributes data across multiple storage service providers to increase its reliability. Apart from systems implementations, a lot of effort is dedicated to build efficient mathematical algorithms that are more efficient than the Reed-Solomon erasure coding technique [41], [42], [43], [44]. The main focus of these systems is in reducing the number of data blocks required to recover a missing block and optimizing usage of XOR based algorithms [45]. *Cider* can benefit from these erasure coding libraries.

Flash and FTL: Systems like CAFTL [15] and Delta FTL [46] explore ways of integrating FTL look ups with other functionality like deduplication. MinFlash [47] proposes to keep the SSDs simple by managing them directly from the kernel. Other systems make a case for moving the complexities out of SSDs and manage the FTLs outside of SSD—NoFTL [48] integrates the FTL into database, REDO [22] and Application Managed Flash [21] propose managing FTL in the filesystem. *Cider* proposes a similar integration for optimizing the translation and can work in conjunction with any of these systems.

IX. Conclusion

With various examples, we have motivated a case to classify data depending on its criticality and the need to have a storage system which can handle this classification efficiently. We have discussed the limitations of existing storage solutions in handling this kind of disparate data sets at cloud-scale. To solve these problems, we have designed *Cider* whose primary goal is to provide a high performance, flexible, and reliable data store at a minimal storage cost. *Cider* supports storing data with different redundancy settings which can be changed over time and allows the redundancy to be set at a granularity of a single block. Being at a very low level, *Cider* can be used by any filesystem or even as a raw device and works well with applications that require or assume a POSIX based block interface. The potential storage savings that *Cider* offers makes it a suitable candidate in cloud storage, complementing the existing model. We also showed how *Cider* can be easily integrated with large scale flash storage arrays and we believe such a system can offer a truly heterogeneous block storage system. Lastly, our prototype implementation and evaluation demonstrates the practicality of our system.

X. Acknowledgements

This research was based upon work supported by (while Dr. Vipin Chaudhary was serving at) the National Science Foundation. Any opinion, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

References

[1] D. A. Patterson, G. Gibson, and R. H. Katz, "A case for redundant arrays of inexpensive disks (raid)," in *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*, SIGMOD '88, (New York, NY, USA), pp. 109–116, ACM, 1988.

[2] R. Harris, "Why raid 5 stops working in 2009." http://goo.gl/0ydSRz.

[3] C. Mellor, "Bigger drives mean the raid rebuild must become a thing of the past." http://goo.gl/TLPqdw.

[4] R. Harris, "The post-raid era begins." http://goo.gl/K7tSsm.

[5] "Amazon s3." https://aws.amazon.com/s3.

[6] "Openstack object store." https://wiki.openstack.org/wiki/Swift.

[7] "Microsoft azure." https://azure.microsoft.com.

[8] J. S. Plank, "A tutorial on reed-solomon coding for fault-tolerance in raid-like systems," vol. 27, pp. 995–1012, Sept. 1997.

[9] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn, "A performance evaluation and examination of open-source erasure coding libraries for storage," in *USENIX FAST '09*.

[10] C. Huang, H. Simitci, Y. Xu, A. Ogus, B. Calder, P. Gopalan, J. Li, and S. Yekhanin, "Erasure coding in windows azure storage," in *USENIX ATC '12*.

[11] K. G. Paul Luse, "Swift object storage: Adding erasure codes," tech. rep., Storage Networking Industry Association, 2014.

[12] P. Misra, N. Roy, S. Naskar, and S. Dey, "An erasure coded archival storage system," in *IEEE Parallel and Distributed Systems, '12*.

[13] S. Chandrashekhara, M. Kumar, and V. Chaudhary, "System and method for fault-tolerant block data storage," Oct. 22 2015. WO Patent App. PCT/US2015/026,267.

[14] J. Mickens and E. B. N. et.al, "Blizzard: Fast, cloud-scale block storage for cloud-oblivious applications," in *USENIX NSDI 14*, Apr.

[15] F. Chen, T. Luo, and X. Zhang, "Caftl: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives.," in *FAST*, vol. 11, 2011.

[16] M. Lopez and P. Arturo Garcia Ares, "The network block device," *Linux Journal*, vol. 2000, no. 73es, p. 40, 2000.

[17] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A survey of flash translation layer," *Journal of Systems Architecture*, vol. 55, no. 5, pp. 332–343, 2009.

[18] L. M. Grupp, A. M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P. H. Siegel, and J. K. Wolf, "Characterizing flash memory: anomalies, observations, and applications," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 24–33, IEEE, 2009.

[19] R. Chennamsetty, B. Dolph, S. Patil, and R. Shiraguppi, "Placement of data fragments generated by an erasure code in distributed computational devices based on a deduplication factor," Dec. 17 2015. US Patent App. 14/307,395.

[20] Y. Hu and D. Niu, "Reducing access latency in erasure coded cloud storage with local block migration," in *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*, pp. 1–9, IEEE, 2016.

[21] S. Lee, M. Liu, S. Jun, S. Xu, J. Kim, and Arvind, "Application-managed flash," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, (Santa Clara, CA), pp. 339–353, USENIX Association, 2016.

[22] S. Lee, J. Kim, and A. Mithal, "Refactored design of i/o architecture for flash storage," *IEEE Computer Architecture Letters*, vol. 14, no. 1, pp. 70–74, 2015.

[23] A. Kadav, M. Balakrishnan, V. Prabhakaran, and D. Malkhi, "Differential raid: Rethinking raid for ssd reliability," in *HotStorage 2009: 1st Workshop on Hot Topics in Storage and File Systems*, Association for Computing Machinery, Inc., October 2009.

[24] Y. Kim, B. Tauras, A. Gupta, and B. Urgaonkar, "Flashsim: A simulator for nand flash-based solid-state drives," in *Advances in System Simulation, 2009. SIMUL'09. First International Conference on*, pp. 125–131, IEEE, 2009.

[25] "Flashsim." https://github.com/MatiasBjorling/flashsim.

[26] "Linux device mapper." https://www.kernel.org/doc/Documentation/device-mapper/.

[27] A. K. KV, M. Cao, J. R. Santos, and A. Dilger, "Ext4 block and inode allocator improvements," in *Linux Symposium*, vol. 1, 2008.

[28] Z. Wilcox-O'Hearn, "Zfec 1.4. 0," *Open source code distribution: http://pypi.python.org/pypi/zfec*, 2008.

[29] P. Karn, "Dsp and fec library," 2007.

[30] L. Rizzo, "Effective erasure codes for reliable computer communication protocols," *SIGCOMM Comput. Commun. Rev.*, vol. 27, pp. 24–36, Apr. 1997.

[31] M. Belhadj, R. Daniels, and D. Umberger, "Raid rebuild using most vulnerable data redundancy scheme first," Feb. 4 2003. US Patent 6,516,425.

[32] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou, "Scalable performance of the panasas parallel file system.," in *FAST*, vol. 8, pp. 1–17, 2008.

[33] J. Li, L.-w. He, and J. Liang, "Distributed data storage using erasure resilient coding," Nov. 2011. US Patent 8,051,362.

[34] D. R. Kenchammana-Hosekote, R. A. Golding, C. Fleiner, and O. A. Zaki, "The design and evaluation of network raid protocols," *Report RJ*, vol. 10316, 2004.

[35] E. Gabber, J. Fellin, M. Flaster, F. Gu, B. Hillyer, W. T. Ng, B. Özden, and E. A. Shriver, "Starfish: highly-available block storage," in *USENIX ATC '03*.

[36] P. Reisner and L. Ellenberg, "Replicated storage with shared disk semantics," in *Proceedings of the 12th International Linux System Technology Conference (Linux-Kongress), Germany*, pp. 111–119, 2005.

[37] K. Morita, "Sheepdog: Distributed storage system for qemu/kvm," *LCA 2010 DS&R miniconf*, 2010.

[38] L. Xu, "Hydra: A platform for survivable and secure data storage systems," in *ACM workshop on Storage security and survivability, '05*.

[39] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *USENIX OSDI '06*.

[40] R. Kotla, L. Alvisi, and M. Dahlin, "Safestore: A durable and practical storage system," in *In USENIX Annual Technical Conference*, pp. 07–20, 2007.

[41] J. S. Plank, M. Blaum, and J. L. Hafner, "Sd codes: erasure codes designed for how storage systems really fail.," in *FAST*, pp. 95–104, 2013.

[42] O. Khan, R. C. Burns, J. S. Plank, W. Pierce, and C. Huang, "Rethinking erasure codes for cloud file systems: minimizing i/o for recovery and degraded reads.," in *FAST*, p. 20, 2012.

[43] M. Xia, M. Saxena, M. Blaum, and D. Pease, "A tale of two erasure codes in hdfs.," in *FAST*, pp. 213–226, 2015.

[44] K. Rashmi, P. Nakkiran, J. Wang, N. B. Shah, and K. Ramchandran, "Having your cake and eating it too: Jointly optimal erasure codes for i/o, storage, and network-bandwidth.," in *FAST*, pp. 81–94, 2015.

[45] Y. Zhu, P. Lee, Y. Xu, Y. Hu, and L. Xiang, "On the speedup of recovery in large-scale erasure-coded storage systems," *IEEE Transactions on Parallel and Distributed Systems, '14*.

[46] G. Wu and X. He, "Delta-ftl: Improving ssd lifetime via exploiting content locality," in *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, (New York, NY, USA), pp. 253–266, ACM, 2012.

[47] M. Liu, S.-W. Jun, S. Lee, J. Hicks, *et al.*, "minflash: A minimalistic clustered flash array," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016*, pp. 1255–1260, IEEE, 2016.

[48] S. Hardock, I. Petrov, R. Gottstein, and A. Buchmann, "Noftl: Database systems on ftl-less flash storage," *Proceedings of the VLDB Endowment*, vol. 6, no. 12, pp. 1278–1281, 2013.

[49] A. Cozzette, "Buse." https://github.com/acozzette/BUSE.

[50] "dd." http://linux.die.net/man/1/dd.