Disaggregated RAID Storage in Modern Datacenters

Junyi Shu
Peking University
China
shujunyi@pku.edu.cn

Ruidong Zhu
Peking University
China
zhurd@pku.edu.cn

Yun Ma
Peking University
China
mayun@pku.edu.cn

Gang Huang
Peking University
China
hg@pku.edu.cn

Hong Mei
Peking University
China
meih@pku.edu.cn

Xuanzhe Liu
Peking University
China
liuxuanzhe@pku.edu.cn

Xin Jin
Peking University
China
xinjinpku@pku.edu.cn

ABSTRACT

RAID (Redundant Array of Independent Disks) has been widely adopted for decades, as it provides enhanced throughput and redundancy beyond what a single disk can offer. Today, enabled by fast datacenter networks, accessing remote block devices with acceptable overhead (i.e. disaggregated storage) becomes a reality (e.g., for serverless applications). Combining RAID with remote storage can provide the same benefits while creating better fault tolerance and flexibility than its monolithic counterparts. The key challenge of disaggregated RAID is to handle extra network traffic generated by RAID, which can consume a vast amount of NIC bandwidth. We present dRAID, a disaggregated RAID system that achieves near-optimal read and write throughput. dRAID exploits peer-to-peer disaggregated data access to reduce bandwidth consumption in both normal and degraded states. It employs non-blocking multi-stage writes to maximize inter-node parallelism, and applies pipelined I/O processing to maximize inter-device parallelism. We introduce bandwidth-aware reconstruction for better load balancing. We show that dRAID provides up to 3× bandwidth improvement. The results on a lightweight object store show that dRAID brings 1.5×-2.35× throughput improvement on various workloads.

CCS CONCEPTS

• **Hardware** → **External storage**; • **Information systems** → **Storage management**.

KEYWORDS

Disaggregated Storage, RAID, NVMe-oF, RDMA

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9918-0/23/03…$15.00
https://doi.org/10.1145/3582016.3582027

ACM Reference Format:
Junyi Shu, Ruidong Zhu, Yun Ma, Gang Huang, Hong Mei, Xuanzhe Liu, and Xin Jin. 2023. Disaggregated RAID Storage in Modern Datacenters. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3582016.3582027

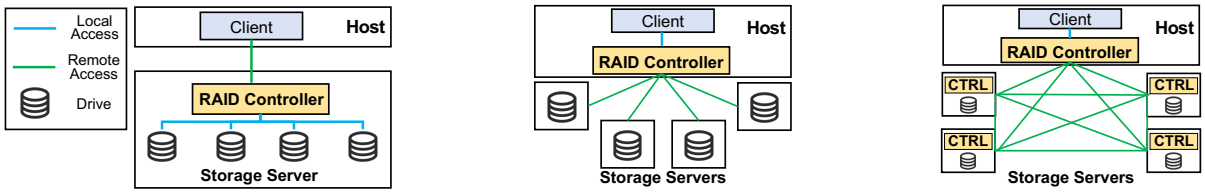
1 INTRODUCTION

RAID [24, 50] is a canonical solution for building reliable, high-performance storage systems. It enables users to build a virtual block device of desired volume, bandwidth, and redundancy from an array of commodity storage drives. It has been widely adopted by industry since its invention in the past three decades. Due to its high impact, RAID has been one of the central research topics in storage systems [31, 36, 40, 44, 45, 60, 61].

Out of all RAID levels, parity-based RAID levels (e.g., RAID-5/6) are particularly popular because they can tolerate device failures with minimal space overhead. Today, RAID is a key feature in enterprise storage solutions [2, 3, 10, 13, 17]. Even though the landscape of storage hardware has changed significantly from hard disk drives (HDDs) to solid state drives (SSDs), parity-based RAID is still crucial for creating a virtual drive with better performance and higher volume while achieving data durability [5, 8].

Recently, there is an ascending trend in datacenters to disaggregate storage from compute [6, 7, 9, 33, 34, 42, 43, 48]. Disaggregated storage enables infrastructure owners to scale compute and storage resources independently and combine them flexibly. In particular, serverless computing [53] leverages such disaggregation to enable resource elasticity and achieve high resource utilization. Storage disaggregation becomes practical nowadays because datacenter networks have improved dramatically over the past decade in terms of throughput and latency [16, 30, 52, 55].

By contrast, RAID is generally built on a monolithic architecture and located on the same machine as its member drives [2, 10]. Although parity-based RAID can tolerate losing drives, there is a risk that the entire array becomes unavailable if the storage server with the RAID controller and the drives experiences an outage.



(a) Single-Machine (b) Distributed (c) dRAID

Figure 1: Remote RAID architectures

Table 1: Comparison of 3 remote RAID architectures

	Single-Machine	Distributed	dRAID
Fault tolerance	Disk	Disk & Server	Disk & Server
Hot spare	Dedicated	Storage pool	Storage pool
Scaling	Pre-provisioning	On demand	On demand
Write overhead	1×	1-4×	1×
D-Read overhead	1×	N×	1×

Unlike single-drive disaggregation which may allow sharing among multiple tenants [43, 48], the common use case for RAID is to satisfy the unmet demand of a single client for bandwidth or volume that a single drive cannot provide. When the only client fails, whether RAID is available becomes irrelevant. This means a user can achieve the highest availability possible by placing the RAID controller at the client side and use remote drives from different storage servers to diverse the risk of a single point of failure.

Furthermore, RAID over disaggregated storage is easier and less expensive to maintain. Conventionally, IT admins must keep at least one hot spare for each RAID array and immediately replace the failed disk in case of disk failures. Also, enough disks have to be planned in advance on each storage array for future scaling. Although parity-based RAID is space-efficient, maintaining it requires over-provisioning and creates a heavy operational burden. With disaggregated storage, IT admins can replace or add a disk from a shared remote storage pool, which can significantly reduce the TCO (total cost of ownership) of the infrastructure.

RAID over disaggregated storage is not a free lunch. It puts extra burden on the datacenter fabric. A major problem is partial parity update of parity-based RAID [24]. A partial stripe write triggers two reads followed by two writes in RAID-5, which results in 4× bandwidth consumption in total. A solution to this problem is to batch partial stripe writes and only submit full stripe writes [46]. This approach requires using non-volatile memory as the cache layer and causes I/O amplification in the background.

Today, a server can only saturate the bandwidth of a few remote SSDs even with a high-end 200 Gbps NIC equipped. The NIC bandwidth can be easily overwhelmed by the extra network traffic generated by partial stripe writes of parity-based RAID. Unlike the inevitable extra I/Os on drives, our key insight is that the extra traffic in the network can be completely eliminated.

We present dRAID, a new approach to minimize the bandwidth overhead of remote RAID by disaggregating part of the I/O handling to storage nodes. We compare different designs of a remote RAID system. Table 1 shows that dRAID can have all the benefits of a remote distributed RAID while keeping network overhead minimal. To achieve that, we argue that a fundamental architecture change is necessary to allow peer-to-peer data access in a storage array. As Figure 1 illustrates, dRAID distinguishes itself from existing RAID systems in terms of system architecture and network topology.

dRAID adopts a hybrid design and has a host-side controller along with a server-side controller on each remote target. The host-side controller serves two purposes. First, RAID does not allow concurrent writes to the same stripe. The host-side controller only admits one write I/O on a stripe at a time and keeps the others in a queue. Second, a full stripe write incurs no remote read I/Os, and thus it is optimal in terms of network and compute usage to simply calculate the parity at the host side.

We propose three key techniques to maximize disaggregated RAID performance for dRAID. First, we design a non-blocking multi-stage write mechanism to achieve maximum parallelism among storage nodes. dRAID removes the dependency between parity reduction and metadata arrival, which allows parity reduction to proceed till the last step without waiting for the metadata to arrive. Second, we introduce an I/O pipeline for each individual dRAID operation, which maximizes parallelism across CPU, NIC and NVMe SSD. Unlike NVMe-oF read and write which must process each I/O sequentially, a dRAID operation consists of multiple network and storage I/Os which can be overlapped and pipelined. Third, We propose a bandwidth-aware load-balancing algorithm for RAID reconstruction. The data-intensive RAID reconstruction can easily overwhelm any single node. We construct a probabilistic reducer-selection model based on available bandwidth to avoid overloading the storage nodes that are already busy.

We note that distributed RAID and distributed parity calculation are not new. Similar ideas have been attempted by researchers since early nineties, such as Tertiary [57] and TickerTAIP [23]. Today, distributed software RAID can be accelerated with new instruction sets of CPUs [17] and high-speed networks [16] in datacenters. The key difference of dRAID is that it targets the bottleneck of NIC bandwidth for the scenario of disaggregated storage in modern datacenters. We propose three new techniques to build an optimal data path that maximizes parallelism of modern NICs and NVMe drives on the host and each remote storage server.

In summary, we make the following contributions.

- We design dRAID, a disaggregated RAID system that exploits inter-node and intra-node parallelism of disaggregated storage to maximize throughput while keeping bandwidth overhead minimal on both host and storage servers.

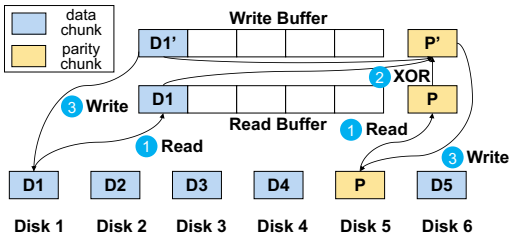


Figure 2: RAID-5 read-modify-write

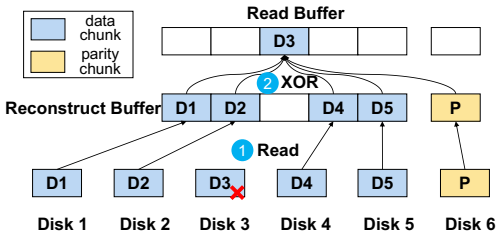


Figure 3: RAID-5 reconstruct read

- We extend NVMe-oF protocol to support disaggregated read and write operations (§4). We design algorithms to efficiently utilize network bandwidth (§5-§6).
- We implement a prototype of dRAID as a fully functional user-space virtual block device atop SPDK [20] that supports standard RAID-5 and RAID-6 (§8).
- We conduct an evaluation on the prototype and show that dRAID scales linearly with the number of remote drives in a RAID array and achieves near-optimal throughput (§9). Overall, dRAID improves the aggregated throughput by up to 3× for disaggregated storage arrays.

2 BACKGROUND AND MOTIVATION

2.1 RAID

Different RAID levels demonstrate unique characteristics in terms of I/O performance, fault tolerance, and space overhead [24]. The most used ones are RAID-0, RAID-1, RAID-5, RAID-6, and combinations of these RAID levels. In practice, RAID-0 and RAID-1 are rarely used alone. In this paper, we focus on parity-based RAID.

RAID-5 and RAID-6 are the most used parity-based RAID levels. They achieve balance among throughput, redundancy, and space overhead. RAID-5 can tolerate losing one drive with only $1/(n - 1)$ space overhead (n is the number of drives). Theoretically, RAID-5 can achieve n times read throughput and $n - 1$ times write throughput of a single drive in the most optimistic case.

In reality, RAID controllers can hardly achieve this throughput. There are three different modes when writing data to parity-based RAID. Read-modify-write and reconstruct write are way less efficient than full stripe write. We take an example of read-modify-write mode in RAID-5 to explain the inefficiency.

Read-modify-write occurs when a minority of data chunks in a stripe need to be written (Figure 2). Note that XOR is an associative operation [22], therefore calculating new parity does not require every data chunk in a stripe. Applying the differences between old chunks and new chunks to the old parity chunk results in the new parity chunk. Read-modify-write is common for partial stripe write. It avoids reconstructing the entire stripe but it is still an expensive operation, as up to two reads and two writes of the requested size are triggered. It has been a subject of intensive research to avoid or reduce read-modify-write in the data path [25, 26, 35, 47, 56].

Also, reconstructing a lost data chunk is an I/O intensive operation which can dramatically impact performance. For RAID-5, a lost data chunk on the failed drive must be recovered by aggregating all the other chunks in the same stripe with XOR. This creates $n - 1$ times read I/Os of a normal-state read. Figure 3 illustrates how to reconstruct a lost data chunk *D3* from the drives that are still alive.

2.2 Disaggregated Storage

Networked storage has been widely used in datacenters. A Storage Area Network (SAN) is a datacenter network that connects block-level storage to servers [58]. SAN is a core product of cloud providers [11, 12]. A SAN is a dedicated network connected through specialized devices. On top of that, a special protocol named Internet Small Computer Systems Interface (iSCSI) [1, 21] is used to transfer data between servers and block devices. There are two major drawbacks of SAN. First, a common SAN uses specialized hardware which is both expensive and inflexible. Second, the latency of SAN is usually in millisecond scale, while direct-attached storage can achieve sub-millisecond or even sub-microsecond latency.

Disaggregated storage is arising and addressing these problems. NVMe-over-Fabrics (NVMe-oF) [4] is one of the most used solutions. First, NVMe-oF is able to operate on common server NICs, so it does not require a dedicated network for storage or any specialized hardware. Second, transmission of small I/Os can be done in μs-scale in high-speed datacenter networks. Prior work has demonstrated the low latency of a disaggregated SSD [33, 34, 43, 48].

Indeed, ~90% of the NVMe-oF protocol is the same as NVMe protocol, so using it does not require another translation layer.

2.3 Challenges and Opportunities of RAID over Disaggregated Storage

With the support of NVMe-oF in Linux kernel, disaggregated storage can be seamlessly integrated with Linux software RAID (a.k.a., Linux MD driver). However, throughput of RAID over disaggregated storage is nowhere close to the theoretical bound [24], especially for partial stripe writes and reconstruct reads.

Challenge: NIC bandwidth. Read-modify-write operation triggers two writes for RAID-5 and three writes for RAID-6. It means that the maximum write throughput is 50 Gbps for RAID-5 and 33.3 Gbps for RAID-6 with a high-end 100 Gbps RDMA NIC. We conduct a motivating experiment on a Dell Ent NVMe AGN MU U.2 1.6 TB solid-state drive. The write throughput of a single drive is around 19 Gbps. The implication is that the best throughput RAID over disaggregated storage can achieve is 2.6× for RAID-5 and 1.8× for RAID-6 versus a single drive no matter how many drives are added to the array. This makes RAID over disaggregated storage impractical because it cannot scale with the number of drives.

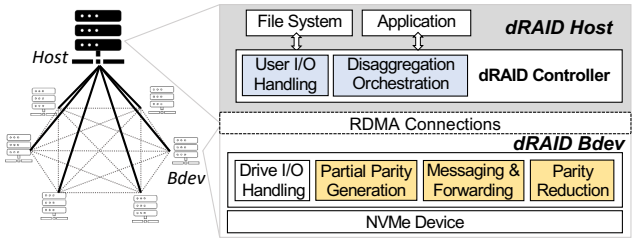


Figure 4: dRAID overview

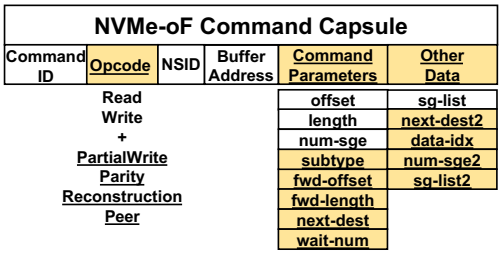


Figure 5: dRAID protocol

Opportunity: direct peer access. We see an opportunity for disaggregated storages to communicate with their peers without host intervention. Conventionally, a local data transfer between two drives must rely on a centralized program to move the data. We argue that a disaggregated drive has both enough bandwidth and its own controller to perform necessary data transfer, so there is no reason for data accesses between remote drives to go through the host and waste network bandwidth of the host.

3 dRAID OVERVIEW

dRAID is a new RAID architecture optimized for disaggregated storage within a datacenter to achieve the best throughput and latency possible. dRAID takes advantage of the unique opportunities exposed by disaggregated storage to directly transfer data between storage servers. Figure 4 shows the architecture overview of dRAID, which consists of a centralized host component and an array of underlying remote block devices.

dRAID host. A dRAID host is where the virtual RAID block device is attached. dRAID host is a user-space software based on SPDK [20] bdev layer that bypasses the kernel stack. dRAID host exposes a standard block interface that allows applications to interact with directly or through a filesystem. Unlike any existing RAID controllers which carry the full responsibility, dRAID host is merely a coordinator that orchestrates the disaggregation of RAID I/Os. All decisions are made by the centralized controller, so that RAID configuration can be hidden from the underlying remote storage servers. In addition to disaggregation orchestration, The host-side controller handles minimal data transfer and parity calculation only in such cases that dRAID makes no gain from disaggregation. At transport layer, dRAID host establishes an RDMA reliable connection (RC) with the server-side controller on each remote storage to send command messages and handle data transfer.

dRAID block device (dRAID bdev). A dRAID bdev is a virtual NVMe block device created by the server-side controller. The server-side controller is a user-space program based on SPDK that controls all the dRAID bdevs on the same storage server. A dRAID bdev is backed by a physical NVMe drive or another virutualized block device. The server-side controller enables a dRAID bdev to handle more than NVMe-oF read and write requests. dRAID extends NVMe-oF protocol to instruct a dRAID bdev to (i) calculate parity in a distributed manner; (ii) forward partial parity to the corresponding peer bdev; (iii) collect partial parities and complete the I/O request. A major change that dRAID makes is that remote storages are connected through RDMA RC in pairs created by the server-side controllers, so that data can be transferred between them. A dRAID bdev is unaware of being in a RAID, and the dRAID protocol includes all the information needed for the extended operations.

4 dRAID PROTOCOL

dRAID protocol is a compatible extension of NVMe-oF. We extend three fields of the NVMe-oF protocol: opcode, command parameters, and other command data. The additional opcodes and fields added by dRAID are underlined in Figure 5.

Opcode. We add four operations to support disaggregated partial stripe write and data reconstruction. *PartialWrite*, *Parity*, and *Reconstruction* are sent by dRAID host to instruct a dRAID bdev to execute partial stripe write, parity reduction, and data reconstruction accordingly. *Peer* operation is used during partial stripe write and data reconstruction for transmitting partial results to a peer.

Command parameters. We re-use the existing command parameters and add extra fields to support the extended operations. *subtype* supports different behaviors on handling the same opcode. *fwd-offset* and *fwd-length* are required whenever data forwarding is necessary because the forwarded segment may not share the same offset and length as that to be written to the drive. *next-dest* defines the destination of the data to be forwarded. And *wait-num* indicates how many pieces of forwarded data must be received.

An alternative design is to broadcast the original I/O request to all remote targets and let each target decides what to do. This design will lead to a simpler protocol. All remote targets must run the same controller logic as the host so additional input from the host is not required. However, We decide not to go with this design, because (i) remote targets have to be aware that they are in a dRAID array and provide full functionality of a centralized RAID controller; (ii) experiments show that saving a few bytes in the request header can merely improve the performance for a block storage.

Other command data. These extra fields are dedicated for RAID-6 because it requires a second parity Q which involves an additional data forwarding call on partial stripe write and uses a different parity generation function that requires additional parameters.

5 DISAGGREGATED PARTIAL STRIPE WRITE

Key idea. Our key idea is to decouple the data path and the control path, so only the host is responsible for coordinating the workflow while allowing remote bdevs to directly share partial parities with peers without host intervention in between.

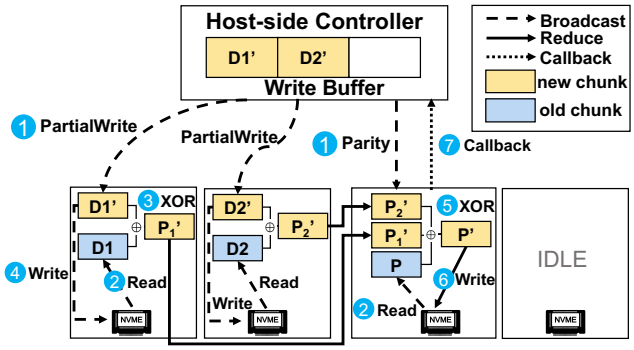


Figure 6: dRAID partial stripe write workflow

Our design depends on the mathematical property of a Galois field. Addition and subtraction in a Galois field is equivalent to XOR. XOR is both *associative* and *commutative*, so multiple XORs do not need to be executed in a specific order. Each bdev is able to derive a partial result independently, so no communication overhead is incurred when parity is generated in a distributed manner.

Based on the property, we divide a partial stripe write into three phases. Figure 6 examplifies the workflow of partial stripe write in dRAID. The *Broadcast* phase is initiated by the host-side controller. The host-side controller broadcasts *PartialWrite* to dRAID bdevs that store the involved data chunks (denoted as $bdev_D$) and *Parity* to the bdev with the parity chunk stored (denoted as $bdev_P$). Upon receiving *PartialWrite*, $bdev_D$ reads/writes data from/to the drive as instructed and calculates a partial parity. $bdev_P$ receives *Parity*, and preloads necessary data for the *Reduce* phase.

During the *Reduce* phase, $bdev_P$ waits for incoming partial results from other bdevs. Partial results are aggregated to the final parity chunk by reduction. All of the expected partial results must be received before the parity chunk is persisted. Upon completion, a signal is sent to the host-side controller to indicate whether the I/O is successful/failed/timed-out in the *Callback* phase.

5.1 Broadcast Phase Handling

Host-side Controller. The host-side controller decides write mode of each I/O and orchestrates the workflow accordingly. In read-modify-write mode, *subtype* for *PartialWrite* and *Parity* is set to *RMW* to indicate the write mode. *fwd-offset* and *fwd-length* must be included because they may not be the same as *offset* and *length* when only part of a data chunk is updated. *next-dest* is also required because the destination of the partial parity must be provided. The key parameter in *Parity* is *wait-num*, which lets $bdev_P$ know how many partial results it should expect. On reconstruct write, *subtype* is set differently to instruct bdevs to skip loading the stored data.

PartialWrite. Algorithm 1 gives the pseudocode of how $bdev_D$ handles a *PartialWrite* command. $bdev_D$ first fetches the data to be written from the host (line 1). The major difference of three subtypes is how they construct the partial parity. A bdev either applies XOR on the segments (line 3-4), concatenates the segments (line 6), or simply reads from the drive (line 8) accordingly. Any updated data blocks are written to the drive, and $bdev_P$ need to be signaled (line 10-11). This signal is yet another dRAID command, and *Peer* opcode is used between bdevs. The parameters must include the address of the partial parity and the corresponding offset and length.

Algorithm 1 Partial stripe write: HandleDataChunk(cmd)

```
   - write_seg: data segment to be written
   - parity_seg: data segment for partial parity
1: fetch remote data to write_seg if there is any
2: if subtype = RMW then
3:     parity_seg_{o,l} ← read from drive
4:     parity_seg_{o,l} ← xor(parity_seg_{o,l}, write_seg_{o,l})
5: else if subtype = RW_WRITE then
6:     parity_seg_{fo,fl} ← read from drive +write_seg_{o,l}
7: else if subtype = RW_READ then
8:     parity_seg_{fo,fl} ← read from drive
9: if write_seg ≠ ∅ then
10:    write write_seg to drive
11: signal(next_dest, addr(parity_seg), fo, fl)
```

Parity. If a stripe is in read-modify-write mode, the existing parity chunk must be read in advance because the sum of the partial results and the old value will be calculated later in the *Reduce* phase. Except for that, $bdev_P$ does not do anything else and just waits for signals from peers. Late arrival of *Parity* command adds more complexity to the *Reduce* phase, which we explain in detail in §5.2.

5.2 Reduce Phase Handling

In the *Reduce* phase, the partial parities are collected and aggregated into the new parity chunk. Algorithm 2 gives the pseudocode. $bdev_P$ is the only actor in the *Reduce* phase [1].

Upon signaled by a peer, $bdev_P$ fetches the partial parity from $bdev_D$ (line 8). If another partial parity with the same offset is already in the memory, then $bdev_P$ immediately reduces them by applying XOR (line 14). Otherwise, $bdev_P$ simply stores the incoming partial parity in the memory (line 16). *offset* is used as the unique identifier to group the partial parities because RAID does not allow concurrent write on a stripe. $bdev_P$ keeps track of the number of received partial parities (line 10). Once all the expected partial parities are received and processed, the final parity is written to the drive and a success signal is sent to the host (line 19-20).

Late arrival of the *Parity* command. Attributed to the mathematical property of parity, dRAID does not require partial parities to arrive in any specific order. However, there is no barrier between the *Broadcast* phase and the *Reduce* phase, so $bdev_P$ may be signaled by a peer before the *Parity* command arrives. The major issue with a late *Parity* is that $bdev_P$ does not know how many partial parities it should expect. $bdev_P$ cannot complete the *Reduce* phase before *Parity* is received, otherwise it may cause data inconsistency.

Adding a barrier between the stages solves the problem, but incurs an expensive synchronization of all the bdevs. Instead, We only block the completion at the last step. $bdev_P$ keeps parity tentatively in memory until (*i*) the *Parity* command is received, (*ii*) all the expected partial parities are processed. Partial parity reduction is not blocked by a delayed *Parity* command because it does not

[1] RAID-6 is dual parity, so there is another $bdev_Q$ that handles the second parity.

Algorithm 2 Partial stripe write: $bdev_P$ handling

```
   - parity_map: offset ⇒ parity buffer
   - wait_num_map: offset ⇒ wait_num
   - buffer_in: buffer for storing incoming data
1: function HANDLE_HOST_PARITY(cmd)
2:     if subtype = RMW then
3:         buffer_in_{fo,fl} ← read from drive
4:         reduce_new_buffer(buffer_in, fo)
5:     wait_num_map[fo]+=wait_num
6:     finish(fo)
7: function HANDLE_PEER_PARTIAL_PARITY(cmd)
8:     fetch remote data to buffer_in
9:     reduce_new_buffer(buffer_in, fo)
10:    wait_num_map[fo]--
11:    finish(fo)
12: function REDUCE_NEW_BUFFER(buffer_in, offset)
13:    if parity_map.has_key(offset) then
14:        parity_map[offset] ← xor(buffer_in,
               parity_map[offset])
15:    else
16:        parity_map[offset] ← buffer_in
17: function FINISH(offset)
18:    if wait_num_map[offset] = 0 then
19:        write parity_map[offset] to drive
20:        signal(host, success)
21:        parity_map.delete_key(offset)
22:        wait_num_map[offset] ← 0
```

depend on any information from the *Parity* command. The *Parity* command arrives before *Peer* commands for most of the time, so further optimization on a delayed *Parity* command is an overkill.

5.3 Parallel I/O Pipeline

Although we limit each bdev to use only one core, there exists an opportunity to accelerate *PartialWrite* by overlapping the I/Os. Because modern NVMe drives allow more concurrent I/Os, queuing I/Os as soon as possible help maximize its parallelism. In dRAID, we observe that fetching remote data and reading the old data from the drive can be done in parallel, while network and drive I/Os in conventional NVMe-oF read and write are serial. Serial execution also unnecessarily delays partial parity reduction because partial parity is ready before the updated data are persisted in the drive.

As Figure 7 shows, we further optimize *PartialWrite* with pipeline parallelism on $bdev_D$. Network I/Os, drive I/Os and CPU-based operations are parallelized to the greatest extent possible. Taking read-modify-write as an example, when a drive read is completed and CPU is notified, both drive write and partial parity generation are ready to be executed next. The CPU starts generating the partial parity and passing it to NIC while data is being written to the drive.

This change brings a consequence that dRAID needs to adapt to. Previously, $bdev_D$ does not make a callback to the host because its completion status is aggregated into the callback of $bdev_P$. Because the drive write is now handled concurrently with partial parity forwarding, $bdev_D$ must make a callback to the host and report its

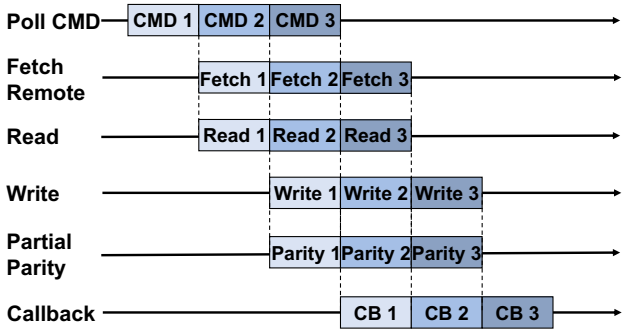


Figure 7: Apply pipeline parallelism to *PartialWrite*

completion status. This callback only depends on completion of the drive write, so that it can overlap with partial parity forwarding.

5.4 Failure Handling Strategies

Our failure model assumes either of the host, servers and drives can fail transiently or for a longer duration. Transient failures, such as timeouts caused by network jitter, are expected on a networked system and should be handled gracefully. Prolonged failures are handled with existing RAID mechanisms. Note that RAID is not expected to support any transactional mechanisms in general. Therefore, dRAID does not provide such support either.

Host failures. Host failures can cause the host-side controller to stop functioning at any moment during a write process. A strawman method for such a failure is to scan the entire array to find all out-of-sync blocks. Linux software RAID uses a bitmap to keep track of which blocks are written to, so a full scan of the array can be avoided. dRAID can just take the same approach.

Server-side failures. For a prolonged server-side failure, dRAID sets the corresponding drive to a faulty state and enters degraded-state as a common RAID. The difference arises on transient failures. RAID over standard NVMe-oF can retry immediately on a transient failure, because NVMe-oF write is an idempotent operation. However, resending a dRAID operation blindly can result in concurrent writes and trigger unexpected behaviors.

A rule of thumb for system design is not to optimize for rare cases. dRAID uses explicit timeout and full stripe retry mechanism to handle transient failures. dRAID sets an upper bound on execution time for each operation. Same as a succeeded operation, once an operation is failed or expired, an event is generated to explicitly notify the host-side controller. The host-side controller only retries after all operations are in one of the final states to avoid concurrent writes on the same stripe. We do not rely on the current state of an expired operation when retrying. A full stripe write is always triggered to prevent data inconsistency.

5.5 Resource Sharing

An enterprise storage server may have tens of drives [14], and thus there is a chance that multiple dRAID bdevs are co-located on the same storage server. Storage, compute and network are three

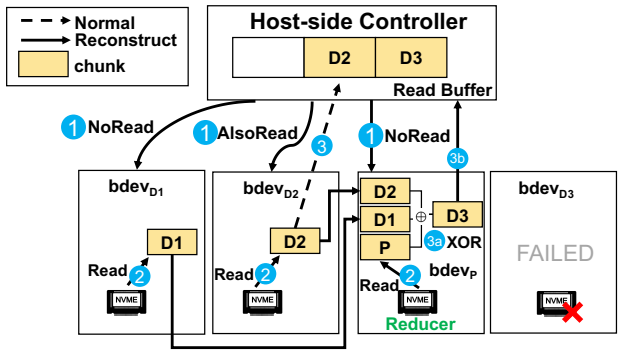


Figure 8: dRAID Degraded-State Read Workflow

types of resources that may be shared on a storage server. The server-side controller on each storage server manages them with corresponding strategies to improve the QoS of dRAID arrays.

Storage sharing. There is a well-known limitation of RAID that it requires all member drives to be homogeneous. The overall performance/size of a RAID array is limited by the wimpiest/smallest drive. A straightforward way to achieve drive homogeneity is to disallow drive sharing and always use drives with the same model and condition. In order to build RAID on shared storage, the key challenge is to partition a physical drive into smaller ones with guaranteed performance [32], which is orthogonal to dRAID. Storage virtualization usually requires drive profiling in advance to understand how a specific drive performs under different workload levels. A QoS controller needs to implement rate limiting at run-time to ensure that a tenant does not exceed its I/O budget.

Compute sharing. Ideally, dedicating one CPU core in poll mode per dRAID bdev avoids interference between the bdevs. However, depending on CPU performance and workload characteristics, a single CPU core for the server-side controller may be enough to process I/Os of multiple bdevs. And using fewer cores when possible help conserve energy in datacenters. Because only a small proportion of end-to-end latency is on CPU, FCFS scheduling should suffice. To guarantee QoS when the a storage server has many dRAID bdevs or a few dRAID bdevs are under heavy workload, the sever-side controller needs to constantly monitor CPU usage and dynamically adjust the number of cores in use.

Network sharing. A storage server usually has more than one NIC. And we assume the total bandwidth is enough to saturate all drives. Because excessive connections can lead to cache thrashing and network I/O latency increases with the growth of load, our goal is to minimize the number of connections created and balance the traffic among NICs. dRAID only creates one shared thread/connection per destination for all dRAID bdevs on the same server. New connections are created on the least used NIC for load balancing. When the workload on a dRAID bdev has substantially changed for a long period, it may be worth migrating the bdev and replanning the NIC placement of the dRAID bdevs.

6 DISAGGREGATED DATA RECONSTRUCTION

Data reconstruction shares the same set of challenges as partial stripe write. All techniques described in §5 also apply to data reconstruction. But there are also two key differences between data reconstruction and partial stripe write.

First, partial stripe write and full stripe write do not happen at once for a stripe, but a read I/O may incur a mix of normal reads and reconstructed reads. We must co-design normal read and reconstructed read to avoid unnecessary storage and network I/Os.

Second, $bdev_P$ is the optimal choice for reducer in partial stripe write because it is where the final result will be persisted. Because parity chunks are evenly distributed among all member drives in RAID-5/RAID-6, the load is already well balanced by default. In contrast, all bdevs play the same role when data reconstruction is performed. It provides more design choices for reduction.

6.1 Degraded-State Read Handling

Key idea. Our key idea on handling simultaneous normal reads and reconstruct reads on a stripe is to combine their drive read I/Os while decoupling the returning data paths of normal reads and reconstruct reads. This allows dRAID to minimize both the number of drive I/Os and network traffic.

Data reconstruction takes a different data path in dRAID to minimize the usage of the host network bandwidth. A reducer is designated to reconstruct the lost data chunk and return it to the host ($bdev_P$ in Figure 8). Passing the normal read data chunks to the reducer unnecessarily wastes bandwidth between the bdevs and memory space of the reducer. Therefore, dRAID lets each bdev return the normal read data chunks directly to the host while aggregating partial results at the reducer.

The complete workflow. Figure 8 illustrates the complete workflow of degraded-state read. When a read I/O requests a lost data chunk, dRAID broadcasts *Reconstruction* commands to all the available bdevs. If read is also requested on the corresponding bdev, *subtype* is set to *AlsoRead*. Among all the available bdevs, a reducer is randomly selected and all the other bdevs are notified.

Upon receiving the *Reconstruction* command, a bdev reads the union of the read segment and the reconstructed segment into memory. Note that the union of the two segments may not be a continuous interval. In such a case, the bdev also reads the blocks in between to avoid an extra I/O. Once data are loaded into memory, a non-reducer bdev prioritizes sending the partial result to the reducer. A reducer needs to wait for the expected partial results before finalizing the reconstructed data. The procedure is almost identical to handling of $bdev_P$ in partial stripe write. And we take the same approach to handle late arrival of a *Reconstruction* command.

Randomized single reducer vs. tree-structured reduce. Intuitively, tree-structured reduce may better balance the load between the bdevs. However, the topology of reduction is irrelevant in our design space because all the bdevs can be randomly assigned to any node in a tree. We prove that random selection is optimal with homogeneous network on each bdev.

Theorem 1. *If each bdev is assigned to a random node in the reduction tree, the average outbound and inbound traffic is fixed for any tree topology.*

Proof. Assume that the given reduction tree has n nodes in total. Because each piece of retrieved data is of the same size in data reconstruction, we further assume that each inbound edge at a node represents a unit of inbound traffic, and each outbound edge represents a unit of outbound traffic.

All nodes in a tree has one outbound edge except the root node. In dRAID design, the root node is the reducer which sends the reconstructed data to host, so there is an extra outbound edge at the root node. Therefore, each node has exactly one outbound edge, which indicates the average outbound traffic is 1.

Assume node n_i has I_i inbound edges. We have the total inbound traffic $\sum_{i=1}^{n} I_i = n - 1$ because there are $n - 1$ edges in the tree. Therefore, the average inbound traffic is $\frac{n-1}{n}$. □

6.2 Bandwidth-Aware Reconstruction

However, network is often heterogeneous at a certain point in the real world, which makes randomization sub-optimal. In dRAID, we can tune the chance that a bdev is selected as the reducer in order to adjust load on that bdev and avoid network congestion.

Problem formulation. Let available bandwidth on bdev i be B_i, total read load per second on the failed bdev be L, the chance that bdev i is selected as the reducer be P_i, and the total number of available bdevs be n. The expected remaining bandwidth on bdev i is formulated as:

$$R_i = B_i - P_i \times (n-1) \times L \qquad \forall\ 1 \le i \le n \tag{1}$$

Our goal is to maximize the smallest R_i so that each bdev has enough headroom for an unexpected load spike.

$$max \min_{\forall\ 1 \le i \le n} R_i \tag{2}$$

Because only one reducer is selected in each reconstruction. The following constraints must be satisfied:

$$\sum_{i=1}^{n} P_i = 1 \tag{3}$$

$$0 \le P_i \le 1 \qquad \forall\ 1 \le i \le n \tag{4}$$

A weakness of this approach is that it assumes a pre-knowledge of the expected reconstruction load. It can only work if the storage array is taken offline during recovery and a background job spawns to reconstruct lost data from a disaster.

In many cases, RAID array is kept online during recovery. To handle dynamic data reconstruction load, we replace L with EWMA (Exponential Weighted Moving Average) of reconstruction load L_{EWMA} as an indicator of future load. And we periodically update all P_i values to react to load changes.

7 DISCUSSION

Offloading the host-side controller. By design, the host-side controller can also be offloaded to a storage server. There are pros and cons by doing that. On the one hand, a full offloading further reduces resource usage on the host side which may benefit the users. On the other hand, it creates another single point of failure and may slightly increase the latency with another NVMe-oF abstraction layer and additional I/O overlay.

RDMA scalability. We choose to use RDMA RC as it provides lower latency and utilizes fewer CPUs than TCP or RDMA unreliable datagram (UD). The limited scalability of RC should not become a bottleneck for dRAID, because there are two key facts unveiled by prior work. (*i*) RC can scale up to 700 queue pairs (QPs) without performance degradation [49]. (*ii*) The common range of number of data drives in a RAID array is between 3 to 26 [31]. Unless an uncommonly large array is considered, a server equipped with a modern RNIC should *not* experience cache thrashing.

Server CPU usage. In general, disaggregated storage is expected to use very limited compute on the server side. i10 uses four x86 cores to saturate the full bandwidth of a high-end SSD [33]. Gimbal argues that a wimpy ARM core is enough to saturate bandwidth of a PCIe SSD [48]. Distributing parity calculation to storage servers certainly creates extra burden. To show dRAID is feasible, we strictly limit dRAID to use only *one* core per SSD on the storage server. Enabled by storage acceleration technology of modern server CPUs [17, 18], our experiment shows that dRAID uses <25% of the CPU cycles, which implies that dRAID is resource-conservative.

Generalization to other erasure coding systems. Erasure coding is the core of parity-based RAID. Besides standard RAID-5/6, there are many variations of RAID systems based on erasure coding [37–39, 51]. Most erasure codes can also be generated in parallel, so I/O disaggregation still applies and can be implemented with some more effort. Note that dRAID requires RDMA NICs and NVMe SSDs to maximize I/O parallelism, which may not be supported by some of the existing storage systems.

8 IMPLEMENTATION

We have implemented a prototype of dRAID on top of SPDK in ~9800 lines of C/C++ code. The core implementation of dRAID consists of a host-side controller and a server-side controller. We refer to the SPDK RAID-5 POC [19] developed by Intel engineers who actively work on Linux software RAID.

The host-side controller implements a virtualized user-space block device interface, and directly establishes an RDMA connection with each remote storage server. Similar to other RDMA-based NVMe-oF implementations, the host-side controller is the passive side in the one-sided RDMA data transfer. The server-side controller is an SPDK application that opens an NVMe drive. It creates a connection with all the other storage servers. We allocate 128 MB memory buffers for each bdev, which is sufficient to saturate the drive bandwidth with a reasonable chunk size.

There are two implementation choices which significantly impact the performance. (*i*) We leverage ISA-L (Intelligent Storage Acceleration Library) [18] to accelerate operations such as XOR and Galois field multiplication when generating parity and reconstructing data. This allows dRAID to benefit from modern x86 instruction sets. (*ii*) We implement a system optimization to enable concurrent read operations on a stripe, while the SPDK POC locks the corresponding stripe during a normal read.

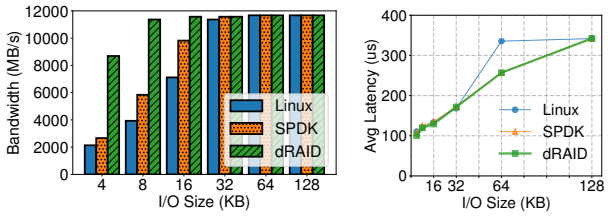


Figure 9: RAID-5 normal-state read on different I/O sizes

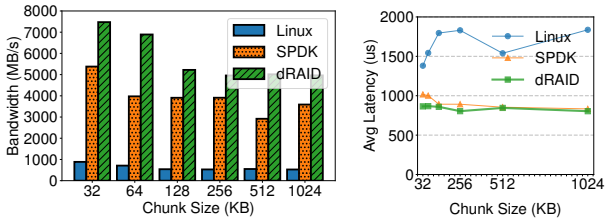


Figure 11: RAID-5 write on different chunk sizes

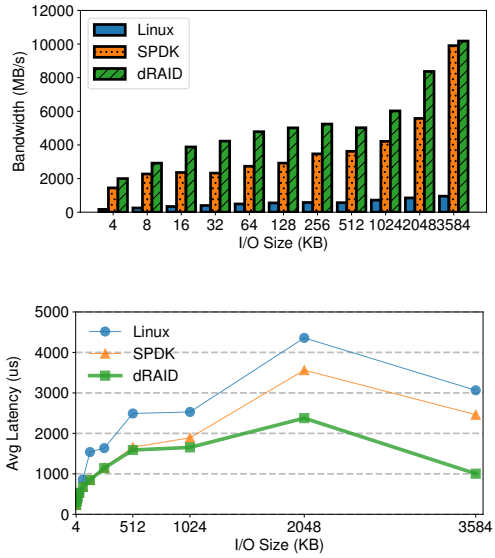


Figure 10: RAID-5 write on different I/O sizes

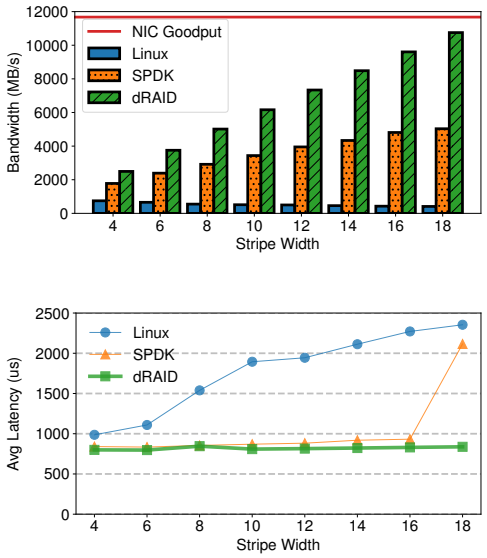


Figure 12: RAID-5 write on different stripe widths

9 EVALUATION

9.1 Experiment Methodology

Testbed setup. Our testbed comprises 19 RDMA-capable x86 servers with NVMe SSDs equipped on CloudLab [29]. Each server has an 24-core AMD EPYC 7402P processor, 128 GB memory, two Dell Ent NVMe AGN MU U.2 1.6 TB SSDs, a Mellanox ConnectX-5 Ex 100 Gbps NIC, and a Mellanox ConnectX-5 25 Gbps NIC. All servers are connected to a Dell Z9264 switch. We run CentOS Stream release 8 on all nodes with the Linux kernel version 4.18.0-358. The SPDK version that we use is 21.10.

Workloads and evaluation metrics. We use FIO [15] to evaluate performance of a raw RAID block device. All workloads are random accesses to the block device. We use YCSB [27] for the application evaluation. For each benchmark, we evaluate both throughput and latency while focusing on throughput. For each metric, we evaluate both RAID-5 and RAID-6. We use 128 KB I/O size, 512 KB chunk size (Linux software RAID default value), and 8 remote targets as the default setting unless otherwise specified. We only show the RAID-5 results in this section. The RAID-6 results are included in Appendix A, where dRAID benefits further from the offloaded parity calculation to storage servers.

Comparison schemes. We compare dRAID to Linux software RAID (MD driver) and the RAID POC for SPDK. We limit both dRAID and SPDK to use *one* core on each remote target.

State-of-the-art RAID system FusionRAID [36] only optimizes for latency reduction. Due to the imperfect implementation of FusionRAID, we are unable to run FIO against it. We confirm with the authors that FusionRAID is *not* expected to outperform Linux software RAID in terms of throughput. IODA [45] also targets tail latency reduction and requires device-level modifications, which does not apply to commodity SSDs.

Our initial experiment with Linux software RAID shows that its performance is nowhere near the theoretical bound. So we turn to the SPDK RAID-5 POC implemented by Intel developers [19], which is the best RAID-5 implementation in terms of performance to the best of our knowledge. It is implemented in user space, and has very limited use of locks. To make a fair and comprehensive comparison, we further enhance it by integrating ISA-L [18] and adding RAID-6 functionality to it.

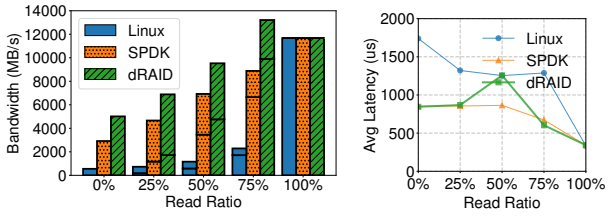


Figure 13: RAID-5 write on different read/write ratios

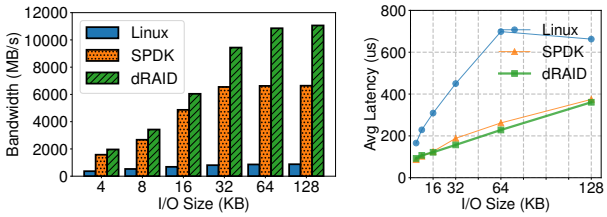


Figure 15: RAID-5 degraded read on different I/O sizes

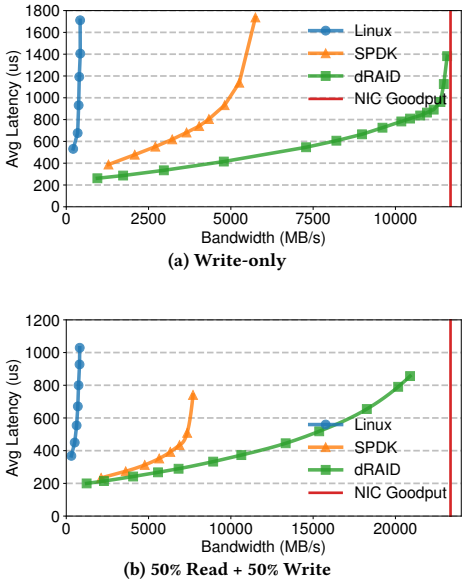


Figure 14: RAID-5 latency vs. bandwidth

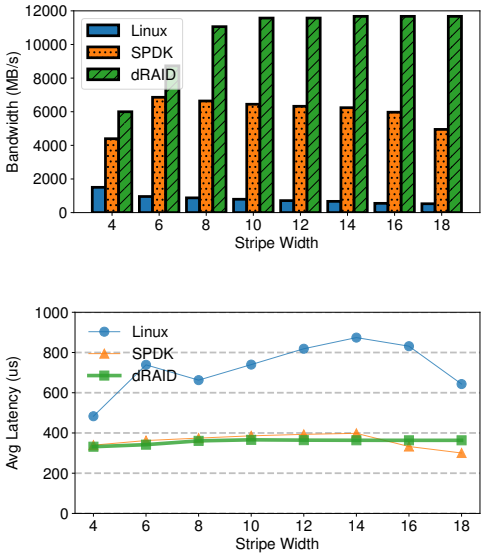


Figure 16: RAID-5 degraded read on different stripe widths

9.2 Normal-State Read

We first evaluate the read performance in normal state. We use six remote storage targets which are more than enough to saturate bandwidth of the 100 Gbps NIC. In Figure 9, we compare bandwidth of all three systems under similar latency for each I/O size. All systems reach the NIC goodput (~92 Gbps) beyond 64 KB I/O size. This is expected because all three systems are implemented in a similar way and read operation incurs extremely low overhead. dRAID shows a significant performance gain on small I/O sizes attributed to the lock-free implementation of normal read in dRAID.

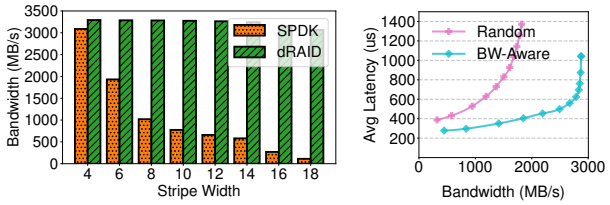
9.3 Normal-State Write

We next evaluate normal-state write performance. Using eight remote targets is not an optimal setup for dRAID as they can only provide roughly 5000 MB/s bandwidth for small writes. Prior work did a large-scale empirical study and showed that more than 80% RAID arrays in production consist of eight or more drives [31]. Therefore, the results should be seen as a *lower bound* of the improvement for a common RAID setup.

Impact of I/O size. dRAID targets partial stripe write. The size of a stripe with eight drives and 512 KB chunk size is 3584 KB for RAID-5. We evaluate full stripe write, reconstruct write, and read-modify-write. For RAID-5, I/O sizes below 1536 KB trigger read-modify-write, reconstruct write corresponds to the range between 1536 KB and 3584 KB, and a 3584 KB write triggers full stripe write.

In Figure 10, dRAID shows 1.7× throughput improvement at 128 KB. The performance does not improve further between 256 KB and 1024 KB, because dRAID reaches the maximum bandwidth that eight SSDs can provide on read-modify-write. Reconstruct write is triggered at 2048 KB and dRAID shows 1.5× throughput improvement. This is expected because the overhead of reconstruct write is less than that of read-modify-write but greater than that of full stripe write. Throughput of dRAID and SPDK is about the same at 3584 KB because they handle full stripe write in the same way.

Impact of chunk size. Chunk size has an opposite effect to I/O size. Chunk sizes that are small enough result in full stripe write for most I/Os. We evaluate chunk sizes between 32 KB and 1024 KB, which are common settings in practice. Figure 11 shows the results. dRAID



(a) Reconstruction scalability (b) Different algorithms

Figure 17: Reconstruction performance

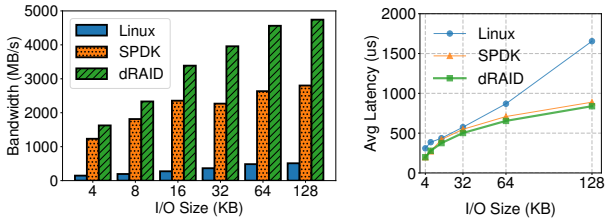


Figure 18: RAID-5 degraded-state write on different I/O sizes

runs at full drive bandwidth with chunk sizes between 128 KB and 1024 KB and provides up to 1.7× throughput improvement.

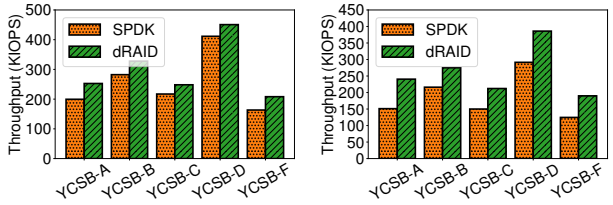
Scalability on stripe width. It is important for RAID to scale out because most production RAID arrays consist of more than eight drives. In Figure 12, Linux software RAID shows an opposite trend due to high overhead. Both SPDK and dRAID can scale out with small stripe widths. However, the best throughput SPDK can achieve is half of the NIC goodput (~46 Gbps), because one SPDK read-modify-write triggers two underlying writes.

In contrast, dRAID shows *linear* scalability from 4 to 18 remote targets, and achieves 84 Gbps with 18 remote targets. It can scale further towards the NIC goodput. We do not evaluate larger stripe widths, as our testbed only has 19 servers.

Impact of read/write ratio. We also evaluate different read/write ratios. In Figure 13, dRAID shows 1.4×-1.7× throughput improvement for all read/write ratios except for read-only workload. With only eight remote targets, the bottleneck is shifted to the drive bandwidth, so dRAID is heavily congested with 50% write, which leads to higher latency. We show that dRAID can provide further improvement on read/write mix workload than on write-only workload with a greater stripe width in the next evaluation.

Latency vs. throughput. We evaluate how latency changes as we increase the load. We use 18 remote targets in order to maximize the performance. We increase the load by adding on-the-fly I/Os gradually. Figure 14a illustrates the relationship between throughput and latency under write-only load. Theoretically, the maximum throughput of dRAID is ~92 Gbps, while SPDK can only reach half of it. Both dRAID and SPDK operate at their theoretical bound.

We also evaluate 50% write workload. Theoretically, dRAID can reach ~92 Gbps for both read and write, while SPDK cannot surpass



(a) Normal-state (b) Degraded-state

Figure 19: RocksDB throughput (KIOPS) on RAID-5

~31 Gbps. In Figure 14b, dRAID shows up to 3× throughput improvement. The maximum throughput dRAID provides is close to the NIC goodput. Note that dRAID may further approach the upper bound if more drives are added, because 18 remote SSDs cannot saturate ~92 Gbps read-modify-write with a reasonable latency.

9.4 Degraded-State Read

Impact of I/O size. Note that only one out of eight I/Os triggers reconstruction on a degraded array of stripe width 8. Figure 15 compares the performance of all three systems. Linux software RAID can only reach 834 MB/s. dRAID reaches 95% of normal-state read throughput, while SPDK achieves only 57% of that.

Scalability on stripe width. In Figure 16, the throughput of Linux software RAID gets worse with larger stripe widths. SPDK reaches its maximum throughput at stripe width 6, and becomes worse as stripe width increases. On the contrary, dRAID throughput approaches normal-state read performance as more remote targets are added. The throughput improvement achieved by dRAID is up to 2.4×. Latency of dRAID becomes higher than SPDK at stripe width 16 because dRAID runs at a significantly higher throughout. In Figure 17a, all reads are degraded when reconstructing a drive, dRAID also shows a near-optimal throughput for all stripe widths.

Impact of network heterogeneity. We also evaluate dRAID performance over remote storage targets with heterogeneous network to verify the effectiveness of our bandwidth-aware reconstruction algorithm. We use a mix of 25 Gbps NICs (enough to saturate the read bandwidth of a single SSD) and 100 Gbps NICs. As Figure 17b illustrates, bandwidth-aware reconstruction algorithm improves read bandwidth by 53% comparing to random reducer selection.

9.5 Degraded-State Write

Although handling degraded-state write is much more complex than normal-state write, minor performance degradation is expected because only one out of eight I/Os triggers data reconstruction of the failed drive. Figure 18 shows that performance of all three systems drops around 5% comparing to normal state. dRAID still outperforms SPDK by up to 1.7×.

9.6 Application Performance

RocksDB performance. RocksDB [28] runs atop a user-space filesystem BlobFS implemented in SPDK. We run a single instance

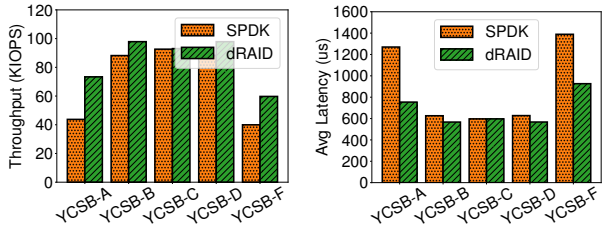


Figure 20: Object store on normal-state RAID-5

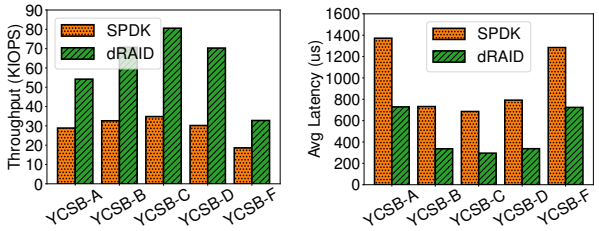


Figure 21: Object store on degraded-state RAID-5

of RocksDB which BlobFS only supports. Figure 19a shows normal-state performance. YCSB-A and YCSB-F have the highest write proportion, and dRAID improves their IOPS by 1.27× and 1.28× respectively. Further improvement can be observed for all workloads in degraded state in Figure 19b.

dRAID provides a lower speedup on RocksDB than on a raw RAID array, because RocksDB and BlobFS incur additional overhead by implementing complex data structures and using locks, resulting in limited throughput of a single instance [48]. Indeed, super-blocks in BlobFS are accessed more frequently than other segments on the array. In most cases, less than 5% of the total bandwidth is utilized by a RocksDB instance. The results indicate bypassing BlobFS and co-optimizing KV store can further boost the performance.

Object store performance. To further evaluate dRAID performance under high throughput, we implement a hash-based object store that runs directly on the block device layer. We tune YCSB to generate 200 K 128 KB objects and make 200 K requests in total. We set the distribution to uniform so that the maximum throughput of the object store can be observed without loading too many objects. (*i*) Figure 20 shows IOPS and latency of each workload under normal state. dRAID achieves 1.7× and 1.5× improvement on YCSB-A and YCSB-F respectively. Very limited improvement on read-heavy workloads (YCSB-B/C/D) is expected because dRAID focuses on write optimization. (*ii*) Figure 21 shows IOPS and latency of each workload under degraded state. dRAID further improves the throughput, especially for the read-heavy workloads. dRAID outperforms SPDK by ~2.35× on YCSB-B/C/D.

10 RELATED WORK

RAID optimization. RAID optimization has been an extensively studied area. Partial stripe write is identified as the major performance weakness of parity-based RAID. Logging is a classic technique to shape write I/Os to RAID-friendly I/Os [56, 59]. Purity [26], Flash-Aware RAID [35], and PPC [25] leverage non-volatile memory to stage writes and shape them as full stripe writes.

New challenges also arise on novel storage media. SWAN [40] and GGC [41] alleviate impact of SSD garbage collection by coordinating GC at array level. FusionRAID [36], ToleRAID [31], and TTFLASH [60] mitigate latency spike by detouring from SSDs under GC. IODA [45] exploits a recent NVMe IOD interface for predictable SSD array performance. RAID layout is also evolving for faster recovery [61] and further space saving [37, 38].

A highly related system is TickerTAIP [23]. Both TickerTAIP and dRAID calculate parity in a distributed manner. Most of the performance gain of TickerTAIP comes from parallel parity calculation. After three decades, today's CPUs can handle parity calculation rather efficiently [17]. dRAID instead optimizes for disaggregated storage architecture and modern datacenter hardware with bandwidth-optimal data movement and asynchronous I/O pipeline.

Disaggregated storage. NVMe-oF [4] is one of the fastest growing disaggregated storage options. Recent research focuses on QOS guarantee of disaggregated storage. ReFlex [43] designs a credit-based QOS scheduler, and Gimbal [48] applies a similar approach to SmartNIC JBOFs. i10 [33] and blk-switch [34] design an in-kernel remote storage stack which runs over TCP network and can achieve comparable performance to RDMA-based solutions. Unlike previous work, dRAID does not optimize the performance of a standalone remote storage target, but rather a redundant array of them.

11 CONCLUSION

We presented dRAID, a software-based disaggregated RAID storage system that achieves linear scalability within NIC bandwidth. dRAID leverages the unique opportunities of disaggregated storage to accelerate RAID storage by optimizing the data path of parity generation and delivery. We believe that dRAID exemplifies a new generation of parity-based storage arrays backed by disaggregated storage. This work does not raise any ethical issues.

ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program of China under the grant number 2020YFB2104100, the National Natural Science Foundation of China under the grant number 62172008, the National Natural Science Fund for the Excellent Young Scientists Fund Program (Overseas), and the Beijing Outstanding Young Scientist Program under the grant number BJJWZYJH01201910001004.

We sincerely thank our shepherd Song Jiang and the anonymous reviewers for their valuable feedback on this paper. We acknowledge Qingchun Song and Pengzhi Zhu from NVIDIA Networking HPC-AI Lab for providing development environment and information on NVIDIA hardware. Junyi Shu, Ruidong Zhu, Yun Ma, Gang Huang, Hong Mei, Xuanzhe Liu, and Xin Jin are with the Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education. Xin Jin and Gang Huang are the corresponding authors.

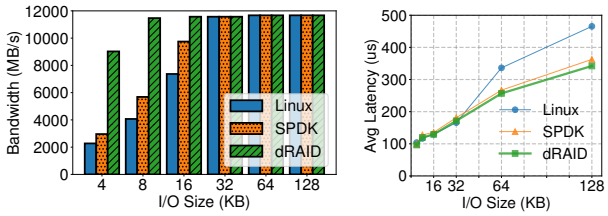


Figure 22: RAID-6 normal-state read on different I/O sizes

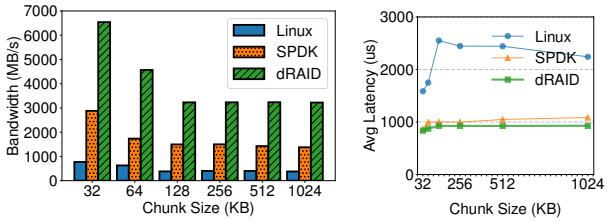


Figure 24: RAID-6 write on different chunk sizes

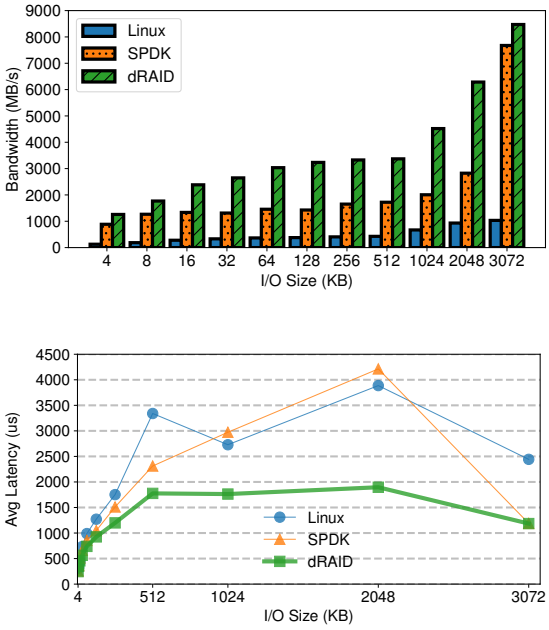


Figure 23: RAID-6 write on different I/O sizes

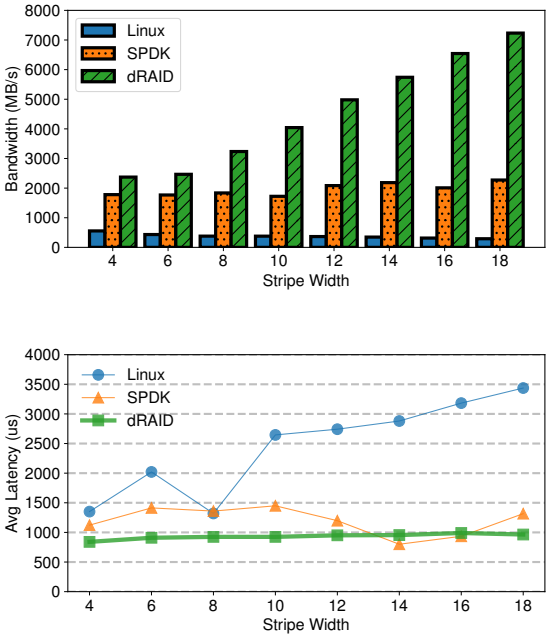


Figure 25: RAID-6 write on different stripe widths

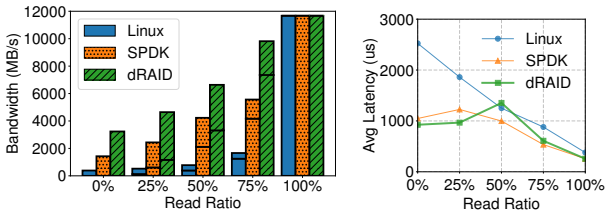


Figure 26: RAID-6 write on different read/write ratios

A RAID-6 EVALUATION

We also implement dRAID support for RAID-6 and conduct the evaluation using FIO. We use the same settings as for the RAID-5 evaluation. Overall, the results match our expectation, and almost the same conclusions can be drawn.

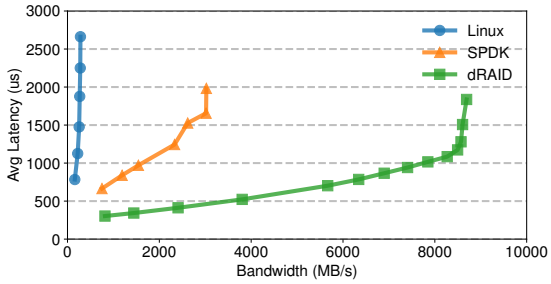
A.1 Normal-State Read

In Figure 22, all systems achieve the NIC goodput (~92 Gbps) beyond 64 KB I/O size. the results are almost identical to RAID-5, because the distributed parity layout of RAID-6 also allows a RAID array to fully utilize the bandwidth of all drives.

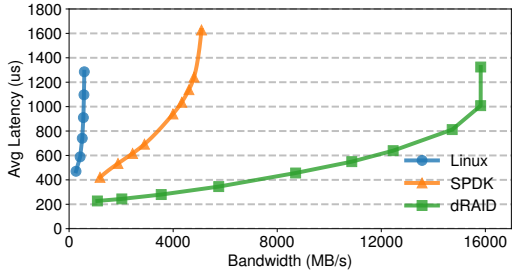
A.2 Normal-State Write

Impact of I/O size. The size of a stripe with eight drives and 512 KB chunk size is 3072 KB for RAID-6, while the stripe size of RAID-5 is 3584 KB. As Figure 23 shows, dRAID improves throughput by 2.3× at 128 KB. The extra improvement is because dRAID saves an additional outbound network I/O to the second parity in RAID-6. Overall, RAID-6 throughput is about two thirds of that of RAID-5 on small writes. This matches our expectation as RAID-5 triggers four drive I/Os while RAID-6 triggers six drive I/Os.

Impact of chunk size. In Figure 24, the performance difference between SPDK and dRAID is also more significant than RAID-5.



(a) Write-only



(b) 50% Read + 50% Write

Figure 27: RAID-6 latency vs. bandwidth

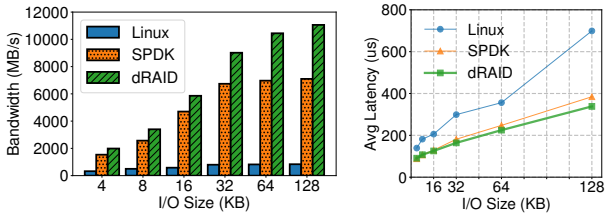


Figure 28: RAID-6 degraded read on different I/O sizes

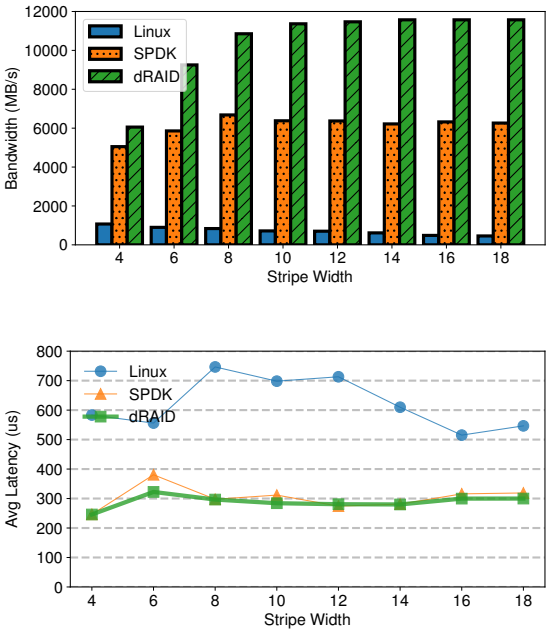


Figure 29: RAID-6 degraded read on different stripe widths

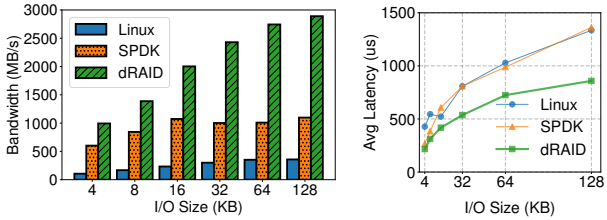


Figure 30: RAID-6 degraded-state write on different I/O sizes

Note that dRAID improves throughput by 2.6× for small chunk sizes. SPDK suffers a higher overhead from RAID-6 when parity calculation is mostly on the host side.

Scalability on stripe width. In Figure 25, SPDK can hardly scale out and has an unsteady performance on RAID-6. dRAID shows near-linearly scalability and consistently low latency on all stripe widths. It should scale further towards the NIC goodput with more servers added to the testbed.

Impact of read/write ratio. In Figure 26, dRAID shows 1.6×-2.3× throughput improvement for all read/write ratios except for read-only workload. The results are similar to those of RAID-5 except for slightly larger performance gap between SPDK and dRAID.

Latency vs. throughput. We also evaluate how latency changes as we increase the load. In Figure 27, given a certain bandwidth, dRAID consistently shows lower latency than SPDK for both WO and RW workload. Indeed, the maximum bandwidths that dRAID achieves on WO and RW workloads are 8692 MB/s and 15 822 MB/s respectively, which gives ~3× improvement over SPDK.

A.3 Degraded-State Read

Impact of I/O size. In Figure 28, the results are close to those of RAID-5. dRAID degraded-state throughput is about 95% of normal-state read, while SPDK is 61%. Reconstructing RAID-6 data does not incur too much additional overhead.

Impact of stripe width. As Figure 29 demonstrates, SPDK reaches its maximum throughput at stripe width 8, and the performance is slightly worse beyond that. dRAID can achieve stable performance with different stripe widths.

A.4 Degraded-State Write

In Figure 30, SPDK experiences 23% performance drop, while dRAID performance only drops by 11%. This leads to a 2.6× performance gap between SPDK and dRAID.

B ARTIFACT APPENDIX

B.1 Abstract

We provide the artifact for the ASPLOS 2023 paper "Disaggregated RAID Storage in Modern Datacenters", including the main implementation of dRAID, CloudLab [29] testbed setup scripts, FIO [15] experiment scripts (§9.2-§9.5), and YCSB [27] experiment scripts (§9.6).

All of the experiments must be conducted on real hardware and require 20 servers, each with an AMD EPYC processor, 128 GB memory, an unmounted enterprise-grade NVMe SSD, a Mellanox ConnectX-5 Ex 100 Gbps NIC. CloudLab Utah c6525-100g instances meet the requirement, and they are publicly accessible and free. To run the experiments, a customized version of SPDK [20] and all of its dependencies are required.

The experiment workflow is described in READMEs of our repository. Assuming the conditions of the NVMe SSDs have not changed significantly, the results in this paper should be reproducible on CloudLab testbed with <20% variation. The provided scripts have been verified only on CloudLab. If you wish to run the experiments on a different testbed, please adjust the scripts accordingly.

B.2 Artifact Check-List (Meta-information)

- **Program:** We use synthetic workloads generated by FIO and YCSB for evaluation. The provided scripts download the benchmark tools for you.
- **Compilation:** We use g++ 8.5 which is included in CloudLab OS Image for code compilation.
- **Run-time environment:** dRAID implementation is Linux-specific. The provided scripts are created for CentOS 8 and require root access. dRAID depends on SPDK and its dependencies.
- **Hardware:** x86 cores, enterprise-grade NVMe SSDs, and RDMA NICs are required. CloudLab Utah c6525-100g instances are publicly available for use.
- **Run-time state:** dRAID is sensitive to the SSDs conditions.
- **Execution:** Scripts or commands for execution are provided in our repository. Please refer to the READMEs.
- **Metrics:** The main metrics are bandwidth, latency and IOPS.
- **Output:** The provided scripts output standard FIO and YCSB statistics to stdout.
- **Experiments:** Scripts or commands for running the experiments are provided in our repository. <20% variation is expected.
- **How much disk space required (approximately)?:** <500 GB.
- **How much time is needed to prepare workflow (approximately)?:** <1 hour.
- **How much time is needed to complete experiments (approximately)?:** <24 hours.
- **Publicly available?:** The code is publicly available at https://github.com/pkusys/dRAID.
- **Code licenses:** Apache License 2.0.
- **Archived:** https://doi.org/10.5281/zenodo.7587687.

B.3 Description

B.3.1 How to Access. Clone the git repository at https://github.com/pkusys/dRAID. An archived copy is also available [54].

B.3.2 Hardware Dependencies. All of the experiments must be conducted on real hardware and require 20 servers, each with an AMD EPYC processor, 128 GB memory, an unmounted enterprise-grade NVMe SSD, a Mellanox ConnectX-5 Ex 100 Gbps NIC. The provided scripts can be executed on CloudLab Utah c6525-100g instances. You will need a CloudLab account to use them.

B.3.3 Software Dependencies. dRAID implementation is Linux-specific. And the provided scripts are created for CentOS 8. To run the experiments, a customized version of SPDK and all of its dependencies (e.g., DPDK, ISA-L, libibverbs +librdmacm) are required. The provided scripts will install them for you.

B.4 Installation

Please see the setup directory of our repository for installing dRAID on CloudLab testbed. You can find the basic test script under the dRAID directory of the repository. Please follow the steps in the corresponding README to run the basic test.

B.5 Experiment Workflow

The detailed workflow of setting up the environments and executing the experiments are described in the corresponding READMEs of our repository.

B.6 Evaluation and Expected Results

The results are output to stdout in the standard FIO/YCSB formats. Assuming the conditions of the NVMe SSDs have not changed significantly, the results in this paper should be reproducible on CloudLab testbed with <20% variation.

B.7 Experiment Customization

If you want to run the experiment with a different parameter (e.g., I/O size, I/O depth), you can modify the run.sh script of the corresponding experiment.

B.8 Notes

- Figure 17 requires a different testbed setup and a bit code hacking, which we do not include in this artifact.
- Ignore the warning "RPC client command timeout". It does not affect the experiments.
- Ignore the error message "io_device Raid0 not unregistered". We have not implemented graceful shutdown yet.
- If you see RDMA related errors or the experiment hangs for >2 minutes at the beginning, this is due to race conditions caused by imperfect implementation of the start-up process. You can safely kill and rerun the script.

REFERENCES

[1] 2003. Internet Small Computer Systems Interface (iSCSI). https://www.ietf.org/rfc/rfc3720.txt. Retrieved Aug 30, 2022.

[2] 2014. EMC XtremIO Data Protection (XDP). https://www.corporatearmor.com/documents/EMC_XtremIO_Data_Protection_Whitepaper.pdf. Retrieved Apr 15, 2022.

[3] 2016. Enterprise NVR Series SAS RAID Storage. https://www.security.honeywell.com/product-repository/enterprise-nvr-series-sas-raid-storage. Retrieved Aug 30, 2022.

[4] 2016. NVMe Over Fabrics. https://nvmexpress.org/wp-content/uploads/NVMe_Over_Fabrics.pdf. Retrieved Mar 31, 2022.

[5] 2018. Do you still set up RAID for SSD drives? https://www.reddit.com/r/sysadmin/comments/aavir0/do_you_still_set_up_raid_for_ssd_drives/. Retrieved Mar 15, 2022.

[6] 2018. Taking Advantage of a Disaggregated Storage and Compute Architecture. https://databricks.com/session/taking-advantage-of-a-disaggregated-storage-and-compute-architecture. Retrieved Mar 15, 2022.

[7] 2019. The Case for Disaggregated Storage. https://www.fungible.com/wp-content/uploads/2019/09/WP009.00.91020918-The-Case-for-Disaggregated-Storage.pdf. Retrieved Mar 15, 2022.

[8] 2019. What Are the Benefits of SSD RAID? https://insights.samsung.com/2019/03/21/what-are-the-benefits-of-ssd-raid/. Retrieved Mar 15, 2022.

[9] 2021. Benefits of Disaggregating NVMe Storage with NVMe-oF Technology. https://nvmexpress.org/benefits-of-disaggregating-nvme-storage-with-nvme-of-technology/. Retrieved Mar 15, 2022.

[10] 2021. HPE Smart Array S100i SR Gen10 Software RAID. https://www.hpe.com/psnow/doc/a00019427enw?jumpid=in_lit-psnow-red. Retrieved Apr 15, 2022.

[11] 2022. Amazon EBS volume types. https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/ebs-volume-types.html. Retrieved Mar 31, 2022.

[12] 2022. Azure Managed Disks. https://docs.microsoft.com/en-us/azure/virtual-machines/disks-types. Retrieved Apr 1, 2022.

[13] 2022. Dell PowerEdge RAID Controller 11 User's Guide. https://www.dell.com/support/manuals/en-hk/poweredge-r7525/perc11_ug/technical-specifications-of-perc-11-cards?guid=guid-aaaf8b59-903f-49c1-8832-f3997d125edf. Retrieved Aug 30, 2022.

[14] 2022. Dell PowerStore Specifications. https://www.dell.com/en-us/dt/storage/powerstore-storage-appliance/powerstore-t-series.htm. Retrieved Jan 26, 2023.

[15] 2022. Flexible I/O Tester. https://github.com/axboe/fio. Retrieved Apr 1, 2022.

[16] 2022. InfiniBand Roadmap. https://www.infinibandta.org/infiniband-roadmap/. Retrieved Mar 15, 2022.

[17] 2022. Intel Rapid Storage Technology enterprise: Product Brief. https://www.intel.com/content/www/us/en/architecture-and-technology/rapid-storage-technology-enterprise-brief.html. Retrieved Aug 30, 2022.

[18] 2022. Intelligent Storage Acceleration Library. https://github.com/intel/isa-l. Retrieved Apr 1, 2022.

[19] 2022. SPDK RAID-5 POC. https://github.com/apaszkie/spdk/blob/raid5_poc/lib/bdev/collections/raid5.c. Retrieved Apr 1, 2022.

[20] 2022. Storage Performance Development Kit. https://spdk.io/. Retrieved Mar 31, 2022.

[21] Dave Anderson and Jim Dykes. 2003. More Than an Interface—SCSI vs. ATA. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*. 245–257.

[22] H Peter Anvin. 2007. The mathematics of RAID-6. http://ftp.dei.uc.pt/pub/linux/kernel/people/hpa/raid6.pdf. Retrieved Mar 31, 2022.

[23] Pei Cao, Swee Boon Lim, Shivakumar Venkataraman, and John Wilkes. 1993. The TickerTAIP Parallel RAID Architecture. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. 52–63. https://doi.org/10.1145/165123.165130

[24] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. 1994. RAID: High-Performance, Reliable Secondary Storage. *ACM Comput. Surv.* 26, 2 (1994), 145–185. https://doi.org/10.1145/176979.176981

[25] Ching-Che Chung and Hao-Hsiang Hsu. 2014. 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* 22, 7 (2014), 1470–1480. https://doi.org/10.1109/TVLSI.2013.2275737

[26] John Colgrove, John D. Davis, John Hayes, Ethan L. Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. 2015. Purity: Building Fast, Highly-Available Enterprise Flash Storage from Commodity Components. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 1683–1694. https://doi.org/10.1145/2723372.2742798

[27] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*. 143–154. https://doi.org/10.1145/1807128.1807152

[28] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-Value Store Serving Large-Scale Applications. *ACM Trans. Storage* 17, 4, Article 26 (2021), 32 pages. https://doi.org/10.1145/3483840

[29] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, Aditya Akella, Kuangching Wang, Glenn Ricart, Larry Landweber, Chip Elliott, Michael Zink, Emmanuel Cecchet, Snigdhaswin Kar, and Prabodh Mishra. 2019. The Design and Operation of CloudLab. In *Proceedings of the 2019 USENIX Annual Technical Conference*. 1–14.

[30] Dan Gibson, Hema Hariharan, Eric Lance, Moray McLaren, Behnam Montazeri, Arjun Singh, Stephen Wang, Hassan M. G. Wassel, Zhehua Wu, Sunghwan Yoo, Raghuraman Balasubramanian, Prashant Chandra, Michael Cutforth, Peter Cuy, David Decotigny, Rakesh Gautam, Alex Iriza, Milo M. K. Martin, Rick Roy, Zuowei Shen, Ming Tan, Ye Tang, Monica Wong-Chan, Joe Zbiciak, and Amin Vahdat. 2022. Aquila: A unified, low-latency fabric for datacenter networks. In *Proceedings of the 19th USENIX Symposium on Networked Systems Design and Implementation*. 1249–1266.

[31] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A. Chien, and Haryadi S. Gunawi. 2016. The Tail at Store: A Revelation from Millions of Hours of Disk and SSD Deployments. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies*. 263–276.

[32] Tejun Heo, Dan Schatzberg, Andrew Newell, Song Liu, Saravanan Dhakshinamurthy, Iyswarya Narayanan, Josef Bacik, Chris Mason, Chunqiang Tang, and Dimitrios Skarlatos. 2022. IOCost: Block IO Control for Containers in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 595–608. https://doi.org/10.1145/3503222.3507727

[33] Jaehyun Hwang, Qizhe Cai, Ao Tang, and Rachit Agarwal. 2020. TCP ≈ RDMA: CPU-efficient Remote Storage Access with i10. In *Proceedings of the 17th USENIX Conference on Networked Systems Design and Implementation*. 127–140.

[34] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. 2021. Rearchitecting Linux Storage Stack for µs Latency and High Throughput. In *Proceedings of the 15th USENIX Conference on Operating Systems Design and Implementation*. 113–128.

[35] Soojun Im and Dongkun Shin. 2011. Flash-Aware RAID Techniques for Dependable and High-Performance Flash Memory SSD. *IEEE Trans. Comput.* 60, 1 (2011), 80–92. https://doi.org/10.1109/TC.2010.197

[36] Tianyang Jiang, Guangyan Zhang, Zican Huang, Xiaosong Ma, Junyu Wei, Zhiyue Li, and Weimin Zheng. 2021. FusionRAID: Achieving Consistent Low Latency for Commodity SSD Arrays. In *Proceedings of the 19th USENIX Conference on File and Storage Technologies*. 355–370.

[37] Saurabh Kadekodi, Francisco Maturana, Sanjith Athlur, Arif Merchant, K. V. Rashmi, and Gregory R. Ganger. 2022. Tiger: Disk-Adaptive Redundancy Without Placement Restrictions. In *Proceedings of the 16th USENIX Conference on Operating Systems Design and Implementation*. 413–429.

[38] Saurabh Kadekodi, Francisco Maturana, Suhas Jayaram Subramanya, Juncheng Yang, K. V. Rashmi, and Gregory R. Ganger. 2020. PACEMAKER: Avoiding HeART attacks in storage clusters with disk-adaptive redundancy. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. Article 21, 17 pages.

[39] Saurabh Kadekodi, K. V. Rashmi, and Gregory R. Ganger. 2019. Cluster storage systems gotta have HeART: improving storage efficiency by exploiting disk-reliability heterogeneity. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*. 345–358.

[40] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H. Noh. 2019. Alleviating Garbage Collection Interference Through Spatial Separation in All Flash Arrays. In *Proceedings of the 2019 USENIX Annual Technical Conference*. 799–812.

[41] Youngjae Kim, Junghee Lee, Sarp Oral, David A. Dillow, Feiyi Wang, and Galen M. Shipman. 2014. Coordinating Garbage Collectionfor Arrays of Solid-State Drives. *IEEE Trans. Comput.* 63, 4 (2014), 888–901. https://doi.org/10.1109/TC.2012.256

[42] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. 2016. Flash Storage Disaggregation. In *Proceedings of the 11th European Conference on Computer Systems*. Article 29, 15 pages. https://doi.org/10.1145/2901318.2901337

[43] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. ReFlex: Remote Flash ≈ Local Flash. In *Proceedings of the 22TH International Conference on Architectural Support for Programming Languages and Operating Systems*. 345–359. https://doi.org/10.1145/3037697.3037732

[44] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan R. K. Ports, Irene Zhang, Ricardo Bianchini, Haryadi S. Gunawi, and Anirudh Badam. 2020. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of the 35th International Conference on Architectural Support for Programming Languages and Operating Systems*. 591–605. https://doi.org/10.1145/3373376.3378531

[45] Huaicheng Li, Martin L. Putra, Ronald Shi, Xing Lin, Gregory R. Ganger, and Haryadi S. Gunawi. 2021. IODA: A Host/Device Co-Design for Strong Predictability Contract on Modern Flash Storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 263–279. https://doi.org/10.1145/3477132.3483573

[46] Jai Menon and Jim Cortney. 1993. The Architecture of a Fault-Tolerant Cached RAID Controller. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. 76–87. https://doi.org/10.1145/165123.165144

[47] Jai Menon, James Roche, and Jim Kasson. 1993. Floating parity and data disk arrays. *J. Parallel and Distrib. Comput.* 17, 1 (1993), 129–139. https://doi.org/10.1006/jpdc.1993.1011

[48] Jaehong Min, Ming Liu, Tapan Chugh, Chenxingyu Zhao, Andrew Wei, In Hwan Doh, and Arvind Krishnamurthy. 2021. Gimbal: Enabling Multi-Tenant Storage Disaggregation on SmartNIC JBOFs. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*. 106–122. https://doi.org/10.1145/3452296.3472940

[49] Sumit Kumar Monga, Sanidhya Kashyap, and Changwoo Min. 2021. Birds of a Feather Flock Together: Scaling RDMA RPCs with Flock. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 212–227. https://doi.org/10.1145/3477132.3483576

[50] David A. Patterson, Garth Gibson, and Randy H. Katz. 1988. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the 1988 ACM SIGMOD International Conference on Management of Data*. 109–116. https://doi.org/10.1145/50202.50214

[51] James S. Plank, Jianqiang Luo, Catherine D. Schuman, Lihao Xu, and Zooko Wilcox-O'Hearn. 2009. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. In *Proccedings of the 7th Conference on File and Storage Technologies*. 253–265.

[52] Leon Poutievski, Omid Mashayekhi, Joon Ong, Arjun Singh, Mukarram Tariq, Rui Wang, Jianan Zhang, Virginia Beauregard, Patrick Conner, Steve Gribble, Rishi Kapoor, Stephen Kratzer, Nanfang Li, Hong Liu, Karthik Nagaraj, Jason Ornstein, Samir Sawhney, Ryohei Urata, Lorenzo Vicisano, Kevin Yasumura, Shidong Zhang, Junlan Zhou, and Amin Vahdat. 2022. Jupiter Evolving: Transforming Google's Datacenter Network via Optical Circuit Switches and Software-Defined Networking. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 66–85. https://doi.org/10.1145/3544216.3544265

[53] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What Serverless Computing is and Should Become: The next Phase of Cloud Computing. *Commun. ACM* 64, 5 (2021), 76–84. https://doi.org/10.1145/3406011

[54] Junyi Shu, Ruidong Zhu, Yun Ma, Gang Huang, Hong Mei, Xuanzhe Liu, and Xin Jin. 2023. *dRAID artifacts*. https://doi.org/10.5281/zenodo.7587687

[55] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, Anand Kanagala, Jeff Provost, Jason Simmons, Eiichi Tanda, Jim Wanderer, Urs Hölzle, Stephen Stuart, and Amin Vahdat. 2015. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. 183–197. https://doi.org/10.1145/2785956.2787508

[56] Daniel Stodolsky, Garth Gibson, and Mark Holland. 1993. Parity Logging Overcoming the Small Write Problem in Redundant Disk Arrays. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. 64–75. https://doi.org/10.1145/165123.165143

[57] Nisha Talagala, Satoshi Asami, Tom Anderson, and David Patterson. 1997. *Tertiary Disk: Large Scale Distributed Storage*. Technical Report UCB/CSD-98-989.

[58] Jon Tate, Pall Beck, Hector Hugo Ibarra, Shanmuganathan Kumaravel, Libor Miklas, et al. 2018. *Introduction to storage area networks*. IBM Redbooks.

[59] John Wilkes, Richard Golding, Carl Staelin, and Tim Sullivan. 1996. The HP AutoRAID Hierarchical Storage System. *ACM Trans. Comput. Syst.* 14, 1 (1996), 108–136. https://doi.org/10.1145/225535.225539

[60] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A. Chien, and Haryadi S. Gunawi. 2017. Tiny-Tail Flash: Near-Perfect Elimination of Garbage Collection Tail Latencies in NAND SSDs. *ACM Trans. Storage* 13, 3, Article 22 (2017), 26 pages. https://doi.org/10.1145/3121133

[61] Guangyan Zhang, Zican Huang, Xiaosong Ma, Songlin Yang, Zhufan Wang, and Weimin Zheng. 2018. RAID+: Deterministic and Balanced Data Distribution for Large Disk Enclosures. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*. 279–293.

Received 2022-10-20; accepted 2023-01-19