RAIZN: Redundant Array of Independent Zoned Namespaces

Thomas Kim
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Jekyeom Jeon
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Nikhil Arora
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Huaicheng Li
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Michael Kaminsky*
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

David G. Andersen*
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Gregory R. Ganger
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

George Amvrosiadis
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Matias Bjørling
Western Digital Corporation
Copenhagen, Denmark

ABSTRACT

Zoned Namespace (ZNS) SSDs are the latest evolution of host-managed flash storage, enabling improved performance at a lower cost-per-byte than traditional block interface (conventional) SSDs. To date, there is no support for arranging these new devices in arrays that offer increased throughput and reliability (RAID). We identify key challenges in designing redundant ZNS SSD arrays, such as managing metadata updates and persisting partial stripe writes in the absence of overwrite support from the device.

We present RAIZN, a logical volume manager that exposes a ZNS interface and stripes data and parity across ZNS SSDs. RAIZN provides more stable throughput and lower tail latencies than an `mdraid` array of conventional SSDs based on the same hardware platform. RAIZN achieves superior performance because device-level garbage collection slows down conventional SSDs. We confirm that the benefits of RAIZN translate to higher layers by adapting the F2FS file system, RocksDB key-value store, and MySQL database to work with ZNS and leverage its benefits by closely controlling garbage collection. Compared to arrays of conventional SSDs experiencing on-device garbage collection, RAIZN leverages the ZNS interface to maintain consistent performance with up to 14× higher throughput and lower tail latency.

CCS CONCEPTS

• **Computer systems organization** → **Reliability**; **Availability**.

KEYWORDS

Zoned namespaces, storage, RAID, reliability

ACM Reference Format:
Thomas Kim, Jekyeom Jeon, Nikhil Arora, Huaicheng Li, Michael Kaminsky, David G. Andersen, Gregory R. Ganger, George Amvrosiadis, Matias Bjørling. 2023. RAIZN: Redundant Array of Independent Zoned Namespaces. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3575693.3575746

*Also with Enriched Ag.

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

ACM ISBN 978-1-4503-9916-6/23/03.
https://doi.org/10.1145/3575693.3575746

1 INTRODUCTION

The NVMe Zoned Namespace (ZNS) standard [2, 6] is a new alternative to the block interface that has been the de-facto storage device interface for decades. ZNS shifts responsibilities such as flash page-level logical block address (LBA) mapping and garbage collection from the SSD's flash translation layer (FTL) to the host, providing an interface that is better matched to the characteristics of the underlying flash media. ZNS allows host and device to collaborate on data placement, so applications can improve performance through tighter control of device-level garbage collection while avoiding the performance fluctuations triggered by on-device garbage collection-related performance fluctuations experienced by conventional FTL-based SSDs [33].

The benefits of ZNS are not free, as the ZNS interface enforces stricter write semantics. ZNS devices divide their address space into large contiguous *zones*, each of which must be written sequentially and reset as a single unit [10]. Prior work compares ZNS to conventional SSDs, showing that in single-device applications [6], applications specialized to run on ZNS SSDs can achieve a 90% reduction in 99.99th-percentile tail latency and 2× higher write throughput compared to the equivalent workload on conventional SSDs. ZNS SSDs also require less on-device DRAM and overprovisioned flash blocks than conventional SSDs. In datacenters, devices are deployed in arrays, thus calling for software support for data striping and redundancy over ZNS device arrays. A standard solution for managing disk arrays is RAID [28], which aggregates devices to deliver higher performance and reliability.

We present RAIZN (Redundant Array of Independent Zoned Namespaces), which adapts RAID-like mechanisms for use with ZNS SSDs. We implement RAIZN as a Linux device mapper (`dm` [1]) logical volume that configures ZNS devices in an array and exposes a single ZNS device to the host, transparently providing redundancy and striping. RAID functionality has been extended to emerging storage technologies such as KVSSDs [29], but this is the first work targeting ZNS SSDs. RAIZN behaves like software RAID—the logical volume accepts IO from the host application, arranges data into

parity coded stripes, and distributes it across the underlying devices. RAIZN's novelty stems from exposing a logical ZNS volume operating atop physical ZNS devices that do not support overwrites.

There are two key challenges in designing RAIZN: the immutability of data written to zones, and the lack of atomicity when writing to multiple physical devices. The former is a problem introduced by the zoned interface on the SSDs, while the latter is a problem that exists in any system that persists data on an array of devices. Immutability poses a challenge in handling small writes, as parity cannot be updated after it is written. The lack of atomicity results in subtle problems when combined with the zoned interface. A side effect of append-only zones is that data is guaranteed to be persisted in sequential order; data at a particular LBA cannot be reported as persisted until data at the preceding LBAs is persisted. Conforming to this guarantee in RAIZN without hurting performance is complicated by the immutability of data once it has been written to the underlying devices. For example, if a stripe of data is only persisted to a subset of the devices before the system loses power, RAIZN cannot naively allow reading of the data, nor can it overwrite the data that has already been written. This paper describes how RAIZN solves these challenges and achieves high performance while tolerating power and device failures.

We compare the performance of RAIZN on ZNS devices to that of conventional software RAID (`mdraid` [4]) on near-identical conventional SSDs. RAIZN provides RAID-like reliability on ZNS devices while achieving maximum read and write throughput within 2% of the aggregate raw device throughput. RAIZN achieves similar throughput, median, tail latency, degraded throughput, degraded latency, and device rebuild throughput to `mdraid` when the conventional devices are not suffering from garbage collection. In addition, RAIZN leverages the ZNS interface to minimize time spent rebuilding a failed and replaced device, achieving shorter time to repair compared to `mdraid` when the array is not 100% filled. In our evaluation, we found that on-device garbage collection can reduce throughput by up to 93% and increase tail latency by 14× in `mdraid`, while RAIZN is not affected due to the absence of on-device garbage collection (Section 6.1). The evaluation demonstrates that RAIZN provides comparable performance to `mdraid` on synthetic microbenchmarks and for applications like RocksDB and MySQL.

RAIZN is open-sourced and can be accessed at https://github.com/ZonedStorage/RAIZN-release.

2 BACKGROUND

This section describes the ZNS interface and ZNS SSDs, highlighting features that create challenges or opportunities for RAIZN's design.

2.1 The ZNS Interface

The traditional block interface allows for atomic reads, writes, and overwrites of regions in a storage device's address space. There are several differences between this block interface and ZNS:

Sequential write-only zones. Under ZNS, blocks in the device's address space space are grouped into *zones* that must be written sequentially, and sectors cannot be overwritten unless the entire zone is erased by a *zone reset* operation. The address space of zoned devices is exposed using these zones, and applications are expected to structure their IO in a way that conforms with the sequential write constraint. Each zone maintains a *write pointer* which can be queried by an application to identify the next writable block address in that zone. Multiple writes can be submitted concurrently to the same zone provided the host submits them in sequential order.

Zone append. In addition to typical write commands, the ZNS standard defines the *zone append* command, where the host specifies the zone to write data to, and upon IO completion receives the block address at which the data was written. Zone append allows the host to submit multiple writes to a single zone without constraints on the write order, but they are not guaranteed to be laid out in the address space in the order they were submitted.

ZNS state machine. Each zone has an associated status describing the current *state* of the zone [2]. We briefly outline the states that are important to our system. A zone starts in the *empty* state, and returns to it after a reset. When a zone is written to, it transitions into the *open* state. Each device has a model-specific limit on the number of simultaneous open zones, which for our devices is 14. An open zone becomes *full* when the last writable block in that zone is written. Finally, *read-only* and *offline* are failure states, transitioned to when enough erase blocks fail that a zone cannot be fully used anymore. Typically, zones will only fail after the device has reached the end of its life, as the firmware handles media wear-leveling.

Many applications that rely on the regular block interface and random writes cannot run atop ZNS devices without modification. Existing solutions to allow compatibility with the ZNS interface include host-side FTLs or file system support; F2FS [18] and btrfs [32] currently support zoned devices. However, end-to-end integration into applications is shown to provide the best performance [6].

2.2 RAID

RAID refers to a family of redundant data distribution techniques that enhance performance and reliability by leveraging multiple independent physical storage devices [28]. Modern RAID is often implemented as a logical volume that can be treated as a normal block device (e.g., `mdraid` [4]), or is sometimes integrated into the filesystem [31]. RAIZN implements distributed parity, akin to RAID-5, to tolerate failures in an array of ZNS devices. Several aspects of RAID are incompatible with ZNS devices, which we elaborate on in Section 3, including the simple stripe to device address translation.

`mdraid` supports write journaling to close the RAID-5 write hole and provide atomicity and durability for writes [8]–the write journal is fully compatible with RAIZN, and as such we consider it orthogonal to our contribution. As we discuss in Section 5.1, RAIZN must implement atomicity of coupled data and parity updates to conform to the ZNS specification, a consequence of which is the elimination of the write hole problem. As we describe in Section 5.2, RAIZN provides atomicity for writes within a single stripe, but using a dedicated journal volume with RAIZN would protect from torn writes spanning multiple stripes.

3 CHALLENGES OF ZONED DEVICE ARRAYS

In this section, we describe how the zoned interface is problematic for a conventional RAID-like setup. RAID organizes data into stripes consisting of stripe units and parity distributed across all devices,

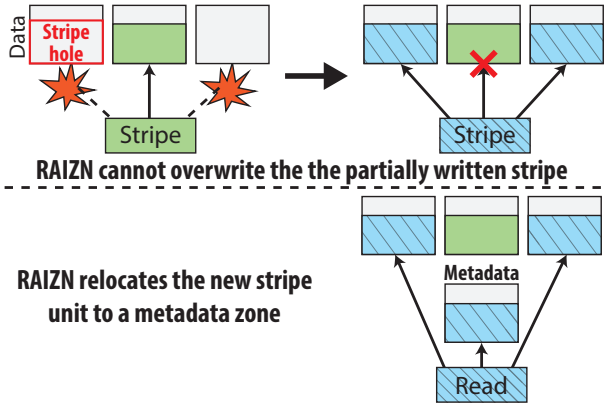

Figure 1: Only a subset of the stripe units are persisted before power is lost, resulting in a hole in the logical address space. The next stripe cannot be written at the correct address due to the persisted stripe unit.

with stripe units mapped arithmetically to specific addresses on particular devices based on array initialization parameters.

Metadata management. Unlike conventional RAID where essential metadata consists of a superblock that is only written once, RAIZN requires additional metadata to handle cases where the lack of overwrite semantics necessitates indirection or logging, such as partial stripe writes or partial zone resets. Metadata cannot be overwritten and must be log structured when stored in a ZNS device, adding further complexity and necessitating garbage collection.

Parity updates. Writes that are not aligned to a stripe boundary, especially those that are smaller than a stripe, reduce performance in conventional RAID due to parity updates [16], but with ZNS devices this becomes a correctness issue. In ZNS devices, LBAs that are written cannot be changed until the entire zone is reset, meaning that it is impossible to update parity after a non-aligned write. However, partially calculated parity must be written before notifying the host of IO completion, otherwise data loss can occur.

Stripe write atomicity. Figure 1 illustrates a pessimistic scenario when working with ZNS devices. Specifically, a subset of stripe units in a stripe are persisted before power loss, but this subset is insufficient to recover the stripe after reboot. Conventional RAID allows the user to overwrite this stripe without issue, but ZNS stripe units that have been persisted cannot be overwritten without resetting the entire zone. As a result, the traditional arithmetic mapping of RAID addresses to device addresses cannot support ZNS, and an additional layer of indirection is required. This layer of indirection poses a challenge in designing RAIZN to handle such edge cases without harming performance.

The problem extends to torn writes on a single physical device, where only part of a stripe unit is written before power is lost. Torn writes can be handled similar to partial stripe writes but many devices, including those in our evaluation, support atomic writes, so torn writes are handled by the device when writes are smaller than the device-defined atomic granularity and alignment.

Zone reset atomicity. It is necessary to take care when resetting a RAIZN zone, as it spans multiple physical zones. Such a request would be translated into a reset for all physical zones involved, but those operations are not atomic, meaning the system could lose power after resetting only a subset of the zones. In most cases, it is possible to detect that this scenario occurred by examining the write pointer of each physical zone, detailed in Section 5.2.

Write persistence. Forced unit access (FUA [3]) writes bypass write caches and instruct the device to write the data immediately, before the write is acknowledged to the application. FUA writes are used in cases where data persistence is important (e.g., database journaling), and pose a challenge as simply propagating the FUA flag to the IO sent to the underlying device is insufficient due to the stripe write atomicity problem described above. If a FUA write at a given LBA is persisted, but the data at previous LBAs in the same (logical) zone are not persisted, sections of the logical address space of a zone are rendered unreadable, violating the ZNS specification. As such, FUA writes must be handled with extra care to ensure the data can be read after power or device failure. We describe how RAIZN handles flushed writes in Section 5.3.

4 ARCHITECTURE OF RAIZN

RAIZN appears as a single, virtual, host managed zoned device; that it is a device array is transparent to applications, and any ZNS-compatible application performing IO through the kernel block layer can run, unmodified, on a RAIZN volume. We leverage the device mapper (`dm`) framework to set up the logical block device and handle routing of logical requests to the RAIZN driver.

A core design question for RAIZN is address space management: how the physical addresses on the physical devices are organized into logical block addresses that are exposed to the host. We define two address spaces, the *logical* address space corresponding to the RAIZN logical device, and a *physical* address space for each of the underlying devices. We refer to the block addresses in the logical device's address space as logical block addresses (LBAs) and block addresses on the physical device as physical block addresses (PBAs). Host applications (e.g., filesystem) submit IOs to RAIZN, which translates the requested LBA into a set of PBAs before submission to the physical devices.

This section describes the basic details of how RAIZN organizes data and metadata, and provides an overview of the types of metadata used in RAIZN. In Section 5 we go into detail about how different metadata are used, persisted, and kept crash consistent.

4.1 Data Placement and Processing

RAIZN uses a data placement scheme similar to that of conventional RAID-5, but extended to support ZNS-specific edge cases. Each LBA is statically mapped to a particular device and PBA using a simple arithmetic translation, and user data is organized into stripes, which are divided into stripe units (referred to as "chunks" in `mdraid`), parity coded, then distributed across the devices in the array. For example, data written to LBA `0x00` is striped and each stripe unit is written to PBA `0x00` on the corresponding physical device. This allows RAIZN to translate reads without additional memory lookups, minimizing impact on IO latency.

Each logical IO request received by RAIZN is processed by computing parity, caching partial stripe data, and dividing the data into smaller IOs to be submitted to the physical devices. We term

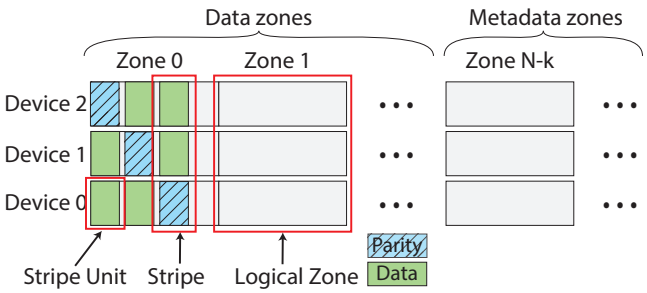

Figure 2: RAIZN data layout for three devices. Data is striped into two data stripe units with one parity stripe unit; the device holding the parity is rotated every stripe. Logical zones consist of one physical zone per array device.

these additional generated physical IOs *sub-IOs*, which include data, parity, and metadata.

Figure 2 illustrates how the address space of the physical devices is organized in RAIZN. Physical zones are grouped into *data* and *metadata* zones; the number of metadata zones is configurable, with a minimum of 3 per device (Section 4.3). Data zones are organized into *logical zones*, each of which corresponds to one physical zone per device. RAIZN presents each logical zone to the host application as a sequential-write only zone. For example, if RAIZN is configured to have D data stripe units and P parity stripe units per stripe (for a total of $D + P$ devices), each logical zone appears to the user as a single ZNS zone with the same capacity as D physical zones. For simplicity, we map physical zone N of each device into logical zone N, and assume all devices have the same zone size and capacity.

This simple LBA to PBA mapping in RAIZN allows logical zones to behave similarly to physical ZNS zones; if multiple objects with similar lifespan are written to the same logical zone in RAIZN, no additional internal garbage collection or indirection is occurs on those objects, and data is laid out on the physical media in a manner similar to if it were written directly to a physical ZNS device.

4.2 Fault Tolerance

Fault tolerance in RAIZN operates with minor deviations from the behavior of conventional RAID. Degraded reads and writes are handled in the same way as conventional RAID; missing stripe units are reconstructed from parity for reads, and omitted on writes.

However, RAIZN rebuilds devices differently from conventional RAID; when a failed device is replaced, RAIZN rebuilds the new device zone by zone. RAIZN prioritizes rebuilding active (open or closed) zones first, before continuing to rebuild finished zones. During rebuild, writes to non-rebuilt open zones are served in degraded mode. Prioritizing active zones minimizes the read overhead of rebuilding, minimizing the delay before all further writes can be served in a non-degraded manner.

One advantage the ZNS interface confers to RAIZN is the ability to easily determine which block addresses contain valid data. RAIZN rebuilds only the subset of LBA ranges that contain user-written data–this results in performance advantages described in Section 6.2. While the non-ZNS NVMe devices are technically capable of determining which LBA ranges are unwritten or deallocated, gathering this information is impractical as it would require checking for errors on every block address on the device.

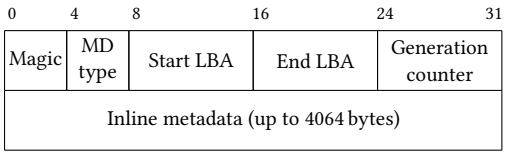

Figure 3: RAIZN metadata header layout when using 4 KiB sectors. Offsets are shown in bytes.

4.3 Metadata Management

In this subsection, we detail how metadata is stored in memory and on-disk in RAIZN, along with an overview of all types of metadata in RAIZN. Later sections will elaborate on each of these metadata.

Unlike RAID, RAIZN cannot store and update metadata in a fixed location, because ZNS disallows overwrites. Furthermore, reserving an entire zone, which for our SSDs is 1077 MiB, for a 1 MiB metadata structure is wasteful. RAIZN has several types of metadata, a subset of which are persisted as log-structured updates; persisting metadata updates in log format allows RAIZN to conform with the sequential write constraint, and by storing multiple types of metadata updates within a single metadata zone RAIZN minimizes the number of reserved metadata zones. RAIZN reserves one zone for *partial parity* (Section 5.1), one zone for all other metadata, and at least one zone (termed the *swap zone*) to facilitate garbage collection of metadata zones. This subsection describes, how metadata is used, organized, persisted, and kept consistent after power loss.

The total size of metadata in RAIZN is relatively small (<100 MiB), allowing RAIZN to cache it in memory. A subset of metadata must be persisted on all devices in the array, including RAID parameters, device ID assignments, generation hashes, and zone reset write-ahead logs. The remaining metadata, including parity log entries, remapped stripe units, and relocation map entries, is written only to its corresponding device. If a device fails, non-replicated metadata on that device is no longer of any use and its loss is inconsequential. The persistent location, storage overhead, and memory footprint of each type of metadata are described in Table 1.

All metadata in RAIZN is cached in memory, and the persistent copy is read when the volume is remounted. In our experiments, valid persistent metadata is typically 192 KiB–4096 KiB, primarily consisting of cached partial parity. Metadata is written using zone appends, ensuring high throughput even in the presence of many concurrent metadata log writes.

A single metadata zone could hold log structured updates for every type of metadata, but most metadata in RAIZN is updated infrequently; the exception is parity logs (Section 5.1), which are generated on every non stripe-aligned write and invalidated when the full stripe is written. This can be quite frequent for many workloads, so RAIZN writes parity logs to a separate metadata zone, isolating the rest of the system from the effects of parity logging.

Metadata headers. Every persisted metadata log in RAIZN contains a metadata header consisting of (1) the metadata type, (2) the LBA range described by this metadata, and (3) the *generation counter* of the logical zone containing the aforementioned LBA.

Table 1: Location and size of RAIZN metadata for a 5-device array with 64 KiB stripe units and 1077 MiB physical zone capacity

Metadata type	Persistent location	Storage per update	Memory footprint
Remapped stripe unit	Affected device only	4 KiB (header) + 64 KiB (stripe unit)	4 KiB + 64 KiB (stripe unit)
Zone reset log	All devices	4 KiB	-
Generation counters	All devices	4 KiB	8.05 bytes per logical zone
Partial parity	device with parity	4 KiB (header) + ≤64 KiB (stripe unit)	-
Superblock	All devices	4 KiB	4 KiB
Stripe buffers	-	-	320 KiB (5 stripe units) × 8 per open logical zone
Persistence bitmaps	-	-	2 KiB per logical zone
Physical zone descriptors	-	-	64 bytes per zone per device
Logical zone descriptors	-	-	64 bytes per logical zone

The precise layout is shown in Figure 3: the first 4 bytes hold a fixed "magic" value to identify the beginning of a metadata entry, followed by 4 bytes describing the type of metadata associated with this header (Table 1). This is followed by 16 bytes to store the start and end LBA, and finally 8 bytes containing the generation count of the logical zone containing the LBA. The remaining 4064 bytes of the metadata header is used for inline metadata to minimize storage overhead. The superblock, zone reset logs, and generation counters are persisted in the inline metadata section of the metadata header.

Generation counters. Generation counters are RAIZN's way of uniquely identifying the contents of a given LBA over the lifetime of the volume–every time a (logical) zone is reset, its generation counter is incremented by one, and every time the RAIZN volume is mounted, the generation counter for every empty zone is incremented. This monotonically increasing counter, when paired with an LBA, is used to track validity of metadata logs. This property is key to RAIZN's metadata management: If a metadata header includes an outdated generation counter, the metadata associated with that header is invalid due to the logical zone being reset.

We implement generation counters as one 64-bit counter per logical zone, essentially eliminating overflows. In the event any counter does reach its maximum value, the RAIZN volume goes into read-only mode and requires the user to run maintenance on the volume. During maintenance, RAIZN garbage collects and resets all of the metadata zones, then resets all generation counters to zero. To ensure the atomicity of maintenance operations, RAIZN uses write-ahead logging for each operation and resumes any interrupted operations after reboot. The atomicity of this maintenance operation, coupled with the guarantee that all stale metadata entries will be deleted before the system completes maintenance, allows generation counters to be reset without impacting data consistency.

Generation counters are laid out in memory in the same format in which they are persisted–32 bytes of metadata header followed by 508 8-byte generation counters. When a generation counter is updated, the entire 4 KiB is persisted.

If a logical zone is reset, but the system loses power before the generation counters can be persisted, RAIZN maintains consistency by handling the following two cases. If only a subset of the physical zones have been reset, this is handled as a partial zone reset as described in Section 5.2. If all physical zones were reset, the logical zone would be detected as being empty and as a result the generation counter would be incremented on initialization, invalidating any existing metadata entries for the logical zone.

Zone descriptors. RAIZN stores the write pointer and zone status for each physical and logical zone in memory. Before determining logical zone status, RAIZN computes the write pointer for each logical zone, by extension detecting any consistency errors from power loss. On mount, RAIZN checks the write pointer for each physical zone in the array. The highest physical zone write pointer per logical zone determines the logical zone write pointer, and RAIZN checks whether the stripe ending at the logical zone write pointer is readable. If any physical zones are missing stripe units in this stripe, a "stripe hole" is detected (illustrated in Figure 1). If a zone reset was logged, RAIZN resets all of the physical zones in the logical zone before resetting the logical zone write pointer. In all other cases, the stripe hole indicates a partial stripe write, which RAIZN first attempts to correct by rebuilding the missing stripe units using parity. If this is impossible, the zone is marked as "remapped", the logical zone write pointer is changed to hide the corrupted stripe unit(s) from the user, and future conflicting stripe unit writes are redirected to the metadata zone.

Note that FUA or flushed writes (Section 5.3) cannot run into this scenario where data is rendered unreadable, as the user is not notified of IO completion until all LBAs in the logical zone, up to and including the data associated with the FUA or flushed write, is persisted, precluding any possibility of a stripe hole.

Metadata garbage collection. RAIZN must periodically garbage collect metadata to free up space in the metadata zones. The RAIZN garbage collector uses swap zones to facilitate garbage collection without interrupting operation, as illustrated in Figure 4. RAIZN first designates a swap zone to replace the full metadata zone, immediately writing any new log entries there. The garbage collector checkpoints any valid in-memory metadata to the swap zone, and does not read any logs from SSD. The metadata type in the metadata header for each of these checkpointed entries is flagged to distinguish them from normal metadata updates. Once the checkpoint is complete, the old metadata zone is reset to serve as a swap zone. Generation counters, the superblock, and relocated stripe units are serialized as-is from memory to stable storage. Zone reset logs and partial parity are calculated and written, the latter of which

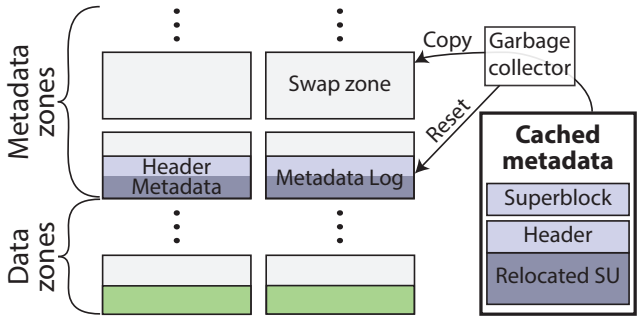

Figure 4: The garbage collector checkpoints metadata before resetting the old metadata zone. Each metadata entry has a header that includes (1) the type, (2) the applicable LBA range, and (3) the generation count of the logical zone.

is calculated by XOR'ing the contents of the stripe buffer of each open logical zone–this is elaborated on in Section 5.1.

If garbage collection is interrupted by power loss, logs from both the old metadata zone and swap zone are ingested and processed–logs may be duplicated but it is impossible for conflicting log entries to exist due to the lack of stripe update semantics and the generation count stored in the metadata headers. Each combination of LBA and generation count is unique over the lifetime of the array, and logs other than the one with the highest generation count are discarded, eliminating ambiguity. The exception to this is partial parity, so for simplicity if there is a checkpointed partial parity that overlaps with a normal partial parity, we discard the checkpointed entry.

Multithreaded write processing. RAIZN uses multiple worker threads to achieve high throughput when serving small writes, with the primary challenge being that zone writes must be submitted sequentially to the kernel block layer. To achieve this, RAIZN tracks the write pointer for each zone on each device in the corresponding physical zone descriptor, waiting to submit sub-IOs until the write pointer matches the PBA of the sub-IO.

RAIZN updates the logical zone write pointer based on the logical IO address, pushes each logical IO onto a single shared queue, then schedules a worker thread to serve IO from the head of the queue. Read and metadata sub-IOs are submitted immediately, but writes and flushes must be submitted in the correct order. For writes, the worker thread waits until the physical zone write pointer matches the sub-IO address, locks the physical zone descriptor, submits the sub-IO, then updates the physical zone descriptor before unlocking it. Flush sub-IOs are submitted last.

Zone resets are ordered with respect to writes by tracking the last written LBA at the time the reset request is received – this LBA is termed the *reset pointer*. Zone reset sub-IOs are submitted to each array device when the corresponding physical zone write pointer matches the reset pointer – this is immediate in practice, as typical workloads do not reset a zone before all in-flight writes complete.

5 SOLVING ZNS CHALLENGES

In this section, we describe the solutions to the problems presented in Section 3, and present a brief explanation of how each of our solutions achieves crash consistency and fault tolerance.

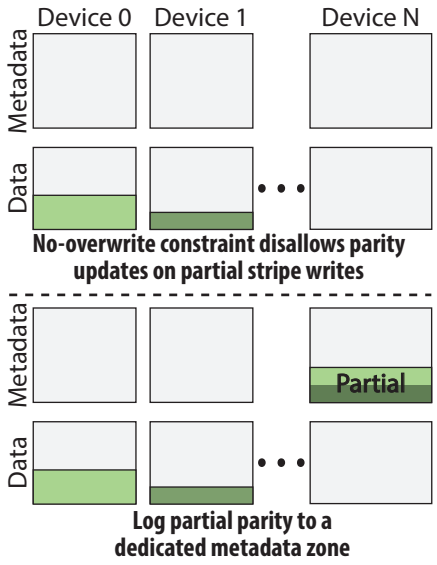

Figure 5: RAIZN writes parity for partially filled stripes to a dedicated metadata zone.

5.1 Parity Updates

Non stripe-aligned writes pose a problem when calculating and writing parity, due to zones being append-only. RAIZN cannot inform the user of write completion until enough data and parity is written to recover following the failure of a device. All data is immediately written to the corresponding device regardless of stripe alignment, but the corresponding parity is unknown until the entire stripe is written. One solution to this problem is to use a separate randomly-writable storage device (e.g., persistent memory, conventional namespace on a ZNS SSD) to buffer parity updates or stripe writes (e.g., the `mdraid` journal). However, our design prioritizes wide compatibility, avoiding the requirement for any additional hardware or non-universal ZNS features.

RAIZN handles partially-written stripes by first caching the written data in a *stripe buffer*, then persisting the partial parity to the partial parity metadata zone. Figure 5 illustrates this mechanism.

The immutability of data in ZNS allows RAIZN to log only the partial parity, rather than both the data and parity. RAIZN only logs the subset of parity that is affected by the write in question, minimizing write amplification; the only additional write amplification (compared to conventional RAID) caused by partial parity logging is the metadata header.

The stripe buffer in RAIZN is similar in function to the stripe cache in `mdraid`, as it enables parity recalculation without incurring disk reads. The key difference is that the ZNS interface prevents written stripes from being updated, and the open zone limit sets an upper bound on the number of "incomplete" stripes, and by extension the number of stripe buffers. Once a stripe buffer is filled, the full stripe parity is calculated, then the stripe buffer is reused for the next partial stripe. A logical zone often has multiple active stripe buffers, for example if a new stripe write is processed before the previous stripe write is persisted to the physical devices. To provide predictable memory use without sacrificing performance,

RAIZN pre-allocates a fixed number of stripe buffers per open zone (8 in our experiments), and blocks write processing if all stripe buffers are occupied–this does not occur in our experiments.

During initialization, if a device is missing, up to one stripe buffer is reconstructed per open logical zone by combining all logged partial parity. The data from the missing device can be reconstructed by taking all of the partial parity for the stripe in order (according to the LBA ranges included in the header), then XOR'ing it with the data from the non-failed devices. When performing this XOR, data from the non-failed devices up to the end LBA in the metadata header is included, and data after this address is treated as zeroes (recreating the conditions under which the partial parity was originally calculated). Once the full LBA range from the beginning of the stripe to the last written LBA is XOR'ed in this manner, the missing data is fully reconstructed. If some portion of the data is missing due to non-persisted partial parity, data at any LBAs at or higher than this missing data is discarded.

5.2 Write and Reset Atomicity

Atomicity for stripe writes. The independence of the devices in both RAID and RAIZN presents a challenge in atomically writing LBA ranges that span multiple physical devices. Two specific problems arise due to the lack of atomicity when writing multiple physical devices: first, the write hole problem[35], where the parity and data can become de-synchronized, and second, torn writes. These problems are optionally solved in `mdraid` if a journal volume is used; Due to the requirements of the ZNS interface, RAIZN is required to close the write hole, but does not suffer to the same degree as `mdraid` with regards to torn writes.

RAIZN must solve the write hole problem due to a ZNS-specific edge case with regards to write atomicity, partial stripe writes (Section 3). Figure 1 illustrates how partial stripe writes break RAIZN's simple data placement scheme. A stripe is written to the RAID device, but the system loses power before all of the stripe units are persisted. There is insufficient data to repair the missing stripe unit, so in this example RAIZN must behave as if the entire stripe was not written. In a normal RAID, the presence of the single persisted stripe unit is not a problem, as an additional write at the same LBA can simply overwrite the corresponding PBAs; ZNS disallows overwrites, so RAIZN must place the new data elsewhere.

RAIZN *relocates* the new stripe unit to a metadata zone on the affected device (bottom of Figure 1), generating a *relocated stripe unit* metadata entry. The modified LBA to PBA mapping for this stripe unit is stored in a hashmap, which is checked on reads if the logical zone being read is flagged as containing a relocated stripe unit. Relocations are uncommon, so RAIZN caches relocated stripe units in memory in addition to persisting them in a metadata zone.

It is possible for the metadata zone to run out of space due to too many remapped stripe units, so if the number of remappings passes a user-modifiable threshold, RAIZN rebuilds the affected physical zones during initialization. All data is copied from the affected physical zone into a swap zone, the zone is reset, and then the data is copied back with the remapped stripe unit written to the correct address. All operations are logged to ensure they can be resumed in case of power loss.

A side effect of this stripe unit relocation mechanism and the ZNS interface is the reduction of severity of torn writes. `mdraid` suffers from a potential problem where experiencing failure while overwriting an LBA range can result in part of the data corresponding to the overwrite with the remainder unmodified–this is a torn write, and is solved by the `mdraid` write journal. While this is still a problem in RAIZN, it is less severe for the following reasons: first, the immutability of written data in ZNS ensures any data returned will represent the contents of a single write request, and second, torn writes will always result in the lower order LBAs being readable with the higher order LBAs returning IO errors. While RAIZN is not able to provide true torn write protection without using a journal volume, this behavior makes it less likely for insidious data corruption to affect applications running on top of RAIZN.

Solving partial zone resets with zone reset logs. Partial zone resets occur when a subset of the physical zones in a logical zone are not reset before power is lost. In many cases, this can be detected and handled during initialization by detecting holes in the logical address space. However, it is impossible to distinguish between a partial stripe write and a partial zone reset if the first stripe unit in a zone is present. To solve this ambiguity we use write-ahead logging for zone resets, logging the intent to reset a zone to the physical device holding the first stripe unit in the logical zone and the physical device holding the parity for the first stripe in the zone. This introduces additional latency to zone resets, but we do not expect this to be a problem because typical workloads do not immediately write to zones after resetting them.

In practice, zone resets are blocking operations, so IOs will typically not be submitted to a zone before the completion of the reset command, but because it is technically possible to issue an asynchronous zone reset through the kernel block layer, as a precaution we block all IOs to a logical zone after receipt of a reset request and unblock it after all of the physical zones are reset.

Zone reset logs are persisted to the general metadata zone on two physical devices: the device holding the first stripe unit in the zone, and the device holding the associated parity. To avoid non-uniform write amplification, the device order is rotated for each zone, so that the physical device holding the first stripe unit is different for successive logical zones. During system initialization, if a logical zone is not empty despite a valid zone reset log indicating that it should be, it is reset. By persisting zone reset logs on two physical devices, RAIZN is tolerant against a single device failure. RAIZN does not reset zones until the zone reset logs are *persisted*, so if zone reset logs fail to persist before power loss the zone is not reset and contains all the original data.

5.3 Write Persistence

Persisting data written to RAIZN can be done in three different ways: a flush, an IO with the preflush flag set, or a forced unit access (FUA [3]) write. Flush requests are duplicated submitted to each of the array devices (`REQ_OP_FLUSH`).

In contrast, FUA writes and preflushed IOs require special handling; if the host is notified of FUA or preflushed IO completion, the data persisted by that IO must be readable after power loss. In conventional RAID, the RAID-5 write hole is the primary challenge

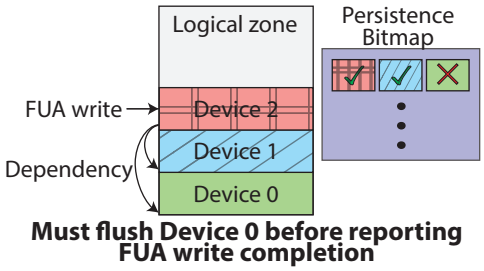

Figure 6: A FUA write in RAIZN must check that all LBAs preceding itself in the same logical zone are persisted before reporting completion. In this example, the FUA write must explicitly flush device 0 before reporting IO completion, because the persistence bitmap shows the stripe unit on physical Device 0 is not persisted.

in providing this guarantee, for example if the data is updated but the corresponding parity update is not persisted before power loss. In RAIZN, there is an additional constraint introduced by the ZNS interface: data at a given LBA should not be readable unless all preceding LBAs in the same zone are also readable.

RAIZN solves this additional constraint by notifying the host of FUA write completion only after all previous write requests within the same logical zone have been persisted; note that in ZNS, writes cannot span multiple zones. This dependency is illustrated in Figure 6. This requires setting the `REQ_PREFLUSH` flag on the FUA write, and submitting a flush sub-IO to each device that contains a non-persisted stripe unit in the same logical zone. To track the persistence of data in a logical zone, RAIZN maintains an in-memory bitmap, called the *persistence bitmap*, to track which LBAs have been persisted. The persistence bitmap has one bit per stripe unit, and every time a flush, preflush, or FUA write completes, the persistence bitmap is updated for each active logical zone, setting the corresponding bits in the persistence bitmap up to the write pointer. If a write starting in the middle of a stripe unit is persisted, it is implied that the beginning of the stripe unit was persisted, as all sectors in a stripe unit are written to the same physical device. This allows RAIZN to track the persistence of writes smaller than a stripe unit while only using 1 bit per stripe unit. RAIZN leverages the ZNS sequential write constraint by only checking the persistence bitmap from the beginning of the stripe immediately preceding the current write. If all the stripe units in the previous stripe are persisted, it implies that all PBAs on all array devices up to and including the stripe units in the previous stripe are also persisted.

RAIZN submits a flush sub-IO to every device containing non-persisted stripe units within the logical zone, and after the completion of all flushes RAIZN notifies the host of write completion.

On remount, the persistence bitmap is reconstructed from the write pointers of the physical zones in the array devices. A FUA write that was reported as complete cannot be "lost" after power failure; the data itself is persisted on the array, and all LBAs from the beginning of the logical zone until the contents of the FUA write must also have been persisted on the array before completion is reported–this precludes any possibility of a "stripe hole" from the beginning of the zone until the end of the FUA-written data.

5.4 Discussion

In this section, we briefly discuss aspects of the ZNS interface, NVMe, and RAID that are not addressed in RAIZN's design.

Zone appends. The intended approach to achieving high throughput on small writes is to use zone appends, as strict sequential submission is not required for zone appends like it is for writes–however, supporting concurrent zone appends to a single logical zone in RAIZN is difficult due to the partial stripe write problem.

As of the time of writing, the precise characteristics of zone append-based workloads is still unclear, as the command only recently became available as a storage construct in the ZNS standard. We look forward to seeing how future append-based workloads behave and how RAIZN can be designed to serve those workloads. Notably, btrfs [32] uses zone appends for the write path, so future work will involve profiling different applications running on btrfs.

Logical block metadata. The NVMe standard supports attaching metadata descriptors to each logical block; these descriptors are typically used for protection information (PI [34]), but can also store user-defined data. There are opportunities for optimizations in RAIZN if the underlying SSDs are formatted with logical block metadata enabled and made available to RAIZN.

First, metadata logs would shrink–RAIZN attaches a 4 KiB header to each metadata log, but most of the 4 KiB is used for inline metadata. The actual header information could be written into the metadata descriptor instead, reducing write amplification and increasing the performance of small writes.

Second, zone appends could be supported without requiring serialized completion. The problem with zone appends in RAIZN is that on-device reordering of appends can result in stripe units being written in an arbitrary order. In the event of power loss before the zone append completion is reported to RAIZN, it is impossible to determine which stripe unit corresponds to which stripe. However, if each append is tagged with a stripe ID in the metadata descriptor, RAIZN could locate stripe units after unexpected shutdown.

Zone Random Write Area (ZRWA). The ZNS standard defines ZRWA, which is a window of randomly-writable blocks that moves together with the write pointer of a zone. ZRWA allows blocks to be overwritten within a sequential write required zone, albeit only within that specific window. Depending on the availability and performance characteristics of ZRWA, it could potentially be used to allow some parity updates to take place in-place and avoid the overhead of the parity logs.

Consequences of large zones. Currently, the capacity of each logical zone in RAIZN scales with the width of the array, resulting in large zones. Larger zones result in coarser erase granularity, which in turn makes it more difficult for host applications to efficiently garbage collect objects, potentially increasing write amplification. Recent work in SmartFTL [13] has shown that for certain workloads, such as cluster filesystems, it is advantageous to sacrifice per-object IO throughput in exchange for finer granularity media-level control over data placement. While the problem of large zones exists for all zoned devices, it is exacerbated in RAIZN, and part of our ongoing work is to solve this problem. One potential solution to co-locate

multiple logical zones in the same set of physical zones, with a table to map logical zones to physical zones.

Atomicity/Durability mechanisms. `mdraid` allows users to optionally use a dedicated journal volume to close the RAID-5 write hole [8] and provide atomicity for writes spanning multiple `mdraid` chunks. The journal achieves this by first buffering stripes on a dedicated journal device before flushing them to the array. As we describe in Section 5.2, RAIZN closes the write hole and the ZNS interface reduces the benefit of atomic writes across multiple chunks.

RAIZN closes the write hole through a combination of the parity log, immutability of data, and immutability of *final* parity. The parity log prevents the case where the parity (or partial parity) of a stripe does not match its data–if a stripe is appended to, then a new parity log is generated and written to describe the latest contents of the stripe, and if a device containing a data stripe unit fails, the parity log is used to reconstruct the contents of the stripe. If the parity log is also lost (i.e., if a power failure and device failure simultaneously occur before the parity log is persisted), RAIZN behaves as if the data in that failed stripe unit never existed and relocates the replacement stripe unit as described in Section 5.2. As a result, writes within a single stripe are atomic, but cross-stripe atomic writes require a journal or similar mechanism.

6 EVALUATION

We evaluate the performance of RAIZN using microbenchmarks and application benchmarks, demonstrating that RAIZN is able to achieve comparable performance to `mdraid` level 5.

All experiments were run on a Dell R7515 server with a 16-core AMD EPYC 7131P CPU, 128 GiB of DRAM, and Ubuntu 20.04 running on a 512 GB conventional SSD. We modified the Linux kernel 5.15 and mkfs.f2fs to remove hard-coded constraints that prevent the creation of (logical) devices with zone sizes larger than 2 GB, but these changes do not affect performance.

For RAIZN experiments, we use 5 Western Digital Ultrastar DC ZN540 2 TB ZNS SSDs, and for the experiments on `mdraid` we use 5 conventional SSDs with the same capacity and hardware platform. Each ZNS SSD zone has a capacity of 1077 MiB. Both RAIZN and `mdraid` are configured to run with 8 worker threads, the latter configured with the maximum possible stripe cache size of 128 MiB. In all experiments, `mdraid` was configured to run without a journal volume, ensuring maximum performance.

6.1 Raw Device Microbenchmarks

We begin by evaluating the basic read and write performance of the ZNS and conventional SSDs using fio 3.28 [5], with libaio [26].

First, we write 1 TiB to the devices, followed by a sequential read of the written data; this was repeated over a variety of block sizes and queue depths to measure the maximum throughput of the ZNS and conventional SSDs. We omit a detailed performance analysis of the devices and point the reader to prior work for more information [6]. The ZNS SSD's throughput is 1052 MiB/s for writes and 3265 MiB/s for reads, 2% and 4% lower respectively than the conventional SSD. We attribute this gap to a difference in firmware maturity between the two devices, and expect it to close over time.

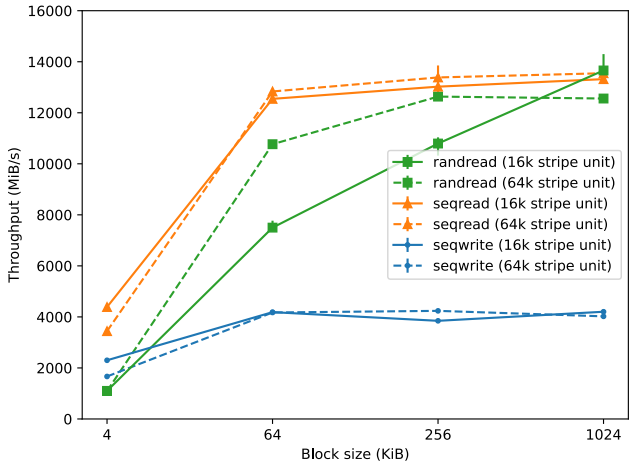

Figure 7: `mdraid` random read throughput is higher with 64 KiB stripe units, but sequential read and write throughput is lower compared to using 16 KiB stripe units.

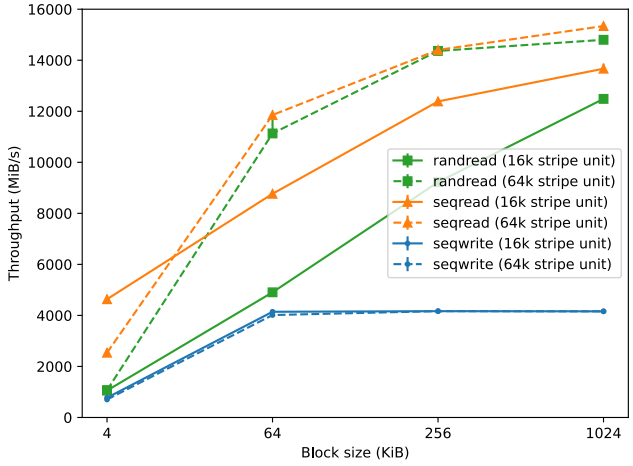

Figure 8: Other than 4 KiB sequential reads, RAIZN performs better with 64 KiB stripe units than 16 KiB stripe units. 4 KiB sequential read performance is less important than larger granularity sequential read performance.

We run several experiments to measure the performance of conventional `mdraid` and RAIZN. We start by running write, sequential read, and random read benchmarks, varying the stripe unit size from 8 KiB to 128 KiB. The optimal stripe size is dependent on workload, so we present throughput graphs (Figures 7 and 8), selecting a stripe size that performs reasonably well across benchmarks. For brevity, we only include graphs of a representative subset of the workload configurations we ran.

Observation 1: 64 KiB *stripe units perform optimally for RAIZN, and maximize random read performance in* `mdraid` *without significantly hurting sequential read or write throughput.*

The points in Figures 7 and 8 are the median of three trials; error bars denote the minimum and maximum, and each series represents `mdraid`/RAIZN configured with a different stripe unit size.

The sequential read workload is driven by 8 jobs, each with a queue depth of 64. Each job performs direct IO reads on the `mdraid`/RAIZN volume starting at different offsets. The volume is primed by writing 1024 GiB of data sequentially from the beginning of the address space, after which the read and random read benchmarks are run in succession.

The sequential write workload is also driven by 8 jobs each with a queue depth of 64, with fio directly writing to the `mdraid`/RAIZN volume starting at different offsets. The volume is unmounted, all devices formatted, and the `mdraid`/RAIZN volume is re-initialized before each trial of the write workload to avoid on-device garbage collection affecting the results.

The random read workload is driven by 1 job with a queue depth of 256, randomly reading addresses within the 1024 GiB of data that was written to the volume during the priming phase. Random reads perform best when the block size is smaller than the stripe unit size, as the number of sub-IOs increases dramatically if many stripe units are involved in the logical IO.

Figure 7 shows that while using 16 KiB stripe units results in higher throughput for large block size reads, this is offset by the large reduction in random read throughput compared to 64 KiB stripe units. In Figure 8, RAIZN performs better with 64 KiB stripe units than 16 KiB stripe units on all workloads other than 4 KiB sequential reads; however, it is unlikely that real applications will perform such small block size sequential reads, so we focused on the performance of the other workloads when determining the optimal stripe unit size for RAIZN. Based on the results of these three workload benchmarks, we determine that a stripe unit size of 64 KiB provides reasonable performance for both systems, and use this stripe unit size for the remainder of the experiments.

Observation 2: *RAIZN achieves similar throughput and latency to* `mdraid`*, but is outperformed on small (4–64* KiB*) reads and writes.*

Figure 9 shows the difference in performance between `mdraid` and RAIZN for the three workloads described above, with the top subgraph showing throughput followed by the median latency and 99.9th-percentile tail latency. The performance of RAIZN on small (4 KiB–64 KiB)) sequential reads and writes lags behind that of `mdraid`; the low 4 KiB sequential write performance is due to the relative overhead of the parity log header, which results in a proportionally large overhead when writing smaller blocks. The difference in sequential read performance at low block sizes is likely caused by two factors: first, the ZNS SSDs inherently have 4% lower read performance than the FTL SSDs. Second, RAIZN's read path prioritizes low latency at the expense of lower IOPS for small block size reads. However, small sequential reads is likely an impractical workload, as it provides little benefit over large sequential reads for most applications. Instead, we draw focus to the strong performance of RAIZN when serving large (256 KiB–1 MiB) sequential reads.

For both median latency and 99.9th-percentile tail latency, RAIZN achieves similar results to `mdraid`, with `mdraid` typically performing slightly better at smaller block sizes and RAIZN performing slightly better at larger block sizes. The reasons for this performance difference are the same as with throughput.

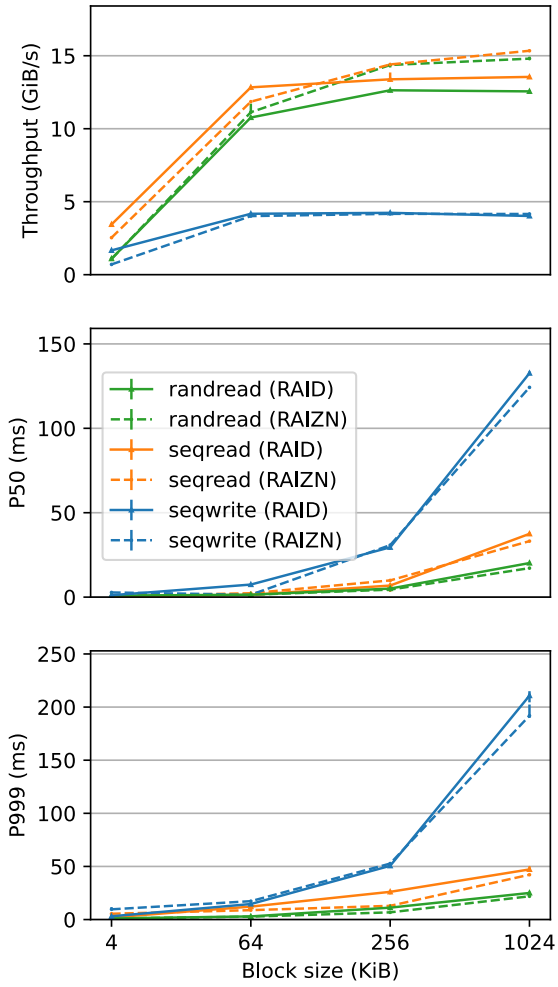

Figure 9: RAIZN achieves comparable throughput and tail latency to `mdraid`

Observation 3: `mdraid` *can experience unpredictable and severe performance degradation from on-device garbage collection. RAIZN is unaffected as ZNS SSDs do not perform on-device garbage collection.*

To illustrate the effects of garbage collection on conventional SSDs, we run a full device overwrite benchmark on `mdraid` and RAIZN. This benchmark is composed of two workloads–first, 5 threads concurrently write the entire capacity of the array, with each thread writing 20% of the address space (the first thread sequentially writes from 0% → 20%, the second thread writes 20% → 40% etc.). After the first workload completes, one thread sequentially overwrites the entire address space of the array. During both workloads, we sample throughput and latency every second, and graph a timeseries of the results in Figure 10. The red vertical dashed line represents the point in time where the second workload starts, and is marked by a slight reduction in throughput and a dramatic reduction in latency–this change is due to the reduction from five threads to one thread between workloads 1 and 2. `mdraid` experiences a sharp drop in throughput after the conventional SSDs exhaust their overprovisioned blocks and begin performing garbage

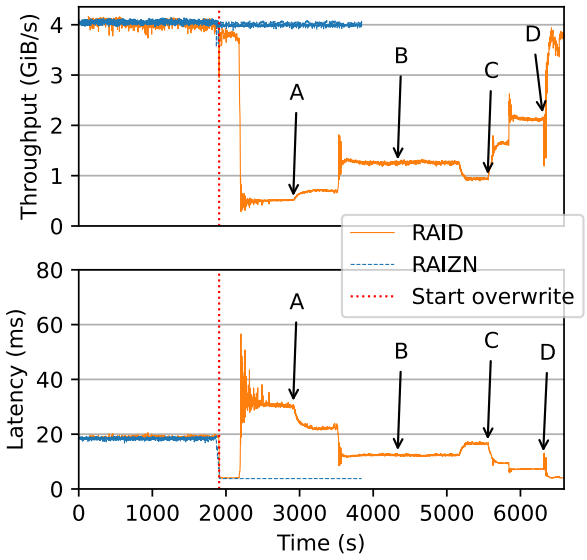

Figure 10: `mdraid` suffers when the SSDs run out of spare blocks and start performing garbage collection. RAIZN is able to maintain high throughput and low latency because ZNS SSDs do not perform on-device garbage collection.

collection, whereas the RAIZN array maintains constant throughput through the entire benchmark. Points Ⓐ, Ⓑ, Ⓒ, and Ⓓ mark the points where 20%, 40%, 60%, and 80% of the array capacity has been overwritten. During the first workload, data from five different LBA offsets are mixed and written into the same erase block, so from the red line to point Ⓐ, roughly 80% of each erase block is still populated with valid data that must be copied during garbage collection. As more LBAs are overwritten, this ratio of valid data per erase block gradually decreases, and throughput eventually returns to steady-state shortly after point Ⓓ.

6.2 Performance During Failure

To illustrate the performance of RAIZN in the event of a (single) device failure, we present two benchmarks: First, we compare the sequential and random read throughput and latency of RAIZN compared to `mdraid` during failure. Second, we measure how long it takes to rebuild a replaced device in RAIZN in isolation, demonstrating the TTR before RAIZN can begin to serve (degraded) writes, and how long it takes to restore full performance and fault tolerance for a given amount of data stored on the volume.

Due to the nature of degraded writes, there is no performance penalty associated with serving writes on a degraded array–as such, we omit writes in the following benchmarks and only show the performance of sequential and random reads. The results are shown in Figure 11. All trials were performed with the same parameters as those in Figure 9, with the exception that after pre-filling the RAID/`mdraid` volume with data, the first device in the array was disabled and removed without replacement.

In both degraded workloads, RAIZN performed slightly worse on small (4 KiB) IOs and outperformed on larger IO sizes. This

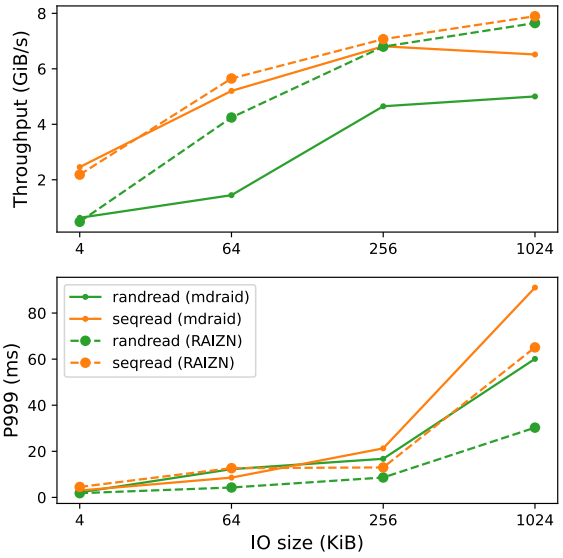

Figure 11: Degraded performance of `mdraid` and RAIZN is comparable.

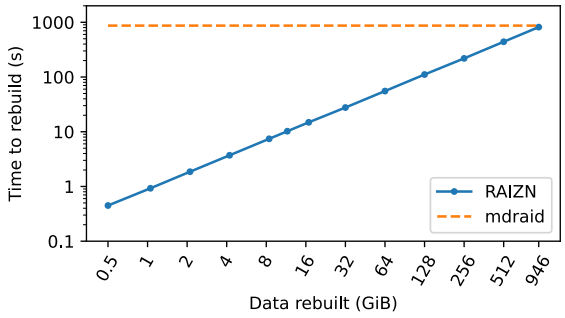

Figure 12: Time to repair (TTR) a replaced device in RAIZN and `mdraid` varying the amount of valid data rebuilt from 64 GiB to approx 946 GiB. RAIZN's TTR scales with the amount of data rebuild, and is bottlenecked by the SSD write throughput. `mdraid` always rebuilds the entire address space, resulting in the same TTR regardless of the amount of valid data present on the array.

experiment shows that RAIZN can achieve roughly equivalent or better performance to `mdraid` in the event of single device failure.

Observation 4: *RAIZN uses the ZNS interface to minimize rebuild time after a failed device is replaced.*

Next, we present the time to repair when replacing a failed device in RAIZN compared to `mdraid`. We ran these experiments on a modified setup, with 960 GiB SSDs of the same make and model instead of 2 TB SSDs. The 960 GiB devices perform similarly to the 2 TB devices. To ensure a fair comparison, the conventional SSDs are formatted with 969.300 MiB (approx. 946 GiB) capacity to match the usable capacity of the RAIZN volume, and the `mdraid` resync rate limit was set sufficiently high to not affect `mdraid`'s resync rate. During rebuild, no IOs are sent to the logical volume,

ensuring both RAIZN and `mdraid` can use the full bandwidth of the underlying devices. RAIZN leverages the ZNS interface to easily identify which stripes must be rebuilt and which can be ignored–any stripes from the beginning of the logical zone up to the write pointer must be reconstructed, while stripes between the write pointer and the end of the logical zone can be ignored. This allows RAIZN to rebuild only valid user data, contrasting with `mdraid` which rebuilds the entire contents of the array regardless of how much data was written to the array.

Figure 12 illustrates the time necessary to restore the volume to full performance and fault tolerance; all data from the non-failed devices in the array is read, then the contents of the failed device are reconstructed and *persisted* to the replacement device. The X axis is not relevant for `mdraid`, as it always resyncs the entire capacity of the replacement drive–however, for RAIZN the X axis describes the amount of user data written to the replacement device (not the amount of user data present on the volume when the rebuild starts).

Both `mdraid` and RAIZN are bottlenecked on the write throughput of the replacement device, and thus require the same amount of time to rebuild a failed device for a completely full volume. However, `mdraid` requires a fixed amount of time to rebuild the array, as it rebuilds the entire address space regardless of how much valid data is present on the volume. In contrast, RAIZN leverages the ZNS interface to rebuild only the fraction of the address space that contains valid data, resulting in rebuild times that scale linearly with the amount of valid data present on the volume.

6.3 Application Benchmarks

Observation 5: *RAIZN achieves similar steady-state throughput and tail latency to* `mdraid` *on RocksDB and MySQL benchmarks.*

We run a suite of RocksDB [11] benchmarks to compare the performance of RAIZN and `mdraid` on real applications, using the optimal array configurations derived from Section 6.1, and formatting each logical volume with the F2FS filesystem. F2FS supports both ZNS and conventional SSDs, with certain optimizations such as threaded logging disabled for ZNS devices [14]. We ran the *fillseq*, *fillrandom*, *overwrite*, and *readwhilewriting* workloads using `db_bench` [12]. All benchmarks perform 100 million operations, with fillseq writing values in sequential key order, fillrandom and overwrite writing values in random key order, and readwhilewriting performing single-threaded random writes concurrently with random reads on 8 threads. After the fillseq benchmark, the array is reset and remounted, then the remaining three benchmarks are run in succession without resetting. In all trials, we pass the `–use_direct_io_for_flush_and_compaction` and `–use_direct_reads` flags to bypass the page cache. For brevity, we show results when running with value sizes of 4000 and 8000 bytes in Figure 13; the overall trend holds for other value sizes.

We also run sysbench [19] on MyRocks [21] using the RocksDB storage engine with MySQL [27] running on top of F2FS formatted `mdraid`/RAIZN. We ran the oltp_read_only, oltp_write_only, and oltp_read_write workloads on a database with 8 tables with 10 million rows each, varying the number of sysbench threads between 64 and 128. Before each trial, the `mdraid`/RAIZN volume is unmounted, all devices are formatted, and mySQL is completely reset. For oltp_read_only, sysbench first prepopulates the database with 8

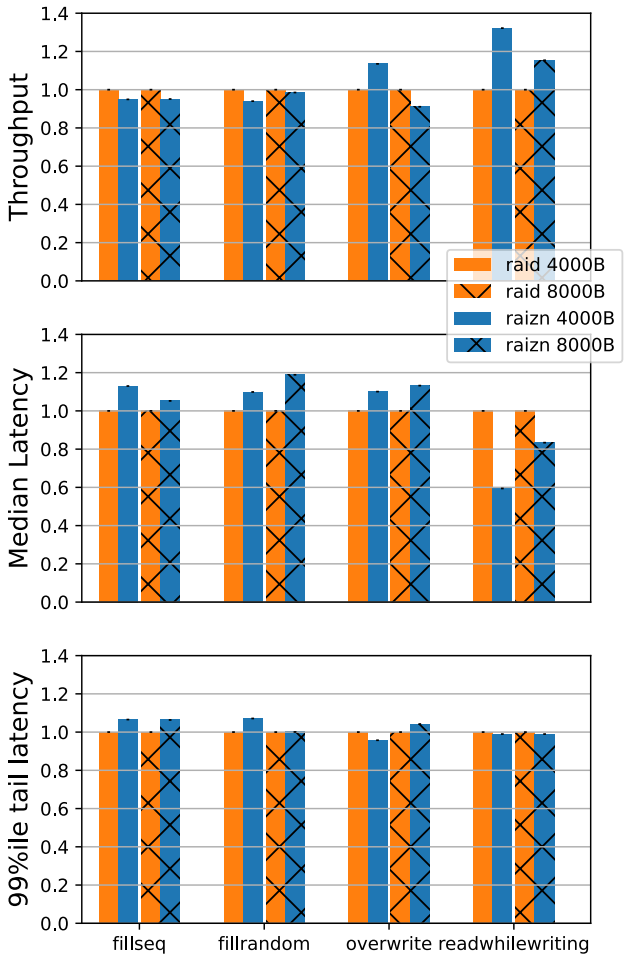

Figure 13: RocksDB performance, normalized. RAIZN achieves throughput and 99th percentile tail latency within 10% of `mdraid`.

tables and 10 million rows, after which it runs SELECT queries for the benchmark duration. oltp_write_only is similar, but does a mix of DELETE, INSERT, and UPDATE queries, and oltp_read_write performs a mix of all four query types. All experiments were run for 600 seconds, and each configuration was run 3 times. Figure 14 shows the results, with the data bar showing the median of these 3 trials and error bars marking the minimum and maximum trial. In almost all experiments, RAIZN performed within error or better than `mdraid` in all three presented metrics (transactions per second, average latency, and 95th percentile tail latency), with the exception of 95th percentile tail latency for the 64-thread oltp_read_write experiment, which was 9.5% higher for RAIZN.

7 RELATED WORK

ZNS is the latest device interface developed to combat the performance and cost penalties of accommodating the block interface on flash-based storage. Other approaches are described here.

Multi-Stream SSDs organize erase blocks into *streams*, in which the host groups writes with similar lifetime [17]. Multi-Stream SSDs

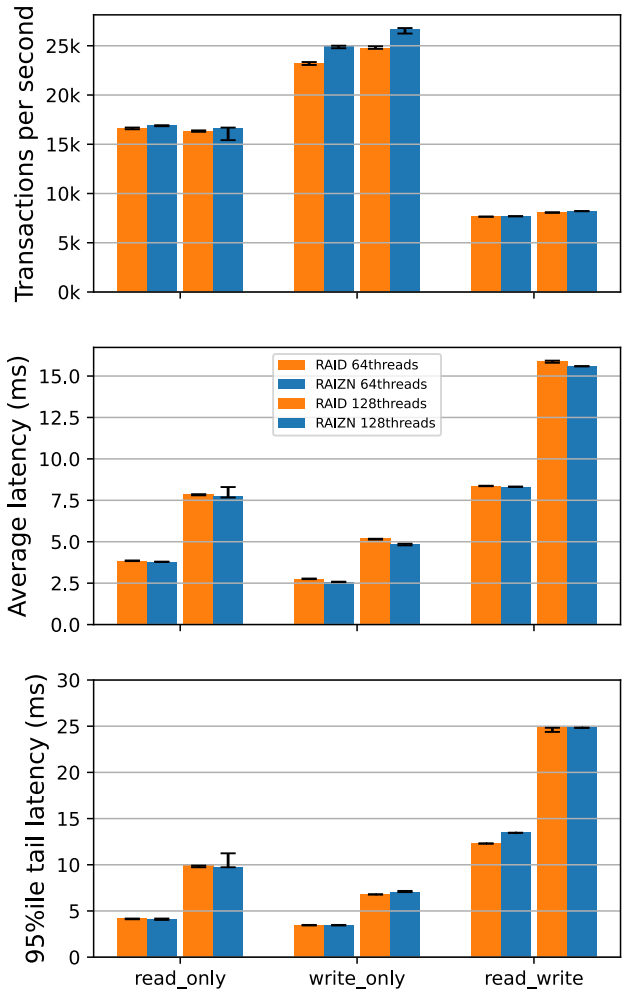

Figure 14: RAIZN achieves similar performance to `mdraid` in sysbench

can significantly reduce the frequency of garbage collection, but cannot eliminate it entirely like ZNS; this is because streams are unbounded in size, so eventually old pages must be cleaned to accept new writes. To accommodate on-device garbage collection, multi-stream SSDs overprovision flash and do not provide cost-per-byte savings over block interface SSDs.

Open-Channel SSDs (OCSSDs) divide flash into fixed-size chunks (*bounded streams*), corresponding to erase blocks which can be written or erased by the host [7]. OCSSDs do not perform garbage collection on-device, obviating the need for overprovisioned flash blocks. A key difference with ZNS is that media-level management such as wear-leveling and media reliability are be handled by the host, necessitating specialized host software for each different SSD model. ZNS SSDs handle media management in firmware.

Application-Managed Flash allows users to directly access segments of flash composed of a fixed set of erase blocks [20]. Each segment is written sequentially and reset as a single unit, behaving somewhat similar to a zone in ZNS. Unlike zones, segment mappings are fixed, and an application-managed flash device cannot remap erase blocks to replace failed ones.

Key-Value SSDs (KVSSDs) forego the traditional block-based addressing scheme, opting instead to associate flash blocks with application-defined keys. Prior work offers RAID-like reliability for KVSSDs [30], but achieving space-efficient parity coding for small objects over an array of KVSSDs has proven difficult [23].

SmartFTL[13] minimizes write amplification when using SSDs as backing storage for cluster filesystems. SmartFTL relies on a host-side library that handles die-level data placement and LBA-to-PBA translation, allowing the host to co-locate data based on expected object lifetime so they are cleaned together.

Shingled Magnetic Recording (SMR) hard disks are similar in many ways to ZNS SSDs, and benefit from a zoned interface due to similarities in the underlying physical media. Overlapping tracks in SMR disks necessitate that large groups of tracks (i.e. zones) must be grouped together and written sequentially.

SMORE [22] is an object store for cold data stored on a SMR disk array. Like RAIZN, SMORE stores important metadata in log structured format on a set of dedicated zones. SMORE is designed for cold objects ranging from a few megabytes to multiple gigabytes in size, whereas RAIZN is designed to work on a wider range of workloads, including small writes and read-heavy workloads.

HiSMRfs [15] is a filesystem designed to achieve high performance on SMR drives. It uses a log structured filesystem for data while storing metadata in random write storage, e.g., SSD. RAIZN exposes a logical device without requiring a separate device for metadata, relying entirely on zoned devices to achieve high performance. As a filesystem, HiSMRfs could be run on a RAIZN volume.

Host-side FTLs. To support unmodified applications atop ZNS SSDs, host-side FTLs such as dm-zap [9] expose a regular block interface using ZNS SSDs. Logical-to-physical block remapping and garbage collection is handled in software. Similar approaches include dm-zoned [25], pblk [7], and SPDK's FTL [24].

8 CONCLUSION

RAIZN provides RAID 5-like performance and reliability for ZNS devices. Several ZNS-specific edge cases, such as partial writes and partial zone resets, make designing ZNS-compatible RAID difficult. RAIZN solves these challenges, ensuring correctness during rare edge cases without sacrificing performance during normal and degraded operation. In contrast to `mdraid`, RAIZN leverages the ZNS interface to rebuild only valid data when a device in the array is replaced, minimizing the time to repair and by extension the performance impact of rebuilding. RAIZN is able to achieve latency characteristics similar to `mdraid` for conventional SSDs, even when the conventional SSDs are not impacted by garbage collection overheads, and throughput within 2% of using the underlying SSDs directly. More importantly, RAIZN preserves the benefits of ZNS, allowing uninterrupted steady performance up to 14× higher for an extended read/write workload when compared to RAID-for-conventional-SSDs suffering from on-device garbage collection. Finally, RAIZN achieves similar performance to industry standard mdraid on real applications such as RocksDB and MySQL.

ACKNOWLEDGMENTS

We thank our shepherd and five anonymous reviewers for their invaluable feedback and guidance through the revision process. We also thank Western Digital for providing resources and technical expertise; we especially thank Damien Le Moal, Hans Holmberg, Niklas Cassel, Dmitry Fomichev, Andreas Hindborg, and Dennis Maisenbacher for their technical help with the ZNS SSDs and associated kernel software stack. We thank Jason Boles from PDL for setting up and maintaining our hardware deployment. We also thank the members and companies of the PDL Consortium (Amazon, Google, HPE, Hitachi, IBM, Intel, Meta, Microsoft, NetApp, Oracle, Pure Storage, Salesforce, Samsung, Two Sigma, Western Digital) and VMware for their interest, insights, feedback, and support. This work was sponsored by Western Digital, and was supported in part by NSF grant CNS1956271, and gifts from the NetApp University Research Fund, a corporate advised fund of Silicon Valley Community Foundation, and Meta Platforms, Inc.

REFERENCES

[1] 2020. Device Mapper. https://docs.kernel.org/admin-guide/device-mapper/index.html.
[2] 2021. NVM Express® Base Specification Revision 2.0 May 13th, 2021. https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2_0-2021.06.02-Ratified-4.pdf.
[3] 2022. Explicit volatile write back cache control. https://www.kernel.org/doc/Documentation/block/writeback_cache_control.txt.
[4] 2022. mdadm(8) - Linux man page. https://linux.die.net/man/8/mdadm.
[5] Jens Axboe. 2017. fio - Flexible I/O tester rev. 3.27. https://fio.readthedocs.io/en/latest/fio_doc.html.
[6] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, DL Moal, G Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21)*.
[7] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. 2017. Lightnvm: The linux open-channel {SSD} subsystem. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*. 359–374.
[8] Neil Brown. 2015. A journal for MD/RAID5. https://lwn.net/Articles/665299/.
[9] Western Digital Corporation. 2021. dm-zap: Host-Side Zoned Host Translation Mapper. https://github.com/westerndigitalcorporation/dm-zap.
[10] Western Digital Corporation. 2022. Zoned Storage. https://zonedstorage.io/.
[11] Facebook. 2015. RocksDB. http://rocksdb.org/.
[12] Facebook. 2021. Performance Benchmarks. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.
[13] Google. 2021. SmartFTL Architecture for SSDs. https://www.youtube.com/watch?v=3O3zDrpt3uM.
[14] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Joo-Young Hwang. 2021. ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*. 147–162.
[15] Chao Jin, Wei-Ya Xi, Zhi-Yong Ching, Feng Huo, and Chun-Teck Lim. 2014. HiSMRfs: A high performance file system for shingled storage array. In *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 1–6.
[16] Hai Jin, Xinrong Zhou, Dan Feng, and Jiangling Zhang. 1998. Improving partial stripe write performance in RAID level 5. In *Proceedings of the 1998 Second IEEE International Caracas Conference on Devices, Circuits and Systems. ICCDCS 98. On the 70th Anniversary of the MOSFET and 50th of the BJT. (Cat. No.98TH8350)*. 396–400. https://doi.org/10.1109/ICCDCS.1998.705871
[17] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The multi-streamed solid-state drive. In *6th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*.
[18] Jaeguk Kim. 2020.
[19] Alexey Kopytov. 2012. Sysbench manual. *MySQL AB* (2012), 2–3.
[20] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, et al. 2016. Application-managed flash. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*. 339–353.
[21] Percona LLC. 2021. Percona MyRocks Introduction. https://www.percona.com/doc/percona-server/8.0/myrocks/index.html.
[22] Peter Macko, Xiongzi Ge, J Kelley, D Slik, et al. 2017. SMORE: A cold data object store for SMR drives. In *Proc. 34th Symp. Mass Storage Syst. Technol.(MSST)*.
[23] Umesh Maheshwari. 2020. StripeFinder: Erasure Coding of Small Objects Over Key-Value Storage Devices (An Uphill Battle). In *12th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*.
[24] Wojciech Malikowski. 2018. SPDK Open-Channel SSD FTL. https://spdk.io/doc/ftl.html.
[25] Damien Le Moal. 2017. dm-zoned: Zoned Block Device device mapper. https://lwn.net/Articles/714387/.
[26] Jeff Moyer. 2017. libaio. https://pagure.io/libaio.
[27] Oracle. 2022. MySQL. https://www.mysql.com/.
[28] 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.
[29] Rekha Pitchumani and Yang-suk Kee. 2020. Hybrid data reliability for emerging key-value storage devices. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*. 309–322.
[30] Rekha Pitchumani and Yang-Suk Kee. 2020. Hybrid Data Reliability for Emerging Key-Value Storage Devices. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA, 309–322. https://www.usenix.org/conference/fast20/presentation/pitchumani
[31] Ohad Rodeh and Avi Teperman. 2003. zFS-a scalable distributed file system using object disks. In *20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings.* IEEE, 207–218.
[32] Marta Rybczyńska. 2021. Btrfs on zoned block devices. https://lwn.net/Articles/853308/. (2021).
[33] Kent Smith. 2011. Garbage collection. *SandForce, Flash Memory Summit, Santa Clara, CA* (2011), 1–9.
[34] Hermann Strass. 2016. An Introduction to NVMe. https://www.seagate.com/files/www-content/product-content/ssd-fam/nvme-ssd/nytro-xf1440-ssd/_shared/docs/an-introduction-to-nvme-tp690-1-1605us.pdf.
[35] Saifeng Zeng, Ligu Zhu, and Lei Zhang. 2013. A High Reliable and Performance Data Distribution Strategy: A RAID-5 Case Study. In *2013 Ninth International Conference on Computational Intelligence and Security*. 318–322. https://doi.org/10.1109/CIS.2013.74

Received 2022-07-07; accepted 2022-09-22