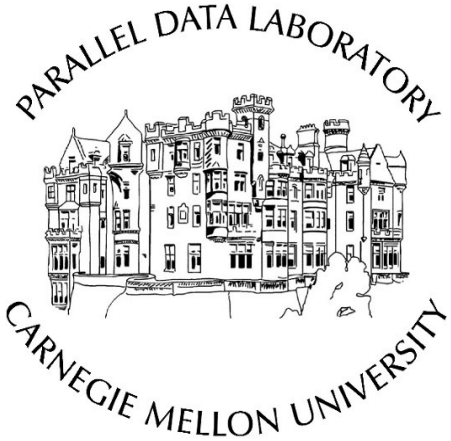


RAIZN: Redundant Array of Independent Zoned Namespaces

Thomas Kim*, George Amvrosiadis*, Jekyeom Jeon*, Huaicheng Li*,
David G. Andersen*†, Greg Ganger*, Michael Kaminsky*†, and Matias Bjørling‡
*Carnegie Mellon University †BrdgAi ‡Western Digital Corporation

CMU-PDL-22-101

January 2022

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Zoned Namespace (ZNS) SSDs are the most recent evolution of host-managed flash-based storage, enabling improved performance at a lower cost-per-byte compared to traditional block interface SSDs. To date, there is no support for arranging these new devices in redundant arrays (RAID), which may limit their deployment in environments where this is the favored mechanism for increasing reliability and throughput. This paper identifies key challenges in the design of a RAID-like mechanism for ZNS SSDs, such as the requirement to manage metadata updates and persist partial stripe writes in the absence of overwrite semantics in the device's interface. We present the design, implementation, and evaluation of RAIZN, a logical volume manager that exposes a ZNS interface and stripes data and parity across ZNS SSDs.

Experiments show that RAIZN provides full expected performance from the aggregate device set, successfully addressing the key challenges from the ZNS interface. RAIZN achieves throughput and latency comparable to the equivalent Linux software RAID implementation running on conventional SSDs that use the same hardware platform, and then RAIZN exceeds its performance once device-level garbage collection inhibits the conventional SSDs. Importantly, RAIZN retains ZNS's opportunities for increased application performance, allowing higher-level software (e.g., F2FS or RocksDB) to carefully control garbage collection. This allows, for example, RAIZN to maintain consistent performance under scenarios where conventional SSD arrays experience up to 87.5% throughput drop due to device-level garbage collection.

Acknowledgements: We thank the members and companies of the PDL Consortium (Amazon, Google, Hewlett Packard Enterprise, Hitachi, Ltd., IBM Research, Intel Corporation, Meta, Microsoft Research, NetApp, Inc., Oracle Corporation, Pure Storage, Salesforce, Samsung Semiconductor Inc., Seagate Technology, Two Sigma, and Western Digital) for their interest, insights, feedback, and support.

Keywords: Storage, Reliability, Zoned Storage, ZNS, RAID

1 Introduction

The NVM Express Zoned Namespace (ZNS) standard [9, 7] has recently been introduced as an alternative to the block interface that has been the de-facto storage device interface for decades. ZNS represents a new division of functionality between the host and device, transferring responsibilities such as page-level logical block address mapping and garbage collection from the SSD firmware to the host. The ZNS standard allows the host and device to collaborate on data placement, allowing applications to unlock increased performance compared to traditional FTL-based block interface SSDs.

These benefits do not come for free, however, as the ZNS interface enforces stricter semantics on writes. ZNS drives divide their address space into large contiguous *zones*, each of which must be written sequentially and reset as a single unit [1]. Prior work by Bjørling et al. [9] provides an apples-to-apples comparison between ZNS and conventional SSDs, showing the efficacy of ZNS SSDs in single-drive applications. However, in datacenters, drives are often deployed in arrays, requiring new software support for data striping and redundancy over arrays of ZNS drives. One standard solution for managing arrays of disks is RAID—RAID transparently aggregates drives to deliver higher performance and higher reliability through erasure coding. This paper presents a new system called RAIZN (Redundant Array of Independent Zoned Namespaces). RAIZN achieves increased reliability and performance for ZNS devices, adapting RAID-like mechanisms for use with ZNS SSDs. We implement RAIZN as a Linux device mapper (dm) device that allows ZNS devices to be configured in an array and exposed to hosts as a single ZNS device, transparently providing redundancy and striping. Prior systems exist for RAID-like redundancy for other emerging storage technologies such as KVSSD [24], but no specialized solution exists for ZNS SSDs. RAIZN behaves in a manner similar to software RAID—the logical device accepts IO from the host application, erasure-codes it, and stripes it across all of the underlying drives. Its novelty comes in retaining the promise of ZNS while addressing challenges introduced by building atop ZNS devices, transparently providing redundancy for ZNS-specialized host applications, including key-value stores (e.g., RocksDB) and filesystems (e.g., F2FS).

The core goal of RAIZN is to implement RAID in a way that retains the performance and cost benefits of the zoned interface. Prior work [9] has suggested that applications specialized to run on ZNS SSDs can achieve a 90% reduction in 99.99^{th}-percentile tail latency and 2× higher write throughput compared to the equivalent workload on conventional SSDs.

The key challenges in designing RAIZN are caused by two main factors: 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 drives. Immutability poses a challenge in handling small writes, as parity cannot be updated once it is written. The lack of atomicity results in many subtle problems when it is combined with the zoned interface. A side effect of the sequential-write-only zone interface is that data is guaranteed to be persisted in sequential order; data at a particular Logical Block Address (LBA) cannot be reported as being persisted until data at the preceding LBAs is persisted, even after power loss. Conforming to this guarantee in RAIZN without hurting performance is made even more difficult by the immutability of data once it has been persisted to the underlying drives. For example, if a stripe of data is only written to a subset of the drives before the system loses power, RAIZN cannot naively allow reading of the data, nor can it overwrite the data that has already been written. We elaborate on these challenges in Section 3 and explain how RAIZN addresses them in Section 4.

We design RAIZN to solve these challenges while taking advantage of the zoned interface to optimize its design. We compare the performance of RAIZN on ZNS drives to that of conventional software RAID on near-identical conventional SSDs. RAIZN provides RAID-like reliability on ZNS devices while achieving maximum read and write throughput within 1% of the aggregate raw device throughput. RAIZN achieves similar throughput, median, and tail latency to the optimized Linux RAID 5 implementation [5] when the conventional drives are not suffering from garbage collection. When garbage collection does impair

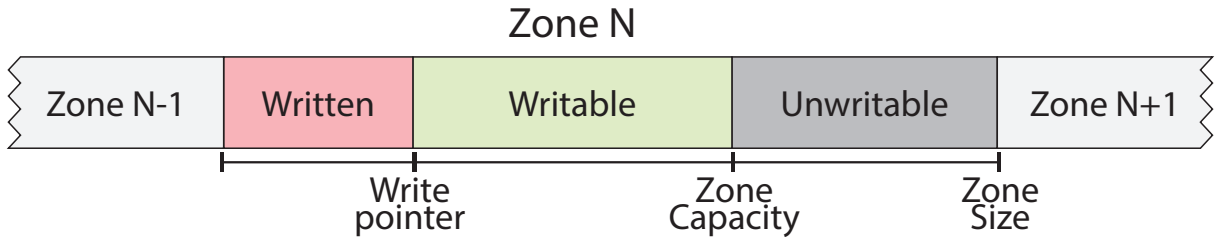


Figure 1: Each zone accepts writes at the write pointer, and can be written up to the zone capacity. Zone size ensures zones start on a power of two address, regardless of the capacity.

conventional drive performance, RAIZN allows ZNS-specialized host applications to deliver significantly higher performance.
RAIZN will be made available as open-source software.

2 Background

This section provides the necessary background knowledge about the ZNS interface and ZNS SSDs to understand the design of RAIZN. We present a brief overview of the key features of the ZNS interface which created both opportunities and challenges in designing RAIZN.

2.1 The ZNS Interface

There are several key differences between the ZNS and block interface standards, which we outline in this section.

Sequential write only zones. Similar to the block interface, the ZNS standard defines its address space as a collection of logical blocks, which are addressable by their logical block address (LBA). These logical blocks are grouped into *zones*, which must be written sequentially, and cannot be overwritten unless the entire zone is erased by an operation called *zone reset*. These zones are how ZNS devices expose their address space to users, with applications expected to structure their IO in a way that conforms with the sequential write constraint. Multiple writes can be submitted to the same zone as long as they are submitted in order, enabling high throughput when writing to a single zone. A ZNS drive maintains a *write pointer* for each zone, which can be queried by an application to figure out the next writable LBA in that zone.

Zone capacity. Zones sizes in ZNS are always set to 2^N bytes to ensure compatibility with the Unix storage stack, but may not necessarily represent the equivalent number of underlying storage blocks. The subset of each zone that is writable is called the *zone capacity*. Figure 1 illustrates how a zone is laid out, with a set of unwritable LBAs from the zone capacity until the end of the zone. For example, the ZNS SSDs used in our evaluation have a zone size of 2GiB and a zone capacity of 1077MiB; this means the first block of each zone starts at a multiple of 2GiB, and the upper 971MiB of the address space of each zone cannot be read or written as it is not backed by physical media. Zone capacity is determined by the layout of the physical media, and zone size is simply the smallest 2^N that is greater than the zone capacity.

Zone append. In contrast to typical write commands where the host specifies the LBA at which to write data, the ZNS standard also defines the *zone append* command, which allows the host to write data to a specified zone, receiving the precise LBA at which the data was written after the IO completes. Moreover, zone appends are not guaranteed to be persisted in the order they were submitted to the device.

ZNS state machine. In addition to a write pointer, each zone in ZNS has an associated status describing the

current condition of the zone. A zone can be empty, implicitly open, explicitly open, closed, full, read-only, or offline. While explaining each of these states in detail is outside the scope of this paper, we briefly outline the states that are important to our system.
A zone starts in the *empty* state, and returns to the empty state every time it is 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 drives was 14.
An open zone becomes *full* when the last writable block in that zone is written. Finally, the read-only and offline states are failure states, transitioned to when enough erase blocks fail that a zone cannot be fully used anymore. Typically, in ZNS SSDs, zone failure will only occur if the drive has reached the end of its life. As erase blocks fail, they are replaced by overprovisioned blocks, and only once the drive runs out of extra blocks do zones start failing.

2.2 Application compatibility

Most applications that rely on the block interface cannot run directly on ZNS drives without some modification of the software stack. However, there are existing solutions to abstract away the ZNS interface and allow current applications to run on ZNS hardware. However, end-to-end integration of the ZNS interface into applications is the ideal way to take advantage of the various performance and cost benefits of ZNS drives.
Host-side FTLs such as dm-zap [3] expose a conventional block interface using ZNS SSDs, by handling logical-to-physical block remapping and garbage collection in software. Similar approaches include dm-zoned [22], pblk [10], and SPDK's FTL [21].
Filesystems are beginning to add ZNS SSD support, starting with F2FS [17] in kernel 5.10 [2] and btrfs [26] in kernel 5.12 [27].

2.3 RAID

RAID refers to a family of data distribution techniques that aim to achieve redundancy, performance, and availability by leveraging multiple physical drives [23]. In modern systems, RAID is often implemented as a logical device that can be treated as a normal block device, or is sometimes integrated into the filesystem. In our work, we implemented a RAID-5-like reliability mechanism to distribute data across ZNS SSDs. Several aspects of RAID are incompatible with zoned interfaces, which we elaborate on in Section 3–for example, the simple stripe to device address translation runs into problems due to the immutability of data once it is written to a zone. In addition, extra care must be taken to ensure consistency after power loss to ensure that the RAID device conforms with the ZNS standard.

3 Challenges of Zoned Device Arrays

In this section, we describe several cases that show how the zoned interface can be problematic for a conventional RAID-like setup. RAID organizes data into stripes, which are then further split into stripe units – these stripe units are distributed across all drives, along with parity. Each stripe unit is mapped arithmetically to an address on a particular drive, determined by the parameters provided when initializing the RAID.

Small writes. Writes that are smaller than a stripe are a performance problem in conventional RAID due to parity updates [15], but when dealing with ZNS drives this evolves from a performance issue to a correctness issue. In ZNS drives, LBAs that are written cannot be changed until the entire zone containing them is reset, meaning that it is impossible to update parity after a small write. However, the partially calculated

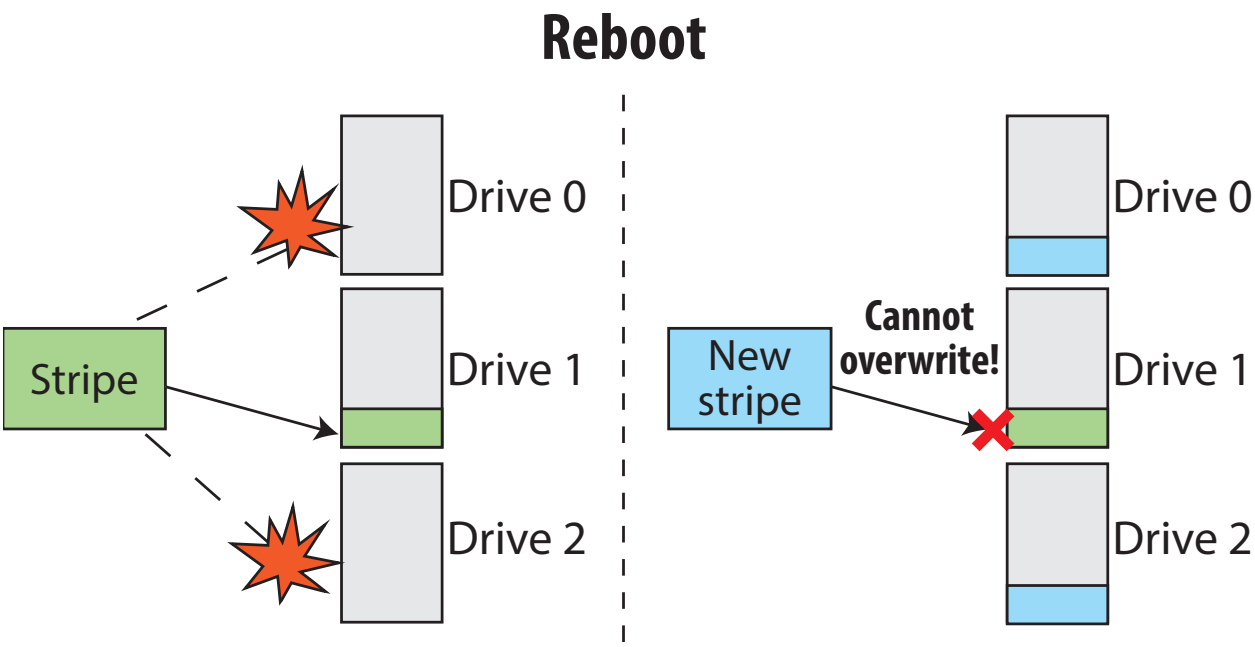


Figure 2: Only a subset of the stripe units are persisted before power is lost. The next stripe write cannot place its data at the correct PBA due to the written stripe unit.

parity must be persisted before notifying the host application of IO completion, otherwise there is a chance of user-visible data loss.

Metadata management. Unlike conventional drives where metadata can simply be overwritten when it is updated, metadata must be log structured when stored in a ZNS device. This adds complexity and necessitates garbage collection for metadata logs.

One key difference between ZNS and block interface is that in ZNS, a write completion at a given LBA implies that all preceding LBAs in the same zone were also written. Put differently, a write cannot be completed until all previous writes within the same zone are also completed. This dependency between writes in a zone leads to several interesting edge cases when dealing with power failure in RAID.

Partial stripe writes. Figure 2 illustrates a scenario that is problematic when working with ZNS drives: suppose that only a subset of the stripe units in a stripe are persisted before power loss, and this subset of stripe units is insufficient to recover the entire stripe after reboot. In a conventional RAID, it is simple to just treat this stripe as if it had never been written, and allow the user to overwrite it as necessary. However, the stripe units that have been persisted cannot be overwritten without resetting the entire zone. As a result, the traditional direct mapping of the RAID address space to drive address space is insufficient to support ZNS, and an additional layer of indirection is required. This layer of indirection poses a challenge in designing RAIZN to handle uncommon edge cases like this without harming performance.

Partial zone resets. Similar to the problem of power loss after partial stripe writes, it is necessary to take care when resetting a zone. Assuming a RAID-like direct mapping between RAID zones and device zones, a user request to reset a RAID zone would be translated into a reset on the corresponding device zone on each device in the array. However, it is impossible to atomically reset zones on multiple independent drives, so there is the possibility that the system can experience a power loss after resetting only a subset of the device zones. In most cases, it is possible to detect that this scenario occurred by examining the write pointers on each of the device zones – if one device has an empty zone while a different device has a full zone, it is clear that a zone reset got interrupted. However, there are some edge cases: suppose the host application issues a

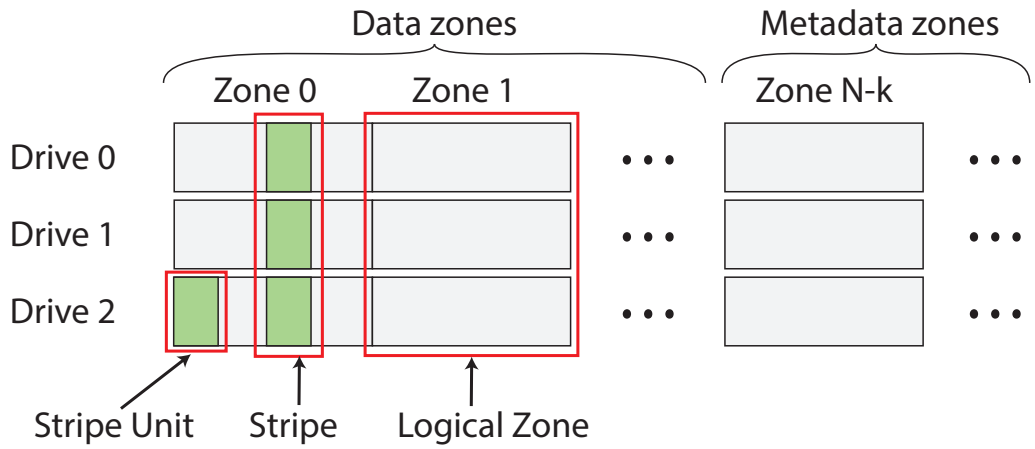


Figure 3: RAIZN data layout for two data and one parity drive. Logical zones consist of physical zones, one per array drive.

zone reset to a RAID zone, but only a single device zone is reset before power is lost. On top of this, after this single device zone is reset, it is completely filled with new stripe units before any other device zone starts resetting zones to make them available. If power is lost at this point, it is impossible to detect that an error has occurred. As a result, it is necessary to wait until all device zones are reset before issuing new writes to any of the device zones.

Even with this constraint, however, there is still a scenario where a partial zone append can result in an ambiguous state on startup. Suppose the host application requests to reset a RAID zone with only one stripe of data written. If power is lost after only the parity and last stripe unit in the stripe is reset, it is impossible to distinguish this from a simple partial stripe write, yet this state violates the atomicity of zone resets.

We describe our solutions to each of these challenges in Sections 4 and 5.

4 RAIZN Design

RAIZN is designed to appear as a single, host managed zoned virtual device; the fact that it is a device array is transparent to host applications. A core question to the design of RAIZN is address space management: how the physical locations on the underlying 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 SSDs. 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). When a host application, such as a filesystem, submits an IO request to RAIZN, the request's LBA is translated into a set of PBAs, and the data is striped and erasure coded before being submitted to the physical drives.

Figure 3 illustrates how the address space of the physical drives is organized with regards to RAIZN. The physical zones are partitioned into two groups, *data* and *metadata* zones. The number of metadata zones is configurable, with a required minimum of 3 per drive (Section 4.2). Data zones are organized into *logical zones* consisting of physical zones, one per device in the array. For simplicity, we map physical zone N of each device into logical zone N, and assume that all drives in the array have the same zone size and capacity. RAIZN presents each of these logical zones to the host application, as a single sequential-write only zone. The capacity of each logical zone is the sum of the capacities of each underlying physical zone, and the size of the logical zone is the capacity of the logical zone rounded up to the nearest power of two. Logical zones conform to the ZNS zone specification, and are treated by host applications in the same way as zones on

ZNS devices. On startup, RAIZN calculates the status of each logical zone; if any device zone in the logical zone reports an offline, read-only, or closed status, that status is used for the logical zone. Otherwise, the logical zone is marked as empty or full if all of the underlying device zones are empty or full respectively.
Data written to logical zones is organized into stripes, which are in turn composed of stripe units. Each stripe contains one stripe unit per device, and all of the stripe units for a given stripe are persisted to zones within the logical zone. A stripe contains one or more *data* stripe units and one or more *parity* stripe units. The data stripe units hold user data, and the parity stripe units contain the necessary parity data to recover the contents of a data stripe unit from a failed drive.
In this section, we present our design for RAID on ZNS devices, and describe the various optimizations to provide reliability and availability with low overhead. We start by presenting a base design that supports zone writes, then discuss various extensions and optimizations, including zone append support, in Section 5.2.

4.1 Data Zones

In RAIZN, we use a scheme for data placement similar to that of conventional RAID, but extend it to support ZNS-specific edge cases. Each LBA is statically mapped to a particular device and PBA using a simple arithmetic translation. User data is organized into stripes, which are divided into stripe units (also referred to as "chunks" in linux RAID), erasure coded, then distributed across the drives 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 drive. This allows reads to be translated without requiring any additional memory lookups, ensuring that RAIZN has minimal impact on IO latency.
One key reason for using this simple LBA to PBA mapping in RAIZN is to provide similar behavior of RAIZN zones to real ZNS SSD zones; if multiple pieces of data with similar lifespan are located in the same zone in a RAIZN volume, the user application can be assured that no additional garbage collection or indirection is occurring, and that the data is laid out on the physical media in a similar manner to how it would be when running directly on a physical ZNS device.
We also enforce write completion ordering within zones – the host is only notified of the completion of a write request if all of the previous write requests within that zone have been successfully persisted to the underlying drives. This is required to ensure that any write that is reported as complete to the host will be available to read even in the event of sudden power loss. If we did not enforce this constraint, it would be possible to report a write completion for a stripe unit following a stripe that was only partially written. In that case, any data at a LBA higher than the hole, including the stripe unit that was reported as persisted, must be invalidated and discarded.

Parity logging for small writes. Small writes pose a problem for RAIZN due to the sequential-write-only constraint of ZNS devices, specifically when calculating and writing parity. It is important that RAIZN does not report to the user that a write was successful until enough data and parity is persisted to recover the user's data following the failure of an SSD. A small write can be immediately written to the corresponding data device, but the parity is unknown until the entire stripe is written by the host application.
To solve this problem, we introduce the *parity log*, a per-device log that logs partial parity. This enables RAIZN to maintain fault tolerance in the presence of fine granularity writes. In addition to the partial parity data itself, each parity log entry contains a header which includes information about the small write request itself. An additional mechanism is also required to detect when parity logs are invalidated, such as after the corresponding logical zone is reset. Our solution is to assign a new unique ID to each logical zone when it is opened, which we call the *generation hash*, and tag each parity log entry with the generation hash. Parity log entries for an invalidated generation hash are discarded instead of being replayed when the RAIZN device is remounted.

Write-ahead logging for partial zone resets. As we mentioned in Section 3, the immutability of written data in ZNS results in subtle problems when power failures are introduced. One of these problems is partial zone resets, which is when only a subset of the zones in a logical zone are reset before power is lost. First, we *plug* a logical zone until all zones in the zone group are fully reset, blocking any further writes to that logical zone until the logical zone is unplugged. While this does imply a potentially large overhead, it is unlikely that applications will block while waiting for zone resets, because zone resets in ZNS are often time consuming and expensive operations. Plugging a logical zone until all of the underlying physical zones are reset is unlikely to significantly affect end-to-end performance, but greatly simplifies consistency and fault tolerance.
We also use write-ahead logging for zone resets if the logical zone has only one or a fraction of a stripe written to it, to solve the ambiguity described in Section 3. This results in significant overhead for the specific case where a logical zone has fewer than one stripe written to it, but it is unlikely that a realistic workload would frequently reset zones that have only a small amount of data written to them.

Relocation maps for partial stripe writes. Partial stripe writes also cause problems for this simple data placement scheme. Figure 2 illustrates this scenario: Suppose a stripe is written to the RAID device, but the system loses power before all of the stripe units are persisted. Upon reboot, there is not enough data to repair the missing stripe unit, so RAIZN must behave as if the entire stripe failed to persist. Importantly, at least one stripe unit did get persisted successfully before power loss. In a normal RAID, the presence of this persisted stripe unit is not a problem, as an additional stripe write at the same LBA can simply overwrite the corresponding PBAs regardless of whether they were written or not. However, ZNS does not allow overwriting the data that was already persisted, so the new data must be placed elsewhere.
We solve this problem by *relocating* any stripe units that cannot be written to a metadata zone. If a partial stripe write causes a stripe unit to be unwritable at the desired PBA, we instead write it to a metadata zone on the affected device, tagging it with the corresponding LBA and generation hash. We then store this LBA to PBA remapping in a hashmap which we call the *relocation map*. To prevent ambiguity regarding the validity of relocation map entries, we tag each relocation map entry with the generation hash, enabling RAIZN to quickly detect if a relocation map entry was invalidated by a zone reset. The relocation map, while providing a solution to the partial stripe write problem, represents a significant overhead if it must be checked on every read. RAIZN prevents this rare case from impairing performance by marking each logical zone with a flag indicating whether any remappings have occurred, and checks the relocation map only if the flag is set. In addition, relocated stripe units are cached in memory, as the relative infrequency of relocations combined with the immutability of the data once written makes it feasible to cache the stripe units in memory to simplify garbage collection.
One last subtle problem that occurs as a result of the ZNS interface is that writes cannot be reported to the user until all preceding writes have been persisted. That is to say, if a host application asynchronously submits 2 zone writes, RAIZN must ensure that all stripe units associated with the first zone write are persisted before notifying the user of IO completion. If the second zone write is reported as complete, then the system loses power before the first zone write is fully persisted, this results in a situation where the first zone write is unrecoverable, violating the ZNS specification.

4.2 Metadata in RAIZN

Unlike RAID, RAIZN cannot store and update metadata, such as the superblock, in a fixed location, because ZNS disallows the modification of written blocks. In addition, reserving a entire zone for a small metadata structure is wasteful, with each zone on our SSDs having over a gigabyte of capacity. In RAIZN, we persist metadata to metadata zones in a log structured update format to conform with the sequential write constraint, and we store multiple varieties of metadata logs within a single metadata zone to reduce wasted capacity.

It is necessary to reserve at least one additional empty zone per drive as a metadata zone to facilitate log truncation and garbage collection; we term these zones the *swap* zones.
A subset of the metadata must be replicated across all drives in the array; this includes RAID parameters, drive ID assignments, generation hashes, and zone reset write-ahead logs. The remainder of metadata is associated with a particular drive, and is written to that drive without replication – this includes parity log entries, remapped stripe unit contents, and the relocation map. In the event that a drive fails, the non-replicated metadata stored on that drive is no longer of any use, and the loss of that metadata is inconsequential.
All metadata updates are stored in log structured format, and a single metadata zone can hold every variety of metadata. However, the majority of metadata in RAIZN is updated very infrequently, with the exception of parity logs. Parity logs are invalidated as soon as the associated full stripe is persisted, and new parity logs are generated every time the host writes data that is not aligned to start and end on a stripe boundary. Depending on the workload, this can be quite frequent, so we take special care in isolating the rest of the system from the effects of parity logging by placing parity logs into their own separate metadata zone. This minimizes garbage collection overhead, but also results in RAIZN requiring at least three metadata zones per device: one for parity logs, one for all other metadata, and one swap zone for garbage collection.
All valid metadata in RAIZN is cached in memory, and the on-disk persistent copy is only read when the volume is remounted. In our experiments, valid metadata is typically on the order of 192–1024KiB, primarily consisting of cached partial parity. In addition, all metadata updates are persisted using zone appends, ensuring high throughput even in the presence of many concurrent small metadata log writes.

4.3 Garbage collection

Metadata in RAIZN must be periodically garbage collected to free up space in the metadata zones. We use swap zones to facilitate garbage collection without interrupting operation of the system. All key metadata in RAIZN, is maintained as an in-memory data structure accompanied by a persistent on-disk log, so garbage collection can serialize the in-memory copy to disk without incurring any SSD reads. When a metadata zone becomes full, we designate a swap zone to replace it, and immediately direct all new log entry writes to that swap zone. We then asynchronously scan the in-memory metadata structures and persist any valid entries to the swap zone. After this process is complete, the original metadata zone is reset, and can serve as a swap zone for the next garbage collection pass. By waiting for the metadata zone to become full before starting garbage collection, we ensure that RAIZN only needs to reserve two open zones for metadata, maximizing the number of zones the user is allowed to open simultaneously.
For the parity log zone, RAIZN can simply check the write pointer of every open and closed logical zone – if the write pointer is not aligned on a stripe boundary, the contents of the in-memory buffer caching the partial stripe is to be XORed and the result is logged to the swap zone.
All other metadata, such as the generation hashes, static parameters, array configuration, relocated stripe units, and zone reset logs, can be asynchronously serialized to the swap zone.

4.4 Fault tolerance

Fault tolerance is fairly straightforward in RAIZN, with only small deviations from the behavior of conventional RAID. Degraded reads and writes are handled in the same way as conventional RAID, with reads handled by reading parity and calculating the missing stripe unit and writes simply omitting the missing stripe unit.
One difference from conventional RAID is the rebuild process – in the event a failed drive is swapped out and replaced by a new drive, RAIZN rebuilds the new drive zone by zone. During rebuild, reads can continue to be served as degraded reads without issue, but writes cannot complete until the corresponding zone is fully rebuilt, due to the sequential write only constraint. RAIZN starts the rebuild process by blocking all

writes to currently open or closed zones, then rebuilds these zones first, unblocking each zone after it is fully rebuilt; this allows RAIZN to continue serving writes after a short delay.

5 Implementation

In this section, we describe the specifics of how RAIZN is implemented, and highlight some of the more interesting implementation details. We implement RAIZN as a dm (device mapper) logical device, and any ZNS-compatible application which uses the kernel block layer can perform IOs on a RAIZN device. The device mapper enables the mapping of physical block devices onto the address space of logical block devices, allowing the logical device to modify IO requests before submitting them to the underlying physical drives. Upon receiving an IO, RAIZN stripes the request, calculates parity if the IO is a write, and translates the address from LBA space to PBA space before resubmitting it to the block layer. To distinguish between host application IOs and RAIZN IOs, we prefix reads and writes from the host with *logical*, and prefix IOs submitted by RAIZN with *SSD* (e.g. logical writes, SSD reads).

Write submission and completion order. We divide the processing of each received IO in RAIZN into two logical steps: preprocessing and completion. Each IO is first preprocessed–if necessary, it is divided into smaller IOs, parity is calculated, additional parity and metadata IOs are generated, and partial stripe data is cached. Once the preprocessing is complete, RAIZN submits the IO to the kernel block layer and begins processing the next IO request. Once the IO is completed, the `endio` callback is invoked by the kernel block layer, and RAIZN begins the completion phase. In most cases, the completion phase is simply a matter of waiting until all additional generated IOs are complete before reporting IO completion to the user. However, as mentioned in Section 4.1, it is sometimes necessary to delay reporting write completion until all previous writes within the logical zone have been persisted. If the write completion needs to be delayed for a specific IO request, RAIZN pushes the request onto a queue, and a dedicated thread will repeatedly check until write completion ordering is fulfilled and the IO can be completed. Our testing found that at worst, <0.01% of writes were delayed to compensate for write completion ordering.

RAIZN tracks which writes have been persisted to the physical drives by maintaining a bitmap for every open logical zone, called the *persistence bitmap*. Each bit represents a stripe unit, and there is one bit per drive per stripe unit. When an SSD write completes, if it wrote the last sector of a stripe unit, the bit in the bitmap corresponding to that stripe unit is set. Note that it is not necessary to track partial stripe unit writes, as writing to a particular PBA implies that all previous PBAs in zone have been persisted.

The persistence bitmap, despite its name, is only stored in memory and does not have to be persisted. On volume remount, the persistence bitmap can be reconstructed based on the write pointers of each of the devices in the array. For a 5-drive 8TiB RAIZN volume using the smallest possible stripe units (4KiB), the total memory footprint for the persistence bitmaps is approximately 256MiB. When considering a more realistic deployment scenario using 32 or 64KiB stripe units, the total memory footprint drops to 32 or 16MiB respectively.

When all of the SSD writes for a logical write complete, the corresponding persistence bitmap is checked to see if all of the previous logical writes were persisted. RAIZN takes advantage of the sequential write constraint of ZNS zones by only checking the persistence bitmap starting from the beginning of the stripe immediately preceding the current write. If all of the stripe units from the beginning of the preceding stripe until the start of the current write have been persisted, RAIZN can safely assume that all stripes from the beginning of the zone until that preceding stripe have also been persisted.

If a logical write starts in the middle of a stripe unit, the fact that it was successfully persisted implies that the beginning of the stripe unit was persisted, as all sectors in a stripe unit are written to the same SSD. This means that the persistence bitmap can correctly track the persistence of writes smaller than a stripe unit

while only using 1 bit per stripe unit.
In the rare case where a logical write checks the persistence bitmap and sees that the preceding logical writes have not completed, it is put onto a queue and periodically checked until the preceding logical writes have been marked as persisted. It is important to note that a logical write does not need to wait for preceding logical writes to be *reported to the user* as complete, but rather it only needs to wait until the preceding logical writes have been *persisted*. As a result, a single failed check of the persistence bitmap does not result in cascading failures, and only a small number of IO will experience longer latency due to the additional queue and check.

Multithreaded preprocessing. As we show in our evaluation, RAIZN must use multiple worker threads to achieve high throughput when serving small writes. However, even when using multiple worker threads it is necessary for RAIZN to submit zone writes to the kernel block layer in an order that respects the sequential write constraint. This is handled in a simple manner–RAIZN keeps track of the write pointer for each zone on each SSD, and the thread processing the IO waits until the device write pointer matches the address specified in the IO. Once the condition is met, the IO is submitted and the device write pointer is updated to reflect the latest IO submission.
When running in multithreaded mode, RAIZN first checks and/or updates the logical zone write pointer based on the address specified in the IO, then pushes each mapped IO onto a single shared queue. After the IO has been pushed onto the queue, one of a configurable number of worker threads is scheduled to run, and will pop the IO from the head of the queue and begin preprocessing it. The worker thread first preprocesses the request, generating some number of additional IOs (due to splitting, parity, etc.), which we term *sub-IOs*. Each of these sub-IOs are submitted by the worker thread as they are generated. If an IO is a read, it can be submitted to the physical drives immediately, but if it is a write, the worker thread waits until the write pointer matches the address specified in the IO request before acquiring a lock on the corresponding physical zone and submitting the IO.
Zone resets are ordered with respect to writes by keeping track of the logical zone generation hash and write pointer when the reset request is received, which we refer to as the reset generation and reset pointer respectively. Unlike writes, zone resets always specify the starting address of a zone rather than the current write pointer, so the reset pointer is different from the address specified in the zone reset IO. After the zone reset is accepted by the RAIZN volume, the corresponding logical zone is plugged and any further writes to that logical zone are rejected until the reset is complete. For each logical zone reset, there are N device reset sub-IOs generated in the preprocessing step, where N is the number of drives in the array. The worker thread handling the zone reset waits until each of the device write pointers matches the reset pointer before submitting the reset commands to the devices in the array, ensuring writes are never reordered with respect to resets. In practice, this detail is unimportant, as applications will typically reset zones long after the zone is completely filled. If the logical zone being reset has only one stripe written to it, a write-ahead log is appended to the general metadata zone of every device in the array before submitting the zone reset commands. This write-ahead log consists of the zone reset command, the logical zone address, and the generation hash of the zone. No writes can be accepted for a given logical zone until the reset is complete and the logical zone is un-plugged.

Per-zone volatile state. RAIZN keeps track of the write pointer and zone status for each logical zone in (volatile) memory, and must recalculate both when RAIZN is remounted. As we described earlier in this section, zone status for each logical zone is determined based on the write pointer and status reported by each device in the array. However, before zone status can be determined, RAIZN must first compute the write pointer for each logical zone, and by extension detect any consistency errors due to unexpected power loss.
When remounted, RAIZN checks the write pointer for each zone on every device in the array – we term these *device write pointers*. Given the set of device write pointers for all of the zones in a particular logical zone,

the first step is "stripe hole" detection. In simple terms, a stripe hole refers to when all or part of a stripe unit is missing despite at least one block of a stripe unit located at a higher LBA being present.
Stripe holes are detected by translating each device write pointer to an LBA, then checking whether a read from the lowest LBA to the highest LBA is serviceable. RAIZN can determine whether a read is serviceable by translating the read request into a set of PBA ranges, and check whether those PBA ranges fall between the start of the device zone and the device write pointer for each device in the array.
If it is possible to service the read normally, then there is no stripe hole. The only exception to this rule is if the highest readable LBA is within the first stripe of the zone, in which case RAIZN must also check to see if a zone reset was logged, as we describe in Section 4.1. If a zone reset is logged, this means a partial zone reset occurred before the host lost power, and the zone must be reset.
If it is possible to handle the read using a degraded read, meaning that the data is not present for a particular subset of PBAs in the stripe, but enough data and parity is available to recompute the data at each of those PBAs, the missing stripe unit(s) is rebuilt and there is no stripe hole. However, if it is impossible to handle the read, there is a stripe hole, which means either a partial zone reset or partial stripe write has occurred.
In the event of a partial zone reset, RAIZN can reset all of the device zones in the logical zone. If one or more partial stripe writes occurred, RAIZN sets the logical write pointer to the LBA immediately preceding the first hole, marks the zone as being remapped, and redirects future conflicting stripe unit writes to the metadata zone as necessary.

Stripe cache. Conventional RAID benefits from maintaining an in-memory stripe cache, caching data stripes so parity be quickly calculated when the stripe is updated. However, RAIZN does not need to maintain a stripe cache, due to two benefits of the ZNS interface: fully written stripes cannot be updated, and the open zone limit provides a hard cap on the number of partial stripes at any given time. Instead of maintaining a global stripe cache, RAIZN allocates a temporary buffer, called the *stripe buffer*, to hold the partial stripe contents until the entire stripe is written to the RAIZN logical device. Once the stripe buffer is filled up, the preprocessing thread can free it immediately after calculating the full stripe parity. Typically, write requests end up being handled in submission order, resulting in 0 or 1 stripe buffers per zone at any given time. However, due to asynchronous preprocessing of writes, there can potentially exist multiple stripe buffers for a single zone. In order to provide predictability of memory usage, RAIZN tracks the number of allocated stripe buffers, and temporarily pause processing of writes if that number exceeds a user-specified threshold.

5.1 Metadata

Stripe unit remappings. As we described in Section 4.1, partial stripe writes can result in stripe units written to a metadata zone instead of the corresponding data zone. We expect this case to be extremely rare, and as such our main goal is to minimize the performance impact on the steady state operation of RAIZN. It is technically possible for enough stripe units to get remapped that there is not enough space in the metadata zone to store all of them. To handle this, if the number of remapped stripe units passes a user-defined threshold, RAIZN uses the swap zone to perform a rebuild of each of the zones that contains a remapped stripe unit. All of the data is copied from the affected data zone onto a swap zone, the data zone is reset, and then the data is copied back with the remapped stripe unit placed in the correct location in the data zone. As an optimization, copying data to the swap zone can happen opportunistically when load is low, and the copy can be aborted by resetting the swap zone if load spikes.

Relocation map. Relocations in RAIZN are rare events, only occurring in the event of partial stripe writes, which in turn only occur when an unexpected power loss occurs in the middle of persisting a zone write to the RAIZN volume. As such, we do not persist the relocation map, instead opting to rebuild it when the volume is remounted by reading the metadata headers on relocated stripe units. This does result in some additional

overhead when remounting the volume, but a 1.1GiB metadata zone can be read in around 1 second, which is an acceptable overhead considering that the read can be done in parallel to other initialization tasks which take several seconds to complete. As with all metadata logs, if the generation counter stored in the metadata header of the relocated stripe unit does not match the most recent generation counter of the zone, the relocated stripe unit is ignored.

Zone reset logs. Zone reset logs are persisted by first plugging the associated logical zone, then submitting a 4k log entry to the general metadata zone on every device in the array. This 4k log entry contains the LBA of the logical zone to be reset, the generation counter of the zone, and a flag to indicate that the log entry is describing a zone reset. When the RAIZN volume is remounted, any zone reset logs with an up-to-date generation counter are read, and if a logical zone is not empty despite a valid zone reset log indicating that it should be empty, the logical zone is reset during system initialization.

Generation hashes. We store generation hashes succinctly by taking advantage of two observations: First, the generation hash must be unique to the zone for only a bounded amount of time–specifically, a new generation hash must not be present in any metadata log entry, otherwise an outdated log entry could incorrectly be processed as if it were valid. Second, zone resets are relatively uncommon, typically occurring after several gigabytes of data have been written.
We store one 32-bit counter per logical zone, which we term the *generation counter*. Every time the logical zone is reset, we increment its associated generation counter, and when the zone is opened again, its generation hash is given to be the value of the generation counter. All generation counters are stored in a contiguous block of memory, which for our SSDs adds up to approximately 7376 bytes. Each time a zone is reset, we persist the most current version of all generation counters into the general metadata zone. It is necessary to provide a global ordering of generation counter updates, so we include a 128 bit global counter in addition to the 7376 bytes of generation counters, which is incremented every time the generation counters are updated.
Due to the fact that zones are expected to be reset only after being fully written, overflowing the 32-bit counter is expected to be a very uncommon event, and as such we deal with it in a manner that is inefficient with the expectation that it will never happen. For example, writing and resetting a single logical zone using our SSDs (1077MiB zone capacity) 65535 times corresponds to over 4300 petabytes of data written *per drive*, an order of magnitude larger than the endurance rating of any commercially available SSD. In the event that a generation counter would overflow, RAIZN performs a "generation counter reset". First, RAIZN temporarily suspends processing of writes, and begins performing synchronous garbage collection on all metadata zones across all drives in the array–however, each valid metadata log entry is written twice to the swap zone–once with the header containing its original generation hash, and once with the header containing a generation hash of 0. Next, RAIZN resets all generation counters to 0, persisting this update to the general metadata zone. Finally, RAIZN once again performs synchronous garbage collection, removing any metadata log entries with nonzero generation hashes. Once the second garbage collection completes, RAIZN resumes normal operation. This process is necessary to ensure that RAIZN can remain in a consistent state even if power is lost at any point during the generation counter reset.

5.2 Discussion

There are some advanced features of the ZNS interface that are not currently part of the RAIZN implementation.

Logical block metadata. The NVMe standard supports attaching metadata to each logical block. The metadata descriptor is typically used for protection information (PI). However, there are opportunities for

optimizations in RAIZN if the underlying SSDs are formatted with logical block metadata enabled and the metadata is writable by RAIZN.
First, metadata logs would be reduced in size – we attach a header to each metadata log in RAIZN, typically containing a LBA, PBA, and generation hash. However, this information could be written to the logical block metadata area instead, reducing write amplification.
Second, zone appends could be handled without requiring serialized submission. The key problem with zone appends is that on-device reordering of appends can result in stripe units being persisted on disk in an arbitrary order. In the event of unexpected power loss before the zone append completion is reported to the host, it is impossible for RAIZN to know which stripe unit corresponds to which stripe after reboot. However, if each append is tagged with a stripe ID in the metadata descriptor, this would allow RAIZN to locate stripe units after unexpected shutdown.

Zone appends. The ZNS standard requires that for zone writes, only one write command can be submitted to the device at a time. This creates a problem, as we illustrated in section 6, where performance for many small writes to a single zone can be low on ZNS SSDs. The intended approach to overcoming this problem is to use zone appends–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 primitive and does not yet have a userspace interface. We look forward to seeing how future append-based workloads behave and how RAIZN can be designed to serve those workloads.

The Zone Random Write Area. ZRWA is an alternative to zone appends that defines a randomly-writable persistent buffer starting from the write pointer, allowing overwrites in a manner similar to conventional block interface devices. By allowing random writes, albeit in a limited scope, ZRWA provides the opportunity for many applications to achieve ZNS compatibility with minimal design modifications. Our devices did not have ZRWA enabled, so we did not integrate it into our system, but it could potentially be used to update parity in-place instead of using parity logs.

5.2.1 Consequences of large zones

One consequence of the design of RAIZN is that the capacity of each logical zone 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 [8] has shown that for certain workloads, such as cluster filesystems, it is advantageous to sacrifice per-object IO throughput in exchange for finer granularity control over placement with regards to the physical media. While the problem of large zones exists for all zoned devices, it is potentially exacerbated in RAIZN, and it is part of our ongoing work to determine solutions to this problem.

6 Evaluation

We evaluate the performance of RAIZN using synthetic microbenchmarks and application benchmarks. Our evaluation demonstrates that RAIZN is able to achieve comparable performance to software RAID running on conventional SSDs, and that RAIZN is able to scale to many drives without introducing significant bottlenecks or overheads. We characterize the differences in IO performance between RAIZN and running directly on the underlying ZNS SSDs.

	max write tput	max read tput	4k rand read
Conv	1074 MiB/s	3398 MiB/s	1306 MiB/s
ZNS	1052 MiB/s	3105 MiB/s	1304 MiB/s

Table 1: Single drive throughput

We perform all experiments on a Dell R7515 server with a 16-core AMD EPYC 7131P CPU, 128GB of DRAM, and a 512GB (conventional) SSD mounted at root. All experiments are run on Ubuntu 20.04, on a modified version of kernel 5.14.0. We slightly modify the kernel and mkfs.f2fs to remove hard-coded constraints and too-narrow data types that prevent the use of (logical) devices with zone sizes larger than 2GB, but these changes do not affect performance. All benchmarks are run using direct IO, ensuring the page cache does not affect any benchmark results.

For the RAIZN experiments, we use five production-grade Western Digital ZN540 2TB ZNS SSDs, and for the experiments on conventional RAID we use five equivalent conventional SSDs. Each zone on the ZN540 SSDs is 2GiB, and has a writable capacity of 1077MiB. We compare RAIZN to the Linux RAID 5 implementation, configuring the latter with a stripe cache size of 32768 pages, or 128MiB.

6.1 Raw device microbenchmarks

We begin by evaluating the basic read and write performance of the ZNS SSD and conventional SSD by using fio [4] to perform direct IO on the devices. All benchmarks in this section were run with fio 3.28 using libaio [6] with direct IO.

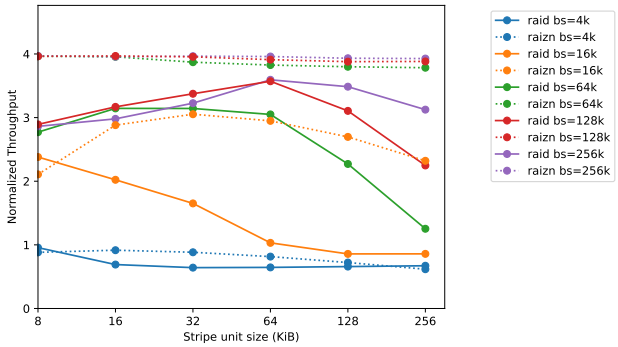
We start with a full drive sequential write, followed by a sequential read and a 4k random read of the entire drive. While a detailed performance analysis of the individual drives is outside of the scope of this project, we present the throughput and latency characteristics of each drive to provide a baseline to compare with RAIZN and RAID. Table 1 shows the throughput of each drive, with the ZNS drive performing slightly worse than the conventional drive in both 64k sequential writes and 64k sequential reads–this difference is likely due to a difference in firmware maturity between the two drives, as ZNS is a new technology.

Now that we have established a baseline level of performance per drive, we run several experiments to measure the performance of conventional RAID and RAIZN. We start by running the same write, sequential read, and random read benchmark on a 5-drive array, again configuring fio to perform direct IO on the RAID/RAIZN volume. We vary the stripe unit size from 8kb to 256kb, comparing the throughput and latency to determine what the ideal stripe size is for both RAID and RAIZN. Choosing the optimal stripe size is heavily dependent on workload, so we present graphs of the throughput and latency of a wide variety of workloads and select one that performs reasonably well on a majority of the benchmarks. For brevity, we only present graphs of a representative subset of the benchmarks we ran.

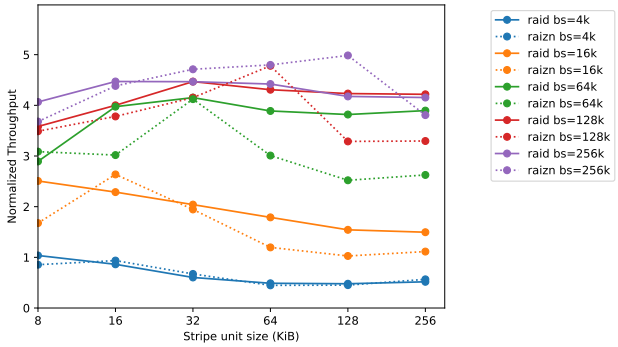
First, we look at throughput-oriented benchmarks run on a 5-drive array, examining the performance of sequential writes, sequential reads, and random reads using 8 client jobs each with a queue depth of 16 (Figure 4). All points on these graphs are normalized to the throughput of a single drive–a normalized throughput of 4 indicates that the throughput is equivalent to 4 times the throughput of a single drive. This normalization is done to account for the discrepancy between throughput between the ZNS and conventional drives.

Next, we examine the median and tail latency performance of RAIZN and RAID in a collection of latency-oriented benchmarks, configured with 8 client jobs each with a queue depth of 1 (Figures 5 and 6).

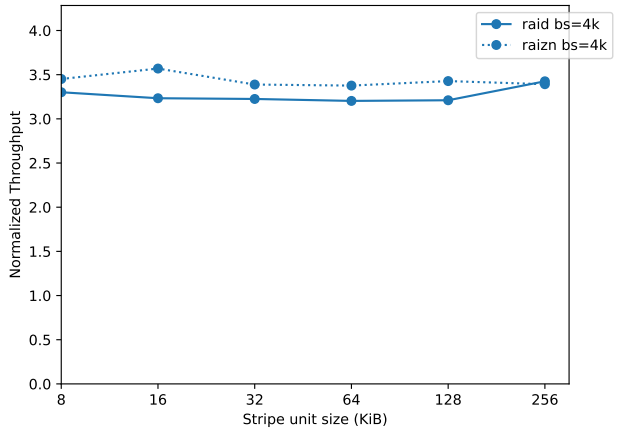
The effects of garbage collection. In order to illustrate the effects of garbage collection on conventional SSDs, we run a full drive overwrite benchmark on a 5-drive array. This benchmark is composed of two workloads–first, we use 5 threads to sequentially write the entire capacity of the array, with each thread



(a) Sequential write

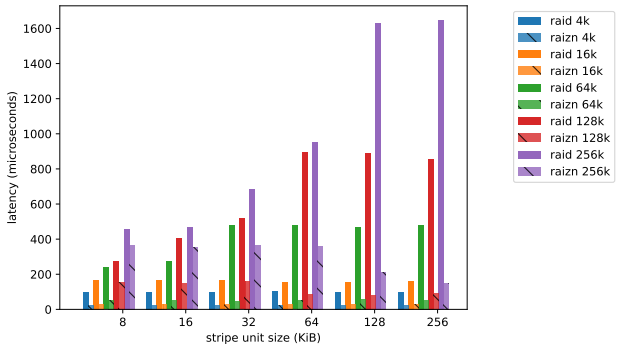


(b) Sequential read

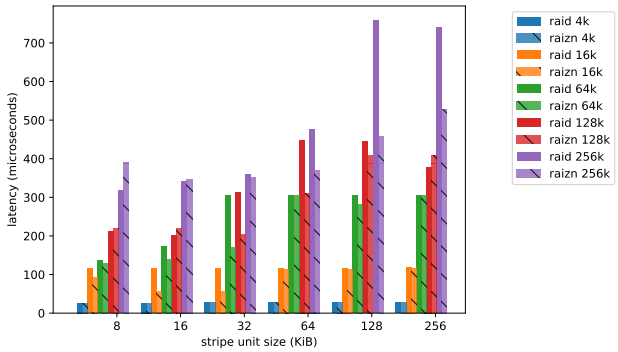


(c) Random read

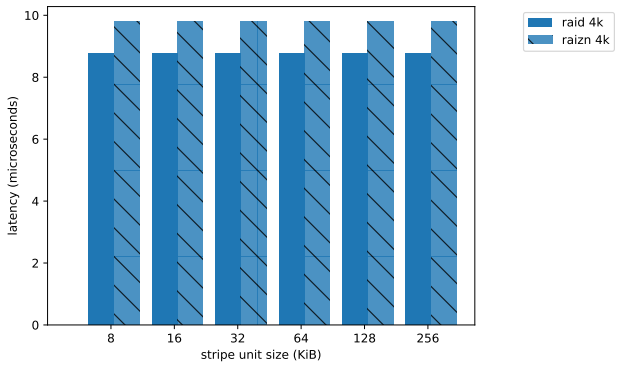
Figure 4: RAIZN achieves roughly comparable throughput to RAID



(a) For sequential writes, RAIZN tends to achieve lower median latency

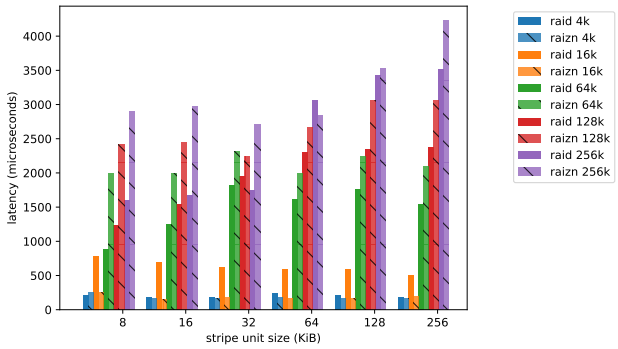


(b) RAIZN achieves comparable or better median latency for sequential reads

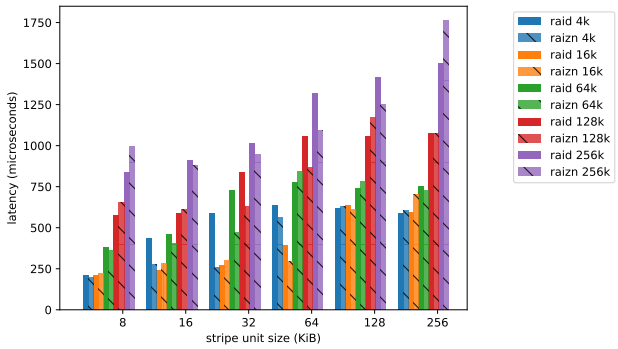


(c) 4k Random reads have slightly worse median latency for RAIZN

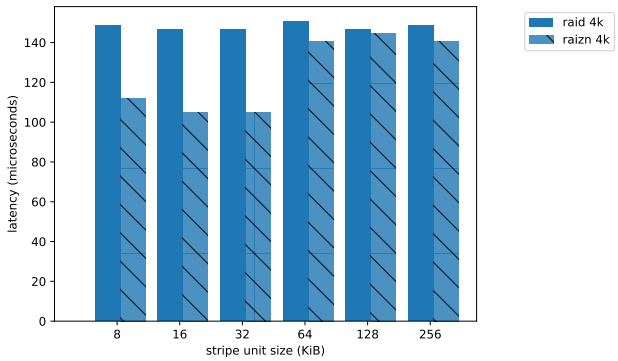
Figure 5: Median latency



(a) RAIZN has somewhat worse tail latency for sequential writes.



(b) RAIZN achieves similar tail latency for sequential reads.



(c) RAIZN has either comparable or slightly better tail latency for 4k Random reads

Figure 6: 99.99%ile tail latency

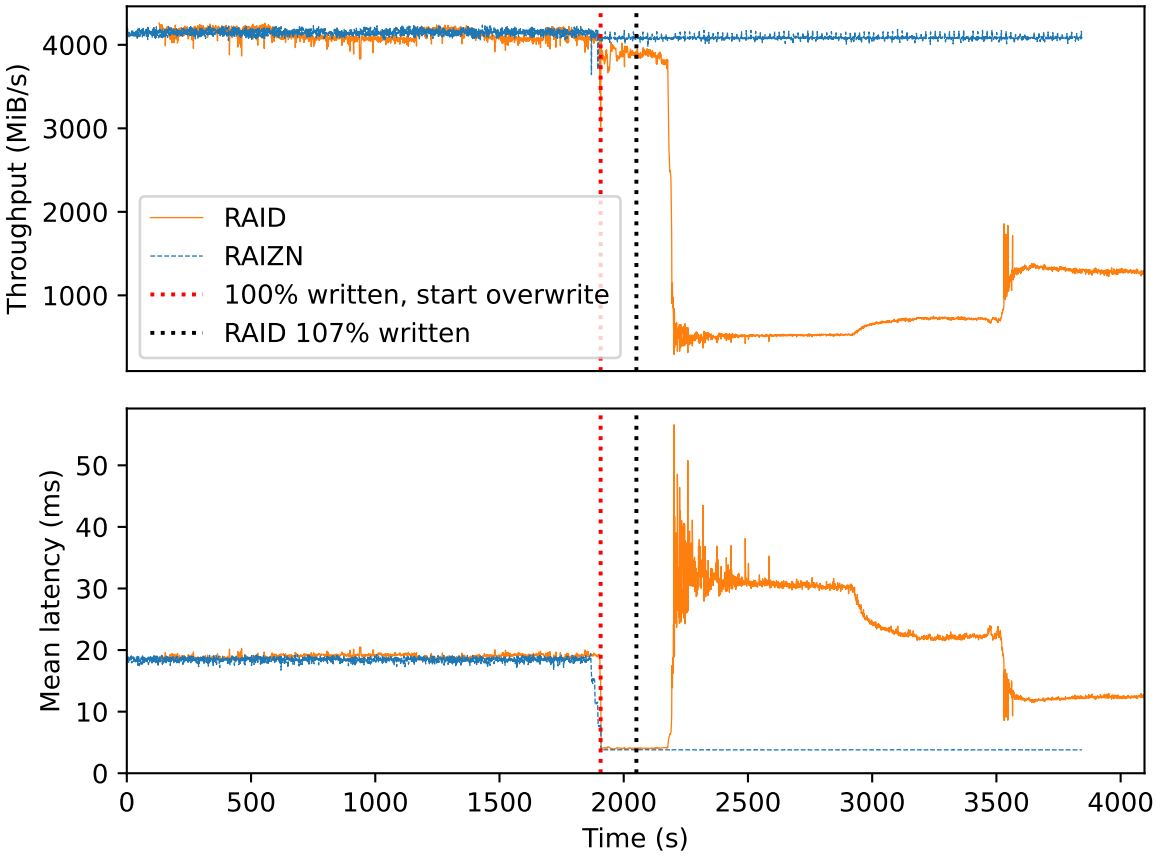
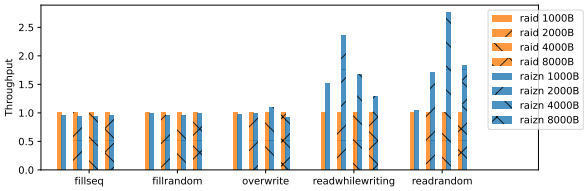
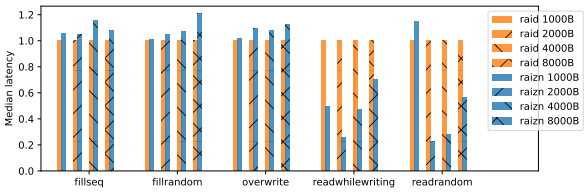


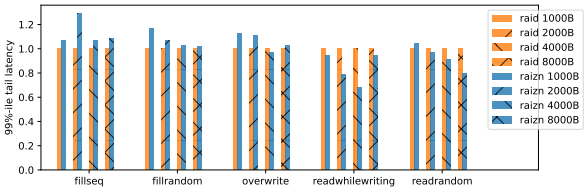
Figure 7: Conventional RAID suffers when the SSDs run out of overprovisioned spare blocks and start performing garbage collection. RAIZN is able to maintain high throughput and low latency because ZNS SSDs do not perform garbage collection



(a) RocksDB throughput, normalized



(b) RocksDB median latency, normalized



(c) RocksDB 99%ile tail latency, normalized

Figure 8: RAIZN is able to achieve comparable performance to RAID on fillseq, fillrandom, and overwrite, and achieves superior throughput and latency on readwhilewriting and readrandom.

writing 20% of the address space (the first thread writes 0% to 20%, the second thread writes 20% to 40% etc.). Then, we run a sequential write workload, with one thread writing the entire address space of the array, overwriting all of the data written in the previous workload. During these two workloads, we sampled the write throughput every second, and the results are graphed in figure 7 as a timeseries. The red vertical dashed line represents the point in time where the second workload of the benchmark started, and for comparison the results of the RAID are shifted to the right, and we stop the benchmark after approximately 4096 seconds. The RAID array experiences a sharp drop in throughput after the conventional SSDs run out of 7% spare erase blocks from where it is marked with a black vertical line and begin performing garbage collection, whereas the RAIZN array maintains constant throughput through the entire benchmark. Average latency is also high performing garbage collection in the RAID array, but consistently low latency in RAIZN array.

6.2 RocksDB

We run a set of benchmarks of RocksDB [11] to show the performance of RAIZN compared with conventional RAID on real workloads. For these experiments, we use the same 5-drive array with 64kb stripe units for both RAIZN and RAID, and format each logical device as an F2FS filesystem. F2FS supports both ZNS and conventional SSDs, though certain optimizations such as threaded logging are disabled for ZNS SSDs [13]. Other than specifying ZNS support for RAIZN, we used all default options for F2FS. We use the *fillseq*, *fillrandom*, *overwrite*, *readwhilewriting*, and *readrandom* workloads defined by `db_bench` [12] in our evaluation, with value sizes of 1000, 2000, 4000, and 8000 bytes. Additionally, we specify `-use_direct_io_for_flush_and_compaction` and `-use_direct_reads` to ensure the page cache is bypassed, preventing the large amount of DRAM on our servers from affecting the benchmark results.

This experiment demonstrates that RAIZN can achieve similar performance to industry standard RAID software on application benchmarks.

7 Related work

The ZNS interface is one of many systems whose aim is to avoid the performance and cost penalties associated with accommodating the block interface on flash-based storage. Other approaches include multi-stream SSDs, open-channel SSDs, key-value SSDs, and application-managed flash.

Multi-Stream SSDs organize erase blocks into *streams*, and allow the host to group writes by expected lifetime, directing each of these groups into a separate stream [16]. Multi-Stream SSDs 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 the SSD will have to garbage collect old pages to accommodate new writes. Due to the requirement for on-device garbage collection, multi-stream SSDs must overprovision flash and thus do not provide any cost-per-byte savings over conventional block interface SSDs.

Open-Channel SSDs (OCSSDs) divide underlying flash into fixed-size chunks, called bounded streams, that correspond to the size of an erase block [10]. The host can then explicitly write or erase these chunks, enabling lifetime-based data placement and host-directed garbage collection. By bounding the size of each stream, OCSSDs do not require on-device garbage collection, obviating the need for overprovisioned flash blocks. While bounded streams may appear analogous to zones at first glance, a key difference is that media-level management tasks such as wear-leveling and media reliability must be handled by the host in OCSSDs, necessitating specialized host software for every different variety of SSD hardware. In contrast, ZNS SSDs handle media management in firmware, and the host is only responsible for data placement and garbage collection.

Application-Managed Flash allows users to directly access segments of flash composed of a fixed set of erase blocks [18]. Each segment of application-managed flash is written sequentially and reset as a single unit, behaving somewhat similar to a zone in ZNS. However, unlike zones, segment to erase block 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. There has been existing work on RAID-like reliability for KVSSDs[25], but achieving space-efficient parity coding for small objects over an array of KVSSDs has proven difficult [20].

7.1 Shingled Magnetic Recording

SMR drives are similar in many ways to ZNS SSDs, with both benefiting from a zoned interface due to similarities in the underlying physical media. The overlapping tracks in SMR drives necessitates that large groups of tracks (i.e. zones) must be written sequentially and reset together before rewriting. In this section, we describe some of the existing systems built to store data on SMR drives, and how they compare to RAIZN.

SMORE is an object store designed to provide cold storage on an array of SMR disks [19]. Similar to RAIZN, SMORE stores important metadata, such as the superblock, in log structured format on a set of dedicated zones. In addition, SMORE stripes data across a set of zones spanning multiple physical devices, similar to RAID. However, unlike SMORE which is an object store, RAIZN exports a logical volume that behaves like a zoned block device, and thus addresses various challenges that are brought about as a result of conforming the external interface to the ZNS specification. In addition, SMORE is designed to store cold objects ranging from a few megabytes to multiple gigabytes in size, whereas RAIZN is designed to work on a wider range of workloads, such as smaller granularity writes and read-heavy workloads.

HiSMRfs is a POSIX filesystem designed to achieve high performance on SMR drives [14]. To achieve high performance, HiSMRfs implements a log structured filesystem for data while storing metadata in high performance random write storage, such as an SSD. HiSMRfs includes support for RAID, allowing data to be parity coded across multiple drives in an array. Superficially, the design of HiSMRfs is similar to RAIZN, as it implements RAID-like striping of data across multiple append-only zones. However, RAIZN is intended to behave like a block device, and solves the problem of efficiently persisting and managing metadata on zoned storage devices, whereas HiSMRfs is a filesystem and uses a separate metadata device to achieve high performance. Being a filesystem, HiSMRfs exists at a different level than RAIZN, and could potentially be run on a RAIZN volume.

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 operation. RAIZN is able to achieve latency characteristics similar to the Linux RAID 5 implementation for conventional drives, even when the conventional drives do not incur garbage collection overheads, and throughput within 2% of using the underlying SSDs directly. More importantly, RAIZN preserves the benefits of ZNS, allowing ZNS-specialized host software to maintain steady performance for an extended read/write workload that caused RAID-for-conventional-drives to lose 90% of its performance after the conventional SSDs started garbage collection.

References

[1] https://zonedstorage.io/.

[2] https://lkml.org/lkml/2020/10/15/844.

[3] dm-zap: Host-side zoned host translation mapper. https://github.com/westerndigitalcorporation/dm-zap.

[4] fio - flexible i/o tester rev. 3.27. https://fio.readthedocs.io/en/latest/fio_doc.html.

[5] A guide to mdadm. https://raid.wiki.kernel.org/index.php/A_guide_to_mdadm.

[6] libaio. https://pagure.io/libaio.

[7] 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.

[8] Smartftl architecture for ssds. https://www.youtube.com/watch?v=3O3zDrpt3uM.

[9] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, DL Moal, G Ganger, and George Amvrosiadis. Zns: Avoiding the block interface tax for flash-based ssds. In *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21)*, 2021.

[10] Matias Bjørling, Javier Gonzalez, and Philippe Bonnet. Lightnvm: The linux open-channel {SSD} subsystem. In *15th {USENIX} Conference on File and Storage Technologies ({FAST} 17)*, pages 359–374, 2017.

[11] Facebook. RocksDB. http://rocksdb.org/, 2015.

[12] Facebook. Performance Benchmarks. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools, 2021.

[13] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Joo-Young Hwang. Zns+: Advanced zoned namespace interface for supporting in-storage zone compaction. In *15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21)*, pages 147–162, 2021.

[14] Chao Jin, Wei-Ya Xi, Zhi-Yong Ching, Feng Huo, and Chun-Teck Lim. Hismrfs: A high performance file system for shingled storage array. In *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6. IEEE, 2014.

[15] Hai Jin, Xinrong Zhou, Dan Feng, and Jiangling Zhang. 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)*, pages 396–400, 1998.

[16] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. The multi-streamed solid-state drive. In *6th {USENIX} Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.

[17] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2fs: A new file system for flash storage. In *13th {USENIX} Conference on File and Storage Technologies ({FAST} 15)*, pages 273–286, 2015.

[18] Sungjin Lee, Ming Liu, Sangwoo Jun, Shuotao Xu, Jihong Kim, et al. Application-managed flash. In *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*, pages 339–353, 2016.

[19] Peter Macko, Xiongzi Ge, J Kelley, D Slik, et al. Smore: A cold data object store for smr drives. In *Proc. 34th Symp. Mass Storage Syst. Technol.(MSST)*, 2017.

[20] Umesh Maheshwari. 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)*, 2020.

[21] Wojciech Malikowski. Spdk open-channel ssd ftl, 2018. https://spdk.io/doc/ftl.html.

[22] Damien Le Moal. dm-zoned: Zoned block device device mapper, 2017. https://lwn.net/Articles/714387/.

[23] David A Patterson, Garth Gibson, and Randy H Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, pages 109–116, 1988.

[24] Rekha Pitchumani and Yang-suk Kee. Hybrid data reliability for emerging key-value storage devices. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, pages 309–322, 2020.

[25] Rekha Pitchumani and Yang-Suk Kee. Hybrid data reliability for emerging key-value storage devices. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 309–322, Santa Clara, CA, February 2020. USENIX Association.

[26] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[27] Marta Rybczyńska. Btrfs on zoned block devices. https://lwn.net/Articles/853308/.