Design principles for replicated storage systems built on emerging storage technologies.

Thomas Kim

CMU-CS-23-109

March 2023

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
David G. Andersen, Chair
Michael Kaminsky
Gregory R. Ganger
Matias Bjørling (Western Digital Corporation)

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This material is based upon work supported by: the National Science Foundation (NSF) under award numbers CNS1956271, CNS1700521, CNS1314721, and CCF1535821; Intel Corporation under award number A018540296213611; Google LLC under award number 5003506; Western Digital Tech under award number OSP00001830; and by gifts from the NetApp University Research Fund, a corporate advised fund of Silicon Valley Community Foundation, and Meta Platforms, Inc.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Storage systems, replicated storage, distributed storage, persistent memory, non-volatile main memory, zoned namespaces, ZNS, SSD, flash memory

For my parents

Abstract

With the slowing down of Moore's law, persistent storage hardware has continued to scale at the cost of exposing hardware-level write idiosyncrasies to the software. Thus, a key challenge for systems developers is to reason about and design around these idiosyncrasies to create replicated storage systems that can effectively leverage these new technologies. Two examples of such new and emerging persistent storage technologies are Intel Optane non-volatile main memory and Zoned Namespace (ZNS) solid-state drives. Intel Optane provides persistent byte-addressable storage with throughput and latency rivaling DRAM, but per-DIMM write throughput is significantly lower than read throughput–this imbalance presents challenges in providing high availability in replicated storage systems, due to the severely limited ability to bulk-ingest data. ZNS is a new interface for NVMe-based SSDs that eliminates the flash translation layer, thus preventing garbage collection-related performance degradation and reducing the need for overprovisioned flash hardware. A consequence of these benefits is the loss of overwrite semantics for blocks in a ZNS device, thus necessitating flash-based replicated storage systems to be redesigned for ZNS compatibility.

Based on our experiences and setbacks when designing, implementing, and evaluating systems based on Optane and ZNS, we propose three guidelines to assist developers in designing storage systems on new and emerging persistent storage technologies: (1) systems, even those expected to serve read-heavy workloads, should prioritize optimizing write performance, (2) set and fulfill performance, durability, and fault tolerance guarantees, but do not exceed them as that may result in excessive write overheads, and (3) systems can overcome limitations of write-constrained persistent hardware by optimizing data placement and internal data flows based on assumptions about temporal and spatial locality of the expected client workload.

The first system we present is CANDStore, a highly-available, cost-effective, replicated key-value store that uses Intel Optane for primary storage, and solves the challenge of bottlenecked data ingestion during primary failure recovery through a novel *online* workload-guided recovery protocol. The second system we present is RAIZN, which is a system that provides RAID-like striping and redundancy for arrays of ZNS SSDs, and solves the various challenges that arise as a result of the lack of overwrite semantics in ZNS. We describe how the above guidelines arose from the setbacks and successes during the development of the above two systems, then apply these guidelines to extend the functionality of RAIZN to create RAIZN+. The final part of this thesis details exactly how we applied these guidelines to achieve near-zero write amplification when serving RocksDB workloads in RAIZN+.

Acknowledgments

This thesis would not have existed without the support and guidance of countless people, some of whom I will forget to include in this document. As such, I'd like to start by extending my gratitude and apologies to everyone who isn't included below.

I would like to thank Dave Andersen, my official advisor, and Michael Kaminsky, my unofficial co-advisor, for guiding me through my doctorate and teaching me how to conduct research. Dave is always able to produce countless insights based on my vague ideas, and Michael helps hammer those ideas into something more concrete.

I thank Greg Ganger for always breaking down my proposals to their core components, then pointing out weaknesses. While I may never be able to replicate Dave's off-the-cuff sparks of genius, Greg's way of breaking down complex problems will stay with me forever. Though not on my committee, George Amvrosiadis has been instrumental in my PhD journey, giving me the opportunity to work with the ZNS SSDs and always being available when I ran into challenges.

Of course the last few years of my PhD would not have been possible without Matias Bjørling, who not only provided the hardware I ran experiments on, but also helped me sort out issues with the ZNS SSDs. Matias has saved me months of debugging by taking the time to answer my nonstop questions on Matrix chat. In addition, I would like to thank Damien Le Moal, Hans Holmberg, Dennis Maisenbacher, Dmitry Fomichev, Shin'ichiro Kawasaki, Andreas Hindborg, and all the other folks at Western Digital for teaching me how to use the ZNS devices and related software.

Thanks Michael Kozuch, for your help on the ISTC-VCS camera deployment, and for your help regarding the Optane NVMM servers. Jason Boles, without your help I would have wasted weeks debugging deployment issues–your technical expertise has saved me on countless occasions.

The FAWN group (including unofficial), Hyeontaek, Dong, Huanchen, Anuj, Conglong, Sol, Chris, Giulio, Daniel, and Angela, have helped me on deadlines, listened to my practice talks, and provided personal and professional advice when I most needed it. Chris, thank you for letting me sleep on your air mattress over my last few visits to Pittsburgh, and supporting me as a friend over the last half decade. Angy Malloy, I'm sorry for all the weird reimbursements I made you handle for me; though I lost the reply-all napkin, it'll bring me traumatic memories for the rest of my life.

The PDL support staff have all been amazing to me, and I'm sure there are some folks who did their job so impeccably that I never knew them. Thanks to Joan Digney for always putting up with my last second PDL poster changes, and for looking over my thesis with a professional eye. Karen Lindenfelser, thank you for always being a beacon of positivity and making all of us PDL students feel individually appreciated.

Finally, thank you to my family, who have supported me through my long journey as a grad student, including my dog, who kept me sane through the Covid-19 pandemic.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Persistent storage has been, and continues to be, a fundamental part of every computer system; our insatiable appetite to store, retrieve, and modify larger and larger amounts of data is one of the primary challenges faced in modern computing [69]. Storage is composed of two components–hardware and software–both of which come in many varieties to suit different tasks. Low performance mass data storage such as magnetic tape or spinning disk provides an economically viable solution to storing large amounts of infrequently-accessed data, while high performance "main-memory class" storage based on battery-backed DRAM is preferable when providing persistent storage for latency-sensitive or high-throughput datacenter applications. Running on top of this hardware, storage systems are the key software component that enable the storage and retrieval of data from hardware.

Storage systems range from large-scale distributed systems to single-node local storage, but share the common goals of durability, availability, and performance. Durability refers to the capability of the storage system to store data in a manner that is resilient to adverse events, such as hardware errors or power loss [52]. Availability describes the ability of a storage system to continuously serve workloads through faults such as network errors, hardware failures, and catastrophic events [39]. Performance, in the context of this document, is the ability of a storage system to use the underlying hardware in a manner that provides the maximum efficiency per unit hardware.

A common technique in achieving these goals of durability, availability, and performance is replication [103], where the storage system will store multiple copies of data to provide resilience against faults and handle client requests from multiple replicas in parallel. In this thesis, we explore the design and implementation of replicated storage systems built on two recent types of emergent storage hardware–Intel Optane non-volatile main memory and Zoned Namespace SSDs–and lay out a set of design guidelines based on the lessons we learned through the design, implementation, and evaluation of these systems.

1.1 Storage after Moore's Law

For decades, the industry's relentless pursuit of Moore's law has enabled cheaper, faster, and larger hardware storage devices [72]. This scaling was important, as demand for data storage continues to increase exponentially. However, in the post-Moore's law era, it has become increasingly difficult to mask the performance idiosyncrasies of computer hardware in general, and nowhere is this

more apparent than in persistent storage. Growth in storage density and performance has slowed considerably [98], but our appetite as a species for larger capacities of faster persistent data storage has only continued to accelerate.

New persistent memory technologies have emerged to potentially alleviate the stagnating advancement in existing persistent storage hardware, sidestepping Moore's law to improve cost or performance through different underlying physical media or storage interfaces. As a direct result, programmers are increasingly exposed to the unique characteristics of the physical media itself, creating a need not only for new software systems that take full advantage of these new storage technologies, but also a set of design principles to guide systems designers as we build storage systems on top of increasingly exotic hardware.

Two such examples of new and emerging storage technologies, which we explore in-depth in this thesis, are Intel Optane non-volatile main memory (NVMM) and Zoned Namespace (ZNS) solid state drives (SSDs). Intel Optane NVMM [55] is intended to provide IO performance rivaling DRAM with the advantage of lower cost per byte, higher density per DIMM, and persistence without reliance on capacitor/battery-backing. However, Optane offers only a fraction of the write throughput of DRAM, with the difference so stark that DRAM-based storage systems would not operate correctly if DRAM were to be substituted for NVMM. To take advantage of the potential cost savings of using Intel Optane in place of battery-backed DRAM, it is necessary to design new storage systems to accommodate the performance idiosyncrasies of NVMM.

Zoned Namespace [27] is a new interface designed to replace the traditional block interface that is standard on flash-based SSDs, and is currently a ratified part of the NVMe standard [37]. The traditional block interface, which was designed with hard disk drives (HDDs) in mind, provides sector-granularity random overwrite; flash memory, which does not support overwriting of data in-place or fine-grained erasure, is incompatible with the traditional block interface. Support for the traditional block interface in SSDs is provided through a firmware component called the flash translation layer (FTL) which lays out data on flash in log-structured format and performs garbage collection to reclaim capacity–however, this comes at the cost of decreased endurance and unpredictable performance from the overhead of garbage collection, as well as increased monetary expense from the compute, DRAM, and overprovisioned flash resources to facilitate the garbage collector. As a result, recent trends in flash-based block storage devices have seen a push towards adopting new interfaces that resolve these problems, such as ZNS. The ZNS interface solves the problems of reduced endurance, performance, and increased expense caused by the on-device garbage collector on traditional SSDs, but comes at the cost of disallowing overwrites. The lack of overwrites makes ZNS incompatible with existing software, and necessitates that storage systems be redesigned for ZNS.

On both ends of the spectrum–high performance byte-addressed persistent storage and high capacity block-addressed persistent storage–it is apparent that the performance and interface characteristics of writing data present the greatest challenges in designing systems software to take advantage of persistent memory hardware. Thus, a key problem in designing future storage systems is providing durability, redundancy, and availability while working around these limitations on writes. Not only is storage hardware starting to expose performance quirks to the software level, advancements like ZNS introduce the need for software to be aware of spatial locality and hardware limitations on erase granularity.

In our work, we aim to explore how these new persistent storage technologies can be integrated into datacenters in a practical manner while taking full advantage of the new hardware

features to provide *highly available* and *reliable* storage. For large scale datacenter deployments, hardware failures are common [38], thus requiring storage systems to gracefully handle hardware failure without losing or interrupting access to stored data. To date, we have explored this thesis through the design, implementation, and evaluation of two systems, *CANDStore* [61] (Section 3) and *RAIZN* (Section 4).

In this document, we present three design principles relevant to designing storage systems to make efficient use of write-limited storage hardware based on the various successes and setbacks we experienced when designing, implementing, and evaluating two systems built on Intel Optane and ZNS SSDs respectively. We observe that the severe limitations on writes, whether it be performance or interface-related, necessitates an emphasis on improving write efficiency regardless of the expected workload; as such, we lay out concrete suggestions regarding the optimization of data flow, fault tolerance, and consistency with regards to writes.

First, we observe that writes and bulk erases on flash memory are so costly that even those systems expected to serve read-heavy workloads frequently benefit from being optimized for write performance first. We argue that ensuring the system is designed in a manner that efficiently handles writes can often take precedence over minimizing read overheads or optimizing the read path.

Second, when working with write-constrained hardware, it is crucial to set explicit guarantees about durability, performance, fault tolerance, and consistency, then design the system to fulfill these guarantees without exceeding them; over-delivering and providing performance in excess of these guarantees will often result in additional write overhead and by extension worse performance of the system as a whole. The headroom created by not exceeding guarantees should be applied to improve the aspects of the system that are most significantly impacted by the write constraints imposed by the storage hardware.

Third, the degree of limitations on writes for new and emerging persistent storage hardware necessitates that storage systems be specialized around particular types of workloads. Systems designers should characterize these workloads and quantify workload constraints, then optimize internal data placement and data flows to take advantage of the workload characteristics.

To summarize, this thesis seeks to provide evidence to support the following thesis statement:

Replicated storage systems should follow the following three design principles to effectively use new persistent storage hardware: (1) prioritize optimization of write performance, even for systems expected to serve read-heavy workloads, (2) set and fulfill performance, durability, and fault tolerance guarantees, but do not exceed them as that may result in excessive write overheads, and (3) systems can overcome limitations of write-constrained persistent hardware by optimizing data placement and internal data flows based on assumptions about temporal and spatial locality of the expected client workload.

1.2 Case studies

We substantiate our claim through the design, implementation, and evaluation of two novel replicated storage systems: CANDStore and RAIZN, which leverage Intel Optane NVMM and ZNS SSDs respectively. In this section, we briefly outline how the final design of these two systems was driven by our design principles; Chapters 3 and 4 go into more detail about the various setbacks and mistakes that helped define and shape these principles.

1.2.1 CANDStore

CANDStore is a low cost, strongly consistent, distributed, replicated persistent key-value store built on top of Intel Optane DIMMs and NVMe SSDs. Data in CANDStore is replicated onto a fast Optane-backed *primary* replica and lower performance but cheaper NVMe-backed *backup* replicas which provide fault tolerance. Intel Optane persistent memory provides byte-addressable persistent storage that achieves DRAM-like latency and read throughput, but requires that applications be designed to tolerate significantly reduced write bandwidth. This reduced write bandwidth necessitates new approaches to crash recovery, which for traditional DRAM-class storage systems such as RAMCloud [81] has relied on the high write throughput of DRAM. In traditional *offline* primary recovery protocols, a replaced primary node is fully populated with the data previously stored on the failed primary node before resuming operation–however, with the $7\times$ reduced write throughput and $4\times$ higher density of Optane NVMM compared to DDR4 DRAM, this is not a feasible approach in Optane NVMM-based storage systems. Under these conditions, the time to recover (TTR) would be increased by $28\times$, resulting in unacceptably long periods of unavailability following primary node failure. To work around this, CANDStore uses a novel fast *online* crash recovery protocol coupled with a heterogeneous replica group design to enable data replication, high availability, and DRAM-class latency at significantly reduced cost compared to existing systems which store primary replicas in DRAM.

The infeasibility of offline recovery required us to use online recovery of data onto replacement primary nodes–the primary node serves client requests while also ingesting and persisting data from replicas. This in turn required us to carefully consider problems that are not an issue for offline recovery protocols, such as workload characteristics and the definition of recovery/availability; an online recovery protocol does not involve any significant downtime but in exchange experiences degraded performance during the recovery phase, making it less clear whether the system is available at a given point in time during recovery. We observed that our storage system is still *available* during periods of degraded performance as long as it is able to meet service level objectives (SLOs); we leveraged this observation, along with knowledge that datacenter workloads are typically skewed, to implement online recovery that can recover after primary replica failure 4.5-10.5x faster than existing offline recovery approaches. Our fast crash recovery protocol makes it possible to implement strongly consistent, distributed, replicated key value storage using non-volatile main memory (NVMM) in conjunction with fast NVMe SSDs, achieving DRAM-class latency at lower cost and higher availability than existing DRAM-based key-value storage systems.

CANDStore demonstrates our three design principles in the following ways:

1. Backup replicas buffer key-value updates on NVMM and batch writes to maximize SSD write throughput.
2. CANDStore rate-limits data ingestion during recovery based only on the client *write* workload.
3. During steady-state operation, only the minimum amount of data and metadata necessary to maintain consistency is written to witness nodes.
4. Logical timestamps in CANDStore are stored in log-structured format on SSD and cached on volatile DRAM rather than a persistent hashmap.
5. During recovery after primary failure, CANDStore takes advantage of performance leeway

between SLOs and available resources to operate with degraded performance while still meeting SLOs.
6. CANDStore expedites primary replica reconstruction by identifying and taking advantage of client workload skew, enabling a quick return to in-SLO operation.

1.2.2 RAIZN

RAIZN is a RAID-like system that improves availability for an array of ZNS SSDs through striping and parity coding data across the devices in the array. RAIZN solves many challenges arising from the lack of overwrite semantics in ZNS to achieve similar steady-state performance to conventional software RAID systems, such as `mdraid`, while also achieving superior performance stability through the elimination of on-device FTL garbage collection.

Implemented as a ZNS logical volume using the device mapper [5] (dm) framework, RAIZN provides high availability for arrays of ZNS SSDs, and achieves comparable steady-state tail latency and throughput to standard software RAID (`mdraid`). For certain workloads, conventional RAID running on SSD arrays can lose over 80% throughput due to on-device garbage collection– however, we show that RAIZN is able to handle these workloads without experiencing any fluctuations in performance, as ZNS SSDs do not perform on-device garbage collection.

The lack of overwrite semantics along with several nuances of the ZNS interface resulted in many challenges when designing RAIZN, as the logical device is required to provide the same consistency guarantees as a single device while asynchronously writing data across multiple physical devices.

RAIZN demonstrates our three design principles in the following ways (detailed in Section 4.6:

1. Metadata writes on the write path were minimized in frequency and size through RAIZN's use of a static arithmetic mapping between logical and physical block addresses.
2. Low level optimizations such as minimizing partial parity size, device cache flushes, and inlining metadata into the header were necessary to bring RAIZN's performance up to par with `mdraid`.
3. Metadata such as persistence bitmaps, physical zone descriptors, and logical zone descriptors are not persisted as they can be reconstructed from observing device status.
4. Remapped stripe units and partial parity are only replicated onto the minimum set of devices necessary to meet the fault tolerance guarantees.
5. ZNS write pointers are leveraged during recovery to copy only valid LBAs to the replacement device.

1.2.3 RAIZN+

RAIZN+ is an extension of RAIZN that enables support for zone appends and configurable logical zone size, two features that are detailed in Section 5.

RAIZN+ was built based on our design principles, and draws on our experience gained from the various challenges and setbacks faced when building and evaluating CANDStore and RAIZN.

1. Zone Appends are handled by optimistically writing parity assuming the common case where data is placed onto the device address space in a way that reflects submission order; this

minimizes the amount of persisted metadata in the common case while introducing some additional overhead in the unlikely edge case where data is scrambled.
2. A metadata entry called a *reorder record* is persisted to keep track of which LBA ranges have been written without reorderings–to reduce the size and quantity of reorder records, RAIZN+ merges these records and piggybacks the contents onto other metadata headers.
3. RAIZN+ enforces that the data corresponding to a given logical zone on a physical zone be contiguous in physical address space; this ensures minimal write overhead as only one mapping per logical zone per device needs to be tracked.
4. Flash-friendly ZNS-compatible applications typically create streams to facilitate garbage collection; RAIZN+ leverages this workload characteristic to facilitate garbage collection with low write amplification despite the impossibility of providing such a mechanism in the general case.

1.3 Summary of Contributions

Our first contribution is the design and implementation of a DRAM-class high performance key-value store on top of write bandwidth-limited Intel Optane NVMM. In particular, we demonstrate how limited write throughput and increased density on the underlying persistent memory hardware can result in challenges when recovering from primary node failure, and how it is necessary to design recovery and steady-state protocols to ensure fast recovery and high availability in such systems. We design, implement, and evaluate CANDStore, which is a strongly consistent, highly available replicated persistent key-value store built on top of Intel Optane and NVMe SSD.

Our second contribution is the exploration of how to design systems to deal with interface idiosyncrasies of flash memory, in particular the exploration of spatial locality as a software-controllable factor to influence and reduce write amplification in ZNS devices. Unlike traditional flash devices which hide this spatial locality and data placement onto physical media from the software layer, ZNS SSDs expose this detail and give the systems designer the opportunity to influence the overheads associated with write amplification; however, ZNS simultaneously requires the programmer to adapt to the idiosyncrasies of flash memory, in particular the lack of overwrite semantics and coarse granularity of data erasure. We design, implement, and evaluate RAIZN, which is a RAID-like system to stripe and replicate data across an array of ZNS SSDs.

Our third contribution is an explicit set of design principles which should be applied when building storage systems on top of new and emerging persistent storage technology. We identify the trend of write-related idiosyncrasies in new storage hardware, and build upon the knowledge gained through the design, implementation, and evaluation of the above two systems to create a general set of design principles:

1. Because of the unusually high costs of writes and bulk erases on flash, even those systems expected to be read-heavy frequently benefit from being optimized for write performance first.
2. When working with write-constrained hardware, it is important to set explicit guarantees about durability, performance, fault tolerance, and consistency. Systems should then fulfill, but not exceed, these guarantees to avoid imposing additional write overheads.

3. System designers should identify temporal and spatial locality characteristics of the expected client workload, then design internal data flows and data placement to make optimal use of writes given these workload constraints; this, in many cases, enables systems to overcome the fundamental write-related limitations of persistent hardware and provide greater consistency, durability, or availability than would otherwise be possible.

Chapter 2

Background

This section describes various technologies relevant to our work on CANDStore and RAIZN, as well as several concepts that are necessary to understand our work.

2.1 Concepts

Service level objectives (SLOs) are concrete descriptions of the performance guarantees of a system or service, and quantify the baseline level of performance that can be expected from the system–in other words, the system will provide performance equal to or greater than its SLOs. In our work, we focus on performance SLOs in the context of storage systems, with an emphasis on tail latency SLOs in CANDStore.

Availability is a concept that quantifies the ability of a system to serve its intended purpose without interruption; a system is considered *available* if it is actively serving the workload it is responsible for, and considered *unavailable* or failed if it cannot adequately serve requests.

Two important metrics, *mean time to failure* (MTTF) [86, 92] and *mean time to repair* (MTTR) [45], help quantify a system's availability. MTTF measures how long, in expectation, a system operates normally before experiencing downtime. MTTR describes how long it takes to restore the system to its previous level of operation after a failure. The *availability* of the system is then $\frac{\text{MTTF}}{\text{MTTF+MTTR}}$.

Improving a storage system's availability is achieved by increasing time to failure or decreasing time to repair. Time to failure is affected by many factors, such as the number of replicas in a system, the quality and redundancy of the underlying hardware, and isolation of replicas with regards to correlated failure-inducing events. Time to repair, in terms of software-level system design, is primarily determined by the design of the recovery protocol and availability of compute/storage resources to facilitate recovery.

Durability refers to the ability of a storage system to store data in a manner that is resilient to corruption or loss. For example, storing data on persistent storage such as hard disk can provide increased durability compared to storing it on volatile DRAM.

Replication is a technique that is commonly used to provide resilience to component faults or failures without compromising the availability or durability of a system. In our work, we focus on data replication in the context of storage systems. In the context of durability, replication of data is important to provide resilience to hardware failures that could cause loss or corruption of data.

In the context of availability, replicating data across multiple physical storage devices or servers is important to ensure minimal interruption of availability in the event of hardware failure.

There are many different schemes by which storage systems replicate data, and a full description of all variations is beyond the scope of this document; the primary replication schemes discussed in our work are primary-backup and erasure coding.

For the sake of brevity, we describe primary-backup in the context of distributed storage systems, and erasure coding in the context of single-node RAID-like storage systems–however, primary-backup and erasure coding are applicable in both distributed and single-node deployment scenarios. Modern primary-backup systems typically involve a single primary that replicates updates to one or more backups. In the event of node failure, typically an external authority (e.g., a configuration management system like Zookeeper [53]) alters the cluster's configuration and instructs backup nodes to undergo whatever steps are necessary to restore availability to the system. This typically involves reconstructing a new primary from the backups, or promoting one of the backups to be the primary.

Erasure coding [103] is a method of encoding data to provide redundancy without the overhead of full replication. An object is erasure coded by dividing it into chunks, then encoding those chunks into an expanded set of fragments that have some degree redundancy. The key property of erasure codes is the ability to reconstruct the original data from any sufficiently large subset of fragments. For example, if an object is divided into K chunks, then encoded into N fragments, the object can be reconstructed by taking any subset of K fragments.

2.2 Hardware technologies

2.2.1 NVMe

Non-Volatile Memory Express (NVMe) is a software-based standard that specifies an interface optimized for PCIe-based SSDs [79, 107]. NVMe-based drives provide significantly improved throughput and latency compared to equivalent SATA-based drives, and has become the primary interface of choice for high performance flash-based block storage.

2.2.2 Intel Optane

Intel Optane is a now-discontinued byte-addressable non-volatile main memory (NVMM) technology with latency and read throughput similar to volatile DRAM, but with persistence, lower cost per byte, and higher density per DIMM. A primary weakness of Intel Optane is that unlike DRAM, the write throughput is much lower ($\frac{1}{7}\times$) than the read throughput.

2.2.3 Traditional block interface SSDs

The *traditional* block interface is a software interface employed by block storage devices such as HDDs and conventional SSDs. Data within these devices is addressed in units of blocks, which are typically 512 or 4096 bytes [25].

Our work focuses on SSDs, which store data in flash blocks (typically 64–128 KiB) which can be written (programmed) at 512 or 4096 byte granularity, but cannot be overwritten without first

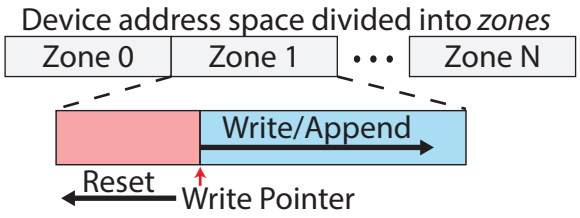


Figure 2.1: ZNS divides the block device address space into *zones*, each of which can be written sequentially and erased as a single unit.

being erased in entirety. This mismatch between the traditional block interface and flash memory has forced manufacturers to introduce the flash translation layer (FTL) to ensure compatibility between SSDs and existing software. The FTL is a firmware component that maps logical block addresses (LBAs) specified by the host software to physical block addresses (PBAs) corresponding to actual locations on physical flash media. Instead of overwriting blocks in-place, overwritten LBAs are simply remapped to different PBAs, with the data in the previous LBA being marked for deletion.

A major consequence of the FTL is that any valid data within a flash block has to be copied to a different location before the flash block can be reset in a process called garbage collection. Garbage collection can result in severely reduced performance in SSDs [27] as the device must allocate bandwidth to copying data internally rather than serving the external workload. In recent years, there has been interest from both academia and industry in moving away from the block interface for SSDs for the reasons outlined above [25, 33, 48, 58, 71, 84, 97].

2.2.4 Zoned Namespaces

There are several key differences between the ZNS and block interface standards, which we outline in this section. At a high level, ZNS moves the responsibility of coarse-grained data placement/deletion and garbage collection from the FTL to the host, while continuing to handle physical media reliability and wear leveling within the device firmware. The host is indirectly aware of the physical layout of flash on the device through fixed-size *zones*, and must place data into these zones and perform deletion on zone-granularity. This eliminates the opaque and unpredictable behavior of the FTL with regards to garbage collection, and reduces the amount of overprovisioned flash memory on ZNS SSDs compared to conventional SSDs.

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*, illustrated in Figure 2.1. 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 determine 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 2.1 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 $2\,\mathrm{GiB}$ and a zone capacity of $1077\,\mathrm{MiB}$, meaning the upper $971\,\mathrm{MiB}$ of each zone cannot be read or written. This unwritable space allows the first block of each zone to start at a multiple of $2\,\mathrm{GiB}$, which simplifies various address calculations.

Zone append. In contrast to typical write commands where the host specifies the LBA at which to write data, the new *zone append* command allows the host to write data to a specified zone, receiving the precise LBA at which the data was written after the IO completes. Unlike normal write commands, zone appends are not guaranteed to be written 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 devices 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 device has reached the end of its life. As erase blocks fail, they are replaced by overprovisioned blocks, and only once the device runs out of extra blocks do zones start failing.

2.2.5 Application compatibility

Most applications that rely on the block interface cannot run directly on ZNS devices without some modification of the software stack. There are existing solutions to abstract away the ZNS interface and allow current applications to run on ZNS hardware, but 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 devices.

Host-side FTLs such as dm-zap [18] expose a block interface for non-block interface devices, such as ZNS or Open Channel SSD (OCSSD), by handling logical-to-physical block remapping and garbage collection in software. Similar approaches include dm-zoned [77], pblk [26], and SPDK's FTL [75].

Filesystems are beginning to add ZNS SSD support, starting with F2FS [67] in kernel 5.10 [7] and btrfs [89] in kernel 5.12 [91]. A key limitation of both host-side FTLs and current ZNS-compatible filesystems is that they typically require a conventional block interface device or namespace to hold metadata [20].

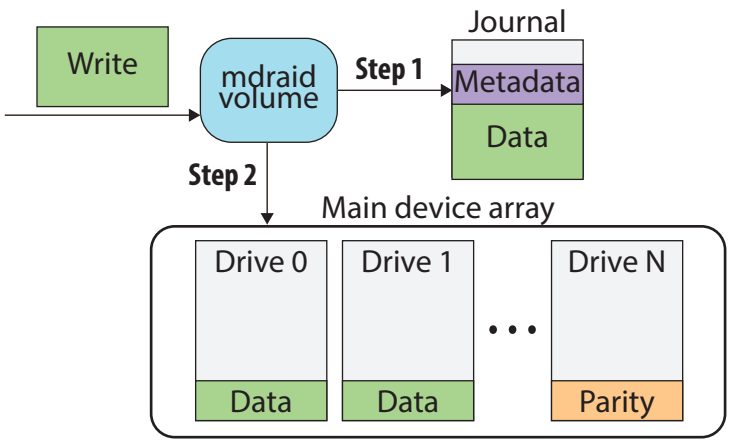


Figure 2.2: When the journal is enabled, `mdraid` first writes data to the journal device before flushing it out to the disk array. The journal must have equal to greater write throughput than the rest of the array combined to avoid becoming a bottleneck.

2.3 Related systems

In this section we briefly describe some of the storage systems that are relevant to our work.

2.3.1 RAMCloud

RAMCloud is a DRAM-based low-latency distributed key-value storage system [81, 83] that uses a primary-backup replication scheme. Primary replicas in RAMCloud store data on high-throughput low-latency DRAM, while backups store data on slower (but cheaper) block storage devices such as HDDs. RAMCloud achieves high availability through its highly parallel primary reconstruction protocol which can replace a failed primary for a single shard in 1–2 seconds [81].

2.3.2 RAID

RAID refers to a family of data distribution techniques that aim to achieve redundancy, performance, and availability by leveraging multiple physical devices [86]. RAID is able to provide availability for an array of drives in the face of device failures by parity coding data across the drive array. This allows continued operation of the array following the failure or removal of one drive from the array. Traditional RAID is known to have key shortcomings in reliability, due to the likelihood of correlated failures[92] as well as the difficulty of ensuring failure independence of drives physically located in close proximity. In modern systems, RAID is often implemented in software as a logical device that can be treated as a normal block device, or is sometimes integrated into the filesystem [89, 112]. Several aspects of RAID are incompatible with the ZNS interface–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.

`mdraid` [11] is a software RAID implementation in the Linux kernel, and serves as the baseline point of comparison against our system RAIZN. `mdraid` optionally supports a write jour-

nal [29] to close the write hole and provide atomic writes. When the write journal is enabled, `mdraid` handles user writes by first writing the data plus some associated metadata to a journal device, then flushes the data to the device array after it has been persisted to the journal, as is illustrated in Figure 2.2. `mdraid` with the journal enabled (to be referred to as `mdraid-journal`) requires the journal volume to be fast enough to serve the client write workload to avoid becoming a bottleneck, and the write throughput and latency of `mdraid-journal` is equal or worse to `mdraid` without a journal.

Chapter 3

CANDStore

In this section, we describe the design, implementation, and evaluation of our highly available replicated Intel Optane-based key-value storage system: CANDStore. CANDStore is a consistent, available, non-volatile, distributed store, that leverages NVMM to provide low cost distributed key-value (KV) storage and a novel online recovery protocol to enable fast recovery after primary failure despite the low write throughput of Optane NVMM.

3.1 Motivation

Distributed storage systems that provide high performance, fault tolerance, and strong consistency are at the core of today's large-scale Internet services [35, 41, 43, 101]. Non-volatile main memories (NVMMs) offer an enticing design option for building these high performance storage systems. One might hope that they could offer the performance of DRAM-based systems such as RAMCloud [82] and FaRM [41], while providing durability and having lower total cost. Such hopes are not fanciful: compared to DRAM, one can pack about 4x more of Intel's Optane [55], for example, into a single machine at approximately 1/7th [22] the cost, twice the read latency, and 1/3rd the read bandwidth.

Unfortunately, today's NVMMs come with drawbacks [106, 108] that complicate the design of failure recovery in distributed storage systems, and overcoming these drawbacks at the system design level is the focus of this paper. The combination of substantially lower write bandwidth ($7\times$) [22] and higher density ($4\times$) compared to DRAM means that the time to recover from a failed node is up to 28x longer than a DRAM-based system with a similar number of total DIMMs. Using traditional mechanisms that temporarily halt serving requests during recovery, each machine failure would take 28x longer to recover when using NVMM.

3.2 CANDStore overview

CANDStore leverages temporal locality in datacenter workloads to enable online recovery from primary failure, and a novel Raft-based [80] protocol that incorporates ideas from Cheap Paxos [66] to maintain strong consistency.

We leverage two observations to allow CANDStore to forego standard offline recovery approaches and focus on techniques to make online recovery feasible and performant. The first ob-

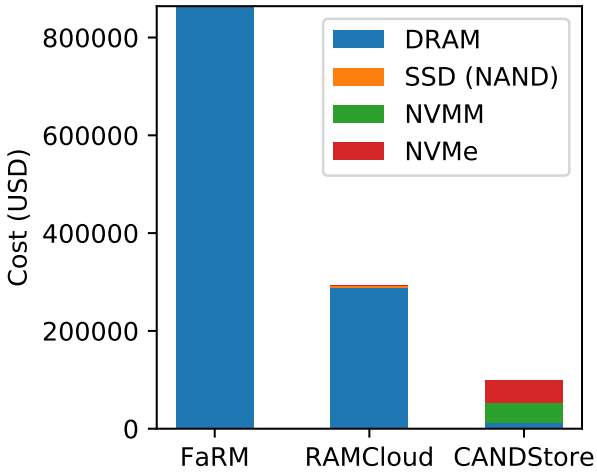


Figure 3.1: Estimated cost to store an $8\,\mathrm{TB}$ dataset on 1 primary and 2 backups.

servation is that real-world datacenter workloads have skewed request distributions [28, 30, 32, 49, 63, 109], allowing CANDStore to guide the online recovery process in a manner that enables the majority of requests to be served at *near-peak* performance before all of the data is copied from a backup to the new primary. Second, we observe that the availability of a distributed storage system is defined by its ability to achieve performance service level objectives (SLOs), and that typical distributed storage systems are deployed with some degree of performance "headroom" [85, 93, 111]. In this thesis, we focus primarily on tail latency SLOs, and any further references to SLOs refer to tail latency SLOs unless otherwise specified. CANDStore takes advantage of these observations when rebuilding a failed primary first copying "hot" keys to the primary, expediting the return of the system to operating within SLO expectations. Keys which are not present on the primary are fetched using point queries to the backup replicas (Section 3.6.2), and the performance headroom combined with workload skew enables the handling of these requests without violating SLOs. After "hot" keys are copied to the primary, "cold" keys are slowly ingested by the new primary, eventually returning CANDStore to full performance and fault tolerance.

To provide consistency and fast failure recovery, CANDStore uses a decentralized distributed consistency protocol based on a modified version of Raft [80]. This protocol ensures that key-value updates are replicated consistently and durably, and that stale or inconsistent data is never returned to the client. Our protocol is designed to enable CANDStore to use a heterogeneous layout of nodes, described in Section 3.3 and Figure 3.3, including a backup that stores keys on NVMe SSD to reduce the cost of replication. We add additional phases to the failure detection and leader election parts of the protocol aimed at minimizing the performance degradation experienced by our system following primary failure, which we describe in Section 3.5. We discuss the design of our protocol for consistently handling non-primary failure in Section 3.7.

Figure 3.1 describes the cost of maintaining a primary and 2 backups in RAMCloud, FaRM, and CANDStore, calculated using the storage prices in Figure 3.2. In this figure, we include the cost of 2 witnesses (equivalent to hot swap space in RAMCloud) for CANDStore, but do not include the cost of hot swap capacity in RAMCloud. We find that by using NVMM as primary storage, it is possible to achieve a $3.4\times$ reduction in storage cost compared to RAMCloud, and a

Storage type	Cost per gigabyte (USD)
DRAM [4]	$35.16
Intel Optane NVMM [4]	$4.51
SSD (NAND) [22]	$0.32
NVMe (Optane) [15]	$2.84

Figure 3.2: Cost of different types of storage.

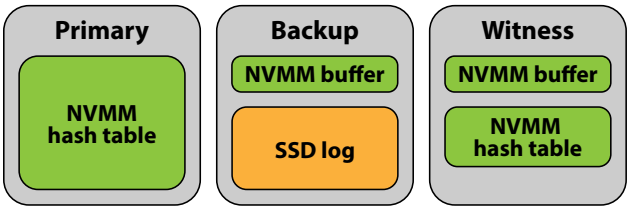


Figure 3.3: Node types in CANDStore.

$10\times$ reduction compared to FaRM. NVMM, while cost-effective, is not without its disadvantages–with reads and writes taking 175ns and 94ns respectively, it has worse latency than the $82\,\mathrm{ns}$ of DDR4 DRAM. NVMM offers only $32\,\mathrm{GB/s}$ read throughput and $11.2\,\mathrm{GB/s}$ write throughput, significantly lower than the $107\,\mathrm{GB/s}$ reads and $80\,\mathrm{GB/s}$ writes for DDR4 DRAM [22]. We evaluate the performance of our recovery protocol and find that it is possible to recover from primary node failure 4.5–$10.5\times$ faster than offline recovery approaches running on the same cluster configuration.

3.3 Design

CANDStore is a distributed KV store designed to provide high availability at low cost in the face of primary failure, despite the infeasibility of fast offline recovery when using Intel Optane NVMM. In this section, we introduce CANDStore and describe the steady state operation of a single shard of a sharded datastore. While not the focus of this work, the system builds knowledge about workload skew during steady state operation, as uses this to speed up the recovery process, detailed in Section 3.4.1. We begin in Section 3.4 by describing how GETs and PUTs are handled and replicated in the context of a simplified three-node cluster consisting of a primary node, a backup node, and a witness node (Figure 3.3). The witness node behaves similarly to an auxiliary node from Cheap Paxos [66], and also serves a similar purpose to hot swap space in a RAMCloud cluster. It is responsible for enabling consistency without paying the full cost of a new primary or backup node, as well as serving as the new primary in the event the primary fails. By including the witness node in the consensus protocol, rather than allocating it on-demand after primary failure, we can take advantage of its knowledge about which local keys are up-to-date when handling requests during live recovery (Section 3.6.2). However, to realize this cost-effective heterogeneous node layout while maintaining consistency, we needed to make several modifications to the Raft protocol, which we explain in Sections 3.4 and 3.5. In Section 3.4.1 we describe the optimizations we used to improve the performance of the backup node. Finally, we describe how our system identifies distribution skew in client requests (Section 3.4.2) and how we took advantage of this

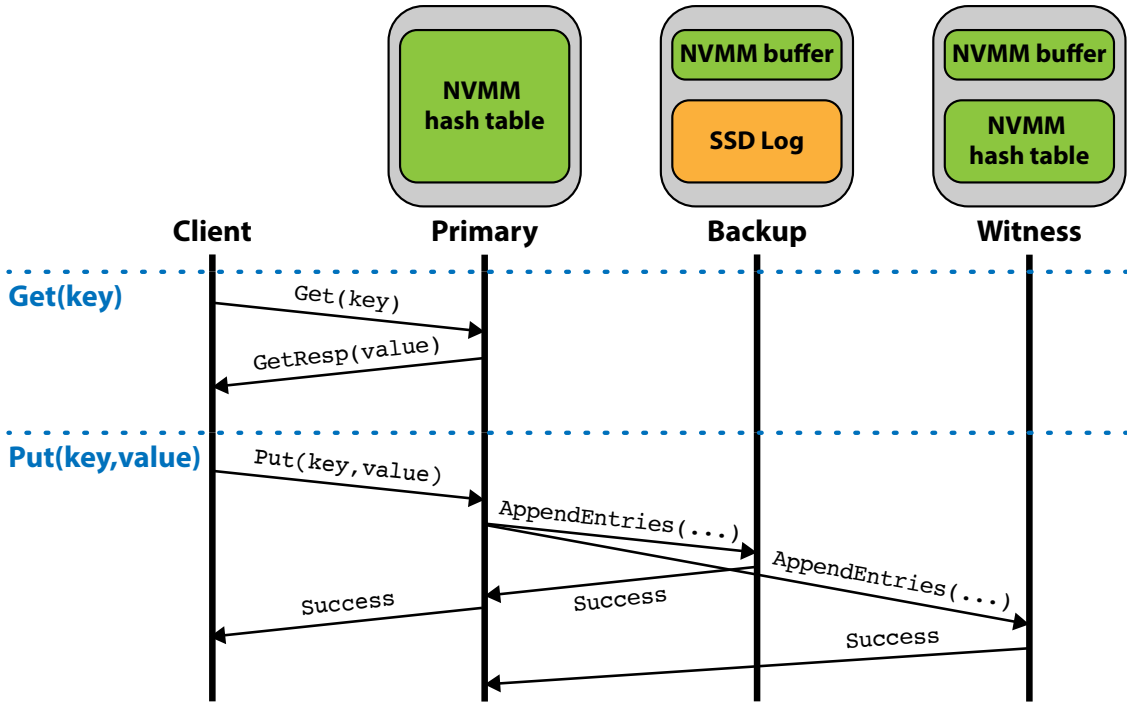


Figure 3.4: CANDStore steady state RPC behavior.

skew to improve the TTR of the system following primary failure (Section 3.5).

3.4 GETs and PUTs in steady state

We use a heterogeneous node layout in CANDStore to provide cost savings compared to a homogeneous node layout like traditional Raft/Paxos formulations and FaRM [41]. In this section, we outline the modifications we made to the Raft protocol to provide consistency when handling client requests in the absence of failures.

Client GET requests, shown in Figure 3.4, can be handled in 1 round trip by the primary using leases [78]. The client sends GET(key) to the primary, and the primary responds with the value corresponding to key.

Client Put requests, also shown in Figure 3.4, require the primary to replicate the request to a quorum of nodes in the cluster before committing and responding to the client. Similarly to Raft, this replication is handled by AppendEntries RPCs. However, because the witness nodes behave similarly to Cheap Paxos and do not store values for committed updates, we impose the additional constraint that the quorum for client updates contains all voting backup nodes in the current cluster configuration. In the event of a backup node failure, the system must execute a viewchange to remove the failed node from the cluster configuration and, by extension, the quorum. Astute readers may have realized that this additional quorum constraint makes Raft's configuration change impossible; our solution to this is described in Section 3.5.3.

Upon receiving an AppendEntries request, the backup persists it to a small NVMM buffer, then responds to the primary. These updates are batched and periodically written out to SSD by a dedicated writeback thread. This design has two main advantages: backup nodes can replicate

keys at NVMM latency (rather than SSD latency), and batching writes to an SSD improves its throughput (Section 3.8.6).

A witness node behaves similarly to a backup node, keeping updates on a small NVM buffer until they are committed. However, the witness discards the value when committing the update, instead storing only the key and a logical timestamp composed of the Raft term and index. This differs from both Raft, where the value would not be discarded, and Cheap Paxos, where the witness would never receive the value in the first place. We design the witness nodes in CANDStore to behave in this way because, unlike Cheap Paxos, only witness nodes can be promoted to become the primary.

Each node commits updates in a different way. The primary node commits updates by writing the key, value and logical timestamp to its local NVMM-backed hash table. The backup commits updates by serializing the key, value, and logical timestamp to SSD, then updating its DRAM index to reflect the location and timestamp of the latest update of the key. The witness commits updates by writing only the key and logical timestamp to its underlying hashtable, discarding the value.

3.4.1 Improving backup performance

As we will discuss in Section 3.6, the performance of the backup node is the most important factor in quickly restoring the system to in-SLO operation. In this section, we discuss the techniques we use to optimize the operation of the backup node to facilitate faster crash recovery.

Figure 3.5 describes the internal design of a backup node. To improve its throughput and latency, each backup node uses multiple *workers*, each maintaining exclusive write access to their own NVMM-resident buffers and SSD-resident logs. Each worker maintains two sets of logically-separate NVMM buffers and SSD log files, one for popular (i.e., "hot") keys and one for unpopular (i.e., "cold") keys. We discuss the purpose of separating hot and cold keys in Section 3.4.2. Each worker has three threads, one to handle RPCs (not explicitly pictured in Figure 3.5), one to write back batches of updates from the NVMM buffer to the SSD log, and one to garbage-collect (GC) outdated log entries.

Upon receiving a key update from the primary, the RPC handler thread checks whether the key is in the hot or cold set, then writes it to the appropriate NVMM buffer before sending a success response to the primary.

The writeback thread sequentially iterates through all of the updates written to the NVMM buffer, waiting for each entry to be committed. Once enough entries on the NVMM buffer have been committed to fill a batch, the batch is then serialized out to SSD. Batching in this way enables the SSD to achieve maximum write throughput. The writeback thread also updates the shared DRAM index after writing a batch to the SSD log, discards the NVMM-resident copy of the batch, and updates the DRAM index to reflect the latest logical timestamp for each batch entry. The DRAM index is only updated if the entry has an equal or greater logical timestamp to the timestamp stored in the index.

The garbage-collection thread periodically reads a batch of entries from the tail of the log, discarding stale entries and writing non-stale entries back onto the corresponding "hot" or "cold" NVM buffer. The staleness determination is handled by reading the latest logical timestamp from the DRAM index, and the determination of which buffer the entry should be written to is handled in the same way as the RPC handler thread. This garbage collection process serves two main

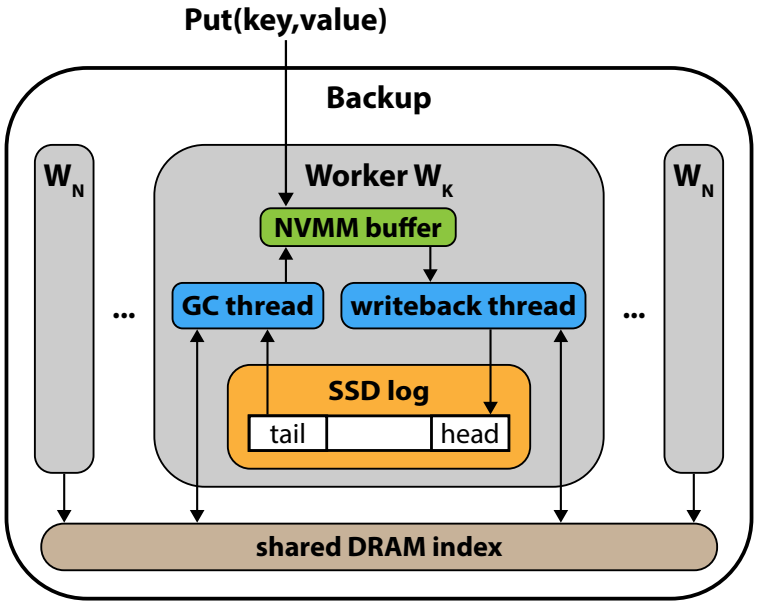


Figure 3.5: Layout of a single backup worker.

purposes: First, it reduces the log's footprint on SSD, and second, it enables keys to move between the hot and cold sections of the log even if they are never updated by the client.

3.4.2 Proactively identifying popular keys

To speed up the recovery process (detailed in Section 3.5), we separate the backup logs into two separate contiguous regions on SSD: one region for the popular or hot keys, and one region for less popular or cold keys. The placement of a key into one of these regions is decided by the writeback threads, and is based on a continuous sampling of the client requests by the primary.

The sampling mechanism we use is similar to that applied by cache admission policies [23]. Upon receiving a client request, the primary, with some low probability (i.e., 0.001), keeps track of the key for that particular request. Periodically, the list of keys that were tracked by the primary is sent to the replica, which maintains a fixed-size list of the most popular keys (using an LRU eviction policy). This list of popular keys is then used by the write-back threads to make a placement decision when updates are committed to the on-SSD log.

CANDStore's garbage collection mechanism, as described in the preceding subsection, ensures that "cold" keys which were, by chance, placed into the "hot" log will eventually be written to the "cold" log, and vice versa.

3.5 Handling failures

In the event of primary failure, there are four differences between our protocol and the standard Raft protocol. In this section, we explain these differences and prove that our protocol maintains both consistency and liveness. A timeline of our recovery protocol is shown in Figure 3.7.

Figure 3.6: CANDStore RPCs

RPC type	Sender	Recipient	Contents	Response contents
AppendEntries	primary	backup, witness	term, leaderid, prevlogindex, prevlogterm, entries	term, success
RequestVote	primary, witness	backup, witness	term, candidateId, lastLogIndex, lastLogTerm	term, lastLogIdx voteGranted
Get	client	primary	key	value
Put	client	primary	key, value	success
BatchPull	witness, primary	backup	lastLogTerm	batch of entries
PriorityPull	primary	backup	key	4k batch of entries
RequestLog	witness	backup	term, index	entry

3.5.1 Leader election

The leader election process in CANDStore is similar to that of Raft, with the additional constraint that only witness and primary nodes can send RequestVote RPCs. This means that Backup nodes cannot become the new leader of the cluster — this design decision ensures that the new primary (i.e., recovery primary) is not coincident with a backup node. The benefit of this is that the backup, which is the most performance-sensitive node in the recovery process, does not experience any contention for resources.

Because the witnesses and primary node constitute at least $F + 1$ nodes in the cluster, our failure assumption guarantees that there is always at least one live witness or primary. Similar to Cheap Paxos, it is possible for the system to lose liveness if the set of non-faulty nodes shifts too quickly. However, unlike Cheap Paxos, which only allows non-witnesses to become the leader, our system only allows voting witnesses to become the leader. We explain in Section 3.6 how a witness transitions from not having complete information about committed KV updates into becoming a fully functioning primary. Additionally, it is possible for all voting witnesses to be unaware of a committable or committed update, as the quorum for KV updates need only contain the primary and all backups in the current view.

To fix this problem, we modify the leader election protocol in the following ways. First, we include the most recent log index in the RequestVote response. If a witness receives a RequestVote response from one of the replicas with `voteGranted = 0`, then it saves its local `lastLogIndex` as `lastLogIndexOld` and updates its `lastLogIndex` to the index received in the RequestVote response. This guarantees that leader election will eventually terminate and choose a new leader.

Upon election, the new leader first compares its `lastLogIndex` with `lastLogIndexOld`. If they are equal, then leader election is complete. However, if they are not equal, then the new leader enters the Reconciliation phase, described in the next section.

3.5.2 Log reconciliation

The purpose of the log reconciliation phase is to allow the new primary to fast forward itself to a sufficiently up-to-date state such that it is possible to handle client PUTs. We call the primary

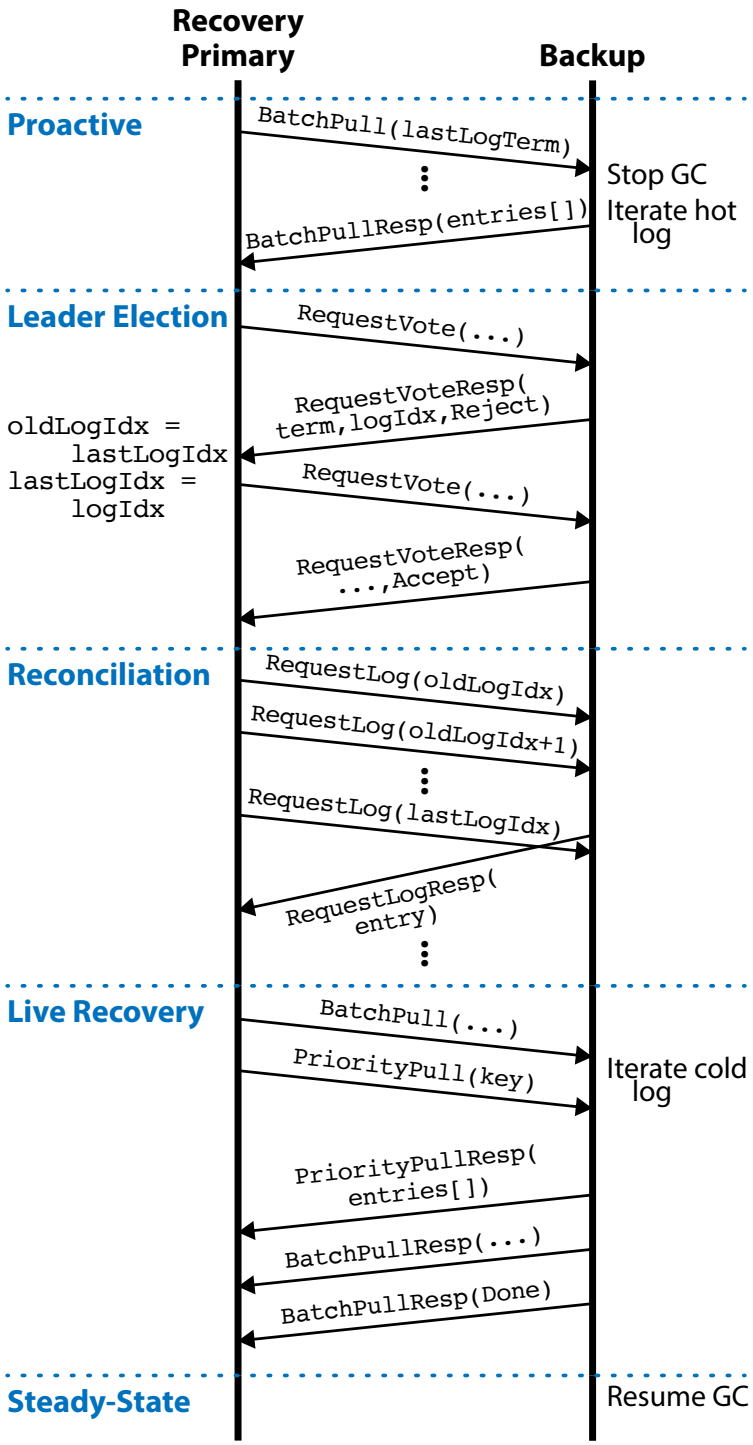


Figure 3.7: Recovery timeline.

during this phase the "provisional leader" of the Raft cluster. During the reconciliation phase, the primary node cannot commit any new log entries. However, it is still able to serve client Get requests using the mechanisms outlined in Section 3.6.2.

The provisional leader must first bring itself up to date by querying the backup(s) for the log entries from `lastLogIndexOld` to `lastLogIndex`. It achieves this by sending a RequestLog RPC to the backup(s) for each log that it suspects it is missing. Because of the Cheap Paxos–style quorum, it is acceptable to issue these requests in parallel to different backups.

Log reconciliation ends when one of two conditions is met: First, if the primary receives a response for each log entry that it is missing, then it can treat these logs as uncommitted and resume accepting client Put requests, replicating and committing these log entries based on the normal operation of the Raft protocol. Second, in the event that one of the backups returns a null entry for index i, then it is acceptable to discard responses for log entries corresponding to index i and above. The modified quorum ensures that any log entries that are not present on all voting backups cannot have been committed.

Proof of modified quorum correctness Suppose that there exists a voting backup B which does not have knowledge of a committed update U. This backup must have been admitted to the cluster as a non-voting node by some log entry L_0 and admitted to the cluster as a voting node by some log entry L_1. By the Raft log prefix invariant and the CANDStore modified quorum, U must have been committed after L_1. This is because after L_0 is committed, all updates must be replicated to B before being committed, as part of Raft's joint consensus state, so if B is a voting node, then L_1 must have been committed, which implies that B has knowledge of L_1. However, due to CANDStore's modified quorum, any updates which are committed after L_1 must be replicated to a quorum including all backups in the current view. Therefore, U must have been replicated to B before being committed.

3.5.3 Configuration change

It is clear that the modified quorum including all backup nodes in the current configuration creates a situation where it is impossible to remove a backup node from the configuration. We solve this problem by amending the quorum definition to include all active backup nodes only for key updates. Both witnesses and backups maintain complete information about the current cluster configuration in stable storage, so this relaxed quorum requirement for configuration change–related log entries does not affect correctness.

3.6 Primary failure

In this section, we describe how our system handles the failure of a primary. We discuss the recovery protocol, and describe how the system serves client requests during the period of degraded performance while the system is repairing. We briefly discuss how the system handles witness and backup failures in Section 3.7.

3.6.1 Proactive recovery

In CANDStore, we begin the recovery process with what we call "proactive" recovery. This phase is called "proactive" because we begin the process before primary failure is detected, whenever we suspect that there may be a problem with the primary. This phase of recovery involves preemptively copying popular KVs to a witness, in anticipation of that witness becoming the new primary.

In addition to standard heartbeats ("hard" timeouts), we introduce a more aggressive (i.e., earlier) "soft" timeout. This soft timeout triggers the first phase of CANDStore's recovery protocol, which we call proactive recovery. Soft timeouts act as a failure detector with a moderate false positive rate; CANDStore aborts the proactive recovery if it receives a heartbeat before the hard timeout elapses.

During proactive recovery, the witness node sends BatchPull RPCs to the backup. Upon receiving a BatchPull request, the backup pauses its garbage collection threads, begins processing its "hot" log and sends the contents to the witness one batch at a time. If at any point during this process the witness receives a heartbeat from the primary, then the witness sends a BatchPull RPC to the backup with the invalid `lastLogTerm` value of 0, which causes the replica to stop iterating through its log and resume garbage collection. Additionally, if the witness sending BatchPull RPCs is removed from the configuration or becomes part of C_{old} in Raft's joint configuration state, the replica stops iterating its log, resumes garbage collection, and rejects BatchPull requests until the witness is restored to voting status and the configuration of the system returns to a non-joint configuration. Informally, if the cluster appears to be reconfiguring to exclude the witness, then the backup ignores RPCs from that witness and returns to steady state operation.

3.6.2 Live recovery

After the hard timeout elapses, the system begins leader election, as described in Section 3.5.1. Immediately following the election of a new primary, which we refer to as the recovery primary, we enter the live recovery portion of CANDStore's recovery protocol.

During live recovery, the recovery primary continues to send BatchPull RPCs to the backup node, which, after completing its iteration of the hot log, begins iterating through the cold log and sending cold keys back to the recovery primary. At the same time, the recovery primary begins handling client requests. One key difference between steady state operation and live recovery is that tasks such as popularity sampling and garbage collection are temporarily paused to reduce resource usage (especially SSD bandwidth) on the backup.

Client Put requests are still handled in the same way as during normal operation of the system. The recovery primary may not be aware of the values for all of the preceding updates, but has still "committed" those updates and is able to commit new client Put requests.

Client Get requests are handled in one of two ways: If the key is present locally on the recovery primary, then it can be served directly, identically to the non-failure case. However, if the key is not present locally, then the recovery primary issues a point query for that key to the backup (Figure 3.8). We call these point queries PriorityPulls, a name taken from a similar mechanism in Rocksteady [65]. The backup uses its DRAM index to determine the SSD page(s) containing the most up-to-date value corresponding to the requested key, and sends the value, along with any other keys stored on the same page, to the primary. In our implementation, this was done to take advantage of bytes read from the SSD regardless of whether we wanted or not, but for certain

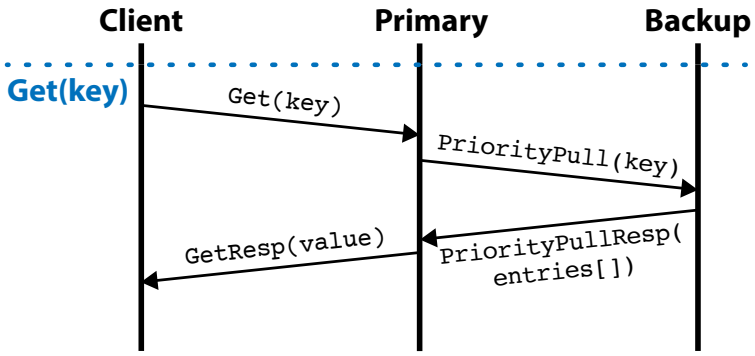


Figure 3.8: Remote Get.

workloads with spatial locality (e.g., clicking one button to add a set of 5 items to the cart), this design could take advantage of that spatial locality. Recent research from Facebook [31] indicates that this sort of spatial locality may be quite common.

Once the backup has determined that it has sent all of its local records to the recovery primary, it notifies the recovery primary via a BatchPull response with 0 entries. Similarly how Squall [42] tracks the progress of migrating tuples, the backup determines when all local records have been successfully transferred to the recovery primary. The recovery primary can then transition into steady state operation as a normal primary.

3.7 Other failure modes

Section 3.5 discussed what happens in the case of primary failure in a CANDStore shard, which is the most complex type of failure. In this section, we discuss the behavior of the system in the event of backup or witness failure. We discuss this in the context of K backup nodes and K witness nodes, where K is tuned by the operator based on the degree of replication desired and the rate of failure of the servers in the cluster. We believe $K = 2$ (triplication) is a reasonable configuration, as many systems such as Amazon DynamoDB [95] use triplication by default, and used $K = 2$ in our price calculations for Figure 3.1.

3.7.1 Backup failure

In the event of backup failure, it is necessary for the system to undergo a configuration change to exclude this backup from the cluster and, by extension, the write quorum. However, in-flight updates that have been replicated to some subset of the backup nodes effectively "block" this configuration change from being committed.

We solve this by temporarily changing the write quorum to exclude the suspected "failed" node, and to include a single witness node in its place. This idea is taken from Cheap Paxos [66], where a witness node is able to help commit log updates and execute a view change in the event of a main processor failure.

It may be desirable to add an additional backup to the cluster to restore the desired level of fault tolerance. This can be done easily and without interruption of availability, despite the constraint of the write quorum including all voting backup nodes. The node can be added to the cluster using

the unmodified Raft configuration change mechanism first as a non-voting node, and only after it is up to date will it be promoted to a voting node and join the write quorum.

3.7.2 Witness failure

If a witness fails, it is not necessary for the system to remove the failed witness from the cluster before making additional progress. To restore the desired level of failure tolerance, it may be desirable to allocate and add a new witness node to the cluster. This can be done according to the protocol outlined in Section 3.5.3.

3.8 Evaluation

This section describes how we evaluated CANDStore's crash recovery mechanism compared to the baseline approach of copying the entire set of keys/values from the replica to the recovery primary. We show that our approach achieves higher availability than traditional primary-backup style approaches while maintaining cost efficiency and high performance.

Due to constraints on available hardware, we did not shard the recovery primary or backups. However, the techniques we use and the resulting relative improvements in availability are still applicable, and our system design benefits from the parallelism afforded by optimizations such as sharded backups and multiple/striped SSDs. In particular, our recovery protocol benefits greatly from spreading load across multiple SSDs, as the main cause of SLO violations during online recovery is high SSD load.

3.8.1 Testbed setup

Our evaluation testbed is a 3-node cluster. Each machine has two Intel Cascade Lake processors with 24 physical cores clocked at 2.2 GHz with hyper-threading enabled. The platform has $192\,\mathrm{GiB}$ of DDR4-2666 DRAM and $768\,\mathrm{GiB}$ of NVMM (Intel Optane DC 2666 Mhz QS [55]) per socket for a total of $384\,\mathrm{GiB}$ of DRAM ($12 \times 32\,\mathrm{GiB}$) and $1.5\,\mathrm{TiB}$ of Optane ($12 \times 128\,\mathrm{GiB}$) per server. One $375\,\mathrm{GB}$ Intel P4800X NVMe SSD [14] was installed on each server. Each node also had a 1-port Mellanox ConnectX-3 NIC [76], and the servers were interconnected by a $56\,\mathrm{Gbps}$ InfiniBand switch.

For all experiments, we set up a single region consisting of the $768\,\mathrm{GiB}$ of NVMM located on NUMA node 0 in AppDirect [100] mode. We format the region into an ext4 filesystem and use libpmem [10] to facilitate reading and persistently writing to NVMM.

We were unable to test a configuration with multiple backups or multiple SSDs to exploit parallelism during the recovery process, as we did not have access to more than 3 servers and 1 SSD per server. As a result, we compare our performance against a baseline of how offline recovery would perform on our testbed configuration. Our performance would be at least proportionally improved by the use of cluster-level parallelism or multiple SSDs on the backup. In particular, tail latency during the live recovery phase would be improved by having multiple SSDs per backup node, thus reducing the per-SSD load and driving down tail latency.

3.8.2 Client design

In our evaluation, we used a closed-loop client with independent threads; each thread sends one or more streams of requests to each server thread, and the next request in the stream is not sent until a response is received for the preceding request.

3.8.3 Workloads

We evaluate our system using YCSB [34] workloads B (95% GET, 5% UPDATE) and C (100% GET) using YCSB's zipfian request distribution. In addition to using YCSB's default parameterization of the zipf distribution ($\theta = 0.99$), we also explore less-skewed and more-skewed workloads ($\theta \in \{0.9, 1.1\}$). We omit workload A (update heavy) due to issues outlined in the next section.

Due to resource constraints, we benchmark the system with a 128 GB shard. We believe that shards of real distributed storage systems using NVMM will be larger, but the relative benefits of our recovery approach will still apply.

3.8.4 Steady state performance

As the focus of this project was on improving performance during recovery, we did not do a comprehensive evaluation of the steady state performance of our system. For YCSB workload C with the default zipfian distribution and $512\,\mathrm{B}$ key-value pairs ($16\,\mathrm{B}$ key, $496\,\mathrm{B}$ value), and a latency-sensitive configuration of 8 server threads handling requests from 8 client threads with each client thread sending a single request at a time, our system can serve at a throughput of $1.5\,\mathrm{GB/s}$ while maintaining a median latency of $7\,\mu\mathrm{s}$ and a 99^{th} percentile tail latency of $51\,\mu\mathrm{s}$. For the equivalent setup running on YCSB workload B, our system achieves $1.0\,\mathrm{GB/s}$ with a median latency of $7\,\mu\mathrm{s}$ and a 99^{th} percentile tail latency of about $73\,\mu\mathrm{s}$.

When run in a throughput-oriented configuration of 8 client threads requesting a window of 64 concurrent requests to 8 server threads, we can achieve approximately $4.5\,\mathrm{GB/s}$ in workload C and $3.9\,\mathrm{GB/}$ in workload B.

The decreased performance on workload B is a result of our system being poorly optimized for PUTs. During live recovery, the behavior of PUTs does not change, but GETs can incur in an additional round trip of communication for each PriorityPull. As a result, we focused mainly on optimizing GETs, but we believe that with additional optimizations and larger NVMe SSDs (large enough to comfortably pre-allocate spare log space without filling up the SSD) the performance of PUTs in our system will increase.

Our goal is to have engineered enough of the system such that our evaluation of the recovery protocol is informative; as such, this performance is reasonable–it achieves 65% the throughput of its InfiniBand link, and has a median latency of $7\,\mu\mathrm{s}$ which is comparable to the $4.7\,\mu\mathrm{s}$ read latency that RAMCloud achieved in its original evaluation environment [83].

3.8.5 Metrics

We compare the time to repair (TTR) of our approach with traditional log-replay style crash recovery protocols. We measure the time elapsed between primary failure and when the system achieves

Block size (KiB)	Read throughput (KiB/s)	Write throughput (KiB/s)
0.5	132,647	16,703
1	286,196	38,715
2	614,236	110,490
4	1,668,613	1,627,756
8	2,625,276	2,148,087
16	2,647,880	2,204,804
32	2,651,732	2,212,097
64	2,652,257	2,212,228
128	2,652,623	2,211,742

Figure 3.9: Block size vs SSD throughput.

and maintains a particular latency SLO. For each workload type and distribution, we measure the TTR for a range of latency SLOs.

We also measure the penalty our approach incurs with regard to the amount of time it takes to fully copy the entire key space from the replica to the primary. This is the primary tradeoff of using CANDStore, as it is necessary to slow down the rate of BatchPulls to avoid increasing the latency of PriorityPulls, i.e. we trade increased total recovery time for the ability to do online recovery.

In all of our experiments, tail/median latency is calculated on a $100\,\mathrm{ms}$ non-overlapping sliding window, and TTR is calculated to be the end of the first window that achieves latency within the SLO such that all following windows do not have latency higher than the SLO.

3.8.6 SSD block size effects throughput

SSDs can achieve higher throughput when the block size for IO requests is sufficiently large [102]. We verify this is the case for our SSD, and determined the optimal block size for reading and writing, shown in Figure 3.9. We find that for our disk, a block size of $32\,\mathrm{KiB}$ maximizes throughput for both sequential reads and sequential writes. The throughput at a block size of $32\,\mathrm{KiB}$ matches the advertised throughput of this drive [14]. We use this optimal block size to determine the batch size used for reading and writing to the log (to maximize throughput).

3.9 Tail latency

We evaluate the availability of our system in terms of median and 99.9^{th} percentile tail latency of client requests. These experiments were run in a latency-sensitive configuration of 8 client threads, each sending one stream of requests to one of 8 server threads. The backup uses 2 threads to ingest the log.

To provide a fair point of comparison against an equivalent offline recovery approach, we calculated the minimum time necessary to read the entire shard ($128\,\mathrm{GiB}$) from SSD at the maximum read throughput of the SSD (as measured in Section 3.8.6). For our drive and a $128\,\mathrm{GiB}$ shard, the baseline for offline recovery is $50.6\,\mathrm{seconds}$.

For our tail latency–sensitive configuration, it takes $201\,\mathrm{s}$ ($4\times$ longer) to completely copy all data from the backup to the new primary due to rate-limited BatchPulls.

In Figures 3.10 and 3.11, the TTR for each SLO shown is the median of 3 trials. Figure 3.10 shows the TTR of our system compared with the baseline offline recovery approach. This figure is organized into 3 blocks of 6 graphs each, each corresponding to a different key-value pair size indicated by the label below the block. Within each block, there are 2 rows and 3 columns; each row corresponds to a different YCSB workload, indicated by the label to the right of the row, and each column corresponds to a different request distribution skew, indicated by the label at the top of each column. For example, the top left graph of each block corresponds to an experiment run with YCSB workload B with a skew of $\theta = 0.9$.

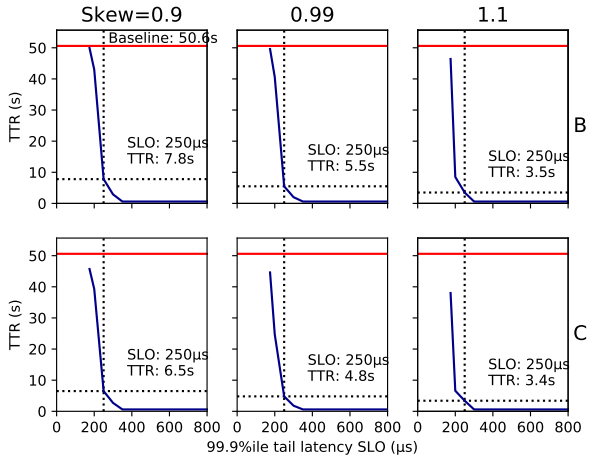
Each graph shows the TTR (y-axis) for a particular latency SLO (x-axis). Lower TTR is better, and in general a higher latency SLO results in lower TTR. For example, looking at the top left graph, the TTR for a 99.9^{th} percentile tail latency SLO of $250\,\mu\mathrm{s}$ is $7.8\,\mathrm{s}$ (labeled), and the 99.9^{th} percentile tail latency for an SLO of $300\,\mu\mathrm{s}$ is $2.9\,\mathrm{s}$ (not labeled).

As the tail latency SLO approaches (left on the x-axis) the steady state tail latency of the system, the TTR of our online recovery approach increases, getting worse. As we mentioned in Section 3.1, our recovery approach is built with the assumption that there will be some performance leeway in the system, with a gap between the steady state performance of the system and the SLO. As the tail latency SLO decreases, the degree of performance leeway decreases, and at a certain point it is better to do offline recovery. However, tail latency–sensitive storage systems will often have resources overprovisioned [38], creating the headroom necessary for our approach to be effective.

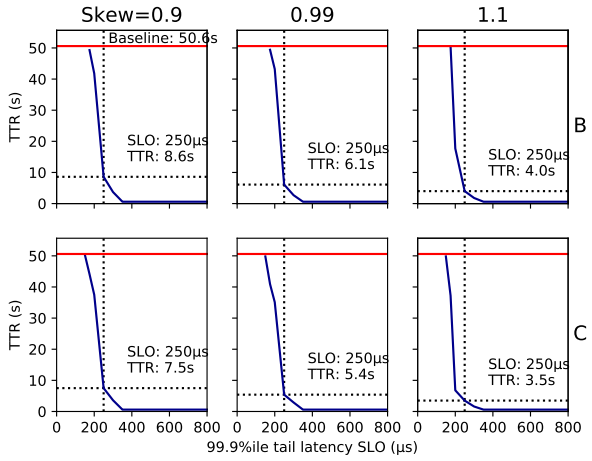
For the standard YCSB zipfian distribution skew of $\theta = 0.99$, our results show that with a 99.9^{th} percentile tail latency SLO of $250\,\mu\mathrm{s}$, CANDStore achieves a TTR of 4.8–$11.1\,\mathrm{s}$, which is 4.5–$10.5\times$ lower than the offline approach running on the same cluster. In all experiments, regardless of workload, tuple size, and SLO, workload skew was directly correlated with CANDStore TTR. CANDStore's recovery protocol is naturally able to exploit marginal increases in temporal locality in the client workload through the prioritization of popular keys in preemptive and live recovery; more skew in the client workload results in a larger proportion of the client's working set ingested during the initial seconds of recovery, which directly reduces tail latency as more client requests can be served directly from the primary rather than through PriorityPulls. In addition to this, larger tuple sizes resulted in increased TTR for CANDStore with all other variables (skew, workload) kept constant. This is due to increased SSD read latency and network latency when reading and transmitting larger values respectively. Note that if the value size exceeds the ability of the SSD backups to serve requests within the latency SLO, CANDStore cannot provide any benefit over offline recovery; however, we expect this situation to be rare, as it would imply little to no headroom between the non-degraded performance of the system and the latency SLO.

3.9.1 Median latency

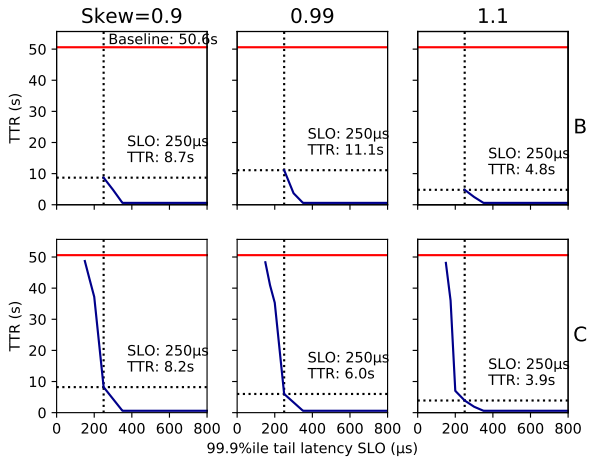
Figure 3.11 shows how long it takes our system to restore a certain level of median latency. It takes only 0.6–$2.4\,\mathrm{s}$ to restore the system to $5\,\mu\mathrm{s}$ median latency (steady state latency) for workloads with a skew of 0.99.



256 B KVs

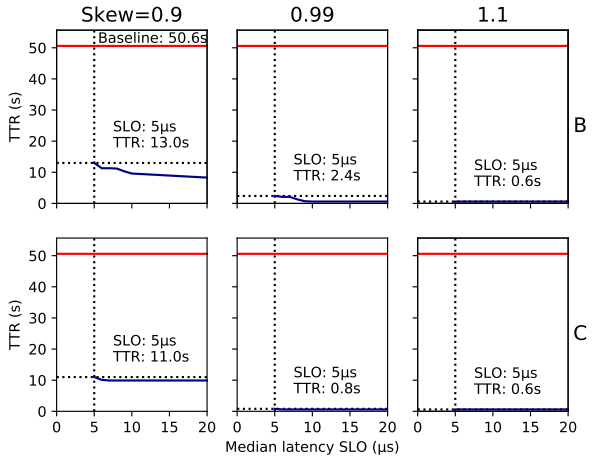


512 B KVs

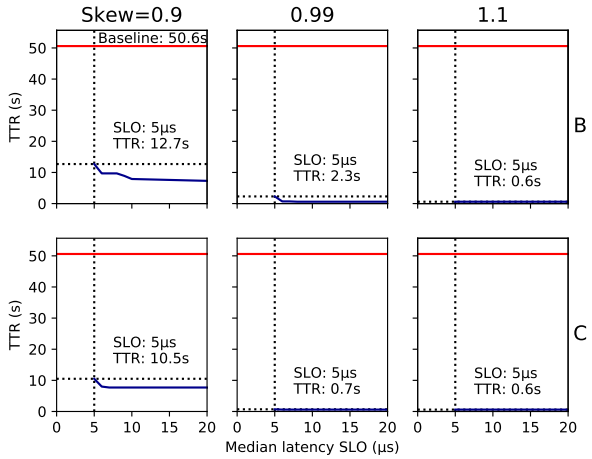


1 KiB KVs

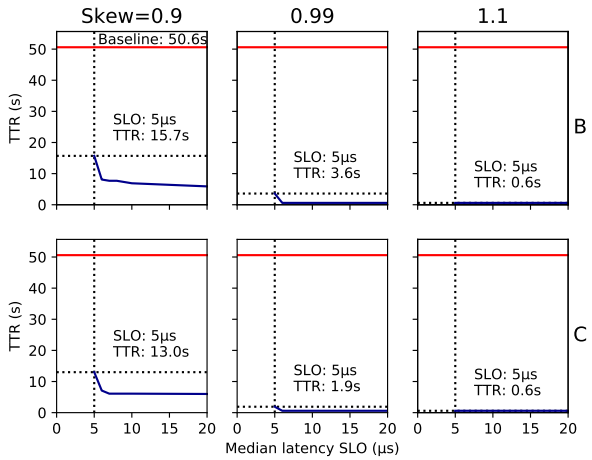
Figure 3.10: TTR for 99.9^{th} percentile tail latency SLO (lower is better).



256 B KVs



512 B KVs



1 KiB KVs

Figure 3.11: TTR for median tail latency SLO (lower is better).

3.10 Related work

We briefly explore related work on high performance storage systems and their differences to CANDStore.

KVell [70] describes the design of a fast, persistent KV store leveraging NVMe SSDs. The simple design of an unstructured SSD-resident log paired with an in-memory index is similar to the design of our backup node. KVell shows that it is possible to get sub-millisecond latency for random reads on modern NVMe SSDs, but that too many simultaneous requests can result in much higher latency. We leverage this in conjunction with workload skew to enable PriorityPulls to be served at low latency.

Unlike CANDStore, KVell handles crash recovery only in the context of recovering local state from persistent storage after a crash, rather than failing over to a replica.

Rocksteady and Squall (H-store) Rocksteady [65] and Squall [42] are both systems that enable reconfiguration of in-memory datastores. Both focus on reconfiguring in a way that minimizes the performance impact during the reconfiguration process.

Both Squall and Rocksteady solve some similar problems to CANDStore, but in the context of database reconfiguration, copying tuples from an in-memory primary partition to another in-memory primary partition using a combination of large asynchronous batched copying and fine-granularity on-demand fetching. In database reconfiguration, there is flexibility in the rate at which the tuples are migrated–this contrasts with primary crash recovery where restoring in-SLO performance is time-critical. Unlike main memory, increasing queue depth for I/O operations to SSD quickly increases latency. In CANDStore, we solve the problem of primary crash recovery, with source tuples stored on SSD, meaning more pressure to copy tuples quickly, and less performance headroom to do so without affecting latency. In CANDStore, this necessitated design choices such as hot/cold tiering of the backup, and aggressive timeouts for proactive recovery.

RAMCloud As discussed earlier, our system resembles RAMCloud [81], with primary nodes serving KVs using fast main-memory based storage, and backups storing updates on SSD in log format. However, in our system, to maintain high performance and availability when using NVMM instead of DRAM we use a different crash recovery protocol, described in Section 3.5.

FaRM [41] is another high performance in-memory datastore, which uses capacitor-backed DRAM to provide persistence. FaRM provides much richer transactional semantics and much higher performance when compared to CANDStore, particularly for writes. However, FaRM maintains multiple copies of the dataset in DRAM, incurring a much larger cost compared to CANDStore, which stores backups on SSD.

FaRM includes a phase of primary crash recovery which involves identifying tuples involved in transactions which were interrupted by the primary failure. We drew inspiration from this design, identifying log updates which were "interrupted" before they could be replicated to the witness node in our modified leader election and log reconciliation phase of recovery (Sections 3.5.1 & 3.5.2).

Silo Silo [99] is another high performance KV store, but they do not discuss replication as a means of crash recovery.

Gemini [47] is a system for fast crash recovery of persistent caches. The focus of Gemini is to enable a recovering primary to immediately serve client reads and writes at high performance without violating consistency. Similar to CANDStore, Gemini prioritizes copying popular cache entries to ensure that a large proportion of requests can be served by a recovering cache server.

Unlike our system, the primary concern when recovering a primary is load balancing, as tuples can be offloaded to other servers in the system immediately following primary failure. Gemini is a cache, so it only provides RAW consistency for single keys, rather than linearizability.

3.11 Lessons from CANDStore

This section describes several of the lessons we learned from designing, implementing, and evaluating CANDStore; these mistakes and consequent discoveries helped shape the design principles we presented in Chapter 1.

3.11.1 Designing for efficient writes

CANDStore is designed for read-heavy workloads, as the stark imbalance between read and write throughput of Optane precludes its usage for write-dominated workloads. As such, as initially designed CANDStore with reads in mind, ensuring that steady-state operation and the primary failure recovery protocol were both well optimized for reads. However, as we describe in this section, this turned out to be sub-optimal.

Initially, we leveraged the low latency and high throughput of NVMe SSDs to replicate keys onto the backup node SSD without any buffering or batching, and persisted metadata separately from keys and values. However, we quickly found that this caused two major problems: First, it severely hurt write throughput on the SSDs on the backup nodes as shown in table 3.9, and second, it increased median and tail latency of PUTs as the NVMe SSDs became a bottleneck. To solve this problem, we first grouped metadata with keys and values, then we introduced an Optane-backed write buffer on each backup node, allowing low latency replication of keys onto NVMM and high throughput writes to the SSD via batching.

In our initial implementation, BatchPulls were rate-limited based on the incoming client read and write workload, with the recovery primary reducing the rate of BatchPull requests in response to PriorityPulls and PUTs. PriorityPulls are small random reads to SSD, and as such their tail latency is strongly influenced by the IO load of said SSD; for a read-oriented system such as CANDStore, it seemed natural to rate-limit BatchPulls, which are large sequential reads on SSD, to reduce the tail latency of PriorityPulls. However, we found that limiting BatchPulls in response to PriorityPulls actually harmed TTR, as it is more important to quickly ingest new keys into the recovery primary than to reduce the tail latency of the backup node.

Conversely, we found that it was very important to ensure the recovery primary had enough write bandwidth to handle all client PUTs, as this directly affects the tail latency of the system and by extension the TTR, so it was necessary for us to design the system to balance the rate of BatchPull requests based on the client PUT frequency. Ultimately, a better approach would have been to rate limit BatchPull based only on client PUTs, then add rate limiting based on PriorityPulls only if the backup node tail latency became an issue.

3.11.2 Avoiding over-delivering on performance guarantees

CANDStore's heterogeneous node layout was devised by first setting explicit guarantees for consistency and durability, then working to meet these goals while minimizing cost. In particular, our use of the witness node is a prime example of meeting performance guarantees without exceeding them–the witness node is a critical part of the fast recovery protocol, and serves as the recovery primary in the event of primary replica failure. The witness node provides the bare minimum functionality with regards to consistency, detecting, but not resolving, consistency problems; as a result, the witness node is able to persistently store logical timestamps without data, greatly reducing the storage cost per node. The witness node is also well matched with our failure model, where triplication of data in conjunction with a metadata-only witness node meets the necessary level of fault-tolerance; by meeting, but not exceeding, these fault tolerance and consistency guarantees, CANDStore is able to reduce cost while still achieving high availability. In the event the witness is promoted to recovery primary, it detects non-present values by reading the logical timestamp associated with the key, and uses a PriorityPull to fetch the non-local value. CANDStore's frugality with persistent data is key to its cost-effectiveness and high performance, as spare write bandwidth and NVMM capacity can be used to allow more primaries, replicas, or witnesses for different shards to be colocated on the same physical server.

One of the key observations that made fast online recovery feasible in CANDStore was that availability can be defined in terms of SLOs–CANDStore can be considered available if it is meeting SLOs, and does not necessarily have to provide full non-degraded performance to do so. Datacenter deployments typically include some degree of performance headroom between the full non-degraded performance of the system and the minimum the amount of hardware necessary to meet the SLOs, and CANDStore leverages this headroom to quickly restore performance to the minimum levels to meet SLOs, after which it slowly restores the system to full non-degraded performance.

Creating leeway through fine-tuning of system guarantees is a technique that generalizes to systems that do not use Optane as primary storage. CANDStore was designed to meet the minimum requirements with regards to consistency and availability, and used the resulting performance headroom to enable low cost and overcome the fundamental limitations on recovery downtime. Other systems, such as flash-based caches, may benefit from reducing consistency or staleness guarantees to improve latency, cost, or throughput. We believe that the limitations of persistent storage technology will necessitate trade-offs to work within the performance constraints of the hardware, and hope the lessons we learned through CANDStore will serve to guide systems designers in optimizing the design of their system for their intended workloads.

3.11.3 Designing around temporal and spatial locality of client workloads

The primary challenge in designing CANDStore was that Optane DIMMs have significantly increased density and reduced write throughput when compared to DRAM; this created a situation where the current state of the art offline recovery protocol would have taken prohibitively long to repopulate a failed and replaced primary replica. Initially, we tried to approach this problem by modifying the protocol and the arrangement of nodes across physical servers, but quickly found that there was no easy workaround to these fundamental performance limitations of Optane.

After these failures, we took a step back and made the observation that typical client workloads

are expected to have a high degree of temporal locality. This observation, coupled with our observation that CANDStore's availability is defined by its ability to meet, but not necessarily exceed, SLOs, allowed us to design CANDStore to achieve high availability through fast online crash recovery, working around the write-throughput constraint of Optane by leveraging temporal locality of expected client workloads.

CANDStore's online recovery protocol leverages the observation that datacenter workloads are skewed, and as a result recovering "hot" data first enables the system to return to in-SLO operation quickly. To take advantage of this, CANDStore identifies popular keys during normal operation, then uses preemptive recovery to quickly BatchPull as much hot data as possible during the initial phase of recovery.

The effectiveness of this approach helped us realize that it was important to understand the expectations of the client workload, such as performance SLOs, as well as the characteristics of the workload itself, such as the workload skew.

Taking advantage of workload skew to improve TTR following node or system failures is a technique that is not limited to primary-backup replicated storage running on Optane NVMM. For example, popular keys identified during normal operation of the system can be prioritized when re-initializing a failed cache, when live-migrating a storage node, or when instantiating additional node instances to load-balance an overburdened storage deployment.

Chapter 4

RAIZN: Redundant Array of Independent Zoned Namespaces

In this section we describe our work implementing RAIZN, a system to provide RAID-like striping and parity coding of data for arrays of ZNS SSDs.

ZNS transfers 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 unexpected on-device garbage collection related performance fluctuations experienced by conventional SSDs [96].

We present the challenges presented by the ZNS interface, with a focus on how the stricter write semantics, and describe how we designed and implemented RAIZN to address these challenges while avoiding FTL garbage collection to achieve performance up to $14\times$ higher than other software RAID systems. We show that RAIZN is able to provide full expected performance from the aggregate device set, successfully addressing the key challenges from the ZNS interface. Importantly, RAIZN retains ZNS's opportunities for increased application performance, allowing higher-level software (e.g., F2FS or RocksDB) to control physical data placement and garbage collection. RAIZN achieves throughput and latency comparable to the equivalent Linux software RAID implementation running on conventional SSDs that use the same hardware platform. We show that RAIZN is able to maintain normal performance under certain workloads that cause conventional RAID to suffer as much as 90% reduced throughput due to device-level garbage collection.

4.1 Challenges of Zoned Device Arrays

The benefits of ZNS are not free, as the 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 [19]. Prior work compares ZNS to conventional SSDs, showing that in single-device applications [27], applications specialized to run on ZNS SSDs can achieve a 90% reduction in 99.99^{th}-percentile tail latency and $2\times$ higher write throughput compared to the equivalent workload on conventional SSDs. ZNS SSDs also require less on-device DRAM and

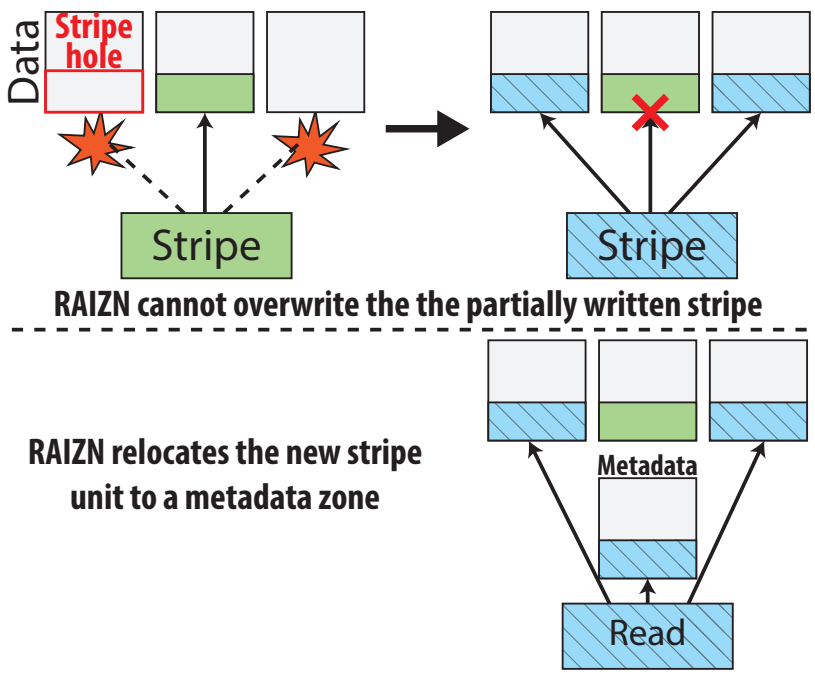


Figure 4.1: Only a subset of the stripe units are persisted before power is lost, resulting in a hole in the logical address space. The next stripe write cannot place its data at the correct address due to the written stripe unit.

overprovisioned flash than conventional SSDs.

There are two key challenges presented by Zoned device arrays: 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.

In this section, we describe how the zoned interface can be problematic for a conventional RAID-like setup. RAID organizes data into stripes consisting of stripe units and parity distributed across all devices, with stripe units mapped arithmetically to specific addresses on particular devices based on parameters provided when initializing RAID.

Metadata management. Unlike conventional RAID where 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, further adding complexity and necessitating garbage collection for metadata logs.

Parity updates. Writes that are not aligned to a stripe boundary, especially those which are smaller than a stripe, reduce performance in conventional RAID due to parity updates [57], 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 4.1 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 [88, 90].

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, with one exception as detailed in Section 4.3.2.

Write persistence. Forced unit access (FUA [21]) writes must be persisted before IO completion can be reported, with the implication that FUA written data will be readable after power loss and device failure. This poses a challenge for RAIZN due to a combination of the ZNS interface and independence of the devices in the underlying array; if a chunk of data is written to the RAIZN volume using FUA, it is not sufficient to simply propagate the FUA flag to the IO sent to the underlying device 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, this creates a situation where 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 without violating the ZNS specification.

4.2 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,

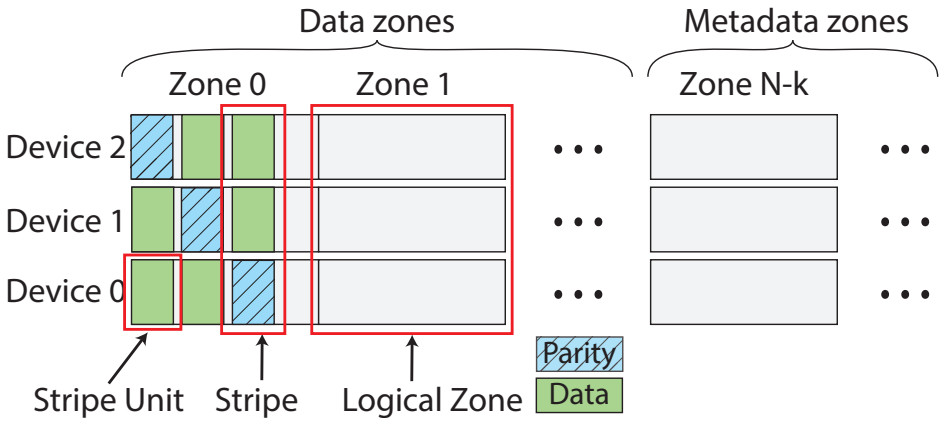


Figure 4.2: RAIZN data layout for three devices. Data is striped into two data stripe units with one parity stripe unit; the device holding the parity is rotated every stripe. Logical zones consist of one physical zone per array device.

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 4.3 we go into detail about how different metadata are used, persisted, and kept crash consistent.

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

Figure 4.2 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 4.2.3). 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 real 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.

4.2.2 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 [46]; 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 4.4.2. While the non-ZNS NVMe devices are technically capable of determining which LBA ranges are unwritten or deallocated [13], gathering this information is impractical as it would require checking for errors on every block address on the device.

4.2.3 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 $1077\,\mathrm{MiB}$, for a $1\,\mathrm{MiB}$ 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 4.3.1), 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 ($<100\,\mathrm{MiB}$), 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

Table 4.1: 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 + 64 KiB (stripe unit)	4 KiB (header) + 64 KiB (stripe unit)	4 KiB
Zone reset log	All devices	4 KiB	-
Generation counters	All devices	4 KiB	8.05 B per logical zone
Partial parity	Parity device	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 B per zone per device
Logical zone descriptors	-	-	64 B per logical zone

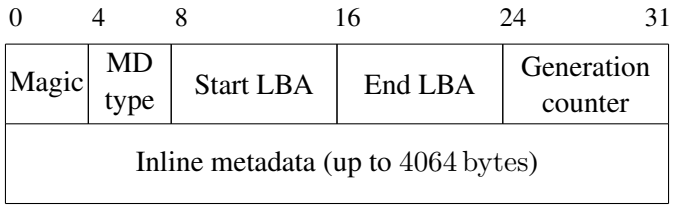


Figure 4.3: RAIZN metadata header layout when using 4 KiB sectors. Offsets are shown in bytes.

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 4.1.

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 4.3.1), 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. The precise layout is shown in Figure 4.3: 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 4.1). 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–$32\,\mathrm{bytes}$ of metadata header followed by 508 8-byte generation counters. When a generation counter is updated, the entire $4\,\mathrm{KiB}$ 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 4.3.2. 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 4.1). 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 4.3.3) 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

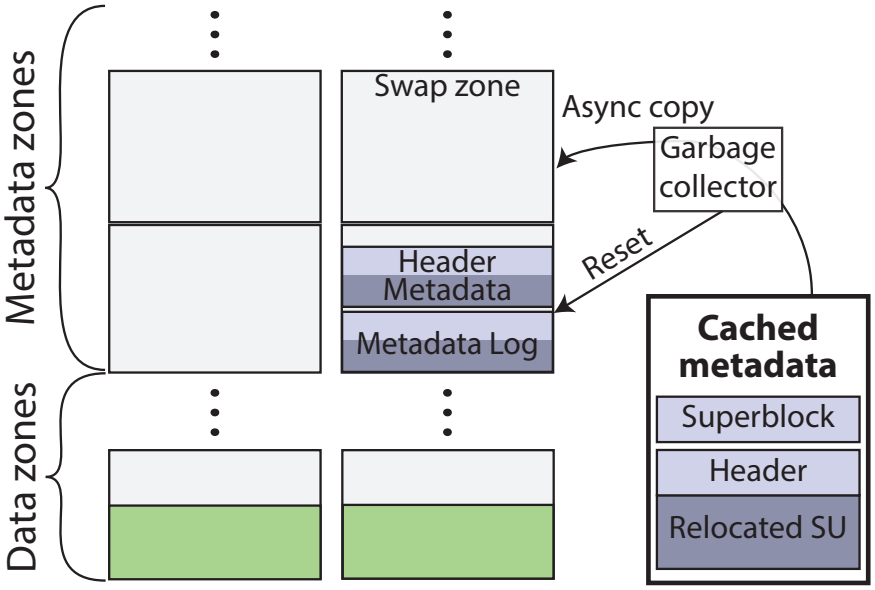


Figure 4.4: The garbage collector checkpoints metadata before resetting the old metadata zone. Each metadata entry has a header that includes (1) the type, (2) the applicable LBA range, and (3) the generation count of the logical zone.

in the metadata zones. The RAIZN garbage collector uses swap zones to facilitate garbage collection without interrupting operation, as illustrated in Figure 4.4. 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 is calculated by XOR'ing the contents of the stripe buffer of each open logical zone–this is elaborated on in Section 4.3.1.

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, then pushes each logical IO onto a single shared queue. After queueing the IO, a worker thread is scheduled 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 to ensure all necessary data is written before it is flushed.

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.

4.3 Solving ZNS challenges

In this section, we describe the solutions to the problems presented in Section 4.1, and present a brief explanation of how each of our solutions achieves crash consistency and fault tolerance.

4.3.1 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 4.5 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 [104] 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.

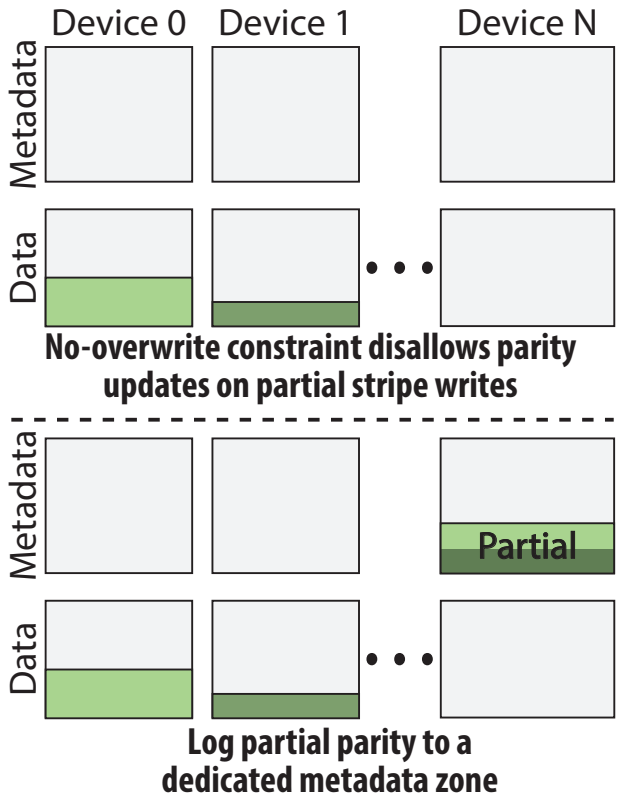


Figure 4.5: RAIZN writes parity for partially filled stripes to a metadata zone.

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.

4.3.2 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[110], where the parity and data can become de-synchronized, and second, torn writes [64]. 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 4.1). Figure 4.1 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 4.1), 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

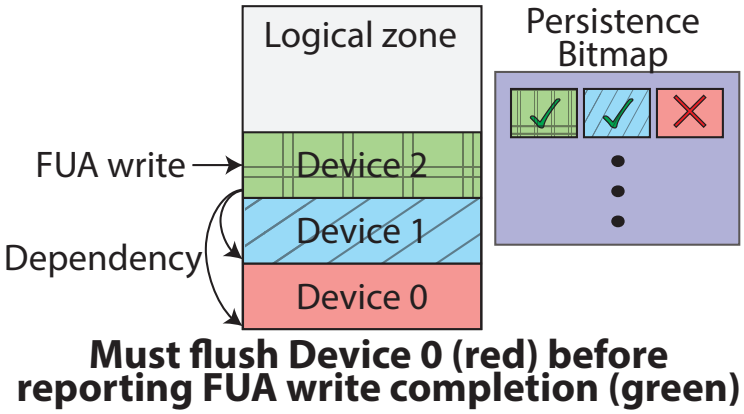


Figure 4.6: A FUA write in RAIZN must check that all LBAs preceding itself in the same logical zone are persisted before reporting completion. In this example, the FUA write (green, device 2) must explicitly flush device 0 (red) before reporting IO completion, because the persistence bitmap shows the stripe unit on physical device 0 is not persisted.

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.

4.3.3 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 [21]) write. Flush requests are duplicated submitted to each of the array devices (`REQ_OP_FLUSH`).

FUA writes and preflushed IOs, on the other hand, require special handling; RAIZN guarantees that if the user is notified of FUA or preflushed IO completion, the data persisted by that IO will be readable after power loss. In conventional RAID, the RAID-5 write hole is the primary challenge 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 4.6. RAIZN sets 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 $1\,\mathrm{bit}$ 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.

4.4 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, $128\,\mathrm{GiB}$ of DRAM, and Ubuntu 20.04 running on a $512\,\mathrm{GB}$ 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 $2\,\mathrm{GiB}$, but these changes do not affect performance.

For RAIZN experiments, we use 5 Western Digital Ultrastar DC ZN540 $2\,\mathrm{TB}$ 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 $1077\,\mathrm{MiB}$. Both RAIZN and `mdraid` are configured to run with 8 worker threads, the latter configured with the maximum possible stripe cache size of $128\,\mathrm{MiB}$. In all experiments, `mdraid` was configured to run without a journal volume, ensuring maximum performance.

4.4.1 Raw device microbenchmarks

We begin by evaluating the basic read and write performance of the ZNS and conventional SSDs using fio 3.28 [8], with libaio [9].

First, we write $1\,\mathrm{TiB}$ 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 [27]. The ZNS SSD's throughput is $1052\,\mathrm{MiB/s}$ for writes and $3265\,\mathrm{MiB/s}$ 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.

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 $8\,\mathrm{KiB}$ to $128\,\mathrm{KiB}$. The optimal stripe size is dependent on workload, so we present

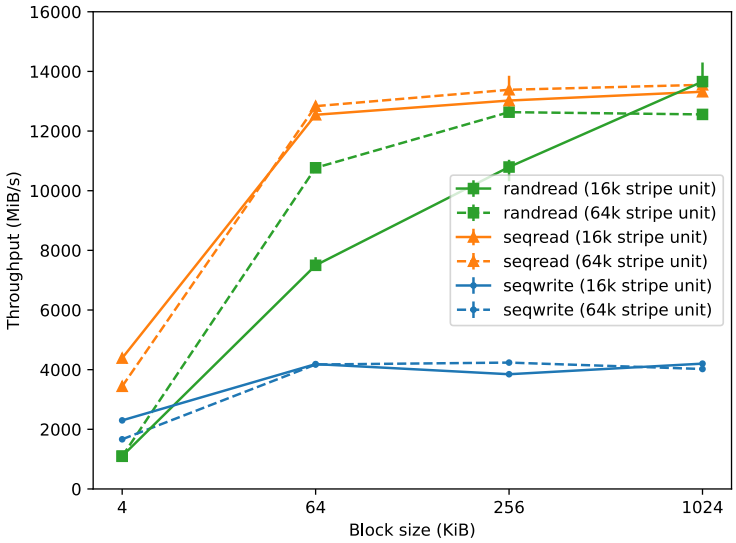


Figure 4.7: `mdraid` random read throughput is significantly higher with 64 KiB stripe units, but sequential reads and writes experience a small drop in throughput compared to 16 KiB stripe units.

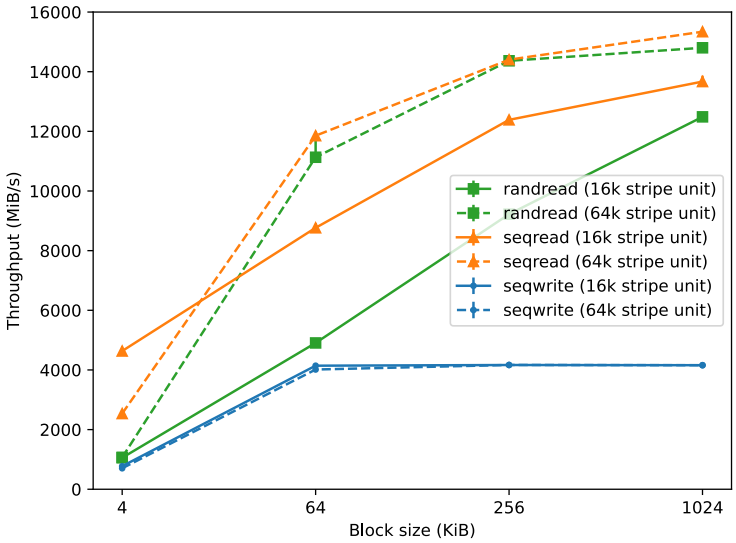


Figure 4.8: Other than 4 KiB sequential reads, RAIZN performs better with 64 KiB stripe units than 16 KiB stripe units. 4 KiB sequential read performance is less important than larger granularity sequential read performance.

throughput graphs (Figures 4.7 and 4.8), 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: $64\,\mathrm{KiB}$ *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 4.7 and 4.8 are the median of three trials; error bars denote the minimum and maximum, and each series represents `mdraid` or 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` or RAIZN volume beginning at different offsets in $\frac{1}{8}$ increments of the total data written to the volume; this results in each job reading a disjoint subset of the data present on the volume and the collection of all jobs reading the entirety of the data present on the volume. In our experiments, the volume is primed by writing $1024\,\mathrm{GiB}$ 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` or RAIZN volume is re-initialized before each trial of the write workload to prevent on-device garbage collection from affecting the results.

The random read workload is driven by 1 job with a queue depth of 256, randomly reading addresses within the $1024\,\mathrm{GiB}$ 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 4.7 shows that while using $16\,\mathrm{KiB}$ stripe units results in higher throughput for large block size reads, this is offset by the large reduction in random read throughput compared to $64\,\mathrm{KiB}$ stripe units. In Figure 4.8, RAIZN performs better with $64\,\mathrm{KiB}$ stripe units than $16\,\mathrm{KiB}$ stripe units on all workloads other than $4\,\mathrm{KiB}$ 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 $64\,\mathrm{KiB}$ 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 (*$4\,\mathrm{KiB}$–$64\,\mathrm{KiB}$*) reads and writes.*

Figure 4.9 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 99.9$^{\text{th}}$-percentile tail latency. The performance of RAIZN on small ($4\,\mathrm{KiB}$–$64\,\mathrm{KiB}$) sequential reads and writes lags behind that of `mdraid`; the low $4\,\mathrm{KiB}$ 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

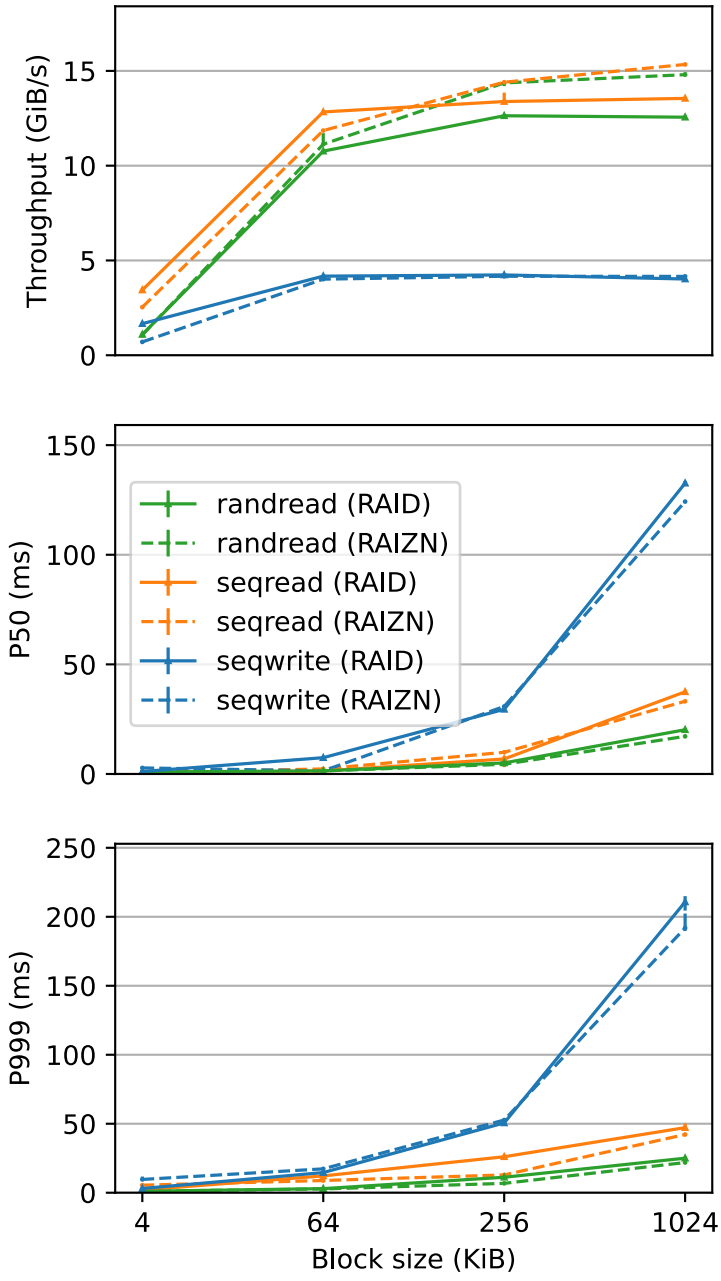


Figure 4.9: RAIZN achieves comparable throughput and tail latency to `mdraid`

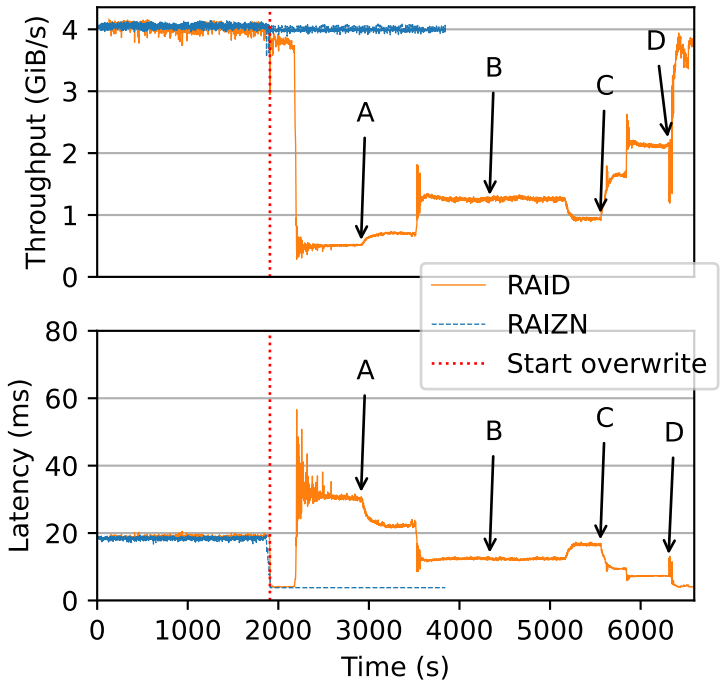


Figure 4.10: `mdraid` suffers when the SSDs run out of spare blocks and start performing garbage collection. RAIZN is able to maintain high throughput and low latency because ZNS SSDs do not perform on-device garbage collection.

draw focus to the strong performance of RAIZN when serving large (256 KiB–1 MiB) sequential reads.

For both median latency and 99.9$^{\text{th}}$-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.

Observation 3: *`mdraid` can experience unpredictable and severe performance degradation caused by on-device garbage collection. This is not an issue for RAIZN 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% $\rightarrow$ 20%, the second thread writes 20% $\rightarrow$ 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 4.10. 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 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 Ⓓ.

4.4.2 Performance during and after 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 4.11. All trials were performed with the same parameters as those in Figure 4.9, with the exception that after pre-filling the RAIZN/`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 ($4\,\mathrm{KiB}$) IOs and outperformed on larger IO sizes. This 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 $960\,\mathrm{GiB}$ SSDs of the same make and model instead of $2\,\mathrm{TB}$ SSDs. The $960\,\mathrm{GiB}$ devices perform similarly to the $2\,\mathrm{TB}$ devices. To ensure a fair comparison, the conventional SSDs are formatted with $969\,300\,\mathrm{MiB}$ (approx. $946\,\mathrm{GiB}$) capacity to match the usable capacity of the RAIZN volume, and the `mdraid` resync rate limit was set sufficiently high to not reduce `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 4.12 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

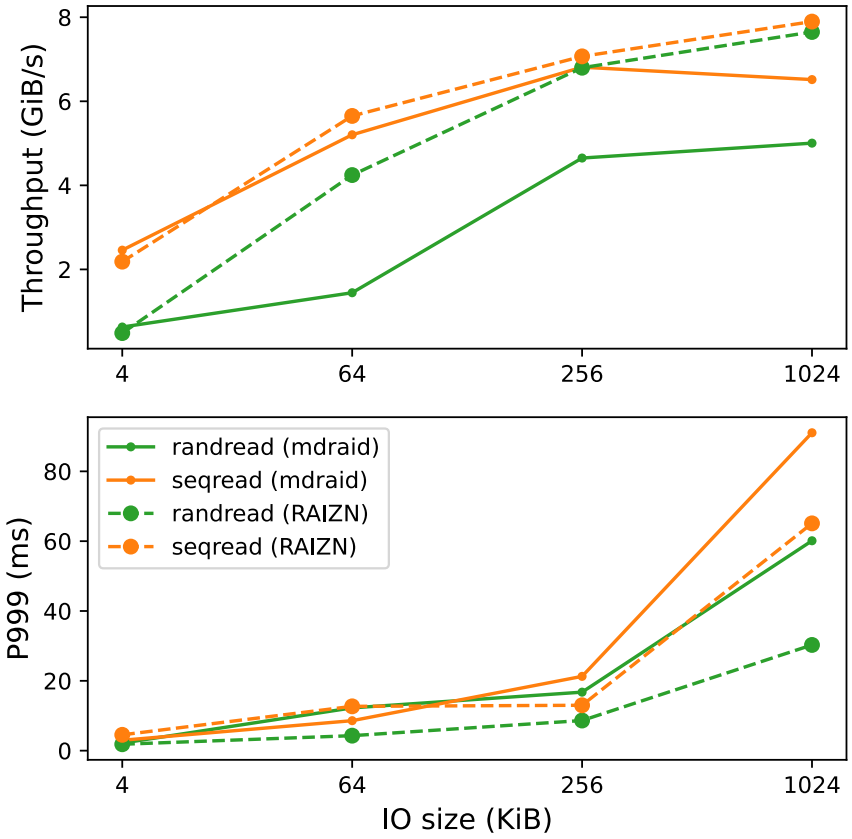


Figure 4.11: Degraded performance is similar between `mdraid` and RAIZN.

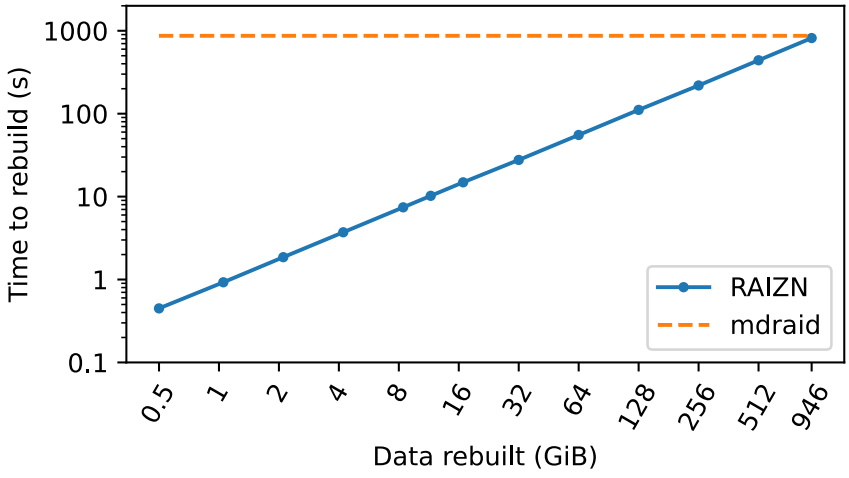


Figure 4.12: Time to repair (TTR) for rebuilding a device in RAIZN and `mdraid` varying the amount of valid data rebuilt from 64 GiB to approx 946 GiB. RAIZN's recovery time is bottlenecked by replacement device write throughput; TTR scales linearly with the amount of data rebuilt. When replacing a failed device, `mdraid` always rebuilds the entire address space, resulting in the same TTR regardless of the amount of user data present on the array.

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.

4.4.3 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 [43] benchmarks to compare the performance of RAIZN and `mdraid` on real applications, using the optimal array configurations derived from Section 4.4.1, 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 [50]. We ran the *fillseq*, *fillrandom*, *overwrite*, and *readwhilewriting* workloads using `db_bench` [44]. 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 4.13; the overall trend holds for other value sizes.

We also run sysbench [62] on MyRocks [16] using the RocksDB storage engine with MySQL [12] 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 pre-populates the database with 8 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 4.14 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.

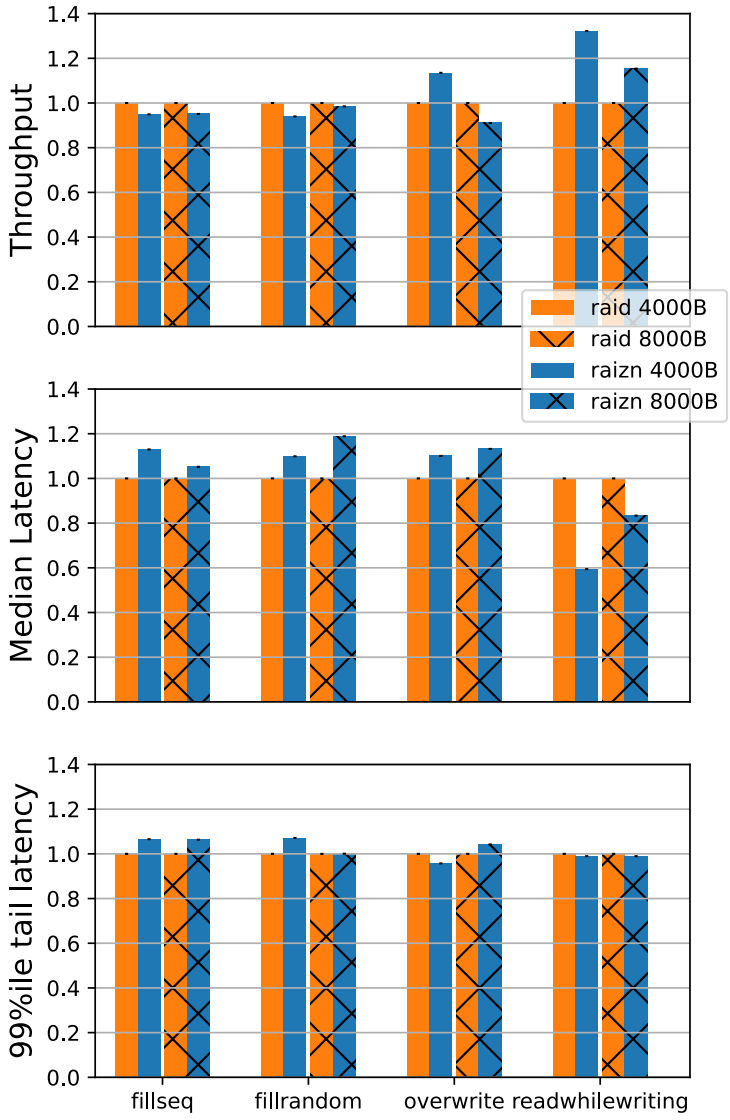


Figure 4.13: RocksDB performance, normalized. RAIZN achieves throughput and 99th percentile tail latency within 10% of `mdraid`.

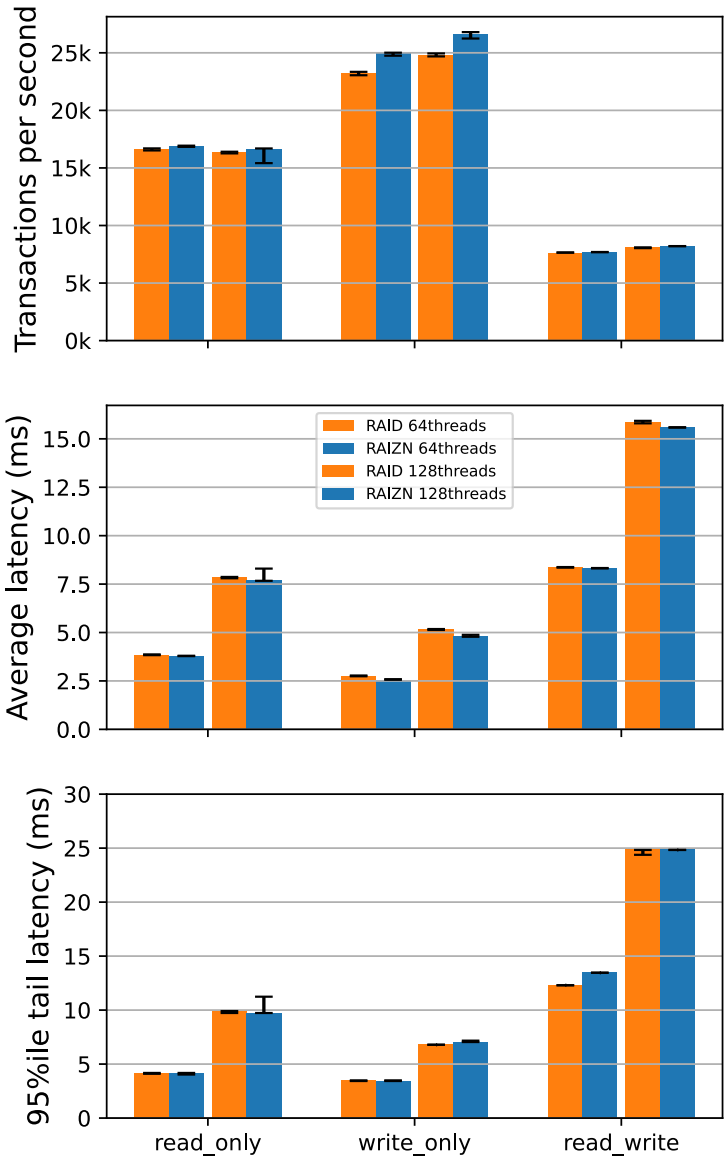


Figure 4.14: RAIZN achieves similar performance to `mdraid` in sysbench, both in 64 thread benchmarks (solid) and 128 thread benchmarks (checkered).

4.5 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 [59]. 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 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 [26]. 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, requiring specialized host software for each different SSD model. ZNS SSDs handle media management in firmware, creating a more appealing division of responsibilities between hardware manufacturers and their clients.

Application-Managed Flash allows users to directly access segments of flash composed of a fixed set of erase blocks [68]. 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 [87], but achieving space-efficient parity coding for small objects over an array of KVSSDs has proven difficult [74].

Flexible Data Placement (FDP)[17, 33], previously known as SmartFTL, minimizes write amplification when using SSDs as backing storage for cluster filesystems. As of the writing of this document, publicly available information about FDP is still scarce, and as such we defer any detailed exploration of FDP to future work. FDP SSDs operate, by default, identically to conventional SSDs, but optionally allow the host to specify a stream identifier when writing data. Data written with the same stream identifier are grouped by the device into *reclaim units* which can be erased as a single unit, enabling low write amplification if deletion of data within a given reclaim unit exhibits temporal locality. The stream identifiers are analogous to zones in ZNS, and the majority of RAIZN's mechanisms can be used as-is if running RAIZN on an FDP device. We expect there to be further opportunities for performance optimizations when running RAIZN on FDP devices, and hope to explore these optimizations in future work as the FDP specification and hardware become available.

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 [73] 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. Unlike SMORE, RAIZN exports a logical volume that provides the same semantics as a zoned device and addresses various challenges that are brought about as a result of conforming the external interface to the ZNS specification.

HiSMRfs [56] is a filesystem designed to achieve high performance on SMR drives by using a log structured filesystem for data while storing metadata in random write storage such as an 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 in conjunction with a conventional block device for metadata.

Host-side FTLs. To support unmodified applications atop ZNS SSDs, host-side FTLs such as dm-zap [6] 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 [77], pblk [26], and SPDK's FTL [75].

4.6 Lessons from RAIZN

This section outlines how we applied the lessons learned from CANDStore when designing RAIZN in addition to additional lessons learned from our various mistakes and failures experienced during the design, implementation, and evaluation of RAIZN. In conjunction with Section 3.11, these are the findings that shaped and defined the principles we laid out in Chapter 1.

4.6.1 Designing for efficient writes

The desire for efficient writes was a primary factor in deciding the data layout in RAIZN; RAIZN uses an arithmetic mapping between logical and physical block addresses, and maintains as much physical contiguity as possible between adjacent logical addresses. This data layout has multiple advantages: first, it requires little to no metadata to describe the mappings between logical to physical addresses, and second, it maximizes the size of each sub-io. Maximizing the size of each sub-io not only improves write throughput, but also improves read throughput, ensuring maximum performance for both writes and reads.

At the implementation level, a large amount of effort was put into optimizing RAIZN's write path compared to the read path. Particular care was taken in ensuring parity logs only included the portion of parity that changed as a result of the newly written data. For example, if a stripe was updated with a $4\,\mathrm{KiB}$ logical write, only up to $4\,\mathrm{KiB}$ of parity is changed as a result of this write, so the parity log entry corresponding to this logical write only includes the $4\,\mathrm{KiB}$ portion of the parity that was changed.

One of the challenges in RAIZN was dealing with the minimum write size of one block ($4\,\mathrm{KiB}$ in our configuration), as this posed a minimum size for the metadata header. As a result of this restriction, the initial RAIZN implementation suffered from poor performance on small logical writes, as the overhead for a single $4\,\mathrm{KiB}$ write could be as high as $16\,\mathrm{KiB}$. However, we were able to make efficient use of this space by merging metadata headers and inlining metadata wherever possible; for example, if a logical write generates metadata sub-ios for partial parity and generation counters, the headers are merged and the generation counters are inlined into the remaining space

in the metadata header. This optimization reduced the maximum overhead for a $4\,\mathrm{KiB}$ write from $16\,\mathrm{KiB}$ down to $8\,\mathrm{KiB}$.

In addition, RAIZN only writes metadata to persistent storage if it is necessary to ensure correctness. Each different type of metadata is replicated to the minimum set of devices in the array, ensuring that the overhead of writing metadata is minimized during normal operation of the system.

For example, RAIZN does not persist stripe buffers, persistence bitmaps, physical zone descriptors, or logical zone descriptors, as they can all be recomputed based on the state of each array device on volume remount. In addition, partial parity and remapped stripe units are only stored on the affected device, with the system designed to gracefully handle the loss of this metadata in the event of device failure.

Minimizing the amount of persisted metadata is important, as the amount of valid persistent metadata during metadata garbage collection is the primary contributing factor to write amplification in RAIZN. It is not possible for RAIZN to completely eliminate write overhead, but we were able to reduce metadata overhead as a percentage of total writes to below 0.1% on the workloads tested in our evaluation.

All of these small optimizations were crucial to the performance of RAIZN, as the write performance of our initial implementation lagged significantly behind `mdraid` due to the various metadata-related overheads on the write path. However, after making these changes, the write performance of RAIZN increased to match `mdraid` on the majority of workloads.

4.6.2 Avoiding over-delivering on performance guarantees

When designing and implementing RAIZN, it was important to clearly separate the handling of flushed and non-flushed writes to ensure the fulfillment of durability and persistence guarantees without over-zealously persisting data. Initially, RAIZN was designed very conservatively, with all writes flushed and serialized to ensure no inconsistencies in the event of unexpected power loss; however, this design proved to be too inefficient, achieving performance significantly lower than that of `mdraid`. It was clear that CANDStore needed finer-grained handling of flushed vs non-flushed writes, so we set the concrete guarantee that if the user is notified of write completion for a synchronous (flushed) IO, the corresponding data will be readable in the event of a power failure and/or single device failure.

Logical writes which are not explicitly flushed are handled with greater efficiency by providing weaker durability guarantees, an idea which shaped one of the major underlying principles behind RAIZN's design: transparently rollback writes when necessary to conform to the ZNS interface.

By carefully managing dependencies between stripe units with regards to ZNS semantics (Section 4.3.2), RAIZN is able to exactly fulfill durability guarantees for flushed writes while optimistically assuming non-flushed writes will persist before power loss. Both metadata and data are written without flushing in the common case, ensuring maximum write performance from the kernel block layer through the device layer.

4.6.3 Designing around temporal and spatial locality of client workloads

RAIZN takes advantage of spatial locality, or more precisely limitations on write order imposed by the ZNS interface, to reduce memory footprint, simplify metadata management, and expedite recovery after failure.

Due to the lack of overwrite semantics, RAIZN is able to maintain only a fixed number of stripe buffers per open logical zone. This allows RAIZN not only to have a predictable memory footprint, but a comparatively small memory footprint to conventional RAID; RAIZN achieves this by eliminating the "stripe cache", which is an in-memory cache used in conventional RAID to reduce read amplification caused by parity updates.

RAIZN also takes advantage of the sequential nature of the incoming write workload to implicitly invalidate certain types of metadata logs. For example, partial parity logs are ignored if the logical block address corresponding to the partial parity log is contained in a stripe that has already been fully written. This allows RAIZN to identify invalid metadata log entries without maintaining additional logical timestamps or other metadata.

One of the biggest ways in which RAIZN takes advantage of the restrictions the ZNS interface places on the incoming workload is by using the ZNS write pointer to precisely determine the minimum amount of data to reconstruct on a replaced array device.

Chapter 5

RAIZN+

In this section, we describe how we extended RAIZN to support two key features: (1) Zone append, and (2) Configurable logical zone capacity. For clarity, we term the original design of RAIZN (Section 4) as *RAIZN*, and RAIZN with the aforementioned extensions as *RAIZN+*.

Zone append (described in Section 2) is a ZNS-specific alternative to the traditional write command. When writing data using zone appends, the host specifies a zone number to append data to, and the device decides which address in the zone at which to write the data. While our specific device models do not have any significant performance differences between zone appends and zone writes, we expect future devices to have different performance characteristics and deem it important to extend the design of RAIZN to support zone appends.

Configurable logical zone capacity allows RAIZN+ to map fewer or more than one logical zone per physical zone, allowing logical zones to have capacity larger or smaller than the sum of the underlying physical zones. As mentioned in Section 4.2, logical zones in RAIZN consist of one physical zone per drive in the underlying array, and thus the logical zone capacity is equal to the number of data drives multiplied by the physical zone size of the SSDs. This inflexible mapping between logical and physical zones is a key weakness of RAIZN, as excessively large zone capacity can result in reduced end application performance due to the larger garbage collection unit size. In RAIZN+, this restriction is relaxed, allowing for multiple logical zones to span the same set of physical zones, or allowing a single logical zone to span multiple physical zones. This functionality is illustrated in Figure 5.1.

This section describes the challenges in implementing these two extensions in RAIZN+ (Sec-

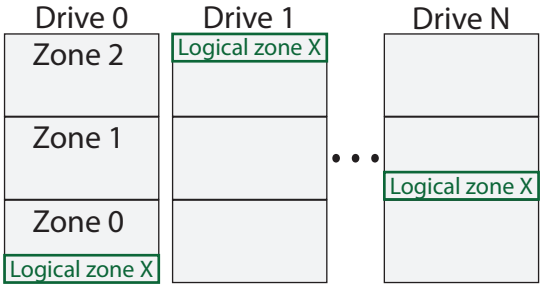


Figure 5.1: RAIZN+ allows zone size to be configured by the user, and multiple logical zones to coexist within a given physical zone.

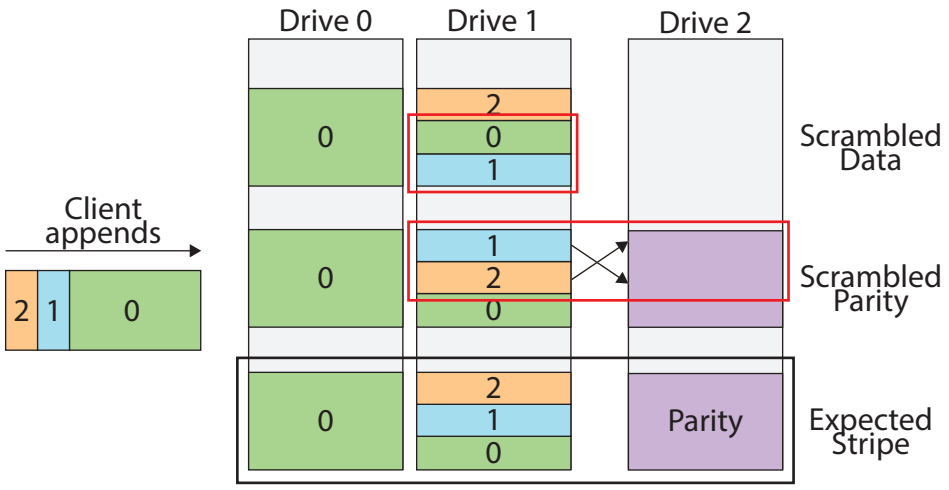


Figure 5.2: The client appends data chunks 0, 1, and 2, scrambled parity and scrambled data can occur due to append reordering.

tion 5.1), the design of RAIZN+ (Section 5.2), and an evaluation of RAIZN+ (Section 5.2.6).

5.1 Challenges

This subsection outlines the goals and challenges in implementing zone append support and configurable logical zone size support in RAIZN+ (Sections 5.1.1 and 5.1.2 respectively).

5.1.1 Zone append

Zone append is an alternate way to write to a ZNS device where the writer specifies a zone and the data to be written, and upon completion of the write the writer is notified of the exact LBA where the data was written [24]. For clarity, we refer to data written to a ZNS device by a conventional write command (i.e., `REQ_OP_WRITE` [1]) as being *written*, and data written by a zone append command as being *appended*. In addition, we refer to zone appends received by RAIZN+ from the user application (e.g., btrfs) as *logical* appends, and any zone appends submitted to the underlying ZNS drives as *physical* appends. We refer to physical appends as being *reordered* if they are not written in submission order.

Concurrent appends are not guaranteed to be written in submission order [36]–any appends that are submitted without either preflushing or waiting for the completion of previous appends can be reordered with respect to each other. This poses two main problems in the design of zone append support in RAIZN+. First, it is possible for physical appends within a stripe unit to be reordered, making it impossible to correctly calculate the parity before the data writes complete–we call this the *scrambled parity* problem. Second, if a logical append spans multiple stripe units, it is possible for the corresponding physical appends to be reordered in such a manner that the user's data is not contiguous in RAIZN+'s logical address space–we call this the *scrambled read* problem.

Figure 5.2 illustrates both of these problems. The user issues three logical appends, labeled 0, 1, and 2, in that order. Data chunk 0 is slightly larger than a stripe unit, and the three chunks

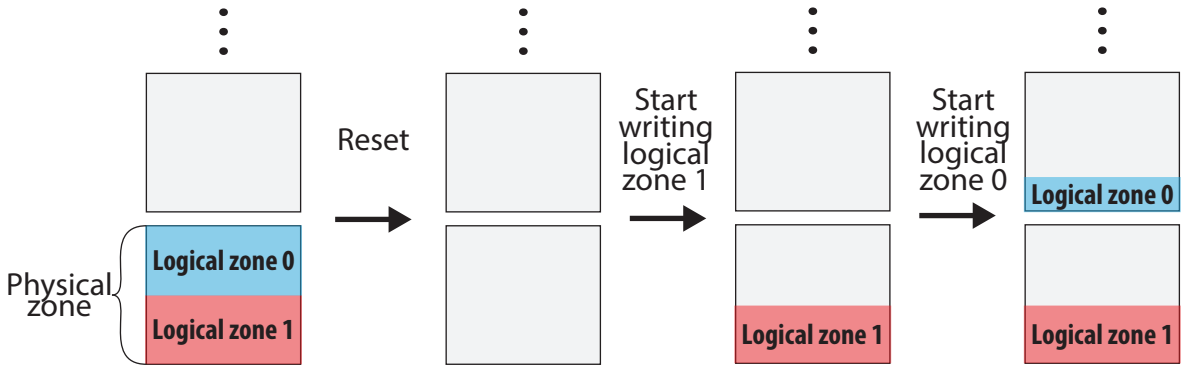


Figure 5.3: In RAIZN+, the physical address corresponding to a logical address can change every time the corresponding logical zone is reset and written to again.

together are exactly the size of a stripe. RAIZN+ splits chunk 0 into two physical appends, and submits all of the physical appends in order. In the typical case, illustrated at the bottom of the figure, the data would be written in the order it is submitted, and the physical layout would match the expected logical layout. If appends 1 and 2 are transposed, this results in a scrambled parity, where the computed parity does not reflect the true layout of the data chunks on disk. If append 0 on drive 1 is not written first, there is now a discontinuity between the first part of chunk 0 on disk 0 and the remainder of chunk 0 on disk 1, resulting in a scrambled read.

5.1.2 Configurable logical zone capacity

A key weakness of RAIZN is that logical zone capacity scales as a function of the stripe width (i.e., the number of drives in the array), which can hurt performance [2] and write amplification of ZNS-optimized applications as zones capacity increases. The large logical zone capacity in RAIZN is a result of the simple mapping of logical to physical zones, with logical zone K corresponding to the collection of physical zone K on each of the drives in the array, resulting in an N-drive RAIZN array having a logical zone capacity of $K \times (N-1)$. We call this logical zone capacity the *nominal zone capacity*. This simplistic design has the advantage that RAIZN corresponds closely to the expected behavior of ZNS devices: resetting a logical zone results in immediately resetting a predetermined set of physical zones, and never introduces any significant amount of write amplification. This restriction is lifted in RAIZN+, allowing for data from multiple logical zones to coexist within the same physical zone.

Zone indirection The location of data in physical address space corresponding with a given logical zone can change over time. For example, if a logical zone is written, reset, and written again, the physical address of the written stripe units can change between the first and second writes, as illustrated in Figure 5.3. In this example, logical zones 0 and 1 are initially located in the same physical zone. Then, both logical zone 0 and logical zone 1 are reset, allowing RAIZN+ to reset the physical zone. After this, logical zone 1 is written to, but not finished; then logical zone 0 is written to, but RAIZN+ cannot place data for logical zone 0 into the same physical zone as logical zone 1, because logical zone 1 is not finished. In addition, stripe units can be moved

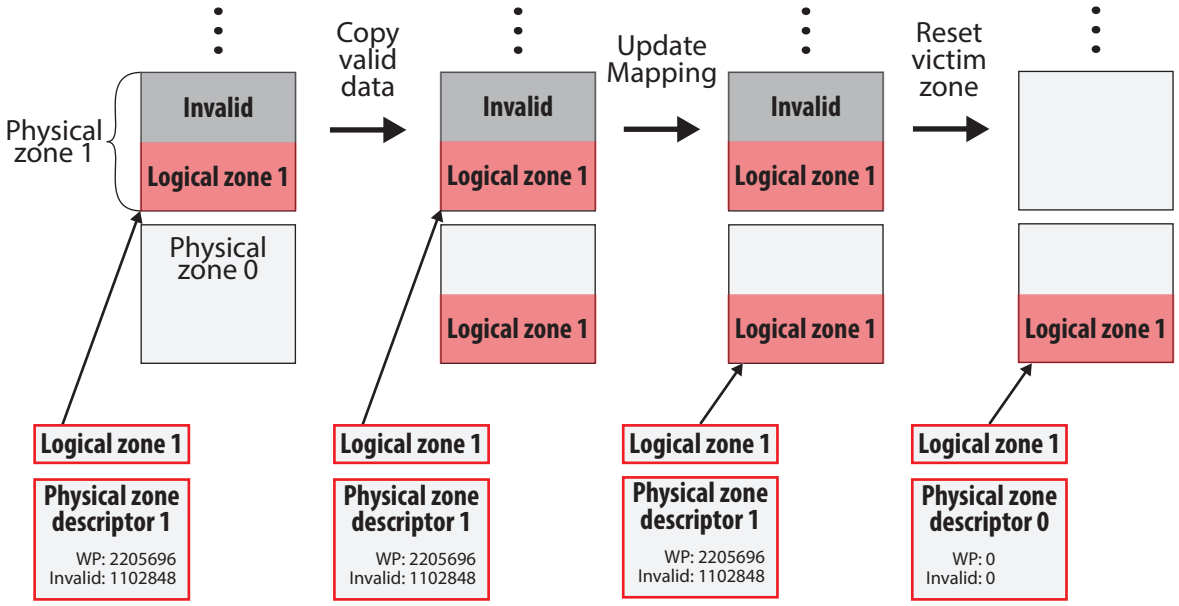


Figure 5.4: RAIZN+'s garbage collector copies valid data from the victim zone to a swap zone before resetting the victim zone.

to different physical zones as a result of garbage collection occurring within the RAIZN+ volume, as illustrated in Figure 5.4. In this example, RAIZN+ selects the physical zone holding data for logical zone 1 as a victim for garbage collection. The garbage collector first copies the valid blocks corresponding to logical zone 1 to an empty zone, then resets the victim physical zone. The end result is that the data for logical zone 1 is now stored at a different physical location.

Logical zone colocation Multiple logical zones can share the same underlying set of physical zones. This is illustrated in Figure 5.11, with logical zones 0 and 1 on device 0 occupying the same physical zone. In RAIZN+, logical zones are restricted to being a factor or multiple of the physical zone size of the underlying drives; this is done to simplify garbage collection. In addition, data for a given logical zone is stored contiguously in physical address space–a physical zone containing data from two different logical zones will have two *contiguous* address ranges each corresponding to a different logical zone, as illustrated in Figure 5.5. If a logical zone is reset under this scheme, the underlying physical zones cannot always immediately be reset due to the potential presence of data corresponding to other (non-reset) logical zones. Instead, the data corresponding to the reset zone is marked as invalid, and eventually the zone is garbage collected to reclaim the unused capacity–this process is explained in detail in Section 5.2.4.

Physical zone colocation On a given device, multiple physical zones can store data corresponding to the same underlying logical zone. This occurs only if the logical zone size is larger than the nominal zone size, and is trivial to handle unlike logical zone colocation. If the logical zone size is specified by the user to be larger than the nominal zone size, RAIZN+ falls back to the same behavior as RAIZN, maintaining an arithmetic mapping of logical to physical zones. Resetting a logical zone in this case results in resetting one or more physical zones, and does not require garbage collection under any circumstances.

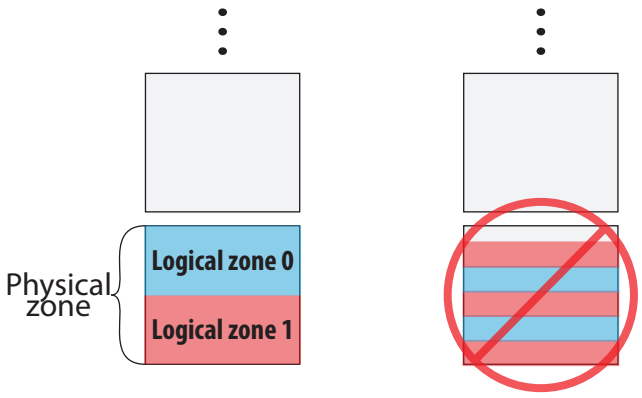


Figure 5.5: Data for logical zones is always contiguous in RAIZN+

In RAIZN+, we allow the user to configure a desired logical zone capacity to be exposed by the RAIZN+ volume; The primary challenge in implementing this is that of *zone fragmentation*, where logical zone colocation results in physical zones containing invalid data corresponding to logical zones that were reset. It is possible for the RAIZN+ volume to have a large amount of *logical* free capacity yet little to no *physical* free capacity remaining, due to the inability to reset a physical zone until all of the data in it is invalidated.

In addition to zone fragmentation, there are challenges posed when writing concurrently to different logical zones. Two concurrently open logical zones cannot naively share the same underlying physical zones, as writes from the two logical zones would get intermixed and necessitate a block-level mapping for logical to physical addresses–we call this problem *physical zone sharing*. This block-level mapping would consume large amounts of memory and storage capacity, and create significant overheads on reads and more importantly increase write amplification due to the per-write logical-to-physical mapping that must be persisted. As we mentioned in Chapter 1, storage systems should be designed to efficiently handle writes, making it imperative to avoid this block-level logical to physical address mapping.

5.2 RAIZN+ design

In this section, we describe how we designed RAIZN+ to support zone appends and configurable logical zone sizes while solving the challenges outlined in Section 5.1. RAIZN+ is designed to provide performance as close as possible to RAIZN while supporting these new features.

5.2.1 Zone append support

Supporting zone appends in RAIZN+ can be done in one of several ways, with the simplest solutions being to either translate logical appends into physical writes, or to synchronize appends within a zone (i.e., submit only one append at a time). Our final solution may involve translating large logical appends into physical writes, as the host-side overhead and throughput difference between appends and conventional writes is negligible for large IOs. Smaller IOs (e.g., 4KiB) will benefit from submitting zone appends to the underlying drives to achieve higher throughput, as

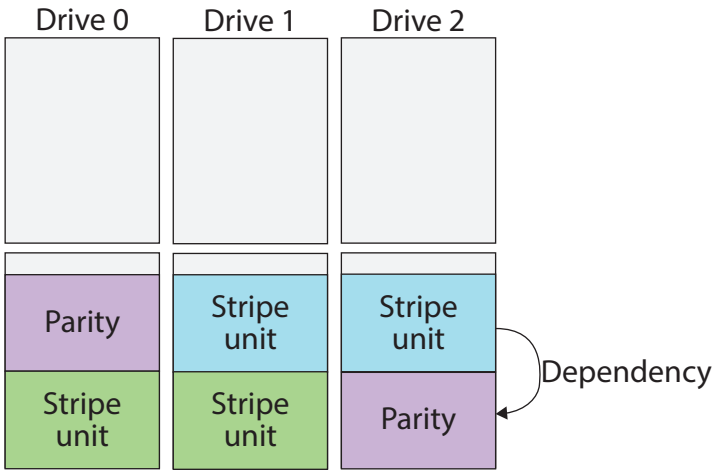


Figure 5.6: Delayed parity can result in stripe units having a dependency on parity from a previous stripe.

our testing has shown that for 4k writes to a single zone on a single ZNS SSD, zone appends can achieve a maximum of $2\times$ the throughput of conventional writes.

However, these two simple solutions fail to capture the performance benefit of using zone appends, as they both forego the ability to concurrently submit multiple appends to the same zone. As such, in RAIZN+ we aim to solve the *scrambled parity* and *scrambled read* problems while allowing at least some degree of concurrent appending. We make one key observation to guide our design: Append reordering is a relatively rare event, so the system should be optimized for the common case where reordering does not occur.

We do not discuss any solutions that involve block-level indirection due to the prohibitively high overhead, and thus require that all stripe units be contiguous in physical address space.

Scrambled parity We proposed two candidate solutions to the scrambled parity problem, which we term *delayed parity* and *optimistic parity*. Building on the design principles we outlined in Chapter 1, we chose to implement and evaluate optimistic parity over delayed parity, as the latter involves adding additional dependencies to the write path whereas the former only involves write overhead in the rare event of scrambled parity. Both delayed parity and optimistic parity are described below, but only optimistic parity is discussed in detail.

The first solution, *delayed parity*, is to simply wait until all data appends are complete before calculating and writing parity. This solution guarantees zero space overhead, no additional computation for degraded reads, and no performance degradation for asynchronous logical writes (i.e., non-FUA non-preflush). However, the parity write cannot start until the data append is complete, so synchronous writes will experience lower throughput and higher latency as they must wait for two rounds of writes to complete rather than just one. In addition, naively adding delayed parity to a RAID-5 layout results in each stripe write depending on the completion of the previous stripe write (illustrated in Figure 5.6)–to solve this problem, we introduce a layer of indirection just for parity (Figure 5.7) and reserve an open zone to handle parity writes. Adding an additional memory lookup for parity reads does not affect non-degraded operation, and the degraded-read performance overhead should be minimal in comparison to the other overheads (calculation and memory write)

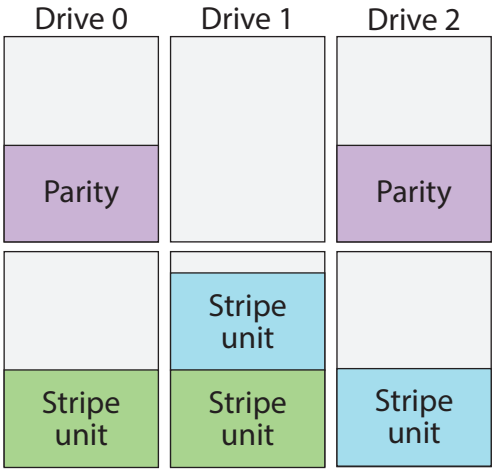


Figure 5.7: Solve the dependency issue in delayed parity by storing parity for multiple logical zones in a dedicated per-drive parity zone.

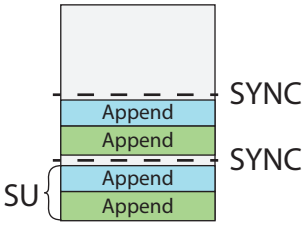


Figure 5.8: For the optimistic parity approach, RAIZN+ synchronizes appends at the stripe unit boundaries to eliminate cross-stripe unit reorderings.

involved during degraded operation.

The second solution, *optimistic parity*, is to optimistically write parity concurrently to physically appending data, and taking care to synchronize appends in a manner that prevents relocations into different stripe units (Figure 5.8). For asynchronous logical writes, the user is notified of write completion before reorderings are accounted for. If the data appends are reordered in any way, we write a metadata entry describing how the data was reordered; this metadata entry is called the *reorder record* (Table 5.1). If the append is synchronous, RAIZN+ waits for the reorder record to persist before notifying the host of append completion. Due to the rarity of append reordering, optimistic parity allows the majority of synchronous writes to complete after one round of writes, with a small fraction having to wait for an additional small (4 KiB) metadata write. The reorder record includes the start and end LBA of the data that was reordered, and the PBA where the data ended up being appended on-device.

Based on the principles we laid out in Chapter 1, we chose to implement optimistic parity. Both delayed parity and optimistic parity persist roughly the same amount of metadata, but the former introduces additional sub-io dependencies on the write path, potentially harming write performance. We implemented optimistic parity in RAIZN+, and as we demonstrate in Section 5.2.6, optimistic parity is able to provide high throughput with low latency. Both delayed parity and optimistic parity as described above assume synchronization of appends per-stripe unit–in other words, physical appends covering addresses spanning different stripe units cannot be issued concurrently.

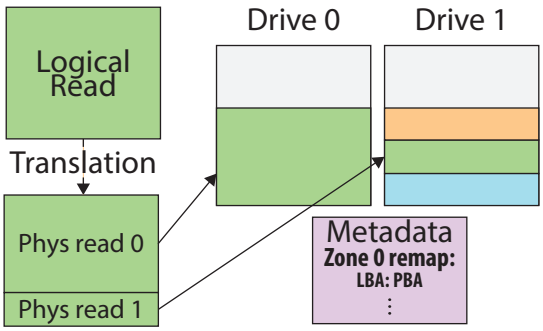


Figure 5.9: Scrambled data can be handled by splitting logical reads and maintaining records of any physically non-contiguous LBA ranges.

Scrambled data The simplest solution to scrambled data would be to synchronize any appends that have a possibility of getting scrambled (i.e., logical appends that cross a stripe unit boundary without filling the entire stripe unit).

However, this approach cannot take advantage of any performance advantages of concurrently issued zone appends, so instead we "unscramble" reads on demand–in the rare event that a series of appends is reordered in a way that results in a scrambled read, we use the reorder record to track how the read was scrambled, and "unscramble" any logical reads that cover the addresses affected by a scrambled read (Figure 5.9).

To avoid all reads incurring expensive metadata lookups, RAIZN+ flags any logical zone that experienced a scrambled read (allowing unflagged zones to bypass the unscrambling step), then uses a bitmap to concisely represent the LBA ranges that must be unscrambled on read (allowing unflagged LBA ranges to quickly skip the unscrambling step).

Scrambled data also poses a problem during power loss: if the system loses power before metadata describing the reodering is persisted, RAIZN+ is unable to detect or determine scrambled data on startup. Because unexpected power loss is rare, we solve this in a manner that potentially involves significant overhead whenever remounting RAIZN+.

First, when handling non-FUA appends, RAIZN+ periodically writes a metadata entry noting the LBA range that has been written without any scrambled data; this is implemented as a reorder record with a bit set in the metadata type field to indicate the lack of any reorderings. Multiple appends can be merged with regards to the reorder record–appends which are contiguous in address space can be represented by a single start and end LBA, and appends which are not contiguous can be sequentially included in the same reorder record to fill any remaining space in the inline metadata portion of the metadata header. This metadata entry can be piggybacked into the header of any other metadata write, such as partial parity, so for most real workloads it will incur no additional IO overhead. If the volume is unmounted, RAIZN+ flushes all outstanding IOs and then persists a metadata record describing all of the LBA ranges that were written without scrambled data. Upon remount, RAIZN+ discards any LBA ranges that are not explicitly recorded as being free of scrambled data, and these LBA ranges are handled as relocated stripe units in the same way as RAIZN.

Next, for FUA appends, RAIZN+ persists the reorder record before notifying the user of IO completion. This ensures that if the user is notified of a FUA append completion, they are guaranteed that all appends, up to and including the most recent FUA append, will be readable even

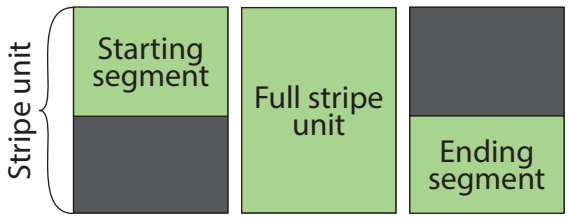


Figure 5.10: A logical append can be divided into up to three types of segments.

after power loss. In theory, this incurs an additional 4k metadata write for every FUA append, but in practice the majority of writes result in some partial parity being written, resulting in minimal added overhead if the reorder record is piggybacked onto the partial parity header.

In RAIZN+, we write these reorder records for zone writes in addition to appends–this allows RAIZN+ to provide atomic writes of arbitrary size. Any data that is present on mount but not accounted for by a reorder record is discarded, enabling RAIZN to detect and discard torn writes. This write atomicity allows RAIZN+ to provide similar benefits to `mdraid-journal`, but without the additional cost or bottleneck of a journal volume.

As an optimization, RAIZN+ can detect scrambled reads without waiting for all sub-ios for a given logical append to complete. This is made possible by the fact that scrambled data is detected by looking at discontinuities in the logical address space. A logical append can be divided into up to three types of physical appends: a *starting* segment, *full stripe unit*, or *ending* segment, illustrated in Figure 5.10. RAIZN+ detects a reordering if (a) the starting segment isn't written to the high order PBAs in the corresponding stripe unit, or (b) the ending segment isn't written to the low order PBAs in the corresponding stripe unit. As a result, RAIZN+ can detect a scrambled read not only when portions of a divided logical append complete, but also if a non-divided logical append to a stripe unit containing a divided logical append occupies the PBAs that should be occupied by a starting or ending segment. This optimization ensures RAIZN+ can issue metadata writes for reorder records as soon as possible, minimizing logical append latency.

5.2.2 Using physical appends for logical writes

The mechanisms described in Section 5.2.1 can be used to translate logical writes into physical appends. While there is no benefit to doing this on current devices (Section 5.2.6), we expect future devices to have different performance characteristics. RAIZN+ can be configured to translate writes smaller than a user-defined threshold to be translated into zone appends.

5.2.3 Zone indirection and configurable logical zone capacity

Configurable logical zone capacity larger than the nominal zone capacity is trivial, so we will not discuss it in detail. However, there are many challenges involved with supporting logical zone capacity smaller than the nominal zone capacity, as this requires placing data from multiple logical zones into a single physical zone. As described in Section 5.1.2, the primary challenges are zone fragmentation and physical zone sharing. To support configurable logical zone capacity, RAIZN+ must implement *zone indirection*, or the ability to decouple logical and physical zones through an indirection layer. In this section, we describe solutions to the aforementioned challenges, and

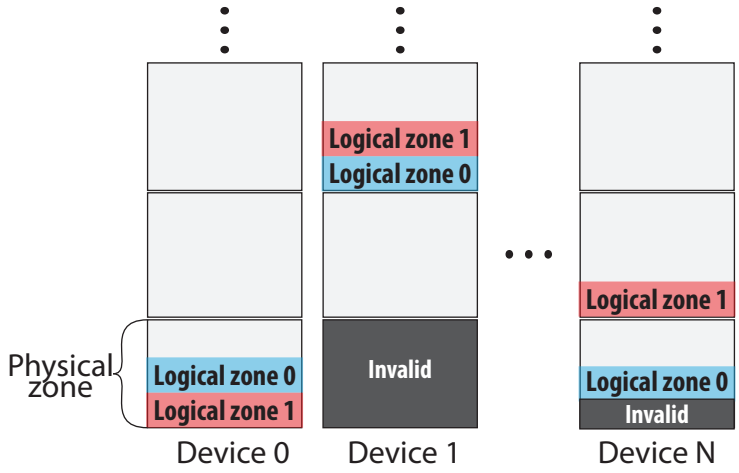


Figure 5.11: Data from different logical zones can reside in the same physical zone, but data corresponding to a given logical zone is always contiguous in physical address space. Logical zones 0 and 1 reside in the same physical zone, but each occupies a separate contiguous physical block address range within that physical zone. Data from logical zones which have been reset is shown as the gray "invalid" boxes in this diagram.

describe how RAIZN+ is designed to handle zone indirection without significantly impacting performance.

Each logical zone in RAIZN+ is striped across the underlying array devices as is illustrated in Figure 5.11; however, within a single physical zone, all data corresponding to a given logical zone is stored in a contiguous set of physical block addresses. This is done to allow RAIZN+ to maintain mappings on logical zone granularity as opposed to stripe unit or sector granularity, minimizing the DRAM overhead of RAIZN+. In addition to reducing the DRAM overhead, this coarse-grained placement provides a subtle benefit with regards to garbage collection, which is detailed in Section 5.2.4.

Table 5.1 describes the metadata, in addition to those described in Table 4.1, used by RAIZN+. For clarity, metadata not included in Table 4.1 is shown at the top of the table before the double horizontal line.

The primary consequence of sharing physical zones between multiple logical zones in this manner is that logical zone resets no longer correspond to a set of physical zone resets–instead, a logical zone reset invalidates some subset of the data in a set of physical zones, necessitating garbage collection to reclaim physical zone capacity. Consider the first physical zone on device N in Figure 5.11; the physical zone contains valid data from logical zone 0 along with invalid data from a previously-reset logical zone. RAIZN+ cannot reset physical zone 0 on device N, as that would result in the loss of data or parity for logical zone 0. Instead, RAIZN+ must garbage collect physical zone 0, copying the valid data to a different physical zone on device N before resetting physical zone 0 on device N. The following section (Section 5.2.4 describes two alternative designs to enable garbage collection for RAIZN+.

Table 5.1: Additional metadata used by RAIZN+

Metadata type	Persistent location	Storage per update	Memory footprint
Reorder record	Relevant device	4 KiB	12 B per reordering
Zone allocation record	Relevant device	4 KiB	6 B per logical zone per device
Remapped stripe unit	Affected device + 64 KiB (stripe unit)	4 KiB (header) + 64 KiB (stripe unit)	4 KiB
Zone reset log	All devices	4 KiB	-
Generation counters	All devices	4 KiB	8.05 B per logical zone
Partial parity	Parity device	4 KiB (header) + $\leq$64 KiB (stripe unit)	-
Superblock	All devices	4 KiB	4 KiB
Stripe buffers	-	-	320 KiB (5 stripe units) $\times$ 8 per open logical zone
Persistence bitmaps	-	-	2 KiB per logical zone
Physical zone descriptors	-	-	64 B per zone per device
Logical zone descriptors	-	-	64 B per logical zone

5.2.4 Garbage collection

RAIZN+'s garbage collector selects a victim device and victim physical zone, copies all the valid data to a replacement zone, then resets the victim physical zone. RAIZN+ schedules garbage collection on a given physical device if (1) the free physical capacity falls below a user-defined threshold, (2) one or more physical zones can be reset without incurring any write amplification, or (3) the user manually commands RAIZN+ to run garbage collection.

Every time a zone is opened or reset, the garbage collector evaluates whether garbage collection should be executed; if a logical zone is opened and there are not enough empty physical zones to hold the corresponding data, garbage collection is executed to free up capacity, and if a logical zone reset results in the entirety of a given physical zone becoming invalid, that physical zone is garbage collected (with no write amplification or performance penalty). The policy by which the garbage collector determines whether garbage collection is necessary or not is configurable, and consists of two parameters: the number of empty zones available, and the lowest-occupancy committed zone.

By default, RAIZN+'s garbage collector is configured to run when there is only 1 empty zone available, as having 0 empty zones can result in the system deadlocking due to the inability to garbage collect metadata. Similarly, RAIZN+'s garbage collector by default is configured to runs whenever the lowest-occupancy committed zone, that is the non-empty zone holding the least amount of valid data, in the system is at 0% occupancy. In the event there is only 1 empty zone available, and there are no non-empty zones at 0% occupancy, RAIZN+ chooses the non-empty zone with the lowest occupancy to garbage collect, breaking ties by choosing randomly.

Once garbage collection is scheduled, RAIZN+ selects the physical zone on the victim device with the least amount of valid data, copies the valid data from that zone into a different data zone, updates and persists the new logical to physical zone mappings, then resets the victim zone. This ensures data is never lost, even if power loss occurs during garbage collection, and ensures that

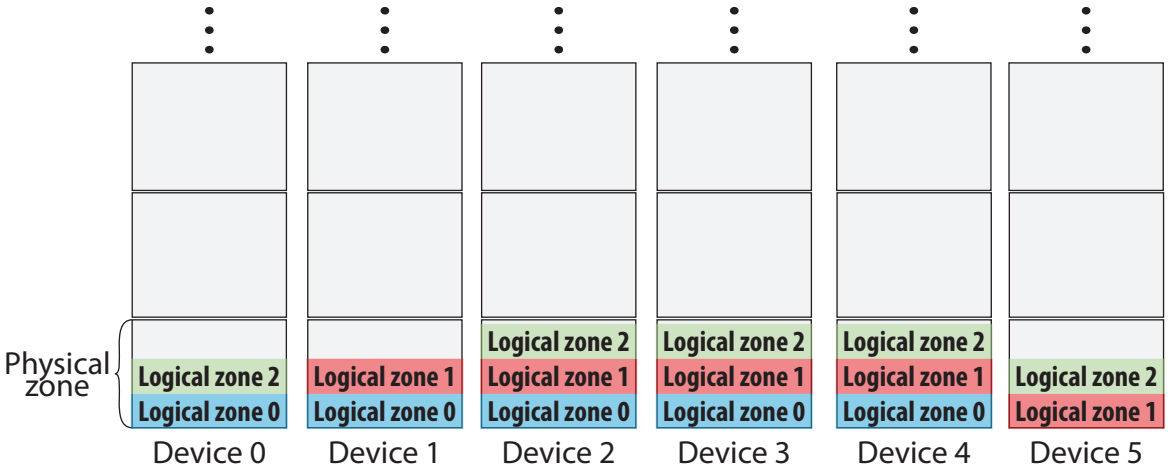


Figure 5.12: In a 6-drive RAIZN+DP array, each logical zone's data is striped into 4 data and 1 parity stripe unit and distributed across 5 devices. Similar to RAIZN, the device holding parity is rotated every stripe.

the logical to physical address mapping is valid at all times. RAIZN+ reserves some number of zones for metadata swap, which can be used to facilitate garbage collection in the event there are no physical zones remaining to serve as the replacement zone.

The architecture of RAIZN+'s zone manager and garbage collector is illustrated in Figure 5.4, which shows the process of garbage collecting a physical zone. Each physical zone descriptor stores information about the corresponding physical zone including, among other metadata, the write pointer and number of sectors that have been invalidated as a result of logical zone reset. Every time a logical zone is reset, the corresponding physical zone descriptors are updated to reflect the invalidated sectors, and if the number of invalid sectors matches the write pointer of a finished physical zone, garbage collection is scheduled. In this example, garbage collection is scheduled as a result of a lack of available physical zones to serve a client write, so physical zone 1 is scheduled to be garbage collected despite the fact that it contains valid data corresponding to logical zone 1. First, all valid data from logical zone 1 is copied from physical zone 1 to physical zone 0, shown in the second column of Figure 5.4. Next, the logical zone descriptor is updated to point at the new location of the data, physical zone 0, shown in the third column. In the final column, physical zone 1 is reset, resulting in an increase from 1 to 1.5 physical zones worth of capacity available to serve client writes.

5.2.5 Improving Garbage Collection with RAIZN+DP

A key weakness of the default garbage collector in RAIZN+ is that some amount of read and write throughput is used by the garbage collector to perform garbage collection, potentially resulting in lower read and write throughput for the client workload. To solve this problem, we design RAIZN+DP, which stands for declustered parity. RAIZN+DP introduces an additional device to the array, providing additional IOPS headroom to facilitate garbage collection. RAIZN+DP is able to provide similar fault tolerance, availability, and write atomicity as `mdraid-journal`, and is able to achieve higher performance than both RAIZN+ and `mdraid-journal`.

We introduce RAIZN+DP in the context of a 6-device array, and later generalize it to larger arrays. RAIZN+DP uses declustered parity [51] to organize 4 data stripe units and 1 parity stripe unit across 6 devices. For each logical zone L, RAIZN+DP chooses 5 of the 6 devices to hold data associated with L, mapping L to a set of physical zones $P_0, P_1, ..., P_5$ with one physical zone from each of the 5 chosen devices. Data is striped and parity is written in a similar manner to RAIZN, ignoring the the 6th not-chosen device. The distribution of logical zones across physical zones is illustrated in Figure 5.12.

This design enables RAIZN+DP to provide maximum write throughput equivalent to 5 individual devices, maximum read throughput equivalent to 6 individual devices, and resilience against one device failure. This is higher than `mdraid-journal`'s maximum write throughput equivalent to 4 individual devices, maximum read throughput equivalent to 5 individual devices, and resilience against one device failure. A portion of the write and read bandwidth is used to facilitate garbage collection in RAIZN+DP, but under normal conditions the performance of RAIZN+DP will not drop below that of `mdraid-journal`.

Similar to RAIZN+, RAIZN+DP performs garbage collection by selecting a victim device and victim physical zone, copying all of the valid data from the victim physical zone to a replacement physical zone, then resetting the victim physical zone. The key difference is that RAIZN+DP leverages additional write bandwidth headroom to reduce the performance penalty of garbage collection. All data in the replacement physical zone is persisted before the victim physical zone is reset, ensuring there is no window of vulnerability for data loss during garbage collection.

Worst case bottlenecks Certain workloads will result in performance degradation below 4 drive write throughput in RAIZN+DP–the characteristics of these workloads and the worst case performance degradation is as follows. Suppose the average write throughput of a client workload is given to be T_c, with the aggregate write throughput of all devices in the array adding up to T_a. The average occupancy of physical zones selected for garbage collection (i.e., the fraction of physical zone capacity occupied by valid data when garbage collection is invoked on that physical zone) is given to be $V, 0 \leq V < 1$ and the physical zone capacity is given to be P. Note that V must be less than 1, as there is no reason to garbage collect a physical zone that only contains valid data.

If $\frac{(T_a - T_c)}{V} < T_c$, RAIZN+DP will eventually run out of empty physical zones and be forced to reduce client throughput to enable garbage collection. In this scenario, the throughput of the client workload will be reduced as some of that write bandwidth will be used to facilitate garbage collection. The total write bandwidth needed by the garbage collector (T_{gc}) and the new lower throughput of the client workload (T_c') are related in the following way: $\frac{T_{gc}}{V} = T_c'$.

To put this in a concrete example, given a 6-drive RAIZN+DP array, and a client workload that involves writing at 3 drives worth of throughput ($4 \times T_p$), performance degradation will begin when $\frac{2 \times T_p}{V} < 4 \times T_p \implies V > \frac{1}{2}$. Intuitively, if garbage collection uses 2 drives worth of write throughput to copy $\frac{1}{2}$ of each physical zone, it can free up 4 physical zones in the time the client workload generates 4 physical zones worth of data. If garbage collection must copy $\frac{3}{4}$ of each physical zone, it can only free up 1.5 physical zones in the time the client workload generates 4 physical zones worth of data.

However, we argue that these worst-case workloads are rare, for the following reasons.

First, a primary concern of flash is the write endurance, and datacenter applications are often designed to limit the total drive writes per day (DWPD) of SSDs. For example, the ZNS drives

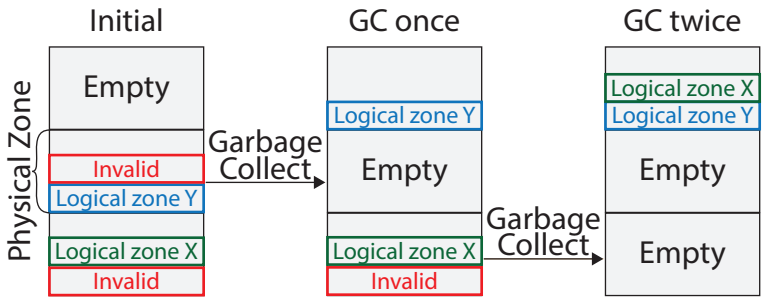


Figure 5.13: Data zones must be garbage collected to solve *zone fragmentation*. The left column shows the state of physical zones on a drive before garbage collection, where logical zones Y and X are both located in physical zones containing invalidated data. The middle column illustrates the state of the same physical zones on the same devices after one garbage collection pass, and the right column shows the state after a second garbage collection pass.

we use are rated for approximately 3.5 DWPD to ensure a 5-year lifespan [3], translating to approximately 42 MiB/s per terabyte average write throughput. For our 2TB devices, this translates to approximately 9% of the maximum sustained write throughput of the SSD, allowing RAIZN+ to provide high read throughput and serve bursty writes without introducing any unpredictable performance degradation.

Second, many flash-friendly systems organize data on SSDs in log-structured format–for example, f2fs typically allocates and resets zones in a manner that closely resembles round-robin. Under these conditions, it is likely that zone occupancy will be low and garbage collection overhead will be correspondingly low. We demonstrate this to be the case in Section 5.2.8, with an experiment showing how the number of non-empty physical zones peaks and plateaus when running a RocksDB workload on a RAIZN+ volume, indicating that zone occupancy is low.

Third, pressure to garbage collect zones is likely to be fairly low in many real-world situations, as many deployments do not involve high capacity utilization of flash [40]. Lower capacity utilization in RAIZN+ results in lower garbage collection pressure, as unused capacity can be used to serve incoming writes; this ultimately would result in more leeway to wait until more logical zones are invalidated before running garbage collection, thus resulting in lower average zone occupancy and by extension lower write amplification.

Zone manager To facilitate the dynamic mapping of logical to physical zones, we introduce the *zone manager*, a data structure that allocates physical zones to logical zones and tracks those logical to physical zone mappings for both data and metadata zones. Unlike RAIZN, the amount of permanent metadata required for RAIZN+ scales based on the size of the array. The garbage collector in RAIZN+ identifies victim physical zones based on occupancy and age, requests a replacement physical zone from the zone manager, copies valid data from the victim zone to the replacement zone, then resets the victim zone.

The zone manager allocates metadata zones in increments of whole physical zones, regardless of the logical zone size–this precludes the possibility of metadata and data being colocated in the same physical zone.

When a new logical zone is opened, the zone manager arbitrarily chooses a free physical zone from each device to hold the data for this logical zone. The zone manager maintains a mapping

between the logical zone and each of these physical zones, as well as a reverse mapping between each physical zone and the logical zones it contains data for. Physical zones that have at least enough free blocks to hold the associated stripe units for one full logical zone, and that are not currently "locked" by a different logical zone, are considered *free*. If a physical zone does not have enough free capacity left to hold a full logical zone, it is finished and the remaining unused capacity, if any, is marked as invalid for garbage collection purposes. The mapping between logical to physical zones is persisted inline with the generation counters, resulting in no additional write amplification or overhead for zone indirection compared to standard RAIZN.

5.2.6 Evaluation

We evaluated the components of RAIZN+ that were affected by the addition of zone append support and zone indirection using microbenchmarks and application benchmarks similar to those used when evaluating RAIZN, and compare the performance of RAIZN+ to `mdraid` level 5 and RAIZN.

When evaluating RAIZN+DP, we compare the performance of 6-drive RAIZN+DP to that of 5-drive `mdraid`–this is intended to simulate the performance of `mdraid-journal` with a theoretical journal device that provides sufficient performance to avoid becoming a bottleneck, as using a 6th SSD as the journal device would reduce write performance by at least 75% and read performance by 80%. RAIZN+ provides atomicity guarantees similar to those provided by `mdraid-journal`, and as such it is fair to compare the performance of 6 drive RAIZN+DP with that of 5(+1) drive `mdraid-journal`.

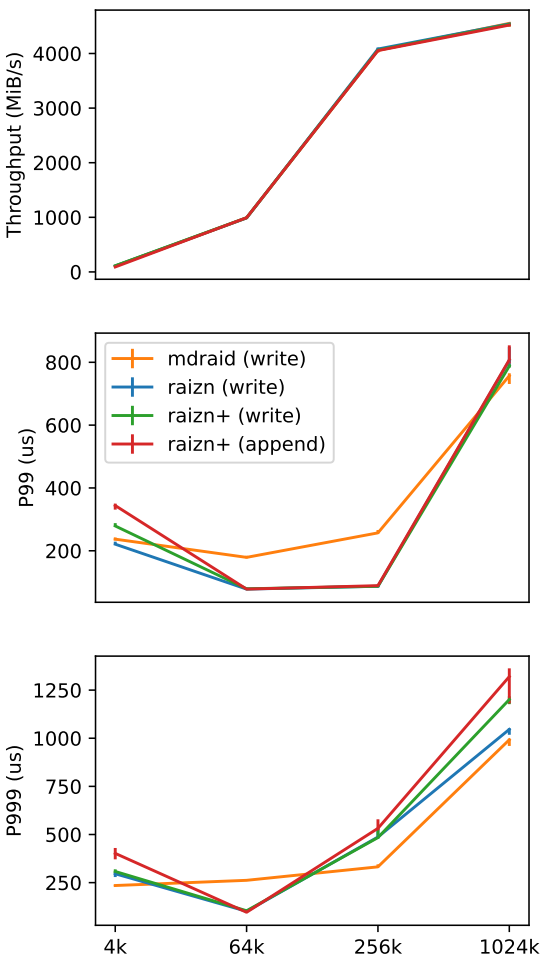
All experiments were run on a Dell R7515 server with a 16-core AMD EPYC 7131P CPU, 128 GiB of DRAM, and Ubuntu 20.04 running on a 512 GB conventional SSD. We modified the Linux kernel 5.19 and mkfs.f2fs to remove hard-coded constraints that prevent the creation of (logical) devices with zone sizes larger than 2 GB, but these changes do not affect performance.

For RAIZN and RAIZN+ trials, we use up to 6 Western Digital Ultrastar DC ZN540 960 GB 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.

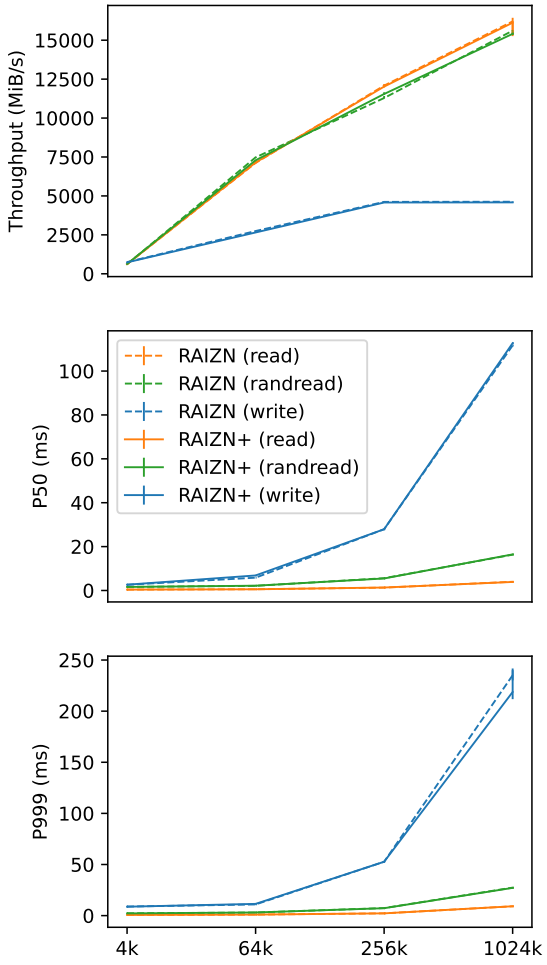
This section is organized in the following manner: Section 5.2.7 details the overhead associated with supporting zone appends and write atomicity, Section 5.2.8 explores the performance of RAIZN+ with zone indirection and multiple logical zones per physical zone, and Section 5.2.9 shows the effect of garbage collection in RAIZN+ and RAIZN+DP in the context of an adversarial workload that causes significant garbage collection related performance degradation in RAIZN+.

5.2.7 Zone append and write atomicity overhead

First, we evaluated the overhead of supporting zone appends and the necessary write atomicity (both for appends and writes). Due to the current lack of a user-space interface for submitting appends to zoned devices, a lack of zone append support in fio, and issues in btrfs, these experiments were run using a custom kernel-space benchmark tool written for the purposes of these experiments. All trials were run using a queue depth of 1 with a single thread submitting IOs to the



(a) Optimistic parity does not significantly affect the performance of RAIZN+. Throughput is unaffected, and tail latency reflects single device performance.



(b) RAIZN and RAIZN+ perform similarly (within error) on all tested workloads, indicating that the overhead introduced by zone indirection is negligible.

Figure 5.14: Overhead of zone append and zone indirection support in RAIZN+

mdraid, RAIZN, or RAIZN+ volume for 600 seconds, recording the submission and completion time for each IO completed during this entire period. After 600 seconds, the average throughput, 99^{th} percentile and 99.9^{th} percentile tail latency were calculated based on the number of IOs completed and the end-to-end latency of each IO. Due to the queue depth of 1 with a single thread, this tool is very sensitive to latency fluctuations, and is thus able to quantify the overheads associated with writing reorder records–running with a higher queue depth masks many of these tail latency effects as shown in Section 5.2.8.

Figure 5.14a shows the throughput and tail latency of mdraid, RAIZN, RAIZN+ serving logical writes, and RAIZN+ serving logical appends. The throughput of all four configurations is near-identical, showing that the amortized overhead of persisting reorder records is negligible. The 99^{th} percentile tail latency shows some slightly increased latency for RAIZN+ compared to mdraid and RAIZN for small ($4\,\mathrm{KiB}$) writes and appends–this is caused by the periodic additional metadata writes necessitated by the reorder records. The 99.9^{th} percentile tail latency is higher for RAIZN+ than mdraid and RAIZN for the same reason, with a slightly more pronounced difference due to the compounding tail latency effects of adding more sub-io dependencies to certain IOs.

5.2.8 Indirection overhead

Next, we examined the overhead associated with zone indirection and configurable zone size on top of supporting write atomicity. RAIZN+ was configured to use a logical zone size of $2154\,\mathrm{MiB}$, corresponding to two logical zones per physical zone, and 14 swap zones to facilitate garbage collection; for the experiments in this section, garbage collection was not an issue so the number of swap zones was inconsequential. We then ran two experiments–first, we ran raw-device write, read, and random read workloads (identical to Figure 4.9). Second, we ran a suite of RocksDB benchmarks with parameters identical to those used in Figure 4.13.

Figure 5.14b shows the performance of the raw device benchmarks on RAIZN+ compared to RAIZN. This experiment clearly shows that RAIZN+ achieves throughput and tail latency near-identical to RAIZN despite the additional overhead of zone indirection.

Figure 5.15 shows the performance of the raw device benchmarks on RAIZN+ compared to mdraid. RAIZN+ achieves similar performance to RAIZN, and performs roughly equal or better than mdraid on the tested workloads, showing that RAIZN+ is able to handle real application workloads without issue. An interesting finding is that garbage collection within RAIZN+ doesn't activate at all during the RocksDB experiments, a phenomenon we will explore further in Section 5.2.9. To illustrate why this is the case, we ran an additional experiment, shown in Figure 5.16, with a relatively small ($240\,\mathrm{GiB}$) RocksDB database running on a RAIZN+ volume configured to only garbage collect physical zones containing 0 valid blocks. In this additional experiment, we first ran the fillsequential workload, followed by the overwrite workload, and sampled the number of non-empty or "committed" zones within RAIZN+. A zone in RAIZN+ is considered committed if it contains any data, whether valid or invalid. Figure 5.16 shows that despite over $800\,\mathrm{GiB}$ of data written to the RAIZN+ volume by RocksDB, the total "committed" capacity within RAIZN+ plateaus at around $400\,\mathrm{GiB}$. This is due to the nature of the RocksDB workload and the way F2FS lays out data on SSD; F2FS writes segments to zones in sequence, then garbage collects old segments as data is overwritten and invalidated. This results in a pattern where despite more and more data being written to the RAIZN+ volume, logical zones are reset in a fashion that allows RAIZN+

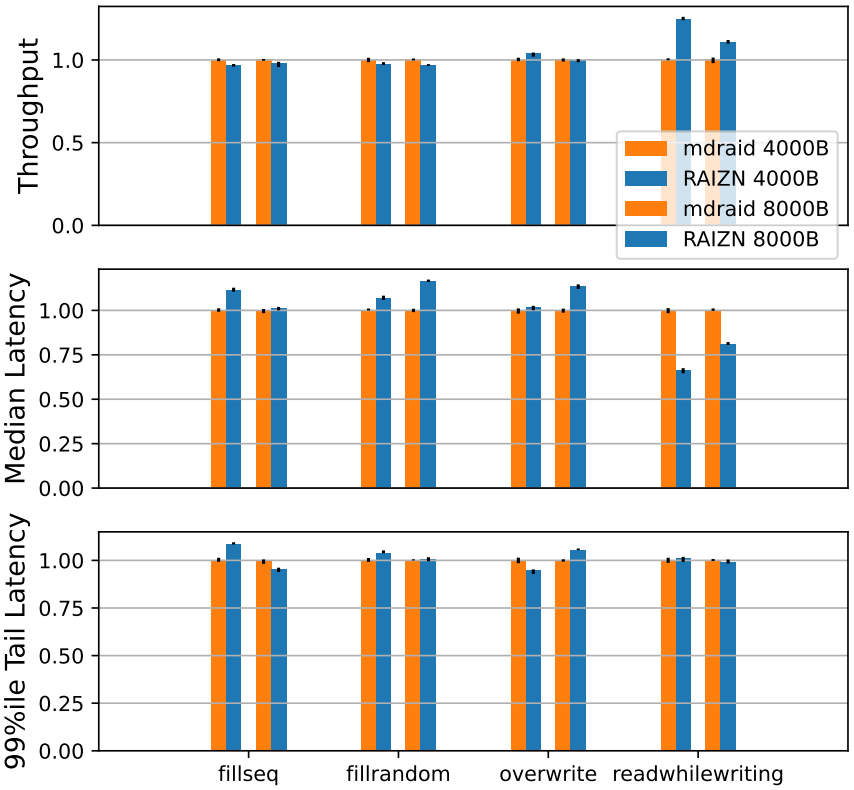


Figure 5.15: RAIZN+ achieves similar or better performance on RocksDB benchmarks compared to `mdraid`.

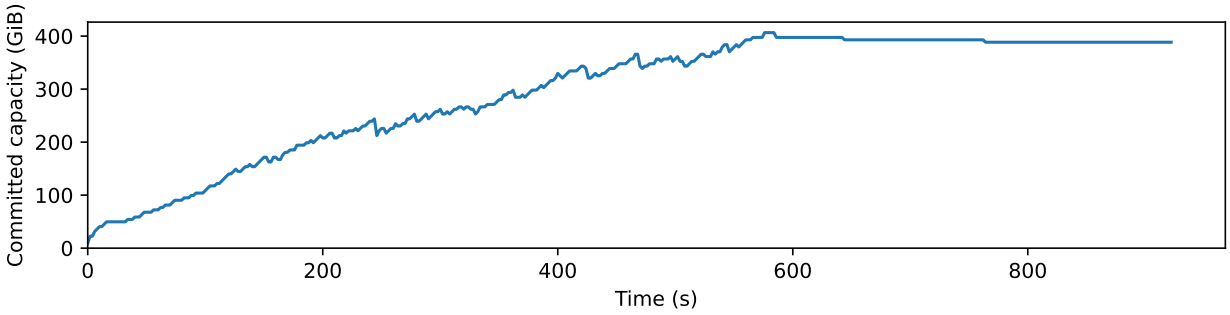


Figure 5.16: RAIZN+ does not experience garbage collection during the RocksDB benchmarks because old logical zones are reset by F2FS, resulting in an upper limit for the number of committed zones during these benchmarks. This graph shows the total used capacity across all devices in a RAIZN+ volume (committed capacity) over the course of a RocksDB fillrandom→overwrite benchmark. After approximately 600 seconds, the size of the database stabilizes, and zones are garbage collected by F2FS in chronological order, resulting in old zones being fully invalidated and reset with no additional garbage collection overhead within RAIZN+.

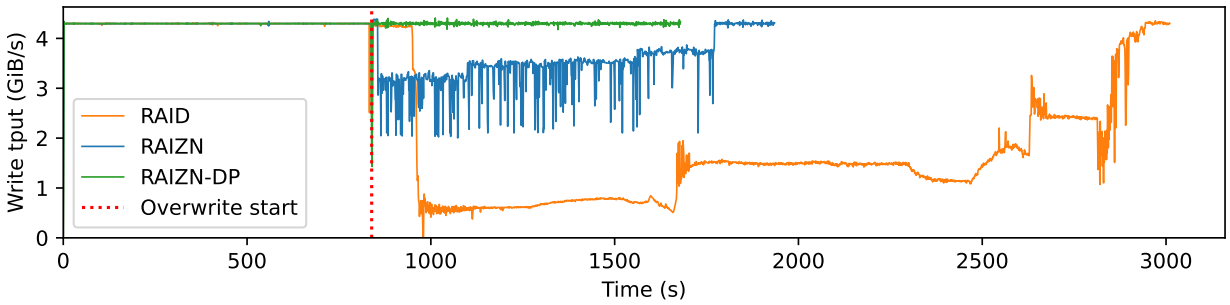


Figure 5.17: Adversarial workloads can trigger garbage collection in RAIZN+, but RAIZN+DP avoids performance degradation by taking advantage of write IOPS headroom from an additional array device.

to reset the corresponding physical zones without incurring any write amplification due to garbage collection. In other words, the probability that a physical zone is only partially invalidated over a long period of time is very low given the workload.

Storage systems based on log-structured merge trees (e.g., RocksDB) typically form RAIZN+-friendly streams. For example, SST compaction in RocksDB typically occurs in a temporally-correlated pattern; old SSTs are eventually deleted and rewritten into the next level, resulting in a situation where data written to a logical zone will, sooner or later, be deleted. Not all workloads have this characteristic, and thus there are workloads that are not friendly to RAIZN+. For example, a workload that does not regularly compact and delete old files, such as SQLite running on F2FS, would likely cause physical zone fragmentation in RAIZN+. However, this class of workload is also not ZNS-friendly, and would result in a high degree of garbage collection not only in RAIZN+, but also within F2FS; given the current state of ZNS-friendly software, we expect many users to use an LSM-based backend when running on ZNS devices (e.g., MySQL configured to use the RocksDB backend).

5.2.9 Garbage collection overhead

Finally, to demonstrate the potential shortcomings of RAIZN+ and the benefits of RAIZN+DP, we crafted an adversarial workload to trigger garbage collection in RAIZN+. Similar to the previous raw device and RocksDB experiments, RAIZN+ was configured to use $2154\,\mathrm{MiB}$ logical zones, and in the case of RAIZN+DP we formatted the 6 devices to have the same aggregate capacity as the 5 devices used for RAIZN+ trials. The workload used in this benchmark is similar to that used in Figure 4.10, with the only difference being that fio is rate limited to $4.4\,\mathrm{GiB/s}$.

To understand why this new workload triggers garbage collection in RAIZN+ while the original workload used in Figure 4.10 does not, it is important to understand the circumstances that can cause garbage collection in RAIZN+. For clarity, we refer to the original workload as *Workload A* and the new adversarial workload as *Workload B*.

Each fully-written physical zone in this configuration of RAIZN+ holds data for at least two logical zones, and for Workloads A and B, after the first pass, every physical zone of every device in the RAIZN+ array holds data for exactly two logical zones.

Garbage collection only occurs in RAIZN+ when serving these workloads if the starting LBAs

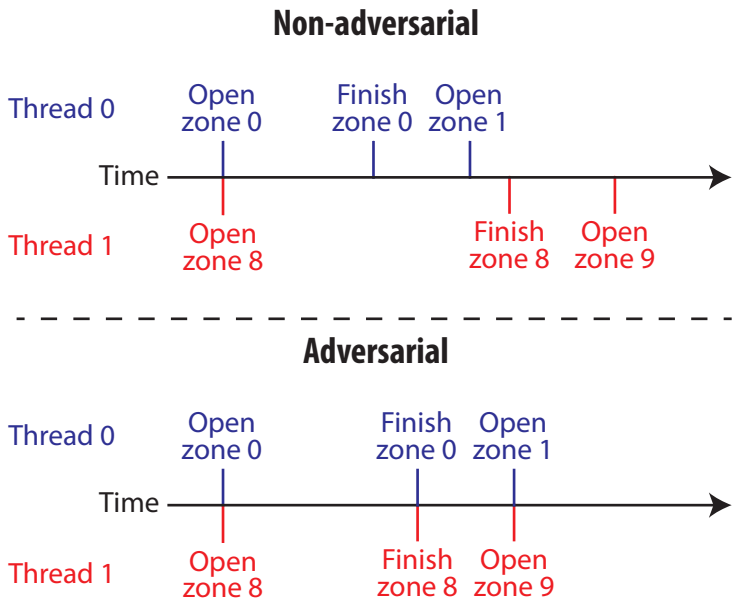


Figure 5.18: The adversarial workload uses rate-limiting to artificially increase the probability of stream mixing.

of the logical zones colocated in the same physical zone are sufficiently separated. For example, if logical zones 0 and 1 are colocated in the same physical zone on a given device, the second pass of workloads A and B will result in sequentially resetting logical zones 0 and 1, thus allowing RAIZN+ to reset the corresponding physical zone without copying any valid data. However, if a given physical zone contains data from logical zones 0 and 500, RAIZN+ will likely run out of spare zones before zone 500 is reset, resulting in a need to copy the data for logical zone 500 to a spare zone before resetting the original physical zone.

Workloads A and B can both be thought of as having five "streams" during the first pass, with each stream covering a different part of the logical address space of the RAIZN+ volume. If one physical zone contains data from two different streams, it will likely result in some amount of write amplification due to garbage collection, as the LBAs for zones corresponding to different streams are quite far apart–we call this *stream mixing*. Stream mixing can only occur if two or more worker threads finish writing a logical zone at the same time, then open their next logical zone at the same time. If, when opening the next logical zone, RAIZN+ happens to swap the physical zone assigned between threads, stream mixing has occurred. Workload A experiences very minimal stream mixing, less than 1% on average–this is because each worker thread is independent and performs IO as fast as possible, resulting in a low probability of two streams finishing a logical zone at the same time. Workload B, however, artificially rate-limits each thread to roughly the same throughput, resulting in many more opportunities for stream mixing and an average of 20% of physical zones experiencing stream mixing over the course of the workload.

The difference is illustrated in Figure 5.18, with a simplified example involving two threads instead of five. In the non-adversarial workload (workload A), the threads are not rate-limited, and as a result they independently proceed at slightly different throughput. Both threads 0 and 1 open their first logical zone at the same time, but thread 0 finishes logical zone 0 slightly faster than thread 1 finishes logical zone 8. Thread 0 then opens logical zone 1 while thread 1 is still writing

logical zone 8, so RAIZN+ allocates the same physical zone to thread 0, resulting in logical zones 0 and 1 being colocated in the same physical zone.

Conversely, the adversarial workload artificially rate limits the threads, preventing thread 0 from finishing its first logical zone faster than the other threads. Both threads 0 and 1 finish writing their respective logical zones at the same time, release the underlying physical zone, then open their next logical zone at the same time. In this scenario, RAIZN+ allocates physical zones in the order in which the open requests were received, resulting in potential stream mixing.

As Figure 5.17 shows, `mdraid` experiences severe performance degradation on the second pass of workload B, and RAIZN+ experiences major but less severe performance degradation. However, RAIZN+DP is able to take advantage of the IOPS headroom to operate without any noticeable garbage collection over the course of workload B.

5.3 Design principles and RAIZN+

This section summarizes how the design principles from Chapter 1 guided our design decisions in RAIZN+.

5.3.1 Designing for efficient writes

Our choice to use optimistic parity instead of delayed parity in RAIZN+ was guided primarily by our understanding that the critical path for writes should be as short as possible to ensure efficient writes. Optimistic parity allows the majority of non-flushed logical appends to be serviced using one round of physical appends, with the reorder record being written in an asynchronous fashion and typically delayed until it can be piggybacked onto a different metadata header. Delayed parity requires that all appends larger than a stripe unit, whether flushed or non-flushed, wait for the first round of physical appends to finish before parity can be calculated and written. The tradeoff is that delayed parity ensures that degraded reads will always have the same performance whether a reordering occurred or not, whereas optimistic parity results in degraded reading of reordered data requiring additional memory copies to "unscramble" it. While delayed parity would have improved efficiency when reading reordered appends, we deemed it more important to optimize writes, and thus chose to implement and evaluate optimistic parity.

Additional persistent metadata overhead was inevitable when adding support for zone indirection and configurable zone size, but RAIZN+ is designed to minimize this additional overhead as much as feasible. RAIZN+ requires logical zone data to be contiguous within a physical zone– this allows a single $4\,\mathrm{B}$ logical zone number coupled with a single $8\,\mathrm{B}$ physical block address to describe the logical to physical zone mapping. This has two key benefits: first, it minimizes the amount of metadata that must be persisted on the critical path for writes, and second, it allows a single $4\,\mathrm{KiB}$ metadata entry to hold mappings for 338 logical zones. Storing logical to physical zone mappings in a concise manner is important for reducing metadata garbage collection overhead and improving the performance of the system overall.

5.3.2 Avoiding over-delivering on performance guarantees

A challenge in designing optimistic parity was, ironically, dealing with the case where no reorderings occurred; given the absence of a reorder record, it is impossible for RAIZN+ to tell whether the reorder record failed to persist before power loss or if it was never written in the first place. To make matters worse, RAIZN+ cannot tell between data that has been written and appended, requiring reorder records for both writes and appends despite the fact that writes cannot be reordered. As a result, RAIZN+ was required to write the reorder record regardless of whether reorderings happened or not, representing a potentially large overhead on every write and append.

However, leveraging our knowledge that power loss is uncommon and that critical data would be written using a synchronous write or append, we gathered and merged reorder records before piggybacking them onto other metadata headers with spare capacity in the inline metadata area. If there were no other metadata headers written within a configurable window of time, RAIZN+ periodically writes a reorder record to ensure a bounded window of data loss.

Zone allocation records are handled in a similar manner to reorder records–they are only flushed when necessary and are inlined into the generation counter metadata entry to avoid additional overhead.

5.3.3 Designing around temporal and spatial locality of client workloads

Adding zone indirection and configurable zone capacity to RAIZN+ was a challenge due to the fundamental problem of data garbage collection. Ultimately, a sufficiently well-crafted adversarial workload will always cause non-negligible garbage collection overhead in RAIZN+, and this is a problem we could not and did not solve for the general case. However, we took advantage of the write characteristics of typical flash-friendly ZNS-compatible applications, and implemented a data placement and garbage collection policy that, as we demonstrated in Section 5.2.8, results in near-zero garbage collection overhead for the expected client workloads. RAIZN+ uses relatively large placement unit sizes (logical zones) in comparison to sector-level data placement granularity for FTL-based SSDs; this large placement unit size, in combination with typical independence of streams, results in a situation where stream mixing can be avoided by "locking" physical zones while the corresponding logical zone is being written, preventing multiple streams from mixing data into the same physical zone.

In our original design proposal, we over-anticipated the level of zone fragmentation and resulting garbage collection overhead in RAIZN+. As a result, we originally proposed mechanisms such as hot/cold tiering of logical zones and intelligent default mapping policies to attempt to minimize garbage collection overhead. However, further analysis of workloads, such as those in Section 5.2.8, showed that ZNS-compatible applications typically form "streams" that write to and reset zones in a roughly sequential fashion. This, coupled with the relatively large size of zones, resulted in a situation where it was unlikely that a given physical zone would contain data from multiple streams, and zone fragmentation was quite rare; this even necessitated the development of an adversarial workload (Section 5.2.9) to trigger garbage collection in RAIZN+.

We do demonstrate that RAIZN+ has its limitations, as an adversarial workload can trigger garbage collection, but we argue that such cases are rare and extreme. RAIZN+DP was designed and implemented as an example of how to design a system for a slightly more adversarial class of workload, such as the one described in Section 5.2.9. However, it is impossible to make a

cost-efficient general-case solution for garbage collection in RAIZN+–instead, it is necessary to craft the solution based on the specific performance, cost, and workload requirements of a given deployment.

Chapter 6

Conclusion and future work

To conclude this thesis, we summarize our contributions, then discuss ongoing and potential future directions for our research on this topic.

6.1 Contributions

Through the analysis of the various challenges and setbacks we faced when designing, implementing, and evaluating CANDStore and RAIZN, we derived a set of three guidelines when building replicated storage systems on top of Intel Optane or ZNS SSDs. These guidelines and the process by which we developed them serve to help systems designers reason about tradeoffs and solve challenges in effectively using emerging persistent storage technologies.

1. Because of the unusually high costs of writes and bulk erases on flash, even those systems expected to be read-heavy frequently benefit from being optimized for write performance first.
2. When working with write-constrained hardware, it is important to set explicit guarantees about durability, performance, fault tolerance, and consistency. Systems should then fulfill, but not exceed, these guarantees to avoid imposing additional write overheads.
3. System designers should identify temporal and spatial locality characteristics of the expected client workload, then design internal data flows and data placement to make optimal use of writes given these workload constraints; this, in many cases, enables systems to overcome the fundamental write-related limitations of persistent hardware and provide greater consistency, durability, or availability than would otherwise be possible.

We applied these guidelines to design and implement RAIZN+, and demonstrated the effectiveness of the guidelines through our evaluation of RAIZN+.

6.1.1 CANDStore

CANDStore is an Optane-based distributed key-value store that provides high availability through a novel online recovery protocol.

Our main contribution through CANDStore is the design, implementation, and evaluation of CANDStore's primary failure recovery protocol, which allows 4.5–10.5× faster recovery than offline recovery. We observe that many systems can tolerate some level of performance degradation

due to headroom between the SLOs and the actual performance of the system, and leverage this observation to precisely define recovery as a return to in-SLO operation. This is a shift from the more traditional notion of recovery as a return to full performance and fault tolerance, and we use this leeway to allow online recovery where hot data is recovered quickly, followed by a long period of time where cold data is slowly copied while the system continues to meet SLOs for client requests. The CANDStore primary replica samples the client workload to identify popular keys, and the system prioritizes the recovery of those popular keys following primary failure. This online recovery protocol is key to overcoming the limited write throughput of Intel Optane and enabling its use as the backing storage medium for primary replicas in main-memory class distributed storage.

CANDStore's workload-guided online recovery protocol is applicable to any system that serves a workload with some degree of temporal locality, or that has performance headroom between the latency/throughput SLO and the steady-state performance of the system. As persistent storage becomes denser per-node, the viability of offline recovery diminishes regardless of whether the underlying storage hardware is write-constrained or not. Online recovery combined with careful protocol-level management of consistency can be used to expedite the repair of a failed primary node in primary-backup storage systems. Outside of replicated storage, any storage system that serves a skewed workload can leverage popularity sampling to prioritize popular data when populating a node; for example, a cache could initialize itself using hot data rather than waiting for cache misses from client requests.

6.1.2 RAIZN

RAIZN is a RAID-5 like system that enables striping and redundancy for arrays of ZNS SSDs. By using arrays ZNS SSDs as underlying storage, RAIZN is able to avoid the on-device garbage collection that plagues conventional SSDs and `mdraid`, resulting in up to $14\times$ higher throughput for certain workloads.

The lack of overwrite and random-write semantics in ZNS SSDs presents many challenges in designing storage systems, but at the same time offers many opportunities for optimization. To overcome the lack of overwrites, RAIZN uses log-structured parity and metadata updates, and to conform with the lack of random-write semantics, RAIZN identifies and performs indirection on addresses affected by torn cross-device writes. However, the ZNS interface also provides some subtle benefits–RAIZN does not need to maintain a stripe cache, and can also leverage the ZNS interface to minimize the amount of data copied onto a replaced device. RAIZN is able to take advantage of the ZNS interface to achieve replacement device rebuild times proportional to the amount of data written to the volume.

The challenges we solved in designing RAIZN are not limited to the context of RAID-like behavior for arrays of ZNS SSDs–instead, we view our contribution as an exploration of the potential challenges faced by any general storage system that manages an array of ZNS SSDs. Persistent metadata management, updating of parity or erasure coded data, and dealing with cross-device torn writes will likely pose a challenge to any replicated storage designed for arrays of ZNS SSDs. In particular, the lack of random overwrites in ZNS will invariably necessitate additional metadata to map a logical namespace on the host to the physical address space of the devices, and minimizing this metadata is crucial to reduce write amplification and make efficient use of ZNS SSDs.

6.1.3 RAIZN+

RAIZN+ is an extension of RAIZN that enables zone appends and configurable zone capacity; we designed RAIZN+ based on the design guidelines derived from our experiences working with CANDStore and RAIZN.

For zone appends, we implemented *optimistic parity*, which is a scheme that optimistically assumes that physical zone appends have been written in submission order, while handling append reorderings in a less efficient manner.

For configurable zone capacity, the primary challenge was garbage collection. We found our simple garbage collection and logical zone placement policy sufficient for our application workloads, but certain synthetic adversarial workloads resulted in performance degradation in RAIZN+. To solve this problem, we designed and implemented RAIZN+DP, which is able to deal with the adversarial workload we tested without experiencing any performance degradation due to garbage collection.

6.2 Future and ongoing work

6.2.1 RAIZN on different ZNS SSDs

In the ZNS SSDs we used in our evaluation, every zone is equal–writing to a single zone can leverage the full parallelism of the entire device, so it is not necessary to operate on multiple zones simultaneously to achieve maximum IO performance. However, certain ZNS SSDs, such as those under development by Samsung, cannot achieve maximum IO throughput when operating on only a single zone [54]. Instead, the zones are divided into multiple categories, with each category spanning a different set of physical resources on the device itself [94]. Thus, it is necessary to concurrently operate on a zone from each category to take full advantage of the hardware resources.

Adding support for such devices, and designing an interface that properly conveys those performance characteristics from the RAIZN logical volume to the host, is potential future work–however, we do not currently have access to such devices.

6.2.2 Zone-aware HDFS

As an extension of our work on RAIZN, we are looking into other systems to improve using ZNS–one such system is HDFS. Part of our ongoing work is on adding ZNS support to HDFS, somewhat similar to work done on adding SMR drive support to HDFS [60].

Current support for ZNS SSDs in HDFS is limited to using F2FS to store HDFS chunks onto the ZNS SSD. We have observed that FTL-induced garbage collection can negatively affect performance of conventional SSDs when used as backing storage for HDFS DataNodes, and that garbage collection within F2FS can hurt performance of ZNS SSDs when used in HDFS.

We believe that it is possible to use application-level support for ZNS in HDFS to achieve lower write amplification and eliminate garbage collection overheads. Current filesystems cannot leverage application-level knowledge without the use of hints, and we believe it is more effective to add ZNS awareness directly into HDFS rather than rely on such hints.

Prior work [105] suggests that application-level ZNS awareness can reduce write amplification in systems like RocksDB, so we hypothesize that taking advantage of the chunking scheme for

HDFS files and some degree of hot/cold tiering could significantly improve the performance of HDFS on workloads that would otherwise cause garbage collection overhead in a conventional SSD or in a filesystem such as F2FS.

6.3 Beyond Optane and ZNS

While our investigation of write-constrained persistent storage technology is limited to Optane NVMM and ZNS SSD, we believe the guidelines we developed are applicable to replicated storage systems built on a wider variety of persistent storage hardware. As highlighted in Section 3.11, existing systems can potentially be improved by taking advantage of workload temporal locality–for example, a cache can be repopulated with popular data identified through sampling during steady-state operation of the cache, rather than waiting for client requests to "naturally" repopulate a cache. In addition, write-constrained persistent storage hardware is likely to necessitate sacrifices and tradeoffs in system characteristics such as availability, durability, consistency, latency, and throughput–understanding the intended workload, setting explicit guarantees, and carefully prioritizing important functionality will be key in providing optimal system performance in the face of hardware-imposed limitations.

The key question that arises in the context of ZNS SSD, Optane memory, and other write-constrained persistent storage technology is which layer of the storage software stack should be aware of and optimized for the hardware idiosyncrasies. While it is clear that adding hardware-awareness to the end-application level is optimal from a technical standpoint (i.e., the application has the greatest knowledge of the workload at hand), it is not always feasible from an engineering or an economic standpoint. Previously, the general trend was to abstract these details away at the device level, such as with the FTL in SSDs–as hardware moves away from this restriction, there is a much greater degree of freedom in implementing hardware-awareness on the host. Moving forward, one of the challenges in using write-constrained persistent storage will be identifying the correct layer to implement hardware-awareness to reap the majority of the benefits from the underlying hardware without incurring excessive engineering costs.

Bibliography

[1] blk_types. https://elixir.bootlin.com/linux/latest/source/include/linux/blk_types.h. 5.1.1

[2] Direct correspondence with western digital engineers. 5.1.2

[3] https://www.westerndigital.com/en-il/products/internal-drives/ultrastar-dc-zn540-nvme-ssd#0TS2094. 5.2.5

[4] Intel Optane DC persistent DIMM prices listed: $842 for 128gb, $2,668 for 256gb, . https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html. **??**, **??**

[5] Device mapper. https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/index.html, . 1.2.2

[6] dm-zap: Host-side zoned host translation mapper, . https://github.com/westerndigitalcorporation/dm-zap. 4.5

[7] . https://lkml.org/lkml/2020/10/15/844. 2.2.5

[8] fio - flexible i/o tester rev. 3.27. https://fio.readthedocs.io/en/latest/fio_doc.html, . 4.4.1

[9] libaio. https://pagure.io/libaio, . 4.4.1

[10] PMDK - libpmem, . https://pmem.io/pmdk/libpmem/. 3.8.1

[11] A guide to mdadm. https://raid.wiki.kernel.org/index.php/A_guide_to_mdadm, . 2.3.2

[12] Mysql. https://www.mysql.com/, . 4.4.3

[13] NVM Express® Base Specification Revision 2.0b January 6th, 2022. https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2.0b-2021.12.18-Ratified.pdf, . 4.2.2

[14] Intel® Optane™ SSD DC P4800X Series, . https://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/data-center-ssds/optane-dc-ssd-series/optane-dc-p4800x-series/p4800x-375gb-aic-20nm.html. 3.8.1, 3.8.6

[15] Intel Optane DC P4800X 375 GB SSD - ReconDeals ($1066.00), . http://recondeals.com/product/intel-optane-dc-p4800x-375-gb-solid-state-drive-internal-pci-express-pci-ex/. **??**

[16] Percona myrocks introduction. https://www.percona.com/doc/percona-server/8.0/myrocks/index.html, . 4.4.3

[17] Smartftl architecture for ssds. https://www.youtube.com/watch?v=3O3zDrpt3uM, . 4.5

[18] Rocksdb: Zenfs storage backend, . https://github.com/westerndigitalcorporation/zenfs. 2.2.5

[19] . https://zonedstorage.io/. 4.1

[20] File systems. https://zonedstorage.io/docs/linux/fs. 2.2.5

[21] Explicit volatile write back cache control. https://www.kernel.org/doc/Documentation/block/writeback_cache_control.txt, 2022. 4.1, 4.3.3

[22] Thomas E. Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N. Schuh, and Emmett Witchel. Assise: Performance and availability via NVM colocation in a distributed file system. 2019. 3.1, **??**, 3.2

[23] Daniel S. Berger, Ramesh K. Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *Proceedings of the 14th USENIX Conference on Networked Systems Design and Implementation*, NSDI'17, page 483–498, USA, 2017. USENIX Association. ISBN 9781931971379. 3.4.2

[24] Matias Bjørling. Zone append: A new way of writing to zoned storage. Santa Clara, CA, February 2020. USENIX Association. 5.1.1

[25] Matias Bjørling, Philippe Bonnet, Luc Bouganim, and Niv Dayan. The necessary death of the block device interface. 2012. 2.2.3

[26] 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. 2.2.5, 4.5

[27] 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. 1.1, 2.2.3, 4.1, 4.4.1

[28] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and zipf-like distributions: Evidence and implications. In *IEEE INFOCOM'99. Conference on Computer Communications. Proceedings. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. The Future is Now (Cat. No. 99CH36320)*, volume 1, pages 126–134. IEEE, 1999. 3.2

[29] Neil Brown. A journal for md/raid5. https://lwn.net/Articles/665299/, 2015. 2.3.2

[30] Mudashiru Busari and Carey Williamson. On the sensitivity of web proxy cache performance to workload characteristics. In *Proceedings IEEE INFOCOM 2001. Conference on Computer Communications. Twentieth Annual Joint Conference of the IEEE Computer and Communications Society (Cat. No. 01CH37213)*, volume 3, pages 1225–1234. IEEE, 2001. 3.2

[31] Zhichao Cao, Siying Dong, Sagar Vemuri, and David H. C. Du. Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook. In *FAST*, 2020. 3.6.2

[32] Yanpei Chen, Sara Alspaugh, Dhruba Borthakur, and Randy Katz. Energy efficiency for large-scale mapreduce workloads with significant interactive analysis. In *Proceedings of the 7th ACM european conference on Computer Systems*, pages 43–56, 2012. 3.2

[33] Ross Stenfort (Meta) Chris Sabol (Google). Hyperscale innovation: Flexible data placement mode (fdp). `https://nvmexpress.org/wp-content/uploads/Hyperscale-Innovation-Flexible-Data-Placement-Mode-FDP.pdf`. 2.2.3, 4.5

[34] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. 3.8.3

[35] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proc. 10th USENIX OSDI*. USENIX, 2012. 3.1

[36] Western Digital Corporation. Nvme zoned namespaces (zns) devices. `https://zonedstorage.io/docs/introduction/zns`. 5.1.1

[37] Western Digital Corporation. Nvme zoned namespaces (zns) devices. https://zonedstorage.io/docs/introduction/zns, 2019. 1.1

[38] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56 (2):74–80, February 2013. 1.1, 3.9

[39] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007. 1

[40] Siying Dong, Andrew Kryczka, Yanqin Jin, and Michael Stumm. Rocksdb: evolution of development priorities in a key-value store serving large-scale applications. *ACM Transactions on Storage (TOS)*, 17(4):1–32, 2021. 5.2.5

[41] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *Proc. 25th ACM Symposium on Operating Systems Principles (SOSP)*, Monterey, CA, October 2015. 3.1, 3.4, 3.10

[42] Aaron J. Elmore, Vaibhav Arora, Rebecca Taft, Andrew Pavlo, Divyakant Agrawal, and Amr El Abbadi. Squall: Fine-grained live reconfiguration for partitioned main memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 299–313, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450327589. doi: 10.1145/2723372.2723726. URL `https://doi.org/10.1145/2723372.2723726`. 3.6.2, 3.10

[43] Facebook. RocksDB. `http://rocksdb.org/`, 2015. 3.1, 4.4.3

[44] Facebook. Performance Benchmarks. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools, 2021. 4.4.3

[45] Daniel Ford, Francois Labelle, Florentina Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. 2.1

[46] Yingxun Fu, Jiwu Shu, Xianghong Luo, Zhirong Shen, and Qingda Hu. Short code: An efficient raid-6 mds code for optimizing degraded reads and partial stripe writes. *IEEE Transactions on Computers*, 66(1):127–137, 2016. 4.2.2

[47] Shahram Ghandeharizadeh and Haoyu Huang. Gemini: A distributed crash recovery protocol for persistent caches. In *Proceedings of the 19th International Middleware Conference*, Middleware '18, page 134–145, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450357029. doi: 10.1145/3274808.3274819. URL https://doi.org/10.1145/3274808.3274819. 3.10

[48] Javier González, Matias Bjørling, Seongno Lee, Charlie Dong, and Yiren Ronnie Huang. Application-driven flash translation layers on open-channel ssds. In *Proceedings of the 7th non Volatile Memory Workshop (NVMW)*, pages 1–2, 2016. 2.2.3

[49] Lei Guo, Enhua Tan, Songqing Chen, Zhen Xiao, and Xiaodong Zhang. Does internet media traffic really follow zipf-like distribution? In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 359–360, 2007. 3.2

[50] 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. 4.4.3

[51] Mark Holland and Garth A Gibson. Parity declustering for continuous operation in redundant disk arrays. *ACM SIGPLAN Notices*, 27(9):23–35, 1992. 5.2.5

[52] Cheng Huang, Huseyin Simitci, Yikang Xu, Aaron Ogus, Brad Calder, Parikshit Gopalan, Jin Li, and Sergey Yekhanin. Erasure coding in windows azure storage. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 15–26, 2012. 1

[53] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, volume 8, 2010. 2.1

[54] Minwoo Im, Kyungsu Kang, and Heonyoung Yeom. Accelerating rocksdb for small-zone zns ssds by parallel i/o mechanism. In *Proceedings of the 23rd International Middleware Conference Industrial Track*, Middleware Industrial Track '22, page 15–21, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450399173. doi: 10.1145/3564695.3564774. URL https://doi.org/10.1145/3564695.3564774. 6.2.1

[55] Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html. 1.1, 3.1, 3.8.1

[56] 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. 4.5

[57] 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. doi: 10.1109/ICCDCS.1998.705871. 4.1

[58] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. Kaml: A flexible, high-performance key-value ssd. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 373–384. IEEE, 2017. 2.2.3

[59] 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. 4.5

[60] Shin'ichiro Kawasaki. Zoned block device support in hadoop hdfs. 6.2.2

[61] Thomas Kim, Daniel Lin-Kit Wong, Gregory R Ganger, Michael Kaminsky, and David G Andersen. High availability in cheap distributed key value storage. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, pages 165–178, 2020. 1.1

[62] Alexey Kopytov. Sysbench manual. *MySQL AB*, pages 2–3, 2012. 4.4.3

[63] Isao Kotera, Ryusuke Egawa, Hiroyuki Takizawa, and Hiroaki Kobayashi. Modeling of cache access behavior based on zipf's law. In *Proceedings of the 9th workshop on MEmory performance: DEaling with Applications, systems and architecture*, pages 9–15, 2008. 3.2

[64] Andrew Krioukov, Lakshmi N Bairavasundaram, Garth R Goodson, Kiran Srinivasan, Randy Thelen, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Parity lost and parity regained. In *FAST*, volume 2008, page 127, 2008. 4.3.2

[65] Chinmay Kulkarni, Aniraj Kesavan, Tian Zhang, Robert Ricci, and Ryan Stutsman. Rocksteady: Fast migration for low-latency in-memory storage. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 390–405, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi: 10.1145/3132747.3132784. URL `https://doi.org/10.1145/3132747.3132784`. 3.6.2, 3.10

[66] Leslie Lamport and Mike Massa. Cheap Paxos. In *Proceedings of the 2004 International Conference on Dependable Systems and Networks*, DSN '04, pages 307–, Washington, DC, USA, 2004. IEEE Computer Society. 3.2, 3.3, 3.7.1

[67] 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. 2.2.5

[68] 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. 4.5

[69] Dominic Lencer, Martin Salinga, and Matthias Wuttig. Design rules for phase-change materials in data storage applications. *Advanced Materials*, 23(18):2030–2058, 2011. 1

[70] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. Kvell: The design and implementation of a fast persistent key-value store. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, page 447–461, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359628. URL https://doi.org/10.1145/3341301.3359628. 3.10

[71] Alberto Lerner and Philippe Bonnet. Not your grandpa's ssd: The era of co-designed storage devices. In *Proceedings of the 2021 International Conference on Management of Data*, SIGMOD '21, page 2852–2858, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383431. doi: 10.1145/3448016.3457540. URL https://doi.org/10.1145/3448016.3457540. 2.2.3

[72] Chris A Mack. Fifty years of moore's law. *IEEE Transactions on semiconductor manufacturing*, 24(2):202–207, 2011. 1.1

[73] 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. 4.5

[74] 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. 4.5

[75] Wojciech Malikowski. Spdk open-channel ssd ftl, 2018. https://spdk.io/doc/ftl.html. 2.2.5, 4.5

[76] Mellanox ConnectX-3 Product Brief. http://www.mellanox.com/related-docs/prod_adapter_cards/ConnectX3_EN_Card.pdf. 3.8.1

[77] Damien Le Moal. dm-zoned: Zoned block device device mapper, 2017. https://lwn.net/Articles/714387/. 2.2.5, 4.5

[78] Iulian Moraru, David G. Andersen, and Michael Kaminsky. Paxos quorum leases: Fast reads without sacrificing writes. 3.4

[79] NVM-Express. Nvme explained. https://nvmexpress.org/wp-content/uploads/2013/04/NVM_whitepaper.pdf, 2012. 2.2.1

[80] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. 3.2

[81] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast crash recovery in RAMCloud. 1.2.1, 2.3.1, 3.10

[82] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. The case for RAMClouds: Scalable high-performance storage entirely in DRAM. In *Operating Systems Review*, volume 43, pages 92–105, January 2010. 3.1

[83] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud storage system. *ACM TOCS*, 2015. 2.3.1, 3.8.4

[84] Chanik Park, Wonmoon Cheon, Jeonguk Kang, Kangho Roh, Wonhee Cho, and Jin-Soo

Kim. A reconfigurable ftl (flash translation layer) architecture for nand flash-based applications. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(4):1–23, 2008. 2.2.3

[85] Pankesh Patel, Ajith H Ranabahu, and Amit P Sheth. Service level agreement in cloud computing. 2009. 3.2

[86] 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. 2.1, 2.3.2

[87] 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. ISBN 978-1-939133-12-0. URL `https://www.usenix.org/conference/fast20/presentation/pitchumani`. 4.5

[88] Hongwei Qin, Dan Feng, Wei Tong, Yutong Zhao, Sheng Qiu, Fei Liu, and Shu Li. Better atomic writes by exposing the flash out-of-band area to file systems. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 12–23, 2021. 4.1

[89] Ohad Rodeh, Josef Bacik, and Chris Mason. Btrfs: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013. 2.2.5, 2.3.2

[90] Andy Rudoff. Write atomicity and nvm drive design. `https://www.bswd.com/FMS12/FMS12-Rudoff.pdf`. 4.1

[91] Marta Rybczyńska. Btrfs on zoned block devices. `https://lwn.net/Articles/853308/`. 2.2.5

[92] Bianca Schroeder and Garth A Gibson. Understanding disk failure rates: What does an mttf of 1,000,000 hours mean to you? *ACM Transactions on Storage (TOS)*, 3(3):8–es, 2007. 2.1, 2.3.2

[93] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, pages 1–14, 2011. 3.2

[94] Hojin Shin, Myounghoon Oh, Gunhee Choi, and Jongmoo Choi. Exploring performance characteristics of zns ssds: Observation and implication. In *2020 9th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–5. IEEE, 2020. 6.2.1

[95] Swaminathan Sivasubramanian. Amazon dynamodb: a seamlessly scalable non-relational database service. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 729–730, 2012. 3.7

[96] Kent Smith. Garbage collection. *SandForce, Flash Memory Summit, Santa Clara, CA*, pages 1–9, 2011. 4

[97] Theano Stavrinos, Daniel S. Berger, Ethan Katz-Bassett, and Wyatt Lloyd. Don't be a blockhead: Zoned namespaces make work on conventional ssds obsolete. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 144–151, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384384. doi: 10.

1145/3458336.3465300. URL https://doi.org/10.1145/3458336.3465300. 2.2.3

[98] Thomas N Theis and H-S Philip Wong. The end of moore's law: A new beginning for information technology. *Computing in Science & Engineering*, 19(2):41–50, 2017. 1.1

[99] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. 3.10

[100] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. Building blocks for persistent memory. *The VLDB Journal*, 29(6):1223–1241, 2020. 3.8.1

[101] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, page 1041–1052, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450341974. doi: 10.1145/3035918.3056101. URL https://doi.org/10.1145/3035918.3056101. 3.1

[102] Hua Wang, Ping Huang, Shuang He, Ke Zhou, Chunhua Li, and Xubin He. A novel i/o scheduler for ssd with improved performance and lifetime. In *2013 IEEE 29th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–5. IEEE, 2013. 3.8.6

[103] Hakim Weatherspoon and John D Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *International Workshop on Peer-to-Peer Systems*, pages 328–337. Springer, 2002. 1, 2.1

[104] Dan J Williams. Md raid acceleration. In *Linux Symposium*, page 409, 2006. 4.3.1

[105] Denghui Wu, Biyong Liu, Wei Zhao, and Wei Tong. Znskv: Reducing data migration in lsmt-based kv stores on zns ssds. In *2022 IEEE 40th International Conference on Computer Design (ICCD)*, pages 411–414, 2022. doi: 10.1109/ICCD56317.2022.00067. 6.2.2

[106] Yinjun Wu, Kwanghyun Park, Rathijit Sen, Brian Kroth, and Jaeyoung Do. Lessons learned from the early performance evaluation of intel optane dc persistent memory in dbms. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*, DaMoN '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450380249. doi: 10.1145/3399666.3399898. URL https://doi.org/10.1145/3399666.3399898. 3.1

[107] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, SYSTOR '15, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336079. doi: 10.1145/2757667.2757684. URL https://doi.org/10.1145/2757667.2757684. 2.2.1

[108] Jinfeng Yang, Bingzhe Li, and David J. Lilja. Exploring performance characteristics of the optane 3d xpoint storage technology. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 5 (1), feb 2020. ISSN 2376-3639. doi: 10.1145/3372783. URL https://doi.org/10.1145/3372783. 3.1

[109] Yue Yang and Jianwen Zhu. Write skew and zipf distribution: Evidence and implications. *ACM transactions on Storage (TOS)*, 12(4):1–19, 2016. 3.2

[110] Saifeng Zeng, Ligu Zhu, and Lei Zhang. A high reliable and performance data distribution strategy: A raid-5 case study. In *2013 Ninth International Conference on Computational Intelligence and Security*, pages 318–322, 2013. doi: 10.1109/CIS.2013.74. 4.3.2

[111] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010. 3.2

[112] Yupu Zhang, Abhishek Rajimwale, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. End-to-end data integrity for file systems: A zfs case study. In *FAST*, pages 29–42, 2010. 2.3.2