# ZRAID: Leveraging Zone Random Write Area (ZRWA) for Alleviating Partial Parity Tax in ZNS RAID


Minwook Kim
Seoul National University
Seoul, Korea
ace@snu.ac.kr

Seongyeop Jeong
Seoul National University
Seoul, Korea
seongyeop.jeong@snu.ac.kr

Jin-Soo Kim
Seoul National University
Seoul, Korea
jinsoo.kim@snu.ac.kr



## Abstract

The Zoned Namespace (ZNS) SSD is an innovative technology that aims to mitigate the *block interface tax* associated with conventional SSDs. However, constructing a RAID system using ZNS SSDs presents a significant challenge in managing partial parity for incomplete stripes. Previous research permanently logs partial parity in a limited number of reserved zones, which not only creates bottlenecks in throughput but also exacerbates write amplification, thereby reducing the device's lifetime. We refer to these inefficiencies as the *partial parity tax*.

In this paper, we present ZRAID, a software ZNS RAID layer that leverages the newly added Zone Random Write Area (ZRWA) feature in the ZNS Command Set, to alleviate *partial parity tax*. ZRWA enables in-place updates within a confined area near the write pointer. ZRAID temporarily stores partial parity within the ZRWA of data zones. Thus, partial parity writes are distributed across multiple data zones, effectively eliminating throughput bottlenecks. Furthermore, any expired partial parity in the ZRWA is overwritten by subsequent data, avoiding unnecessary flash writes. With the introduction of ZRWA, ZRAID can leverage general schedulers, overcoming the queue depth limitations of ZNS-compatible schedulers. Our evaluation with actual ZNS SSDs demonstrates a significant improvement in write throughput: up to 34.7% in the fio microbenchmark, and an average of 14.5% in db_bench on RocksDB, along with up to a 1.6x reduction in flash write amplification.



***CCS Concepts:*** • **Information systems** → **RAID**; • **Computer systems organization** → **Reliability**; • **Hardware** → **External storage**.

***Keywords:*** Zoned Namespaces, RAID, Zone Random Write Area (ZRWA), Partial Parity





*ASPLOS '25, March 30-April 3, 2025, Rotterdam, Netherlands*



ACM ISBN 979-8-4007-0698-1/25/03
https://doi.org/10.1145/3669940.3707248

**ACM Reference Format:**
Minwook Kim, Seongyeop Jeong, and Jin-Soo Kim. 2025. ZRAID: Leveraging Zone Random Write Area (ZRWA) for Alleviating Partial Parity Tax in ZNS RAID. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3669940.3707248


## 1 Introduction

The Zoned Namespace (ZNS) SSD is a state-of-the-art solution designed to mitigate the overhead associated with the block interface in conventional SSDs [11, 15, 32, 37, 38]. The ZNS interface organizes the storage space into distinct zones that only support sequential write operations, exposing write pointers to the host [5, 9, 15]. The host is required to submit write requests at the designated write pointer of a zone, with the pointer incrementally advancing in the size of each write request. Additionally, the responsibility of managing zone erasures for reuse, a task previously handled by the Flash Translation Layer (FTL) in conventional SSDs, now falls to the host. While this approach does increase management tasks for the host, it also effectively reduces the internal overhead and costs associated with SSDs. Due to its competitive pricing and predictable performance, ZNS SSDs are particularly attractive within the data center storage market [7, 14].

In data center environments where storage devices are commonly deployed in arrays of multiple units, RAID is the conventional strategy employed to enhance both performance and reliability [10, 26, 29]. This methodology is also applicable to ZNS SSDs. ZNS RAID aggregates multiple ZNS SSDs to provide a unified ZNS abstraction to the host through the ZNS interface, typically incorporating redundancy mechanisms such as parity to meet reliability needs [23].

However, implementing a parity-based RAID system on ZNS SSDs introduces significant challenges. Writing data that is not aligned with stripe boundaries leads to the generation of an incomplete, partial stripe that also necessitates protection via *partial parity* (referred to interchangeably as *PP*). The partial parity cannot be written directly to the corresponding stripe's parity location because it needs to be updated when subsequent data belonging to that stripe arrives. Consequently, partial parities should be managed in a separate, dedicated area in a persistent manner until the full stripe's data becomes available.

The simplest way to address the issue of partial parity on ZNS RAID is to buffer incoming data in the host RAID driver until the full stripe's data accumulates. When the full stripe is ready, the driver can calculate the full parity and write it to the parity area in the ZNS SSD. However, this approach is not acceptable in data center environments due to the potential risk of data loss on a sudden power failure. By employing Non-Volatile Memory (NVM) such as Optane, the risk of data loss inherent in the previous scheme can be avoided. While this approach enhances data reliability, it incurs additional costs and introduces a dependency on specific hardware components that have been discontinued from production.

In RAIZN [23], several *partial parity zones* are reserved for the persistent storage of partial parities. This approach, however, causes a bottleneck in write throughput as partial parities from numerous data zones are directed toward significantly fewer partial parity zones. Moreover, permanently storing partial parities wastes valuable device resources and incurs zone management overhead. We refer to the additional overheads that arise from managing partial parities in ZNS RAID as the *partial parity tax*. It encapsulates the additional resource consumption and potential performance impacts of ensuring data durability with partial parity in ZNS RAID, where data cannot be directly overwritten. In addition to these overheads, it is also observed that existing ZNS RAID systems share the ZNS SSD's inherent limitation of a restricted queue depth, while further contributing to the reduction in throughput.

In this paper, we introduce ZRAID, a novel software-driven ZNS RAID framework designed to address the *partial parity tax* problem. By leveraging the *Zone Random Write Area (ZRWA)* feature of ZNS SSDs, ZRAID not only alleviates the *partial parity tax*, but also boosts the efficiency and performance of ZNS-based RAID systems. ZRAID temporarily stores partial parity within the ZRWA of the originating data zone, which is then quickly overwritten by subsequent data writes. This design effectively distributes the writing of partial parities across multiple data zones, avoiding the need for separate parity zones as employed in the existing ZNS RAID systems. Since partial parities only exist until they are overwritten by data, there is no need to store them permanently on flash media. Moreover, ZRAID enables the use of general disk schedulers, increasing the queue depth for write operations and consequently boosting write throughput further by utilizing the ZRWA feature.

However, integrating ZRWA into ZNS RAID is not straightforward and introduces two main challenges. First, write submissions must be confined to a specific range to prevent data block overwriting by partial parity and implicit advancement of the write pointer. Second, the write pointer must be carefully advanced, as it serves as a crucial indicator during recovery. This paper presents ZRAID's approaches to address these challenges, ensuring high performance and robustness against power and device failures.

Our evaluation results, derived from a variety of micro- and macro-benchmarks, demonstrate that ZRAID enhances write throughput by 13% to 48% compared to the existing ZNS RAID systems, while simultaneously reducing the flash write by up to a factor of 1.6. The source code of ZRAID is publicly available at https://github.com/snu-csl/zraid. The contributions of this paper can be summarized as follows:

- We identify the problem of *partial parity tax* in existing ZNS RAID systems, causing significant throughput bottlenecks and increased write amplification.
- We introduce ZRAID, the first ZNS RAID system that leverages the ZRWA feature to maximize write throughput and device lifetime.
- We implement and evaluate ZRAID on real ZNS SSDs, demonstrating its effectiveness and efficiency in various scenarios.

This paper is structured as follows. Section 2 overviews ZNS SSDs and ZNS RAID. Section 3 delves into challenges encountered by current ZNS RAID systems. Section 4 presents the architecture of ZRAID and its approach to overcoming the aforementioned challenges. Section 5 discusses the corner cases of ZRAID. Section 6 provides the evaluation results of ZRAID through a series of micro- and macro-benchmarks. Finally, Section 7 concludes the paper.

## 2 Background

### 2.1 ZNS SSD

Unlike conventional SSDs, ZNS SSDs segment their address space into zones that only permit sequential writes, with each zone representing the minimum erasure unit. Each zone has a Write Pointer (WP) that indicates the next writable block address in that zone, and a write request that does not start at the WP fails. The transition to the ZNS interface yields significant benefits for a device. However, this shift transfers the burden of managing write constraints and performing garbage collection (GC) operations to the host, necessitating substantial modifications to the conventional storage I/O stack [15].

A key element of the I/O stack that requires modification is the I/O scheduler. Within multi-queue environments, the sequential submission of write requests by applications does not guarantee their sequential dispatch [36], posing a risk of write failures in normal ZNS zones. In the latest Linux kernel, the `mq-deadline` scheduler addresses this issue by rearranging the dispatch sequence of write requests to comply with the sequential write requirements [2, 5].

Existing ZNS SSDs are available in two configurations: large-zone and small-zone models. The large-zone model maximizes throughput within a single zone but can lead to interference between zones. In contrast, small-zone ZNS SSDs sacrifice channel parallelism within a single zone to

achieve better performance isolation across zones. Western Digital's Ultrastar DC ZN540 [3] is the world's first mass-produced ZNS SSD, classified as a large-zone model. The 1TB model used for the ZRAID implementation includes 904 zones with a zone capacity of 1077MB and supports up to 14 maximum open zones. Its performance rates at 1230MB/s for sequential writes. Meanwhile, Samsung's PM1731a [17] is a small-zone ZNS SSD with 40,704 zones, each 96MB in size. A single zone of this device has a throughput of around 45MB/s, with the accumulated throughput increasing in proportion to the number of open zones.

### 2.2 ZNS RAID

Conventional RAID [30] operates on overwrite-capable block devices, where a *stripe* refers to a set of contiguous blocks across all devices, and a *chunk* is a segment of data stored on a single device within that stripe. In a RAID array, all stripes are initially written with predefined values to ensure that the parity chunk is accurately encoded based on the data chunks [6]. After initialization, subsequent data chunks overwrite the full stripe, triggering an update to the full parity chunk.

This scheme ensures that even writes not aligned to stripe boundaries (called *stripe-unaligned writes*) are absorbed into the full stripe and protected by full parity. In conventional RAID, such stripe-unaligned writes introduce significant overhead due to the need for read-modify-write operations or dedicated journaling space for parity updates. However, parity chunks are updated immediately without waiting for full-stripe completion, to maximize data durability [33].

On the other hand, ZNS RAID combines the physical zones of each ZNS SSD to form a large logical zone, exposing the ZNS interface to the host. In ZNS RAID, the overall architecture is similar to conventional RAID, with statically determined data and parity positions. Nevertheless, the write constraints of ZNS SSDs prevent pre-writing during initialization, as zones must be written sequentially, and subsequent data cannot overwrite the initial values. This results in the creation of partial stripes from stripe-unaligned writes.

Given that ZNS SSDs do not allow overwriting, the full parity can only be written once the entire stripe is complete. As a result, partial parities (PPs) needed for protecting partial stripes must be managed in a separate space outside the stripe [23]. The amount of these PPs is not negligible, as each chunk-sized write produces a PP chunk of the same size, except for the last chunk within a stripe. To maintain the same durability semantics as conventional RAID, partial parity should be written immediately without waiting for the full stripe to complete.

In ZNS RAID, where only sequential writes are allowed, partial stripes tend to be quickly promoted into full stripes, making PP blocks short-lived metadata. Despite this, PP blocks must be stored in non-volatile storage to safeguard against power failures before full stripe promotion. This incurs additional overhead in terms of performance, storage space, and device lifetime.

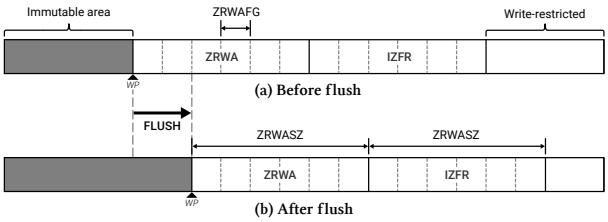



**Figure 1. Operation of the ZRWA [9].**

### 2.3 Zone Random Write Area (ZRWA)

The Zone Random Write Area (ZRWA) feature has been introduced in the NVMe ZNS Command Set to mitigate the restrictive write constraints inherent in the ZNS interface [8, 12, 14, 16, 20]. ZRWA provides a relatively small-sized, non-volatile write buffer to the host, enabling in-place random writes in areas adjacent to the Write Pointer (WP). The ZRWA extends from the WP, covering a fixed-sized area towards the larger block address, and moves forward as the WP advances.

ZRWA is typically implemented using separate backing storage, distinct from the main flash area. This backing storage is usually made of battery-backed DRAM or SLC (Single-Level Cell) flash memory. Both options offer substantially higher endurance and superior performance compared to TLC (Triple-Level Cell) or QLC (Quad-Level Cell) technologies, which currently dominate or are expected to dominate the mainstream flash storage market [4]. In this scheme, overwritten data in the ZRWA is not flushed to the main flash area but instead expires within the ZRWA's backing storage. Thus, ZRWA is well-suited to handle short-lived data, thereby enhancing performance, optimizing flash space utilization, and extending the device's overall lifetime.

Figure 1 illustrates both the WP and the adjacent ZRWA area. Here, ZRWASZ and ZRWAFG denote the total size of the ZRWA and the granularity of the explicit ZRWA flush operation, respectively. IZFR refers to the Implicit Zone Flush Region, which is adjacent to the ZRWA and of the same size. The 1TB model of the ZN540 supports ZRWA features with a ZRWA size of 1MB and a flush granularity of 16KB.

The WPs in ZRWA-enabled zones advance in different ways: either through specific commands or by an implicit advancement mechanism, as outlined below. These mechanisms advance the WP in units of the ZRWA flush granularity rather than the device block size.

**ZRWA Explicit Flush.** The ZRWA explicit flush command specifies the location to which the WP should advance. Figure 1 depicts the change in zone states before and after the command. Two regions adjacent to the WP, ZRWA and IZFR, advance together with the WP. This specific command is currently being supported in the latest NVMe command line interface, `nvme-cli` [1]. We also confirmed that this

command operates correctly on both ZN540 and PM1731a devices.

**ZRWA Implicit Flush.** The IZFR represents the maximum allowable range for random writes within a ZRWA-enabled zone. When the end block address of a write falls within this region, the WP advances implicitly, along with the ZRWA and IZFR. This advancement is carried out in units of ZRWAFG until the end block address of the write is included within the ZRWA [9].

When the WP approaches the end of the zone (i.e., $WP = zone\_capacity - 2 \times ZRWASZ$), the IZFR begins to contract. Once the WP reaches the point where $WP = zone\_capacity - 1 \times ZRWASZ$, the IZFR disappears, making further ZRWA implicit flushes impossible. From this point on, the WP can only advance through an explicit ZRWA flush command. When the WP finally reaches the zone capacity, the zone transitions to a *full* state.

### 2.4 Related Work

RAIZN [23] is the first ZNS RAID system, which organizes data and full parity in a RAID-5 configuration, exposing the ZNS interface to the host through the Linux's device mapper (`dm`) layer. RAIZN designates a reserved zone for separate partial parity storage. During initialization, RAIZN allocates five physical zones per device: one for the PP, one for the superblock and metadata log, and the remaining three as spare zones for garbage collection. PP blocks from partial stripes are appended to the PP zone of the parity device. The association between a specified partial-stripe write and its PP is not static, necessitating metadata header blocks for identifying the related write during recovery.

RAIZN addresses specific challenges of ZNS RAID, maintaining correctness even in edge cases without compromising performance compared to conventional RAID systems. However, it suffers from the *partial parity tax* issue and inherits the write queue depth limitation from the underlying ZNS SSDs.

ZapRAID [35] is an extended Log-RAID design tailored for ZNS SSDs, leveraging the Zone Append command to enhance write throughput compared to the standard Zone Write command. However, it presents a traditional block interface to the host by mapping logical block addresses to the physical block addresses of the underlying ZNS SSDs, requiring complex metadata management. ZapRAID also stores block metadata for fault tolerance in the flash page's out-of-band area, which limits its general applicability. Furthermore, it is implemented using SPDK, making it incompatible with kernel-based file systems like F2FS. A key difference between ZapRAID and RAIZN lies in their consistency policies. ZapRAID writes parity only when the stripe is complete, while delaying the completion of stripe-unaligned writes. These differences complicate direct performance comparisons between ZapRAID and ZNS RAID systems.

To the best of our knowledge, ZRAID is the first approach to enhance ZNS RAID systems by leveraging two principal advantages of ZRWA: the overwriting capability and higher write queue depth. The overwriting capability can be utilized to manage a significant volume of partial parity, which is known to be short-lived, a potential noted in prior research [27, 31]. Furthermore, the performance improvement in an individual ZNS SSD from higher queue depths directly impacts the performance of the entire RAID array.

## 3 Challenges of ZNS RAID

In this section, we explore the fundamental issues of the *partial parity tax*, a primary challenge in existing ZNS RAID methodologies that rely on normal zones. Additionally, we discuss how the limited queue depth of normal zones inherently leads to reduced write throughput. We also highlight the challenges faced by existing ZNS RAID systems in maintaining write atomicity, which requires the use of complex recovery and redirection algorithms.

### 3.1 Partial Parity Zone Contention

In the existing ZNS RAID, it is imperative to minimize the number of zones dedicated to partial parity in favor of maximizing data zone availability. This is because the maximum number of active zones is limited in ZNS SSDs [3] due to the device's resource limitations. For example, the 1TB ZN540 model supports only 14 active zones out of a total of 904 zones (1.5%), while the 3.7TB PM1731a model supports 384 active zones out of 40,704 zones (0.94%). RAIZN utilizes one PP zone and 12 data zones on 1TB ZN540 devices, resulting in a data-to-PP zone ratio of 12:1 [22, 23].

This relative scarcity of PP zones introduces the possibility of contention. In a five-disk ZN540 array, the additional writes generated by PP can account for up to 60% (3 chunks out of a 5-chunk stripe) of the total traffic to the 12 data zones, all of which would be concentrated in a single PP zone. This is an unavoidable problem in ZNS RAID architectures with dedicated PP zones, and the situation is further exacerbated by the additional metadata header blocks (see §3.2).

This contention in a single PP zone inherently leads to system-wide performance degradation. In accordance with RAID semantics, the confirmation of write completion depends on the successful completion of all associated physical I/Os, including the writes to PP blocks. Thus, any delay in writing PP blocks directly impacts overall write performance.

### 3.2 Inefficient Resource Usage for Partial Parity

The lifespan of PP is very brief, becoming outdated once the associated partial stripe is fully written. Yet, existing ZNS RAID systems persist PP in a separate zone, retaining it far beyond its necessary duration. This results in space waste by requiring the allocation of separate dedicated PP zones. Moreover, the significant volume of PP blocks (see

§3.1) written to these zones triggers frequent GC operations, further worsening the inefficiency. RAIZN attempts to reduce GC overhead by keeping valid PP blocks in memory, yet zone erasures remain unavoidable. This accelerates wear on the flash media, which is particularly problematic for Quad-Level Cell (QLC) flash, given its lower endurance of around 1000 Program/Erase (P/E) cycles [4].

The locations of PP blocks are not statically determined in existing ZNS RAID systems, requiring metadata headers to locate them during recovery. Although these headers are very small in size, they pose an issue because they must be written in the device's minimum block size (e.g., 4KB in ZN540). This significantly increases the volume of writes, particularly affecting small-sized write workloads. For instance, a 4KB write request generates a 4KB PP block and its 4KB metadata header block, reaching a write amplification factor of three. In our evaluation, we also observed that RAIZN reaches a WAF of 2.44 due to the overhead introduced by PP-related blocks when running the VARMAIL workload on F2FS, which primarily generates small-size writes to the RAID array.

### 3.3 Limitation of ZNS-compatible Scheduler

In environments with multiple I/O queues, the sequential submission of write requests by applications does not guarantee their sequential dispatch to the device queue [36]. This discrepancy raises the potential for request failures in normal ZNS zones, necessitating scheduler intervention.

The `mq-deadline` scheduler, the only ZNS-compatible scheduler in the latest Linux Kernel, resolves this challenge through the implementation of per-zone locks [2, 5]. Each write holds the lock for its target zone at dispatch and releases it upon completion, preventing other writes to the same zone during this interval. This mechanism limits the effective per-zone write queue depth to one, severely hurting performance. Conversely, the ordering constraints are relaxed within the ZRWA, eliminating the need for ZNS-compatible scheduler enforcement. This allows for the use of a generic scheduler such as `no-op`, enabling operations at higher queue depths within a single zone.

Existing ZNS RAID systems, comprised of normal zones, inherit the limitations of ZNS-compatible schedulers. However, simply switching to ZRWA-enabled zones and employing the `no-op` scheduler without adjusting the original structure is insufficient. Generic schedulers do not consider the ZRWA range when dispatching commands, leading to potential write failures due to the random order of dispatched requests. Specifically, if a request for a higher address that ends within the IZFR is dispatched first, it can trigger an implicit ZRWA flush, advancing the WP (see §2.3). A subsequent request for a lower address than the advanced WP would then fail. We have frequently encountered write failures when using the `no-op` scheduler without properly managing the request range in ZRWA-enabled zones.

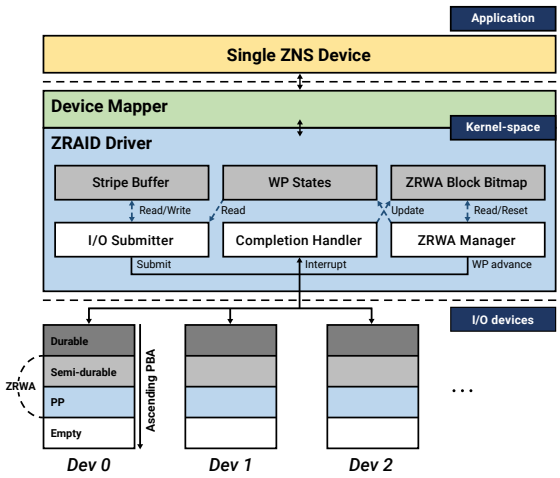



**Figure 2. Overall architecture of ZRAID.**

### 3.4 Complex Recovery for Write Atomicity

Sudden power loss in ZNS RAID systems can lead to scenarios where only part of a multi-chunk write operation completes successfully, leaving the remainder incomplete. If the number of failed chunks surpasses the stripe's recovery capabilities, the successfully completed portion of the write operation should be rolled back. However, due to the WP in ZNS SSDs being non-retrogressive, rolling back is not feasible. The only alternative for rollback is zone-level migration, which is highly impractical in such situations.

RAIZN [23] addresses this issue through space redirection. During recovery, if inconsistencies among WPs are identified, the space corresponding to the *protruded* WP is allocated in the *superblock* zone. A mapping to this space is then stored in both the memory cache and the superblock zone, redirecting subsequent read/write operations to the newly allocated area in the superblock zone. This recovery method effectively resolves the write atomicity challenge in the short term. However, over time, cumulative redirections can complicate the garbage collection process and potentially create an I/O bottleneck within the superblock zone.

## 4 Design

ZRAID presents a single zoned device abstraction to the host, identical to RAIZN. This ensures that all ZNS-compatible applications can function on ZRAID without any alterations. Utilizing the Linux device mapper (`dm`) framework, ZRAID sets up a logical block device visible to the host and routes logical requests from the host to the ZRAID module.

This section outlines the fundamental operation of two pivotal mechanisms in ZRAID: the PP placement logic for a write request and the advancement rules for WPs. The handling of several corner cases is detailed in Section 5.

### 4.1 Overall Architecture of ZRAID

ZRAID utilizes ZRWA-enabled zones where data chunks are initially placed in the ZRWA. These chunks are later

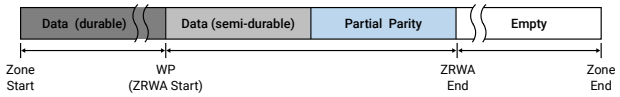



**Figure 3. Layout of a single physical zone.**

made immutable through the WP advancement, triggered by the explicit ZRWA flush command. Partial parity blocks are placed ahead of the data chunks within the ZRWA and are subsequently overwritten by the data chunks from future writes.

The overall architecture of ZRAID is illustrated in Figure 2. For simplicity, only one of the multiple physical zones in each device is depicted. The white boxes within the ZRAID driver represent core functional components of ZRAID. The grey boxes signify the key in-memory structures utilized by those components.

The role of the *I/O submitter* is to generate sub-I/Os from host requests and submit them simultaneously to the I/O devices. Here, sub-I/Os refer to the physical I/Os derived from a single logical I/O request, including data, parity, and metadata. The *I/O submitter* caches data chunks within an incomplete stripe in the in-memory *Stripe buffer*, facilitating the generation of both partial and full parity chunks. Additionally, the *I/O submitter* ensures that all data and PP sub-I/Os remain within their respective regions in the ZRWA by checking the *WP states* and delaying sub-I/Os submissions until the necessary conditions are satisfied.

The primary function of the *Completion handler* is to aggregate the completions of all sub-I/Os and return the completion of the original `bio` request back to the host, similar to the mechanism used in RAIZN. In addition, ZRAID updates the *ZRWA block bitmap* with the logical blocks included in the completed write and requests the corresponding WP advancement to the *ZRWA manager*. Each logical block within the ZRWA is represented by a single bit in the *ZRWA block bitmap*, which occupies 128 bytes of in-memory space in a five-ZN540 array setup.

The *ZRWA manager*, informed by the *ZRWA block bitmap*, tracks the completion progress within the data area in ZRWA. Once all logical blocks in the requested write and its predecessors are confirmed as completed, the *ZRWA manager* advances the WPs using an explicit ZRWA flush command, following the WP advancement rules (§4.4). The *WP states* are updated for the *I/O submitter* upon each WP advancement.

### 4.2 Partial Parity Placement in ZRWA

Figure 3 depicts the layout of a physical zone in a ZNS SSD under ZRAID. The space between the start of a zone and the current WP is occupied by durable data that can be recovered in the event of a failure. The ZRWA region, beginning from the WP, contains semi-durable data chunks and PP chunks. These chunks are not yet durable and are subject to rollback upon failure. The data chunks in the ZRWA are awaiting the completion of other sub-I/Os of the associated request or the advancement of the WP by background jobs (§4.4) to ensure durability.

To explain ZRAID's operations, we first define the following notations using a RAID-5 system consisting of $N$ devices as an example. In this case, each stripe is composed of $N$ chunks, comprising $N-1$ data chunks and one (full) parity chunk. Let $C_{start}(X)$ and $C_{end}(X)$ denote the *logical* start and end chunk number for a write operation $X$, respectively, within the ZNS RAID device exposed to the host. We call $Str(\mathrm{c}) = c \ / \ (N-1)$ the logical stripe number of a chunk $c$.

For simplicity, we assume that each device contains only one zone. Let *Dev*(c) and *Offset*(c) indicate the disk number and the offset in chunk units within a physical zone, respectively, where a chunk $c$ is written. In the RAID-5 configuration, the placement of the full parity chunk is distributed in a round-robin fashion across the devices. Therefore, the chunks in a full-stripe write $F$ are written to the disks, starting from the device number $Dev(C_{start}(F)) = Str(C_{start}(F)) \ \% \ N + C_{start}(F) \ \% \ (N-1)$ and continuing to the next devices in sequence, wrapping around to the beginning of the array if necessary. The full parity of $F$, $P_F$, is written to the device number $Dev(P_F) = (Str(C_{end}(F)) + N - 1) \ \% \ N$.

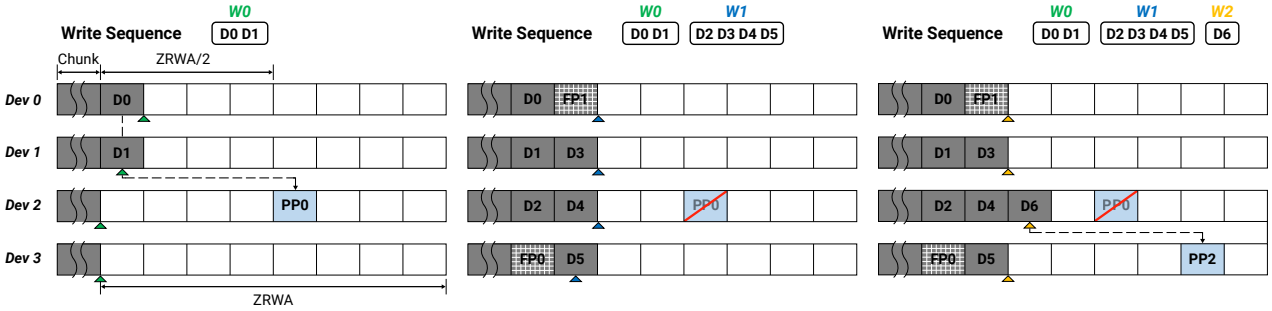
For a partial-stripe write $P$, where $(C_{end}(P)+1) \ \% \ (N-1) \neq 0$, each data chunk is written to the same location as it would be in a full-stripe write[1]. The location of the partial parity chunk of $P$, $P_P$, is determined by the following two static rules, where $N_{zrwa}$ represents the size of the ZRWA in chunks. Note that the partial parity is written in the back half of the ZRWA area, offset by $N_{zrwa}/2$ from its original intended location. This enables concurrent sharing of the ZRWA between data and PP chunks across multiple active stripes, allowing them to be handled simultaneously.

**Rule 1:** Placement of PP, $P_P$, for a partial-stripe write $P$

- $Dev(P_P) = (Dev(C_{end}(P)) + 1) \ \% \ N$
- $Offset(P_P) = Str(C_{end}(P)) \ + \ N_{zrwa}/2$

Figure 4 illustrates how the placement rule of PP works in a RAID-5 configuration with four ZNS devices. Each square represents a chunk, while the triangles beneath the squares represent the WP of each device, which will be further described in §4.4. In this example, write requests $W0$, $W1$, and $W2$ are submitted, and the three subfigures respectively depict the state after each request is processed. The data chunks comprising each request are shown at the top of the figure. It is assumed that the stripe prior to $W0$ is in a full state, with the WPs located at the end of that stripe. The device number where the first write starts is set to 0, and all write operations are assumed to be aligned to the chunk size, with $N_{zrwa}$ set to 8 chunks.

[1]For brevity, we only consider a partial stripe that does not span multiple stripes here. In general, a single write request can complete more than one full stripe, leaving at most one partial stripe.




**(a)** After writing W0 **(b)** After writing W1 **(c)** After writing W2

**Figure 4. ZRAID operation example in RAID-5 configuration.**

The first write, $W0$, forms a partial stripe that necessitates the PP, $PP0$. Given that $Dev(C_{end}(W0)) = 1$, **Rule 1** dictates that $Dev(PP0) = 2$ and $Offset(PP0) = Str(C_{end}(W0)) + 8/2 = 4$. Therefore, the position of $PP0$ is determined to be chunk number 4 on device 2, while the content is derived from XORing $D0$ and $D1$. The generated sub-I/Os for writing $D0$, $D1$, and $PP0$ are simultaneously submitted to their respective devices, by the *I/O submitter*.

The following write $W1$ contains four data chunks spanning stripes 0 and 1. $W1$ is completed when both stripes are fully populated by writing full parities $FP0$ and $FP1$ to devices 3 and 0, respectively, along with the data chunks $D2$~$D5$. As $W1$ completes two full stripes, all of their data chunks are safeguarded by full parities, eliminating the need for partial parities.

However, $W2$, consisting of a single chunk, does not complete a full stripe, mandating the writing of $PP2$ to device 3 as dictated by **Rule 1**. In this case, $PP2$ is associated only with the single chunk $D6$, thus its content is identical to $D6$.

Managing chunk-unaligned writes follows the outlined procedure, with the locations of the PP chunks determined in the same way. The PP blocks are written in block size units (e.g., 4KB) similar to data blocks, maintaining the same in-chunk offset as the data chunks. For instance, during repetitive 4KB sequential writes, the PP block is also written in 4KB size, with the in-chunk offset increasing by 4KB for each write. When such writes complete a data chunk, the PP also occupies a full chunk.

Three key points emerge from the PP placement logic. First, a PP chunk precedes any data chunk in the associated partial stripe, thereby ensuring no device is shared between them. This guarantees tolerance against the event of a single device failure. Second, in a RAID-5 setup, the rotational nature of the array sequence ensures that PP chunks are evenly distributed across all devices. Third, writing the last data chunk in a stripe or the full parity chunk does not generate a PP chunk. This indicates that the PP chunk locations in the ZRWA corresponding to these two chunks are never utilized for PP chunks. Hence, these spaces are reserved for metadata areas to handle corner cases (see §5.1 and §5.3).

It should be noted that, for a data chunk and its corresponding PP chunk to be placed concurrently, at least two chunks must fit within the ZRWA range. This creates a hardware requirement for ZRAID, where the ZRWA size for each device needs to be at least twice the size of a chunk. However, since the chunk size in RAID is configurable, this is not an overly restrictive requirement.

### 4.3 Benefits of Partial Parity Logging in ZRWA

ZRAID's in-place partial parity logging within the ZRWA effectively mitigates the *partial parity tax* in three key aspects.

First, ZRAID distributes PP chunks evenly across the same number of zones as data chunks by storing them within their originating data zones. This prevents bottlenecks that can arise in existing ZNS RAID systems where PP chunks are placed in dedicated zones, as discussed in §3.1.

Second, ZRAID locates PP chunks based on the static relationship with the corresponding partial stripes (§4.2), eliminating the need for additional metadata. In contrast, when PP chunks are placed in separate zones, location metadata becomes necessary. ZRAID reduces these collateral writes, which can be substantial, as described in §3.2.

Finally, ZRAID stores PP chunks temporarily in the ZRWA until the corresponding partial stripe is promoted to a full stripe, after which the PP chunk is overwritten by another data chunk. This approach offers significant benefits in both space and time. By avoiding persistent storage of PP chunks in dedicated zones, ZRAID reduces unnecessary space usage and eliminates the need for garbage collection in those zones. It also frees up active zone resources previously reserved for PP storage, making them available for data zones. Furthermore, since the ZRWA is typically implemented using fast, high-endurance DRAM or SLC flash memory, ZRAID enhances write performance and extends device lifespan.

### 4.4 Advancement of Write Pointers

ZRAID allows the use of a general scheduler, such as `no-op`, because it writes data chunks and parity chunks in the ZRWA. However, this means that sequentiality between sub-I/Os is not guaranteed, which can lead to trailing sub-I/Os causing

an implicit flush before the submission of preceding sub-I/Os, resulting in write failures (see §3.3). To prevent this, the *I/O submitter* restricts sub-I/Os to be submitted only within the ZRWA region. Consequently, an explicit WP advancement mechanism is required to push forward the ZRWA to process the next writes. One way to achieve this is by advancing the WPs of all devices whenever a full stripe is completed.

Meanwhile, the logical WPs reported to the host upon power recovery indicate the write progress before the failure [21, 25]. In RAIZN, additional metadata headers are recorded in dedicated PP zones to provide information about the range of each write before the power failure. Since the completion status of sub-I/Os for each write can be verified through the WP of each device, the progress of a partial stripe, if any, can be restored.

In contrast, ZRAID does not record any metadata during normal writes, so the WPs are the sole information available during power recovery. Therefore, if the WPs are advanced only on a full stripe, the information on any ongoing partial stripe will be lost. ZRAID employs a sophisticated WP advancement scheme to address this issue without requiring additional metadata.

The key problem is to determine the exact location of the last chunk $C_{end}(W)$, where $W$ represents the latest durable write (whether it is a full-stripe or partial-stripe write) successfully completed before the power failure. ZRAID utilizes the WP locations of two devices, namely $Dev(C_{end}(W))$ and $Dev(C_{end}(W)$ - 1), as checkpoints for the completed write $W$. Upon completion of all sub-I/Os within $W$, the *Completion handler* requests the WP advancement of $Dev(C_{end}(W))$ and $Dev(C_{end}(W)$ - 1). The background task, the *ZRWA manager*, then awakens and waits until all writes preceding $W$ have completed, while checking the *ZRWA block bitmap*.

The WP advancement by the *ZRWA manager* is divided into two distinct steps within a chunk $c$, with each step indicating the relative position between the chunk $c$ and $C_{end}(W)$. Let $WP(d)$ represent the WP for the disk number $d$ in chunk units. If $WP(Dev(\text{c}))$ is equal to $Offset(\text{c}) + 0.5$, it shows that the chunk $c$ is the last chunk of $W$. Alternatively, if $WP(Dev(\text{c}))$ is equal to $Offset(\text{c}) + 1$, the last chunk of $W$ is located in the next chunk, $C + 1$. The reason ZRAID uses the WPs of the last two devices is due to the possibility of concurrent device and power failures. In such cases, the WP of the failed device becomes unreadable, thereby necessitating a second WP location as a backup.

To ensure that the WPs of the two devices reach these states, the WPs of those devices are advanced according to the following rules for the given write $W$:

**Rule 2**: Advancement of WPs for a write $W$

- $WP(Dev((C_{end}(W))) = Offset(C_{end}(W)) + 0.5$
- $WP(Dev((C_{end}(W) - 1)) = Offset(C_{end}(W) - 1) + 1$

Note that, upon completion of a full stripe, the *ZRWA manager* automatically advances the lagging WPs of all devices. When performing this advancement on a full stripe $F$, the last two data chunks, located at $C_{end}(F)$ and $C_{end}(F)$ - 1, are advanced first, followed by the remaining data chunks. This prevents the full stripe from being mistakenly identified as a partial stripe during crash recovery.

Referring back to Figure 4, we take a closer look at the WP advancement process as each write request proceeds. After completing all sub-I/Os of $W0$, since $C_{end}(W0)$ is 1, $WP(1)$ is advanced to $Offset$(D1) + 0.5, while $WP(0)$ is concurrently advanced to $Offset$(D0) + 1. The WP positions after $W0$ are denoted by green triangles.

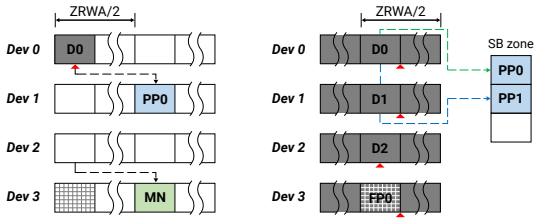
Similarly, when $W1$ completes, $WP(3)$ advances to $Offset$(D5) + 0.5, and $WP(2)$ to $Offset$(D4) + 1. This advancement shifts both stripes 0 and 1 into the durable data region. Once a stripe is completed, the *ZRWA manager* advances any lagging WPs—$WP(0)$ and $WP(1)$ in this case—to the same position as $WP(2)$, as indicated by blue triangles.

A single-chunk write $W2$ leads to the advancement of $WP(2)$ and $WP(3)$. Since $C_{end}(W2)$ - 1 is 5 ($D5$), $WP(3)$ advances to $Offset$(D5) + 1, as dictated by **Rule 2**. $W2$ serves as an example of how the WP advancement for the first chunk of the stripe is handled. In this case, the WP of the last data chunk of the previous stripe ($Str(C_{end}(W2))$ - 1) is advanced. The final WP positions after $W2$ are denoted by yellow triangles.

Due to the necessity for the WP to advance in two steps within a chunk, the chunk size must be set at least twice the ZRWA flush granularity (ZRWAFG). This constraint, in conjunction with the hardware requirement outlined in §4.2, enforces that the size of ZRWA must be at least four times the ZRWAFG. ZN540 devices meet these requirements with a ZRWA size of 1MB and a 16KB ZRWAFG. However, PM1731a, a small-zone device with a 64KB ZRWA size and a 32KB ZRWAFG, does not fully meet these criteria. ZRAID can still utilize this device by aggregating multiple physical zones into a larger zone, which is also beneficial for enhancing performance (see §6.5).

### 4.5 Crash Recovery using Write Pointers

During recovery, ZRAID reads the WPs of all devices to identify the most recent consistent state before the failure. As an example, consider a situation where power and device 2 fail simultaneously after $W2$ is written as shown in Figure 4(c). Given that $WP(2)$ is unreadable, ZRAID tries to find the most recent write $X$, which was $W2$, from the remaining WPs. $WP(0)$, $WP(1)$, and $WP(3)$ indicate that $D5$, $D4$, and $D6$ are the last chunks of the most recently completed write $X$, respectively. Because $D6$ is at the last position in the array, ZRAID can determine that $C_{end}(X)$ before the crash was $D6$. The location of the corresponding PP chunk ($PP2$) can be statically determined by **Rule 1**. $PP2$ is then used to reconstruct the chunk $D6$, which was lost due to the failure.




**(a)** The first chunk **(b)** Near the last stripe

**Figure 5. Corner cases and their handling.**

ZRAID advances the WPs only after all sub-I/Os in a write are completed. For partially completed writes, the WPs do not advance for the successful chunks, which thus remain in the ZRWA. During recovery, these blocks are reused for subsequent writes. This simple rollback-based recovery solves the complexity of supporting write atomicity in existing ZNS RAID systems, as mentioned in §3.4.

# 5 Operation Details

## 5.1 The First Chunk

One corner case where the WP advancement rule (§4.4) does not apply is when writing to the first chunk of a logical zone. While ZRAID advances $WP(Dev(C_{end}(P)))$ and $WP(Dev(C_{end}(P)$ - 1)) during a single-chunk write request $P$, the first chunk of a zone has no previous chunk, rendering the rule inapplicable. For such cases, a 4KB-sized *magic number* block is recorded in the reserved PP region mentioned in 4.2. This block contains a predefined unique pattern that can be identified during the recovery process.

Figure 5(a) illustrates the situation where a single data chunk $D0$ and its PP chunk $PP0$ are written. After writing these chunks, the *ZRWA manager* advances $WP(0)$ and writes the additional magic number block. The location of this magic number block is determined by applying Rule 1 to the last data chunk of the stripe, as depicted by *MN* in the figure. During recovery, ZRAID checks the magic number block if device 0 fails and the remaining WPs are all zeroes. If the magic number is found, it indicates that the first chunk $D0$ has been written, and $PP0$ is used to reconstruct $D0$.

## 5.2 Near the Last Stripe

As the WPs advance, the distance between the active stripe and the end of a zone may become smaller than the default data-to-PP distance set by ZRAID, which is half the ZRWA size. In such cases, ZRAID falls back to the method used in RAIZN, logging PP chunks in a reserved zone. This situation is illustrated in Figure 5(b). ZRAID scans the reserved zone during recovery if $N_{zone} - WP(d) < N_{zrwa}/2$ for any device $d$, where $N_{zone}$ represents the size of the physical zone in chunks.

This corner case is relatively rare, constituting only 0.093% of occurrences in an array based on ZN540 devices. For this reason, ZRAID does not allocate a separate zone exclusively for this purpose. Instead, those partial parity chunks are logged in the *superblock (SB)* zone, which is used for storing array-wide metadata. Also, ZRAID provides a configurable option that can dynamically adjust the data-to-PP distance to further reduce the amount of PP chunks written into the superblock zone.

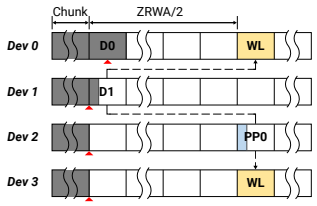



**Figure 6. Handling chunk-unaligned flush.**

## 5.3 Chunk-unaligned Flush

On a flush request, ZRAID advances the WPs as described in §4.4 to guarantee both the durability of data and its consistency with the WPs, similar to the assurances offered by a ZNS device [23]. However, since the WPs only point to locations at the chunk level in ZRAID, it is difficult to guarantee consistency for data that is not aligned with the chunk size.

To handle flush operations concerning these chunk-unaligned writes, ZRAID uses a special *WP logging* technique. ZRAID stores two identical WP logs on different devices to ensure durability against a single device failure. The WP logs utilize the first and last chunk in the PP stripe, which are not occupied by PP blocks as noted in §4.2. Each WP log entry, sized at 4KB, comprises a logical address of the latest durable write and a timestamp.

Fig. 6 shows a situation where a flush operation is invoked after writing the entire $D0$ chunk and a portion of $D1$. During recovery, ZRAID scans the WP log blocks labeled as *WL* and identifies the most recent entry based on the timestamp. The write progress obtained from this entry is then compared to the progress calculated using the WP states, as outlined in §4.5, and the greater of the two is chosen to ensure data consistency.

# 6 Evaluation

## 6.1 Experimental Setup

All experiments were conducted on a server equipped with one 40-core Intel Xeon Silver 4114 CPU running at 2.20GHz and 352GB of memory. The server runs Ubuntu version 20.04 with a Linux kernel 5.15.0. We utilized five Western Digital Ultrastar DC ZN540 [3] 1TB ZNS SSDs. They are configured in a RAID-5 array with a chunk size of 64KB and a stripe size of 256KB.

ZRAID was implemented by modifying the code of RAIZN, which is publicly available on Github [22]. However, in our attempts to reproduce the results of RAIZN as published in the paper, the performance numbers of the code were far below those reported. We identified that the single FIFO structure used to dispatch tasks to the I/O workqueue was a

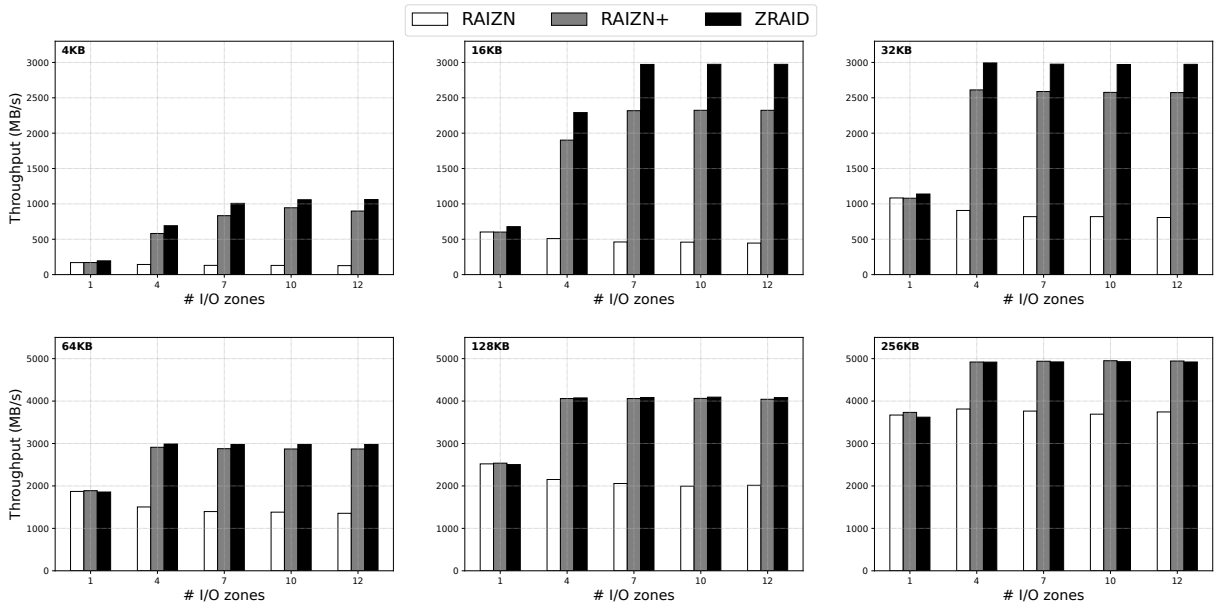



**Figure 7. `fio` write throughput over different request sizes.**

bottleneck, and to resolve this issue, we modified it to multiple FIFO structures. In subsequent experimental results, "RAIZN" denotes the original RAIZN code, while "RAIZN+" refers to the version we modified to fix the single FIFO queue bottleneck. RAIZN+ demonstrates performance that aligns more closely with the results presented in the paper compared to RAIZN. When comparing the FIO results, RAIZN+ performs approximately 20% lower than the reported results at a 64KB block size, 20% higher at 256KB, and shows similar performance at 4KB.

### 6.2 FIO Performance

First, we conducted sequential write tests using `fio` [13] benchmark version 3.36 with the `libaio` [28] ioengine. We used direct I/O and set the queue depth to 64 for all experiments. In `fio`'s zoned mode, individual threads are assigned to perform sequential writes to their specific dedicated open zones. Note that ZRAID's read path is identical to RAIZN's. Consequently, `fio`'s read performance shows no difference between the two, and these results are therefore omitted.

Figure 7 compares the sequential write performance among RAIZN, RAIZN+, and ZRAID, varying the request size from 4KB to 256KB (except for 8KB, which is available in Figure 8). In our environment, the maximum write throughput for a single ZN540 device is 1230MB/s, resulting in a cumulative maximum throughput of 6150MB/s for a 5-device array. However, the additional writes of partial or full parities impose an upper limit on ZNS RAID performance. For instance, with 128KB writes, one partial parity chunk and one full parity chunk are generated per stripe, resulting in an upper throughput limit of 4100MB/s. Similarly, writes up to and including 64KB are saturated at 3075MB/s, while the limit for 256KB writes is 4920MB/s.

When the request size is 64KB or smaller, ZRAID consistently outperforms RAIZN+, with an average improvement of 18.1%. With a request size of 16KB, ZRAID's write performance begins to saturate with seven I/O zones. In contrast, RAIZN+ fails to fully utilize the array's performance potential, even with a 32KB request size, due to the *partial parity tax* and limitations in the scheduler.

With request sizes of 64KB and 128KB, both ZRAID and RAIZN+ reach the upper throughput limit, thereby showing no significant difference in performance. However, due to the presence of the PP metadata header, RAIZN+ exhibits a slightly higher Write Amplification Factor (WAF) compared to ZRAID, accounting for the minor performance difference between the two.

For 256KB request sizes, ZRAID's performance is slightly lower than RAIZN+ by an average of 0.86% due to the synchronization overhead between the *I/O submitter* and the *ZRWA manager*. This is the worst-case scenario for ZRAID, as every write is stripe-aligned, which is unrealistic; however, the degradation is negligible. RAIZN consistently shows lower performance and faces a scalability issue where throughput decreases as the number of I/O zones increases, as seen in the graph.

### 6.3 Factor Analysis

To analyze how much each enhancement made by ZRAID contributes to the overall performance, we developed several variations of ZRAID by incrementally applying each one to RAIZN+. Figure 8 compares the `fio` sequential write throughput for an 8KB request size across these variations.

**ZRWA** (or simply **Z**) denotes the version that introduces ZRWA-enabled zones to RAIZN+ in place of normal zones. **ZRWA** demonstrates lower performance than RAIZN+ in

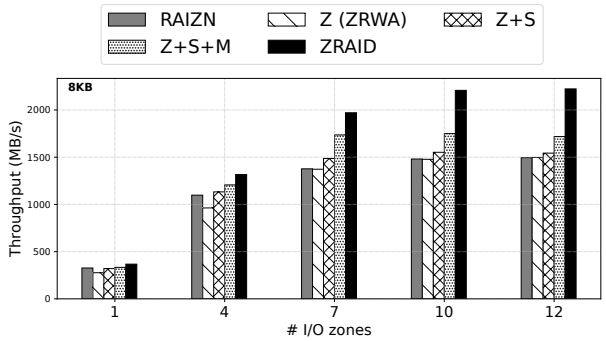



**Figure 8. Throughput comparison for `fio` 8KB writes across ZRAID variants.**

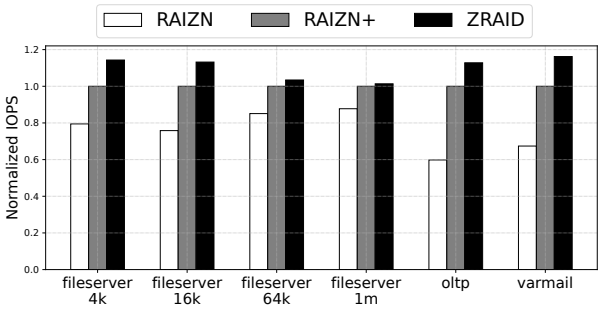



**Figure 9. Throughput of `filebench` workloads.**

some cases due to the synchronization overhead. **Z+S** replaces the `mq-deadline` I/O **S**cheduler in **Z** with the `no-op` scheduler which supports high queue depths. **Z+S** shows an average performance increase of about 10% over **Z**, addressing the throughput loss caused by the `mq-deadline` scheduler mentioned in §3.3. **Z+S+M** further removes the **M**etadata header for PP chunks from **Z+S**. PP metadata header blocks amplify the write amount by about 19% in the 8KB `fio` workload. By eliminating these blocks, **Z+S+M** outperforms **Z+S** by an average of 10.3%.

Finally, **Z+S+M+P**, which is equivalent to ZRAID, stores **P**artial-parity chunks in data zones instead of dedicated zones unlike **Z+S+M**. **Z+S+M+P**, or ZRAID, improves average performance by 17.7% over **Z+S+M** due to the elimination of contention in PP zones. This has a significant impact with more write zones; with 12 open zones, ZRAID exhibits up to a 30% performance improvement over **Z+S+M**. The performance difference between ZRAID and **Z+S+M** supports the claim in §3.1 that contention in PP zones is a major problem.

By leveraging the aforementioned enhancements, ZRAID achieves an average performance improvement of 34.7% over RAIZN+ for 8KB sequential writes, with the improvement reaching up to 48% in the experiment with 12 open zones.

### 6.4 File System Benchmarks

To assess ZRAID's performance with realistic workloads, we ran `filebench` [34] and `db-bench` [18] over F2FS [24] and ZenFS [21], respectively.

**Filebench.** To conduct `filebench` tests, we used F2FS, a representative ZNS-compatible file system in the Linux kernel. F2FS requires a conventional, randomly writable zone or device to be designated as the metadata area when operating in zoned mode. We provided a 32GB partition on a separate conventional SSD as the metadata area for the 4TB ZNS RAID array.

We selected FILESERVER (write-heavy), OLTP, and VARMAIL (small random read/write) for our tests. To highlight the performance of the underlying ZNS RAID system, all file operations in the workloads, except for VARMAIL, were executed as direct I/Os. The number of files for each workload was increased to 40 times the default settings to ensure an adequate working set size. For the FILESERVER workload, the *iosize* parameter was varied from 4KB to 1MB to demonstrate the effect of the write size. For the OLTP workload, the *iosize* was changed from 2KB to 4KB (the block size of the ZN540) to meet the direct I/O restriction. All other parameters were kept at their default settings.

Figure 9 illustrates the IOPS comparisons among RAIZN, RAIZN+, and ZRAID, with results normalized to RAIZN+. In the FILESERVER workload with a 4KB iosize, ZRAID achieves a throughput improvement of 14% over RAIZN+. However, at a 1MB iosize, the influence of PP overhead diminishes, resulting in ZRAID's performance being nearly identical to RAIZN+. Smaller write sizes increase the PP-to-data ratio, thereby amplifying ZRAID's benefits. In the OLTP and VARMAIL workloads, ZRAID shows performance improvements of 12.8% and 16.2% over RAIZN+, respectively. The write sizes of these workloads are mostly smaller than 16KB, creating an environment well-suited for ZRAID's optimizations.

Note that F2FS, without specific hints, does not perform hot/cold separation and logs all data into a single zone [2, 24]. Typically, it maintains only two simultaneously active zones, one for the data blocks and the other for its node blocks. Consequently, the impact of PP zone contention is not as significant as when using many zones with `fio`, leading to a reduced performance benefit for ZRAID.

**db_bench.** We used a suite of benchmarks from RocksDB [19] to evaluate the performance differences in database applications among ZRAID and its variations described in §6.3. ZenFS is a plugin that provides a file system interface for RocksDB on ZNS devices. ZenFS utilizes the maximum number of active zones provided by the device to achieve hot/cold separation, switching their states between open and closed. Although the number of simultaneously open zones is not significantly different from F2FS, the use of multiple active zones enables greater parallelism. To take advantage of this and assess ZRAID's portability across different file systems, we incorporated ZenFS in our RocksDB experiments.

The FILLSEQ, FILLRANDOM, and OVERWRITE workloads provided by the `db_bench` [18] tool were used for our experiments. Each workload has ten million keys and operations with a value size of 8000 bytes. The RocksDB environment

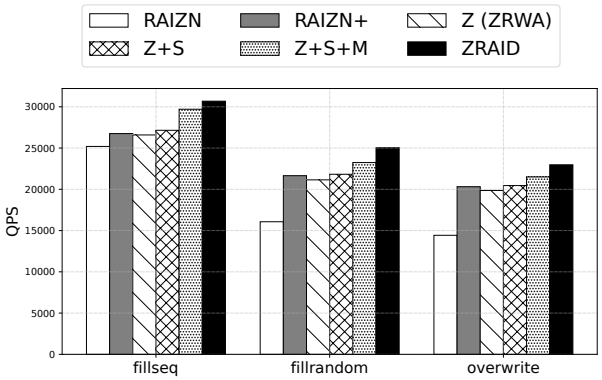



**Figure 10. Throughput comparison of ZRAID variants using `db_bench` workloads.**

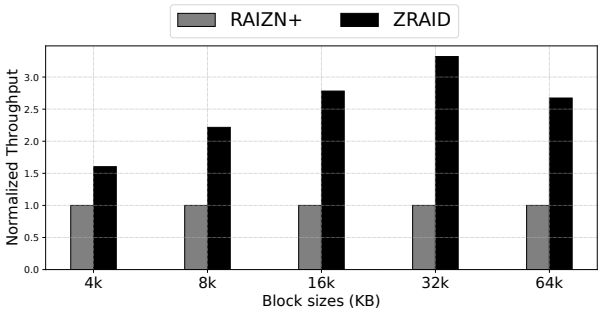



**Figure 11. Throughput of `fio` on PM1731a with 15 opened zones.**

was configured with four worker threads, and the number of background jobs was set to 16 for both flush and compaction. To circumvent the effect of the page cache in all experiments, we used the flags `-sync`, `-use_direct_io_for_flush_and_compaction` and `-use_direct_reads`.

Figure 10 summarizes the results of these workloads. ZRAID shows an average performance improvement of 14.5% over RAIZN+, with contributions from each ZRAID enhancement similar to those observed in the 8KB `fio` write workload in Figure 8. As discussed in §4.3, ZRAID reallocates the physical active zones that were previously reserved for dedicated PP zones to data zones, effectively increasing the logical active zone limit available to the host. This allows ZenFS to exploit greater parallelism and more granular hot/cold data separation, uncovering additional advantages of ZRAID that were not observed in prior experiments.

We also measured RAID internal statistics related to PP chunks and garbage collections (GCs) during `db_bench`. In ZRAID, all PP blocks were overwritten within the ZRWA except for the corner cases mentioned in §5.2, As a result, ZRAID exhibits a flash WAF of just 1.25 due only to full parity chunks. In contrast, RAIZN+ has an average WAF of 1.6 across the three workloads, reaching up to 2 in the FILLSEQ workload. Specifically, FILLSEQ submits approximately 130GB of data writes to the array. RAIZN+ generates 98GB of permanently logged PP blocks, while ZRAID produces only 26 MB of permanent PP blocks and 65GB of temporary PP blocks.

Due to the permanent storage required for PP blocks, RAIZN+ experienced a total of 345 GC operations and subsequent zone reset operations for dedicated PP zones during the three workloads, while ZRAID did not perform any GC operations. Furthermore, by eliminating the PP metadata headers, ZRAID reduced the amount of PP-related writes by an average of 24% compared to RAIZN+.

### 6.5 Performance on DRAM-based ZRWA Device

We used `fio` to compare the performance of sequential writes to normal zones and the ZRWA on ZNS devices. On ZN540, the performance was nearly identical. However, on PM1731a, writes to the ZRWA were found to be 26.6 times faster. This suggests that the ZRWA area on the PM1731a device is configured with battery-backed DRAM.

We applied ZRAID on the PM1731a device to assess the impact of eliminating unnecessary flash writes caused by outdated partial parity. We aggregated four small zones to overcome the hardware limitations of PM1731a (see §4.4). Sub-I/Os are internally interleaved across aggregated zones, with each zone using an aggregation chunk size of 64KB, which matches the ZRWA size. Unfortunately, we have only one PM1731a device, so five equal-sized `dm-linear` partitions are used for emulating five distinct devices. Each partition contains 8000 consecutive zones, and the addresses of ZRWA explicit commands for each partition are converted and sent to the original PM1731a device, while I/O requests are forwarded to the `dm` layer. For a fair comparison, RAIZN+ was also configured to utilize the aggregated zones.

Figure 11 presents the normalized throughput of `fio` configured with 15 open zones, varying the request size from 4KB to 64KB. In RAIZN+, all PP blocks are persistently stored on flash, consuming the flash channel bandwidth required for data blocks. On the other hand, in ZRAID, PP blocks are overwritten by subsequent data blocks in the ZRWA, allowing the flash channels to be fully utilized for writing data blocks. This difference results in up to a 3.3x throughput improvement.

Based on these results, we anticipate that on devices where the ZRWA is implemented with high-performance media such as DRAM, the performance gains of ZRAID would be significantly greater than those observed with ZN540 devices.

### 6.6 Crash Consistency

To verify ZRAID's correctness against crashes, we conducted fault injection tests using QEMU. We ran a synthetic workload that issues sequential writes with the FUA flag set, with random sizes ranging from 4KB to 512KB. The written data consists of a predefined repeating 7-byte pattern, which is not a divisor of the block size (4096 bytes). This pattern is filled using the byte address as an offset, allowing for verification of data correctness. After each successful write, the

| Consistency policy | Failure rate | Data loss |
|---|---|---|
| Stripe-based | 76% | 134.2 KB |
| Chunk-based | 53% | 32.5 KB |
| WP log | 0% | 0 KB |

**Table 1. Evaluation of consistency policies in ZRAID.**

end LBA, which must be preserved after a crash, is logged and redirected to the host machine. To simulate a power failure, QEMU was forcibly terminated at arbitrary moments during workload execution. Following the simulated crash, one device was reset to mimic a device failure, and ZRAID was reconfigured to initiate the recovery and rebuild process.

Recovery correctness is evaluated based on two key criteria: 1) Ensuring that the reported logical WP after recovery is greater than the last LBA in the redirected log, and 2) verifying the integrity of the predefined pattern within the range of the reported WP. If the first criterion is violated, the discrepancy between the logged WP and the reported WP is treated as data loss, which is then accumulated to compute the average data loss per failure.

Our experiment evaluated the effectiveness of three consistency policies: a baseline and two policies that progressively applied ZRAID's consistency techniques. Table 1 shows the results of 100 fault injection tests conducted for each policy. A failure was recorded if either of the correctness criteria was not met. However, in all cases, the pattern verification for the second criterion was successful, and every data rebuild was correctly executed. Therefore, the failures shown in the table only reflect violations of the first criterion, meaning the logged WP was not reported as durable after receovery.

In Table 1, the *Stripe-based* policy is the simplest approach, where the WP advances only when a full stripe is completed, serving as the baseline. This policy treats FUA-flagged writes the same as regular writes, without any special handling. It resulted in a 76% failure rate with an average data loss of 134.2KB per failure.

The *Chunk-based* policy enhances durability by securing chunk-level durability using ZRAID's *2-step WP advancement* technique, as described in §4.4. This policy also ignores the FUA flag, but it results in a 53% failure rate with an average data loss of 32.5KB, demonstrating a 4.1x reduction in data loss compared to the baseline.

The *WP log* policy employs *WP logging* for FUA handling in addition to the 2-step WP advancement, as outlined in §5.3. This approach recovered successfully in all 100 test cases, indicating that ZRAID functions correctly under crash scenarios.

### 6.7 Overhead of ZRWA Explicit Flush Command

To assess the overhead of WP advancement, we conducted an experiment involving repeated ZRWA explicit flushes on a ZRWA-enabled zone in the ZN540 device. The address assigned to each command was incremented by 32KB, continuing until the end of the zone. As a result, the average latency per command was 6.8µs, which is negligible compared to typical NAND write latency. In addition, WP advancement is performed in the background only after a chunk is completed, ensuring that its overhead does not interfere with the critical path.

## 7 Conclusion

ZRAID introduces a novel approach to ZNS RAID systems by leveraging the ZRWA feature. This innovative strategy addresses the significant challenge of managing partial parity for incomplete stripes, which has traditionally resulted in throughput bottlenecks and increased write amplification. By temporarily storing partial parity within the ZRWA of data zones, ZRAID effectively distributes these writes across multiple zones, thus eliminating the bottlenecks associated with reserving dedicated zones for partial parity. Additionally, this method ensures that expired partial parity is overwritten by subsequent data, avoiding unnecessary flash writes and enhancing the longevity of the SSDs.

ZRAID exhibits superior write performance across diverse environments compared to existing ZNS RAID solutions. Moreover, in devices where the ZRWA is implemented using DRAM, ZRAID's approach is anticipated to have a more favorable impact on both performance and flash endurance.

## Acknowledgements

We would like to thank our shepherd, Huaicheng Li, and the anonymous reviewers for their valuable feedback. We are also grateful to Matias Bjørling at Western Digital Corporation for his generous donation of the ZN540 devices. This work was supported by the National Research Foundation of Korea (NRF) grant (No. RS-2023-00222663), and the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant (No. IITP-2021-0-01363), funded by the Korean government (MSIT). This work was also supported in part by a research grant from Samsung Electronics.

# A Artifact Appendix

## A.1 Abstract

Our artifact contains all source files for ZRAID, as well as scripts to reproduce the evaluation results in the paper. It includes source files for comparison baselines (RAIZN) and application patches. This appendix describes how to access the artifact, the preprocessing steps required, and instructions for running experiments with the provided scripts.

## A.2 Artifact check-list (meta-information)

- **Run-time environment:** Linux kernel 5.15.0 (modified), Ubuntu 20.04.
- **Hardware:** Five Western Digital ZN540 devices (required), one Samsung PM1731a (optional).
- **Execution:** All experiments are run using Python scripts.
- **Metrics:** Throughput, recovery failure rate.
- **How much time is needed to complete experiments (approximately)?:** One repetition in Figures 7–11 takes 4.5 hours, five repetitions will require 22.5 hours. Table 1 requires an additional 25 hours.
- **Publicly available?:** Yes.
- **Code licenses?:** GPL.

## A.3 Description

**A.3.1 How to access.** The artifact is available on GitHub: https://github.com/kminuki/zraid-asplos25-ae.

**A.3.2 Hardware dependencies.** ZRAID assumes that all ZNS SSDs in the array are identical. Each ZNS SSD must have a ZRWA size at least four times larger than the ZRWA flush granularity (see §4.4). If this condition is not met, custom modifications are required.

**A.3.3 Software dependencies.** ZRAID relies on a modified NVMe driver to send ZRWA-related commands directly to the ZNS SSD. Compatible software versions and modifications are as follows:

- **Linux kernel:** 5.15.0 (modified to support zone specifications for ZN540 and PM1731a).
- **fio:** 3.36.
- **filebench:** 1.5-alpha3 (modified to support direct I/O).
- **rocksdb:** 8.3.2.
- **zenfs:** latest (modified to prevent exceeding the open zone limit).

## A.4 Installation

1. Clone the artifact.
2. Apply the patch from `patch/linux-5.15.0.patch` to the Linux v5.15 source code, then compile and reinstall the kernel.
3. Compile the modified NVMe driver in `nvme/` directory:
   `cd nvme; bash ./make.sh`
4. Clone fio, filebench, rocksdb, and zenfs from their GitHub repositories using:
   `bash ./script/setup-all.sh`
5. Apply patches to filebench and zenfs by running:
   `bash ./script/apply-patch.sh`
6. Set the application paths in the global configuration file:
   `vi eval/global.ini`
7. Build the ZRAID and RAIZN variant modules by executing:
   `cd eval; python3 make_modules.py`

## A.5 Experiment workflow

Enter the directory for the desired figure (e.g., *fig7*) in the paper and follow the instructions in the README. In each directory, `run.py` executes the experiment, and `collect.py` analyzes results and generates the corresponding figure. The only exception is *table1*, where the script prints results to the prompt upon completion.

## A.6 Evaluation and expected results

Running each experiment with five repetitions should produce Figures 7, 8, 9, 10, and 11 as shown in the paper. The experiment for Table 1 should yield results close to those reported in the paper, and the *WP Log* policy should perform without any failures.

## References


[1] nvme-cli: NVM-Express user space tooling for linux. https://github.com/linux-nvme/nvme-cli.

[2] The Linux Kernel Archives. https://git.kernel.org/pub/scm/linux/kernel/git/stable/.

[3] Ultrastar DC ZN540. https://www.westerndigital.com/en-kw/products/internal-drives/ultrastar-dc-zn540-nvme-ssd?sku=0TS2094.

[4] Western Digital and Toshiba talk up penta-level cell flash. https://blocksandfiles.com/2019/08/07/penta-level-cell-flash/.

[5] Zoned Namespaces (ZNS) SSDs. https://zonedstorage.io/.

[6] Initial Array Creation. https://raid.wiki.kernel.org/index.php/Initial_Array_Creation, 2008.

[7] Benefits of ZNS in datacenter storage systems. https://www.flashmemorysummit.com/Proceedings2019/08-06-Tuesday/20190806_ARCH-102-1_Chung.pdf, 2019.

[8] Open Compute Project NVMe HDD Specification Revision 0.7a. https://www.opencompute.org/documents/ocp-nvme-hdd-spec-rev0-7a-2022-11-21-docx-pdf, 2022.

[9] 2024. NVM Express® Zoned Namespace Command Set Specification Revision 1.2 August 5th, 2024. https://nvmexpress.org/wp-content/uploads/NVM-Express-Zoned-Namespace-Command-Set-Specification-Revision-1.2-2024.08.05-Ratified.pdf, 2024.

[10] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. RACS: a case for cloud storage diversity. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 229–240, 2010.

[11] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. Design tradeoffs for SSD performance. In *USENIX 2008 Annual Technical Conference*, ATC'08, page 57–70, USA, 2008. USENIX Association.

[12] Mike Allison. What's New in NVMe Technology: Ratified Technical Proposals to Enable the Future of Storage. https://nvmexpress.org/wp-content/uploads/Whats-New-in-NVMe-Technology-Ratified-Technical-Proposals-to-Enable-the-Future-of-Storage-1.pdf, 2023.

[13] Jens Axboe. fio: flexible I/O tester. https://github.com/axboe/fio.

[14] Hanyeoreum Bae, Jiseon Kim, Miryeong Kwon, and Myoungsoo Jung. What you can't forget: exploiting parallelism for zoned namespaces. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage 22)*, pages 79–85, 2022.

[15] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. ZNS: Avoiding the block interface tax for flash-based SSDs. In *2021 USENIX Annual Technical Conference (ATC 21)*, pages 689–703, 2021.

[16] Matias Bjørling. Zoned namespaces (ZNS) SSDs: Disrupting the storage industry. https://snia.org/sites/default/files/SDC/2020/075-Bj%C3%B8rling-Zoned-Namespaces-ZNS-SSDs.pdf, 2020.

[17] Samsung Electronics. Samsung Introduces Its First ZNS SSD With Maximized User Capacity and Enhanced Lifespan. https://news.samsung.com/global/samsung-introduces-its-first-zns-ssd-with-maximized-user-capacity-and-enhanced-lifespan.

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

[19] Facebook. Rocksdb. https://rocksdb.org.

[20] Javier González. Zoned Namespaces: Use Cases, Standard and Linux Ecosystem. In *SNIA Storage Developer's Conference SDC EMEA*, volume 20, 2020.

[21] Hans Holmberg. RocksDB: ZenFS Storage Backend. https://github.com/westerndigitalcorporation/zenfs/.

[22] Thomas Kim. RAIZN source code. https://github.com/ZonedStorage/RAIZN-release.

[23] Thomas Kim, Jekyeom Jeon, Nikhil Arora, Huaicheng Li, Michael Kaminsky, David G Andersen, Gregory R Ganger, George Amvrosiadis, and Matias Bjørling. RAIZN: Redundant Array of Independent Zoned Namespaces. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 23), Volume 2*, pages 660–673, 2023.

[24] 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.

[25] Euidong Lee, Ikjoon Son, and Jin-Soo Kim. An Efficient Order-Preserving Recovery for F2FS with ZNS SSD. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*, pages 116–122, 2023.

[26] Jing Li, Peng Li, Rebecca J Stones, Gang Wang, Zhongwei Li, and Xiaoguang Liu. Reliability equations for cloud storage systems with proactive fault tolerance. *IEEE Transactions on Dependable and Secure Computing*, 17(4):782–794, 2018.

[27] Umesh Maheshwari. From blocks to rocks: A natural extension of zoned namespaces. In *Proceedings of the 13th ACM Workshop on Hot Topics in Storage and File Systems*, pages 21–27, 2021.

[28] Jeff Moyer. libaio. https://pague.io/libaio.

[29] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, et al. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231, 2021.

[30] 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.

[31] Devashish Purandare, Pete Wilcox, Heiner Litz, and Shel Finkelstein. Append is near: Log-based data management on zns ssds. In *12th Annual Conference on Innovative Data Systems Research (CIDR 22)*., 2022.

[32] Dongjoo Seo, Ping-Xiang Chen, Huaicheng Li, Matias Bjørling, and Nikil Dutt. Is Garbage Collection Overhead Gone? Case study of F2FS on ZNS SSDs. 2023.

[33] Junyi Shu, Ruidong Zhu, Yun Ma, Gang Huang, Hong Mei, Xuanzhe Liu, and Xin Jin. Disaggregated RAID storage in modern datacenters. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 147–163, 2023.

[34] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41(1):6–12, 2016.

[35] Qiuping Wang and Patrick PC Lee. ZapRAID: Toward High-Performance RAID for ZNS SSDs via Zone Append. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems (ApSys 23)*, pages 24–29, 2023.

[36] Youjip Won, Jaemin Jung, Gyeongyeol Choi, Joontaek Oh, Seongbae Son, Jooyoung Hwang, and Sangyeun Cho. Barrier-enabled IO stack for flash storage. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 211–226, 2018.

[37] Suzhen Wu, Yanping Lin, Bo Mao, and Hong Jiang. GCaR: Garbage collection aware cache management with improved performance for flash-based SSDs. In *Proceedings of the 2016 International Conference on Supercomputing*, pages 1–12, 2016.

[38] Shiqin Yan, Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Andrew A Chien, and Haryadi S Gunawi. Tiny-tail flash: Near-perfect elimination of garbage collection tail latencies in NAND SSDs. *ACM Transactions on Storage (TOS)*, 13(3):1–26, 2017.