ZapRAID: Toward High-Performance RAID for ZNS SSDs via Zone Append

Qiuping Wang
The Chinese University of Hong Kong
Shatin, Hong Kong, China

Patrick P. C. Lee
The Chinese University of Hong Kong
Shatin, Hong Kong, China

ABSTRACT

Zoned Namespace (ZNS) provides the Zone Append primitive to boost the write performance of ZNS SSDs via intra-zone parallelism. However, making Zone Append effective for a RAID array of multiple ZNS SSDs is non-trivial, since Zone Append offloads address management to ZNS SSDs and requires hosts to dedicatedly manage RAID stripes across multiple drives. We propose ZapRAID, a high-performance software RAID layer for ZNS SSDs by carefully using Zone Append to achieve high write parallelism and lightweight stripe management. ZapRAID's core idea is a group-based data layout with coarse-grained ordering across multiple groups of stripes, such that it can use small-size metadata for stripe management on a per-group basis. Our prototype evaluation shows that ZapRAID achieves a 2.34× write throughput gain compared with using the Zone Write primitive.

CCS CONCEPTS

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

KEYWORDS

Zoned namespaces, RAID, Storage

ACM Reference Format:
Qiuping Wang and Patrick P. C. Lee. 2023. ZapRAID: Toward High-Performance RAID for ZNS SSDs via Zone Append. In *14th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '23), August 24–25, 2023, Seoul, Republic of Korea.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3609510.3609810

1 INTRODUCTION

Zoned Namespace (ZNS) [4, 5] abstracts flash-based solid-state drives (SSDs) as append-only *zones*, so as to eliminate the costly operations of the traditional block interface and shift the storage management to the host. Compared with conventional SSDs, ZNS SSDs are shown to achieve higher write throughput, lower tail read latencies, and less device-level DRAM usage [5]. ZNS SSDs support two write primitives, namely *Zone Write* and *Zone Append*. Specifically, a ZNS SSD tracks per-zone write pointers, such that any write to a zone must specify the same offset indicated by the write pointer. Zone Write requires the host to specify the block address when writing a block as in conventional SSDs. However, to match the offset of a Zone Write with the on-device write pointer of a zone, the host can only issue one request to each zone at a time, thereby limiting intra-zone parallelism. In contrast, Zone Append eliminates the need for the host to specify block addresses in writes by offloading address management to ZNS SSDs, such that the host only specifies the zone to which a write is issued, and the ZNS SSD returns the address to the host upon the write completion. Thus, the host can issue multiple Zone Append commands to exploit intra-zone parallelism for improved write performance.

Despite the performance gains from Zone Append, its offloading of address management to ZNS SSDs implies that the host not only cannot directly specify the block addresses of writes in a zone, but it also cannot control the ordering of writes in concurrent Zone Append commands. This creates new challenges to host-level address management, particularly when applying RAID [19] to form an array of multiple ZNS SSDs for reliable storage. In traditional $(k + m)$-RAID, a RAID controller runs atop $k + m$ drives and organizes data in *stripes*, each of which encodes k data blocks into m parity blocks and distributes the stripe of blocks across drives. Also, the RAID controller statically assigns the same block address in each drive for the blocks of the same stripe, so that the repair of any lost block can directly retrieve any k alive data and parity blocks of the same stripe from other drives for decoding. However, under Zone Append, the RAID controller needs to maintain dedicated address mapping information to specify the block locations for each stripe, which unavoidably incurs performance penalties to stripe management.

We present ZapRAID, a high-performance software RAID

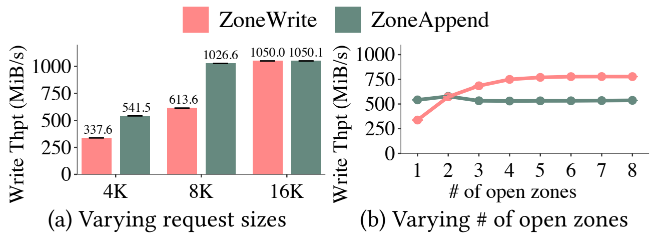


Figure 1: Write throughput of Zone Write and Zone Append.

layer for ZNS SSDs by carefully exploiting Zone Append to achieve high write performance. ZapRAID extends log-structured RAID (Log-RAID) [8, 9, 11, 15] with a novel group-based data layout that partitions stripes into *stripe groups*. It issues Zone Append to the stripes within the same stripe group for high write performance, while the group-based data layout organizes stripes with coarse-grained ordering and enables ZapRAID to manage stripes efficiently on a per-group basis. We evaluate our preliminary ZapRAID prototype on real ZNS SSD devices and show that ZapRAID achieves a 2.34× write throughput gain compared with using the Zone Write primitive.

We now release the source code of our ZapRAID prototype at https://github.com/fallfish/zapraid.

2 PRELIMINARIES

2.1 Why Zone Append?

We conduct evaluation on a testbed with a Western Digital Ultrastar DC ZN540 ZNS SSD [3] (see §4 for testbed details) to show how Zone Append improves write performance over Zone Write. We issue writes of 4-KiB, 8-KiB, and 16-KiB requests to the ZNS SSD. We write a total of 64 GiB of data five times and measure the average write throughput.

We issue writes to a single zone; if the zone is full, we issue writes to a different zone. From Figure 1(a), Zone Write achieves a write throughput of 337.6 MiB/s, 613.6 MiB/s, and 1,050.0 MiB/s for 4-KiB, 8-KiB, and 16-KiB requests, respectively. Recall that Zone Write specifies the per-zone write pointer for the write position (§1), so a ZNS SSD can have only one outstanding Zone Write at any time, and the write throughput cannot be further increased with concurrent write requests. In contrast, Zone Append achieves a write throughput of 541.5 MiB/s, 1,026.7 MiB/s, and 1,050.1 MiB/s for 4-KiB, 8-KiB, and 16-KiB requests, respectively with four concurrent write requests (note that if we issue more than four concurrent write requests, it does not further increase the write throughput of Zone Append as it already saturates intra-zone parallelism for this specific ZN540 model). This shows that Zone Append can increase the write throughput via intra-zone parallelism.

Note that Zone Write already achieves the maximum write throughput of a zone in ZN540 for 16-KiB requests, so Zone Append does not see more growth. Nevertheless, the write request size of 4 KiB is commonly observed in practical storage workloads, such as in production servers [12] and cloud block storage [17], so Zone Append can benefit such applications. Thus, we target the write request size of 4 KiB.

We further examine how *inter-zone* parallelism affects the effectiveness of Zone Append, by issuing writes to multiple zones in parallel. We set the number of concurrent write requests as one and four for Zone Write and Zone Append, respectively. We vary the number of open zones and focus on 4-KiB requests. From Figure 1(b), while Zone Append shows scalable write throughput in the single-zone experiment above, it can only achieve a write throughput of 577.5 MiB/s under two open zones. Zone Write can scale to a larger number of open zones, with a write throughput of 777.2 MiB/s under six open zones. The reason is that the current firmware implementation of Zone Append is more computationally intensive, so Zone Append has even lower write throughput than Zone Write under a larger number of open zones due to the limited computational power in existing hardware. We conjecture that such a limitation can be addressed in future ZNS SSD models.

In this paper, we consider the scenario where an application is deployed in a single open zone per drive; we pose the inter-zone design in future work. The scenario addresses performance isolation in SSD-based shared storage [6, 10, 13]. By exploiting Zone Append for intra-zone parallelism, applications can boost write performance and sustain the bursts of writes, even with only a single open zone.

2.2 Log-structured RAID (Log-RAID)

SSDs adopt out-of-place updates, and small writes trigger frequent device-level garbage collection that degrades I/O performance and flash endurance [7, 14, 18]. Log-RAID [8, 9, 11, 15] applies the log-structured design [20] to SSD RAID by issuing sequential host-level writes to remove small writes.

Log-RAID manages stripes in append-only *segments*. Each segment holds a number of stripes (up to some pre-specified capacity) and is mapped to $k + m$ fixed-size contiguous areas that reside in $k + m$ drives (one area per drive). Log-RAID aggregates newly written blocks as new stripes. It writes each stripe of new $k + m$ blocks to the same offset of the $k + m$ ~~segmen~~ an append-only manner, such that $k + m$ blocks of the same stripe are aligned at the same offsets of the ~~segment~~., *static mapping*) and are stored in different drives for fault tolerance. When recovering any lost block, Log-RAID can deterministically retrieve any k alive blocks of the same stripe at the same offsets from other ~~segmen~~ for decoding. If a segment reaches its full capacity, Log-RAID *seals* the segment and creates a new segment from the free contiguous areas in the underlying drives.

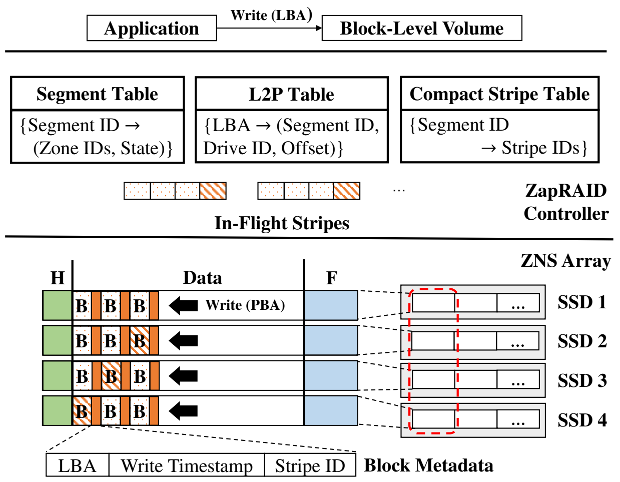


Figure 2: ZapRAID architecture. We show a (3+1)-RAID-5 array and the layout of a segment of four zones, where the parity blocks are rotated across drives.

By treating written blocks as new stripes, Log-RAID needs garbage collection to reclaim the space from stale blocks. When garbage collection is triggered (say, when the available space drops below some threshold), Log-RAID selects a sealed segment by some policy (e.g., using a greedy algorithm to select the one with the most stale blocks), rewrites all non-stale blocks into a new open segment, and releases the space of the selected sealed segment for reuse.

3 ZAPRAID DESIGN

3.1 Design Overview

Architecture. ZapRAID is an extended Log-RAID design for ZNS SSDs. Figure 2 shows the architecture of ZapRAID. ZapRAID exposes a block-level *volume* that supports random reads/writes. We assume that the block size is 4 KiB. A user application can read or write an arbitrary number of blocks, each being identified by a *logical block address (LBA)*. Each drive is a ZNS SSD, and ZapRAID maps each contiguous area of a drive under Log-RAID to a zone, which only allows sequential writes. Let Z be the total number of zones in a drive, so there are Z segments in a ZNS SSD array. For example, a 4-TiB ZN540 ZNS SSD [3] configures $Z = 3{,}690$ zones. Each of the drives, segments, zones, and stripes is associated with an identifier (ID).

Segment organization. Each segment corresponds to $k+m$ zones in $k+m$ drives. It comprises three regions that span across $k+m$ drives: the *header region* and the *footer region* for keeping metadata for crash recovery (§3.3), and the *data region* for storing data and parity blocks. The header region stores the zone IDs of all zones in the segment. The footer region keeps the *block metadata*, including the LBA, write timestamp, and stripe ID of each block in the data region.

All three regions have pre-specified sizes. The header region contains exactly one stripe of $k+m$ blocks, each of which resides at the beginning of a zone. The data region contains a fixed number of stripes, denoted by S, following the header region. Suppose that the LBA size is 8 bytes, the write timestamp size is 8 bytes, the stripe ID size is 4 bytes, and the block size is 4 KiB. Thus, each block in the footer region can store the block metadata of $\lfloor \frac{4096}{20} \rfloor = 204$ blocks, so the footer region occupies $\lceil \frac{S}{204} \rceil$ stripes following the data region. For example, the size of a zone in a ZN540 ZNS SSD [3] is 1,077 MiB (or equivalently, 275,512 blocks). Thus, the header, data, and footer regions occupy 1 block, 274,160 blocks, and 1,351 blocks in a zone, respectively.

Block metadata. ZapRAID stores the block metadata for each block in the out-of-band area of the corresponding flash page for persistence. Each data block has its LBA, write timestamp, and stripe ID as its block metadata. ZapRAID provides fault tolerance for the block metadata: for LBAs and write timestamps, ZapRAID generates parity-based redundancy for them from all data blocks in the same stripe and stores the parity results in the block metadata of the parity blocks, while for stripe IDs, ZapRAID replicates them into all the data and parity blocks in the same stripe.

Recall that the footer region also stores the block metadata for all blocks in the segment, so ZapRAID keeps two copies of block summaries (i.e., in the out-of-band area of each flash page and the footer region). Both copies are necessary for different purposes: the block metadata in the out-of-band area associated with each block provides persistence for block writes, while the block metadata in the footer region provides fast crash recovery (§3.3).

In-memory items. ZapRAID keeps a number of in-memory *in-flight stripes* for newly written blocks. Each in-flight stripe is kept in memory until all of its k data blocks and m parity blocks are formed and persisted. To maintain durability, ZapRAID acknowledges the writes of an in-flight stripe only after the whole in-flight stripe is persisted (note that acknowledging the write of each data block can lead to data loss if a drive storing an acknowledged block fails but the parity blocks are yet generated).

ZapRAID also maintains three in-memory index structures: (i) the *segment table*, which maps each segment ID to its corresponding $k+m$ zones (identified by the zone IDs in the respective $k+m$ drives) and the segment state; (ii) the *logical-to-physical (L2P) table*, which maps the LBA of each block issued by an application to the *physical block address (PBA)* (i.e., the segment ID, the drive ID, and the offset in the respective zone); and (iii) the *compact stripe table*, which maps each segment ID to the stripe IDs of all blocks in the segment. Both the segment table and L2P table are similarly found in Log-RAID, and we adapt them for zoned storage.

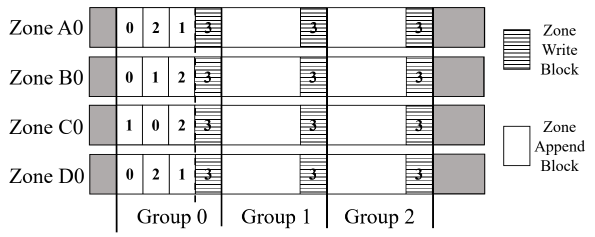


Figure 3: Group-based data layout. Each block is labeled with its stripe ID in the group.

The compact stripe table is newly introduced to ZapRAID (elaborated in §3.2), since Zone Append can make the blocks of the same stripe reside at different offsets across the drives. Note that ZapRAID ensures fault tolerance for the index structures by persisting the segment-to-zones mappings in the segment table into the header region of each segment and persisting the LBAs, write timestamps, and stripe IDs as block metadata into the out-of-band area of each block. Currently, our prototype keeps the index structures all in memory. We can also adopt persistent log-structured indexes [9] to reduce their memory space.

3.2 Group-Based Data Layout

ZapRAID adopts a *group-based data layout* to organize stripes with coarse-grained ordering for low stripe management overhead. It partitions a fixed number of contiguous stripes, denoted by G, within a segment into *stripe groups*, where G is a configurable parameter. For each stripe group, it first issues Zone Append for all but the last stripes within the same stripe group, such that all blocks of each stripe are in the same stripe group but may reside in different offsets within the stripe group. It then issues Zone Write for the last stripe. Thus, each Zone Write serves as an explicit ordering barrier between adjacent stripe groups. Each stripe group is in the same offset ranges across all zones, so the offset ranges of its blocks can be identified via static mapping. Most importantly, ZapRAID only needs to track a small number of stripes within each stripe group, so it can use fewer bits for metadata for significant memory savings.

Figure 3 depicts one segment with $G = 4$ stripes per stripe group. Within the segment, the data region now comprises a fixed number of stripe groups, each of which further comprises a fixed number of stripes. In general, the number of stripe groups in a segment is determined by both S (i.e., the data region size) and G. Each stripe in a stripe group is associated with a unique stripe ID, which can be viewed as a sequence number of when the stripe is generated. Due to Zone Append, the blocks of the same stripe may reside in different offsets, as shown in Figure 3. The last stripe of each stripe group always has the stripe ID $G - 1$.

It is possible for ZapRAID to issue Zone Append to all G stripes per stripe group, instead of issuing Zone Write for the last stripe. Currently, we use the last stripe for data storage, yet we can also use it to store block checksums and intra-device redundancy for higher fault tolerance; such redundancy can only be generated when all $G - 1$ stripes are persisted. Thus, we use Zone Write as an explicit barrier to provide such design flexibility. Also, issuing Zone Write for the last stripe does not cause much performance degradation (§4), as it is only issued for a fraction of stripes.

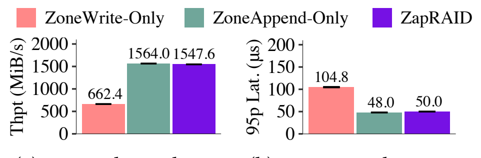
ZapRAID tracks the stripe IDs of all stripe groups in the compact stripe table. For each segment, the compact stripe table stores a two-dimensional $(k + m) \times S$ matrix, in which each entry stores the stripe ID of each block in the segment. In the matrix, the i-th row corresponds to the zone of the i-th drive of the RAID array, and the j-th column corresponds to the j-th block in the segment. Given the LBA of a block, ZapRAID first identifies the PBA from the L2P table, and then retrieves the stripe ID from the compact stripe table.

Trade-off analysis. The choice of G determines the trade-off between the degree of intra-zone parallelism via Zone Append and the stripe management overhead. A larger G allows more stripes to be issued via Zone Append, but it also increases the stripe management overhead.

We analyze the maximum memory usage and query overhead of the compact stripe table. For the maximum memory usage, each stripe ID is represented in $\lceil \log_2 G \rceil$ bits. Thus, the maximum memory size of the compact stripe table is $(k + m) \cdot S \cdot Z \lceil \log_2 G \rceil$ bits. For the query overhead, we measure the number of entries in the compact stripe table being accessed during a degraded read (§3.3), which is $k \cdot G$. With a proper choice of G, ZapRAID can achieve high write performance via Zone Append, while limiting the memory usage and query overhead of the compact stripe table. For example, we consider a (3+1)-RAID-5 array of four 4-TiB ZN540 drives, where $S = 274{,}160$ and $Z = 1{,}351$ (§3.1). For $G = 256$ (our default), the maximum memory size of the compact stripe table is 3.77 GiB, while the query overhead is to access 768 entries, which translates to only around 1μs from our evaluation. In contrast, Zone Append represents the case of $G = S$, which translates to 19 bits per stripe ID and hence 8.95 GiB of memory for the compact stripe table. A large compact stripe table also incurs high query overhead (e.g., in degraded reads).

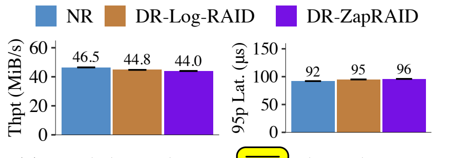
3.3 Complete Workflows

Writes. To write a block (identified by an LBA), ZapRAID first assigns the block to a stripe associated with an open segment. It writes the block into an in-flight stripe in memory (§3.1). It also issues a Zone Append command to write the block and block metadata based on the block's position in the stripe and the RAID scheme. When an in-flight stripe contains k data blocks, ZapRAID encodes them to generate m parity blocks and their block metadata. It then issues a Zone Append command for each of the parity blocks to



(a) Write throughput (b) Write 95p latency

Figure 4: Exp#1 (Write performance).



(a) Read throughput [illegible]ad 95p latency

Figure 5: Exp#2 (Normal and degraded read performance).

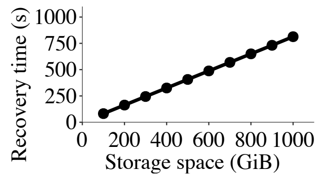


Figure 6: Exp#3 (Full-drive recovery).

its respective zone. Only after all the blocks of a stripe are persisted, ZapRAID updates the L2P table with the corresponding LBAs and PBAs as well as acknowledges the completion of the block writes; the in-flight stripe is also released from memory. If there are insufficient data blocks to form a full stripe after a small timeout since the stripe is created (currently set as 1 ms in our prototype), ZapRAID fills the remaining stripe with zero blocks and invalid LBAs.

If the stripe group is also the last one in the data region, ZapRAID writes the block metadata of all blocks in the segment into the footer region. It also creates a new segment and writes the segment information to the header region, so that the new segment can serve new writes.

Reads. To read a block (identified by an LBA), ZapRAID queries the L2P table for its PBA (i.e., the segment ID, the drive ID, and the offset in the respective zone). It then locates the zone from the segment table and retrieves the block from the specified offset of the zone.

Degraded reads. To issue a degraded read to a lost block, ZapRAID queries the L2P table for its PBA and also queries the compact stripe table to find out the stripe ID of the requested block. Since all alive blocks of the same stripe reside in the same stripe group, ZapRAID searches for the offsets of the k alive blocks in the stripe group from the compact stripe table; note that the search is efficient due to the limited group size. It then reads the alive blocks from the other zones and decodes the requested block.

Full-drive recovery. When a drive fails, ZapRAID recovers the lost data into a new drive. It first identifies the segments that contain the lost zones in the failed drive by examining the segment-to-zones mappings in the header regions of all stored segments. For each lost zone, ZapRAID retrieves all available zones in the same segment from other alive drives into memory, and reconstructs the stripe groups that cover the lost zone. It examines the block metadata of the alive blocks and identifies the blocks from the same stripe. It then decodes the lost blocks for each stripe independently. After repairing all the stripes in a segment, ZapRAID writes the recovered zone to the new drive.

4 EVALUATION

We implement ZapRAID as an SPDK user-space block device module [2] in C++ with around 4.8 K LoC. We present preliminary evaluation results.

Testbed. We use a server that runs Ubuntu 22.04 LTS with Linux kernel 5.15. It has a 16-core Intel Xeon Silver 4215 2.5 GHz CPU and 96 GiB DRAM. It is attached with four 4-TiB Western Digital Ultrastar DC ZN540 ZNS SSDs [3]. Each SSD has 3,690 zones, with a zone capacity of 1,077 MiB each. We format each SSD with a logical block size of 4 KiB and a block metadata size (in the out-of-band area) of 64 bytes.

We consider two baselines by configuring the stripe group size G of ZapRAID (§3.2): *ZoneWrite-Only* (i.e., $G = 1$) and *ZoneAppend-Only* (i.e., $G = S$), which exclusively use Zone Write and Zone Append to write blocks to the SSDs, respectively. For ZapRAID, we set its default group size as $G = 256$. We focus on (3+1)-RAID-5. We configure the logical space with 200 GiB. We report the average results over five runs.

Exp#1 (Write performance). We use Flexible IO Tester (FIO) (v3.30) [1] with the `randwrite` option to issue random writes of 64 GiB of data with a request size of 4 KiB. We fix the queue depth as 16 to saturate system parallelism. Figure 4 shows the write throughput and 95th-percentile latency results. Compared with ZoneWrite-Only, ZapRAID achieves 2.34× write throughput and 52.3% lower 95th-percentile latency. Compared with ZoneAppend-Only, ZapRAID achieves similar performance, yet ZoneAppend-Only incurs much higher memory usage in the compact stripe table (§3.2).

Exp#2 (Normal and degraded read performance). We fill up the 200-GiB logical address space to store every logical block being read, and use FIO with the `randread` option to issue random reads to 16 GiB of data with a request size of 4 KiB. We set the queue depth as one, so as to focus on the performance of individual read requests and exclude the interference among requests. We consider three cases: normal reads (NR), degraded reads under the static mapping in Log-RAID (DR-Log-RAID) (see §2.2), and degraded reads under the group-based data layout in ZapRAID (DR-ZapRAID); note that both Log-RAID and ZapRAID have the same workflow for normal reads. To evaluate degraded reads, we fail a drive and issue reads to the lost blocks of the failed drive. Figure 5 shows the read throughput and 95th-percentile latency results. All read operations have less than 5% of difference in performance results. The degraded reads of both Log-RAID and ZapRAID only have slightly worse performance than normal reads, as both systems retrieve alive blocks in parallel. Also, ZapRAID achieves comparable performance to Log-RAID, which uses static mapping.

Exp#3 (Recovery time). We sequentially write data to ZapRAID configured with a fixed size of logical space (varying from 100 GiB to 1,000 GiB). We erase all data in one drive, and recover the lost data in the same drive. We report the average recovery time as the total time spent on reading alive data from existing drives, reconstructing the stripes, and writing the recovered data to the new drive. Figure 6 shows the results. The full-drive recovery time is proportional to the logical space size. For example, ZapRAID takes 81.9 s and 813.2 s to recover the lost data for the 100-GiB and 1,000-GiB storage space, respectively.

5 RELATED WORK

Some studies propose Log-RAID architectures for SSD RAID to improve both write performance and flash endurance. SOFA [8] places the FTL in the RAID controller for efficient data management. Purity [9] manages both indexing and data storage under Log-RAID and supports data compression. SALSA [11] implements a general translation layer for SSDs and Shingled Magnetic Recording (SMR) disks. SWAN [15] proposes spatial data separation to reduce garbage collection interference. ZapRAID targets ZNS SSDs. A recent work, RAIZN [16], exposes a ZNS SSD array as a single ZNS interface to applications, and focuses on fault tolerance, correctness, and crash consistency. In contrast, ZapRAID focuses on exploiting Zone Append for high performance.

6 CONCLUSION AND FUTURE WORK

We make a case for showing how to effectively deploy RAID on ZNS SSDs. ZapRAID is a software RAID layer for ZNS SSDs and aims for high performance, lightweight stripe management, and reliability. It exploits Zone Append for high write performance, and proposes the group-based data layout to mitigate stripe management overhead. Prototype evaluation on real ZNS SSDs shows that ZapRAID improves the write throughput over the exclusive use of Zone Write, while maintaining efficient degraded reads and crash recovery. Our future work includes: (i) exploring mixed request sizes (e.g., beyond 4 KiB) and inter-zone parallelism atop Zone Append (§2.1), (ii) optimizing memory usage of index structures (§3.1), (iii) exploring the design space of issuing Zone Write for the last stripe per stripe group (§3.2), (iv) a more detailed analysis on crash consistency, and (v) extending evaluation (e.g., CPU/memory overhead, impact of the stripe group size and RAID scheme, etc.).

Acknowledgments. This work is supported by Research Grants Council of Hong Kong (GRF 14214622 and AoE/P-404/18). The corresponding author is Patrick P. C. Lee.

REFERENCES

[1] Accessed in 2023. Fio - Flexible I/O Tester Synthetic Benchmark. http://git.kernel.dk/?p=fio.git.
[2] Accessed in 2023. SPDK Block Device Layer Programming Guide. https://spdk.io/doc/bdev_pg.html.
[3] Accessed in 2023. Western Digital Ultrastar DC ZN540. https://www.westerndigital.com/products/internal-drives/data-center-drives/ultrastar-dc-zn540-nvme-ssd.
[4] Accessed in 2023. Zoned Storage Website. https://zonedstorage.io.
[5] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *Proc. of USENIX ATC*.
[6] Da-Wei Chang, Hsin-Hung Chen, and Wei-Jian Su. 2015. VSSD: Performance Isolation in a Solid-State Drive. *ACM Trans. on Design Automation of Electronic Systems* 20, 4 (2015), 51:1–51:33. https://doi.org/10.1145/2755560
[7] Feng Chen, David A. Koufaty, and Xiaodong Zhang. 2009. Understanding Intrinsic Characteristics and System Implications of Flash Memory based Solid State Drives. In *Proc. of ACM SIGMETRICS*.
[8] Tzi-cker Chiueh, Weafon Tsao, Hou-Chiang Sun, Ting-Fang Chien, An-Nan Chang, and Cheng-Ding Chen. 2014. Software orchestrated flash array. In *Proc. of ACM SYSTOR*.
[9] John Colgrove, John D Davis, John Hayes, Ethan L Miller, Cary Sandvig, Russell Sears, Ari Tamches, Neil Vachharajani, and Feng Wang. 2015. Purity: Building fast, highly-available enterprise flash storage from commodity components. In *Proc. of ACM SIGMOD*.
[10] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. 2017. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *Proc. of USENIX FAST*.
[11] Nikolas Ioannou, Kornilios Kourtis, and Ioannis Koltsidas. 2018. Elevating commodity storage with the SALSA host translation layer. In *Proc. of IEEE MASCOTS*.
[12] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. 2008. Characterization of Storage Workload Traces from Production Windows Servers. In *Proc. of IEEE IISWC*.
[13] Bryan Suk Kim. 2018. Utilitarian Performance Isolation in Shared SSDs. In *Proc. of USENIX HotStorage*.
[14] Hyojun Kim and Seongjun Ahn. 2008. BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage.. In *Proc. of USENIX FAST*.
[15] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H. Noh. 2019. Alleviating garbage collection interference through spatial separation in all flash arrays. In *Proc. of USENIX ATC*.
[16] Thomas Kim, Jekyeom Jeon, Nikhil Arora, Huaicheng Li, Michael Kaminsky, David Andersen, Gregory R. Ganger, George Amvrosiadis, and Matias Bjørling. 2023. RAIZN: Redundant Array of Independent Zoned Namespaces. In *Proc. of ACM ASPLOS*.
[17] Jinhong Li, Qiuping Wang, Patrick P. C. Lee, and Chao Shi. 2020. An In-Depth Analysis of Cloud Block Storage Workloads in Large Scale Production. In *Proc. of IEEE IISWC*.
[18] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. 2012. SFS: Random write considered harmful in solid state drives.. In *Proc. of USENIX FAST*.
[19] David A. Patterson, Garth A. Gibson, and Randy H. Katz. 1988. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proc. of ACM SIGMOD*.
[20] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. *ACM Trans. on Computer Systems* 10, 1 (1992), 26–52.