BIZA: Design of Self-Governing Block-Interface ZNS AFA for Endurance and Performance

Shushu Yi
Peking University

Shaocong Sun
Peking University

Li Peng
Peking University

Yingbo Sun
Peking University

Ming-Chang Yang
The Chinese University of Hong Kong

Zhichao Cao
Arizona State University

Qiao Li
Xiamen University

Myoungsoo Jung
KAIST and Panmnesia

Ke Zhou
WNLO, HUST

Jie Zhang
Peking University

Abstract

All-flash array (AFA) has become one of the most popular storage forms in diverse computing domains. While traditional AFA implementations adopt the block interface to seamlessly integrate with most existing software, this interface hinders the host from managing SSD internal tasks explicitly, which results in both short endurance and poor performance. In comparison, ZNS AFA, such as RAIZN, adopts ZNS SSDs and exposes the ZNS interface to the users. This solution attempts to raise the level of responsibility for SSD management. Unfortunately, it faces severe compatibility issues as most upper-layer software only takes block I/O accesses for granted.

In this work, we propose *BIZA*, a self-governing block-interface ZNS AFA to proactively schedule I/O requests and SSD internal tasks via the ZNS interface while exposing the user-friendly block interface to upper-layer software. BIZA achieves both long endurance and high performance by exploiting the zone random write area (ZRWA) and internal parallelisms of ZNS SSDs. Specifically, to mitigate write amplification, BIZA proposes a ghost-cache-based algorithm to identify hot data and absorb their updates in the scarce ZRWA. BIZA also employs a novel guess-and-verify mechanism to detect the mappings between zones and I/O resources at a low cost. Thereafter, BIZA can serve write requests and internal tasks in parallel with our customized I/O scheduler for high throughput and low latency. The evaluation results show that BIZA can reduce write amplification by 42.7% while achieving 93.2% higher throughput and 62.8% lower tail latency, compared to the state-of-the-art AFA solutions.

CCS Concepts: • **Information systems → RAID**; • **Software and its engineering → Secondary storage**.

Keywords: Zoned Namespace, All-Flash Array, Solid State Drive, Zone Random Write Area, Garbage Collection

ACM Reference Format:
Shushu Yi, Shaocong Sun, Li Peng, Yingbo Sun, Ming-Chang Yang, Zhichao Cao, Qiao Li, Myoungsoo Jung, Ke Zhou, and Jie Zhang. 2024. BIZA: Design of Self-Governing Block-Interface ZNS AFA for Endurance and Performance . In *ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24), November 4–6, 2024, Austin, TX, USA.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3694715.3695953

1 Introduction

The last decade has witnessed all-flash arrays (AFA) [34, 42, 53, 77, 99, 100] becoming one of the dominant storage forms in datacenters and high-performance computers [10, 54], which have successfully accelerated I/O-intensive tasks, including big data analysis, scientific computing, and machine learning [9, 40, 64, 76, 88]. Compared to arrays of traditional storage media like hard disk drives (HDD), AFAs capitalize on the advantages of flash-based solid-state drives (SSDs), such as high throughput, high random I/O performance, low latency, and better power efficiency [74, 89, 101].

While AFA has experienced continuous technical shifts, there remains an open question in its design, that is, exposing which *external* I/O interface to upper-layer software and selecting which *internal* I/O interface of underlying SSDs. We analyze representative AFA designs that explore different I/O interfaces [13, 34, 45, 89], which are shown in Figure 1.

Traditional AFA solutions (cf. Figure 1a) inherit block interfaces from the legacy HDD-based RAID as both external and internal I/O interfaces. Block interface is the de-facto

SOSP '24, November 4–6, 2024, Austin, TX, USA

ACM ISBN 979-8-4007-1251-7/24/11
https://doi.org/10.1145/3694715.3695953

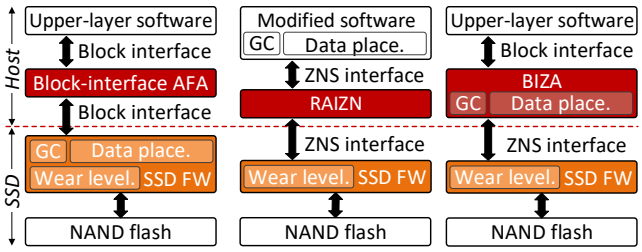


Figure 1. Comparison of different AFA designs.

choice for most upper-layer software (e.g., file systems and databases), which provides the most universality for developers. However, the block interface is not well aligned with the unique features of SSDs. To be precise, the storage backbone of SSD (i.e., NAND flash [44]) only supports out-of-place updates due to its physical constraints. To shield users from the complex SSD internal architecture and provide a uniform block interface to the host, SSD firmware implicitly takes on several internal tasks, such as garbage collection (GC) and data placement. These tasks interfere with regular I/O requests, severely degrading the storage performance [43].

As an alternative, RAIZN [45] chooses to construct AFA from zoned namespace (ZNS) SSDs and also exposes the ZNS interface to upper-layer software (cf. Figure 1b). By delegating several device-side responsibilities (e.g., GC scheduling and data placement) to the host, ZNS interface provides exclusive opportunities for holistic designs. However, RAIZN faces severe compatibility issues as it directly exposes ZNS interface to upper layers. ZNS interface only allows sequential writes, whereas most existing software takes random writes for granted. In addition, RAIZN requires the involvement of upper-layer software to explicitly manage GC events and data placement, which are rarely supported by applications.

To address the compatibility issue, dm-zap [13], an adapter that bridges the block interface and ZNS interface by maintaining mappings between block numbers and zone-related addresses (i.e., zone numbers and in-zone offsets), can be used. From the semantic perspective, stacking dm-zap on top of RAIZN is promising, as it exposes the user-friendly block interface to upper-layer software while constructing AFA with ZNS SSDs. However, we observe that this simple combination wipes out all the benefits brought by the ZNS interface. This, in turn, shortens SSD endurance, decreases overall throughput, and prolongs tail latency. The root cause is that neither dm-zap nor RAIZN explicitly exploits the ZNS interface to achieve cross-layer optimizations. To quantitatively analyze the impacts of dm-zap and RAIZN on the storage system, we conduct extensive experiments on a dm-zap+RAIZN prototype building from 4 commodity ZNS SSDs, i.e., Western Digital ZN540 [16] (cf. § 5 for details). We summarize the prominent shortcomings as follows:

• *SSD endurance*: dm-zap shortens SSD endurance. Specifically, dm-zap is unaware of the lifetimes of data (i.e., duration until being invalidated) sent by applications and will muddle data with diverse lifetimes in the same zones. When such zones are selected as victims of GC, lots of data within them are still valid and require extra flash writes for migration. These unexpected writes exaggerate write amplification (i.e., the ratio of flash writes to user writes) by 33.3%.

• *Throughput*: Both dm-zap and RAIZN cannot exhaust the potential throughput of ZNS SSDs. dm-zap only allows one in-flight write per zone to avoid write failures caused by I/O reorders within the storage stacks [82]. On the other hand, RAIZN employs a centralized zone to accommodate the frequently updated metadata (i.e., partial parities [45]). While this approach is GC-friendly, as most metadata in the zone will be invalid when collecting, it unfortunately puts a cap on the peak throughput. As a result, dm-zap+RAIZN only achieves 47.7% of the ideal peak throughput.

• *Tail latency*: Another shortcoming is latency spikes caused by SSD GC [43, 53]. Specifically, the upper-layer software is unaware of GC events controlled by dm-zap. When GC and I/O accesses occur simultaneously, GC interferes with the I/O requests, resulting in a significantly long tail latency. Experimental results show that dm-zap+RAIZN suffers 10.3× higher 99.99^{th} tail latency when GC occurs.

In response to these challenges, we introduce **BIZA**[1] , a self-governing **B**lock-**I**nterface **Z**NS **A**FA that can harvest the benefits of fine-grained SSD control brought by ZNS interface to achieve long endurance and high performance while providing the user-friendly block interface to applications (cf. Figure 1c). To this end, BIZA merges the block-to-ZNS interface adapter (dm-zap) and RAIZN into a uniform indirection layer. On behalf of the upper-layer software, BIZA coordinates the I/O requests and SSD internal tasks via the ZNS interface, which achieves cross-layer optimizations.

Our key insight is that the newly introduced features and internal parallelisms of ZNS SSDs can be exploited to address the write amplification and performance issues in existing AFA solutions. Specifically, recognizing the write amplification, BIZA takes advantage of the emerging *zone random write area* (ZRWA) feature [70, 81] of ZNS SSDs. ZRWA is an abstraction of the write buffer within SSD. It allows in-place updates in a limited area of each zone. With ZRWA, BIZA can absorb the frequently updated data and parities in the write buffer and avoid flushing them to the flash. For the throughput issue, our preliminary study reveals that a single in-flight write only delivers 34.7% of a zone's bandwidth. This observation encourages us to develop a new I/O scheduler that enables parallel writes on a zone. Moreover, improperly serving user requests and GC events with the same I/O resources causes 3.1× higher tail latency. Therefore, BIZA cautiously directs them to isolated I/O resources which alleviates the latency spikes. Comprehensive evaluation results demonstrate that BIZA outperforms leading

[1]BIZA is accessible at https://github.com/ChaseLab-PKU/BIZA.

✓ Good ✘ Bad ○ Depend on upper-layer software

Design choice	Endurance	Throughput	Tail latency	Compatibility
Block-interface	✘	✓	✘	✓
ZNS-interface	○	○	○	✘
Adapter	✘	✘	✘	✓
Our solution	✓	✓	✓	✓

Table 1. Comparison of different AFA design choices.

AFA designs, alleviating write amplification by 42.7%, delivering 93.2% higher write throughput, and reducing 99.99th percentile tail latency by 62.8%.

Our main **contributions** can be summarized as follows:

• *Constructing self-governing block-interface ZNS AFA:* We conduct an in-depth analysis of the interface choices of the existing AFA solutions. Our experimental results reveal that block-interface-alone, zone-interface-alone, and simple-combination designs of AFAs are all unsatisfactory in terms of endurance, performance, and compatibility. Based on these lessons, we propose a self-governing AFA engine, BIZA, that solves all these issues by exploiting the new features and internal parallelism of ZNS SSDs.

• *Exploring the ZRWA feature to reduce write amplification:* As an abstraction of SSD-internal write buffer, ZRWA can be employed to absorb data and parity updates thereby mitigating write amplification. However, it is non-trivial to exploit ZRWA efficiently given its limited size. To better utilize ZRWA, we design an innovative placement policy, which can reserve the scarce ZRWA for data and parities that will be updated multiple times in a short duration. To the best of our knowledge, BIZA is the first study to reveal and exploit the characteristics of ZRWA.

• *Probing zone-level parallelism for better performance:* One in-flight write cannot exhaust a zone's parallelism. Therefore, we design a new I/O scheduler that enables concurrent writes on a zone while avoiding write failures caused by I/O reorders. In addition, scheduling user writes and SSD-internal tasks (e.g., GC) to access the different I/O resources is helpful for tail latency. The main obstacle is that ZNS interface hides the mappings between zones and internal I/O resources from users for wear leveling. This uncertainty causes huge trouble for users to properly choose isolated zones for I/O serving. Tackling this issue, we propose a novel guess-and-verify mechanism that can confirm the zone-to-I/O-resources mappings online with low costs.

2 Background and Challenge

All-flash arrays (AFA, also named SSD-based RAID) are a type of storage form that bundles multiple SSDs into an array to aggregate their capacity and throughput. AFA also guarantees fault tolerance by introducing redundancy for data stripes (i.e., a group of data lying in different SSDs). This includes XOR parity for RAID 5 and Reed-Solomon code [90] for other general scenarios. For simplicity, this paper mainly focuses on RAID 5. Our designs can also be applied to other RAID levels and AFA schemes (e.g., RAID 6). Based on their

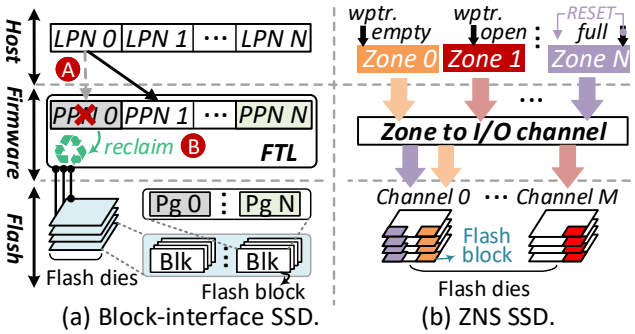


Figure 2. Block-interface SSD versus ZNS SSD.

I/O interfaces, existing AFAs can be classified into three types: block-interface AFA, ZNS-interface AFA, and AFA with the adapter. Table 1 summarizes the key differences between these design choices and our solution.

2.1 Block-Interface All-Flash Array

As a successor of the legacy hard disk drives (HDD) based RAID, AFAs, such as the default Linux kernel AFA, *mdraid* [58], typically consist of block-interface SSDs and expose block interface to users. Block interface, which abstracts storage space as a set of logical blocks and allows users to write blocks randomly, is the de facto choice for most upper-layer software (e.g., file systems, databases, and distributed storage). Thanks to the uniform block interface, AFA requires minimal effort when adopting applications originally deployed on HDD-based RAID.

However, from the perspective of SSDs, the block interface is just a compromise of compatibility. Limited by the physical attributes, the flash backbone of SSDs (i.e., NAND flash [23, 43]) disallows in-place updates and follows an *erase-before-program* manner. To hide the flash limits from users, SSD firmware internally constructs an indirection layer, known as *flash translation layer (FTL)*, to remap incoming write requests (i.e., *LPN*) to new flash pages (i.e., *PPN*) and invalidate the stale pages (cf. A in Figure 2a). The invalidation induced by data updates can significantly diminish the available SSD capacity exposed to users. To tackle this problem, SSD firmware performs *garbage collection (GC)* to reclaim invalid pages (B).

While FTL succeeds in shielding users from the complex details of NAND flash and provides a compatible block interface, it also introduces an insurmountable gap between the software (i.e., AFA engine) and hardware (i.e., flash). This gap not only shortens the endurance of SSDs but also increases the tail latency of AFAs [42, 96].

2.2 ZNS-Interface All-Flash Array

The emerging NVMe zoned namespace (ZNS) standard and its derived ZNS interface [5, 71] are promising solutions to break the boundary between software and hardware, enabling cross-layer optimizations. Figure 2b depicts a typical

architecture of ZNS-interface SSDs (i.e., ZNS SSDs). ZNS interface packs the flash backbone as zones and authorizes parts of decisions (e.g., which zone to write and when to clean zones) to host, enabling fine-grained control of the storage media. A zone is an abstraction of multiple parallel I/O resources (e.g., flash blocks in groups of flash dies) within ZNS SSD. For simplicity, we refer to each group of I/O resources as *I/O channel*. For instance, in Figure 2b, the left and right groups of flash dies make up *Channel 0* and *M*, respectively. *Zone 0* is mapped with flash blocks scattered across the flash dies in *Channel 0*.

ZNS SSD marks each zone with different states based on the zone's storage space usage. We briefly summarize the states as follows. Zones start with *empty* state and can only serve write requests in *open* state (after receiving the first write request or *OPEN* command). Constrained by the physical attribute of NAND flash, zones only allow sequential write. A *write pointer* (i.e., "*wptr*" in Figure 2b) is maintained for each zone to record the next write position. In comparison, zones can serve both sequential and random read requests in any state except the *offline* state (i.e., hardware damaged). When a zone is fully written, it enters *full* state and rejects subsequent writes. The host can transit the zone back to the empty state with *RESET* commands. However, RESET also drops all data stored in the zone. Limited by the hardware resources (e.g., on-device DRAM), only a finite number (e.g., 14 [16] or 384 [75]) of zones can stay in open state simultaneously.

RAIZN [45] is a representative work that constructs a ZNS-interface AFA from multiple ZNS SSDs. This design choice is straightforward as it leaves the SSD management tasks to software developers, which requires huge manpower for adaptation. Moreover, the restriction of sequential writes hinders RAIZN from adopting existing applications that generate random writes without constraints. For example, F2FS [48], a ZNS-friendly file system that writes files in a sequential manner, still requires a two-zone-sized random-write space for updating metadata [5].

2.3 Simple Solution and Challenge

Based on the lessons of the above design choices, in this work, we explore how to optimize AFA with the support of ZNS SSD while enjoying the benefits brought by the developer-friendly block interface. With block interface, AFA can support not only ZNS-friendly software but also other general filesystems and applications (e.g., EXT4 [63], XFS[85], and FAT [65]), which provides the most universality to users.

Adapter of block and ZNS interfaces. A straightforward idea is employing adapter [13, 56] to convert ZNS interfaces to block interfaces. For example, dm-zap [13] achieves this by maintaining mappings from block numbers to zone-related addresses (i.e., zone numbers and in-zone offsets). With dm-zap, users can first group ZNS SSDs with RAIZN and then employ dm-zap to translate the ZNS interface of RAIZN to the block interface (called dmzap+RAIZN). An alternative way is using dm-zap to map each ZNS SSD as a block-interface SSD and then constructing AFA with these SSDs by utilizing a conventional AFA engine (e.g., mdraid+dmzap).

Challenges. However, these simple combinations of AFA and adapter are just solutions to bridge the semantic gap between block and ZNS interfaces. They fail to harvest the benefits brought by ZNS interface and cannot exploit the capability of ZNS SSDs. To quantitatively analyze this, we set up an experiment on AFAs (RAID 5) consisting of 4 commodity ZNS SSDs (i.e., Western Digital ZN540 4TB [16]). Each SSD can achieve 2170 MB/s peak write throughput. We construct the AFAs with both dmzap+RAIZN and mdraid+dmzap methods (cf. § 5 for detailed experimental setups). We summarize the prominent shortcomings as follows:

• *Short endurance*: dmzap+RAIZN and mdraid+dmzap generate up to 33.3% and 54.6% more flash writes than the write requests initiated by the users in real workloads. This is because dm-zap muddles data with different lifetimes (i.e., duration until being invalidated) in the same zones. When such zones are selected as victims for GC, a large fraction of data within them may still be valid and need to be migrated to a free zone via extra flash writes. We call the ratio of flash writes to user writes as *write amplification*. Write amplification hampers the endurance of SSDs [83].

• *Low throughput*: dmzap+RAIZN and mdraid+dmzap can only achieve 3.1 GB/s and 1.2 GB/s peak write throughput, respectively, which are 47.7% and 18.4% of the ideal (i.e., 6.4 GB/s when 3 of the 4 SSDs serve user I/O with maximum throughput while the other one handles parities). This is because both dm-zap and RAIZN cannot exhaust the potential throughput of underlying devices. Specifically, dm-zap submits writes serially and only allows one in-flight write per zone. On the other hand, RAIZN relies on a centralized zone to buffer the frequently updated metadata [45], which puts a cap on the peak throughput.

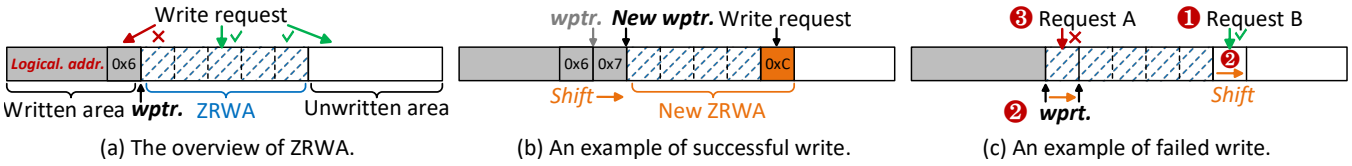
• *High tail latency*: dmzap+RAIZN and mdraid+dmzap increase 99.99^{th} tail latency by 10.3× and 2.2× respectively, after GC starts. This is because the upper-layer software is unaware of GC events controlled by dm-zap. When GC and I/O accesses occur simultaneously, GC interferes with the I/O requests, resulting in a significantly long tail latency.

3 Preliminary Study: ZNS SSD Exploration

By performing a comprehensive analysis of commodity ZNS SSDs, we find that the newly introduced features and internal parallelisms within ZNS SSDs can be employed to tackle the aforementioned endurance and performance issues.

3.1 Zone Random Write Area

Zone Random Write Area (ZRWA) is a new feature that has been ratified by NVMe specification [70, 81] and realized in commodity ZNS SSDs, examples of which are listed in



(a) The overview of ZRWA. (b) An example of successful write. (c) An example of failed write.

Figure 3. Overview of ZRWA and examples of writes on top of it.

ZNS SSD prototype	Zone capacity	ZRWA size per open zone	Max. # of open zones	Total ZRWA size
WD ZN540 [16]	1077 MB	1 MB	14	14 MB
DapuStor J5500Z [14]	18144 MB	1 MB	16	16 MB
Inspur NS8600G [31]	2880 MB	1440 KB	8	11.25 MB
Samsung PM1731a [75]	96 MB	64 KB	384	24 MB

Table 2. ZRWA-related configurations of different ZNS SSDs.

Table 2. As shown in Figure 3a, if a zone is opened with the ZRWA feature, a fixed-size area after its write pointer becomes ZRWA. ZRWA allows random writes as well as in-place updates, which breaks the strict sequential write constraint of the ZNS interface. ZRWA, somehow, is an efficient abstraction of the write buffer within SSDs, which can be physically implemented with battery-backed DRAM, NVM, or FTL-mapped high-endurance flash (e.g., SLC [1]). ZRWA can shift with the write pointer after receiving write requests. Figure 3b shows an example. When users send a write targeting 0xC (one block out of ZRWA), the ZRWA will implicitly shift right by one block to include the 0xC within it. At the same time, the data pointed by the original write pointer (i.e., 0x7) is flushed from the write buffer and stored in the flash backbone. With ZRWA, we can absorb the frequently updated data within the write buffer, thereby mitigating write amplification.

Unfortunately, it is non-trivial to harness ZRWA efficiently. As listed in Table 2, the size of ZRWA is very limited (e.g., 1 MB per open zone and at most 14 MB in ZN540 SSD), which seldom utilizes the temporal locality of the existing workloads. To better understand this, we compare ZRWA size with the reuse distance of different workloads. We define *reuse distance* as the amount of written data between two consecutive visits of the same address. Figure 4 illustrates the cumulative distribution function (CDF) of reuse distance in SYSTOR [49] workload set, including 2188 traces collected from different drives at different times. Only 17% of data has a reuse distance shorter than 14 MB. Therefore, if AFA randomly writes data in zones, the obsolete data (i.e., no longer be updated) will occupy the ZRWA and advance the write pointer quickly. Consequently, 83% of updates, which have reuse distances greater than 14 MB, cannot be absorbed by ZRWA as the write pointer is out of its destination address and the initial data has been evicted from the write buffer. To address this issue, we propose a novel algorithm (cf. § 4.2) that carefully chooses zones for data. By separating the consistently refreshed data from obsolete ones, we prevent the obsolete data from polluting ZRWA and thus can absorb more updates with ZRWA.

3.2 Intra-zone Parallelism

The existing I/O stack has no guarantee on the orders of request submissions. Both block layer and NVMe driver may reorder user requests for higher performance [38, 50, 59]. These implicit reorders can cause failures for parallel writes. Figure 3c depicts a bad example where users send two write requests (i.e., `Request A` and `Request B`) in parallel. If `Request A` is served before `Request B`, both of them will be completed successfully. However, when I/O reorders occur, `Request B` is served first (❶), and thus ZRWA shifts right unexpectedly (❷). In this situation, `Request A` cannot be served (❸) as its destination address lags behind the write pointer. This issue also exists without enabling ZRWA [82], which can be considered a special case where the ZRWA size is 0.

A promising solution is sending I/O requests serially and only allowing one in-flight request per zone (same as dm-zap). However, such a single-write strategy fails to utilize the intra-zone parallelism. We demonstrate this by examining ZN540 SSD [16]. We test two different designs that allow 1 and 32 in-flight writes per zone, respectively. The latter is facilitated by our solution, which will be described in § 4.4. Figure 5 illustrates the throughput of each design. The red line represents the maximum performance of a single zone. In comparison with 32 in-flight writes, one in-flight write loses up to 65.3% throughput (54.5% by average) across different write sizes.

Another possible solution is *APPEND* command [71], which enables parallel data appending to a zone. APPEND avoids write failures by disallowing the host to specify a certain write address, instead returning an SSD-determined address to the host after data is written. APPEND is useful in scenarios where upper-layer software explicitly writes data in a log-structured style (e.g., files opened with *O_APPEND* tag). However, in practice, most software is constructed based on the assumption of block interface and thus is unlimited to generate diverse I/O access patterns such as random writes (e.g., metadata writes in F2FS, cf. § 2.2). Moreover, APPEND is mutually exclusive with ZRWA. According to the stipulation of NVMe specification [71], zones opened with the ZRWA feature will directly abort APPEND commands. While prior work [87] chooses APPEND to exhaust intra-zone parallelism, in this paper, we prefer ZRWA to enjoy its benefits of write amplification reduction. Considering the intra-zone

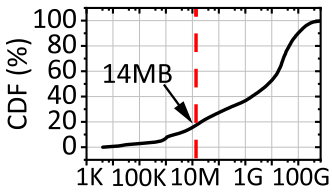


Figure 4. CDF of reuse distance.

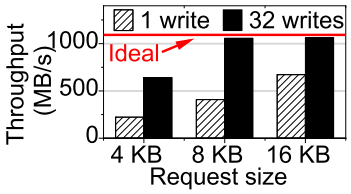


Figure 5. Intra-zone parallelism.

Scenario	Bandwidth (MB/s)	Avg. lat. (us)	50^{th} lat. (us)	99.99^{th} lat. (us)
1. Write a **single** zone	1092	59	41	570
2. Write **two** zones in **identical** I/O channel	1092	118	66	2310
3. Write **two** zones in **diverse** I/O channels	2170	59	46	685

Table 3. Write performance in different write scenarios.

parallelism issue, we design a new I/O scheduler that also enables concurrent writes by carefully submitting write requests with a sliding-window [8, 47, 86] based algorithm (cf. § 4.4). Note that our choice of ZRWA does not compromise the support of append semantics (e.g., O_APPEND) as we provide a generic block interface to upper-layer software.

3.3 Inter-zone Parallelism

ZNS SSDs have multiple independent I/O channels. Zones that are mapped with separated I/O channels can handle I/O requests in parallel to achieve high throughput while minimizing interference to reduce tail latency. We verify this claim with a ZN540 SSD [16]. Table 3 shows the throughput and latency when serving 64 KB write requests with a single zone (`Scenario 1`) or two zones mapped with the same/different (`Scenarios 2 and 3`) I/O channels. Serving writes with zones mapped to identical I/O channels (`Scenario 2`) has no improvement on the throughput and results in 1.0×, 0.6×, and 3.1× higher average, 50^{th}, and 99.99^{th} latency, compared to `Scenario 1`. Nevertheless, spreading write requests to zones mapped with different I/O channels resolves the I/O contention (`Scenario 3`). It doubles write bandwidth of `Scenarios 1` and 2. In addition, the I/O latency is close to that of `Scenario 1`. The slight latency increment is caused by contention on unique hardware within SSD (e.g., ARM-based SSD controller [23, 61]).

This analysis encourages us to take a decentralized design that can exploit all the I/O channels for request handling. In particular, we should give up the design of centralized metadata zone in RAIZN, which is used to accommodate partial parity updates [45]. Partial parity is the XOR sum of the parts of the data in a stripe. As an intermediate result of final parity, partial parity get updated very frequently (i.e., when new writes arrive). While data writes are spread across multiple zones, the partial parities only compete for the centralized metadata zone, which easily throttles the overall throughput. One way to resolve this is to reserve more open zones for partial parities. However, this also means fewer open zones to serve data. Furthermore, determining the optimal assignment of open zones for partial parities and data is challenging, given the variability of SSD configurations as listed in Table 2 (e.g., maximum number of open zones). Another promising solution is placing data and partial parities in the same open zones. Nevertheless, this approach exaggerates write amplification as data and partial parities usually have different reuse distances. Fortunately, with ZRWA, we can absorb partial parities within the on-device write buffer, which allows us to avoid flushing them to the flash backbone frequently and mixing them up with data (cf. § 4.2).

Challenge	Endurance	Throughput	Tail Latency
Key insight	ZRWA	Intra- & Inter-zone parallelism	Inter-zone parallelism

Table 4. Challenges and key insights.

On the other hand, the GC-induced latency spikes can be alleviated if we can synergistically schedule user I/O requests and internal GC events with zones mapped to different I/O channels (i.e., `Scenario 3`). Unfortunately, it is non-trivial to realize this. To be precise, the discretion of mapping zones to which I/O channel is reserved by ZNS SSDs for wear leveling and hidden behind the ZNS interface [3, 5]. The decisions are not made until the zones are opened. This uncertainty hinders AFAs from serving requests with proper zones. Prior work [3] solves this problem with an aggressive method, that is, when opening zones, pausing user requests and internally sending I/O requests to all pairs of open zones. By comparing the latency, it knows whether a pair of open zones are mapped on the same I/O channel (i.e., latency is significantly higher than the average if so). This sophisticated diagnosis is accurate but causes severe performance degradation, as it starts whenever zones are opened (i.e., previously opened zones are full) and user I/O is postponed until the completion of diagnosis. As a silver lining, we observe that most commodity ZNS SSDs [16, 66] typically map zones to I/O channels in a round-robin manner. Therefore, we first guess the mappings between zones and I/O channels following the round-robin manner. Afterward, we monitor the latency of each user I/O online and correct our conjectures with a vote-based algorithm (cf. § 4.3) when latency spikes occur.

Summary of preliminary study. Taking our explorations of ZNS SSD into consideration, Table 4 summarizes our key insights to each of the challenges mentioned in § 2.3.

4 Design and Implementation

Inspired by the aforementioned preliminary analysis, we propose BIZA, a self-governing AFA engine that proactively utilizes the ZNS interface to achieve long SSD endurance and high performance whilst exposing a compatible block

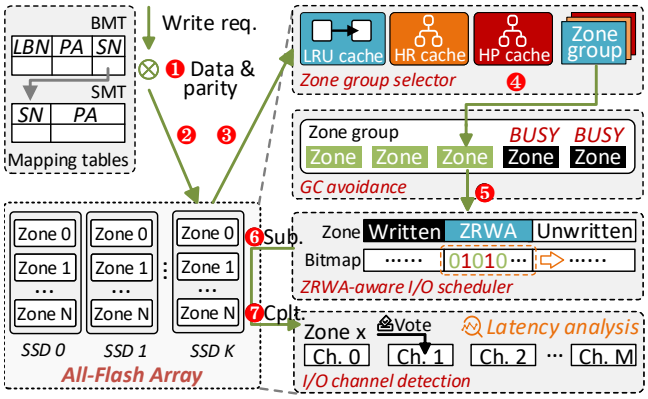


Figure 6. Overview of BIZA.

interface to applications. In the following, we first describe the architecture of BIZA along with its I/O path (§ 4.1). Then, we illustrate the detailed designs of each component in BIZA (§ 4.2, § 4.3, and § 4.4). Finally, we state how we implement BIZA in the production environment (§ 4.5).

4.1 Overview

I/O path. BIZA handles write requests in a log-structure-like manner [34] that appends new data sequentially in zones and maintains a mapping table to record where data is stored. Different from the conventional log-structure method, BIZA can overwrite data in place with the support of ZRWA. This relaxation is useful for alleviating write amplification.

Figure 6 illustrates the architecture of BIZA. In the following, we describe it along the write path. When a write request arrives, BIZA computes parities (❶) and decides which SSDs to store the data and parities (we call both data and parities as *chunks* for simplicity) according to the RAID level (e.g., left-asymmetric [62] for RAID 5, ❷). Afterward, the zone group selector (§ 4.2) in BIZA chooses destination zone groups for chunks (❸). The main goal of the zone group selector is to isolate hot chunks from obsolete ones. A zone group is a set of open zones mapped with different I/O channels. BIZA confirms the mapping between the logical zones and physical I/O channels with a guess-and-verify mechanism (§ 4.3). Thereafter, the GC avoidance mechanism (§ 4.3) intervenes and makes a final decision on which zone in the zone group to handle writes (❹). GC avoidance mechanism schedules GC events and user write requests to access zones belonging to different I/O channels, which avoids potential resource contentions and thereby mitigates latency spikes. Subsequently, the chunk write tasks are submitted to the ZRWA-aware I/O scheduler (❺, § 4.4). The I/O scheduler dispatches I/O requests (❻) in parallel with a sliding-window-based algorithm, which can avoid write failures induced by I/O reorders. When writes are completed (❼), their completion latency is collected and will be used as the reference for correcting our assumptions on the mapping between zones and I/O channels (§ 4.3). For chunk update, if the chunk is in ZRWA, we update it in place. Otherwise, we choose a new location to write it following the above procedure. Read requests can be simply served by looking up the mapping tables to locate the data and then submitting read commands to the corresponding SSDs.

Mapping table. BIZA employs two mapping tables to locate data and its parities on ZNS SSDs: *Block Mapping Table (BMT)* and *Stripe Mapping Table (SMT)*. As shown in Figure 6, each row in the BMT corresponds to a *logical block (LBN)* and contains two items: (1) a 40-bit *physical address (PA)* that records where the data is stored (the leftmost 8 bits are used as SSD index while the other 32 bits represent in-SSD offsets); and (2) a 32-bit *stripe number (SN)* that represents the stripe that the data belongs to. SN is used as the key for looking up SMT. Each row in SMT consists of m 40-bit physical address (PA) that records the locations of parities in the stripe; m is the degree of fault tolerance specified by the users (e.g., 1 and 2 for RAID 5 and 6, respectively). This design can support at most 4 PB physical storage capacity with a block size of 4 KB (the default setting in BIZA). BIZA maintains BMT and SMT in host DRAM for fast lookups. For a RAID5 consisting of four 4 TB SSDs, BIZA consumes 32 GB memory (0.19%). For crash consistency, BIZA also persists them in the SSD OOB area, similar to prior work [24, 84, 87]. OOB area is scattered across different physical pages of SSDs. BIZA consumes 72-bit OOB area of each physical page to store the BMT entry and SMT entry of the corresponding logical block. The union of a BMT entry and an SMT entry is a 40-bit LBN, a 40-bit PA, and a 32-bit SN. As PA is the self-contained attribute of physical pages, there is no need to store it. Thus, BIZA only needs to persist the 40-bit LBN and 32-bit SN for each block. Note that OOB is an essential component in every type of SSD to store error-correction codes [27]. The OOB quota is typically 64 B per 4 KB. OOB area can be written with the support of NVMe Protection Information feature [69]. SSDs can write OOB area by hitchhiking the same flash programming operation of data writes [24, 25]. Therefore, this method avoids the write amplification caused by persisting the mapping tables.

4.2 Zone Group Selector

Randomly placing chunks among open zones wipes out most of the benefits brought by ZRWA as the frequently updated chunks can only exist in ZRWA momentarily (cf. § 3.1). BIZA solves this problem with a zone group selector that cautiously chooses zone groups for serving chunk writes. Our key insight is that parts of the chunks have high *revenue* (i.e., the chunk will be reaccessed multiple times, *reaccess number* for simplicity [33, 39]) and small *cost* (i.e., short reuse distance [18, 37, 60]). BIZA should distinguish them and avoid writing them to the same zones with obsolete chunks. By doing so, it can keep the high-revenue-low-cost chunks in ZRWA for a longer time and thus get a higher probability of absorbing updates in the write buffer (i.e., ZRWA).

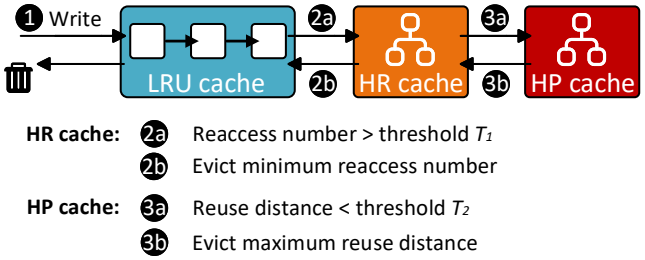


Figure 7. Illustration of ghost-cache-based algorithm.

Algorithm. To this end, BIZA employs a ghost-cache-based algorithm. Ghost cache [67, 94] is a type of cache that only stores the attributes (e.g., reaccess number and reuse distance) of chunks. As illustrated in Figure 7, BIZA first employs an *LRU cache* to filter out the chunks with poor temporal locality, as these chunks typically will not be updated in the near future [73]. When a chunk is hit in the LRU cache (❶), we update its predicted reuse distance and reaccess number according to its last reaccess. Prior work [18, 33, 37, 39] has proposed multiple methods for predicting the future reuse distance and reaccess number. BIZA selects the accumulated reaccess number and weighted moving average of previous reuse distances in the most recent period as the predictions of the reaccess number and reuse distance, respectively. If the predicted reaccess number of a chunk is greater than a threshold (e.g., 3 in BIZA empirically), we consider it a high-revenue target (i.e., may be consistently updated in the future) and promote the chunk to the *high-revenue (HR) cache* (❷a). HR cache is a priority queue [93] that will always evict the chunk with the least reaccess number to LRU cache (❷b). If a chunk in the HR cache has a predicted reuse distance smaller than a threshold (e.g., set as 2 times the size of ZRWA in BIZA), it is considered a high-profit chunk (i.e., high revenue and low cost) and further promoted to the *high-profit (HP) cache* (❸a). HP cache is another priority queue that will evict the chunk with the maximum reuse distance to HR cache (❸b). In all the ghost caches, we use the logical block number (LBN) and stripe number (SN) as the identification for data and parities, respectively.

In tandem with the ghost cache designs, BIZA sets up three types of zone groups: ZRWA-aware, GC-aware, and trivial. The zone group selector selects ZRWA-aware, GC-aware, and trivial zone groups for chunks that are in HP cache, HR cache, and others, respectively. Note that, for partial parities, which will be updated soon when the next write arrives, BIZA always reserves ZRWA for them without the involvement of our ghost-cache-based algorithm.

Rationale. The rationale behind this design is as follows. First, if chunks are in the HP cache, they are expected to be updated multiple times in a short reuse distance. By reserving dedicated zone groups (i.e., ZRWA-aware zone groups) for these chunks and preventing worthless chunks from tainting the ZRWA, BIZA can update the high-profit chunks in ZRWA multiple times with the limited ZRWA resources. Second, although the high-revenue chunks (i.e., in the HR cache) will also be updated multiple times, the cost of reserving them in ZRWA is exorbitant as they have long reuse distances. Therefore, we separate them from the high-profit chunks, thereby stopping them from competing for the scarce ZRWA. However, from another perspective, the high-revenue (but high-cost) chunks are the main source of GC. To be specific, as their overwrites cannot be absorbed within the scarce write buffer, the out-of-place updates of these chunks leave lots of invalid space in zones. By placing them together, BIZA can reclaim more space and reduce the amount of data for migration when collecting victim zones, as most chunks in these zones are already invalidated. By doing so, we shorten the time needed for GC and mitigate the write amplification induced by data migration.

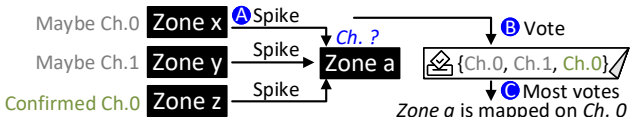


Figure 8. Examples of vote-based algorithm.

4.3 Solutions of GC-induced Latency Spikes

GC avoidance. BIZA alleviates the GC-induced latency spike by serving internal GC events and user I/Os with zones mapped to different I/O channels (i.e., different zones in a zone group). Specifically, when the AFA is idle or its capacity is smaller than a user-specified watermark, BIZA starts GC and the corresponding GC avoidance mechanism. In each GC event, BIZA selects a victim zone and moves the valid chunks in the victim zone to a new zone (called *GC-interfered* zone). GC avoidance mechanism tags the I/O channel mapped with the GC-interfered zone as BUSY. Afterward, if a write request arrives when conducting GC events, BIZA chooses a zone (in a zone group), which is mapped with an I/O channel that has no BUSY tag, as the destination for writing. Note that, the in-place updates can ignore the BUSY tag. This is because ZRWA is typically implemented with on-device DRAM, which is separated from the I/O channels in the flash backbone and thus only suffers minor interference of the GC.

I/O channel detection. AFA is unaware of the mappings between zones and I/O channels as ZNS SSDs make decisions internally for wear leveling. This uncertainty causes a huge obstacle to GC avoidance. AFA may mistakenly cluster zones sharing the same I/O channel to a zone group. Thereafter, during GC, they will serve write requests with unexpected BUSY I/O channel and suffer latency spikes.

Tackling this issue, we propose a guess-and-verify mechanism to confirm the mappings. Our key insight is that most of the time, SSD controllers simply bind zones and I/O channels in a round-robin manner. Prior work [3, 66] also verified this on diverse commodity ZNS SSDs. With this premise,

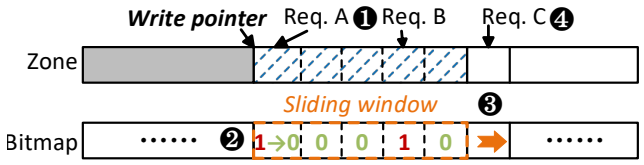


Figure 9. ZRWA-aware I/O scheduler.

BIZA first guesses the mapping based on the round-robin behavior. However, these conjectures are not always accurate, especially when SSD ages. Thus, BIZA uses a vote-based algorithm to verify and correct our assumptions. Figure 8 shows the algorithm. When creating AFA, BIZA confirms the I/O channels for a few zones (e.g., `Zone z`) by taking the zone-to-zone diagnosis (cf. § 3.3). These zones are used as the criterion for following verification and correction. Afterward, BIZA monitors the latency of user write requests online. If latency spikes occur during GC (Ⓐ), in other words, the latency of recently completed requests targeting a zone (e.g., `Zone a`) is significantly higher than the average, BIZA recognizes that it mistakenly handles writes with BUSY I/O channel (e.g., `Channel 0`). Therefore, the target zone is considered as **maybe** mapped on the BUSY I/O channel (e.g., `Channel 0`) and gets a `vote` on this assumption (Ⓑ). If the latency spikes continuously appear on the same zone by multiple times (e.g., 3), BIZA corrects its guesses based on the `votes`. In particular, the predicted I/O channel of the zone is rectified to the channel with the highest number of votes (e.g., `Channel 0`, Ⓒ). Note that if a vote is from an I/O channel that has been confirmed in advance (e.g., `Zone z`), BIZA will trust it without considering other votes.

4.4 ZRWA-aware I/O Scheduler

To avoid write failures induced by I/O reorders (cf. § 3.2) and support asynchronous I/O, BIZA employs a ZRWA-aware I/O scheduler for request submissions. The I/O scheduler is customized for zones that are opened with ZRWA by taking the behaviors of ZRWA (e.g., shift with write pointer) into consideration. The main obstacle to proper I/O scheduling is that ZRWA shifts right implicitly upon the completion of write requests (cf. § 3.1). The host (i.e., I/O scheduler) is unaware of the accurate position of ZRWA after I/O reorders occur. One solution is to directly query the position of the write pointer from ZNS SSDs with ZNS *REPORT* command [71]. However, as I/O reorders appear regularly, frequent REPORT commands will impose a huge burden on the host-side CPU and SSD controller.

Our I/O scheduler tackles this issue by maintaining two data structures, a bitmap and a sliding window, in host-side DRAM for each zone opened with ZRWA. These data structures can track the position of ZRWA accurately with the available information (i.e., I/O submissions and completions). Figure 9 depicts these data structures and their workflows. Each entry in the bitmap corresponds to a block in the zone and indicates whether there is an in-flight write request targeting this block. The sliding window is the portion of the bitmap that represents the ZRWA. Only write requests (e.g., `Request A` and `Request B` in Figure 9) that target the blocks in the sliding window can be submitted to the underlying devices (❶), while other requests (e.g., `Request C`) are postponed. When a write request completes, its corresponding bit in the bitmap is cleared (❷). If the leftmost bit in the sliding window is zero, the I/O scheduler can shift the sliding window rightward (❸) and serve requests that are originally out of the ZRWA (e.g., `Request C`, ❹).

4.5 Implementation

BIZA is implemented as a pluggable device mapper [92] in Linux kernel [57] with 4 K LOC. To support the ZRWA feature and its related management operations (e.g., open zones with ZRWA), we complement the Linux block layer and NVMe driver with 1 K LOC according to the NVMe Zoned Namespaces Command Set Specification 1.1a and NVMe Technical Proposal 4076b [70, 71]. The ZRWA-aware I/O scheduler is also implemented in the Linux block layer as an alternative to the conventional I/O scheduler (e.g., mq-deadline [59]). BIZA requires no hardware modification. All the hardware features used in BIZA (e.g., ZRWA feature and OOB area) are already part of the off-the-shelf commodity ZNS SSDs [14, 16, 31, 75].

5 Evaluation

5.1 Experimental Setup

Testbeds. We evaluate our BIZA designs on a server equipped with a 26-core Intel Xeon 5320 CPU [32] and 512 GB DDR4 DRAM. All experiments are conducted on Ubuntu 22.04 LTS with Linux kernel v5.15 [57]. We construct BIZA and its counterparts with four commodity ZNS or block-interface SSDs as RAID 5. We choose Western Digital ZN540 4 TB SSD [16] as the representative of ZNS SSDs. Each zone in ZN540 SSD has a capacity of 1077 MB and can be opened with a 1 MB ZRWA. ZN540 SSD allows at most 14 open zones. The peak throughput of ZN540 SSD is 2170 MB/s and 3265 MB/s for write and read, respectively. For AFAs consisting of conventional block-interface SSDs, we employ Western Digital SN640 4 TB SSD [15], which is developed on a similar hardware basis as ZN540 SSD and thus yields comparable throughput, i.e., 2250 MB/s and 3331 MB/s for write and read, 4% and 2% higher than ZN540 SSD, respectively. We ascribe the minor performance discrepancy to a difference in firmware maturity between these two devices [45]. The key configurations are listed in Table 5.

AFA platforms. We compare BIZA with two AFA settings that consist of ZNS SSDs and expose block interface. We also examine two representative AFA designs, i.e., RAIZN [45] and mdraid [58], for reference. (1) `BIZA`:

Host System	**ZNS SSD**	Western Digital Ultrastar DC ZN540	**Conv. SSD**	Western Digital Ultrastar DC SN640
		Cap.: 4TB, 1077 MB / logical zone		Cap.: 4TB, 4 KB / logical block
		ZRWA: 1 MB × 14 open zones		Similar hw. basis as ZNS SSD
		R/W Bw.: 3265 / 2170 MB/s		R/W Bw.: 3331 / 2250 MB/s
	CPU	Intel Xeon 5320	**DRAM**	8 x 64 GB / DDR4
		26 Core / 2.2 Ghz		
		with hyper-threading		
Software	**OS**	Ubuntu 22.04	**fio**	v3.30
	Kernel	Linux v5.15	**perf**	v5.15

Table 5. System configurations.

Collections	**FIU**			**Microsoft MSRC**	
Workloads	casa	online	ikki	proj	web
Write ratio (%)	98.6	67.1	92.8	3.0	45.9
Avg. read size (KB)	13.3	4	10.2	6.2	46.4
Avg. write size (KB)	4	4	4	18.5	9.8
Collections	**Microsoft MSPC**		**SYSTOR**		**Tencent Block Storage**
Workloads	DAP	MSNFS	lun0	lun1	tencent
Write ratio (%)	51.9	31.5	17.6	38.0	52.9
Avg. read size (KB)	64	9.8	30.4	20.6	31.5
Avg. write size (KB)	121.3	13.3	9.3	12.3	39.2

Table 6. Workload characteristics.

a block-interface ZNS AFA that includes all the designs proposed in this paper; (2) `RAIZN`: constructing AFA from ZNS SSDs and also exposing ZNS-interface to applications, which only supports sequential write; (3) `dmzap+RAIZN`: stacking block-ZNS-interface adapter (dm-zap) on top of `RAIZN`; (4) `mdraid+dmzap`: employing dm-zap in each ZNS SSD to convert ZNS interface to block interface and then stacking a conventional block-interface AFA engine (mdraid) on top of them to construct AFA; (5) `mdraid+ConvSSD`: employing mdraid to manage conventional block-interface SSDs. Note that the original implementation of dm-zap [13] only utilizes a single zone simultaneously, which cannot exhaust the inter-zone parallelism (cf. § 3.3). We've revised it to write all open zones in parallel. Moreover, the lock issues of mdraid are widely denounced [89, 100]. We've integrated a state-of-the-art work [100] in mdraid to evolve it for higher performance.

Workloads. We conduct evaluations with workloads from various benchmark suites. Specifically, we measure the performance of different AFA engines by employing fio v3.30 [2] to execute microbenchmarks. By default, we employ one job to generate asynchronous I/O requests and set the I/O depth as 32. Moreover, we evaluate BIZA with diverse I/O traces collected from productions, including FIU [79], Microsoft MSRC [80], Microsoft MSPC [36], SYSTOR [49], and Tencent Block Storage [102]. These traces have different ratios of reads and writes with varied I/O sizes, which is helpful for thoroughly examining BIZA in different I/O patterns. Table 6 summarizes the key characteristics of the selected workloads. We also conduct comparisons on ZNS-friendly F2FS [48] and RocksDB [17] with `filebench` [21] and `db_bench` [19], respectively, which demonstrates the end-to-end performance improvement brought by our designs. We conduct all the evaluations by repeating the experiments 10 times and showing the average results.

5.2 Overall Performance

Write performance in microbenchmark. Figure 10 shows the write throughput and average latency of different tested platforms in microbenchmarks. We vary the access patterns from sequential writes to random writes and examine different I/O sizes from 4 KB to 192 KB. There are no bars for `RAIZN` in random write tests due to its lack of random write

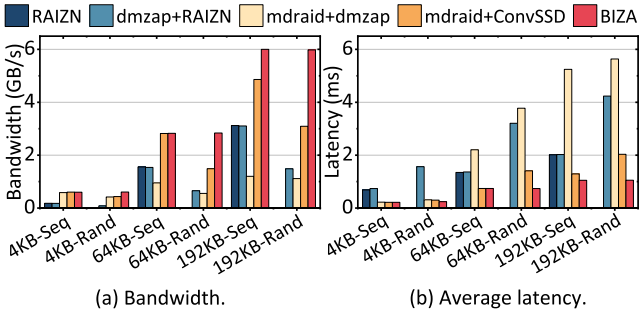


Figure 10. Write performance in microbenchmark.

support. The ideal write throughput for our 4-SSD RAID 5 constructed from ZNS and conventional SSD are 6.4 and 6.6 GB/s, respectively (i.e., 3 SSDs serve data write while the other one handles parities). Both mdraid-based methods (i.e., `mdraid+dmzap` and `mdraid+ConvSSD`) have higher sequential performance than random ones. This is because mdraid takes an in-host-DRAM write buffer to merge multiple sequential write requests and handles them simultaneously. `dmzap+RAIZN` achieves almost the same sequential write throughput as `RAIZN`. However, it can only exploit 47.7% of ideal performance due to centralized designs in RAIZN (cf. § 3.3). `mdraid+dmzap` lags far behind `mdraid+ConvSSD` in 64 KB tests. This is because, when handling write requests, mdraid splits the 64 KB data in page size (4 KB) and submits the 4 KB requests to underlying SSDs. 4 KB requests cannot fully exploit the throughput of SSDs. Conventional SSD is not affected by the splitting, because the Linux block layer [6] can merge the 4 KB requests into large-size requests again. However, dm-zap is unable to perform this, as it only allows one in-flight 4 KB write. 192 KB tests show similar results. Interestingly, even employing the state-of-the-art implementation of mdraid, `mdraid+ConvSSD` cannot exhaust the throughput of SSDs (in 192 KB tests) because of the unsolved design issues in mdraid [89, 100]. In contrast, `BIZA` achieves 2.7×, 2.5×, and 0.4× higher bandwidth than `dmzap+RAIZN`, `mdraid+dmzap`, and `mdraid+ConvSSD` in all scenarios on average and almost exhausts the throughput of 4 ZNS SSDs (92.2% of the ideal). This is because the key techniques in `BIZA`, such as decentralized metadata updates

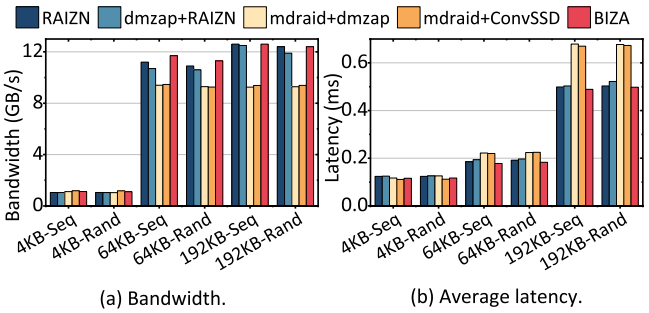


Figure 11. Read performance in microbenchmark.

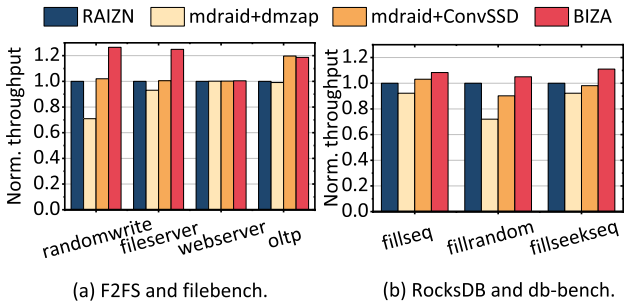


Figure 13. Comparison in ZNS-friendly applications.

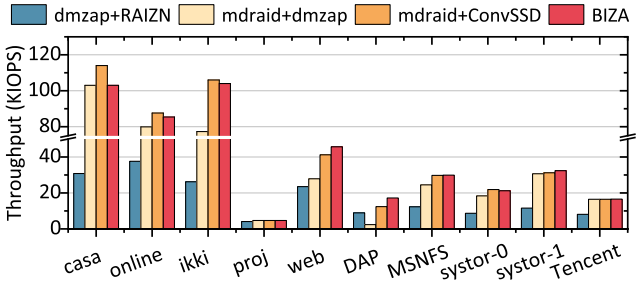


Figure 12. Throughput comparison in I/O traces.

and ZRWA-aware I/O scheduler, enable it to exploit all parallelisms within ZNS SSDs. This superiority also exists in average latency comparisons (cf. Figure 10b). For example, BIZA outperforms RAIZN by 53.8%.

Read performance in microbenchmark. The comparison of read performance is shown in Figure 11. All the platforms have similar throughput and average latency for 4 KB reads. This is because these designs take roughly the same read path, that is, lookup a mapping table to locate data in the flash backbone and then read it out. mdraid+dmzap and mdraid+ConvSSD lag behind BIZA and dmzap+RAIZN in 64 KB and 192 KB scenarios, because of the software bottleneck in mdraid, echoing the findings in prior work [45, 89, 100]. Both BIZA and dmzap+RAIZN achieve comparable read throughput as the ideal (i.e., 12.8 GB/s when all 4 ZNS SSDs handle requests with maximum bandwidth) since there is no obvious performance bottleneck on the read path.

Performance in I/O traces. Figure 12 illustrates the throughput of different AFA platforms in diverse I/O traces. dmzap+RAIZN lags behind mdraid+dmzap by 98.1% in all workloads, on average, because of the centralized metadata zone design in RAIZN. Based on mdraid+dmzap, BIZA further improves the average write throughput by 76.5% as BIZA can better exploit the intra-zone parallelism of ZNS SSDs. BIZA also achieves comparable throughput to the conventional solution (i.e., mdraid+ConvSSD). This result proves that our design can be utilized as an alternative to the available AFAs based on block-interface SSDs while benefiting the reduced write amplification and low tail latency (cf. § 5.4 and § 5.5). We attribute the minor performance lag of BIZA to mdraid+ConvSSD in casa, online, and ikki workloads to two reasons. First, these workloads have small average write sizes (cf. Table 6), which do not impose burdens on the parallelism of SSDs. Therefore, they rarely reap the benefits brought by BIZA. Second, the conventional SSDs (SN640) achieve higher throughput than ZNS SSDs (ZN540, cf. § 5.1).

5.3 Real-World Applications

File system. To demonstrate the superiority of BIZA on applications, we first conduct evaluations on F2FS [48], a log-structured file system, with filebench [21]. Although F2FS is commonly considered a ZNS-friendly file system that can exploit the capability of ZNS SSDs [45], it cannot support ZNS interface alone and requires at least a two-zone-sized block-interface space to store metadata [5]. Fortunately, ZN540 SSD has internally integrated a 4 GB block-interface storage area. We first orchestrate the 4 GB areas from 4 ZN540 SSDs as a RAID 5 by employing mdraid and then construct F2FS on both the block-interface RAID 5 and RAIZN. For simplicity, we still call this prototype as RAIZN. Note that the 4 GB area doesn't yield extra performance benefits as it shares the same flash backbone with the ZNS-interface area. We use four representative benchmarks from filebench including write-dominated (randomwrite), write-read-mixed (fileserver and oltp), and read-dominated (webserver). Figure 13a illustrates the throughputs of different AFA settings. We normalized the results to that of RAIZN. Thanks to our decentralized design and I/O scheduler, BIZA can exploit the internal parallelism of ZNS SSDs and thereby achieve the highest throughput in all workloads. To be specific, BIZA outperforms RAIZN by 26.6%, 24.9%, and 18.7% in randomwrite, fileserver, and oltp, respectively. The improvement is minor in webserver, as write requests only account for 4.8% in this workload. Note that our designs focus on optimizing the write path while taking nearly the same read path as other AFA platforms.

Key-value store. We further evaluate BIZA on RocksDB [20], a popular key-value store, with db_bench [19]. We run RocksDB on F2FS file system. Both fillseq and fillrandom

are write-dominated workloads that write values in sequential or random key order, while `fillseekseq` first writes values sequentially and then reads them by seeking each key. We set the key and value sizes as 16 B and 1 KB, respectively. Figure 13b shows the comparison of normalized throughput. Although on top of the ZNS-friendly RocksDB and F2FS, `BIZA` can still outperform `RAIZN`, echoing the findings of prior evaluations. Specifically, `BIZA` outperforms `RAIZN` by up to 10.5%. For all workloads, on average, the improvement stands at 8.0%.

5.4 Write Amplification Reduction

To evaluate how much our novel zone group selector design can mitigate the damage of write amplification with ZRWA, we set an experiment to analyze the number of writes (i.e., write counts) *fallen on flash backbone* (i.e., excludes the writes absorbed by write buffer). We normalized the results to the number of writes sent by the user. `BIZA` opens 14 zones (the maximum allowed number in ZN540 SSD) for each ZNS SSD and thereby can exploit 56 MB of ZRWA for all 4 SSDs. We set the sizes of the LRU cache, HR cache, and HP cache as 1048576, 262144, and 16384 entries empirically, which can track 4 GB, 1 GB, and 64 MB of chunks, respectively. Note that all these caches are ghost caches (cf. § 4.2) that only maintain the attributes (i.e., reaccess number and reuse distance) of chunks. Therefore, the memory footprint of these caches is minor (i.e., 7.6 MB with our settings). Both mdraid and RAIZN employ an in-host-DRAM write buffer (named stripe cache [45]) for absorbing updates. These designs hamper the fault tolerance of AFA, as chunks that stay in DRAM will be lost after system failures. However, for fair comparisons, we still equip them with the same size (i.e., 56 MB) of write buffer. Note that, dm-zap is a simple address translation layer that does not produce extra write requests. Therefore, `mdraid+dmzap` has similar results as `mdraid+ConvSSD`. We omit the latter due to space limitations. In addition, `BIZAw/oSelector` means the scenario where we do not employ the zone group selector (cf. § 4.2) in BIZA. To be specific, it randomly selects zone groups to serve write requests. Finally, `no cache` and `ideal` represent the scenario where no update and all updates are absorbed in the write buffer, respectively.

Figure 14 illustrates the results. The lighter and darker shaded segments of each bar represent parity and data writes, respectively. Compared with `no cache`, `dmzap+RAIZN` and `mdraid+dmzap` successfully reduce 42.5% and 4.4% parity writes, respectively. This is because both mdraid and RAIZN can gather multiple scattered write requests in their write buffer and handle them as a whole. This method reduces the need for parity computations and parity writes [45, 58, 89]. Note that, in RAIZN, all write buffer is used for buffering parity whilst mdraid uses write buffer for both parity and data. Therefore, `dmzap+RAIZN` reduces more parity writes but fewer data writes than `mdraid+dmzap`. `BIZAw/oSelector`

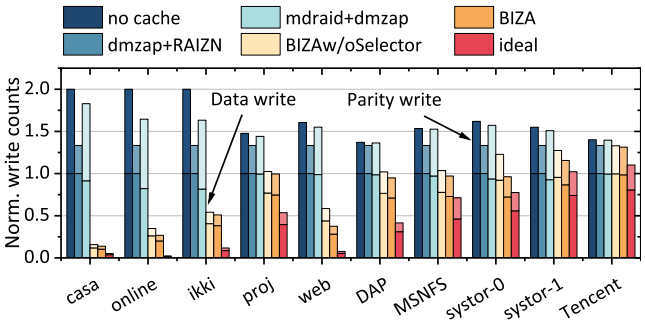


Figure 14. Comparison of write amplification.

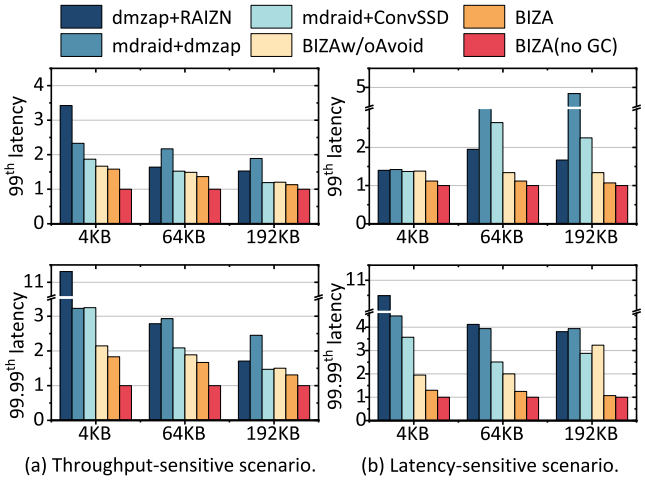


Figure 15. Comparison of tail latency after GC starts.

outperforms `mdraid+dmzap` by 32.5% among all workloads, on average, in terms of data writes. This is because mdraid needs to periodically flush data from the volatile in-host-DRAM write buffer to SSDs as compensation for the fault tolerance issue, which causes more flash writes. In contrast, BIZA relies on the on-device non-volatile ZRWA to absorb updates, which avoids periodical flushing. Benefiting from our ghost-cache-based zone group selection algorithm (cf. § 4.2), `BIZA` succeeds in reserving the high-profit chunks in ZRWA, thereby further shrinking the number of writes by 12.6%, compared with `BIZAw/oSelector`. `BIZA` reduces write amplification less in some workloads (e.g., `tencent`) because these workloads have larger reuse distances, making it harder for BIZA to utilize the temporal locality with the limited size of ZRWA. For example, 90.2% of chunks in `tencent` have reuse distances larger than 56 MB, while the ratio is only 8.3% in `casa`.

5.5 Avoiding the GC-induced Latency Spike

Figure 15 shows the 99^{th} and 99.99^{th} sequential write latency of different AFA settings after GC occurs. We set the I/O depth as 32 and 1 to mimic the throughput-sensitive and latency-sensitive scenarios, respectively. `BIZAw/oAvoid`

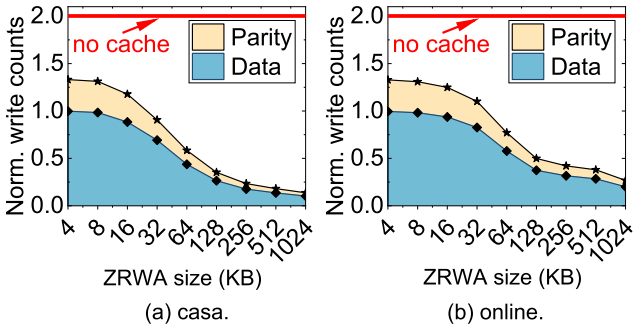


Figure 16. Sensitivity analysis.

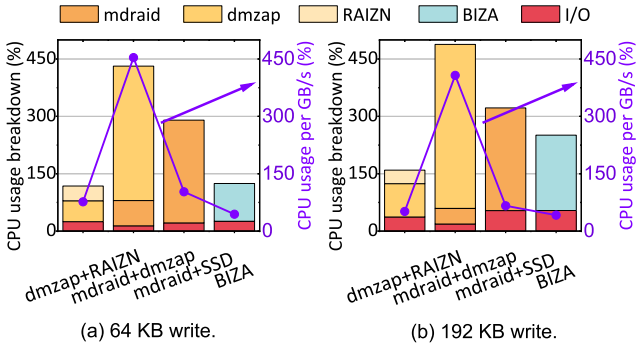


Figure 17. Comparison of CPU overhead.

means removing the GC avoidance mechanism (cf. § 4.3) from BIZA, therefore it may handle user I/O and internal GC events with the same I/O channels. The results have been normalized to `BIZA(no GC)`, which represents the ideal case where no GC happens in BIZA. Affected by the GC, all AFA platforms suffer higher tail latency. For instance, `BIZAw/oAvoid` experiences 1.0×, 0.9×, and 1.4× higher 4 KB, 64 KB, and 192 KB 99.99th write latency than the ideal (i.e., `BIZA(no GC)`), respectively, in both scenarios on average. Compared with `BIZAw/oAvoid`, `BIZA` alleviates the latency spikes by 27.4% in throughput-sensitive scenario. This is because, with our I/O channel detection technique and GC avoidance mechanism, BIZA can serve user I/O and GC events properly with diverse I/O channels, which mitigates the interference between them. This improvement further expands to 74.9% in the latency-sensitive scenario. This is because, in this scenario, BIZA has more idle storage resources for better isolating GC and user I/O, thereby reducing more latency spikes on user I/O.

5.6 Sensitivity Analysis: Different Sizes of ZRWA

BIZA harnesses ZRWA to buffer both the frequently updated data and parities. We vary the ZRWA size per open zone from 4 KB to 1024 KB to examine the benefits brought by different sizes of ZRWA. Figure 16 shows the variation of write count with ZRWA size in `casa` and `online` workloads. The results have been normalized to the number of writes sent by the user. Note that all writes in `casa` and `online` are 4 KB, which is the same as the chunk size of BIZA. Therefore, without cache (i.e., red line in Figure 16), BIZA writes 1× data and 1× parities to the flash backbone, since every data write triggers a parity write. 2/3 of these parities are partial parities while the remaining 1/3 are final parities of stripes. In both workloads, data writes and parity writes decrease with increasing ZRWA size, which demonstrates our zone group selector can absorb more writes with larger buffer. Interestingly, when the ZRWA size is 4 KB, no data updates are absorbed, while all partial parity writes are eliminated. This is because BIZA employs the only-one-chunk-sized ZRWA to accommodate partial parities that will be refreshed soon (i.e., when the next write arrives).

5.7 CPU Overhead

We examine the CPU overhead of different implementations in 64 and 192 KB sequential write patterns. We employ perf v5.15 [91] to capture CPU cycles. We define CPU efficiency as the CPU usage used to achieve per GB/s bandwidth. Each 100% CPU usage means full exploitation of a CPU core. Figures 17 illustrates the results. `mdraid`, `dmzap`, `RAIZN`, and `BIZA` represent the software overheads of each component, while `I/O` is CPU ticks consumed by SSD I/O. `dmzap` is the main source of CPU overhead for `dmzap+RAIZN` and `mdraid+dmzap` and accounts for 50.4 % and 84.7% of the total CPU usage in these two settings, respectively. This is because dm-zap realizes one in-flight write control with a centralized lock [82]. This lock solves the I/O reorder issue (cf. § 3.2) but causes a huge waste of CPU as CPU spins purposelessly most of the time. Compared with `dmzap+RAIZN`, `BIZA` takes 31.5% more CPU usages to parallelize I/O services, in 64 and 192 KB tests, on average, and thereby achieves 88.5% higher throughput (cf. Figure 10a). Consequently, `BIZA` outperforms all other AFA platforms in terms of CPU efficiency (cf. the right axis in Figure 17).

6 Related Work and Discussion

Large zone versus small zone. Available ZNS SSDs can be classified into two types, large zone and small zone, according to their mapping policy between zones and I/O channels. Large-zone ZNS SSDs (e.g., ZN540 [16]) map a single zone to a set of flash blocks from tens of flash dies whilst the zones in small-zone ZNS SSDs [29, 30, 78] only consist of one or few flash blocks. Although we focus on the large-zone ZN540 SSD in this paper, we argue that our design can be employed on small-zone ZNS SSDs as they also equip parallel I/O resources [66] that can be used for GC avoidance. Another requirement of BIZA designs is the hardware support for ZRWA feature. Table 2 lists this support in diverse commodity ZNS SSDs from different vendors [14, 16, 31, 75], which is evidence of the popularity and prospect of ZRWA.

Diverse I/O interfaces of SSDs. Extensive discussions [4, 5, 7, 68] have been taken on the I/O interface of SSDs

since the last decade. We classify them into two categories based on their trade-offs among compatibility, cost, and openness. Multi-stream SSD [4] and its successor, FDP SSD [68], still expose the compatible block interface to upper-layer software. They are aimed to offer isolated environments to different I/O streams by utilizing extra hardware resources (e.g., ARM cores and DRAM) to process I/O hints sent by the host. On the other hand, open-channel SSD [7] and its successor, ZNS SSD [5], are more cost-efficient and flexible. They simplify the SSD architecture and rely on host-side management to mitigate write amplification and achieve high performance. By shifting parts of SSD internal tasks to the host side, they provide exclusive opportunities for cross-layer optimization. Therefore, in this paper, we opt to construct AFA with ZNS SSDs.

Alleviating the impact of GC. Multiple studies [11, 26, 34, 41, 46, 52, 53, 55, 95, 97] have been proposed to alleviate the performance degradation caused by GC in AFA. SWAN [42], FusionRAID [34], and IODA [53] mitigate latency spikes by redirecting I/O requests to other SSDs that are not in GC. In ZNS SSD scenarios, the unique challenge is that, without confirming the mappings between zones and I/O channels, we cannot properly schedule GC and I/O to independent I/O resources. Tackling this issue, BIZA employs a guess-and-verify mechanism to diagnose the mappings online.

Inter-zone parallelism. eZNS [66] tries to exploit inter-zone parallelism by simply writing multiple zones concurrently. However, blindly utilizing zones from identical I/O channels could not enjoy inter-zone parallelism (cf. `Scenario 2` in Table 3). Tackling this issue, we propose a guess-and-verify mechanism to first confirm the mappings between zones and I/O channels, and thereafter handle I/O requests with parallel zones from diverse I/O channels for higher performance.

Write amplification reduction with ZNS SSDs. Great efforts [5, 12, 35, 51] have been taken to alleviate the write amplification with ZNS SSDs. ZenFS [5] and LL-compaction [35] achieve this by gathering data from the same SSTable (thereby having similar lifetimes) in the same zones. LSM_ZGC [12] diminishes write amplification with an efficient GC algorithm that exploits the characteristics of log-structured file systems. All these designs are tailored for specified applications (e.g., RocksDB [20]). In contrast, BIZA mitigates write amplification by absorbing updates in ZRWA, which is a general-purpose design for all scenarios.

Considering ZRWA as a cache. As ZRWA is an abstraction of write buffer (i.e., cache), prior work that targets improving the efficiency (or hit rate) of caches [22, 72, 73, 98] could be utilized for better exploiting the ZRWA. However, there are two unique challenges in ZRWA scenarios. First, ZRWA has very limited sizes (e.g., 14 MB in ZN 540[16]), which can seldom utilize the temporal locality of existing workloads. In comparison, prior work [67, 73, 98] usually assumes a large buffer size. Second, ZRWA always evicts data in the leftmost position (cf. § 3.1), while prior work [72] allows evicting any entry from the write buffer. Therefore, we have to choose the admission and eviction policies cautiously when adopting cache designs. In this paper, we take a ghost-cache-based design [28, 67, 94] for ZRWA, which has been proven to be useful (cf. § 5.4). We leave the exploration on more cache algorithms for future work.

Future ZNS designs. We expect future ZNS SSDs can clearly point out the mappings between I/O channels and zones, which helps users exploit inter-zone parallelism (cf. § 3.3). One way to achieve this is piggybacking the mappings in NVMe completion queue entries (CQE) [69] of OPEN commands. Moreover, the emerging CXL techniques, that allow cache coherence and fast synchronization between the host and peripheral devices, may be helpful for making the shift of write pointer explicit, thereby enabling better I/O scheduling for exhausting intra-zone parallelism.

7 Conclusion

A deep analysis reveals that the existing interface choices of AFA are defective in terms of endurance, performance, and compatibility. Based on these lessons, we propose BIZA, a self-governing ZNS AFA that can benefit from the openness of ZNS interface whilst exposing the user-friendly block interface to upper-layer applications. BIZA achieves these by fully exploiting the emerging ZRWA feature and the internal parallelisms of ZNS SSDs. Our evaluation results reveal that BIZA mitigates write amplification by 42.7%, enhances throughput by 93.2%, and decreases tail latency by 62.8%, compared to the state-of-the-art AFA solutions.

Acknowledgement

We sincerely thank the anonymous reviewers and our shepherd, Angela Demke Brown, for their insightful feedback. We also thank Western Digital for their product support and technical expertise. This work is mainly supported by the National Key Research and Development Program of China under Grant No. 2023YFB4502702 and the Natural Science Foundation of China under Grant No. 62332021 and 62472007. Dr. Yang is partly supported by the Research Grants Council of Hong Kong SAR (Project Nos. CUHK14210320 and CUHK14218522). Dr. Cao is partly supported by the US National Science Foundation under grant CNS-2412436. Dr. Li is partly supported by the National Natural Science Foundation of China under Grant No. 62202396. Dr. Jung is partly supported by NRF2021R1A2C4001773, IITP 2021-0-00524, IITP 2022-0-00117, IITP RS-2023-00221040, G01230749, IITP-2024-RS-2023-00256472, KAIST IDEC, and Samsung Electronics. Shushu Yi and Jie Zhang are affiliated with both Peking University and Zhongguancun Laboratory. The corresponding author is Jie Zhang.

References

[1] Ahmed Izzat Alsalibi, Sparsh Mittal, Mohammed Azmi Al-Betar, and Putra Bin Sumari. A survey of techniques for architecting slc/mlc/tlc hybrid flash memory–based ssds. *Concurrency and Computation: Practice and Experience*, 30(13):e4420, 2018.

[2] J. Axboe. Flexible i/o tester. https://github.com/axboe/fio.

[3] 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*, pages 79–85, 2022.

[4] Janki Bhimani, Jingpei Yang, Zhengyu Yang, Ningfang Mi, NHV Krishna Giri, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. Enhancing ssds with multi-stream: What? why? how? In *2017 IEEE 36th International Performance Computing and Communications Conference (IPCCC)*, pages 1–2. IEEE, 2017.

[5] 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 (USENIX ATC 21)*, pages 689–703, 2021.

[6] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th international systems and storage conference*, pages 1–10, 2013.

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

[8] Vladimir Braverman, Rafail Ostrovsky, and Carlo Zaniolo. Optimal sampling from sliding windows. In *Proceedings of the twenty-eighth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 147–156, 2009.

[9] John Canny, Huasha Zhao, Bobby Jaros, Ye Chen, and Jiangchang Mao. Machine learning at the limit. In *2015 IEEE International Conference on Big Data (Big Data)*, pages 233–242. IEEE, 2015.

[10] Zhichao Cao. *High-Performance and Cost-Effective Storage Systems for Supporting Big Data Applications*. PhD thesis, University of Minnesota, 2020.

[11] Tzi-cker Chiueh, Weafon Tsao, Hou-Chiang Sun, Ting-Fang Chien, An-Nan Chang, and Cheng-Ding Chen. Software orchestrated flash array. In *Proceedings of International Conference on Systems and Storage*, pages 1–11, 2014.

[12] Gunhee Choi, Kwanghee Lee, Myunghoon Oh, Jongmoo Choi, Jhuyeong Jhin, and Yongseok Oh. A new {LSM-style} garbage collection scheme for {ZNS}{SSDs}. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*, 2020.

[13] Western Digital Corporation. dm-zap: Host-based ftl for zns ssds. https://github.com/westerndigitalcorporation/dm-zap, 2021.

[14] DapuStor. Dapustor j5500z. https://en.dapustor.com/product.html, 2023.

[15] Western Digital. Ultrastar dc sn640 nvme ssd. https://www.westerndigital.com/support/wdc/data-center-drives/ssd/ultrastar-dc-sn640.

[16] Western Digital. Zns ssds just got real – ultrastar® dc zn540 now sampling. https://blog.westerndigital.com/zns-ssd-ultrastar-dc-zn540-sampling/.

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

[18] Nam Duong, Dali Zhao, Taesu Kim, Rosario Cammarota, Mateo Valero, and Alexander V Veidenbaum. Improving cache management policies using dynamic reuse distances. In *2012 45Th annual IEEE/ACM international symposium on microarchitecture*, pages 389–400. IEEE, 2012.

[19] facebook. db bench. https://github.com/facebook/rocksdb/wiki/Benchmarking-tools.

[20] Facebook. Rocksdb. http://rocksdb.org/.

[21] filebench. A model based file system workload generator. https://github.com/filebench/filebench.

[22] Binny S Gill and Dharmendra S Modha. Sarc: Sequential prefetching in adaptive replacement cache. In *USENIX Annual Technical Conference, General Track*, pages 293–308, 2005.

[23] Donghyun Gouk, Miryeong Kwon, Jie Zhang, Sungjoon Koh, Wonil Choi, Nam Sung Kim, Mahmut Kandemir, and Myoungsoo Jung. Amber: Enabling precise full-system simulation with detailed modeling of all ssd resources. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 469–481. IEEE, 2018.

[24] Aayush Gupta, Raghav Pisolkar, Bhuvan Urgaonkar, and Anand Sivasubramaniam. Leveraging value locality in optimizing {NAND} flash-based {SSDs}. In *9th USENIX Conference on File and Storage Technologies (FAST 11)*, 2011.

[25] Kyuhwa Han, Hyukjoong Kim, and Dongkun Shin. Wal-ssd: Address remapping-based write-ahead-logging solid-state disks. *IEEE Transactions on Computers*, 69(2):260–273, 2019.

[26] Mingzhe Hao, Gokul Soundararajan, Deepak Kenchammana-Hosekote, Andrew A Chien, and Haryadi S Gunawi. The tail at store: A revelation from millions of hours of disk and {SSD} deployments. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 263–276, 2016.

[27] Ping Huang, Pradeep Subedi, Xubin He, Shuang He, and Ke Zhou. {FlexECC}: Partially relaxing {ECC} of {MLC}{SSD} for better cache performance. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 489–500, 2014.

[28] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. Improving flash-based disk cache with lazy adaptive replacement. *ACM Transactions on Storage (TOS)*, 12(2):1–24, 2016.

[29] SK hynix. Sk hynix demonstrates industry's first zns-based ssd solution for data centers. https://news.skhynix.com/sk-hynix-demonstrates-industrys-first-zns-based-ssd-solution-for-data-centers-2/.

[30] Minwoo Im, Kyungsu Kang, and Heonyoung Yeom. Accelerating rocksdb for small-zone zns ssds by parallel i/o mechanism. In *Proceedings of the 23rd International Middleware Conference Industrial Track*, pages 15–21, 2022.

[31] Inspur. Inspur ns8600g. https://www.inspur.com/lcjtww/2526546/index.html, 2023.

[32] Intel. Intel® xeon® gold 5320 processor. https://www.intel.com/content/www/us/en/products/sku/215285/intel-xeon-gold-5320-processor-39m-cache-2-20-ghz/specifications.html.

[33] Aamer Jaleel, Kevin B Theobald, Simon C Steely Jr, and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). *ACM SIGARCH computer architecture news*, 38(3):60–71, 2010.

[34] Tianyang Jiang, Guangyan Zhang, Zican Huang, Xiaosong Ma, Junyu Wei, Zhiyue Li, and Weimin Zheng. Fusionraid: Achieving consistent low latency for commodity ssd arrays. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 355–370, 2021.

[35] Jeeyoon Jung and Dongkun Shin. Lifetime-leveling lsm-tree compaction for zns ssd. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pages 100–105, 2022.

[36] Swaroop Kavalanekar, Bruce Worthington, Qi Zhang, and Vishal Sharda. Characterization of storage workload traces from production windows servers. In *2008 IEEE International Symposium on Workload Characterization*, pages 119–128. IEEE, 2008.

[37] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. Cache replacement based on reuse-distance prediction. In *2007 25th*

International Conference on Computer Design, pages 245–250. IEEE, 2007.

[38] Linux kernel. Nvme driver. https://github.com/torvalds/linux/tree/master/drivers/nvme.

[39] Mazen Kharbutli and Yan Solihin. Counter-based cache replacement algorithms. In *2005 International Conference on Computer Design*, pages 61–68. IEEE, 2005.

[40] Aleksandr Khasymski, M Mustafa Rafique, Ali R Butt, Sudharshan S Vazhkudai, and Dimitrios S Nikolopoulos. On the use of gpus in realizing cost-effective distributed raid. In *2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 469–478. IEEE, 2012.

[41] Byungseok Kim, Jaeho Kim, and Sam H Noh. Managing array of {SSDs} when the storage device is no longer the performance bottleneck. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, 2017.

[42] Jaeho Kim, Kwanghyun Lim, Youngdon Jung, Sungjin Lee, Changwoo Min, and Sam H Noh. Alleviating garbage collection interference through spatial separation in all flash arrays. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 799–812, 2019.

[43] Jiho Kim, Myoungsoo Jung, and John Kim. Decoupled ssd: Rethinking ssd architecture through network-based flash controllers. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–13, 2023.

[44] Jiho Kim, Seokwon Kang, Yongjun Park, and John Kim. Networked ssd: Flash memory interconnection network for high-bandwidth ssd. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 388–403. IEEE, 2022.

[45] 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, Volume 2*, pages 660–673, 2023.

[46] Youngjae Kim, Sarp Oral, Galen M Shipman, Junghee Lee, David A Dillow, and Feiyi Wang. Harmonia: A globally coordinated garbage collector for arrays of solid-state drives. In *2011 IEEE 27th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–12. IEEE, 2011.

[47] Chang-Hung Lee, Cheng-Ru Lin, and Ming-Syan Chen. Sliding-window filtering: an efficient algorithm for incremental mining. In *Proceedings of the tenth international conference on Information and knowledge management*, pages 263–270, 2001.

[48] 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.

[49] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *Proceedings of the 10th ACM International Systems and Storage Conference*, pages 1–11, 2017.

[50] Gyusun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W Lee, and Jinkyu Jeong. Asynchronous {I/O} stack: A low-latency kernel {I/O} stack for {Ultra-Low} latency {SSDs}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 603–616, 2019.

[51] Hee-Rock Lee, Chang-Gyu Lee, Seungjin Lee, and Youngjae Kim. Compaction-aware zone allocation for lsm based key-value store on zns ssds. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pages 93–99, 2022.

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

[53] Huaicheng Li, Martin L Putra, Ronald Shi, Xing Lin, Gregory R Ganger, and Haryadi S Gunawi. loda: A host/device co-design for strong predictability contract on modern flash storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 263–279, 2021.

[54] Qiang Li, Qiao Xiang, Yuxin Wang, Haohao Song, Ridi Wen, Wenhui Yao, Yuanyuan Dong, Shuqi Zhao, Shuo Huang, Zhaosheng Zhu, et al. More than capacity: performance-oriented evolution of pangu in alibaba. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 331–346, 2023.

[55] Yongkun Li, Helen HW Chan, Patrick PC Lee, and Yinlong Xu. Elastic parity logging for ssd raid arrays. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 49–60. IEEE, 2016.

[56] Linux. dm-zoned. https://docs.kernel.org/admin-guide/device-mapper/dm-zoned.html.

[57] Linux. Linux kernel v5.15. https://github.com/torvalds/linux/releases/tag/v5.15.

[58] Linux. mdraid layer. https://github.com/torvalds/linux/tree/master/drivers/md.

[59] Linux. mq-deadline scheduler. https://github.com/torvalds/linux/blob/master/block/mq-deadline.c.

[60] Weiguang Liu, Jinhua Cui, Junwei Liu, and Laurence T Yang. Mlcache: A space-efficient cache scheme based on reuse distance and machine learning for nvme ssds. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

[61] Yu-Chia Liu and Hung-Wei Tseng. Nds: N-dimensional storage. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 28–45, 2021.

[62] Yu Mao, Jiguang Wan, Yifeng Zhu, and Changsheng Xie. A new parity-based migration method to expand raid-5. *IEEE Transactions on Parallel and Distributed Systems*, 25(8):1945–1954, 2013.

[63] Avantika Mathur, Mingming Cao, Suparna Bhattacharya, Andreas Dilger, Alex Tomas, and Laurent Vivier. The new ext4 filesystem: current status and future plans. In *Proceedings of the Linux symposium*, volume 2, pages 21–33. Citeseer, 2007.

[64] Rino Micheloni, Alessia Marelli, Kam Eshghi, and G Wong. Ssd market overview. *Inside Solid State Drives (SSDs)*, pages 1–17, 2013.

[65] Microsoft. Overview of fat, hpfs, and ntfs file systems.

[66] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. {eZNS}: An elastic zoned namespace for commodity {ZNS}{SSDs}. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*, pages 461–477, 2023.

[67] Yuanjiang Ni, Ji Jiang, Dejun Jiang, Xiaosong Ma, Jin Xiong, and Yuangang Wang. S-rac: Ssd friendly caching for data center workloads. In *Proceedings of the 9th ACM International on Systems and Storage Conference*, pages 1–12, 2016.

[68] NVMe. Hyperscale innovation: Flexible data placement mode (fdp). https://nvmexpress.org/wp-content/uploads/Hyperscale-Innovation-Flexible-Data-Placement-Mode-FDP.pdf.

[69] NVMe. Nvm command set specification. https://nvmexpress.org/wp-content/uploads/NVM-Express-NVM-Command-Set-Specification-1.0c-2022.10.03-Ratified.pdf.

[70] NVMe. 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.

[71] NVMe. Nvm express® zoned namespace command set specification. https://nvmexpress.org/wp-content/uploads/NVMe-Zoned-Namespace-Command-Set-Specification-1.1a-2021.07.26-Ratified.pdf, 2021.

[72] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. The lru-k page replacement algorithm for database disk buffering. *Acm Sigmod Record*, 22(2):297–306, 1993.

[73] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. Frozenhot cache: Rethinking cache management for modern hardware. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 557–573, 2023.

[74] Samsung. Samsung pm1743. https://semiconductor.samsung.com/ssd/enterprise-ssd/pm1743/, 2023.

[75] Samsung. Samsung pm1743a. https://semiconductor.samsung.com/news-events/news/samsung-introduces-its-first-zns-ssd-with-maximized-user-capacity-and-enhanced-lifespan/, 2023.

[76] Xuanhua Shi, Ming Li, Wei Liu, Hai Jin, Chen Yu, and Yong Chen. Ssdup: a traffic-aware ssd burst buffer for hpc systems. In *Proceedings of the international conference on supercomputing*, pages 1–10, 2017.

[77] 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.

[78] SNIA. Benefits of zns in datacenter storage systems. https://www.flashmemorysummit.com/Proceedings2019/08-06-Tuesday/20190806_ARCH-102-1_Chung.pdf.

[79] SNIA. Fiu traces. http://iotta.snia.org/traces/block-io/390.

[80] SNIA. Msr cambridge traces. http://iotta.snia.org/traces/block-io/388.

[81] SNIA. Zoned namespaces use cases, standard and linux ecosystem. https://www.snia.org/sites/default/files/SDCEMEA/2020/3%20-%20Javier%20Gonzalez%20Zoned%20namespacese.PDF, 2020.

[82] Zone Storage. Write ordering control. https://zonedstorage.io/docs/linux/sched.

[83] Hui Sun, Xiao Qin, Fei Wu, and Changsheng Xie. Measuring and analyzing write amplification characteristics of solid state disks. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 212–221. IEEE, 2013.

[84] Jinghan Sun, Shaobo Li, Yunxin Sun, Chao Sun, Dejan Vucinic, and Jian Huang. Leaftl: A learning-based flash translation layer for solid-state drives. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 442–456, 2023.

[85] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. Scalability in the xfs file system. In *USENIX Annual Technical Conference*, volume 15, 1996.

[86] Yufei Tao and Dimitris Papadias. Maintaining sliding window skylines on data streams. *IEEE Transactions on Knowledge and Data Engineering*, 18(3):377–391, 2006.

[87] 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*, pages 24–29, 2023.

[88] Rui Wang, Yongkun Li, Hong Xie, Yinlong Xu, and John CS Lui. {GraphWalker}: An {I/O-Efficient} and {Resource-Friendly} graph analytic system for fast and scalable random walks. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 559–571, 2020.

[89] Shucheng Wang, Qiang Cao, Ziyi Lu, Hong Jiang, Jie Yao, and Yuanyuan Dong. {StRAID}: Stripe-threaded architecture for parity-based {RAIDs} with ultra-fast {SSDs}. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 915–932, 2022.

[90] Stephen B Wicker and Vijay K Bhargava. *Reed-Solomon codes and their applications*. John Wiley & Sons, 1999.

[91] Linux wiki. Perf wiki. https://perf.wiki.kernel.org/.

[92] Wikipedia. Device mapper. https://en.wikipedia.org/wiki/Device_mapper.

[93] Wikipedia. Priority queue. https://en.wikipedia.org/wiki/Priority_queue#:~:text=Article%20Talk,before%20elements%20with%20low%20priority.

[94] Theodore M Wong and John Wilkes. My cache or yours?: Making storage more exclusive. In *USENIX Annual Technical Conference, General Track*, pages 161–175, 2002.

[95] Fei Wu, Jiaona Zhou, Shunzhuo Wang, Yajuan Du, Chengmo Yang, and Changsheng Xie. Fastgc: Accelerate garbage collection via an efficient copyback-based data migration in ssds. In *Proceedings of the 55th Annual Design Automation Conference*, pages 1–6, 2018.

[96] Suzhen Wu, Weidong Zhu, Yingxin Han, Hong Jiang, Bo Mao, Zhijie Huang, and Liang Chen. Gc-steering: Gc-aware request steering and parallel reconstruction optimizations for ssd-based raids. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4587–4600, 2020.

[97] 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.

[98] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. Fifo queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 130–149, 2023.

[99] Shushu Yi, Xiurui Pan, Qiao Li, Qiang Li, Chenxi Wang, Bo Mao, Myoungsoo Jung, and Jie Zhang. ScalaAFA: Constructing User-Space All-Flash array engine with holistic designs. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, pages 141–156, Santa Clara, CA, July 2024. USENIX Association.

[100] Shushu Yi, Yanning Yang, Yunxiao Tang, Zixuan Zhou, Junzhe Li, Chen Yue, Myoungsoo Jung, and Jie Zhang. Scalaraid: Optimizing linux software raid system for next-generation storage. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, pages 119–125, 2022.

[101] Jie Zhang, Mustafa Shihab, and Myoungsoo Jung. Power, energy, and thermal considerations in {SSD-Based}{I/O} acceleration. In *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, 2014.

[102] Yu Zhang, Ping Huang, Ke Zhou, Hua Wang, Jianying Hu, Yongguang Ji, and Bin Cheng. {OSCA}: An {Online-Model} based cache allocation scheme in cloud block storage systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 785–798, 2020.