ZARC: Re-inventing Zone Allocation for LSMs on ZNS SSD

Farhan S. Chowdhury
Bangladesh University of Engineering and Technology
Dhaka, Bangladesh
farhan.shahriar.2015@gmail.com

Shadman Saqib Eusuf
University of Massachusetts Boston
Boston, Massachusetts, USA
S.Eusuf001@umb.edu

Subhadeep Sarkar
Brandeis University
Waltham, Massachusetts, USA
subhadeep@brandeis.edu

Tarikul Islam Papon
University of Massachusetts Boston
Boston, Massachusetts, USA
t.papon@umb.edu

Abstract

Log-Structured Merge (LSM)-based storage engines offer high write throughput by storing the data on storage as *immutable* files that are opportunistically reorganized through *compactions*. Immutability of the files and their frequent garbage collection (GC), however, leads to a major performance bottleneck: high write amplification. This is further exacerbated by the device-level GC when LSM engines run on top of ZNS SSDs. While prior research has shown that compaction-aware file placement at the device level can reduce the overall write amplification, we point out that the proposed solutions are tuned exclusively for specific compaction strategies, SSD configurations, and workloads.

In this early-stage work, we highlight that for LSM engines on ZNS SSDs, performance is governed by a critical four-way trade-off between the compaction file picking policy (LSM-side) and the file placement, GC, and new zone allocation policies (ZNS-side). To this end, we introduce ZARC, a novel framework for **Z**one **A**llocation & **R**eclamation after **C**ompaction that offers a continuum of *hybrid file placement policies* to bring the overall write amplification closer to its theoretical lower bound and allows the user to navigate this vast design space for a given workload and configuration. ZARC's hybrid file placement policies optimize performance for each LSM-level independently, leading to level-specific file placement, garbage collection, and zone allocation policies. Preliminary results from experimentation on ZenFS, a RocksDB file system for ZNS SSDs, show that ZARC's hybrid file placement policies reduce data movement by up to 86% and ZARC-GC improves ZenFS garbage collection performance by (i) reclaiming sufficient free space and (ii) maintaining a small reserve of empty zones to prevent GC-induced stalls, and (iii) reducing GC-induced data movement by 8.6%.

CCS Concepts

• **Information systems** → **Database management system engines**; **Storage management**; *Flash memory*.

Keywords

LSM-tree, ZNS SSD, Zone Allocation, Garbage Collection, ZenFS

ACM Reference Format:
Farhan S. Chowdhury, Shadman Saqib Eusuf, Subhadeep Sarkar, and Tarikul Islam Papon. 2026. ZARC: Re-inventing Zone Allocation for LSMs on ZNS SSD. In *22nd International Workshop on Data Management on New Hardware (DaMoN '26), May 31–June 05, 2026, Bengaluru, India.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3789237.3809132

DaMoN '26, Bengaluru, India

ACM ISBN 979-8-4007-2455-8/26/05
https://doi.org/10.1145/3789237.3809132

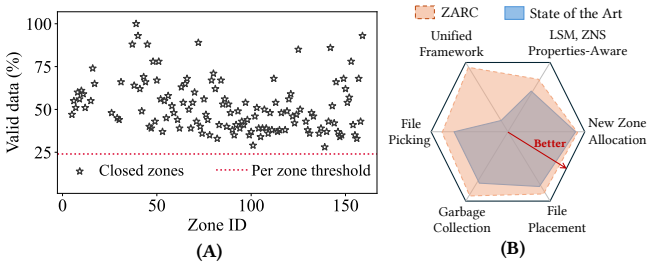

Figure 1: (A) GC is not performed because no zone satisfies the slope-based heuristic. (B) Limitations of the SoA. ZARC aims to strike a balance while providing a unified framework.

1 Introduction

LSM-based Data Stores. Log-structured merge (LSM) trees are widely used in modern NoSQL storage engines due to their superior ingestion performance, competitive query performance, and compact data storage [20, 21, 24]. LSM-trees store data as *immutable files* organized across multiple sorted runs on storage and opportunistically reorganize the data layout through *compactions* to ensure fast queries. While immutability boosts ingestion throughput, it leads to high write amplification (WA) [6, 7]. This is because a compaction invalidates all participating files and creates new ones, reducing space amplification (SA) and improving query performance at the cost of additional WA [1, 25]. Compactions thus dictate data movement and, thereby, the *file lifetime* in LSM-based systems.

The Emergence of Zoned Namespace SSD. Zoned Namespace (ZNS) SSDs [2, 9] divide the logical address space into fixed-size zones that allow *only sequential writes* and must be erased at zone granularity using the `Zone RESET` command. In contrast to conventional SSDs, ZNS SSDs transfer the responsibilities of file placement, empty zone selection, and garbage collection (GC) from the flash controller to the host application [2]. This allows for greater control over data placement and performance, since the host-level data organization directly impacts GC overhead and overall WA.

LSMs on ZNS SSD. The properties of LSM-trees and ZNS SSDs align naturally: both write immutable files sequentially and realize updates (and deletes) logically by accumulating invalidated data that must be periodically reclaimed through compaction or GC. For LSM engines on ZNS SSDs, this creates a critical four-way trade-off between the *compaction file picking policy* (LSM-side) and the *file placement*, *GC*, and *new zone allocation* policies (ZNS-side). The *file picking policy* determines which files participate in a compaction, dictating the file lifetimes [22, 25]. The *file placement policy* decides the mapping of files to zones [15, 19, 26]. If files with significantly different lifetimes share a zone, valid files must be relocated before the zone can be reset, incurring additional writes and increasing WA. The *GC policy* [27, 28] determines when to reclaim, which zones to target, and when to stop reclamation. The *new zone allocation policy* [17, 18] determines which empty zone to select for new

Table 1: Comparison of file placement, GC, and new zone allocation policies for LSM engines on ZNS SSDs. ✓ = supported, ~ = partial/limited, × = not addressed. *Configuration awareness* includes LSM size ratio, file size, and zone size.

	System	Partitioning	Lifetime Estimation/ Operating Principle	Key-range Aware	Compaction Aware	New Compaction Policy	Configuration Awareness	GC Design	Wear Leveling	Unified Framework
File Placement	LIZA [5]	Vertical	Level-based	×	×	×	×	×	×	×
	CAZA [15]	Vertical	Cross-level overlap	✓	✓	×	×	✓	×	×
	ZoneKV [19]	Horizontal	Same level only	×	×	×	×	×	×	×
	OAZA [26]	Horizontal	Compaction rank	✓	✓	×	~	✓	×	×
Compaction	LL-Compaction [13]	Vertical	Round-robin order	✓	✓	✓	×	×	×	×
	Prophet [16]	Mixed	Compaction rank + level	✓	✓	~	✓	×	×	×
	CAZA+ [3]	Vertical	Zone membership	✓	✓	✓	~	×	×	×
Zone Mgmt.	SplitZNS [12]	–	Enabling smaller zones	×	×	×	✓	×	✓	×
	Brick-ZNS [27]	–	Address remapping	×	×	×	✓	×	✓	×
	Z-GC [28]	–	Address remapping	×	×	×	✓	✓	✓	×
	WA-Zone [18]	–	Level-based wear inference	×	×	×	×	✓	✓	×
	GAP [17]	–	Address remapping	×	×	×	~	×	✓	×
	ZARC (proposed)	**Hybrid**	**Multi-policy**	✓	✓	✓ [planned]	✓	✓	✓	✓

writes, affecting wear distribution and future GC efficiency. Thus, these four policies are closely interdependent. To support LSM-based key-value stores on ZNS devices, ZenFS [5] was developed to manage zone allocation and reclamation for RocksDB [11].

Problem 1: Inefficient GC in ZenFS. Most systems hide the GC strategy behind a black box [19, 26], while others, such as ZenFS [5] and CAZA [15], suffer from several major limitations. GC in ZenFS and CAZA is triggered based on a heuristic-based *threshold* and *slope*: the GC module computes a *maximum valid data ratio* using the slope, and only zones below this ratio are selected as victims. If no such zone exists, GC fails to reclaim space even when reclaimable zones exist (Fig. 1A), potentially causing a stall. ZenFS also does not reserve zones and, thus, can crash despite having reclaimable space because there is nowhere to relocate valid data. Other limitations include the overhead of too many open zones and contention between GC and file placement, both leading to latency spikes and higher data movement (§3). Brick-ZNS [27], SplitZNS [12], and Z-GC [28] modify the ZNS standard to enable better GC, making them incompatible with existing ZNS SSDs. FAR [4] reduces zone-reset operations, but does not improve data movement. These highlight the need for more robust GC mechanisms.

Problem 2: Inefficient File Placement Policies. The file placement policy determines *which block of data goes to which zone*. Existing solutions focus on predicting file lifetimes and grouping files with similar lifetimes within the same zone. For example, LIZA [5] and CAZA [15] use vertical partitioning, placing files from different LSM levels in the same zone. These policies do not work well in practice because files from different levels have drastically different lifetimes, which depend on the *file picking policy* [25], making lifetime equalization extremely difficult. OAZA [26] and ZoneKV [19] perform horizontal partitioning, where files belonging to the same level are placed into zones together, avoiding level mixing. However, ZoneKV ignores lifetime differences within the same level, and OAZA's compaction-rank-based placement becomes obsolete after each compaction as rankings shift. CAZA+ [3], LL-compaction [13], and Prophet [16] either modify the file picking policy or make extensive changes to RocksDB. CAZA+ and LL-compaction abandon LSM efficiency to improve file placement, often increasing WA and space amplification. Prophet predicts lifetime using compaction ordering, but runs into the same problem as OAZA, i.e., rankings change after each compaction. Table 1 summarizes the limitations of existing approaches across various dimensions.

Problem 3: Lack of Unified Framework. While several file placement, GC, and new zone allocation policies have been proposed for LSM engines on ZNS SSDs, these efforts typically evaluate individual techniques in isolation. Thus, it remains unclear how the four-way trade-off between file picking, file placement, GC, and new zone allocation policies interacts with one another, or how their effectiveness varies across different configurations and workload distributions. For instance, changing the zone size, file size, or LSM size ratio can drastically affect which file placement and GC policy performs best. Similarly, some policies perform better under uniform workloads while others excel under skewed workloads. There is no unified framework for systematic comparison across this broad design space, spanning LSM parameters, ZNS-specific settings, and diverse workload configurations, as shown in Fig. 1B, motivating the need for a unified framework.

Our Contribution. In this vision paper, we highlight the key factors dictating the performance of LSM engines on ZNS SSDs. We present ZARC (Zone Allocation & Reclamation after Compaction), a unified framework to systematically navigate the four-way trade-off between file picking, file placement, GC, and new zone allocation policies. ZARC contains three modules illustrated in Fig. 2: (A) a *file placement module*, (B) a *garbage collection module*, and (C) a *new zone allocation module*. Each module exposes several policies and tunable parameters, and together they account for both LSM-specific properties (size ratio, file size) and ZNS-specific settings (zone size). For a given workload, LSM/ZNS configuration and file picking policy, ZARC allows the user to navigate this vast design space and identify the optimal combination of file placement, GC, and new zone allocation policies (Fig. 1B). As a first step, we (a) implement all four state-of-the-art file placement baselines (LIZA, CAZA, OAZA, ZoneKV) in ZARC, (b) propose multiple new placement policies, including a proximity-aware policy, (c) propose a continuum of hybrid policies that allow different LSM levels to adopt different placement strategies, (d) redesign the ZenFS GC module to ensure efficient space reclamation and prevent GC-induced stalls, and (e) provide efficient new zone allocation policies for improved wear leveling. In the long run, we envision ZARC as a platform to also co-design and evaluate novel file picking policies that better align with placement and GC strategies. Initial experimental evaluation shows that ZARC's hybrid placement policies reduce WA, and ZARC-GC outperforms the default ZenFS GC under tight space constraints.

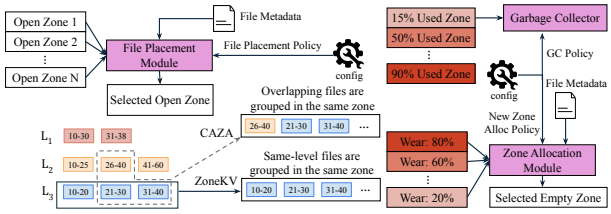

Figure 2: ZARC's three key modules: (i) File Placement Module, (ii) Garbage Collector, and (iii) New Zone Allocation Module.

2 ZARC: Design & Implementation

We now present the design details of the three modules of ZARC.

2.1 File Placement Module

The lifetime of a file is determined by both the compaction policy and the workload [16]. Ideally, all files placed within a zone would have the same lifetime and become invalid at the same time, allowing the zone to be reset without relocating valid data. ZARC's flexible file placement module provides a wide array of policies to choose from, allowing the user to optimize performance based on the LSM engine's file picking policy, size ratio, file size to zone size ratio, and workload distribution. ZARC also allows different file placement policies for different LSM levels, enabling the creation of a large number of hybrid policy combinations. Lower levels have fewer but much longer-lived files than upper levels, making level-specific placement beneficial. ZARC also supports a fallback policy per level to maintain baseline performance if the intended policy fails. We implement four state-of-the-art baselines and one naïve policy on ZARC: (A) *LIZA* [5]: assigns a lifetime label to each file based on its level and places it alongside files with the same or higher lifetime label, (B) *CAZA* [15]: places overlapping files from adjacent levels in the same zone, (C) *ZoneKV* [19]: stores files from the same level in the same zone in the order of creation, (D) *OAZA* [26]: computes a ranking of files based on their overlap with those in the next level and places similarly-ranked files in the same zone, and (E) a *naïve* policy: greedily fills zones with same-level files in arrival order.

New File Placement Policies. We propose and implement the following new file placement policies in ZARC: (A) **Nearest:** places a new file close to files with adjacent key ranges in the same level; it follows a weighted system where zones with more adjacent files are given more priority than zones with files that are farther away, (B) **Hybrid:** allows different file placement policies to be applied in different LSM levels, enabling a large combinatorial space of level-specific strategies, (C) **Delete-aware:** places files with more tombstones into separate zones, inspired by Lethe [23], and (D) **Ranking-based:** computes a file ranking based on overlap with adjacent levels and/or file sizes compensated for tombstones, and places similarly-ranked files in the same zone.

2.2 Garbage Collector

ZARC's novel garbage collector is designed to always select the zone with the least amount of valid data and continue reclamation until a user-defined target is reached, ensuring forward progress regardless of device saturation. The garbage collector wakes up every t seconds ($t = 10$ by default) and checks whether the free space percentage F falls below a *GC start threshold* T_s. If so, ZARC-GC reclaims zones one at a time until T_e% of the SSD is free ($T_e > T_s$), where T_e is the

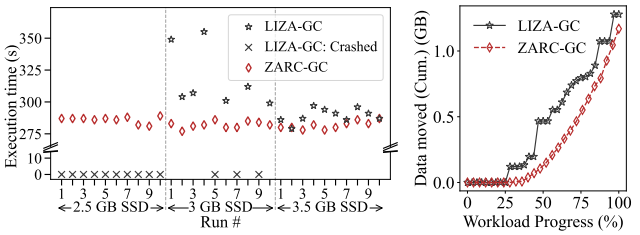

Figure 3: (A) ZARC-GC is efficient in tight space constraints. (B) ZARC-GC consistently moves less data compared to LIZA-GC.

GC stop threshold. GC victim zones are selected in ascending order of the amount of valid data. For each victim zone, valid files are migrated to a target zone chosen by ZARC's file placement policy. To guarantee forward progress during relocation, ZARC-GC maintains a small reserve of empty zones (5% by default). Once all valid files are migrated, the victim zone is reset and returned to the empty pool. A locking mechanism prevents concurrent writes during GC, avoiding latency spikes due to resource contention. ZARC exposes key GC parameters (e.g., thresholds, #reserve zones, etc.) to the user for fine-grained control. An optional GC mode (adapted from WA-Zone [18]) selectively migrates long-lived files from low-wear zones to high-wear zones, trading modest WA for device longevity.

2.3 New Zone Allocation Module

When the file placement module cannot find a suitable open zone to place a new file, it invokes the new zone allocation module to select an empty zone. This module operates on metadata of all currently empty zones and returns one based on a selection policy. Most policies select an empty zone based on its current wear level to maximize device longevity. ZARC provides the following zone allocation policies: (i) **first-available empty** zone selection (ZenFS default), (ii) **round-robin**, (iii) **random**, and (iv) **lifetime-based** selection, adapted from WA-Zone [18]. The lifetime-based policy utilizes the fact that files in deeper LSM-levels typically have longer lifetimes and can remain in a zone for extended periods without triggering frequent zone resets. It attempts to place files with shorter lifetimes in low-wear zones and files with longer lifetimes in high-wear zones. The default ZenFS policy can be suboptimal because it iterates through zones in ascending order of their indices and chooses the first empty zone, causing zones with lower indices to suffer disproportionately higher wear.

3 Experimental Evaluation

We present an initial evaluation of ZARC in two parts: (i) ZARC-GC vs. the default ZenFS GC (aka LIZA-GC), and (ii) hybrid file placement policies compared against the baselines, showcasing the potential of ZARC as a unified framework for systematic evaluation. All experiments are conducted on the Chameleon testbed [14].

3.1 Garbage Collection

Setup. We run a 1 GB uniform insertion workload with both LIZA-GC and ZARC-GC on three different SSD sizes: 2.5 GB, 3 GB, and 3.5 GB. The SSDs are simulated in ConfZNS++ [8] with a zone size of 16MB. The LSM-tree's size ratio is set to 2 and file size is 1MB. We run the workload 10 times for each GC policy and SSD size.

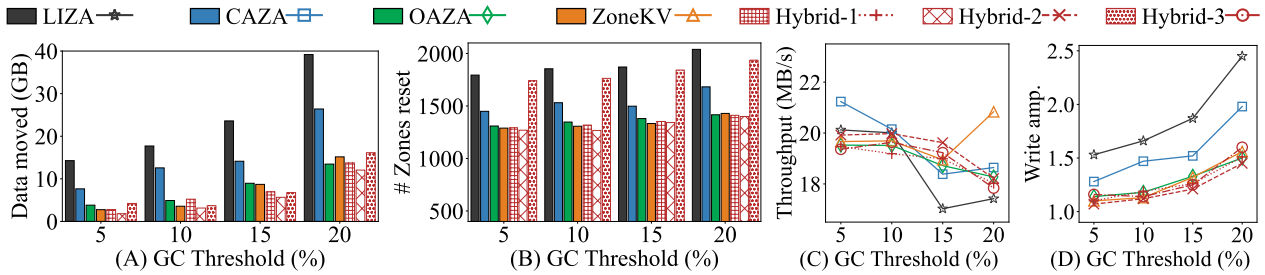

Figure 4: Performance of Hybrid policies with LSM size ratio 10. (A) Hybrid-2 greatly reduces data movement. (B) Hybrid-2 slightly reduces zone reset counts. (C) Hybrid-2 shows comparable throughput. (D) Hybrid-2 reduces WA.

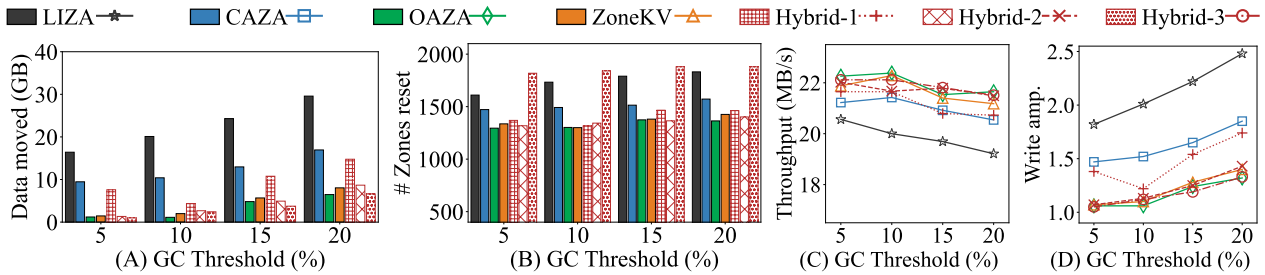

Figure 5: Performance of Hybrid policies with LSM size ratio 2. (A) Hybrid-3 significantly reduces data movement. (B) Hybrid-3 increases zone reset count. (C) Hybrid-3 shows slightly higher throughput. (D) Hybrid-3 reduces WA.

ZARC-GC Works Well under Space Constraints. Fig. 3A shows that ZARC-GC is much more successful than LIZA-GC at lower SSD sizes, when space constraint is tight. In all 10 runs on the 2.5 GB SSD, LIZA-GC crashes before finishing the workload, whereas ZARC-GC completes it successfully on all occasions. In the 3 GB SSD, LIZA-GC finishes only 7 out of 10 times, whereas ZARC-GC finishes all 10, with 11% better average runtime (282s vs. 318s). At 3.5 GB, both succeed with comparable runtimes (282s vs. 289s). In essence, LIZA-GC only performs well under relaxed space constraints. We also observe that ZARC-GC consistently keeps a lower number of zones open compared to LIZA-GC (on average by 9%) and its reserved zone policy guarantees forward progress. Further, on average, the percentage of valid data in closed zone for LIZA-GC and ZARC-GC is 28.42% and 26.33%, respectively. The percentage of valid data in reset victims is also significantly lower for ZARC-GC vs. LIZA-GC (12.63% vs. 21.19%), indicating better victim selection; we omit the graphs for brevity. Overall, as shown in Fig. 3B, ZARC-GC reduces GC-induced data movement by 8.6%.

3.2 File Placement Policy

Setup. We conduct experiments on a simulated 32 GB SSD with 128 MB zones in ConfZNS++ [8], running a 27 GB uniform insertion workload with 16-byte keys, 4096-byte values, and 7 million entries using db_bench [10]. The file size is set to 32 MB; we experiment with size ratios 2 and 10. We evaluate four baselines: ZenFS default (LIZA), CAZA, OAZA, and ZoneKV, and three hybrid policies:
(A) **Hybrid-1:** Naïve on levels 0-1 and CAZA on other levels,
(B) **Hybrid-2:** Naïve on levels 0-1 and *Nearest* on other levels, and
(C) **Hybrid-3:** CAZA on levels 0-1 and *Nearest* on other levels.

Hybrid Policies Reduce Data Movement. Fig. 4 presents results with size ratio 10 as we vary the GC threshold. We observe that Hybrid-2 achieves the lowest data movement, performing 79%, 66%, 34%, and 25% better than LIZA, CAZA, OAZA, and ZoneKV (Fig. 4A) and 30%, 14%, 3.2%, and 1.5% better in zone reset counts (Fig. 4B), while achieving comparable throughput (up to 4.1% better as shown in Fig. 4C). Hybrid-3 shows the worst zone reset count among hybrid policies, as it uses CAZA on upper levels, which groups files with varying lifetimes in the same zone, increasing zone resets.

Best Policy Varies with System Configuration. Fig. 5 presents results with size ratio 2. We observe that Hybrid-3 performs best in data movement (Fig. 5A), outperforming LIZA, CAZA, OAZA, and ZoneKV by 86%, 75%, 5.4%, and 16%, with comparable or better throughput (up to 9% better as shown in Fig. 5C). However, it performs 7%, 23%, 39%, and 36% worse in zone reset counts (Fig. 5B) than LIZA, CAZA, OAZA, and ZoneKV, respectively, signaling a tradeoff between data movement and SSD wear. This is because Hybrid-3's proximity-aware placement opens more zones to maintain key-range locality, increasing zone resets. These results demonstrate that the optimal policy varies across LSM and ZNS settings, motivating ZARC's systematic exploration of the hybrid policy design space across diverse configurations and workloads.

4 Next Steps

As our immediate next step, we plan to conduct rigorous evaluations across diverse workloads, LSM configurations (e.g., size ratio and compaction policies), and realistic SSD settings with larger capacities and zone sizes. We also intend to explore richer hybrid designs and eventually co-design compaction policies that better align with placement strategies. Ultimately, we plan to investigate ML-based approaches for SST lifetime estimation.

Acknowledgments. We thank the reviewers for their constructive feedback. Results presented in this paper were obtained using the Chameleon testbed supported by the National Science Foundation.

References

[1] Manos Athanassoulis and Stratos Idreos. 2016. Design Tradeoffs of Data Access Methods. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Tutorial.*

[2] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *Proceedings of the USENIX Annual Technical Conference (ATC).* 689–703. https://www.usenix.org/conference/atc21/presentation/bjorling

[3] Sungjin Byeon, Joseph Ro, Jun Young Han, Jeong-Uk Kang, and Youngjae Kim. 2024. Ensuring compaction and zone cleaning efficiency through same-zone compaction in zns key-value store. In *Proc. 38th Int. Conf. Massive Storage Syst. Technol.(MSST).*

[4] Sungjin Byeon, Joseph Ro, Safdar Jamil, Jeong-Uk Kang, and Youngjae Kim. 2023. A Free-Space Adaptive Runtime Zone-Reset Algorithm for Enhanced ZNS Efficiency. In *Proceedings of the USENIX Conference on Hot Topics in Storage and File Systems (HotStorage).* 109–115. https://doi.org/10.1145/3599691.3603410

[5] Western Digital Corporation. 2025. ZenFS: RocksDB Storage Backend for ZNS SSDs and SMR HDDs. *https://github.com/westerndigitalcorporation/zenfs* (2025).

[6] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 79–94. https://doi.org/10.1145/3035918.3064054

[7] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 505–520. http://doi.acm.org/10.1145/3183713.3196927

[8] Krijn Doekemeijer, Dennis Maisenbacher, Zebin Ren, Nick Tehrany, Matias Bjørling, and Animesh Trivedi. 2024. Exploring I/O Management Performance in ZNS with ConfZNS++. In *Proceedings of the ACM International Conference on Systems and Storage (SYSTOR).* 162–177. https://doi.org/10.1145/3688351.3689160

[9] Krijn Doekemeijer, Nick Tehrany, Balakrishnan Chandrasekaran, Matias Bjørling, and Animesh Trivedi. 2023. Performance Characterization of NVMe Flash Devices with Zoned Namespaces (ZNS). In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER).* 118–131. https://doi.org/10.1109/CLUSTER52292.2023.00018

[10] Facebook. 2024. db_bench. *https://github.com/facebook/rocksdb/wiki/Benchmarking-tools* (2024).

[11] Facebook. 2024. RocksDB. *https://github.com/facebook/rocksdb* (2024).

[12] Dong Huang, Dan Feng, Qiankun Liu, Bo Ding, Wei Zhao, Xueliang Wei, and Wei Tong. 2023. SplitZNS: Towards an Efficient LSM-Tree on Zoned Namespace SSDs. *ACM Trans. Archit. Code Optim.* 20, 3 (2023), 45:1—-45:26. https://doi.org/10.1145/3608476

[13] Jeeyoon Jung and Dongkun Shin. 2022. Lifetime-leveling LSM-tree compaction for ZNS SSD. In *HotStorage '22: 14th ACM Workshop on Hot Topics in Storage and File Systems, Virtual Event, June 27 - 28, 2022.* 100–105. https://doi.org/10.1145/3538643.3539741

[14] Kate Keahey, Jason Anderson, Zhuo Zhen, Pierre Riteau, Paul Ruth, Dan Stanzione, Mert Cevik, Jacob Colleran, Haryadi S. Gunawi, Cody Hammock, Joe Mambretti, Alexander Barnes, François Halbach, Alex Rocha, and Joe Stubbs. 2020. Lessons Learned from the Chameleon Testbed. In *Proceedings of the 2020 USENIX Annual Technical Conference (USENIX ATC '20).* USENIX Association.

[15] Hee-Rock Lee, Chang-Gyu Lee, Seungjin Lee, and Youngjae Kim. 2022. Compaction-aware zone allocation for LSM based key-value store on ZNS SSDs. In *HotStorage '22: 14th ACM Workshop on Hot Topics in Storage and File Systems, Virtual Event, June 27 - 28, 2022.* 93–99. https://doi.org/10.1145/3538643.3539743

[16] Gaoji Liu, Chongzhuo Yang, Qiaolin Yu, Chang Guo, Wen Xia, and Zhichao Cao. 2024. Prophet: Optimizing lsm-based key-value store on zns ssds with file lifetime prediction and compaction compensation. In *38th Intl. Conf. on Massive Storage Systems and Technology.*

[17] Renping Liu, Cheng Ran, Han Hu, Anping Xiong, Peng Chen, and Linbo Long. 2024. GAP: A Global Wear-Aware Block Pool for Enhancing Lifetime of ZNS SSDs. In *Proceedings of the IEEE Non-Volatile Memory Systems and Applications Symposium (NVMSA).* 1–6. https://doi.org/10.1109/NVMSA63038.2024.10693672

[18] Linbo Long, Shuiyong He, Jingcheng Shen, Renping Liu, Zhenhua Tan, Congming Gao, Duo Liu, Kan Zhong, and Yi Jiang. 2024. WA-Zone: Wear-Aware Zone Management Optimization for LSM-Tree on ZNS SSDs. *ACM Transactions on Architecture and Code Optimization (TACO)* 21, 1 (2024), 16:1—-16:23. https://doi.org/10.1145/3637488

[19] Mingchen Lu, Peiquan Jin, Xiaoliang Wang, Yongping Luo, and Kuankuan Guo. 2023. ZoneKV: A Space-Efficient Key-Value Store for ZNS SSDs. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC).* 1–6. https://doi.org/10.1109/DAC56929.2023.10247926

[20] Chen Luo and Michael J Carey. 2018. LSM-based Storage Techniques: A Survey. *CoRR* abs/1812.0 (2018). https://arxiv.org/abs/1812.07527

[21] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The log-structured merge-tree (LSM-tree). *Acta Informatica* 33, 4 (1996), 351–385. http://dl.acm.org/citation.cfm?id=230823.230826

[22] Subhadeep Sarkar, Kaijie Chen, Zichen Zhu, and Manos Athanassoulis. 2022. Compactionary: A Dictionary for LSM Compactions. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 2429–2432.

[23] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, and Manos Athanassoulis. 2020. Lethe: A Tunable Delete-Aware LSM Engine. In *Proceedings of the ACM SIGMOD International Conference on Management of Data.* 893–908.

[24] Subhadeep Sarkar, Tarikul Islam Papon, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2023. Enabling Timely and Persistent Deletion in LSM-Engines. *ACM Transactions on Database Systems (TODS)* 48, 3 (2023), 8:1—-8:40. https://doi.org/10.1145/3599724

[25] Subhadeep Sarkar, Dimitris Staratzis, Zichen Zhu, and Manos Athanassoulis. 2021. Constructing and Analyzing the LSM Compaction Design Space. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2216–2229. http://vldb.org/pvldb/vol14/p2216-sarkar.pdf

[26] Jingcheng Shen, Lang Yang, Linbo Long, Renping Liu, Zhenhua Tan, Congming Gao, and Yi Jiang. 2024. Overlapping Aware Zone Allocation for LSM Tree-Based Store on ZNS SSDs. In *Proceedings of the Asia and South Pacific Design Automation Conference (ASPDAC).* 448–453. https://doi.org/10.1109/ASP-DAC58780.2024.10473944

[27] Zhenhua Tan, Linbo Long, Jingcheng Shen, Renping Liu, Congming Gao, Kan Zhong, and Yi Jiang. 2024. Optimizing Garbage Collection for ZNS SSDs via In-storage Data Migration and Address Remapping. *ACM Trans. Archit. Code Optim.* 21, 4 (2024), 77:1—-77:25. https://doi.org/10.1145/3689336

[28] Yuyan Zhang and Ping Xie. 2025. A Fine-Grained Garbage Collection Scheme for ZNS-SSD Storage Architecture. In *2025 5th International Conference on Electronic Information Engineering and Computer Science (EIECS).*