ColdMap: Compaction-Aware Cost-Benefit Zone Cleaning for ZNS-Based Key-Value Stores

Sungjin Byeon
Sogang University
Seoul, Republic of Korea
sg20180546@gmail.com

Kyungwook Min
Sogang University
Seoul, Republic of Korea
minsky@sogang.ac.kr

Jaewan Park
Sogang University
Seoul, Republic of Korea
brian7567@sogang.ac.kr

Sangyun Lee
Sogang University
Seoul, Republic of Korea
tkddbs9801@sogang.ac.kr

Hong-Yeon Kim
ETRI
Daejeon, Republic of Korea
kimhy@etri.re.kr

Junyoung Han
Samsung Electronics
Suwon, Republic of Korea
jy0.han@samsung.com

Joo-Young Hwang
Samsung Electronics
Suwon, Republic of Korea
jooyoung.hwang@samsung.com

Zhichao Cao
Arizona State University
Tempe, AZ, USA
zhichao.cao@asu.edu

Youngjae Kim
Sogang University
Seoul, Republic of Korea
youkim@sogang.ac.kr

Abstract

LSM-tree–based key-value stores are widely deployed on Zoned Namespace (ZNS) SSDs to exploit their sequential write semantics. However, despite extensive research on data placement and compaction in LSM–ZNS systems, the problem of victim zone selection during zone cleaning (ZC) has been largely overlooked. Existing designs either rely on greedy heuristics or implicitly assume that victim selection is trivial, which results in unnecessary valid-data copying and avoidable I/O blocking during ZC. This paper presents ColdMap, the first ZC framework that explicitly targets accurate victim zone selection in LSM–ZNS file systems. The key insight behind ColdMap is that zone coldness is not governed by history-based modification metrics, but by the future evolution of the LSM-tree driven by compaction. To capture this behavior, ColdMap (i) predicts zone coldness by proactively simulating future compactions and estimating file invalidation timing, and (ii) applies a cost–benefit ZC algorithm tailored to ZNS file systems. We implement ColdMap in RocksDB and ZenFS and evaluate it on real ZNS SSDs under diverse workloads. Experimental results show that ColdMap reduces ZC overhead by up to 49% compared to state-of-the-art victim selection schemes, resulting in an average 15.7% improvement in overall throughput.

CCS Concepts

• **Information systems → Storage management**; • **Operating systems → File systems management**;

*Equal contribution.
+Y. Kim is the corresponding author.

ICS '26, Belfast, United Kingdom

ACM ISBN 979-8-4007-2522-7/26/07
https://doi.org/10.1145/3797905.3807851

Keywords

File systems, Zone Namespace SSD, Garbage Collection

ACM Reference Format:
Sungjin Byeon, Kyungwook Min, Jaewan Park, Sangyun Lee, Hong-Yeon Kim, Junyoung Han, Joo-Young Hwang, Zhichao Cao, and Youngjae Kim. 2026. ColdMap: Compaction-Aware Cost-Benefit Zone Cleaning for ZNS-Based Key-Value Stores. In *2026 International Conference on Supercomputing (ICS '26), July 06–09, 2026, Belfast, United Kingdom.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3797905.3807851

1 Introduction

LSM-tree–based key–value stores naturally fit ZNS SSDs due to their log-structured, append-only design. ZNS SSDs partition storage into fixed-size zones and enforce sequential writes, shifting address mapping and space management to the host [14, 28, 29, 36, 50]. This execution model closely matches LSM-trees, where Write-Ahead Logs (WALs) and Sorted String Tables(SSTs) are written sequentially and data updates are realized through compaction. Motivated by this alignment, recent work has extensively explored efficient execution of LSM-tree–based key–value stores on ZNS SSDs [15, 16, 19, 33, 34, 40, 43, 45, 57].

Although LSM-trees naturally align with zone-level sequential writes, this compatibility does not eliminate all costs in ZNS environments. Because in-place updates are disallowed, updates are written out-of-place, leading to the gradual accumulation of invalid data within zones and eventually triggering ZC. ZC selects a victim zone, migrates the remaining valid data to other zones, and resets the zone. The resulting data movement incurs I/O blocking, increases latency, and exacerbates Write Amplification (WA). As a result, ZC overhead remains a fundamental performance bottleneck in ZNS-based log-structured systems.

Prior work has largely pursued two classes of approaches to reduce valid-data copying during ZC in ZNS environments. The first class relies on *lifetime-aware zone placement*, which colocates data with similar lifetimes in the same zone to minimize data migration during ZC. CAZA [15] and Prophet [43] use lifetime prediction to guide placement, while OAZA [57] optimizes zone allocation

based on SST key-range overlap. Although conceptually appealing, sustaining such ideal placement in practice is difficult. LSM-tree compaction continuously creates, merges, and deletes files, and prediction errors or workload shifts inevitably cause data with heterogeneous lifetimes to co-reside within the same zone (§6.2.1). As a result, valid-data copying during ZC persists even with lifetime-aware placement, as observed in prior evaluations.

The second class of work reduces ZC overhead by *structurally coupling zone cleaning with LSM-tree compaction*. Instead of treating ZC as an independent reclamation step, these approaches synchronize data invalidation within a zone through compaction to minimize or eliminate valid data at ZC time. HiZNS [45], for example, aligns SST sizes with zone sizes so that each zone contains a single SST, enabling zero-copy ZC when that SST is invalidated by compaction. LL-Compaction [40] and LifetimeKV [34] modify compaction logic to ensure that files colocated in the same zone are removed at similar times.

However, these approaches commonly suffer from the *compaction compensation* problem [43]. Enlarging SST sizes or expanding compaction scope to achieve zone-level invalidation increases compaction cost, resulting in higher write amplification and longer compaction latency. Consequently, ZC overhead is reduced only by shifting its cost to the compaction stage, leading to I/O blocking and overall performance degradation. In essence, these techniques trade ZC overhead for compaction overhead rather than eliminating it.

In addition, FAR [16] proposes a proactive ZC scheme that reduces ZC frequency by early-resetting fully invalidated zones. Rather than reducing valid-data copying, FAR focuses on suppressing ZC invocations themselves. However, this approach does not eliminate compaction overhead and therefore provides only limited relief from the fundamental cost of ZC (§6.2.2).

Revisiting Zone Cleaning: Victim Selection as a First-Class Problem. Despite extensive efforts, existing approaches either fail to eliminate valid-data copying or merely shift the cost to compaction. Consequently, performance degradation remains unavoidable once ZC is triggered. In this setting, *victim zone selection at ZC time* becomes a decisive factor in determining ZC overhead, yet it has received little attention as a first-class optimization problem.

Current ZenFS and RocksDB-on-ZNS systems primarily employ a simple greedy policy that selects the zone with the highest invalid ratio as the victim. Because this policy considers only already invalidated data, it ignores the expected remaining lifetime of valid data. As a result, zones containing *hot data* that will soon be invalidated by compaction may be selected, causing unnecessary valid-data copying. In short, the greedy policy fails to account for the *future lifetime* of valid data and prematurely migrates data that is about to become invalid.

Cost–benefit–based victim selection has been widely studied for garbage collection in log-structured file systems (LFS) [2, 39, 49, 56] and SSD firmware [18, 32, 35, 38, 51, 61] (§7). These techniques typically rely on history-based coldness metrics and assume relatively homogeneous data within a cleaning unit. LSM–ZNS systems violate both assumptions: (i) a single zone contains SSTs and WALs with heterogeneous lifetimes (§3.1.1), and (ii) file invalidation is governed primarily by LSM-tree compaction rather than access history (§3.1.2). Consequently, directly applying conventional cost–benefit schemes fails to capture future invalidation behavior in ZNS environments.

Contributions. Rather than directly adopting cost–benefit heuristics, we propose ColdMap, a novel approach that redefines zone coldness based on *LSM-tree compaction semantics* and performs victim zone selection accordingly. The key insight of ColdMap is that victim selection should consider not the past modification history of zone contents, but the *future invalidation time* induced by LSM-tree compaction. To this end, ColdMap simulates the compaction policy and scheduling rules of the LSM-tree to predict when each SST and WAL file will be invalidated, and directly incorporates this prediction into cost–benefit–based victim selection. Importantly, ColdMap does not modify the compaction algorithm or data placement policy, nor does it attempt to avoid ZC itself. Instead, it fundamentally improves decision accuracy at the moment when ZC is unavoidable. Unlike prior approaches that reduce ZC overhead by altering compaction or allocation, ColdMap addresses the victim selection problem itself.

Our contributions are summarized as follows.

- **We identify fundamental limitations of existing cost–benefit victim selection in ZNS environments (§3).** Conventional cost–benefit schemes are designed for MB-scale segments and assume relatively homogeneous data lifetimes within a cleaning unit. These assumptions break down in ZNS systems, where zones span tens to hundreds of MB and contain heterogeneous SST and WAL files. We systematically analyze how this structural mismatch significantly degrades cost–benefit effectiveness in ZNS environments.
- **We propose a compaction-aware zone coldness model for accurate victim selection (§4).** ColdMap demonstrates that zone coldness estimation based on *future invalidation time* is key to ZC efficiency. Specifically, it (i) simulates LSM-tree compaction to predict SST and WAL coldness, (ii) aggregates file-level predictions into zone-level coldness with size-aware weighting, and (iii) integrates the result into cost–benefit scoring. This approach effectively reduces unnecessary valid-data copying and minimizes ZC-induced I/O overhead.
- **We design a compaction-policy–agnostic zone cleaning mechanism that treats victim selection as a first-class concern (§5.4).** ColdMap is independent of specific compaction policies and does not modify the compaction algorithm itself. By modeling and exploiting compaction semantics, it consistently improves ZC efficiency under both Leveled and Universal Compaction.

To evaluate ColdMap, we implement CAZA [15], the state-of-the-art lifetime-aware placement scheme, in RocksDB (v7.4.4) with ZenFS (v2.1). We compare ColdMap against ZenFS's default greedy ZC policy, file-system–level cost–benefit schemes [56], and state-of-the-art firmware-level cost–benefit techniques. Across nine key–value workloads and three RDBMS-style workloads, ColdMap improves ZC efficiency by an average of 49% and achieves up to 15.7% throughput improvement in write-intensive scenarios. Under Universal Compaction, ColdMap reduces ZC overhead by up to 68.6% and improves throughput by 7%, demonstrating consistent benefits across compaction policies.

2 Background

LSM-tree–based key–value stores, such as LevelDB [27] and its multi-threaded successor RocksDB [54], are widely adopted as storage engines for high-throughput applications at scale [10, 26, 47]. RocksDB's log-structured, append-only write pattern naturally matches the sequential-write constraints of ZNS SSDs, motivating prior work on integrating RocksDB with zoned storage, most notably via ZenFS [13, 14, 19, 23, 25, 41]. Figure 1 illustrates the resulting software stack and runtime interaction between RocksDB, ZenFS, and the underlying ZNS device.

2.1 RocksDB Compaction Mechanisms

Leveled Compaction. RocksDB triggers *compaction* when a level exceeds its size threshold [15, 43]. By default, it employs the *leveled compaction* algorithm [55], which selects SST files for compaction based on level size constraints. The leveled compaction mechanism in RocksDB is illustrated on the right side in Figure 1. First, RocksDB selects the level i (L(i)) of compaction based on the *compaction score*, denoted as $L\_Score_{L(i)}$. The compaction score for L(i) is calculated as the ratio of the total size of the SST files in L(i) to its predefined size limit [15, 43].

For example, let the size limits for levels L(0), L(1), and L(2) be 128 MB, 192 MB, and 256 MB, respectively. Since L(1) has the highest compaction score, calculated as $\frac{231}{192} = 1.2$, it is chosen for compaction. Next, within the selected L(1), the SST file with the highest *Overlap Score*, represented as $O\_Score_{SST(j)}$ [6, 42, 57], is determined. The overlap score of an SST file SST_j in L(1) is defined as the ratio of its size to the total size of SST files in L(2) whose key ranges overlap with SST_j. In this example, the first SST file (highlighted in blue) in L(1) has the highest overlap score, $\frac{64}{44} = 1.45$. This file is referred to as *Pivot SST*, while the overlapping SST files in L(2) are denoted as *Unpivot SSTs*. Finally, the selected Pivot and Unpivot SST files are read, merged through a merge-sort operation, and written as a new SST file in L(2). Upon completion, the Pivot and Unpivot SST files are removed from the system.

Universal Compaction. In addition to *leveled compaction*, RocksDB provides *universal compaction* to better accommodate write-intensive workloads. Under this policy, SST files produced by flushes or compactions are managed as a set of sorted runs, which are selectively merged based on multiple criteria, such as space amplification and the size ratio between adjacent runs. Compared to leveled compaction, universal compaction reduces write amplification at the cost of increased read amplification and space amplification. A detailed description of the algorithmic behavior is provided in §5.4.

2.2 Zone Allocation and Cleaning in ZenFS

RocksDB manages both WAL and SST files in an append-only manner, avoiding in-place overwrites and reorganizing data through compaction. This write pattern naturally aligns with ZNS SSDs, which enforce sequential writes at zone granularity. ZenFS [23, 52] is a user-space file system designed to efficiently run RocksDB on ZNS SSDs. It directly exposes the ZNS device to the host and explicitly manages LSM-tree–generated SST and WAL files at the zone level. To this end, ZenFS provides two core mechanisms. *Zone Allocation (ZA)* assigns newly created SST and WAL files to appropriate zones and guarantees sequential writes within each zone, while

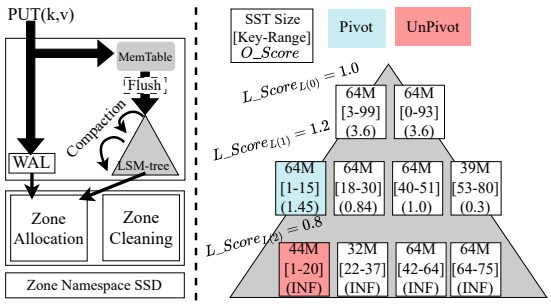


Figure 1: An illustrative example of RocksDB's leveled compaction on ZenFS, showing level selection (*L_Score*) and pivot/unpivot SST selection (*O_Score*).

Zone Cleaning reclaims zones that accumulate invalid data due to compaction and returns them to a reusable state.

When the available free space on a ZNS SSD falls below a predefined threshold, ZenFS triggers ZC. ZC selects a victim zone, migrates the remaining valid SST and WAL files in that zone to other zones, and resets the entire zone to reclaim free space. The reclaimed zone is then reused for subsequent SST creation. Through this process, ZenFS preserves the sequential-write constraint of ZNS SSDs while enabling space recycling.

However, ZC in ZenFS is performed in the foreground. While ZC is in progress, ZA is suspended and I/O requests are blocked. Moreover, inefficient victim zone selection can incur excessive valid-data copying, leading to additional I/O, write stalls, and overall throughput degradation [15, 33, 43, 45]. As a result, in ZenFS-based systems, *victim zone selection* becomes a first-order factor that directly determines ZC overhead and overall system performance.

2.3 Victim Zone Selection Policies: From Greedy to Cost–Benefit

When ZC is triggered, ZenFS employs a simple greedy victim selection policy that chooses the zone with the highest invalid ratio. This design minimizes implementation complexity and metadata overhead by considering only the amount of already invalid data. However, the greedy policy ignores the future lifetime of the remaining valid data, failing to distinguish whether they correspond to *hot data* that will soon be invalidated by compaction. As a result, zones containing hot data may be prematurely cleaned solely due to a high current invalid ratio, leading to unnecessary copying of valid data.

In log-structured file systems [2, 39, 49, 56] and SSD firmware [18, 32, 35, 38, 51, 61], cost–benefit–based victim selection has been widely studied to address this limitation by prioritizing cold segments and deferring the cleaning of hot ones. These techniques estimate coldness primarily from past modification histories. *However, such history-based metrics are fundamentally inadequate in ZenFS, where data lifetime is dominated by LSM-tree compaction rather than access patterns.* Data that appear cold based on past history may still be rewritten or deleted shortly thereafter due to compaction, making history-based coldness a poor predictor of future data survival.

3 Limitations of History-Based Victim Selection in LSM-ZNS Systems

This section revisits history-based victim selection in the context of LSM-tree–based ZNS systems and shows that coldness estimation fundamentally breaks down when compaction semantics are ignored. We first present two key observations that expose the limitations of applying classic cost–benefit–based approaches to ZenFS. We then distill these observations into concrete challenges and opportunities that motivate a compaction-aware design.

3.1 Limitations of History-Based Zone Coldness Estimation in LSM–ZNS

3.1.1 ***Zone Coldness Cannot Be Inferred from a Single File or History.*** CAZA [15, 40] and Prophet [43] represent the state of the art in coldness-aware SST placement, predicting the lifetime of SST files at placement time to batch them into the same zone based on an understanding of LSM-tree compaction behavior. Both approaches estimate coldness using static features available at placement time—such as the LSM-tree level, key-range overlap, and SST file size—and demonstrate high prediction accuracy under this assumption.

While these techniques focus on where SSTs should be placed, they do not directly address which zone should be reclaimed once zone cleaning is triggered. In practice, this decision is governed by the victim zone selection policy. They adopt a greedy victim zone selection policy, which we adapt to follow the classic cost–benefit segment cleaning (CBSC) algorithm [56], originally designed for log-structured file systems. CBSC assumes that blocks within a segment exhibit similar lifetimes and therefore represents segment coldness using the most recently modified block.

While this assumption holds in traditional LFS environments such as F2FS, it breaks down in ZenFS. In ZenFS, a segment corresponds to a ZNS zone whose size is orders of magnitude larger (96 MB–1 GB) than the 2 MB segments used in F2FS [3, 9]. As a result, a single zone typically aggregates multiple SST files with highly heterogeneous sizes and lifetimes, invalidating the use of a single, history-based metric to represent zone coldness (Figures 5 and 6).

This heterogeneity is not merely theoretical. Our analysis using YCSB workloads shows that even within a single zone, SST files exhibit widely varying sizes and update patterns, leading to significantly different effective lifetimes.

Table 1: Distribution of SST file sizes in RocksDB LSM-tree under YCSB workloads.

Workload	<42 MB	43–60 MB	61–64 MB	>64 MB
YCSB-zipfian	2,438 (12.4%)	2,425 (12.4%)	10,235 (52.1%)	4,529 (23.1%)
YCSB-uniform	1,950 (14.3%)	5,931 (43.6%)	3,150 (23.1%)	2,578 (19.0%)

Table 1 presents the size distribution of SST files within a zone. Under the Zipfian workload, approximately 52.1% ($\frac{10,235}{19,627}$) of SST files have sizes close to the maximum (64–65 MB), while the remaining 47.9% are distributed across a much wider range of sizes. Consequently, treating a zone as a homogeneous unit for coldness estimation systematically misrepresents its true temperature.

Observation 1. *In ZenFS, zone coldness cannot be reliably inferred from a single file or recent modification history due to heterogeneous SST sizes and lifetimes within a zone.*

3.1.2 ***History-Based Coldness Fails Under LSM Compaction Dynamics.*** Beyond heterogeneity, history-based coldness estimation fails for a more fundamental reason: in LSM-tree–based systems, file lifetimes are governed by compaction rather than chronological aging (e.g., creation and invalidation events). SST files are repeatedly created, rewritten, and deleted as they move across levels according to size constraints and compaction policies.

When history-based metrics such as modification time are directly applied to ZenFS, they implicitly assume that past updates correlate with future invalidation. However, compaction events can abruptly invalidate large numbers of SST files regardless of their recent modification history. Our trace-driven analysis around zone cleaning (ZC) events shows that zones predicted to be the coldest by CBSC are frequently the hottest in reality once compaction is triggered.

Observation 2. *In LSM–ZNS systems, coldness is dominated by compaction-induced level transitions, rendering history-based modification metrics unreliable for predicting future invalidation.*

3.2 Compaction-Induced Coldness Inversion

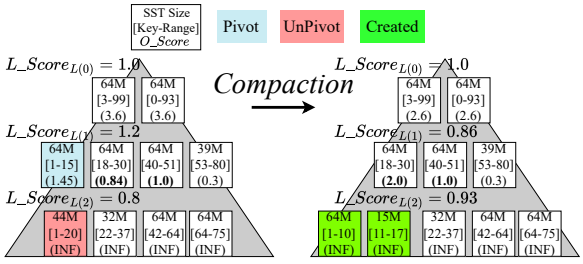


Figure 2: Compaction-induced changes in LSM-tree state and SST temperature.

Figure 2 illustrates how leveled compaction fundamentally alters coldness estimates in an LSM-tree. The left subfigure shows the SST layout before compaction, while the right shows the state after an L(1)-to-L(2) compaction. Before compaction, L(1) has the highest level score ($L\_score_{L(1)} = 1.2$), leading CAZA and Prophet to classify it as the hottest level. After compaction, however, the hottest level shifts to L(0) ($L\_score = 1.0$), while L(1) becomes the coldest with $L\_score_{L(1)} = 0.86$. This example illustrates that level temperature inferred from $L\_score_{L(i)}$ can be completely reversed by a single compaction event.

A similar effect is observed at the SST granularity. Consider the two L(1) SSTs with key ranges [18–30] and [40–51] before compaction. Initially, the SST [40–51] has the highest object score ($O\_score = 1.0$), making it the hottest SST in L(1). After the L(1)-to-L(2) compaction, pivot and unpivot SSTs are removed and new SSTs are created in L(2), altering the access characteristics of the remaining L(1) SSTs. As a result, the O_score of the SSTs [18–30] and [40–51] changes to 2.0 and 1.0, respectively, reversing their relative temperature ordering within the same level.

Compaction induces substantial reordering in coldness at the level granularity. Under the same workload and experimental setup

as Figure 5 in §6.2, the ordering of $L\_score_{L(i)}$ across levels changes with a probability of 64.2% after compaction, whereas the ordering of O_score_{SST} within a level changes only 2.2% of the time.

3.3 False-Positive Analysis of History-Based Coldness Prediction

To quantify the inaccuracy of history-based coldness prediction, we measure the false-positive rate of CBSC, defined as cases where a zone is predicted to be the coldest but is not cold in reality.

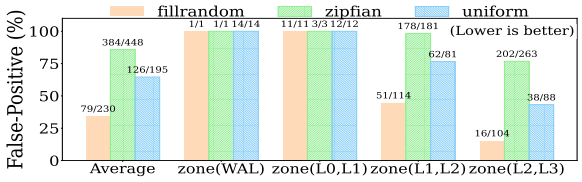


Figure 3: False-positive rates of history-based coldness prediction (CBSC) across workloads and zones (lower is better).

Figure 3 reports the results for three workloads—fillrandom, zipfian, and uniform—with different key distributions (Table 2). The experimental setup is identical to that assumed in the previous section. In the figure, the numerator of each bar indicates the number of false positives, while the denominator represents the total number of coldest-zone predictions.

Across all workloads, CBSC exhibits a high false-positive rate ranging from 34% to 86%. This indicates that a substantial fraction of zones selected as victims based on history-based coldness metrics are in fact not the coldest at the time of zone cleaning. The root cause of these false positives is that CBSC relies solely on past access history, failing to account for future compactions that significantly reshape data layout and dominate subsequent access behavior in LSM-tree–based ZNS systems.

Together, these results motivate a compaction-aware approach that can predict zone coldness beyond historical access patterns.

4 ColdMap Overview

The key idea of ColdMap is to predict zone coldness by modeling LSM-tree compaction behavior. To this end, ColdMap proactively simulates future compactions and uses the resulting coldness estimates to select the victim zone at zone-cleaning time. Figure 4a overviews the ColdMap workflow. When ZC is triggered, ColdMap (1) takes an LSM snapshot and constructs ColdMap(LSM), (2) simulates compaction to refine file-level coldness, and (3) derives ColdMap(ZNS) for zone-level victim selection, as detailed below.

(1) Initial Coldness Mapping: ColdMap(LSM) is initially constructed to record the coldness of all SST and WAL files generated by the LSM-tree, based on the current state of RocksDB (§5.1).

(2) Coldness Refinement via Compaction Simulation: Since the coldness of SST files changes as compaction progresses, future compactions are simulated to refine the coldness estimation. The results of this simulation are used to update ColdMap(LSM), reflecting expected changes in file coldness (§5.2).

(3) Zone-Level Conversion: Although ColdMap(LSM) tracks the coldness of individual SST files, it lacks zone-level information. By identifying the corresponding zone for each SST file, ColdMap(LSM) is converted into ColdMap(ZNS), enabling zone-level coldness inference (§5.3).

Using the resulting ColdMap(ZNS), we apply the final cost–benefit based zone-cleaning algorithm to select a victim zone. The overall workflow, triggered at ZC time, is illustrated in Figure 4b.

5 Design and Implementation

RocksDB supports multiple compaction policies. We present ColdMap under *leveled compaction* (§5.1–§5.3) and show how the same principles extend to *universal compaction* (§5.4).

5.1 Building ColdMap(LSM)

Under leveled compaction, the coldness of each SST file is computed using two factors and then min–max normalized to a value in the range [0.0, 1.0]. A value closer to 0.0 indicates a hotter file, whereas a value closer to 1.0 indicates a colder file.

Vertical Factor(VF): Each level's $L\_Score_{L(i)}$ is normalized to a value between 0 and 1 through min-max normalization, and this value is referred to as the Vertical Factor (VF). Therefore, each SST file SST_j is assigned the VF value VF_{SST_j} corresponding to the level it belongs to.

Horizontal Factor(HF): Each level has a VF value, and during compaction, the level with the highest VF value is selected to perform the compaction. From this level, the SST file that performs a merge-sort (Pivot SST) is chosen. To do this, the HF value of all SST files is calculated as follows. Let the number of Unpivot SST files that will be merge-sorted with the Pivot SST be $u-1$.

$$HF_{SST_k} = \frac{Size(Pivot_k)}{\sum_{i=0}^{u-1} Size(Unpivot_i)}$$

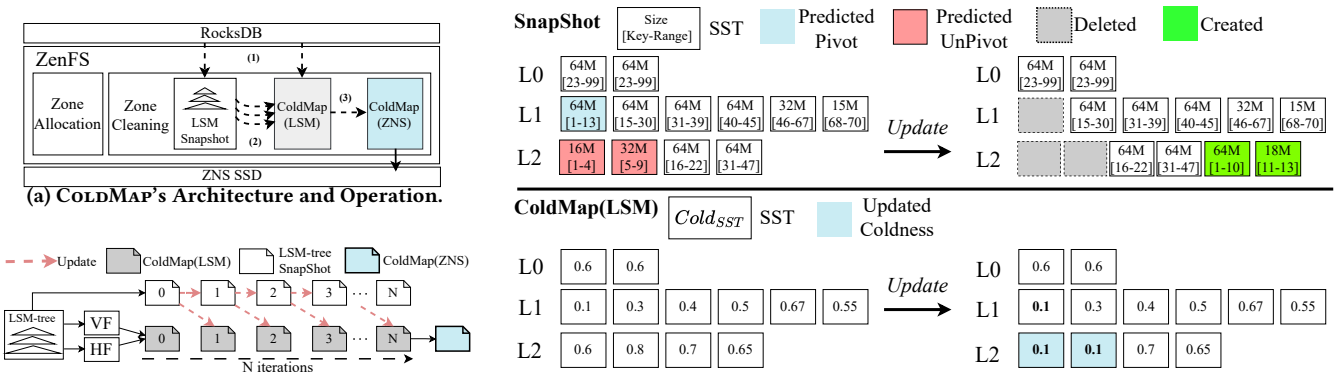
The SST file with the largest HF value is chosen as the Pivot SST. And then, we convert HF value a value between 0.0 and 1.0 by min-max normalization for fair reflection with VF. Finally, all SST file having an HF value (HF_{SST_j}) with range [0.0-1.0].

Note that the reason for selecting the largest HF value is that, in the current RocksDB implementation, a larger value corresponds to a smaller merge-sort computation time, as this approach minimizes the compaction time [1, 6].

After calculating the VF and HF for each SST file, we compute the $Cold_{SST_i}$ for each SST as follows.

$$Cold_{SST_i} = VF_{SST_i} \times \alpha + HF_{SST_i} \times (1-\alpha) \quad (1)$$

In this study, we set α to 0.5 to evenly weight the VF and HF. Additionally, if there are no Unpivot SST files overlapping with the Pivot SST file in RocksDB, a merge-sort is not performed, and the Pivot SST is simply moved to the next level. This special compaction process is referred to as a Trivial Move. The Pivot SST file involved in a Trivial Move is not deleted or invalidated until the next compaction at the next level selects it. Therefore, such Pivot SST files are considered as very cold SSTs, and their $Cold_{SST}$ is fixed to 1.0.



(a) ColdMap's Architecture and Operation.

(b) ColdMap(ZNS) Construction Procedure.

(c) An illustrative example of updating ColdMap(LSM) and LSM-tree Snapshot

Figure 4: Illustrations of ColdMap's (a) architecture, (b) construction, and (c) update mechanism with the LSM-tree.

5.2 Compaction Simulation and Updating ColdMap(LSM)

Each compaction event reshapes the LSM-tree and alters the coldness of SST files (Figure 2). To capture this dynamic behavior, we introduce an algorithm that predicts SST coldness by explicitly modeling future compactions.

Figure 4b illustrates the overall process. Starting from the initial ColdMap(LSM), ColdMap simulates future compactions for N iterations, updating the coldness of SST files after each simulated compaction. The resulting state constitutes the final ColdMap(LSM). We discuss how to determine an appropriate value of N in §5.2.1.

In LSM-trees, the coldness of an SST is governed not by past access history but by when it will be invalidated by future compactions. Accordingly, ColdMap aims to predict which SSTs will be selected by upcoming compactions and when they will be removed, without performing actual merge operations. To this end, ColdMap follows RocksDB's compaction selection rules and approximates compaction outcomes to incrementally simulate changes in the LSM-tree state. Each simulation step abstracts the essential semantics of compaction: (i) selecting compaction targets, (ii) invalidating existing SSTs, and (iii) generating new SSTs.

The detailed steps are described below.

(1) Building the LSM-tree Snapshot. ColdMap first captures a snapshot of the current LSM-tree as the baseline for simulation. The snapshot records only essential metadata for each SST file—its level, key range, and size—which are sufficient to reproduce compaction decisions without accessing data contents.

(2) Predicting Pivot and Unpivot SSTs. Using the snapshot, ColdMap predicts the next compaction by applying the same selection rules as RocksDB. Specifically, it computes L_Score to identify the level to compact and O_Score to select the Pivot SST at that level. Unpivot SSTs are then identified as those in the next level whose key ranges overlap with the Pivot. This step emulates which files the compaction algorithm would select next, as illustrated by the blue (Pivot) and red (Unpivot) SSTs in Figure 4c.

(3) Updating Coldness in ColdMap(LSM). *Files merged by the same compaction share the same remaining lifetime.* In a real compaction, the Pivot and Unpivot SSTs are merged and subsequently deleted, implying that they share the same remaining lifetime. To reflect this semantics, ColdMap assigns the same $Cold_{SST}$ value to the predicted Unpivot SSTs as that of the Pivot SST. This update models the fact that these files are invalidated together by the same compaction event. The effect is shown by the first two SSTs at L(2) in Figure 4c, whose $Cold_{SST}$ values are updated to match that of the Pivot SST at L(1).

(4) Updating the LSM-tree Snapshot. ColdMap then updates the snapshot to reflect the structural outcome of compaction. *(Structural update)* The Pivot and Unpivot SST files selected for compaction are first removed from the LSM-tree snapshot (shown as gray SSTs in the right subfigure of Figure 4c). The newly generated SST files are then inserted into the corresponding level (green). *(Approximating SST properties)* Because ColdMap does not perform an actual merge-sort, the sizes and key ranges of new SSTs are approximated. To address this, ColdMap approximates the properties of newly created SST files in the snapshot. First, it estimates the total output size by multiplying the combined size of the Pivot and Unpivot SST files by the average compression ratio observed in previous compactions at that level.

For example, with a 112 MB input and an average compression ratio of 0.73, the total output size is estimated as 82 MB. This output is then split according to RocksDB's maximum SST size, resulting in one 64 MB SST and a remainder SST of 18 MB, as illustrated in Figure 4c.

Next, ColdMap predicts the key range of each newly created SST by proportionally dividing the combined key range of the input SSTs based on their relative sizes. In the above example, the input key range [1–13] is divided into [1–10] for the 64 MB SST and [11–13] for the 18 MB SST. This approximation preserves the relative key-range layout required for subsequent compaction decisions.

(5) Iterative Simulation (Steps 2-4). The updated snapshot becomes the input to the next iteration. Steps (2)–(4) are repeated for N iterations until no further compaction is predicted. SST files deleted in earlier iterations are excluded from subsequent predictions, ensuring consistency across iterations.

Intra-L0 Compaction Exception. RocksDB disallows concurrent L0–L1 and L1–L2 compactions. To prevent excessive accumulation of unsorted SSTs in L0, RocksDB performs intra-L0 compaction, which merges overlapping SSTs within L0 only [4, 5]. Intra-L0 compaction merges SST files with overlapping key ranges within L0

and produces new SST files that remain in L0, ensuring that the compaction is confined to the L0 level. ColdMap explicitly models this behavior during simulation, ensuring that coldness updates remain consistent with RocksDB's actual compaction semantics.

5.2.1 ***Adaptive Determination of Simulation Depth.*** ColdMap predicts future invalidation by simulating the next N compactions at the time ZC is triggered. A small N fails to capture compactions that will actually occur before the next ZC, while a large N expands the prediction horizon but accumulates approximation errors from snapshot-based simulation, degrading accuracy. Thus, the optimal N depends on the number of compactions that occur between ZC events, which varies with write intensity.

Based on this observation, we adopt an adaptive scheme that sets N to the moving average of the number of compactions observed over the most recent K ZC intervals. Let C_t denote the number of compactions executed by RocksDB between ZC_{t-1} and ZC_t. At ZC_t, we compute $N_t = \left\lfloor \frac{1}{\min(t,K)} \sum_{i=1}^{\min(t,K)} C_{t-i+1} \right\rfloor$, with $K = 3$ in our implementation. Using a moving average smooths short-term fluctuations in compaction frequency, yielding a stable N. Because RocksDB already maintains compaction counters, this scheme introduces no additional overhead.

5.2.2 ***Coldness Modeling for WAL Files.*** In addition to SST files, RocksDB maintains WAL files to ensure crash consistency. WAL files are deleted only after a flush completes, when all key–value pairs in the Memtable have been safely persisted as SST files. As a result, WAL lifetime is governed by flush behavior rather than by compaction. Notably, RocksDB temporarily suspends flush operations during L(0)-to-L(1) compactions [60, 62]. Based on this observation, ColdMap assigns a WAL file the same coldness as L(0) SST files when the compaction simulation predicts an upcoming L(0)-to-L(1) compaction, indicating that flush—and thus WAL deletion—will be deferred. In contrast, when the predicted next operation is not an L(0)-to-L(1) compaction, implying that a flush may occur immediately, the coldness of the WAL file is set to 0.

5.3 Cost–Benefit–Based Zone Cleaning with ColdMap(ZNS)

To apply cost–benefit analysis to zone cleaning (ZC), we first construct ColdMap(ZNS), which assigns a coldness value to each zone. In ZNS-based systems, a zone may contain multiple SST and WAL files with heterogeneous sizes. As discussed in § 2.3, this heterogeneity makes it insufficient to characterize a zone using the access history of a single file.

Let $zone_k$ contain n SST or WAL files. The coldness of each file is obtained from ColdMap(LSM), which tracks file-level coldness within the LSM-tree. We derive the coldness of $zone_k$ as a size-weighted average of the coldness of its constituent files:

$$Cold_{Zone_k} = \frac{\sum_{i=1}^{n} \left(Cold_{SST_i \text{ or } WAL_i} \times Size(SST_i \text{ or } WAL_i) \right)}{\sum_{i=1}^{n} Size(SST_i \text{ or } WAL_i)} \quad (2)$$

Once ColdMap(ZNS) is constructed, the computed $Cold_{Zone_k}$ values are substituted into the cost–benefit formulation [56], which is a ZNS-adapted version of Equation 3, to compute the score for each zone. The zone with the highest cost–benefit score is then selected as the victim for ZC.

$$\frac{Benefit_{Zone_k}}{Cost_{Zone_k}} = \frac{F_{Zone_k} \times Cold_{Zone_k}}{C_{Zone_k}} \quad (3)$$

Here, F_{Zone_k} denotes the amount of free space reclaimed by cleaning zone $Zone_k$, while C_{Zone_k} represents the amount of valid data that must be copied during the cleaning process. $Cold_{Zone_k}$ captures the coldness of zone $Zone_k$, reflecting how unlikely the data in the zone is to be modified in the future.

5.4 ColdMap beyond Leveled Compaction

Universal Compaction unifies several variants of tiered compaction and is known to achieve low write amplification under write-intensive workloads. However, by deferring merges and maintaining multiple sorted runs concurrently, it incurs higher read overhead and increased space usage due to prolonged data duplication.

Universal Compaction selects merge candidates by evaluating four trigger conditions in a fixed priority order. It first checks *periodic compaction*, which forces merges after a time threshold. If unmet, it evaluates the *space amplification* condition based on the ratio of total data size to non-bottom runs. Next, it applies the *size ratio* condition, cumulatively merging adjacent runs whose size ratios fall below a threshold, starting from the newest run. Finally, it enforces a *sorted-run count* limit by merging the smallest runs when the number of runs exceeds a bound. Once a higher-priority condition is satisfied, lower-priority conditions are not considered, thereby determining both the timing and scope of compaction.

To reflect this behavior, ColdMap models Universal Compaction by faithfully emulating its trigger rules. Given the current LSM-tree state, ColdMap evaluates the triggers in priority order to identify files most likely to be compacted next. Files selected by an active trigger (e.g., via the size-ratio condition) are assigned low coldness (hot), while files not matching any trigger are initialized as cold.

ColdMap then refines coldness by iteratively simulating N rounds of compaction. In each round, it updates the LSM-tree snapshot to reflect predicted merges and re-evaluates the trigger conditions, marking newly selected compaction candidates as hot.

The subsequent zone-level coldness aggregation and cost–benefit based victim selection follow the same procedure as in leveled compaction (§5.1).

Generalizability across Compaction Policies. RocksDB supports a wide range of compaction policies beyond the leveled and universal schemes evaluated earlier, including variants such as Leveled-N [21]. Even within leveled compaction itself, multiple variants are employed depending on the pivot selection priority [53]. For example, in addition to the default `kMinOverlappingRatio`, which prioritizes SSTs with minimal overlap with the next level, RocksDB provides alternative policies such as `RoundRobin`, `kOldestSmallestSeqFirst`, and `kByCompensatedSize` [53].

Despite these variations, all policies share a common structure in determining *which level* to compact (corresponding to ColdMap's VF), while differing only in *which SST* is selected as the pivot within the chosen level (corresponding to ColdMap's HF). As a result, ColdMap achieves structural generality by adapting only the HF computation logic for each policy, while reusing the rest of the pipeline unchanged.

Generalizability to other LSM KVS. Our simulation method can be generalized in different LSM-tree based KV DB. LSM-tree KVS compaction policies fall into two families: Leveled and Tiered [20]. In the Leveled family, BadgerDB's [8] compaction policy is nearly identical to RocksDB's, differing only in that it skips L_1 when the DB size is below a threshold. Adding this single condition to the simulation is sufficient for adaptation. In the Tiered family, Cassandra's [7] default strategy STCS is representative. Universal Compaction simulates four trigger conditions including a size-tiered condition, while STCS operates using only the size-tiered condition among them. Although STCS's size-tiered condition differs from Universal's size ratio condition in its grouping mechanism, the simulation pipeline structure remains identical, so extension requires only replacing the trigger logic.

6 Evaluation

6.1 Experimental Setup

Implementation and Configuration. We implement CAZA [15], a state-of-the-art lifetime-aware data placement scheme, on RocksDB v7.4.4 with ZenFS v2.1. On this baseline, we evaluate ZenFS's default greedy zone cleaning, a file-system–level cost–benefit scheme [56], a state-of-the-art firmware-level cost–benefit scheme [61], and our proposed ColdMap. We also implement Prophet [43] and, together with CAZA, use it to assess the limitations of zone allocation and cleaning in §6.2.

To formularize a highly concurrent environment, we configured RocksDB with four flush threads, four compaction threads, and eight subcompaction threads. FAR [16] was enabled throughout execution and triggered when free space dropped below 25%, while all other RocksDB parameters remained at their default settings.

Workloads. We used four distinct categories of workloads, each designed to capture different access patterns and operational characteristics (Table 2). The characteristics of each workload group are summarized as follows.

- **Production-Level Workloads**: Two realistic MixGraph benchmarks [17, 59] are used, along with a production-derived operation mix (57:41:2) from Nutanix Inc., whose cloud storage stack is built on RocksDB [10, 11, 44].
- **Write-Intensive Synthetic Workloads**: RocksDB's `db_bench` workloads (`fillrandom` and `readwriterandom`) and write-only YCSB workloads (uniform and zipfian, $\alpha = 0.9$) are employed to stress the system under heavy write pressure [12].
- **Read-Dominant Synthetic Workloads**: Three standard YCSB workloads with read-heavy access patterns are selected to evaluate system behavior under mixed and read-dominant workloads.
- **Transactional OLTP Workloads**: Three OLTP workloads from Sysbench [37] and OLTPBase [22] are executed on MariaDB with the RocksDB storage engine (MyRocks [48]) to assess performance under realistic transactional database workloads.

We compare the following four algorithms.

- **Greedy**: A baseline ZC algorithm that selects the zone with the highest invalid ratio as the victim. This policy has been used as the default ZC algorithm in all prior LSM–ZNS studies.
- **CBSC [56]**: A cost–benefit segment cleaning algorithm originally proposed for LFS. We reimplement CBSC in the context of LSM–ZNS for the first time by mapping segments and blocks to zones and SST files, respectively.
- **FaGC+ [61].** A state-of-the-art cost–benefit strategy proposed at the SSD firmware level. We implement FaGC+ in ZenFS to represent firmware-inspired cost–benefit ZC policies.
- **ColdMap.** Our proposed algorithm, which predicts file coldness via compaction-aware simulation. Unless otherwise stated, ColdMap is evaluated under Leveled Compaction.

Table 2: Workloads used in our evaluation. (P:G:S) denotes the ratio of *Put:Get:Scan* operations.

Group	Workload	Scale	(P:G:S)	Characteristic
Production-level	nutanix	80M OPS	57:41:02	Write-intensive
	mixgraph1	120M OPS	80:17:03	Write-intensive
	mixgraph2	120M OPS	14:83:03	Read-intensive
RDBMS (MyRocks)	oltp-insert [37]	3600 sec	50:50:00	Write-intensive
	oltp-rw [37]	4800 sec	08:71:21	Read-intensive
	tpc-c [22]	4800 sec	02:93:05	Read-intensive
Write synthetic	fillrandom	164M OPS	100:00:00	Write-intensive
	YCSB-zipfian	120M OPS	100:00:00	Write-intensive
	YCSB-uniform	66M OPS	100:00:00	Write-intensive
	rwrandom	60M OPS	80:20:00	Write-intensive
Read synthetic	YCSB-A	10M OPS	50:50:00	Read-intensive
	YCSB-B	5M OPS	05:95:00	Read-intensive
	YCSB-D	10M OPS	05:95:00	Read-intensive

Our evaluation uses a WD ZN540 SSD with 904 zones, each of 1077 MB [3]; for experimental convenience, 103 zones are mounted in ZenFS. Experiments are conducted on a server with an Intel(R) i7-14700K CPU (26 cores), 32 GB of memory, running Linux kernel v6.8.0 and Ubuntu 20.04 LTS.

6.2 Inherent Limitations of Allocation- and Cleaning-Based ZC Mitigation

In this section, we show that even state-of-the-art zone allocation schemes (CAZA [15] and Prophet [43]) and proactive zone cleaning (ZC) schemes (e.g., FAR [16]) suffer from fundamental limitations.

6.2.1 ***Residual Lifetime Heterogeneity under Allocation-Based Placement.*** Although CAZA [15] and Prophet [43] perform lifetime-aware placement to co-locate data with similar expected lifetimes, residual lifetime heterogeneity persists within zones, inevitably leading to valid data copying during ZC and undermining the effectiveness of both zone allocation and cleaning.

Figure 5 illustrates file lifetimes within a representative zone managed by CAZA. Each horizontal line represents the lifetime of a file, from its placement into the zone to its deletion. Most files placed during the [50–150] second interval are deleted shortly thereafter; however, a small subset (files 6, 11, and 13) persist for an additional ~100 seconds, significantly longer than the others. A similar pattern appears in the [700–800] second interval, where files 1–8 are deleted within 20 seconds, while files 11, 13, and 21 survive until a subsequent ZC event. These long-lived files ultimately trigger valid data copying during ZC, despite being co-located with predominantly short-lived data. This observation indicates that CAZA does not consistently group files with truly similar lifetimes within the same zone.

This limitation is not unique to CAZA. Figure 6 shows that Prophet exhibits similar lifetime divergence within a single zone.

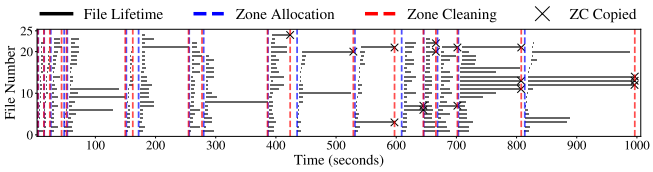


Figure 5: Lifetime difference of files placed in a sample zone using CAZA [15] (until 1000 seconds, YCSB-zipfian).

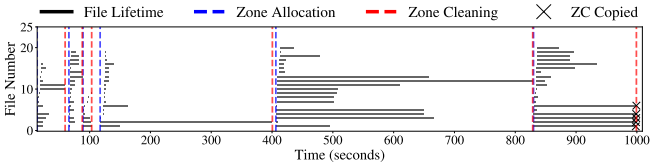


Figure 6: Lifetime difference of files placed in a sample zone using Prophet [43] (until 1000 seconds, YCSB-zipfian).

Thus, despite explicitly targeting lifetime-aware placement, state-of-the-art zone allocation techniques remain insufficient to fully separate hot and cold data, inevitably leading to valid data copying during ZC.

6.2.2 ***Limits of Proactive Zone Cleaning under Open-Zone Constraints.*** To mitigate the overhead of valid data copying that remains unavoidable under allocation-based schemes such as CAZA and Prophet, prior work has explored proactive zone cleaning (ZC) mechanisms. FAR [16] represents a representative approach in this direction, aiming to reduce ZC-induced overhead and I/O blocking by proactively cleaning zones selected by a greedy policy. However, the effectiveness of FAR is fundamentally constrained by the open-zone limit enforced by ZNS SSDs for DRAM and power efficiency. In practice, this limit is typically restricted to a small number of zones (e.g., 14 or 256) [3, 31, 50], which severely limits the flexibility of proactive ZC and restricts its ability to avoid valid data copying.

We used the FEMU-based ZNS emulator, ConfZNS [24, 58], to quantitatively evaluate how the open zone limit constrains the effectiveness of proactive ZC by measuring the amount of valid data copied during cleaning. Since commercial ZNS devices, such as the WD ZN540 [3], do not allow the open zone limit to be modified, ConfZNS was the only practical option for isolating and studying this effect.

In our evaluation, we configured the zone size to 1 GB and the total ZNS SSD capacity to 100 GB. We employed CAZA as the zone allocation algorithm and varied the open zone limit to examine how this hardware-imposed constraint affects the potential performance gains of proactive ZC.

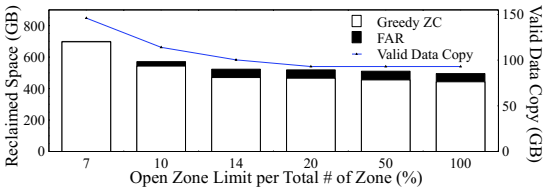
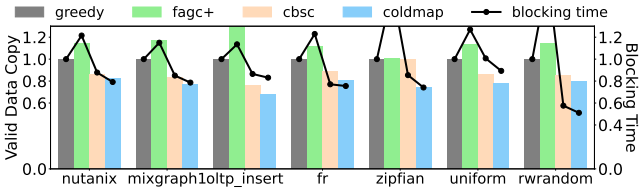
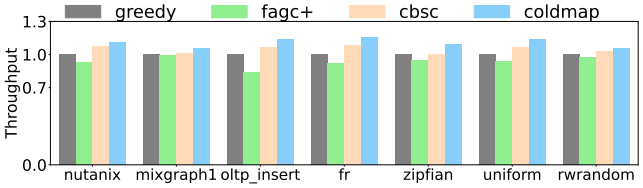


Figure 7: Space reclaimed by FAR and ZC with different number of open zone limit in YCSB-zipfian workload.



(a) Normalized Data Copy and I/O Blocking from ZC.



(b) Normalized Throughput.

Figure 8: ZC overhead and throughput under write-intensive workloads (PUT ≥ 50%).

Figure 7 shows that FAR reclaims little free space when the open zone limit is restricted to 7% of the total number of zones. Under this setting, the effectiveness of FAR is largely eliminated, and the amount of valid data copying incurred by greedy ZC reaches its maximum. Notably, real ZNS SSDs impose an even stricter constraint: for example, the WD ZN540 allows only 14 open zones out of 901, corresponding to an open zone limit of approximately 1.5%. This suggests that FAR's potential benefits are severely limited in practical deployments.

Increasing the open zone limit alleviates this constraint only partially. Even when the open zone limit is increased to approach the total number of zones, valid data copying is not eliminated and instead saturates at approximately 90–100 GB. This saturation indicates that proactive cleaning alone is insufficient to fundamentally remove ZC overhead.

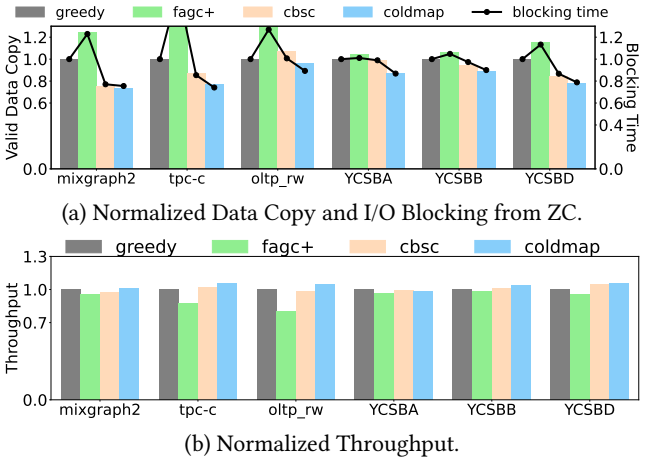
A low open zone limit also directly restricts lifetime-based data segregation. When the open zone limit is extremely small (e.g., one open zone), files with different lifetimes must be placed into the same zone, leaving no opportunity for separation. As a result, long-lived data remain valid during ZC, preventing zones from becoming fully invalid and limiting the number of reclaimable zones.

Overall, these results show that FAR struggles to generate sufficient free space or reduce ZC overhead under the open zone limits imposed by ZNS SSDs. Even with a significantly increased open zone limit, proactive ZC cannot eliminate valid data copying, revealing a fundamental limitation of cleaning-based approaches.

6.3 Performance across Workloads

We evaluate ColdMap under diverse workload characteristics to assess how its prediction-driven cost–benefit design improves performance in LSM–ZNS systems.

6.3.1 ***Write-Intensive Workloads.*** We evaluate ColdMap under seven write-intensive workloads where ZC events occur frequently (Figure 8 (a)). We compare ColdMap against Greedy (the ZenFS default algorithm) and our implemented methods FaGC+ and CBSC. The results show that ColdMap reduces ZC overhead by an average



(a) Normalized Data Copy and I/O Blocking from ZC.

(b) Normalized Throughput.

Figure 9: ZC overhead and throughput under read-intensive workloads (PUT Ratio < 50%).

of 35% (up to 189–4%) compared to all comparison methods, achieving the lowest overhead across all workloads. Notably, ColdMap achieves up to 49% ZC overhead reduction compared to the Greedy baseline. This reduction in ZC overhead translates to improved throughput (Figure 8 (b)). ColdMap improves throughput by an average of 11% (up to 27–3%) compared to all methods, with up to 15.7% throughput improvement over the Greedy baseline.

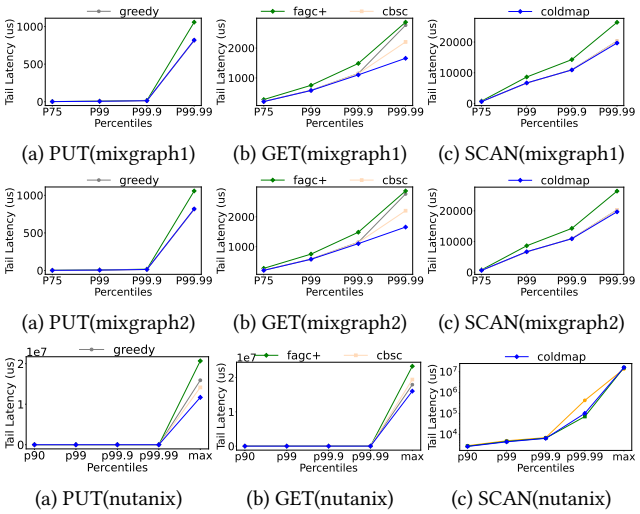
6.3.2 ***Read-Intensive Workloads.*** ColdMap also mitigates ZC overhead in read-intensive scenarios, reducing it by 138–3% compared to the baselines (Figure 9 (a)). However, in these cases, the throughput improvement of ColdMap is limited (Figure 9 (b)). This is because most read operations (e.g., GET, SCAN) can proceed immediately, even if ZC events occur, leaving little room for further performance gains.

6.3.3 ***Tail Latency on Production Workloads.*** To evaluate whether ColdMap degrades tail latency under intensive ZC activity, we measure the tail latency of PUT, GET, and SCAN operations using three production-level workloads that reflect real-world request patterns. ColdMap reduces both the total ZC copy volume and the invocation frequency, thereby mitigating ZC-induced I/O blocking. Figure 10 shows that these improvements do not come at the cost of tail latency degradation. Across all workloads and operation types, ColdMap achieves comparable or lower latency at all reported percentiles compared to prior approaches. These results confirm that ColdMap's throughput gains are achieved without sacrificing tail latency.

6.4 How Can ColdMap Improve ZC Efficiency

We analyze ColdMap's coldness metric ($Cold_{zone}$, Equation 2) used for victim zone selection, and examine the remaining lifetime of files migrated from those zones to assess how many truly cold files are copied to improve ZC efficiency.

6.4.1 ***How cold are the zones selected by ColdMap?*** We compare the coldness of victim zones selected by ColdMap and Greedy, the latter being the default algorithm commonly adopted in prior LSM-ZNS studies. This comparison is intended to verify whether



(a) PUT(mixgraph1) (b) GET(mixgraph1) (c) SCAN(mixgraph1)

(a) PUT(mixgraph2) (b) GET(mixgraph2) (c) SCAN(mixgraph2)

(a) PUT(nutanix) (b) GET(nutanix) (c) SCAN(nutanix)

Figure 10: Tail latencies of production-level workloads.

ColdMap can more accurately identify and select truly cold zones as victims.

Figure 11 presents the results of this comparison. For the evaluation, we used two metrics: the average coldness of victim zones selected during each zone cleaning (ZC) call, and a weighted average coldness metric, $WA_{Cold_{Zone}}$, which accounts for the amount of data copied in each ZC call.

$$WA\_Cold_{Zone} = \frac{\sum_{k=1}^{n}(Cold_{Zone_k} \times Copied_{z_k})}{\sum_{k=1}^{n} Copied_{z_k}} \quad (4)$$

The results show that, under write-intensive workloads, ColdMap consistently selects victim zones with 10–20% higher values in both $Cold_{Zone}$ and $WA_{Cold_{Zone}}$ compared to Greedy. The results confirm that ColdMap effectively selects cold victim zones.

6.4.2 ***How long do the victim files selected by ColdMap actually survive?*** $Cold_{zone}$ represents the coldness value of a victim zone as predicted by the ColdMap algorithm. To verify whether this value truly reflects the coldness of a zone, we measure how long the files copied during ZC remain alive. For this purpose, we adopt a modified version of the FC_Ticks metric originally proposed in Prophet [43].

FC_Ticks is a term derived from the first letters of Flush and Compaction, which represent two core operations in LSM-tree systems. This metric serves as an indicator of a file's lifetime, increasing the file's FC_Ticks by one each time a compaction or flush occurs while the file is still valid. In our version, we extend the original FC_Ticks metric to also include WAL file write operations, such that the tick value increases for each compaction, flush, or WAL write event that occurs while the file is alive. That is, cold files that persist longer tend to experience more compaction, flush, or WAL operations, resulting in higher FC_Ticks values.

Figure 12 visualizes the post-copy lifetime of files copied during ZC using the FC_Ticks distribution. As shown in the figure, across all three workloads, files copied by ColdMap record, on average, approximately 50 higher FC_Ticks values than those copied by Greedy. Specifically, in the case of Greedy, the frequency is concentrated in the lower region of the graph, indicating that many copied files

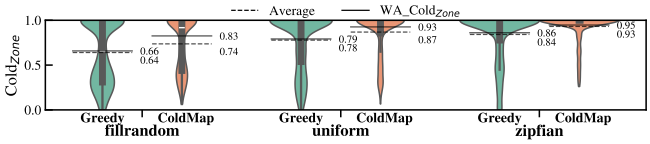


Figure 11: Distribution of $Cold_{Zone}$ and WA_Cold_{Zone}.

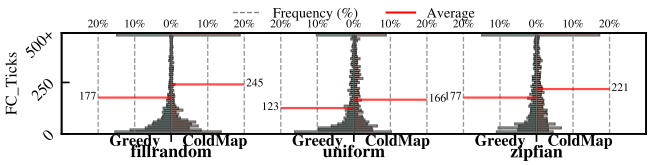


Figure 12: Distribution of copied files' FC_Ticks. Higher tick value means longer lifetime.

were deleted shortly after the copy. In contrast, ColdMap shows a higher frequency in the upper region, meaning that the copied files remained valid for a longer time after ZC.

These results demonstrate that ColdMap more effectively selects truly cold files as victims, and that those files tend to remain alive for a longer duration.

6.5 ColdMap Construction Overhead

We evaluate the overhead incurred by running ColdMap.

Time Complexity. Constructing ColdMap takes $O(N)$ time, where N is the number of SST files. Each compaction simulation also incurs $O(N)$ time to identify the next compaction candidate via a max operation over SSTs. However, in practice, ColdMap is built statelessly only when ZC is triggered, which occurs infrequently as it is activated only when free space drops below a threshold. Across all workloads, the construction overhead accounts for less than 0.08% of total runtime, making the cost of building ColdMap negligible.

Memory Overhead. As for memory overhead, only 5MB of DRAM is required when operating with a 10TB ZNS SSD. Assuming the default SST file size is 64MB, Each SST entry in ColdMap(LSM)/Snapshot requires 32 bytes of metadata: 8 bytes each for the start and end key ranges, 8 bytes for file size, and 8 bytes for coldness. A 10TB SSD stores approximately 160K SST files, leading to a total metadata memory requirement of $160K \times 32$ bytes = 5MB.

CPU Overhead. ColdMap exhibited the lowest cumulated CPU usage (ColdMap = 7.51%, CBSC = 8.64%, greedy = 9.11%) during runtime. This is because the reduction in CPU usage gained from fewer ZC copies and fewer ZC operations outweighs the additional CPU cost incurred for building ColdMap.

6.6 Design Sensitivity and Ablation Study

We present a sensitivity and ablation study of ColdMap, focusing on key design parameters that affect prediction accuracy and performance.

6.6.1 ***Impact of Compaction Simulation Depth on Prediction Accuracy.*** The prediction accuracy of ColdMap depends on how deeply future compactions are simulated. This section evaluates the effectiveness of the simulation-depth setting introduced in §5.2.1, focusing on the trade-off between under- and over-simulation.

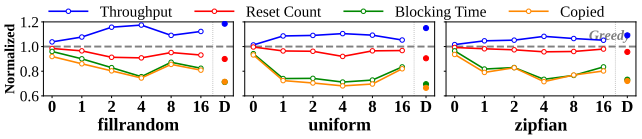


Figure 13: Impact of compaction simulation iterations on system behavior. 0–16 denote static N values (hollow markers), while D denotes the adaptive N setting (filled markers).

Figure 13 compares a static simulation depth N with an adaptive scheme that dynamically adjusts N based on observed compaction behavior. With a static setting, increasing N initially improves performance by better capturing compactions that occur before the next ZC. However, beyond a moderate depth, performance degrades as approximation errors accumulate, leading to inaccurate coldness estimation and suboptimal victim selection.

The adaptive scheme addresses this limitation by aligning the simulation depth with the actual number of compactions occurring between ZC events. In practice, the number of compactions between consecutive ZCs varies over time depending on write intensity, making a fixed N inherently brittle. By setting N to a moving average of recently observed compaction counts, the adaptive scheme tracks this variability and avoids both under- and over-simulation.

As a result, the adaptive scheme consistently matches or exceeds the best statically tuned configuration across workloads. More importantly, it achieves this robustness without prior tuning or workload-specific parameter selection, automatically adjusting the simulation depth as compaction behavior changes over time. This confirms that dynamically adapting N is essential for maintaining accurate coldness prediction under diverse and evolving workloads.

6.6.2 ***Impact of Weighting Vertical and Horizontal Factors.*** In §5.1, we defined $Cold_{SST}$ as a weighted combination of the HF and the VF, controlled by the parameter α. To evaluate the importance of each factor, we vary α while fixing the number of compaction simulation iterations to 4 as our earlier results show that performance is maximized around this value.

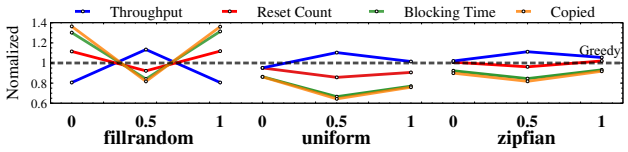


Figure 14: Impact of balancing vertical and horizontal factors on performance. Numbers on x-axis are α value, described on Equation 1 (§5.1).

In Figure 14, $\alpha = 0.5$, which assumes the same weight for VF and HF, shows better performance across all three different workloads. We averaged the results over three trials for each configuration. The standard deviation ranged from 0.03 to 0.1, indicating consistent outcomes across runs. For example, in the fillrandom workload, if α is set to 0 or 1, it degrades performance. Therefore, we recommend that both factors be equally considered, by setting α as 0.5.

6.7 Effectiveness under Large Zone Sizes

With the emergence of peta-scale SSDs, zone sizes are expected to grow even larger. Accordingly, we evaluated whether the proposed

Table 3: Performance under Universal Compaction (fillrandom). All values are normalized to Greedy.

Metric	Greedy	FaGC+	CBSC	ColdMap
ZC Data Copy ↓	1.00 (6.7 GB)	2.86	1.61	**0.31**
Throughput ↑	1.00 (214.8 Kops/sec)	0.87	0.93	**1.07**

ColdMap remains effective under such conditions. The WD ZN540 has a fixed zone size of 1077MB. To assess the impact of larger zone sizes, we used ConfZNS [58], configured with a 2GB zone size, and conducted experiments using the Nutanix workload. As a result, ColdMap reduced the amount of valid data copied and I/O blocking time by 10% compared to the existing baselines. It also improved throughput by about 8%. These results confirm that ColdMap remains effective even with larger zone sizes.

6.8 Policy-Agnostic Effectiveness of ColdMap

ColdMap is applicable to a wide range of compaction policies by incorporating policy-specific trigger conditions into its simulation model. We experimentally evaluate the effectiveness of ColdMap under *universal compaction*. The evaluation follows the same setup as §6.1, replacing leveled compaction with universal compaction and using the *fillrandom* workload.

Table 3 compares ColdMap against Greedy, the default ZC policy used in ZenFS. ColdMap reduces ZC overhead by 68.6% and improves throughput by 7% compared to Greedy. These results demonstrate that ColdMap remains effective across compaction policies despite differences in their internal logic.

7 Related Work

Victim Selection in LFS and SSD Firmware. Cost–benefit–based victim selection has been widely adopted in LFS and SSD firmware. CBSC [56], originally proposed for LFS, estimates segment coldness using the timestamp of the most recently modified block to minimize recopying cost. F2FS [39] builds on this idea but observes that data age and utilization have different units and conflicting effects; it therefore applies min–max normalization and computes a weighted sum for more balanced victim selection. ATGC [2, 49] further extends F2FS by excluding hot segments likely to self-invalidate in the near future, thereby reducing write amplification and improving GC efficiency.

Similar ideas have evolved in SSD firmware, including FeGC [38], FAST-CB [51], and FaGC+ [61]. FeGC shows that invalidation histories better capture a block's future modification behavior than the age of valid pages, and thus identifies blocks that have reached a stable state with low self-invalidation likelihood. FAST-CB addresses the scalability of cost–benefit selection, whose complexity grows linearly with the number of blocks, by grouping blocks into age ranges and scanning from the oldest group to reduce selection time while preserving WA characteristics. FaGC+ further refines block-level estimation by recording per-page modification times and computing a cost–benefit score that combines the average age of valid pages with the invalid ratio.

LSM-tree Optimization on ZNS SSDs. Beyond the data placement and compaction optimizations discussed in §1 [15, 34, 40, 43, 45, 57], prior work has explored improving other aspects of LSM–ZNS systems. SplitZNS [30] improves LSM-tree GC efficiency by introducing small zones via zone-to-chip remapping, while WA-Zone [46] optimizes device lifetime through wear-aware zone management. WALTZ [41] reduces WAL tail latency using Zone Append, and Hwang et al. [33] minimize data movement by coordinating compaction with ZenFS GC. However, none of these approaches directly optimize victim zone selection at ZC time.

8 Conclusion

Despite recent advances in zone allocation and proactive ZC, ZNS file systems still suffer from substantial zone-cleaning overhead. To address this issue, we propose ColdMap, which accurately predicts cold zones at ZC time by simulating LSM-tree compaction. By explicitly modeling future invalidation, ColdMap eliminates false positives inherent in history-based predictors. Across diverse workloads, ColdMap reduces ZC-induced data copying by up to 49% and improves overall throughput by an average of 15.7%, outperforming Greedy, CBSC, and FaGC+, the state-of-the-art victim selection schemes at the file-system and firmware levels.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (RS-2025-00564249 and RS-2024-00416666), by the Institute for Information & Communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. RS-2025-02215256), by Samsung Electronics Co., Ltd. (IO260120-15258-01), and by the US National Science Foundation under grant Numbers #2412436 and #2443219.

References

[1] 2016. RocksDB 4.8 Released. https://rocksdb.org/blog/2016/07/26/rocksdb-4-8-released.html.
[2] 2020. Age-Threshold Segment Cleaning in F2FS. https://lwn.net/Articles/828027/.
[3] 2023. Western Digital Ultrastar DC ZN540. https://hypertecsp.com/wp-content/uploads/data-sheet-ultrastar-dc-zn540-1.pdf.
[4] 2024. compaction_picker_level.cc::line201. https://github.com/facebook/rocksdb/blob/main/db/compaction/.
[5] 2024. RocksDB Compaction Documents. https://github.com/facebook/rocksdb/wiki/Leveled-Compaction.
[6] 2025. kMinOverLappingRatio. https://github.com/facebook/rocksdb/blob/main/include/rocksdb/advanced_options.h.
[7] 2026. Apache Cassandra. https://github.com/apache/cassandra.
[8] 2026. BadgerDB. https://github.com/dgraph-io/badger.
[9] Hanyeoreum Bae, Jiseon Kim, Miryeong Kwon, and Myoungsoo Jung. 2022. What You Can't Forget: Exploiting Parallelism for Zoned Namespaces. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*.
[10] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*. 363–375.
[11] Oana Balmau, Florin Dinu, Willy Zwaenepoel, Karan Gupta, Ravishankar Chandhiramoorthi, and Diego Didona. 2019. SILK: Preventing Latency Spikes in Log-Structured Merge Key-Value Stores. In *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*. 753–766.
[12] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 753–768.
[13] Shai Bergman, Niklas Cassel, Matias Bjørling, and Mark Silberstein. 2023. ZNSwap: Un-block Your Swap. *ACM Transactions on Storage* 19, 2 (2023), 1–25.
[14] 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 2021 USENIX Annual Technical Conference (ATC)*. 689–703.
[15] 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 *Proceedings of the 38th International Conference on Massive Storage Systems and Technology (MSST)*.
[16] 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 15th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*. 109–115.
[17] Zhichao Cao, Siying Dong, Sagar Vemuri, and David HC Du. 2020. Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST)*. 209–223.
[18] M-L Chiang and R-C Chang. 1999. Cleaning Policies in Mobile Computers Using Flash Memory. *Journal of Systems and Software* 48, 3 (1999), 213–231.
[19] Gunhee Choi, Kwanghee Lee, Myunghoon Oh, Jongmoo Choi, Jhuyeong Jhin, and Yongseok Oh. 2020. A New LSM-style Garbage Collection Scheme for ZNS SSDs. In *Proceedings of the 12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*.
[20] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal navigable key-value store. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 79–94.
[21] 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 2018 International Conference on Management of Data (SIGMOD)*. 505–520. https://doi.org/10.1145/3183713.3196927
[22] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *PVLDB* 7, 4 (2013), 277–288. http://www.vldb.org/pvldb/vol7/p277-difallah.pdf
[23] Western Digital. 2022. ZenFS. https://github.com/westerndigitalcorporation/zenfs.
[24] 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 17th ACM International Systems and Storage Conference (SYSTOR)*. 162–177.
[25] Krijn Doekemeijer, Zebin Ren, Nick Tehrany, and Animesh Trivedi. 2024. ZWAL: Rethinking Write-ahead Logs for ZNS SSDs with Zone Appends. *Proceedings of ACM SIGOPS Operating Systems Review* 58, 1 (2024), 53–60.
[26] Inc. Facebook. 2024. Facebook. https://engineering.fb.com/2016/08/31/core-infra/myrocks-a-space-and-write-optimized-mysql-database/. Accessed: 2024-10-19.
[27] Inc. Google. 2011. LevelDB. https://github.com/google/leveldb. Accessed: 2024-10-15.
[28] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. 2021. ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction. In *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.
[29] Hans Holmberg. 2020. ZenFS, Zones and RocksDB - Who Likes to Take out the Garbage Anyway? https://snia.org/sites/default/files/SDC/2020/074-Holmberg-ZenFS-Zones-and-RocksDB.pdf.
[30] 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 Transactions on Architecture and Code Optimization* 20, 3 (2023), 1–26.
[31] Joo-Young Hwang, Seokhwan Kim, Daejun Park, Yong-Gil Song, Junyoung Han, Seunghyun Choi, Sangyeun Cho, and Youjip Won. 2024. ZMS: Zone Abstraction for Mobile Flash Storage. In *Proceedings of the 2024 USENIX Annual Technical Conference (ATC)*.
[32] Sang-Ho Hwang, Ju Hee Choi, and Jong Wook Kwak. 2016. Migration Cost Sensitive Garbage Collection Technique for Non-Volatile Memory Systems. *IEICE TRANSACTIONS on Information and Systems* 99, 12 (2016), 3177–3180.
[33] Hamin Hwangbo, Joseph Ro, Sungjin Byeon, Safdar Jamil, Jun Young Han, Jooyoung Hwang, and Youngjae Kim. 2024. Towards A Unified Garbage Collection Strategy in ZNS Key-Value Store File Systems Using Same-Victim GC. In *Proceedings of the 32nd International Conference on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 1–8.
[34] Jeeyoon Jung and Dongkun Shin. 2022. Lifetime-Leveling LSM-Tree Compaction for ZNS SSD. In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*.
[35] Atsuo Kawaguchi, Shingo Nishioka, and Hiroshi Motoda. 1995. A Flash-Memory Based File System. In *Proceedings of the 1995 USENIX Annual Technical Conference (ATC)*. 155–164.
[36] Thomas Kim, Jekyeom Jeon, Nikhil Arora, Huaicheng Li, Michael Kaminsky, David G Andersen, Gregory R Ganger, George Amvrosiadis, and Matias Bjørling. 2023. 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 (ASPLOS)*. 660–673.
[37] Alexey Kopytov. 2012. Sysbench Manual. *Mysql Ab* (2012), 2–3.
[38] Ohhoon Kwon, Kern Koh, Jaewoo Lee, and Hyokyung Bahn. 2011. FeGC: An Efficient Garbage Collection Scheme for Flash Memory Based Storage Systems. *Journal of Systems and Software* 84, 9 (2011), 1507–1523.
[39] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. 2015. F2FS: A New File System for Flash Storage. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST)*.
[40] 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 *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*.
[41] Jongsung Lee, Donguk Kim, and Jae W Lee. 2023. WALTZ: Leveraging Zone Append to Tighten the Tail Latency of LSM Tree on ZNS SSD. *Proceedings of the VLDB Endowment* 16, 11 (2023), 2884–2896.
[42] Biyong Liu, Yuan Xia, Xueliang Wei, and Wei Tong. 2023. LifetimeKV: Narrowing the Lifetime Gap of SSTs in LSMT-Based KV Stores for ZNS SSDs. In *Proceedings of the 2023 IEEE 41st International Conference on Computer Design (ICCD)*. IEEE, 300–307.
[43] 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 *Proceedings of the 38th International Conference on Massive Storage Systems and Technology (MSST)*.
[44] Ming Liu, Arvind Krishnamurthy, Harsha V Madhyastha, Rishi Bhardwaj, Karan Gupta, Chinmay Kamat, Huapeng Yuan, Aditya Jaltade, Roger Liao, Pavan Konka, et al. 2020. Fine-Grained Replicated State Machines for a Cluster Storage System. In *Proceedings of the 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 305–323.
[45] Renping Liu, Junhua Chen, Peng Chen, Linbo Long, Anping Xiong, and Duo Liu. 2024. Hi-ZNS: High Space Efficiency and Zero-Copy LSM-Tree Based Stores on ZNS SSDs. In *Proceedings of the 53rd International Conference on Parallel Processing (ICPP)*.
[46] 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* 21, 1 (2024), 1–23.
[47] ByteDance Ltd. 2024. ByteDance. https://www.bytedance.com. Accessed: 2024-10-19.
[48] Yoshinori Matsunobu, Siying Dong, and Herman Lee. 2020. MyRocks: LSM-Tree Database Storage Engine Serving Facebook's Social Graph. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3217–3230.
[49] Jai Menon and Larry Stockmeyer. 1998. An Age-Threshold Algorithm for Garbage Collection in Log-Structured Arrays and File Systems. In *High Performance Computing Systems and Applications*. Springer, 119–132.
[50] Jaehong Min, Chenxingyu Zhao, Ming Liu, and Arvind Krishnamurthy. 2023. eZNS: An Elastic Zoned Namespace for Commodity ZNS SSDs. In *Proceedings of the 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 461–477.
[51] Lars Nagel, Tim Süß, Kevin Kremer, M Umar Hameed, Lingfang Zeng, and André Brinkmann. 2018. Time-Efficient Garbage Collection in SSDs. *arXiv preprint arXiv:1807.09313* (2018).
[52] Myounghoon Oh, Seehwan Yoo, Jongmoo Choi, Jeongsu Park, and Chang-Eun Choi. 2023. ZenFS+: Nurturing Performance and Isolation to ZenFS. *IEEE Access* 11 (2023), 26344–26357.
[53] RocksDB. 2016. Option of Compaction Priority. https://rocksdb.org/blog/2016/01/29/compaction_pri.html. Accessed: 2026-02-07.
[54] RocksDB. 2022. RocksDB. https://github.com/facebook/rocksdb.
[55] RocksDB. 2025. Leveled Compaction Wiki. https://github.com/facebook/rocksdb/wiki/Leveled-Compaction.
[56] Mendel Rosenblum and John K Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems (TOCS)* 10, 1 (1992), 26–52.
[57] 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 29th Asia and South Pacific Design Automation Conference (ASP-DAC)*.
[58] Inho Song, Myounghoon Oh, Bryan Suk Joon Kim, Seehwan Yoo, Jaedong Lee, and Jongmoo Choi. 2023. ConfZNS: A Novel Emulator for Exploring Design Space of ZNS SSDs. In *Proceedings of the 16th ACM International Systems and Storage Conference (SYSTOR)*.
[59] Viraj Thakkar, Madhumitha Sukumar, Jiaxin Dai, Kaushiki Singh, and Zhichao Cao. 2024. Can Modern LLMs Tune and Configure LSM-based Key-Value Stores?. In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)*. 116–123.
[60] Ruihong Wang, Jianguo Wang, Prishita Kadam, M Tamer Özsu, and Walid G Aref. 2023. dLSM: An LSM-Based Index for Memory Disaggregation. In *Proceedings of the 39th International Conference on Data Engineering (ICDE)*. IEEE.
[61] Hua Yan, Yong Huang, Xinzhi Zhou, and Yinjie Lei. 2018. An Efficient and Non-Time-Sensitive File-Aware Garbage Collection Algorithm for NAND Flash-Based

Consumer Electronics. *IEEE Transactions on Consumer Electronics* 65, 1 (2018), 73–79.

[62] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *Proceedings of the 2020 USENIX Annual Technical Conference (ATC)*.