

Eliminating the Hidden Cost of Zone Management in ZNS SSDs

Teona Bagashvili
Boston University
teona@bu.edu

Tarikul Islam Papon
UMass Boston
t.papon@umb.edu

Subhadeep Sarkar
Brandeis University
subhadeep@brandeis.edu

Manos Athanassoulis
Boston University
mathan@bu.edu

ABSTRACT

Zoned Namespace (ZNS) SSDs offer a new storage model that allows for high-throughput and low-latency storage by eliminating device-side garbage collection. The ZNS interface exposes storage as *append-only zones*, thus requiring host applications (e.g., database systems) to append, read, and garbage collect their pages. However, the ZNS storage abstraction hides the substantial differences across ZNS SSD controller designs. These differences affect both the performance and predictability of host applications. We find that existing ZNS designs exhibit (a) increased *device-level write amplification* (DLWA), (b) increased *wear*, and (c) increased *interference with host I/O*. We identify that (i) zone allocation granularity, (ii) zone geometry, (iii) write order, and (iv) zone mapping and management are the four main causes behind this.

To provide a predictable storage device, we propose *SilentZNS*, a new holistic zone management approach that expands the design space of zones and allocates blocks to zones on the fly, while minimizing wear, maintaining parallelism, and avoiding superfluous writes to the device. SilentZNS is a flexible zone allocation scheme that departs from traditional logical-to-physical zone mapping and allows arbitrary collections of blocks to be assigned to a zone. SilentZNS uses wear-aware allocation and maintains competitive read latency, while substantially reducing DLWA. We implement SilentZNS using the state-of-the-art ConfZNS++ emulator and evaluate it on synthetic microbenchmarks and a key-value storage engine. We show that SilentZNS reduces DLWA by 92% at a 10% zone occupancy threshold, reduces wear by 12%, and achieves up to 3.7× faster workload execution.

PVLDB Reference Format:
Teona Bagashvili, Tarikul Islam Papon, Subhadeep Sarkar, and Manos Athanassoulis. Eliminating the Hidden Cost of Zone Management in ZNS SSDs. PVLDB, 20(2): 270 - 283, 2026.
doi:10.14778/3849423.3849439

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/BU-DiSC/SilentZNS-bench.

1 INTRODUCTION

SSDs are Everywhere. Solid-state drives (SSDs) have become the standard for persistent storage media, offering low-latency access and high bandwidth. Traditional SSDs offer a block-based API so that they can be a drop-in replacement for traditional hard drives, which creates the expectation that performance should be as predictable as it is for hard drives. However, SSD performance and endurance can vary substantially with different access patterns and even across devices with similar headline specifications. This variability arises from differences in internal architecture, controller policies, and garbage collection strategies [17, 25]. Such workload- and device-dependent behavior has long complicated the design and optimization of data-intensive systems [9, 17, 18, 24].

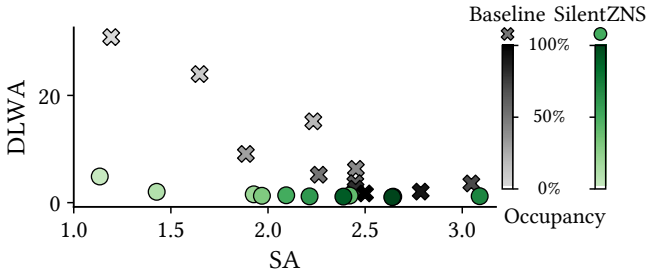


Figure 1: While state-of-the-art ZNS approaches reduce DLWA at the expense of space amplification (SA), SilentZNS removes the DLWA penalty for getting low SA. This enables data systems on ZNS to reclaim zones according to logical lifetime and free-space needs, rather than delaying reclamation solely to avoid hidden device-side writes.

Block API Causes Garbage Collection. A major source of this variability is the mismatch between the block API and NAND flash. The block interface exposes in-place updates, whereas NAND supports page-level reads and writes but block-level erasures, requiring updates to be written out of place [12, 30]. SSDs bridge this gap with a flash translation layer (FTL), which remaps updates, invalidates old pages, and reclaims space through garbage collection. Garbage collection copies valid pages from partially invalid blocks before erasure, causing device-level write amplification (DLWA), performance loss, wear, and higher tail latency under update-heavy workloads [6]. Because the FTL hides physical placement, prior work on block-based SSDs largely focused on avoiding random writes and exploiting efficient random reads, with limited control over device-level data placement [2–5, 7, 9, 10, 20, 21, 23, 28, 35, 40, 41].

ZNS: Exposing Flash-Aware Semantics to the Host. Zoned Namespace (ZNS) SSDs replace the block interface with large, sequentially written *zones* that expose data placement and reclamation to the host. As Figure 2 illustrates, each zone has a write pointer that advances with every write and must be reset before it can be overwritten [15]. This sequential-write model largely eliminates the need for complex FTL mapping and device-side garbage collection [6]. Instead, the host manages zone reuse through `RESET` and `FINISH`. `RESET` returns the write pointer to the beginning of a zone and marks its previously written blocks for erasure [14]. `FINISH` marks a zone read-only and releases the controller resources associated with it [15]. Prior research showed that an LSM-based

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 20, No. 2 ISSN 2150-8097.
doi:10.14778/3849423.3849439

key-value store running on top of a ZNS SSD can achieve 4× lower random-read latency and 2× higher throughput by using the zoned interface compared to a block-interface SSD [6].

Challenges with Zone Management. While the ZNS interface offers benefits, it also introduces new overheads. When the host issues a `FINISH` command for a zone that is only partially filled, to satisfy device-level constraints, the baseline ZNS design fills the rest of the zone with dummy data (device-issued writes) before sealing it [14]. This leads to several problems in current ZNS SSDs:

1. Unnecessary Write Amplification. While the dummy writes are invisible to the host, they cause **significant device-level write amplification (DLWA)**, i.e., a substantial increase in internal writes due to padding, and not due to workload demand [14]. Partially used zones that were filled with dummy data must be reclaimed entirely before reuse. This results in inefficient block utilization and additional erase operations that could be avoided.

2. Interference with Host I/O. Dummy writes compete with host I/O for the same flash channels and controller resources. Prior studies show that issuing a `FINISH` command concurrently with a write-heavy workload can significantly degrade write throughput as it interferes with host-issued I/Os [14].

3. Managing Complex SA vs. DLWA Tradeoff. ZNS SSDs support a limited number of open and active zones. When this limit is reached, a system such as RocksDB with ZenFS can either `FINISH` a partially written zone to open a new one or continue writing to an already-open zone. The latter avoids dummy writes and reduces DLWA, but may mix data with different lifetimes. Long-lived data can then delay reclamation of the entire zone, increasing space amplification (SA). Figure 1 illustrates this tradeoff for `FINISH` thresholds ranging from 1% to 100% zone occupancy. We compute DLWA as the ratio of total device writes to host writes and use SA to quantify the space overhead of invalidated data retained by the system, as defined in Section 6. Increasing the occupancy threshold from 10% to 90% reduces RocksDB's DLWA by approximately 91%, but increases SA by 69%. Thus, the current ZNS design forces applications to trade off DLWA against SA.

Implications for Data Management. Dummy writes consume bandwidth that would otherwise serve application requests, thereby increasing flash wear and introducing latency variability that directly affects operations like WAL appends, bufferpool flushing, and LSM compaction. Current ZNS-backed systems often compensate by delaying `FINISH` or by mixing data with different lifetimes within the same zone, which, in turn, increases SA and complicates host-side GC. As a result, core data system decisions such as compaction, file placement, and log recycling become tightly coupled with (often undocumented) controller behavior rather than being driven purely by logical efficiency [11]. Similar in spirit to SSD-iq [17], our thesis is that storage behavior hidden behind a standard interface can substantially affect data systems. Our approach removes this coupling by making `FINISH` inexpensive enough that data systems can manage zones according to logical data lifetimes rather than device-specific quirks.

The Solution: SilentZNS. To help data systems reason about these costs, we decompose zone management into a four-dimensional design space: (i) *zone allocation granularity*, the smallest storage element finished and reset as a unit; (ii) *zone geometry* (parallelism and size); (iii) *write order*, how writes are distributed across blocks; and (iv) *zone mapping (fixed vs dynamic) and management (*`FINISH` *and* `RESET`*)*. We show that coordinating these four design decisions reduces the dummy-write cost of `FINISH`. Therefore, we introduce SilentZNS, which writes dummy data only to the minimum required storage elements and releases the rest for future zone allocations. However, fine-grained reclamation introduces new controller-side challenges that require evaluating tradeoffs in write amplification, throughput, wear, and allocation latency. We then translate this evaluation into practical guidance for data systems. On today's ZNS designs, this analysis is already actionable. Data systems can use our guidelines to select a fixed-zone device suited to their workload and choose `FINISH` timing to shift the DLWA–SA tradeoff. Looking ahead, no single configuration is optimal across all workloads. Our results therefore motivate ZNS vendors to expose *bounded*, host-selectable options that let data systems choose configurations without incurring the current cost of `FINISH`.

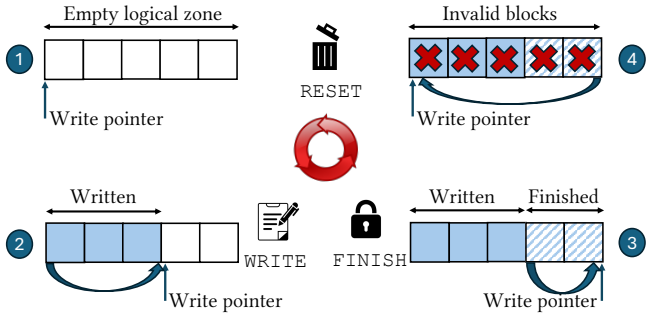


Figure 2: Baseline policy. (1) A zone begins in the empty state. (2) Writes advance the write pointer, (3) `FINISH` moves the pointer to the end and fills the remaining space with dummy data, and (4) `RESET` invalidates all data in the zone.

Contributions. Our main contributions are as follows.

- We identify the causes of increased device-level write amplification, increased wear, and unpredictable performance as suboptimal combinations of zone allocation granularity, zone geometry, write order, and zone mapping and management strategy.
- We define a spectrum of mapping strategies where the SSD uses a set of allocation units with different granularity.
- We propose SilentZNS, which assigns allocation units to zones on demand and writes dummy data only to partially written units.
- We implement SilentZNS in the state-of-the-art ZNS SSD emulator ConfZNS++ [14]. Our implementation spans the full design space and also serves as a tool for data systems engineers to explore these tradeoffs before deployment.
- We evaluate SilentZNS using synthetic and real-world workloads, including RocksDB with ZenFS, and demonstrate how SilentZNS reduces DLWA by 92% on KVBench-II at a 10% zone occupancy threshold, while keeping SA at 1.5 and providing 3.7× faster workload execution, thereby simplifying host-side lifetime management for data systems.
- We provide practical guidelines for data systems engineers to select a configuration for a given workload.

2 ZONED INTERFACE STANDARD

We now present the necessary background on the ZNS standard.

Traditional SSDs consist of multiple chips connected to a controller via channels [1, 30]. As shown in Figure 3, each chip includes dies, planes, blocks, and pages, enabling high parallelism [12, 31]. Concurrent I/O is essential to harness bandwidth [29, 32, 33, 43]. Data is written/read at the *page* level but erased at the *block* level. Due to flash's *erase-before-write* constraint, SSDs perform out-of-place updates, invalidating old pages. This creates blocks with valid and stale data [39], requiring *garbage collection* [8, 19] to reclaim space, causing device-level write amplification, increased write latency, and accelerated wear of flash cells [1, 30].

ZNS SSDs. To address these limitations, the zoned storage interface organizes logical block addresses (LBAs) into zones, bringing the storage abstraction closer to the organization of the device [6]. Every zone has a write pointer that allows the host to write pages sequentially. The ZNS standard also provides the `APPEND` command, which abstracts away the write pointer by allowing the host to specify only the target zone for the write operation. Devices limit the number of zones that can be active at any given time [13]. The sequential write requirement eliminates device-side garbage collection, as the controller no longer needs to handle out-of-place updates. Instead, the host assumes the responsibility for both data placement and garbage collection by managing the zone state [6], through two new ZNS commands for zone management: `RESET` and `FINISH`.

`RESET`. When all data in a zone becomes invalid, the host can reclaim it via the `RESET` command that moves the write pointer to the zone's beginning [14]. The NVMe ZNS standard does not mandate when or how the data must be erased upon a `RESET`. Devices choose based on: (i) `RESET` Timing – either *synchronous* (erase immediately) or *asynchronous* (invalidate metadata and defer erasure); and (ii) `RESET` Granularity – *full* (erase all blocks regardless of content) or *partial* (erase only blocks with invalid data). While full `RESET`s are simple to design and maintain leveled wear across all the erase blocks within the zone, they cause unnecessary wear and delay subsequent writes, as the full zone must be erased. Partial `RESET`s reduce wear and latency, since reset only applies to the blocks with invalid data [14].

`FINISH`. A zone transitions into the *Full* state when it is completely filled with data. However, the host may also mark the zone as *full* using the `FINISH` command, moving the zone's write pointer to the end and preventing any further writes [14]. Issuing `FINISH` also indicates to the controller to release the resources associated with that zone, allowing a new zone to be opened [13]. This is useful when the host has reached the limit of open zones but needs to open a new one. In the baseline design, when a partially written zone is finished, the controller fills the remaining space with dummy writes to satisfy device-specific constraints [14]. Prior work also proposes using the unwritten pages as a read cache instead of filling them with dummy data. However, this requires maintaining backup mappings and additional controller-side scheduling logic, increasing controller complexity [44].

Zone Mapping. The SSD controller handles logical zone to physical zone mapping: (i) *static mapping* assigns a fixed physical zone to each logical zone, offering control over placement but shifts the burden of wear-leveling to the host [14], while (ii) *dynamic mapping* allocates physical zones on demand. If a free physical zone is available, the controller maps it directly; otherwise, it finds an invalidated (already reset) zone and erases it before reallocating. This allows the controller to manage wear and amortize *reset latency* [14], an important aspect of zone management, as we also see in our experimentation.

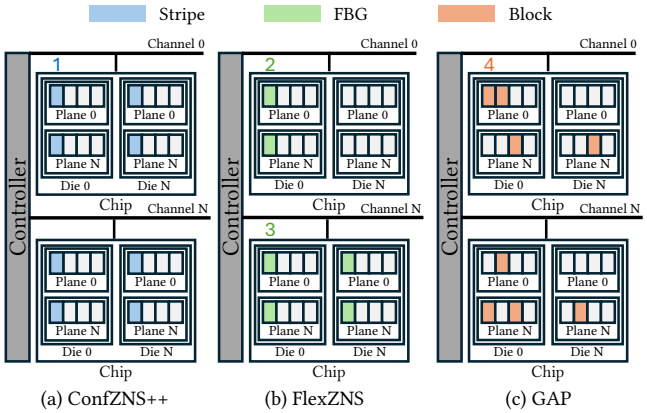
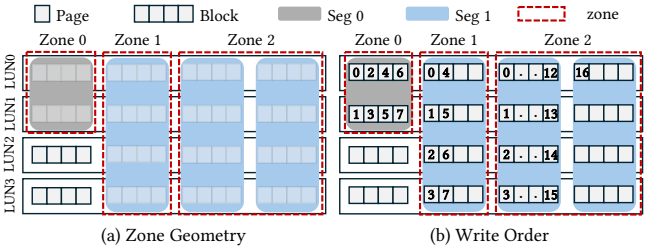


Figure 3: ConfZNS++ constructs zones from superblocks, each spanning all planes. FlexZNS organizes erase blocks into Flash Block Groups (FBG) spanning a fixed number of dies. GAP treats each chip as a pool of erase blocks and dynamically reconstructs zones from these blocks.

Due to the hierarchical structure of SSDs, physical zones can be (i) mapped to a fixed set of erase blocks or (ii) dynamically constructed. ConfZNS++ SSD emulator constructs a physical zone from a fixed set of superblocks where a superblock is composed of erase blocks at the same offset across all planes, as demonstrated in Figure 3(a) [14]. FlexZNS organizes storage into Flash Block Groups (FBGs), where each FBG is composed of erase blocks at the same offset across a fixed number of dies, as shown by units 2 & 3 in Figure 3(b) [42]. GAP treats chip as a pool of erase blocks and constructs the physical zone dynamically as shown in Figure 3(c) [27]. How physical zones are mapped across the hierarchy of parallel units (chips, dies, planes) introduces different tradeoffs in terms of parallelism, DLWA, and zone allocation overhead, which we analyze in Section 3.

3 ZNS HIDDEN WRITE AMPLIFICATION

Next, we examine the hidden write amplification caused by `FINISH`. Prior work established that finishing a partially programmed zone generates dummy writes that interfere with host I/O [14]. However, the existence of dummy writes alone does not determine their cost. We show that the volume of dummy writes depends on the interaction of four controller-side zone-management decisions: (i) *zone geometry*, (ii) *write order*, (iii) *physical zone allocation* and (iv) *zone reclamation*. Because these device-level choices directly affect host-visible costs such as SA and throughput, data systems cannot remain agnostic to how zones are constructed and managed. We therefore establish a common framework for interpreting ZNS configurations and their tradeoffs. For simplicity, throughout the remainder of the paper, we abstract the underlying storage hierarchy and refer to each parallel component of the SSD as a logical unit (LUN), where a LUN consists of erase blocks, as shown in Figure 4.



(a) Zone Geometry (b) Write Order

Figure 4: (a) Zone geometry defines the size of a zone and the number of parallel units it spans. (b) Write order controls how data is striped across parallel units. For multi-segment zones, writes fill one segment before advancing to the next.

A zone is constructed from segments, where the segment consists of erase blocks across LUNs. For example, as shown in Figure 4, segment *Seg0* has parallelism of 2 as it spans 2 LUNs.

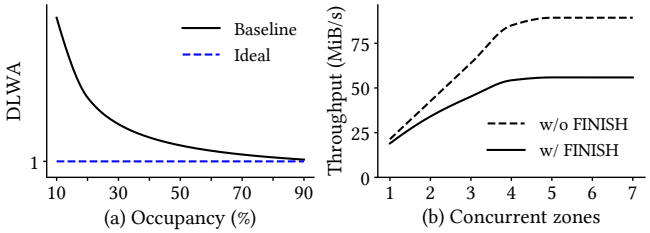
Zone Geometry determines the type of segments used in the zone (i.e., parallelism of a segment) and how many segments are used to construct a zone (i.e., its size). Segment parallelism sets intra-zone parallelism. For example, in Figure 4(a), *Zone0* maps to *Seg0* with parallelism 2, while *Zone1* maps to *Seg1* with parallelism 4. Prior designs often span all parallel units to maximize intra-zone throughput [14]. They also vary zone size by combining multiple segments, as in *Zone2*. Larger zones reduce FTL metadata but increase the number of blocks that may require dummy writes during `FINISH`, increasing dummy writes and interference with host I/O.

Write Order. To exploit SSD parallelism, prior work stripes pages across all parallel units [14]. In Figure 4(b), pages 0–7 of zones 0 and 1 are striped across the LUNs. For multi-segment zones, such as *zone 2*, each segment is completed before the next begins: first segment spans pages 0–15, after which the write pointer advances to page 16 and starts next segment. Although this maximizes intra-zone throughput, it touches many erase blocks, all of which must be padded if partially written at `FINISH`, thereby increasing DLWA.

Physical Zone Allocation. Prior work such as ConfZNS++ maps each zone to a fixed set of superblocks [14]. FlexZNS maps each zone to a smaller die-spanning group of blocks (FBG), letting zones be smaller [42]. The advantage of such zone allocation is reduced metadata overhead, since logical zones can be directly mapped to physical storage units. On the other hand, GAP reconstructs a zone's blocks on demand from a global pool to balance wear [27]. This enables more fine-grained control over zone allocation, but it also requires additional metadata to maintain the mapping table between logical zones and allocated blocks.

Zone Reclamation. Prior work proposes full zone reclamation, where if a zone is partially written and the host issues a zone `FINISH` command, the entire zone is filled with dummy data, and consequently the entire zone is erased during `RESET`. This is tied to the fixed physical zone allocation strategy, where erase blocks from one physical zone cannot be assigned to another zone. The baseline additionally pads the zone's remaining unwritten pages on FINISH; the padded blocks subsequently require erasure before reuse, causing unnecessary writes and wear [14].

Hidden Write Amplification in State-of-the-art Designs. Now we show how the four design decisions above determine the required dummy writes, using ConfZNS++ [14], a state-of-the-art



(a) Occupancy (%) (b) Concurrent zones

Figure 5: (a) DLWA decreases as zone occupancy increases, approaching the ideal baseline at high occupancy. (b) Concurrent `FINISH` operations interfere with host writes in the baseline ZNS configuration.

ZNS SSD emulator configured to model a real ZNS SSD (ZN540). The emulated SSD uses fixed physical zones, follows the large-zone model, and stripes writes across all parallel units [14]. While this design is optimized for intra-zone parallelism and low physical zone allocation time, it increases dummy writes and, thus, DLWA. Figure 5(a) shows that DLWA is significantly higher at low occupancy because finishing a partially written zone forces dummy writes across all erase blocks in the zone. Figure 5(b) shows the interference caused by concurrent `FINISH` operations. We compare baseline throughput with and without concurrent finish operations. For example, at concurrency 2, the baseline (w/o finish) operation measures throughput when two zones are written concurrently by dedicated host threads. In the baseline (w/finish), we measure the host writes to two concurrent zones while the device simultaneously finishes two other pre-filled zones. The figure shows that `FINISH` operations reduce throughput, especially at higher concurrency. Throughput alone hides part of the cost, which shows up as higher latency. With finish, mean host write latency is 1.12× higher at concurrency 1 and 1.6× higher at concurrency 7. Therefore, static zone management designs that maximize intra-zone parallelism through full striping to reduce allocation overhead increase DLWA and I/O interference. We present more details in Section 6.

4 AUGMENTING THE ZNS DESIGN SPACE

The Augmented ZNS Design Space. Reducing DLWA and I/O interference requires minimizing the dummy writes. In the fixed design, zone reclamation and allocation are tightly coupled, leaving no room to avoid the dummy writes. This motivates a broader design space that decouples reclamation from allocation, so that empty blocks can be released and reused. Achieving this requires the controller to construct zones dynamically from finer-grained storage elements, while preserving intra-zone parallelism and wear-leveling, and keeping allocation overhead acceptable. Therefore, in this section, we introduce a new *zone reclamation strategy*, expand the *zone allocation granularity* options, and analyze their tradeoffs. This lets data systems interpret how each configuration affects their workload. Table 1 summarizes the terminology used in this section, and Figure 6(a) illustrates the zone allocation granularity options.

Zone Reclamation. Prior designs fill the remaining empty blocks of a zone with dummy data regardless of the occupancy of a zone, causing DLWA [14]. We introduce a new `FINISH` design in which only the partially written blocks are padded with dummy data and subsequently erased during the `RESET` operation. The remaining empty blocks are released and available for allocation to the next

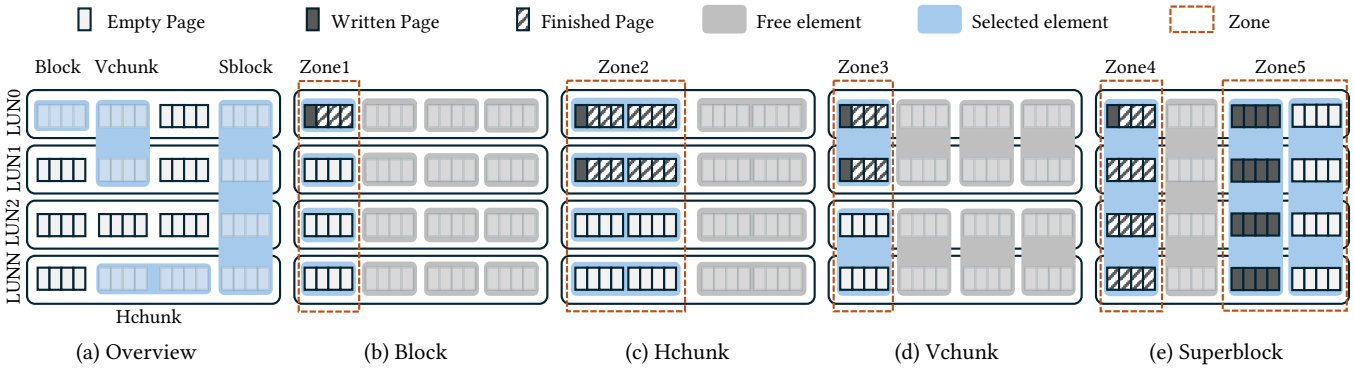


Figure 6: (a) Overview of storage elements. (b) Zone1 is constructed from N storage elements, where each storage element is an erase block. (c) Zone2 is constructed from horizontal chunks (Hchunks), where each Hchunk contains two erase blocks. (d) Zone3 is constructed from vertical chunks (Vchunks), where each Vchunk contains two erase blocks. (e) Zone4 is constructed from a superblock (Sblock), which consists of one erase block from each LUN.

Table 1: Terms to describe the augmented ZNS design space

Term	Description
Block	An SSD erase block
LUN	Parallel unit consisting of multiple blocks
Superblock	Consists of one block per LUN
Vchunk-X	Group of X blocks across adjacent LUNs
Hchunk-X	Sequence of X blocks within a LUN
Storage element	Basic unit used to construct zones
Zone segment	Maximum subset of a zone that fully exploits the zone's LUN parallelism

zone. This is possible by allowing physical zones to be allocated dynamically.

Zone Allocation Granularity. The new zone reclamation strategy introduces an additional dimension in the design space: the zone allocation granularity. Specifically, it defines the smallest physical unit from which a zone is constructed. Prior work varies this granularity. ConfZNS++ maps a zone to a fixed physical zone of superblocks [14]. FlexZNS allows varying zone size by mapping each zone to a FBG that spans a configurable number of dies [42]. GAP keeps a fixed zone size and instead reconstructs a zone's blocks from a global pool using a greedy selection [27]. Neither FlexZNS nor GAP designs or evaluates how `FINISH` operates on these allocation units, whereas SilentZNS makes `FINISH` granularity-aware. Our design space spans from a single erase block to a full superblock. GAP's block allocation unit is at the fine end, and ConfZNS++'s and FlexZNS's are at the coarse end. We also introduce two intermediate elements, the *horizontal* and *vertical chunk*, to cover the middle of the continuum that prior work leaves unexplored. We identify that choosing the size and geometry of the allocation unit involves balancing throughput, DLWA, allocation time, and wear-leveling.

Block. In the augmented design, we model storage as a collection of erase blocks and construct each zone such that blocks are evenly distributed across parallel units as illustrated in Figure 6(b). The primary advantage of block-based allocation is that it allows more fine-grained control over DLWA. During a `FINISH` operation, only the partially written erase blocks need to be written with dummy data. For example, Figure 6(b) shows that only the partially written block from *LUN0* is written with dummy data, while the rest of the empty blocks can be released and allocated to the next zone. The downside is higher zone allocation latency due to searching through erase blocks to construct the zone, along with additional metadata overhead to maintain the mapping between logical zones and allocated erase blocks.

Horizontal Chunk. To reduce the search space and required metadata, we introduce horizontal chunks (Hchunk). As shown in Figure 6(c), a horizontal chunk is a contiguous sequence of erase blocks within a single parallel unit. The main advantage of Hchunk is a smaller search space, which reduces the zone allocation time. However, as illustrated in Figure 6(c), the limitation is that if any page within a chunk is written, all remaining erase blocks in that chunk must be padded with dummy data during `FINISH`, even if they were never used. Therefore, increasing the chunk size may reduce the zone allocation and metadata overhead, but may increase DLWA and interference with host I/O.

Vertical Chunk. Vertical chunks group erase blocks across adjacent LUNs (Fig. 6(d)), with the largest chunk forming a superblock. During `FINISH`, every partially written chunk must be padded, so larger chunks increase DLWA and host-I/O interference. However, vertical chunks better match striped writes than horizontal chunks. In Figure 6, Hchunk and Vchunk each contain two blocks. If one page is written in the first block on each of LUNs 0 and 1, FINISH must complete two Hchunks (four blocks), but only one Vchunk (two blocks).

Superblock. A vertical chunk that spans all parallel units is called a superblock, as shown in Figure 6(e). A superblock consists of one erase block from each parallel unit. Using a superblock as the allocation unit simplifies zone construction, as there is no need to explicitly enforce parallelism constraints. The superblock inherently ensures that all parallel units contribute one block. However, as illustrated in Figure 6(e), the downside is that the superblock has more coarse-grained allocation granularity, and during a `FINISH` operation, all erase blocks in the superblock must be padded with dummy data, even if some were never written.

Fixed physical zone. In this mapping scheme, the physical zone is mapped to a fixed set of erase blocks. The advantage of this direct

mapping is that allocation time and metadata size are kept minimal. The downside is that due to the fixed physical zone design, dynamic zone reclamation is not possible. Therefore, all empty blocks in the zone must be filled with dummy data during `FINISH`, causing interference with host I/O and unnecessary wear, as these blocks must later be erased once they are allocated to a new logical zone.

5 SilentZNS: FLEXIBLE ZONE ALLOCATION

Design Goals. We now show how these configurations can be implemented. Reducing dummy writes requires reclaiming and allocating storage elements at a finer granularity than a fixed physical zone. However, constructing a valid zone is no longer implicit. The device needs to select available storage elements while preserving the zone size, sufficient intra-zone parallelism, and wear-leveling. We design SilentZNS, a fine-grained device-level zone management strategy for ZNS SSDs that represents storage as a collection of *storage elements* with configurable sizes and allocates the storage elements to logical zones on demand. By configuring the granularity of the storage element, SilentZNS navigates the continuum of ZNS design space from fine-grained block-level allocation to coarse-grained superblock allocation. SilentZNS provides a flexible abstraction for zone allocation, introducing tradeoffs in allocation latency and zone management granularity.

From Blocks to Superblocks: The Storage Continuum. The flexible design of SilentZNS allows it to model a continuum of ZNS zone allocation strategies – from a *block*-based allocation that optimizes for device write amplification to a *superblock*-based allocation that optimizes for zone allocation time. SilentZNS allocates zones based on two metadata fields: (i) *wear* – the number of times a storage element has been erased, and (ii) *availability* a, indicating the element's status: $a = 0$ (empty and available for allocation), $a = 1$ (allocated but still empty), $a = 2$ (allocated and contains valid data) and $a = 3$ (free for (re-)allocation but contains invalid data). Allocation is defined by three parameters: the storage-element size e, the minimum zone parallelism L_{min}, and the per-LUN element limit K. The granularity ranges from block ($e = 1$) to superblock ($e = L$). At namespace initialization, the host selects these parameters. The configuration options would be exposed by the device (Section 7). For each allocation, the device computes the eligible LUN set L_{elig}. When $L_{\text{min}} < L$, SilentZNS rotates this set in round-robin order across consecutive zones to reduce inter-zone interference. Table 2 summarizes this notation.

SilentZNS with Block and Hchunk Allocation. Block-based zone allocation in SilentZNS aims to minimize DLWA, while Hchunk aims to reduce zone allocation time. In this formulation, we consider block allocation as a special case of Hchunk allocation, where e denotes the chunk size and $e = 1$. Therefore, we use Hchunk to describe the formulation for the remainder of this subsection. Our full notation is summarized in Table 2. An *Hchunk* is treated as the smallest storage element, and zones are built on the fly using one or more Hchunks per LUN. The Hchunk allocation algorithm selects available Hchunks with the lowest wear, subject to a minimum number of participating LUNs and a maximum number of chunks per LUN. We formulate the objective as:

$$\text{Minimize} \quad \sum_{n=1}^{N} c_n \cdot w_n$$

Table 2: Notation for modeling ZNS SSD

Term	Definition
N	total number of storage elements
L	total number of LUNs
Z	number of storage elements selected for a zone
K	max number of selected chunks from an active LUN
w	storage element wear array of size N
a	storage element availability array of size N
c	chunk selection array of size N, where $c_n \in \{0, 1\}$
e	chunk size in erase blocks
s	LUN activation array of size L, where $s_l \in \{0, 1\}$
g_l	chunk membership vector for LUN l
L_{min}	minimum number of active LUNs required for a zone
L_{elig}	set of LUNs eligible for allocation

where $c_n \in \{0, 1\}$ indicates if Hchunk n is selected ($c_n = 1$) and w_n is the wear associated with Hchunk n. A chunk must satisfy the following constraints to ensure that we use Hchunks with the lowest wear while maintaining parallelism: (i) *chunk availability*, (ii) *chunk selection*, and (iii) *zone parallelism*.

Hchunk availability The availability constraint ensures that only available (i.e., $a_n \in \{0, 3\}$) chunks can be selected. In the implementation, unavailable chunks are excluded by fixing their decision variables to zero:

$$c_n = 0 \quad \text{if } a_n \notin \{0, 3\}, \quad \forall n \in \{1, \ldots, N\} \tag{1}$$

Hchunk selection This constraint ensures that each zone is allocated a fixed number of chunks Z. In our evaluated configurations, each newly allocated zone contains exactly (Z) equal-sized chunks:

$$\sum_{n=1}^{N} c_n = Z \tag{2}$$

Zone parallelism To ensure that the zone meets parallelism requirements, we introduce binary variables $s_l \in \{0, 1\}$, which indicate whether LUN l participates in constructing the zone. $s_l = 1$ means that LUN l is active and contributes at least one chunk, while $s_l = 0$ means it is not active. To ensure sufficient parallelism, at least L_{min} LUNs must contribute to the zone:

$$\sum_{l=1}^{L} s_l \geq L_{\text{min}} \tag{3}$$

To model chunk-to-LUN relationships, we define $g_l \in \{0, 1\}^N$ as the membership of chunks in LUN l, where $g_l[n] = 1$ if chunk n belongs to LUN l. The total number of selected chunks from LUN l is given by $\sum_{n=1}^{N}(c_n \cdot g_l[n])$. We couple chunk selection with LUN participation using the following constraints:

$$\sum_{n=1}^{N}(c_n \cdot g_l[n]) \geq s_l \quad \forall l \in \{1, \ldots, L\} \tag{4}$$

$$\sum_{n=1}^{N}(c_n \cdot g_l[n]) \leq K \cdot s_l \quad \forall l \in \{1, \ldots, L\} \tag{5}$$

Constraint (4) ensures that if a LUN is marked as active, it must contribute at least one chunk. Constraint (5) bounds the number of chunks assigned to each LUN. If $s_l = 0$, the upper bound forces the number of selected chunks to be zero, preventing allocation from inactive LUNs. If $s_l = 1$, the LUN can contribute up to K chunks. Here, K denotes the maximum number of chunks that can be selected from an active LUN. With these constraints, the Hchunk-based mapping strategy enables flexible allocation across LUNs while maintaining bounded parallelism. By tuning L_{min} and K, the

allocator can relax parallelism requirements in favor of improved wear-leveling.

Additionally, since we model zone geometries with varying levels of parallelism, when the desired zone parallelism is smaller than the total number of LUNs L, we select LUNs in a round-robin manner for each zone allocation. This approach reduces inter-zone interference across consecutive allocations. If zones were allocated on the same set of LUNs, they would compete for the same write resources during host-issued writes. Thus, for each allocation, we identify a set of eligible LUNs, and LUNs outside the eligible set L_{elig} are disabled:

$$s_l = 0 \quad \forall l \notin L_{\text{elig}} \tag{6}$$

SilentZNS with Vchunk Allocation. Vchunk-based zone allocation in SilentZNS aligns allocation with striped write order while reducing metadata overhead and allocation time. A Vchunk groups storage elements across e consecutive LUNs, where $1 \leq e \leq L$ and e must evenly divide L. When $e = 2$, each Vchunk consists of 2 blocks that span 2 LUNs. Vchunk allocation does not change the physical number of LUNs. Instead, it just groups e LUNs into a single logical group. Thus, the allocator operates over $L_{\text{grp}} = L/e$ logical groups. Therefore, Vchunk allocation uses the same formulation as Hchunk allocation, with the only change being the definition of the storage element. Each decision variable c_n now corresponds to selecting a Vchunk, and the allocator operates over L_{grp} logical groups instead of L LUNs. Varying e changes the Vchunk size. Smaller Vchunks provide fine-grained DLWA control but require more metadata, whereas larger Vchunks reduce metadata and allocation time at the cost of higher DLWA because more unwritten pages must be padded during `FINISH`. At e=L, a Vchunk spans all LUNs and becomes a superblock, maximizing intra-zone parallelism.

SilentZNS with Superblock Allocation. Superblock-based zone allocation in SilentZNS is optimized for low zone allocation time and less metadata. A *superblock* is a fixed set of erase blocks across all LUNs and is treated as the smallest storage element. As shown in Figure 6(e), superblock-based allocation constructs zones from superblocks, each with one block per LUN. Superblock abstraction inherently enforces parallelism across all LUNs, removing the need for an explicit parallelism constraint and, therefore, simplifying the search space. The objective of the superblock allocation algorithm is to minimize wear by selecting superblocks with the lowest wear. We formulate the objective as follows:

$$\text{Minimize} \quad \sum_{n=1}^{N} s_n \cdot w_n$$

where $s_n \in \{0, 1\}$ indicates whether superblock n is selected and w_n is the wear associated with superblock n. The selected superblocks must satisfy the *superblock availability* and *superblock selection* constraints:

Superblock availability ensures that only available units ($a_n = 0$ or $a_n = 3$) can be selected for the zone.

$$s_n \leq [a_n = 0 \vee a_n = 3] \quad \forall n \in \{1, \ldots, N\} \tag{7}$$

Superblock selection This constraint ensures that each zone is allocated exactly Z superblocks. Since every zone must maintain a fixed size, the total number of selected superblocks is equal to Z, i.e.,

$$\sum_{n=1}^{N} s_n = Z \tag{8}$$

The main advantage of superblock-based allocation over chunk-based allocation is that it inherently satisfies parallelism constraints due to the definition of superblocks. However, the tradeoff is reduced flexibility. Because a superblock spans across all LUNs and is treated as a single storage element, even a single page write requires the entire superblock to be written with dummy data during the `FINISH` command. As a result, `FINISH` operates at a coarser granularity.

Integration with SSD Emulator. We use MOSEK to implement SilentZNS and integrate it into the `Zone Allocator`, `READ`, `WRITE`, `FINISH`, and `RESET` modules in the `ConfZNS++` SSD emulator.

`Zone Allocator.` The allocator maintains a mapping table between zones and their storage elements. When a zone receives its first write, SilentZNS allocates storage elements, marks them as allocated ($a = 1$), and adds them to the table.

`READ/WRITE.` Once SilentZNS receives an I/O request, SilentZNS computes the zone index from the LBA and retrieves its storage elements from the table. LBAs are always striped sequentially across elements. For writes, the metadata is updated to valid ($a = 2$).

`FINISH.` With on-demand allocation, `FINISH` is applied at the storage element level (Fig. 6). Dummy writes are issued only to partially written elements. After `FINISH`, SilentZNS releases unused storage elements ($a = 1$) from the mapping table and marks them free ($a = 0$), making them available for future zone allocations, while written ones ($a = 2$) remain mapped to support reads. Though DLWA is reduced, some dummy writes may still occur based on mapping granularity.

`RESET.` We support partial, asynchronous `RESET` as in ConfZNS++ and ZN540. On `RESET`, written elements ($a = 2$) are marked invalid ($a = 3$), indicating that they need to be erased first before reuse. As for empty storage elements ($a = 1$), they are marked as free ($a = 0$) so they can be reused. After updating the metadata, SilentZNS releases these storage elements from the mapping table. On reallocation, invalid elements ($a = 3$) are erased before reuse; empty elements ($a = 0$) require no erasure.

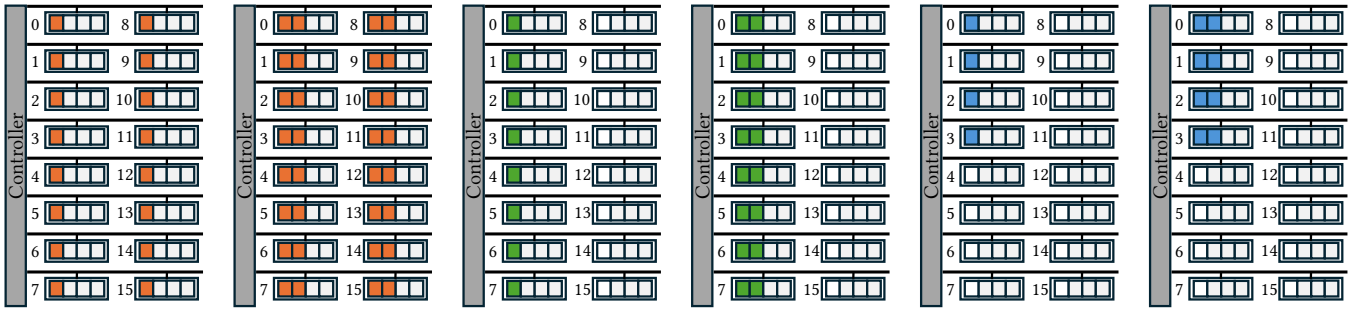
Discussion. In a production implementation, mapping updates can be persisted using existing checkpointing and logging strategies [34]. To mask allocation latency, the controller could also buffer a small pool of zone mappings during idle periods for future allocations.

6 EVALUATION

Now we evaluate the ZNS design space and provide guidelines for the host to navigate the variety of ZNS SSD configurations.

6.1 Evaluation Methodology

SSD Emulator. The SSD emulator used in this work, ConfZNS++, is built on top of FEMU. FEMU enables fine-grained control over internal device parameters such as channels, chips, dies, and planes, as well as configurable page-level read, program, and erase latencies. From the host's perspective, a FEMU-based device behaves like a standard NVMe SSD, allowing applications to interact with the emulated device as if it were real hardware [26].



(a) P=16, S=128 (b) P=16, S=256 (c) P=8, S=64 (d) P=8, S=128 (e) P=4, S=32 (f) P=4, S=64

Figure 7: Zone geometries (zone parallelism P, size S) on an SSD with 16 parallel units. Matching colors indicate equal intra-zone parallelism. (a) 128 MiB: one segment spanning 16 units. (b) 256 MiB: two 16-unit segments. (c) 64 MiB: one 8-unit segment. (d) 128 MiB: two 8-unit segments. (e) 32 MiB: one 4-unit segment. (f) 64 MiB: two 4-unit segments.

Emulated SSD Configuration. Throughout this paper, we use two SSD configurations: (1) a model of a real ZNS SSD (ZN540) and (2) a custom SSD design to explore a broader range of zone geometries and allocation strategies.

Baseline ZNS SSD. We use the ConfZNS++ emulator to model a Western Digital ZN540 ZNS SSD. The device has 4 channels, a page size of 16 KiB, and a block size of 768 pages. It provides a zone capacity of approximately 1 GiB, with a total of 48 zones and a limit of 14 open and active zones. The SSD supports intra-zone parallelism of 4 and simulates latencies of 700 µs for writes, 60 µs for reads, and 3.5 ms for erases [14]. It follows a fixed physical zone design where each physical zone consists of 22 superblocks and each superblock contains 4 erase blocks.

Custom SSD Configuration. To explore different zone geometries and allocation strategies, we also use a custom SSD model. This SSD has 8 channels and 2 ways, resulting in a total of 16 LUNs. It uses a page size of 4 KiB and a block size of 2048 pages. The device simulates latencies of 500 μs for writes, 50 μs for reads, 25 μs for channel transfer, and 5 ms for erases [42]. The total number of zones and zone capacity depend on the zone geometry configuration.

Zone Geometry Configurations. Figure 7 shows six zone geometries on our custom SSD, where boxes 0–15 represent LUNs containing multiple erase blocks. P denotes zone parallelism, or the number of LUNs contributing to a zone segment, and S denotes the zone size in MiB. Figures 7(a), (c), and (e) construct 128, 64, and 32 MiB zones from one segment spanning 16, 8, and 4 LUNs, respectively. Figures 7(b), (d), and (f) use two segments doubling the zone sizes to 256, 128, and 64 MiB.

Zone Storage Elements. Using our custom SSD configuration, we evaluate six allocation granularities: *fixed*, *superblock*, *block*, *Vchunk-2*, *Vchunk-4*, and *Hchunk-2*. The *fixed* baseline, as in ConfZNS++, maps each physical zone to a fixed set of erase blocks. A *superblock* contains one block from each of the 16 LUNs, yielding 128 MiB and 128 superblocks in total. *Block*-allocation uses one erase block per allocation unit. Vertical-chunk sizes must divide the 16 LUNs, where size 16 is equivalent to a superblock. We also evaluate finer *Vchunk-2* and *Vchunk-4* units. *Hchunk-2* groups two consecutive blocks within one LUN. Together, these configurations form a continuum of storage elements that SilentZNS can represent.

Workloads. We evaluate with microbenchmarks and KVBench [45]. Our workloads are:

DLWA. This benchmark quantifies the dummy-write cost of `FINISH` as a function of zone occupancy. We fill a zone to a target occupancy, issue `FINISH`, and record the number of pages written to complete the zone.

Interference. We measure the impact of concurrent `FINISH` on host I/O. A `FINISH` concurrency level of N indicates that N partially filled zones are being finished while the host writes to N other zones concurrently.

Raw-device Write. To evaluate intra-zone and inter-zone parallelism, we use a FIO-based sequential write workload where we vary the request size within a zone and the number of concurrently written zones. Unless otherwise stated, FIO jobs use requests with direct, synchronous I/O, writing to a dedicated zone.

KVBench-II. The microbenchmarks isolate individual metrics. To test end-to-end performance and the DLWA–SA tradeoff, we use KVBench-II [45], a recent mixed KV benchmark that exercises a realistic workload. This mixed workload interleaves 50% inserts, 10% deletes, 15% point queries, and 25% updates [45]. We run KVBench on RocksDB using ZenFS as the filesystem backend.

RocksDB with ZenFS. ZenFS integrates with RocksDB through a zone-based interface, placing files according to write-lifetime hints and allocating a new zone when no suitable one exists. When open or active zone limits are reached, it may issue `FINISH` based on a configurable threshold, which we report as occupancy (e.g., 10% remaining space corresponds to 90% occupancy). If no open zone satisfies the threshold, ZenFS delays `FINISH` by relaxing lifetime matching and placing incoming files into existing open zones, potentially mixing lifetimes. Since a zone can be reset only after all its data becomes invalid, long-lived data may delay reclamation and increase space amplification.

Evaluation Metrics. We measure (i) *device-level write amplification*, (ii) *space amplification*, (iii) *wear*, and (iv) *interference*.

Device-level write amplification quantifies the additional writes introduced by the device: DLWA $= (W_h + W_d)/W_h$, where W_h is host-issued writes and W_d is device-issued writes.

Space amplification measures retained invalidated data relative to the workload's total inserted payload: SA $= (I + W_i)/I$, where I is

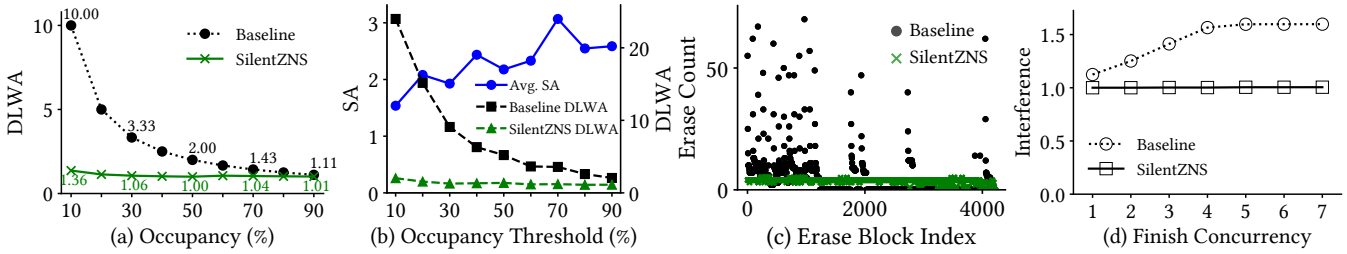


Figure 8: (a) SilentZNS's DLWA benefits are highest at low zone occupancy. (b) A lower occupancy threshold reduces SA, but increases DLWA for the baseline, while SilentZNS maintains low DLWA. (c) SilentZNS distributes wear more evenly. (d) SilentZNS reduces interference with host writes.

the total logical size of inserts, and W_i is the average amount of invalidated data retained over the experiment.

Wear measures how many times each block is erased. SilentZNS tracks wear at the storage element level. All blocks of a storage element have the same wear due to `FINISH` and `RESET` granularity.

Interference is defined as the ratio of baseline throughput (no concurrent `FINISH` operations) to the throughput observed during concurrent `FINISH` operations.

6.2 SilentZNS Wins Across the Board

We compare SilentZNS against a baseline SSD modeled after ZN540 using the ConfZNS++ emulator. For SilentZNS, we only use the configuration where the segment is a superblock. This is because the device has four parallel units and striped writes naturally span all units, resulting in minimal differences across SilentZNS storage elements. Although we later evaluate fine-grained storage elements using a custom SSD configuration, we first focus on highlighting the benefits of dynamic zone reclamation for the baseline SSD.

SilentZNS Largely Eliminates DLWA. In our first experiment, we use the *DLWA benchmark* to evaluate DLWA by filling the zone to an occupancy between 10% and 90% before issuing the `FINISH` command. Figure 8(a) demonstrates that DLWA is consistently lower for SilentZNS, reducing DLWA by up to 86.36% (10% zone occupancy with the *superblock* configuration). This is due to SilentZNS writing dummy data only to partially written storage elements while releasing unused ones, thereby avoiding full-zone dummy writes. In SilentZNS, DLWA depends on the position of the write pointer. If all erase blocks within a segment are fully written, no dummy writes are required. Consequently, the maximum number of dummy writes is bounded by the segment size, since the host writes one entire segment before moving to the next. As a result, at 50% zone occupancy, approximately half of the segments are fully written with host data, allowing SilentZNS to achieve $DLWA = 1$. In contrast, the baseline approach exhibits DLWA that is strongly correlated with zone occupancy. It incurs high DLWA at low occupancy due to full-zone dummy writes, whereas SilentZNS maintains low DLWA. At higher occupancy levels, the baseline is competitive as fewer pages remain unwritten. However, as we show next, delaying `FINISH` introduces additional costs.

Reducing SA without Adding DLWA. In this experiment, we run KVBench-II (4 million total operations with 512B entry size) on RocksDB with ZenFS as the filesystem backend. We vary the occupancy threshold between 10% and 90%. As Figure 8(b) shows, increasing the threshold and therefore delaying finish increases space amplification, since ZenFS has to relax file lifetime matching requirements. SA is about 1.5 at a 10% threshold and 2.6 at 90%.

In this experiment, we report the average SA between baseline and SilentZNS, as SA is determined by host-side placement (the two differ by at most 15%, with no trend). Therefore, LSM-users can improve SA by lowering the occupancy threshold. However, this improvement comes at a cost for the baseline SSD. As shown in Figure 8(b), frequent `FINISH` commands at low thresholds (which fully write physical zones) result in large amounts of device-side dummy writes. In contrast, SilentZNS consistently keeps dummy writes low through efficient zone management. At a 10% occupancy threshold, SilentZNS has 92% less DLWA than the baseline.

Reducing the Unnecessary Wear. We evaluate erase block wear under SilentZNS and baseline using KVBench-II on RocksDB with ZenFS at an occupancy threshold of 10%. To capture the cumulative wear, we repeat KVBench-II 8 × due to memory constraints of the SSD emulator. Figure 8(c) shows that SilentZNS reduces unnecessary wear by writing only required storage elements during `FINISH`, whereas the baseline fills entire zones with dummy writes. Specifically, our superblock allocation has a total erase count of 15340, compared to 17344 for the baseline. As Figure 8(c) shows, SilentZNS also achieves better wear-leveling, as erase counts are more evenly distributed across the erase blocks. This is expected, since the baseline ignores wear and allocates the first available physical zone, while SilentZNS enforces wear-aware allocation.

SilentZNS Reduces Interference with Host I/O. We now highlight the benefits of SilentZNS for host-issued writes with the *Interference Benchmark*. In this experiment, we first fill the zones up to 40% occupancy. Then we run the *interference* benchmark with concurrency level ranging from 1 to 7. As shown in Figure 8(d), SilentZNS consistently maintains less interference than the baseline, with an interference factor of ~1. The baseline experiences interference as high as 1.6 after 4 threads due to contention between host writes and device-issued dummy writes. Additionally, in our KVBench-II experiment, we observe up to 3.7× faster workload execution with SilentZNS (10% occupancy threshold), as it eliminates unnecessary writes and reduces interference with host I/O.

6.3 ZNS Design Space Evaluation

In this section, we use our custom SSD to study how broader ZNS design choices, including zone geometry and storage-element granularity, affect data systems through tradeoffs among DLWA, SA, throughput, and zone-allocation overhead.

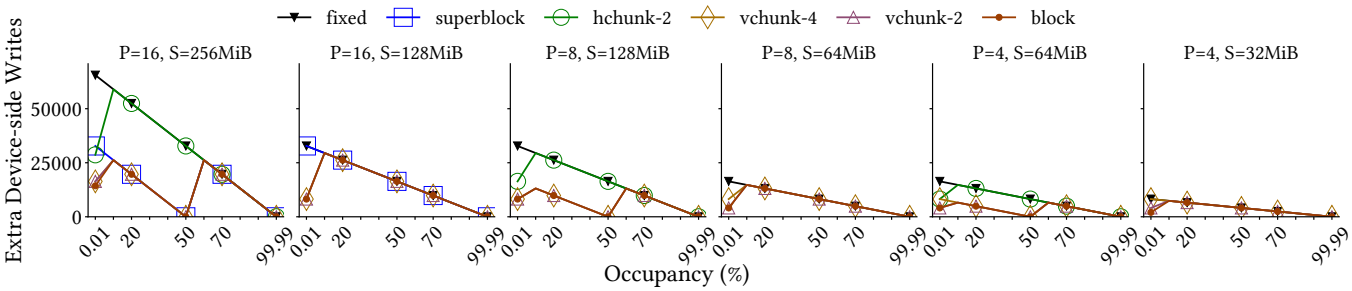


Figure 9: Pages finished across different zone geometry configurations (zone parallelism P and zone size S). Fine-grained allocation schemes (e.g., block and Vchunk) reduce the number of pages that must be finished, especially at lower occupancy.

Unnecessary Page Writes Reduce with Small Zones. In this experiment, we vary the zone size and use the *DLWA Benchmark* to fill each zone to occupancy levels between 0.01% and 99.99% before issuing `FINISH`. Figure 9 shows that for the baseline *fixed* zone, the number of dummy pages written consistently decreases as the zone size becomes smaller. This is because under baseline fixed allocation, `FINISH` applies to the entire zone regardless of how much data has been written. As a result, larger zones incur substantially more unnecessary writes, while smaller zones limit the amount of unwritten space and therefore reduce the need for dummy writes. This effect is most pronounced at low occupancy. For example, at 0.01% occupancy, reducing the zone size from 256 MiB to 128 MiB reduces the number of dummy writes by ~2×, and further reducing zone size to 64 MiB and 32 MiB reduces dummy writes 4× and 8×, respectively. This trend persists across all occupancy levels, with the largest improvements observed at 0.01% and 10%, where the fraction of unwritten space is highest. As occupancy increases, the benefit of smaller zones diminishes. For instance, at 90% occupancy, the number of dummy writes is already low across all configurations, and at 99.99% occupancy it approaches zero.

Guideline G1: Prefer smaller zones for workloads that trigger early FINISH (e.g., WAL, OLTP logs, or mixed-lifetime data in RocksDB) to reduce dummy writes per FINISH.

SilentZNS Further Reduces Unnecessary Writes. Beyond reducing the zone size, SilentZNS can further reduce DLWA. The benefits depend on the storage element granularity, zone geometry, and the zone occupancy level. Figure 9 shows that SilentZNS with *block*, *Vchunk*, and *superblock* provides the most benefit where a zone spans more than one segment (Fig. 7(b), 7(d), and 7(f)). In these geometry configurations at 50% occupancy, SilentZNS eliminates dummy writes. For example, Zone5 in Figure 6(e) has two superblock-sized segments. At 50% occupancy, host writes complete one segment, while the untouched segment is released for future allocations. In SilentZNS, the upper bound of dummy writes depends on the size of the segment (e.g., a superblock for $P = 16$, and 8 erase blocks for $P = 8$). For single-segment zones, SilentZNS generally matches fixed allocation because the write order partially writes every block, requiring `FINISH` to complete the entire segment. At very low occupancy (0.01%), however, finer units still help as *block*, *Vchunk-2*, and *Vchunk-4* reduce dummy writes by ~4×.

Table 3: Interference factor for different zone geometries (parallelism P, size S) and storage elements.

Geometry	fixed	sblock	block	Hchunk2	Vchunk2	Vchunk4
$P4, S32$	1.5	N/A	1.5	N/A	1.5	1.5
$P4, S64$	1.6	N/A	1.1	1.6	1.1	1.1
$P8, S64$	1.5	N/A	1.6	N/A	1.6	1.6
$P8, S128$	1.6	N/A	1.1	1.6	1.1	1.1
$P16, S128$	1.6	1.6	1.6	N/A	1.6	1.6
$P16, S256$	1.6	1.1	1.1	1.6	1.1	1.1

Guideline G2: Use SilentZNS to further reduce DLWA, especially when zones span more than one segment. At very low occupancy, prefer finer-grained allocation (e.g., *block* or *Vchunk*).

Managing DLWA vs. Throughput Tradeoff. Reducing zone size lowers dummy writes but may limit intra-zone parallelism. In this experiment, we use the Write Benchmark and vary the request size from 4 KiB to 128 KiB and the number of concurrent zones from 1 to 32. We identify that if the zone size is reduced while maintaining the same zone parallelism, the system can still achieve peak intra-zone bandwidth, as all parallel units continue to be utilized. For example, in Figure 10, both 256 MiB and 128 MiB configurations with zone parallelism of 16 achieve approximately 110 MiB/s with a single thread and 64 KiB requests, with the same mean request latency of approximately 0.6 ms. In both cases, this peak can be reached with a single thread at 64 KiB request size, without increasing mean request latency. We also identify that if reducing the zone size comes at the cost of reducing zone parallelism, intra-zone parallelism is diminished, which directly impacts bandwidth. As shown in Figure 10, configurations with lower parallelism require more concurrent zones to saturate the device. For example, with (P=4), a single thread achieves only around 29 MiB/s with 16 KiB request size without increasing the request latency. With 16 KiB requests, this configuration reaches approximately 100 MiB/s with 16 threads. In these experiments, we notice that higher concurrency mainly increases the number of in-flight I/Os. CPU utilization remains below 2%, and context switches stay near one per I/O across all the configurations.

Guideline G3: Preserve high zone parallelism for throughput-critical workloads (e.g., large compactions or bulk ingest). Use smaller zones to reduce dummy writes. If this lowers parallelism, increase the number of concurrent zones to maintain throughput.

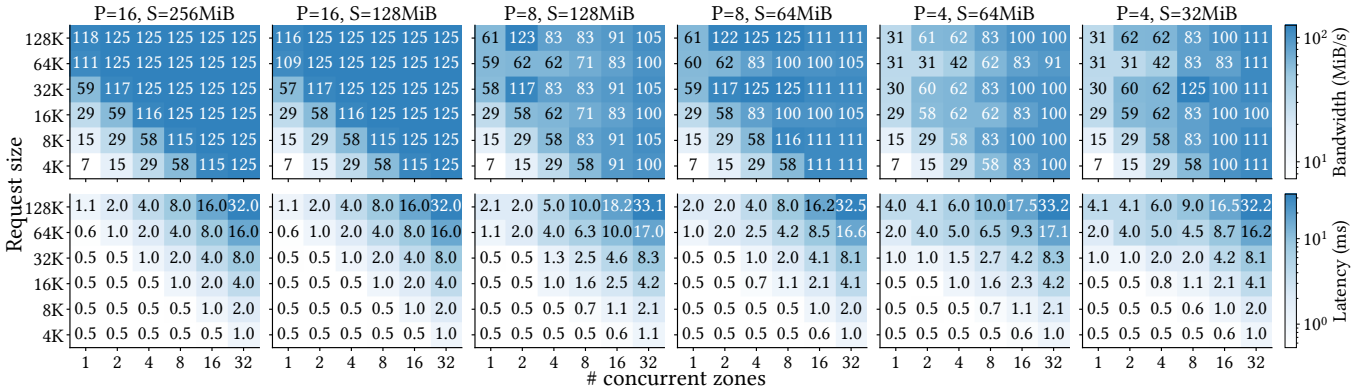


Figure 10: Performance across six zone geometries (parallelism P, size S) under fixed allocation. Reducing zone size while preserving parallelism maintains comparable throughput; lower-parallelism configurations require more concurrent zones.

Table 4: Median zone allocation latency (μs) for different zone geometries (parallelism P, size S) and storage elements.

Geometry	fixed	sblock	block	Hchunk2	Vchunk2	Vchunk4
$P4, S32$	0.5	N/A	7991	N/A	6627	6283
$P4, S64$	0.6	N/A	8247	6649	6424	6026
$P8, S64$	0.7	N/A	7995	N/A	6658	6014
$P8, S128$	0.5	N/A	8475	7348	7183	6108
$P16, S128$	0.7	1660	9030	N/A	6975	6358
$P16, S256$	0.5	1710	9068	7403	7096	6936

Fine-grained Storage Elements Reduce Interference. In this experiment, we run the *Interference* benchmark at concurrency level 8. We identify that the interference benefits depend on the storage-element type and the number of segments in a zone. As shown in Table 3, SilentZNS with fine-grained allocation (*block* and *Vchunk*) provides the most benefit when a zone spans multiple segments (Fig. 7(b)). In these configurations, interference decreases from 1.6 (under fixed and *Hchunk* allocation) to 1.1, as fine-grained SilentZNS configurations (e.g., *Block* and *Vchunk-2*) only write the minimum required storage elements with dummy writes.

Guideline G4: Use SilentZNS to further reduce `FINISH` interference, especially when zones span more than one segment.

Managing DLWA vs. Zone Allocation Tradeoff. In this experiment, we measure the zone allocation time with different storage elements and zone geometry. As shown in Table 4, fixed allocation incurs negligible latency (0.5–0.7 μs). *Block* allocation is most expensive, reaching 9 ms, while *Vchunk-2*, *Vchunk-4*, and *Hchunk-2* reduce the cost to about 6.0–7.4 ms. *Superblock* provides a middle ground, incurring moderate overhead (1.6–1.7 ms). We also evaluate metadata: fixed allocation tracks per-zone availability and mapping indices, requiring $O(N \log N)$ bits for N zones, whereas SilentZNS additionally tracks wear per storage element, requiring $O(U \log U)$ bits for U elements. For a 1 TiB ZNS SSD, metadata ranges from 6–96 KiB for fixed allocation and 40–896 KiB for SilentZNS, decreasing from block (896 KiB) to chunk (160–320 KiB) and superblock (40 KiB) granularity.

Guideline G5: Use finer-grained allocation for DLWA-sensitive workloads, and prefer coarser granularity for workloads where DLWA is less critical (e.g., read-mostly).

SilentZNS Maintains Stable Latency. To evaluate the impact of zone allocation on read and write performance, we run a workload with 4 KiB requests across multiple zones (10 - 40). The baseline fixed allocation and all SilentZNS configurations achieve a median write latency of 528 μs and a 99th-percentile latency within the 553–561 μs histogram range. Sequential-read latency is similarly stable, with a median of 79 μs and a 99th percentile of 81–82 μs. Thus, fine-grained allocation increases zone-allocation time but not steady-state read or write latency. Only the first write to each newly opened zone incurs this cost. We plan to amortize it further by buffering storage elements for subsequent zone allocations.

SilentZNS Unifies the ZNS Design Space. Prior ZNS allocators each explore only a single axis of our design space. GAP dynamically assembles fixed-size zones from erase blocks, whereas FlexZNS maps each size-configurable zone to a fixed FBG; neither evaluates `FINISH`. Therefore, we implement them as GAP++ and FlexZNS++ with our striping, `RESET`, and `FINISH` policies. At P=16, S=128 MiB, and 0.01% occupancy, SilentZNS writes about 8K dummy pages versus 33K for GAP++, since SilentZNS releases empty blocks. At 10% occupancy, both write about 29K pages, as every block is now partially written and none can be released. At P=8, S=64 MiB, and 10% occupancy, FlexZNS++ and SilentZNS each write 15K dummy pages. Thus, SilentZNS matches prior work when finer granularity provides no benefit and reduces `FINISH` cost when empty elements can be released (Fig. 9).

7 A RECOMMENDATION FOR ZNS VENDORS

Navigating Tradeoffs on ZNS Design Space. Our results highlight fundamental tradeoffs in the ZNS design space across DLWA, interference, throughput, and allocation latency. Smaller zones reduce dummy writes per FINISH, especially at low occupancy, but may require higher inter-zone concurrency if intra-zone parallelism is reduced. SilentZNS further reduces unnecessary writes and interference, particularly when zones span multiple segments and at very low occupancy, where finer-grained allocation is most effective. However, these benefits come with higher allocation latency, making coarser storage elements (e.g., *Vchunk* or *superblock*) preferable when balancing DLWA and allocation overhead.

The Need for a Flexible Interface. Overall, there is no one-size-fits-all configuration. Different workloads may require different

Table 5: Practical guidance for Data Systems based on ZNS design tradeoffs.

DBMS scenario	Recommended setting	Host policy	Takeaway
(A) WAL / OLTP logs	`block`/`Vchunk-2`, small zones	Issue early `FINISH`	WAL segments are frequently finished at low occupancy; small zones limit dummy writes per FINISH (G1), and fine-grained elements help further for multi-segment zones or very low occupancy (G2).
(B) LSM flushes / minor compactions	`superblock`/`Vchunk-4`, medium zones	Moderate concurrency	Flushes may leave partially filled regions; `Vchunk-4` and `superblock` balance DLWA and allocation latency (G5) while preserving throughput (G3).
(C) Large compactions / bulk ingest	`superblock`, large zones	Prioritize throughput; maintain high intra-zone parallelism	Throughput dominates; preserving parallelism is critical (G3), and coarse granularity is sufficient since DLWA is low at high occupancy.
(D) Mixed-lifetime ZenFS / RocksDB data	block/`Vchunk-2`, small zones	Issue early `FINISH` to reduce SA	SilentZNS reduces DLWA and `FINISH` interference (G2 & G4), eliminating the need for delaying finish.
(E) Read-mostly workloads	`superblock`/`Vchunk-4`, large zones	Minimize allocation overhead	DLWA is less critical; prefer coarse granularity to reduce allocation latency and metadata (G5).

zone sizes and storage elements. During our experiments, we further observed that at very low occupancy thresholds (e.g., 1%), both the baseline and SilentZNS can fail due to insufficient host-visible address space. This is because frequent `FINISH` operations are triggered at low occupancy. After a zone is finished, the remaining unwritten LBAs in that zone become unusable, preventing the application from utilizing available empty physical capacity. While SilentZNS can reclaim unused physical space, fixed logical zones limit its ability to fully utilize this capacity.

This motivates the need for a more flexible and configurable zoned interface, where the device exposes multiple configuration options, and the host selects an appropriate design at namespace initialization. SilentZNS enables host applications to explore and evaluate these design choices without modifying the underlying device. In this work, we provide metrics and guidelines to help applications reason about the ZNS design space, and to simulate different SSD configurations before deployment.

8 A PRACTICAL GUIDE FOR DATA SYSTEMS

Using the observations and the guidelines from Section 6, we now present a practical guide for data systems engineers that can help make the most of ZNS devices, focusing mostly on LSM Stores [37]. Table 5 summarizes the practical recommendations detailed below. These recommendations extrapolate from our benchmarks; validating each scenario end-to-end is future work.

Use Case 1: Write-Ahead Logging. For database logging, a data system may need to be able to issue an early `FINISH` command to discard unnecessary parts of a log aggressively. This might lead to high DLWA due to the potential low zone occupancy. Our results suggest smaller zones (G1) and, when zones span multiple segments or occupancy is very low, fine-grained elements such as *block* or *Vchunk-2* (G2). For multi-segment zones, this also reduces interference with writes from other parts of the data system.

Use Case 2: LSM Flushes and Partial Compactions. Memtable flushes [16, 22] and partial compactions [38] generate small files that may have similar lifetimes and can occur concurrently. Our results suggest using medium-sized zones that support both inter-zone and intra-zone parallelism (G3), using `superblock` or `Vchunk-4` allocation units.

Use Case 3: Major or Full Compactions. When the files to be written are much larger in size, like in major or full compactions [36, 38], then the primary goal is to maximize ingestion performance, so our results suggest exploiting the full parallelism of the device (G3). Superblock is typically sufficient and simplifies zone allocation.

Use Case 4: Mixing Files with Different Lifetimes. When the workload leads to zones with different lifetime hints, our results suggest setting low occupancy threshold, since SilentZNS keeps DLWA low. Smaller zones with fine-grained storage elements may further reduce DLWA.

Use Case 5: Read-Heavy Workloads. For read-heavy workloads, DLWA is less critical, and we focus on minimizing allocation and metadata overhead. This can be achieved with large zone sizes with coarse allocation granularity (G5).

Overall, our analysis informs data systems and highlights the need for *bounded* flexibility from device vendors, so that data-intensive systems can make the most of the underlying ZNS SSDs.

9 CONCLUSION

ZNS SSDs provide a new append-only interface that eliminates device-side garbage collection and provides stable throughput. However, existing designs suffer from high DLWA, which causes increased wear, interference with host I/O, and a need to balance a complex SA vs. DLWA tradeoff. To address this, we design and evaluate a spectrum of flexible zone allocation strategies that allocate only the necessary storage elements on demand and apply management commands like `RESET` and `FINISH` at the granularity of these elements. We integrate SilentZNS into the ConfZNS++ SSD emulator and show that SilentZNS reduces DLWA by up to 92%, while maintaining SA at 1.5 and achieving a 3.7 × speedup in workload execution. We conclude by sharing recommendations to ZNS SSD vendors and data systems engineers who use such devices.

ACKNOWLEDGMENTS

This work was partially supported by the US National Science Foundation under Grants No. IIS-2144547 and CCF-2403012, a Facebook Faculty Research Award, and a Meta Gift.

REFERENCES

[1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. 57–70. http://research.microsoft.com/pubs/63596/usenix-08-ssd.pdf

[2] Manos Athanassoulis. 2014. *Solid-State Storage and Work Sharing for Efficient Scaleup Data Analytics*. Ph. D. Dissertation. http://infoscience.epfl.ch/record/197103

[3] Manos Athanassoulis and Anastasia Ailamaki. 2014. BF-Tree: Approximate Tree Indexing. *Proceedings of the VLDB Endowment* 7, 14 (2014), 1881–1892. http://www.vldb.org/pvldb/vol7/p1881-athanassoulis.pdf

[4] Manos Athanassoulis, Anastasia Ailamaki, Shimin Chen, Phillip B. Gibbons, and Radu Stoica. 2010. Flash in a DBMS: Where and How? *IEEE Data Engineering Bulletin* 33, 4 (2010), 28–34. http://sites.computer.org/debull/A10dec/athanassoulis.pdf

[5] Manos Athanassoulis, Shimin Chen, Anastasia Ailamaki, Phillip B. Gibbons, and Radu Stoica. 2011. MaSM: Efficient Online Updates in Data Warehouses. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 865–876. http://dl.acm.org/citation.cfm?id=1989323.1989414

[6] 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

[7] Luc Bouganim, Björn Þór Jónsson, and Philippe Bonnet. 2009. uFLIP: Understanding Flash IO Patterns. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*.

[8] Werner Bux and Ilias Iliadis. 2010. Performance of greedy garbage collection in flash-based solid-state drives. *Performance Evaluation* 67, 11 (2010), 1172–1186.

[9] Mustafa Canim, George A. Mihaila, Bishwaranjan Bhattacharjee, Kenneth A. Ross, and Christian A. Lang. 2010. SSD Bufferpool Extensions for Database Systems. *Proceedings of the VLDB Endowment* 3, 1-2 (2010), 1435–1446. http://dl.acm.org/citation.cfm?id=1920841.1921017

[10] Shimin Chen. 2009. FlashLogging: Exploiting Flash Devices for Synchronous Logging Performance. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 73–86. http://dl.acm.org/citation.cfm?id=1559855

[11] Farhan S Chowdhury, Shadman Saqib Eusuf, Subhadeep Sarkar, and Tarikul Islam Papon. 2026. ZARC: Re-inventing Zone Allocation for LSMs on ZNS SSD. In *Proceedings of the 22nd International Workshop on Data Management on New Hardware*. https://doi.org/10.1145/3789237.3809132

[12] Michael Cornwell. 2012. Anatomy of a Solid-State Drive. *Communications of the ACM (CACM)* 55, 12 (2012), 59–63.

[13] Western Digital. 2026. Zoned Storage. https://zonedstorage.io/docs/introduction. (Accessed: September 27, 2026).

[14] 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

[15] 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

[16] Facebook. 2021. MemTable. https://github.com/facebook/rocksdb/wiki/MemTable. (Accessed: September 27, 2026).

[17] Gabriel Haas, Bohyun Lee, Philippe Bonnet, and Viktor Leis. 2025. SSD-iq: Uncovering the Hidden Side of SSD Performance. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4295–4308. https://www.vldb.org/pvldb/vol18/p4295-haas.pdf

[18] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, And How To Exploit It: High-Performance I/O for High-Performance Storage Engines. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2090–2102. https://doi.org/10.14778/3598581.3598584

[19] Xiao-Yu Hu, Robert Haas, and Evangelos Eleftheriou. 2011. Container Marking: Combining Data Placement, Garbage Collection and Wear Levelling for Flash. In *Proceedings of the International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 237–247.

[20] Woon-Hak Kang, Sang-Won Lee, and Bongki Moon. 2012. Flash-based extended cache for higher throughput and faster recovery. *Proceedings of the VLDB Endowment* 5, 11 (2012), 1615–1626. http://dl.acm.org/citation.cfm?id=2350229.2350274

[21] Woon-Hak Kang, Sang-Won Lee, and Bongki Moon. 2016. Flash as cache extension for online transactional workloads. *The VLDB Journal* 25, 5 (2016), 673–694. https://doi.org/10.1007/s00778-015-0414-1

[22] Shubham Kaushik and Subhadeep Sarkar. 2024. Anatomy of the LSM Memory Buffer: Insights & Implications. In *Proceedings of the International Workshop on Testing Database Systems (DBTest)*. 23–29. https://doi.org/10.1145/3662165.3662766

[23] Ioannis Koltsidas and Stratis D. Viglas. 2008. Flashing up the storage layer. *Proceedings of the VLDB Endowment* 1, 1 (2008), 514–525. http://dl.acm.org/citation.cfm?id=1453856.1453913

[24] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. 2008. A Case for Flash Memory SSD in Enterprise Database Applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 1075–1086. http://dl.acm.org/citation.cfm?id=1376616.1376723

[25] Alberto Lerner and Philippe Bonnet. 2021. Not your Grandpa's SSD: The Era of Co-Designed Storage Devices. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 2852–2858. https://doi.org/10.1145/3448016.3457540

[26] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S Gunawi. 2018. The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*. 83–90. https://www.usenix.org/conference/fast18/presentation/li

[27] 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

[28] Suman Nath and Aman Kansal. 2007. FlashDB: dynamic self-tuning database for NAND flash. *Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Network (IPSN)* (2007).

[29] Tarikul Islam Papon. 2024. Enhancing Data Systems Performance by Exploiting SSD Concurrency & Asymmetry. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 5644–5648. https://doi.org/10.1109/ICDE60146.2024.00454

[30] Tarikul Islam Papon and Manos Athanassoulis. 2021. A Parametric I/O Model for Modern Storage Devices. In *Proceedings of the International Workshop on Data Management on New Hardware (DAMON)*. 2:1–2:11. https://doi.org/10.1145/3465998.3466003

[31] Tarikul Islam Papon and Manos Athanassoulis. 2021. The Need for a New I/O Model. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR)*.

[32] Tarikul Islam Papon and Manos Athanassoulis. 2023. ACEing the Bufferpool Management Paradigm for Modern Storage Devices. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 1326–1339.

[33] Tarikul Islam Papon, Taishan Chen, Shuo Zhang, and Manos Athanassoulis. 2024. CAVE: Concurrency-Aware Graph Processing on SSDs. *Proceedings of the ACM on Management of Data (PACMMOD)* 2, 3 (2024), 125:1–125:26. https://doi.org/10.1145/3654928

[34] Jong-Hyeok Park, Dong-Joo Park, Tae-Sun Chung, and Sang-Won Lee. 2021. A crash recovery scheme for a hybrid mapping FTL in NAND flash storage devices. *Electronics* 10, 3 (2021), 327.

[35] Mohammad Sadoghi, Kenneth A Ross, Mustafa Canim, and Bishwaranjan Bhattacharjee. 2016. Exploiting SSDs in operational multiversion databases. *The VLDB Journal* 25, 5 (2016), 651–672. https://doi.org/10.1007/s00778-015-0410-5

[36] 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.

[37] Subhadeep Sarkar, Niv Dayan, and Manos Athanassoulis. 2023. The LSM Design Space and its Read Optimizations. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*. 3578–3584. https://doi.org/10.1109/ICDE55515.2023.00273

[38] 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

[39] Radu Stoica and Anastasia Ailamaki. 2013. Improving Flash Write Performance by Using Update Frequency. *Proceedings of the VLDB Endowment* 6, 9 (2013), 733–744.

[40] Radu Ioan Stoica. 2014. *Flash-aware Database Transactions*. Ph. D. Dissertation. https://doi.org/10.5075/epfl-thesis-6121

[41] Jason Taylor. 2013. Flash at Facebook: The Current State and Future Potential. In *Keynote talk at the Flash Memory Summit*. http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2013/20130813_Keynote1_Taylor.pdf

[42] Yu Wang, You Zhou, Zhonghai Lu, Xiaoyi Zhang, Kun Wang, Feng Zhu, Shu Li, Changsheng Xie, and Fei Wu. 2023. FlexZNS: Building High-Performance ZNS SSDs with Size-Flexible and Parity-Protected Zones. In *Proceedings of the IEEE International Conference on Computer Design (ICCD)*. 291–299. https://doi.org/10.1109/ICCD58817.2023.00052

[43] Tianru Zhang, Tarikul Islam Papon, Teona Bagashvili, Salman Toor, and Manos Athanassoulis. 2026. ReStore: A Reinforcement Learning Approach for Data Migration in Multi-Tiered Storage. *Proceedings of the ACM on Management of Data* 4, 3 (2026), 1–28. https://doi.org/10.1145/3802104

[44] Weilin Zhu and Wei Tong. 2023. Turn Waste Into Wealth: Alleviating Read/Write Interference in ZNS SSDs. In *Proceedings of the IEEE International Conference*

on Computer Design (ICCD). 316–319. https://doi.org/10.1109/ICCD58817.2023.00055

[45] Zichen Zhu, Arpita Saha, Manos Athanassoulis, and Subhadeep Sarkar. 2024. KVBench: A Key-Value Benchmarking Suite. In *International Workshop on Testing Database Systems (DBTest)*. 9–15. https://doi.org/10.1145/3662165.3662765