ARTIFACT EVALUATED USENIX AVAILABLE
ARTIFACT EVALUATED USENIX FUNCTIONAL
ARTIFACT EVALUATED USENIX REPRODUCED

Lost in Translation: Giving SMR Drives the Second Chance They Deserve!

Surbhi Palande[1], Puneet Mehrotra[1], Alexandra (Sasha) Fedorova[1, 2], and Margo Seltzer[1]

[1]*University of British Columbia* [2]*MongoDB*

Abstract

Shingled Magnetic Recording (SMR) drives offer higher density and lower cost than Conventional Magnetic Recording (CMR) drives but are limited to sequential writes. To support random I/O, SMR drives require a Shingled Translation Layer (STL) located either in the device firmware (Device-Managed) or the host OS (Host-Managed). While anecdotal evidence suggests Device-Managed SMR drives exhibit unacceptable performance [1], leading to the misconception that the shingled medium is inherently inferior [2,3], we show that the root cause is the resource-constrained design of on-device STLs. We demonstrate that moving the STL to the host OS makes SMR drives competitive with CMR without requiring application or filesystem changes. Our host-based STL improves 99^{th+} percentile tail latency by ~40× compared to on-device implementations. In RAID reconstruction workloads previously cited as evidence of SMR failure [1], our solution achieves a ~1.4× performance gain over CMR drives and a ~11× improvement over Device-Managed SMR.

1 Introduction

Conventional Magnetic Recording (CMR) drives contain a write head that magnetizes bits on the drive and a read head that senses this magnetic orientation. As density increases and bit size shrinks, the read head can shrink accordingly. The write head cannot: making it smaller compromises magnetic field strength, so it remains wider than the read head to preserve reliability. To accommodate a wider write head, CMR drives use guard tracks, but these gaps waste surface area and limit density.

Shingled Magnetic Recording (SMR) drives, on the other hand, overlap tracks, as shown in Figure 1, and increase disk density. The drawback is that the wider write head overwrites data in the subsequent tracks; writes must be sequential to prevent data corruption. To constrain sequential writing, SMR drives are divided into zones of overlapping tracks, with a gap between zones. *Writes in a zone must be sequential.*

Random writes from applications are converted to sequential writes by an SMR translation layer (STL). The STL resides on the drive in *device-managed* (DM-SMR) drives, in the host OS in *host-managed* (HM-SMR) drives, and in both locations in *host-aware* (HA-SMR) drives.

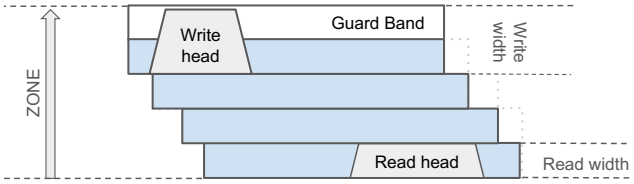


Figure 1: (Adapted from [4]) In SMR, with no inter-track gaps, the wider write head overlaps the next track, creating "shingled" tracks.

SMR's sequential-write constraint aligns with log-structured architectures [5] by treating zones as segments. Data is written sequentially rather than overwritten in place, requiring mapping tables to translate logical addresses to physical. Fine-grained 4 KB block mapping requires 2 GB memory per 1 TB capacity. Multi-terabyte SMR drives limit expensive on-device DRAM to ~512 MB for mappings, track buffering, and other structures [6]. Consequently, a fully log-structured STL is impractical for DM-SMR drives; to our knowledge, **no DM-SMR drive implements it**.

A common alternative to a log-structured design is the *hybrid-cache layout*, which places randomly written data into a small number (typically <1% of device capacity) of *write cache zones*, while placing sequentially written data directly in their persistent location in the data zones. Cache zones are on the *same* device as data zones. The mapping table size shrinks, since only the cache zones require fine-grained mapping. However, once cache zones fill, their contents must be merged with the persistent data zones, introducing significant performance overhead [7] (Section 5.1).

Several architectures similar to the hybrid-cache go by other names: *Set-Associative Cache* [8], *Hybrid Log-Block* [9], *Hybrid-Mapping FTL* [10], and *Block Associative Sector translation* [11]. **The on-device STL on DM-SMR drives uses the *hybrid-cache architecture*** [12–16]. Sustained random writes, such as those in a ZFS RAID reconstruction [17], exhaust the on-device cache in these architectures, causing

the "SMR performance cliff" and notoriously slow reconstruction [1, 2].

Log-structuring similarly requires garbage collection (GC) to reclaim invalidated space, which degrades performance [18–20]. However, our study (Section 5) demonstrates that log-structuring provides less performance degradation compared to the aggressive stalls of hybrid-cache eviction. While log-structuring is impractical on-device, it can be implemented on the host managing a HM-SMR drive.

Prior work demonstrates that Host-Managed drives outperform Device-Managed drives [21, 22] by bypassing conventional block I/O and handling random writes via application-aware file systems or custom software layers [23–27]. However, these solutions require *non-trivial* modifications to applications or file systems, because they eschew conventional block I/O. Consequently, *host-managed devices are largely restricted to enterprise environments* [28] with the engineering resources to implement these substantial software rewrites.

To the best of our knowledge, we are the first to provide a comparative analysis of a *block-based* hybrid-cache STL implemented both on-device and in-host. We are the first to evaluate a host-based *log-structured* STL to hybrid-cache architectures across both device and host environments.

For Zipfian workloads, our log-structured STL consistently outperforms the hybrid-cache STL regardless of placement, delivering ~4-8× better 90th percentile and ~40× better 99th percentile tail latencies. During ZFS RAID reconstruction, moving the hybrid-cache STL to the host improves reconstruction time by ~2.2× over the on-device STL. Switching to our host-based log-structured STL reduces reconstruction time further: by ~11× compared to the on-device STL, ~4.9× compared to the host-based hybrid-cache, and ~1.4× even when compared to a baseline CMR drive.

However, when the random writes are concentrated to fewer data zones, hybrid-cache eviction can outperform log-structured GC operating under harsh conditions of high zone utilization, low free disk space, and nearly no over-provisioned space. We stress tested our log-structured STL under these adversarial conditions by running a parallel Linux kernel compile workload. Despite the sharper performance drop due to GC, our host-based log-structured STL still performed 1.2× better than the on-device STL.

This demonstrates that if the manufacturers *simply shift the on-device translation layer to the host*, random write performance can improve dramatically at little cost to the consumer. This is a **call for action to the device manufacturers**. This shift leverages host DRAM to support the larger mapping tables required of the log-structured architecture. While a host-based STL consumes system memory, host DRAM is significantly more cost-effective and extensible than its on-device equivalent [29–32] [1].

Moving the STL to the host OS provides flexibility in STL design, maintainability, and dynamic metadata scaling. This controls DRAM usage, maintaining HM-SMR cost-effectiveness over CMR or DM-SMR alternatives.

[1] Reasons why DRAM on SSDs increases cost is analogous for SMR

Table 1: Terminology for SMR Translation Layers

Architecture	Placement	
	On-device	**In host OS**
Hybrid-Cache	Device-Hybrid	Host-Hybrid
Log-structured	✗	Host-LS

To date, SMRs have been relegated to archival workloads, *due to their inferior on-device STL implementation, whose performance degrades unacceptably under sustained random IO*. **Manufacturer-supported host-based STLs** can position HM-SMR as a high-performance, cost-effective alternative to both DM-SMR and CMR.

The contributions of our work are:

- We reverse engineer the on-device STL design to determine the cache implementation and eviction process.
- We introduce a **host-based STL** that replicates the hybrid-cache architecture of on-device firmware to demonstrate immediate performance gains that stem solely from increased memory resources.
- We use this host-based STL replica to further our understanding of the on-device STL.
- We design and implement a **state-of-the-art host-based, log-structured STL**. We show that it provides significantly better worst-case random write latency compared to both device-based and host-based hybrid-cache designs.
- We provide an **empirical demonstration** that host managed SMR drives, when paired with a log-structured STL, serve as a low-cost, high-performance alternative to both DM-SMR and conventional CMR drives for NAS boxes.

From now on, we name our STLs as indicated in Table 1. Section 2 details our reverse engineering characterization of the on-device STL, `Device-Hybrid`. This informs the design of our host-managed Hybrid-Cache device mapper, `Host-Hybrid` (Section 3). Next, we introduce the design of our host-managed log-structured device mapper, `Host-LS`, in Section 4. Finally, we compare the performance of our three STL implementations in Section 5, review related work in Section 6, and conclude in Section 7.

2 Reverse Engineering the On-Device STL

To isolate STL placement from hardware variability, we reverse-engineer `Device-Hybrid` on an HA-SMR drive (ST8000AS022); this model supports both a Device-Managed

Drive Model	Size	RPM	Buffer	Sector	Max Sust. Rate	Exposes Zones	STL
HA-SMR ST8000AS0022	8TB	5900	128MB	512B	190 MB/s	256MB	✓
DM-SMR ST8000DM004	8TB	5400	256MB	512B	190 MB/s	✗	✓

Table 2: Characteristics of SMR Drives Used in on-device STL (`Device-Hybrid`) Reverse Engineering (8TB = 7.2TiB).

Blk Size	Cache (GB)	# Mappings
4K	~0.74	191,172
32K	~5.83	191,172
64K	~11.66	191,172
128K	~23.33	191,172
256K	28.00	114,688
512K	28.00	57,344
1MB	28.00	28,672

Table 3: **HA-SMR:** Cache Size.

Blk Size	Cache GB	# Mappings
4K	~ 1.4	368,795
32K	~ 11.2	368,795
64K	~ 22.4	368,795
128K	~ 44.8	368,795
256K	48.3	196,608
512K	48.3	98,304
1MB	48.3	49,152

Table 4: **DM-SMR:** Cache Size.

Location	BW MB/s
17GB	193.17
4TB	167.82
6TB	127.84
6.1TB	123.02
6.5TB	113.59
Cache	123.57

Table 5: **HA-SMR:** Location.

Location	BW MB/s
0GB	197
4TB	154
6TB	122
6.5TB	107
Cache	102

Table 6: **DM-SMR:** Location.

mode for random writes and a Host-Managed mode that exposes zone information to the host for sequential-only writes. This enables a direct comparison between device- and host-managed STLs on identical physical media. We also verify that `Device-Hybrid` on a DM-SMR drive (ST8000DM004) (Table 2) has the same cache architecture as the on-device HA-SMR STL.

Our HA-SMR drive (ST8000AS022) has 29809 zones; each zone is 256 MiB (268 MB). It takes ~1.3 to 2.8 s to read/write a zone from the host.

2.1 Cache parameters

We first characterize the hybrid cache by its capacity, zone mapping, overwrite support, and physical location.

Size of the Cache: To determine cache capacity, we iterated over a set of blocks sizes between 4KB and 1MB. For each, we issued 35 GB of uniform random writes using `fio` [33]. When we observe an order-of-magnitude drop in write bandwidth, we conclude that the cache filled and began eviction. Figure 2 (left) illustrates this behavior for the 1MB workload.

Column 2 in Tables 3 and 4 shows the effective cache size as a function of the block size. Notice that **the effective cache size shrinks as a function of I/O count, not total data volume**. (This finding is consistent with Skylight's [12] findings for the ST8000DM004 [34].) The fact that the effective cache size is limited by I/O count indicates that the bottleneck resource is the fixed-size mapping table that contains the disk address for each block in the cache.

Column 3 in Table 3 shows the number of mappings supported by the STLs, For block sizes 256 KB and larger, bandwidth drops only after reaching the drive's physical capacity; these workloads reach the capacity limit before exhausting mapping entries. We assume that each mapping stores a disk address (8 bytes) and a length (2 bytes, sufficient for an ATA write size up to 32 MB). The HA-SMR drive's 191,172 entries require only ~1.8 MB of DRAM. In contrast, a table capable of mapping the entire 28 GB cache containing 4 KB blocks would require ~7 million entries, consuming ~56 MB. The on-device DRAM is limited and shared among multiple firmware components [6], including track buffering, I/O scheduling, and sector remapping. Consequently, manufacturers strive to minimize the memory footprint of resident data structures, including mapping tables.

Effective cache size is determined by both physical cache capacity and on-device DRAM mapping limits.

Cache medium: Transitioning to skewed (Zipfian) writes (Figure 2; skew increases from left to right) reveals that overwriting an already-cached block yields a larger effective cache size than writing to new blocks. Since the uniform random cache size already matches the maximum physical capacity, this increase indicates support for **in-place cache updates**. Given drive cost constraints and the host-invisibility of these tracks, we conclude the cache resides in a dedicated, CMR region distinct from the host-visible CMR section.

Mapping: We varied the number of zones to which we write to determine how data zones are mapped to cache zones. The effective cache size is constant with respect to the number of modified zones, suggesting that the **cache is fully associative**.

Location on the Disk: To determine the physical location of the cache, we compared the sequential write bandwidth across different disk regions to the observed cache bandwidth: ~120 MB/s for HA-SMR (Figure 3) and 102 MB/s for DM-SMR. As shown in Table 5 and Table 6, these rates most closely match sequential performance at 6.1 TB for the HA-SMR drive and 6.5 TB for the DM-SMR drive. We conclude that the **cache resides on the slower inner tracks of the drives**, reserving faster outer tracks for data.

2.2 Cache eviction

In the discussion that follows, we analyze the 1 MB workload; behavior for other block sizes is similar. We issued 1 MB uniform random writes (35 GB for HA-SMR; 55 GB for DM-SMR) followed by pauses of 1–10 hours to observe background cache eviction (BG-eviction).

On HA-SMR (Figure 3), eviction failed to start within two hours (Panel 2), indicating a lazy BG-eviction mechanism. Cleaning began just before the three-hour mark, initially freeing only ~100 MB (Panel 3). After a seven-hour pause, the cache absorbed 16.6 GB before resuming eviction (Panel 4). Ultimately, evicting the full 28 GB cache took ~5 hours, completing only after an 8-hour total pause.

DM-SMR exhibits more aggressive, yet slower eviction; eviction began within two hours, freeing 6 GB, but took nine hours to evict the entire cache. This slower rate is not entirely explained by the DM-SMR's lower rotational speed (5400 vs. 5900 RPM). However, without internal architectural access, the precise cause of this performance delta remains opaque.

Irrespective of the performance delta, cache eviction on both drives is remarkably slow. To analyze these proprietary algorithms, we implemented `Host-Hybrid` (Section 3),

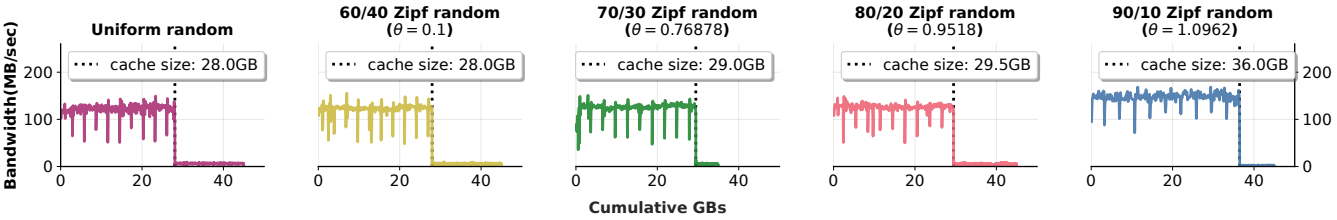


Figure 2: **`Device-Hybrid` (HA-SMR) Cache Medium:** Panel 1: 28 GB effective cache for 35 GB of uniform 1 MB random I/Os. Panels 2–5: Increasing Zipfian skew (overwrites). Higher skew increases overwrite likelihood on the same 1 MB block. If the on-device cache were implemented in SMR zones, these overwrites would consume extra space without changing effective cache size. However, effective cache size increases with skew, proving the cache overwrites repeated writes and resides on conventional tracks, not shingled ones.

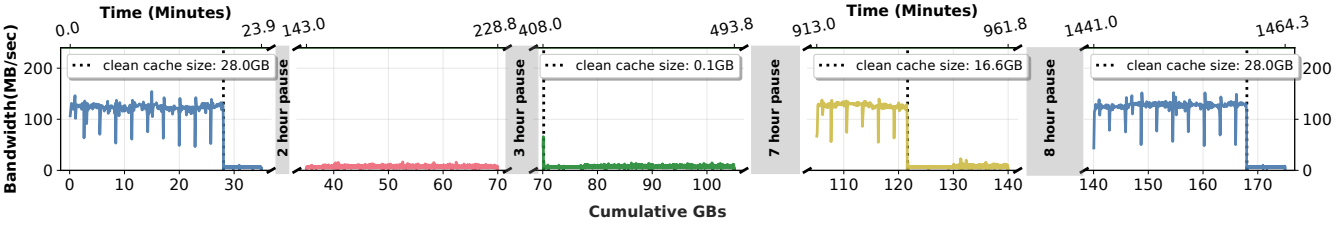


Figure 3: **`Device-Hybrid` (HA-SMR) Performance:** Writing 35GB of 1MB uniform random writes asynchronously reveals that the effective cache size for 1MB writes is 28GB (left most panel). Panel 2 shows that after a 2-hour pause, no Background (BG) cache eviction takes place. Panel 3 shows that BG eviction starts around the 3 hour mark. Panel 4 shows that after a 7 hours pause more than half of the cache is emptied, and Panel 5 shows that the entire cache is evicted within 8 hours.

a host-based STL that executes on the HA-SMR drive. We iteratively refined it until its eviction performance matched the observed `Device-Hybrid` behavior on the same drive. To understand BG-eviction, we analyzed `Host-Hybrid`'s kernel logs showing cache-mapped data zones. The 28 GB HA-SMR cache contains 112 zones, all of which are full after the uniform workload. Each cache zone contained data from ~255 data zones; the entire cache held writes from ~22.8 K unique data zones. This trend held for smaller block sizes as well. Since BG-eviction began at the 3-hour mark and finished by the 8-hour mark, the drive evicted ~22.8 K data zones in 5 hours, averaging ~0.8 s per data zone.

Having understood the BG-eviction behavior, we shifted our focus to characterize its Foreground cache eviction (FG-eviction) policies. A key for understanding `Device-Hybrid` is observing how many stalled writers are unblocked during FG-eviction. We used the same 1MB uniform random workload as in the BG-eviction experiments. After the cache fills at 28 GB (Figure 3, Panel 1), FG-eviction begins. Analysis of `Device-Hybrid`'s bandwidth during eviction and `Host-Hybrid`'s logs shows that `Device-Hybrid` achieves bandwidth that nearly matches the number of data blocks released per evicted data zone. We conclude that since **`Device-Hybrid`'s cache resides in CMR tracks, it accepts one new write for each block evicted from the cache.**

2.3 Victim selection heuristics

Next, we investigate how `Device-Hybrid` selects data zones to evict.

A greedy policy selects data zones with the greatest number of cached blocks. A round-robin policy 1) iterates over all the cache zones, 2) identifies all data zones present in the current cache zone, 3) for each such data zone, collects all its blocks in the cache and rewrites the data zone, merging live data in the data zone with the new data from the cache. Both policies are simple to implement on resource-constrained devices. We implemented both of them and compared results by matching the number of data blocks released (from `Host-Hybrid` logs) with `Device-Hybrid` bandwidth.

When `Host-Hybrid` uses the greedy algorithm, both the number of blocks evicted and the resulting bandwidth decrease over time. However, `Device-Hybrid`'s bandwidth varies unpredictably, indicating that it does not exhibit greedy behavior. When `Host-Hybrid` implemented round-robin, we observed not only the same unpredictable behavior but also that `Device-Hybrid`'s bandwidth closely matched with the number of data blocks released.

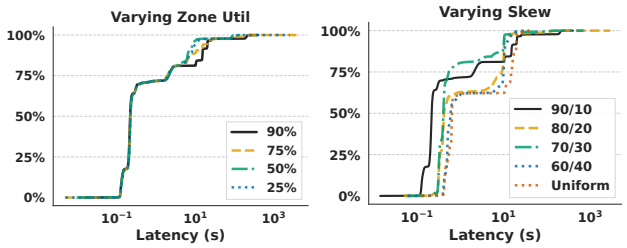
This result differs from the prior conclusions that the cache was log-structured and evicted the oldest data zones first using a FIFO policy. In a log-structured cache, FIFO and Round-Robin eviction are equivalent because data are written sequentially. However, we found that `Device-Hybrid` instead implements an overwrite-in-place cache. In this design, FIFO requires tracking the order in which data zones are written, increasing the amount of DRAM-resident metadata. By contrast, Round-Robin eviction requires no additional metadata.

2.4 Zone Utilization and Skew Effects

Finally, we discover that the write performance of `Device-Hybrid` is independent of both the zone utilization and the

Work	Cache Medium	Cache Impl.	Victim Selection	Cache Mappings	Cache Loc.	Indep. Util.	Indep. Rand.	Policy (Lazy/ Aggressive)	Queuing Behavior
Skylight [12] (ST8000DM004)	Shingled	Log-structured	FIFO	~200 K	Outer Tracks	N/A	N/A	Aggressive	N/A
HA-SMR [15, 16] (ST8000AS0022)	Shingled	Log-structured	FIFO	N/A	Outer Tracks	N/A	N/A	Aggressive	N/A
Our Work (ST8000AS0022, ST8000DM004)	**CMR**	**Overwrite-in-place**	**Round-robin**	**191 K / 368 K**	**Inner Tracks**	✓	✓	**Lazy / Aggressive**	**Writes unblocked as free blocks created**

Table 7: Comparison of SMR Cache Design, Capacity, and Policy across literature (Skylight [12] and Wu et al. [15, 16]).



(a) Zone Utilization (b) Random Writes

Figure 4: Latency CDF plots for `Device-Hybrid` across different (a) zone utilizations and (b) random write skew.

random write skew (Figure 4).

We populated *all* the data zones to a target zone utilization and ran a workload that issued random (uniform or Zipfian) writes to the drive. Approximately 80% of the 1MB workload fits in the cache, so the 80th-percentile tail latency remains the same across all workloads, regardless of the initial data-zone utilization. During eviction, the number of cached data blocks associated with each data zone is independent of the initial zone utilization and is determined primarily by the workload skew. Consequently, the same number of writers can proceed regardless of the initial data-zone utilization. Eviction time is dominated by the highest LBA that has received a random overwrite, as eviction rewrites sequentially up to the highest valid LBA in the data zone. Each random write has a 0.1 probability of landing within the last 10% of the zone capacity. Suppose each cached data zone receives, on average, $w \geq 1$ random writes. Then, approximately 10w% of the cached data zones will contain at least one overwrite within the last 10% of the zone. Consequently, these zones are rewritten nearly to the end of the zone during eviction, making their tail latency independent of the original data-zone utilization.

While higher skew increases effective cache size, lower skew ensures that more blocks from a specific data zone are mapped in the cache. During eviction, the release of these data blocks from the cache zones allows more application writes to be unstalled simultaneously, which offsets the reduced cache efficiency. This balance ensures that write tail latency remains largely independent of skew. Figure 4b shows that the tail latency is dependent on the skew up to the 80^{th} percentile (reflecting cache writes). At greater skew, tail latency is independent of skew. In fact, the 99^{+} percentile uniform random writes tail latency (black line in Figure 4b) is significantly

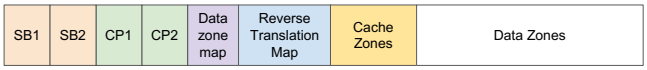


Figure 5: **Metadata layout used by `Host-Hybrid`.** SB1/SB2: Duplicate superblocks. CP1/CP2: Checkpoint regions.

lower than that of the skewed workloads (42.2 versus 345.3 s).

2.5 Discussion

Two prior works also characterized SMR drive internals: Skylight [12] reverse-engineered the STL on the DM-SMR drive (ST8000DM004). Wu et al. [15, 16] characterized the STL on the HA-SMR drive (ST8000AS0022). Our work provides a more detailed characterization of the STL on both these drives, particularly with respect to their cache behavior. We needed these additional details so that we could faithfully emulate their behavior in `Host-Hybrid`. This more detailed characterization (Table 7) enhances the research community's understanding of the inner-workings of on-device STLs.

3 `Host-Hybrid` design

We implemented `Host-Hybrid`, a Linux device mapper that exports an unrestricted virtual block device, following the design in Section 2. `Host-Hybrid` serves three purposes: it allows us to faithfully reverse engineer `Device-Hybrid` on a Host Aware drive, enables direct placement comparisons between STLs on the host and on the device, and allows us to directly compare different host-based STL architectures (Hybrid vs log-structured), using the same device.

3.1 Overview

`Host-Hybrid` partitions the disk into a fully-associative cache region that accepts random writes and a data region that accepts write-pointer–aligned sequential writes.

Both Host-Managed and Host-Aware SMR drives include CMR regions that can be used for the cache. Our HA-SMR drive provides only 64 CMR zones (16 GB). Because `Device-Hybrid` implements a larger 28 GB cache (Section 2), we must match this physical capacity in `Host-Hybrid` to ensure a fair comparison of cache eviction. Consequently, `Host-Hybrid` cannot use a simple in-place overwrite cache; it instead approximates this behavior using a log-structured cache.

`Host-Hybrid` models the cache as zone-sized segments, sequentially logging random data to an active segment and advancing a persistent write pointer. Once a segment fills, `Host-Hybrid` designates a new free zone active. All out-of-place

data in the cache is tracked using a block level translation table maintained for the blocks in the cache.

As the on-disk cache fills, `Host-Hybrid` triggers FG-eviction or opportunistic BG-eviction. Both merge cached blocks with on-disk data, zero-filling gaps to maintain write pointer alignment. Merged data is written to a newly allocated data zone, with logical-to-physical mappings tracked via a zone translation map. Similar to `Device-Hybrid`, `Host-Hybrid` implements a credit-based system by unblocking waiting application writers during FG-eviction based on the number of freed blocks. This process repeats per data zone within the victim cache zone, allowing FG-eviction to sustain continuous random write streams.

`Host-Hybrid` finds data as follows: Data in the cache (i.e., the LBA appears in the mapping table) is more recent than the data in the data zone. A cache absent LBA could be found in the LBA's data zone, as long as the LBA zone offset is smaller than the zone's write pointer. Otherwise it is invalid.

A formatting tool writes the `Host-Hybrid` metadata (see Figure 5) to the conventional tracks. For an 8 TB drive with a 28 GB cache, this metadata fits within a single 256 MB CMR zone.

3.2 On-disk Data Structures:

Superblock: The superblock stores metadata for all other on-disk data structures and records device-level information, including the total number of zones, the number of cache zones, the number of data zones etc. `Host-Hybrid` maintains two copies of the superblock, alternately writing them to the first and second blocks of the device.

Data Zone Map: As we saw in Section 3.1, `Host-Hybrid` relocates data zones during cache eviction. The Data Zone Map translates a logical zone number to a physical zone number. Since logical zone numbers are sequential, the map is simply an array, indexed by the logical zone number. Each entry in the array holds the physical zone number and the location of the zone's write pointer to distinguish sequential writes from random writes, resulting in a total zone map of ~465KB. Data zone map entries do not span sectors, ensuring atomic updates. In case of a crash, discrepancies in the data zone map are identified between write-frontier pointers recorded in the data zone map and the SMR drive provided zone write pointers. We update the data zone map accordingly.

Reverse Translation Map: Applications access data via LBAs, which `Host-Hybrid` writes to different physical block addresses (PBAs) in the cache. `Host-Hybrid` maintains a persistent Reverse Translation Map for every 4 KB cache block to manage these mapping. Reverse-map entries do not span sectors, ensuring atomic updates. By leveraging the contiguous and statically assigned nature of cache PBAs, the map is a simple array, indexed by PBA. Each 8-byte entry stores the corresponding LBA, resulting in a total mapping table size of 56 MB for the 28 GB physical cache.

When data is written, it is appended to sequential PBAs, causing the corresponding new reverse-map entries to be clustered within one or a few sequential blocks. The mappings for the overwritten data are then invalidated. Because these invalidations may be scattered throughout the cache, their corresponding reverse-map entries are also scattered. `Host-Hybrid` flushes the sequential reverse-map updates before committing the scattered invalidation updates.

Checkpoints: Checkpoints record the counts of free data zones and cache blocks, the active write frontier, and the current write pointer. `Host-Hybrid` maintains two checkpoint copies, updating them alternately. In contrast, only a single persistent copy is maintained for the Data Zone Map and the Reverse Translation Map.

3.3 In-Memory Data Structures

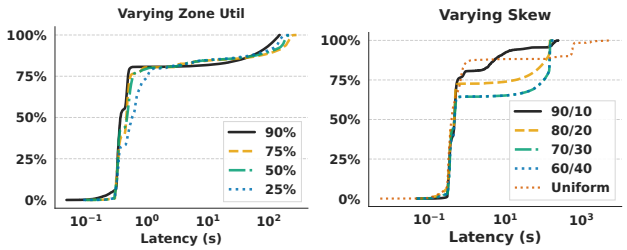
Applications issue LBA-based requests, which `Host-Hybrid` maps to physical locations. While the on-disk Reverse Translation Map stores PBA-to-LBA entries, application operations require LBA-to-PBA translation. To minimize overhead, `Host-Hybrid` maintains DRAM-resident forward mappings via a Red-Black tree of variable-length extents $(LBA, PBA, length)$ using Linux's standard Red Black tree APIs. For eviction, `Host-Hybrid` uses a corresponding reverse extent tree to translate victim PBAs back to LBAs; bidirectional pointers ensure synchronization. These two entries occupy 64 bytes. During cache eviction, `Host-Hybrid` merges cache and data zone contents into a new zone, tracking availability via an in-memory Data Zone bitmap.

3.4 Batched Updates

`Host-Hybrid` batches updates to balance performance and consistency. While in-memory metadata, extent trees, bitmaps, and mapping entries, are updated synchronously, persistent flushes are decoupled. A background thread lazily flushes dirty reverse translation and data zone map entries every 10 seconds or whenever a full page (4KB) of updates accumulates. A final checkpoint write ensures metadata consistency.

This buffering strategy mirrors techniques used in log-structured file systems and Linux writeback (`writeback()`) to coalesce I/O and reduce write amplification [35]. Similarly, disk write caches improve performance through reordering and batching [36] but introduce durability risks during power failure [37].

Durability and write ordering are guaranteed only after a 'disk-cache FLUSH' command, which acts as a write barrier, ensuring that all writes acknowledged before the command become durable upon its completion [38]. However, disk-cache FLUSHes are expensive [6, 37]. Because FLUSHes are synchronous, they interfere with disk I/O scheduling, increase arm movement, stall reads, and prevent writes from accumulating in the disk cache. Consequently, issuing a FLUSH after every write can significantly degrade performance. Instead, to guarantee zero data loss up to an explicit FLUSH command, `Host-Hybrid` intercepts it, synchronously flushes the 512-entry translation-page buffer, and then propagates the



(a) Zone Utilization (b) Random Writes

Figure 6: Latency CDF plots for `Host-Hybrid` for different (a) zone utilizations and (b) random write skew.

FLUSH command to the underlying SMR device. If FLUSH commands are not used, the data at risk on power failure may be greater than for a conventional drive.

Nonetheless, a crash between device mapper metadata flushes could lose up to 512 reverse translation and 4 data zone map entries in the worst case. We consider this durability trade-off orthogonal to our architectural contributions, as such risks are well-understood ([39–41]) and typically mitigated via battery-backed RAM or similar robustness mechanisms in production environments. While this increases hardware costs, enterprise servers typically share a centralized battery backed pool across multiple drives.

3.4.1 Crash Consistency

Although `Host-Hybrid` weakens durability guarantees, it preserves data consistency through log structuring and carefully ordered metadata updates.

`Host-Hybrid` does not buffer application data. Incoming writes are immediately forwarded to the underlying SMR device; only metadata is buffered in memory. Sequential writes within a data zone require no device-mapper metadata, as their location is predetermined. In contrast, random writes depend on reverse-translation entries and may be lost if a crash occurs before their metadata is persisted. Because reverse-translation entries do not span sectors and newer reverse-translation entries are persisted before older entries are updated, recovery observes either the previous or the updated mapping, depending on whether the metadata reached the device before the crash.

During cache eviction, merged data is first written to the sequential data area. The data-zone table is then updated to reference the new location, the corresponding reverse-translation entries are invalidated, and a checkpoint is issued to persist the metadata. Only after the checkpoint completes are the original data zone reset and the reclaimed cache space reused. This ordering preserves consistency across crashes. If a crash occurs before the checkpoint completes, recovery observes either the previous or updated metadata. Consequently, data is never lost during eviction and remains reachable either through the original data zone and cache or through the newly written data zone.

3.5 Skew and Utilization Independence

Similar to `Device-Hybrid`, `Host-Hybrid` is also zone utilization and skew independent (Figure 6). This is expected for the same reasons discussed in (Section 2.4), since this is fundamental to the STL's hybrid architecture.

4 `Host-LS` Implementation

We designed `Host-LS`, a Linux device-mapper that implements the log-structured architecture to manage SMR devices in host-managed mode (HM-SMR, HA-SMR).

4.1 `Host-LS` Overview

`Host-LS` models the SMR drive as a collection of zone-sized segments. It writes data sequentially to an active write segment and advances a persistent write pointer as data are written. When the write segment fills, `Host-LS` selects a new free segment as the "write frontier" and continues sequential writing. All overwrites occur out of place and invalidate older copies of data. This creates free blocks within segments. Since these free blocks are scattered, `Host-LS` cannot reuse them directly for sequential writes.

`Host-LS` reclaims space via GC. `Host-LS` maintains two active frontiers: an active "write frontier" for application data and a "GC frontier" for relocated data during GC. When free zones drop below a threshold, foreground GC (FG-GC) blocks application writes to select and evacuate a victim zone. Valid blocks move to a "GC frontier" before the victim is marked free.

To maintain progress, `Host-LS` implements a credit-based policy to unblock application writes incrementally, matching the volume of invalid blocks processed. This continues until the number of free zones exceeds the threshold. When idle, `Host-LS` performs background GC (BG-GC) similarly, without needing to wake blocked writers.

By maintaining two separate frontiers, `Host-LS` isolates hot application writes from cold GC writes, placing recently written data in different zones from older GC data. This separation minimizes both write amplification and GC frequency, enhancing performance for non-uniform random write workloads [42].

`Host-LS` maintains metadata to find out-of-place data and to conduct GC. A format tool initially writes this required metadata (shown in Figure 7) to the **conventional tracks** on the Host-Managed or Host-Aware drive. This allows `Host-LS` to overwrite frequently updated metadata in-place, while the rest of the drive is managed in a log-structured fashion. For an 8 TB drive, this metadata fits within the 16 GB CMR zones available on the HA-SMR drive.

We now discuss the on-disk and in-memory data structures necessary for `Host-LS`'s working.

4.2 On-Disk Data Structures

Superblock: The location of the on-disk data structures, device related information such as the number of conventional and shingled zones on the disk, and other metadata are stored

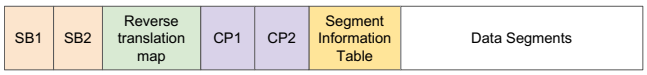


Figure 7: Metadata layout used by Host-LS. The metadata is written in the conventional zones on the SMR Drive.

in a Super Block; two copies are alternately written to the first and second blocks (4 KB each).

Reverse Translation Mapping: Applications access data using LBAs, but they are written to different PBAs. PBAs change whenever data are modified or relocated during GC. Consequently, storage systems must support two primary operations: searching for data by LBA during reads and writes and identifying data by PBA during GC. Consistent with the LFS-like approach, and similar to Host-Hybrid, we maintain both forward and reverse extent trees in memory (Section 4.3) but persist only the reverse translation map to the device's conventional track region, which makes GC operation faster. Since application writes stall while FG-GC creates sequential space for writing, faster FG-GC improves the performance of application writes [43].

To summarize, Host-LS stores the reverse translation map on the conventional tracks of the SMR drive. Our 8TB SMR drive with 29,807 zones, consumes 14.5 GB of on-disk space to store the Reverse Translation Table.

Per-Segment Metadata: GC selects a segment for cleaning based on its mode of operation. BG-GC selects a segment using the Cost-Benefit heuristic [5] based on the age and the number of free blocks it contains. FG-GC must create sequential space as quickly as possible to accommodate incoming application writes, so it greedily selects the segment with the greatest number of free blocks [5].

To facilitate segment selection, Host-LS maintains a **Segment Information Table (SIT)**, where each entry stores the valid block count and last modification time for its respective segment. The SIT occupies 644 KB on disk. Each write updates the segment entry in the in-memory SIT. Host-LS uses the SIT to generate an in-memory segment bitmap to identify free segments. Every segment entry is stored in additional in-memory trees (Section 4.3): the FG-GC victim tree, SIT tree, and a BG-GC tree.

In case of a crash, lost segment time-statistics only affect GC victim selection heuristics; the valid counts can be reconstructed.

Checkpoints: Every flush interval, Host-LS batches and flushes the Reverse Translation Table and Segment Information Table pages to the disk. To indicate a successful metadata flush, Host-LS then writes a **checkpoint** to the disk. We maintain two copies of the checkpoint of one 4KB block each, written alternately, to ensure that we always have a valid checkpoint. We maintain only one persistent copy of the SIT, since we can reconstruct the number of valid blocks from the Reverse Translation Table by reading it to identify all blocks containing valid data. Checkpoint is triggered when new zone segments/frontiers are assigned.

4.3 In-Memory Data structures

We borrow the extent and reverse extent tree implementations from Host-Hybrid (Section 3.3), and like Host-Hybrid, Host-LS caches these in memory to expedite GC. We implement an **in-memory SIT tree** that indexes a segment by the segment number for use during the BG-GC mode.

The BG-GC process constructs a **BG victim tree** by indexing segments from the in-memory SIT tree using a cost-benefit metric. This metric selects victim zones by balancing low valid block counts with segment age. Once background cleaning concludes, the tree is deallocated.

Victim segment selection is time-critical during FG-GC, because application writes might be stalled. We maintain an in-memory balanced tree, indexed by the number of valid blocks per segment. Each node stores an entry from the SIT, consisting of the segment number, the number of valid blocks, and the segment's modification time. Segments with the same number of valid blocks are ordered by modification time. FG-GC selects the leftmost node, which corresponds to the segment with the fewest valid blocks. Host-LS maintains an in-memory Segment bitmap that indicates if a zone segment is free or not.

4.4 Batched updates and Crash consistency

Similar to Host-Hybrid (Section 3.4), Host-LS batches metadata updates. Host-LS ensures crash consistency by relying on the sector atomicity guarantees provided by SMR, enforcing strict write ordering, and preventing application writes from reusing a recycled GC segment until the mappings for the new destination segment have been flushed. Similar to Host-Hybrid, the newer updates to the reverse translation map remain sequential and are flushed before the scattered invalidations to the older data. This ensures that recovery observes either the older or newer mapping; data is never lost during garbage collection.

4.5 Difference from the Original LFS design

To improve overall performance over traditional LFS [18, 19], Host-LS introduces four key optimizations. First, Host-LS stores metadata in host-visible CMR zones where it is updated in-place. By decoupling metadata from data, Host-LS ensures metadata never requires GC. Second, it segregates hot application data from cooler GC-relocated data to minimize write amplification. Third, to mitigate tail latency during foreground (FG) GC, Host-LS selects victim segments using an in-memory resident FG-GC tree. Finally, LFS cleaning waits until an entire free segment is available before unblocking any writers. In contrast, Host-LS implements a credit-based unblocking policy that cleans a segment and unblocks only as many writers as can be accommodated by the free blocks reclaimed from that segment. Host-LS utilizes limited reserved capacity to accommodate these writes during cleaning.

Disk	Type	STL	RPM	Max Transfer Rate
WD80EAZZ [44]	CMR	None	5640	185 MB/s
ST8000AS022 [45]	HA-SMR	`Device-Hybrid` `Host-Hybrid` `Host-LS`	5900	190 MB/s

Table 8: **8TB Drives used in the evaluation**

5 Evaluation

Our experimental setup consists of a Dell Precision 7820 workstation (Intel Xeon 3.3 GHz, 128GB DRAM) running Linux kernel 6.6.0, with the devices shown in Table 8.

We compared three configurations on the same HA-SMR drive: `Device-Hybrid`, `Host-Hybrid`, and `Host-LS`. The `Device-Hybrid` vs. `Host-Hybrid` comparison isolates STL placement (device-managed vs. host-managed), while the `Host-Hybrid` vs. `Host-LS` comparison isolates architectural design (hybrid-cache vs. log-structured). Our evaluation characterizes the impact of STL design versus placement (device vs. host) by addressing two research questions:

RQ1) What are the tradeoffs between implementing the hybrid architecture on the device or on the host?

RQ2) What are the tradeoffs between host-based hybrid-cache and log-structured architectures?

Before each run, we ensured a consistent initial state. For `Device-Hybrid`, we reset zone pointers to clear the host-invisible disk cache. For host-resident configurations (`Host-Hybrid` and `Host-LS`), we performed sequential writes immediately following initialization to enforce a known host-managed state. Because `Device-Hybrid`'s internal algorithms are opaque, `Host-Hybrid` serves as a transparent proxy for analyzing underlying performance trade-offs.

We begin with a block-level microbenchmark analysis, which provides the foundation for understanding the filesystem and application-level workloads presented later.

5.1 Block level microbenchmark

We use the `fio` [33] benchmark to issue writes following uniform and Zipfian distributions. Using `fio-genzipf`, we identify θ values that yield Pareto-like access patterns[2] that provide a more intuitive characterization of the write access pattern. We characterize these workloads using the parameters detailed in Table 9: block size 4KB represents the standard filesystem blocks, and 1MB represents large writes [46, 47].[3]

We give `Host-Hybrid` a cache whose size matches that of `Device-Hybrid` (Table 9). Similarly, the number of free zones on `Host-LS` matches the `Device-Hybrid` cache size, so that cleaning occurs at the same time on all three STLs. Note that, on `Host-LS`, ~99% zones are filled with workload-defined zone utilization. This specific alignment allows us to isolate architectural performance under identical capacity pressure ensuring a fair comparison.

[2]For discrete events, Zipf and Pareto distributions are functionally equivalent in this context.

[3]1MB is the maximum I/O size supported by the Linux kernel.

In the following sections, we discuss results for only the 90/10 Zipfian and uniform random write distributions, representing the most friendly and most adversarial write patterns respectively [48]. We omit other qualitatively similar plots, mentioning performance differences wherever relevant.

Zone Util. (%)	Write Size (Burst Size)	Random Write Distribution		CMR Bandwidth (MB/sec)	Effective Cache Size (GB) for `Device-Hybrid` on HA-SMR
90% 75% 50% 25%	1MB (45 GB)	Zipf	90/10	98.39	36.5
			80/20	82.51	29.5
			70/30	79.80	29.5
			60/40	81.05	28
		Uniform		81.05	28
	4KB (5 GB)	Zipf	90/10	2.46	2
			80/20	1.50	0.75
			70/30	1.22	0.75
			60/40	1.28	0.75
		Uniform		1.26	0.75

Table 9: **Workload Parameters and Baseline Performance.** Each experiment is parameterized by a - zone utilization, write size (burst size), and random write distribution. The burst sizes are chosen to trigger eviction in `Device-Hybrid`, `Host-Hybrid`, and `Host-LS`. The table also reports the baseline performance of a comparable CMR drive and the effective cache size of `Device-Hybrid` on the HA-SMR drive. The effective cache size varies with the write size and workload skew but is independent of the zone utilization.

5.1.1 RQ1: Device-Hybrid versus Host-Hybrid

Figure 8 ([a,d] and [b,d]) compare `Device-Hybrid` and `Host-Hybrid` (respectively) showing the observed bandwidth for 4KB and 1MB block sizes under a 90/10 Zipfian workload.

Eviction Impact: Regardless of the block size, once cache eviction begins, `Host-Hybrid`'s throughput drops below that of `Device-Hybrid`. This degradation is a consequence of transferring eviction-related data across the SATA interface. Unlike `Device-Hybrid`, which performs merges internally on the device, `Host-Hybrid` must transfer both evicted data and data-zone contents over the host interface. The resulting "bus-tax" includes the SATA bus transfer time as well as additional, presently uncharacterized costs associated with host-device data transfer.

Cache size and DRAM cost: Since `Host-Hybrid` does not buffer any data that it receives, the predominant DRAM overhead comes from the metadata data structures. Recall that `Device-Hybrid`'s memory constraints restrict the number of translation entries to 191,172 (Table 3), constraining the effective cache size as a function of the write size. In contrast, the host can maintain a fixed cache size, since DRAM is plentiful and cheaper on the host, allowing for a larger translation table. On our 8TB HA-SMR drive, provisioning the same physical cache size as on `Device-Hybrid` (28GB), while accommodating 4KB blocks, requires a meager 56MB mapping table.

CPU cost: We measured the CPU utilization of the `Host-Hybrid` cache eviction thread during the `fio` tests, using the `pidstat` command, and observed that the *peak* CPU utilization during cache eviction is 7%.

Conclusion: Moving the STL to the host degrades perfor-

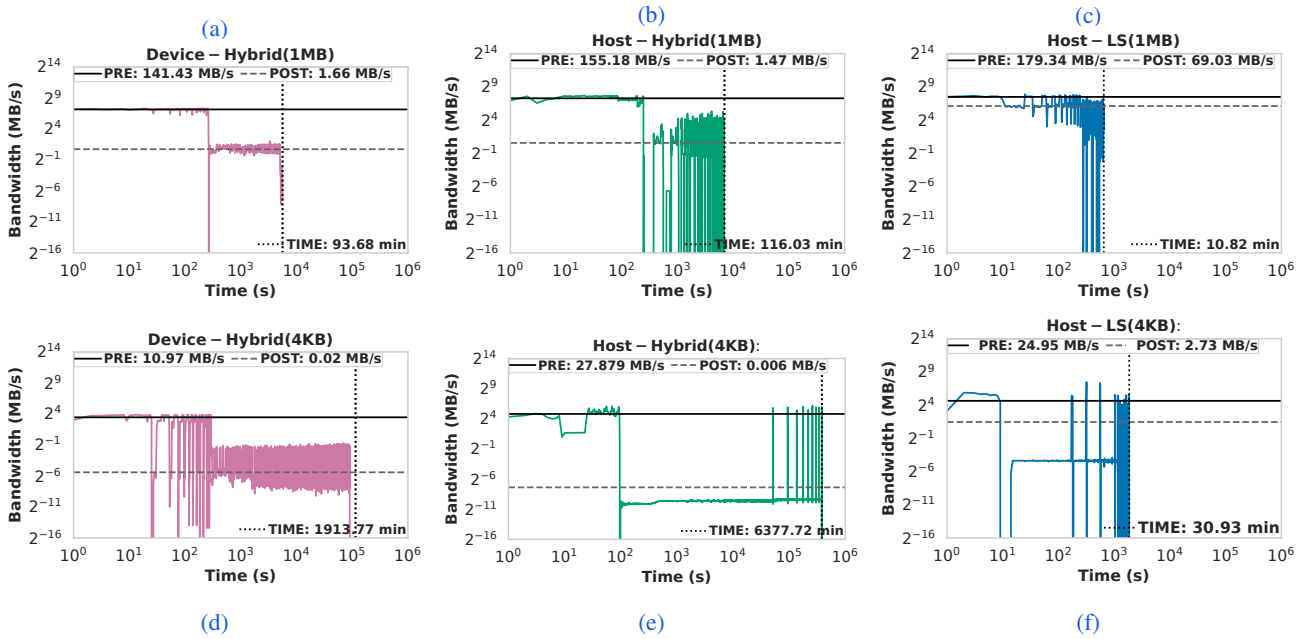


Figure 8: Bandwidth (log_2 MB/sec) vs time (log_{10} seconds) for random writes at 90% zone utilization, Zipfian workload ($\theta = 1.0962$ -> equivalent to 90/10 Pareto distribution). Top row shows the 1MB workload performance, the bottom row shows the 4KB workload performance. `Host-LS` outperforms both `Device-Hybrid` and `Host-Hybrid`. (PRE = pre-cleaning bandwidth; POST = post-cleaning bandwidth.)

mance due to the "bus tax" that occurs during eviction operations. However, it provides flexibility in cache size, debugging, and design that can improve application write performance.

5.1.2 Host-Hybrid versus Host-LS

Leveraging the flexibility of host management, we now evaluate the performance of `Host-LS`. Figure 8 shows the observed bandwidth for 4KB and 1MB blocks under a 90/10 Zipfian workload in `Device-Hybrid`, `Host-Hybrid`, and `Host-LS`.

Recall that, both `Device-Hybrid` and `Host-Hybrid` are insensitive to data zone utilization (Sections 2.4 and 3.5), due to their merge-based eviction strategy. In contrast, `Host-LS` is sensitive to zone utilization. A victim segment with higher occupancy contains fewer invalid blocks, yielding less free space upon reclamation. Thus, to generate a single free segment, `Host-LS` must clean a number of victim segments proportional to their utilization; the higher a zone's utilization, the fewer clean blocks are produced by cleaning it.

The hybrid architecture is also insensitive to workload skew (Sections 2.4 and 3.5). Conversely, `Host-LS` is highly sensitive to skew – in a good way; a more skewed workload naturally generates more invalid blocks within active zones. Since GC unstalls writes in direct proportion to the free space generated by cleaning a victim segment, bandwidth and overall performance improve significantly as skew increases.

Skewed workloads performance comparison: Figure 8 shows the result for our Zipfian 4KB and 1MB workloads. Comparing `Host-LS` to `Device-Hybrid` shows that `Host-LS` completes the 1MB workload in slightly over one-tenth the time that `Device-Hybrid` takes; the 4KB workload completes 60× faster. The average cleaning bandwidth of `Host-LS` (indicated by POST in Figure 8c and Figure 8f) is 100× greater on the 4KB workload and 41× greater on the 1MB workload. The tail latency report for this workload (Table 10) shows that, for the 1MB workload, compared to `Device-Hybrid`, `Host-LS` improves the 90^{th} percentile tail latency by ~8.7×, the 99^{th} and the 99.9^{th} percentiles improve by ~41.7× and ~34.9× respectively. Similarly, for the 4KB workload, it improves the 90^{th} percentile tail latency by ~446×, the 99^{th} and the 99.9^{th} percentiles by ~17.2× and ~17.8×.

	Block size	p50 (s)	p90 (s)	p99 (s)	p99.9 (s)
`Device-Hybrid`	1MB	0.21	15.21	200.11	221.88
`Host-Hybrid`		0.45	6.88	195.80	236.65
`Host-LS`		0.67	**1.74**	**4.80**	**6.35**
`Device-Hybrid`	4KB	0.14	13.37	89.42	96.02
`Host-Hybrid`		0.01	162.44	193.18	204.40
`Host-LS`		0.03	**0.03**	**5.20**	**5.38**

Table 10: Zipfian 90/10 tail latency: 1 MB writes, 45 GB burst, 90% zone utilization.

`Host-LS` Adversarial Workload: We run uniform random 4KB and 1MB write workloads with 90% zone utilization for `Device-Hybrid` and `Host-LS`. This workload is particularly challenging for `Host-LS` in that, as in the previous section, it uses 99% of the available capacity while nearly all filesystems reserve ~10% of a drive to avoid degenerate behavior. `Device-Hybrid` completes the 1MB workload in 50.4 minutes, while `Host-LS` takes 57.6 minutes. Similarly `Device-Hybrid` completes the 4KB workload in 74.2 minutes, while `Host-LS` takes 117.1 minutes. `Host-LS` performance improves as zone utilization decreases; results consis-

Arch.	Blk Size Dep.	Zone Util. Dep.	Skew Dep.	Burst Size Dep.	Disk Space Avail.	DRAM Req.	Peak CPU	Pwr Loss Risk	Bus Tax (Cln.)	Design Flex.
`Device-Hybrid`	✓(↑)	✗	✗	✓	✗	Lowest (2 MB)	–	✓	✗	✗
`Host-Hybrid`	✓(↑)	✗	✗	✓	✗	Low (56 MB)	7%	✓	✓	✓
`Host-LS`	✓(↑)	✓(↓)	✓(+)	✗	✓(+)	Highest (>512 MB)	13%	✓	✓	✓

Table 11: Comparative analysis of `Device-Hybrid`, `Host-Hybrid`, and `Host-LS`. Unlike fixed-cache designs, `Host-LS` is uniquely sensitive to total Disk Space Availability. Notation: ↑ = larger is better; ↓ = lower is better; + = more helps. (Blk = Block, Util = Utilization, Dep = Dependence, Avail = Availability, Req = Requirement, Pwr = Power, Cln = Cleaning, Flex = Flexibility.)

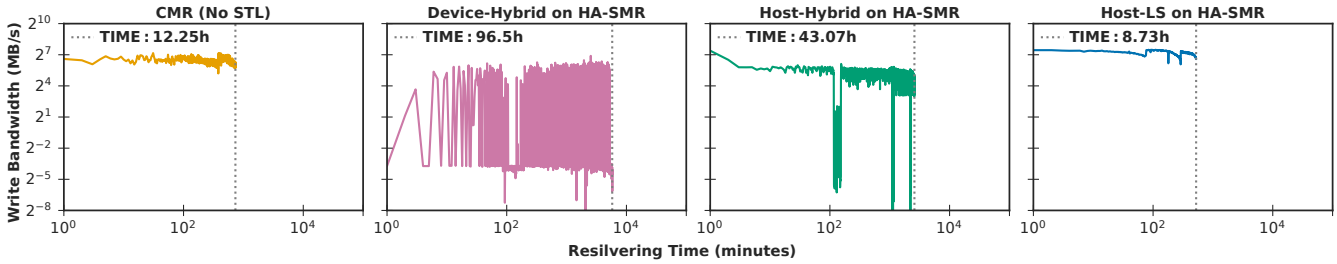


Figure 9: **ZFS Resilvering bandwidth (log_2 MB/sec) vs time (log_{10} minutes)** on comparable magnetic recording media - `Host-LS` writes random resilvered writes sequentially at ∼full bandwidth (∼140MB/sec)

tent with the original LFS work [5]. The performance also improves when more zones are reserved for GC writes. Such over-provisioning is the industry standard for improving GC performance under extreme conditions [20, 49, 50].

We do not discuss `Host-Hybrid`, as running it with the same cache size as `Device-Hybrid` is always much slower due to SATA overhead, and running it so that it accommodates the burst does not provide additional insight.

DRAM cost of `Host-LS`: `Host-LS` requires more DRAM on the host than `Host-Hybrid` does, because it maintains larger translation tables. `Host-LS`'s extent-based translation tables make reducing this size feasible. Without extent-based translation, we would map every 4KB block, requiring 14.5GB for the translation table for an 8TB SMR.

`Host-LS` uses extent-based translation tables with 64-bytes per extent. Sequential writes need only one extent for the entire drive, while 100% random 4KB rewrites would require 128GB — an unlikely scenario on spinning media. In practice, larger random writes spanning multiple 4KB pages need fewer extents. With 1MB average write sizes, the mapping table consumes only ~512MB, If the table grows, the OS can use demand paging. However, even at 512MB, this remains an order of magnitude larger than `Host-Hybrid`'s 56MB translation table.

CPU cost of `Host-LS`: We measured the CPU utilization of the `Host-LS` GC and flusher threads during the `fio` tests, using the `pidstat` command. We observed *peak* CPU utilization of the GC thread of 13% and averages of ~2%. The peak CPU utilization during eviction in HM-hybrid was 7%.

Summary: The comparison of all the 3 STLs is summarized in Table 11. The hybrid cache architecture performs well when the active duty cycle fits within the cache, and the write size is large. However performance drops precipitously when eviction is triggered. Extra data written as a result of eviction is proportional to the number of data zones mapped in the cache. These factors can rarely be controlled a priori.

On the other hand, while a device-based log-structured STL is economically infeasible, a host-based implementation is both feasible and advantageous. The performance drops for `Host-LS` during GC are gentler than those in the hybrid architecture. `Host-LS` consistently outperforms `Device-Hybrid` for skewed workloads with 90^{th}+ percentile tail latency improved by at least ~8.7× for the 90/10 Zipfian distributions.

5.2 ZFS Resilvering

ZFS [17] is a copy-on-write filesystem often used on Network-Attached Storage (NAS). It manages devices using RAIDz with variable-size parity for fault tolerance. When a drive fails, rather than performing conventional block-level RAID reconstruction, ZFS reconstructs the data through a "resilvering" process that replays filesystem transactions. Live data is reconstructed using the locations recorded in those transactions. Because ZFS is a copy-on-write filesystem, these locations are scattered, resulting in random writes on SMR drives.

Western Digital was sued, when they replaced CMR drives with SMR drives on their NAS [2, 51, 52], alleging that RAID-Z resilvering times became functionally unacceptable [53]. The NAS community blamed this poor performance on SMR's physical characteristics [54–57].

We investigated whether the root cause is the SMR medium or the STL, using the setup described in the lawsuit [2,58]. We created a ZFS RAID-z pool of 4 drives and populated it to 60% capacity (~5.28TB per disk). We emulated drive failure by

replacing one drive with an empty drive to trigger resilvering 5.28 TB of data. As before, we compared Device-Hybrid to Host-Hybrid and Host-LS on HA-SMR to evaluate STL placement and architecture on the same physical medium.

We monitored I/O performance during resilvering using the "zpool iostat" utility. Figure 9 illustrates the observed resilvering bandwidth over the duration of the operation. Our analysis reveals that the resilvering workload consists of both sequential and random writes of varying sizes between 32 KB and 1 MB: ~63% of the random writes were 320 KB in size, ~30% of the writes were 704 KB, and the remaining 7% were of various different sizes. The workload was heavily skewed: less than 0.4% of the total zones (110 out of 29,808) account for 10% of all write traffic.

Crucially, all resilvering experiments were performed on a freshly initialized, empty drive to ensure consistent baseline conditions. On Host-LS, all writes are sequential and obtain an average bandwidth of ~143 MB/second, unimpeded by cleaning. In contrast, all writes to the CMR drive are random, and resilvering takes ~1.4× longer than Host-LS. Host-LS completes the workload ~11× faster than Device-Hybrid, and ~4.9× faster than Host-Hybrid using the same drive.

Random writes to Device-Hybrid and Host-Hybrid fill the on-disk cache, after which they compete with cache eviction. When we examine the kernel log output of Host-Hybrid, we see that it evicts ~17.6K cache zones resulting in ~35.7K data zones (8.74 TB) of additional rewriting.

Surprisingly, Host-Hybrid finished 2.2× faster than Device-Hybrid running on HA-SMR. This is due to Host-Hybrid using a fixed-size 28GB physical cache. Using the Host-Hybrid logs, we calculated the effective cache size for Device-Hybrid based on its constraints (maximum of 191,172 entries and 28 GB). Due to mapping limitations, Device-Hybrid's effective cache size fluctuated between ~7.1 GB and ~10.92 GB, with a median and average of ~9.6 GB. Device-Hybrid likely evicted ~68 K data zones resulting in ~16.6 TB of additional rewriting. Performance of Host-Hybrid is further boosted as its cache is located on the outer track providing higher in-cache writing bandwidth than Device-Hybrid with an inner-track located cache (Section 2.1). **The results show that ZFS's poor resilvering performance on SMR drives arises, not from the limitations of the medium itself, but from the suboptimal STL design imposed by the on-device placement.**

5.3 Parallel Linux Compilation

We evaluate the performance of parallel Linux kernel compilations on ext4, running on Device-Hybrid, Host-Hybrid, and Host-LS. A single kernel compilation generates files whose sizes range from 4KB to 1MB and generates ~11GB of random writes. This volume proved insufficient for a rigorous comparison, as the data fit entirely within the cache of the hybrid STLs and induced no garbage collection (GC) in Host-LS on an 8 TB device.

Experimental Setup and Workload: We stressed the file systems, by executing five parallel runs of make -j 12 using gnu parallel [59]. The five runs wrote a total of ~146GB, accessed 580 zones and used more than ~37M unique LBAs. Critically, only ~221K LBAs were accessed more than once, revealing a negligible overwrite ratio. Only 3.2GB of the workload consisted of zone-aligned sequential writes.

We gave the virtual devices for Host-LS and Host-Hybrid 150 GB (barely larger than the workload) and partitioned Device-Hybrid similarly. In this configuration, both hybrid STLs have a physical cache of 28 GB ~15% of the device size). This is significantly higher than the <1% cache ratio found on a real drive.

We configured Host-LS to initiate GC when free space falls below 7 GB, ~5% of the device size (a standard GC watermark [20, 60]), ensuring that the 146 GB triggers cleaning. As in Section 5.1, this scenario is adversarial to Host-LS. The constrained < 5% free space and low overwrite ratio result in reclaimed zones with high utilization. Both Host-LS and hybrid-cache STLs implement credit-based application writer unblocking during GC/cache eviction; the number of writes released is equal to the number of blocks freed. *The efficiency of this mechanism is inversely proportional to the zone utilization in Host-LS and to the number of data zones mapped per cache zone in hybrid architectures*. This specific workload has high zone utilization that hinders Host-LS GC, whereas a low data-zone-to-cache mapping ratio that favors hybrid-cache eviction. Consequently, the workload is cache-eviction friendly but remains adversarial to Host-LS GC efficiency.

Comparative Results: Despite these stringent requirements, Host-LS completed the workload in 87 minutes, ~1.2× faster than both Device-Hybrid (108 minutes) and Host-Hybrid (101 minutes). Host-Hybrid performed ~1.1× faster than Device-Hybrid.

Using the Host-Hybrid logs, we calculated the effective cache size for Device-Hybrid, which fluctuated between 3.1GB and 23.8GB, with a median of 18 GB and an average of 16 GB. In contrast, Host-Hybrid consistently had access to its full 28GB cache. Host-Hybrid triggered 430 cache zone evictions, involving 154 GB of additional writes to migrate data to 689 data zones. Host-LS cleaned 335 zones, migrating approximately 100 GB of data. Our analysis of GC segments confirms a high valid block count (mean and median of ~64 K), indicating 98% segment occupancy—a direct result of the workload's low overwrite ratio.

Despite the poorer Host-LS GC performance compared to the cache eviction performance of the hybrid architecture, Host-LS completes more quickly, because the hybrid cache architecture needs to evict repeatedly, even when there is free disk space available, while Host-LS performs FG-GC only when free space falls below 5%.

Notably, while all three architectures managed the same filesystem size, the hybrid architectures had access to an additional 28 GB of physical cache. When Host-LS is granted

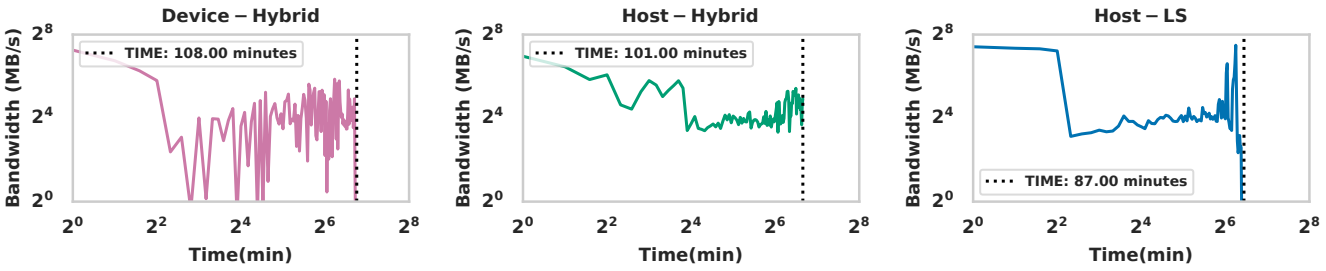


Figure 10: Bandwidth (log_2 MB/minutes) vs time (log_2 minutes) for 5 parallel runs of Linux kernel compilation on ext4 formatted on top of the device exported by `Device-Hybrid`, `Host-Hybrid`, and `Host-LS`. `Host-LS` outperforms both `Device-Hybrid` and `Host-Hybrid`.

equivalent over-provisioning space, it completes the workload with zero GC overhead.

5.4 Summary:

`Device-Hybrid` performs best when workloads exhibit strong spatial and temporal locality, allowing repeated updates to concentrate within the same zones over short periods. In contrast, `Host-Hybrid` performs better when random write bursts are large and write sizes are mixed.

`Host-LS` is highly effective for workloads in which random writes follow a power-law distribution and a small fraction of LBAs are updated repeatedly. Crucially, even with fewer overwrites and little or no data-zone modification locality, `Host-LS` outperforms the hybrid architecture. The flexibility of host-based STLs also enables them to almost always outperform `Device-Hybrid`.

6 Related Work

Log-structured management for SMR was first proposed by New et al. [61], with subsequent work exploring varied shingled-drive implementations [7, 20, 62–68].

In flash storage, Kim et al. [9] introduced the *Hybrid Log-Block* design to shrink the mapping table, later popularized as *BAST* [11] or *Hybrid-Mapping FTL* [10] with variants used in production [11, 69, 70]. However, log-block FTLs show poor performance for random writes lacking locality [71–75].

For SMR, Cassuto et al. [8] introduced the *hybrid-cache* design (labeled *Set-Associative Cache*). While Skylight [12] and others [15, 16] later reverse-engineered on-device STLs, we identified key eviction parameters absent in their work (see Table 7). `dmzoned` [76] implements a hybrid cache, but uses 1:1 zone mapping and inefficient cleaning. Similarly, `fstl` [14] provides a host-based framework but omits critical persistent metadata (translation/segment maps) and incurs overhead by cleaning in userspace.

A gap remains in practical systems: no prior kernel-level STL exports a block interface and implements both a log-structured architecture (`Host-LS`) and the hybrid-cache found on-device (`Host-Hybrid`). Moreover, performance trade-offs between these designs remain under-explored. While prior studies compared hybrid and log-structured FTLs via simulators [9, 72], they did not perform a controlled, empirical comparison of equivalent host vs. device-based architectures.

Existing work uses a log-structured merge tree for managing the metadata that helps look up the data on a device [65, 77–80]. However, these designs still require data reorganization for space reclamation, which is equivalent to log-structured cleaning. In contrast, `Host-LS` writes its metadata directly to the drive's native CMR zones, completely eliminating the need for specialized metadata management frameworks.

Other designs include VPC [81], which manages HA-SMR drives through cooperation between a host-managed virtual cache and the device's persistent cache. It assumes a log-structured on-device persistent cache. However, an empirical finding of our work is that the STL on this identical HA-SMR drive implements an overwrite-in-place cache.

For a broader SMR survey, see Khan et al. [82].

7 Conclusions

SMR drives have been relegated to sequential workloads due to perceived medium limitations. We demonstrate that poor performance stems from the design of the translation layer, not fundamental technology constraints. Host-based log-structured STLs can unlock SMR's potential. Our NAS deployment shows that `Host-LS` achieves a ~11× improvement in ZFS resilvering performance over `Device-Hybrid`.

Implementing a host-based STL is non-trivial. Despite the emergence of host-managed interfaces, no device manufacturer currently provides a host-based log-structured STL. We hope these results serve as a call to action for device manufacturers to provide host-based log-structured STLs. Doing so would enable cost-effective HM-SMR drives to outperform DM-SMR drives while approaching CMR performance, making SMR a mainstream storage technology.

Acknowledgements

We thank our shepherd, Peter Desnoyers, the anonymous reviewers, and Ethan Miller for their thoughtful feedback and suggestions.

References

[1] Jim Salter. Western Digital Gets sued for sneaking SMR disks into its NAS channel. *Ars Technica*, May 2020. [Link].

[2] Western Digital. Western Digital SMR ZFS lawsuit. Western Digital Lawsuit for Shipping Slower SMR Hard Drives Including WD Red NAS.

[3] TechWiser. SMR vs CMR drives: A comparison. *TechWiser*, 2021. [Link].

[4] Western Digital. Shingled Magnetic Recording + HelioSeal® Technology. [Link].

[5] Mendel Rosenblum and John K Ousterhout. The Design and Implementation of a Log-structured File System. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[6] Wikipedia contributors. Disk buffer — wikipedia, the free encyclopedia, 2026. [Link].

[7] M. H. Hajkazemi, M. Abdi, and P. Desnoyers. Minimizing Read Seeks for SMR Disk. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pages 146–155, Sep. 2018.

[8] Yuval Cassuto, Marco AA Sanvido, Cyril Guyot, David R Hall, and Zvonimir Z Bandic. Indirection Systems for Shingled-Recording Disk Drives. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–14. IEEE, 2010.

[9] Jesung Kim, Jong Min Kim, Sam H Noh, Sang Lyul Min, and Yookun Cho. A Space-Efficient Flash Translation Layer for Compactflash Systems. *IEEE Transactions on Consumer Electronics*, 48(2):366–375, 2002.

[10] Hunki Kwon, Eunsam Kim, Jongmoo Choi, Donghee Lee, and Sam H Noh. Janus-FTL: Finding the Optimal Point on the Spectrum between Page and Block Mapping Schemes. In *Proceedings of the tenth ACM International Conference on Embedded Software*, pages 169–178, 2010.

[11] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(3):18–es, 2007.

[12] Abutalib Aghayev, Mansour Shafaei, and Peter Desnoyers. Skylight— a Window on Shingled Disk Operation. *ACM Transactions on Storage (TOS)*, 11(4):16, 2015.

[13] Mansour Shafaei, Mohammad Hossein Hajkazemi, Peter Desnoyers, and Abutalib Aghayev. Modeling Drive-Managed SMR Performance. *ACM Transactions on Storage (TOS)*, 13(4):1–22, 2017.

[14] Mohammad Hossein Hajkazemi, Mania Abdi, Mansour Shafaei, and Peter Desnoyers. FSTL: A framework to Design and Explore Shingled Magnetic Recording Translation Layers. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 40–52. IEEE, 2018.

[15] Fenggang Wu, Ming-Chang Yang, Ziqi Fan, Baoquan Zhang, Xiongzi Ge, and David HC Du. Evaluating Host Aware SMR Drives. In *8th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 16)*, 2016.

[16] Fenggang Wu, Ziqi Fan, Ming-Chang Yang, Baoquan Zhang, Xiongzi Ge, and David HC Du. Performance Evaluation of Host Aware Shingled Magnetic Recording (HA-SMR) Drives. *IEEE Transactions on Computers*, 66(11):1932–1945, 2017.

[17] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The Zettabyte File System. In *Proc. of the 2nd Usenix Conference on File and Storage Technologies*, volume 215, 2003.

[18] Margo I Seltzer, Keith Bostic, Marshall K McKusick, Carl Staelin, et al. An Implementation of a Log-Structured File System for UNIX. In *USENIX Winter*, pages 307–326, 1993.

[19] Margo I Seltzer, Keith A Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, and Venkata N Padmanabhan. File system logging versus clustering: A performance comparison. In *USENIX*, pages 249–264, 1995.

[20] Changman Lee, Dongho Sim, Jooyoung Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 273–286, Santa Clara, CA, February 2015. USENIX Association.

[21] Jian Ouyang, Shiding Lin, Song Jiang, Zhenyu Hou, Yong Wang, and Yuanzheng Wang. SDF: Software-defined flash for web-scale internet storage systems. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 471–484, 2014.

[22] Laura Caulfield. Project Denali to define flexible SSDs for cloud-scale applications, mar 2018. Accessed: August 26, 2026.

[23] Abutalib Aghayev, Sage Weil, Greg Ganger, and George Amvrosiadis. Reconciling LSM-Trees with Modern Hard Drives using BlueFS. Technical report, Technical Report CMU Parallel Data Laboratory, 2019.

[24] Peng Wang, Guangyu Sun, Song Jiang, Jian Ouyang, Shiding Lin, Chen Zhang, and Jason Cong. An Efficient Design and Implementation of LSM-tree based Key-Value store on Open-Channel SSD. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[25] Minwoo Im, Kyungsu Kang, and Heonyoung Yeom. Accelerating RocksDB for Small-Zone ZNS SSDs by Parallel I/O Mechanism. In *Proceedings of the 23rd International Middleware Conference Industrial Track*, pages 15–21, 2022.

[26] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R Ganger, and George Amvrosiadis. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 689–703, 2021.

[27] Western Digital. Host-managed SMR in Dropbox's storage infrastructure, 2025. [Link].

[28] David Meyer. Four years of SMR storage: What we love and what's next. [Link], 2023.

[29] Eric Brewer, Lawrence Ying, Lawrence Greenfield, Robert Cypher, and Theodore T'so. Disks for data centers. 2016.

[30] Kim Kyu and Kim Tae-seok. Performance Evaluation of DRAM-Less NVMe SSD supporting HMB.

[31] Anshul Rana. Understanding DRAM vs DRAM-less SSDs and making the right purchase choice. [Link], July 2025.

[32] Kyusik Kim, Eunji Lee, and Taeseok Kim. HMB-SSD: Framework for Efficient Exploiting of the Host Memory Buffer in the NVMe SSD. *IEEE Access*, 7:150403–150411, 2019.

[33] Jens Axboe. Flexible I/O tester (fio), 2022. Software. [Link].

[34] Seagate Technology. Barracuda Product Manual, 2024. Available online.

[35] Aneesh Kumar KV, Mingming Cao, Jose R Santos, and Andreas Dilger. Ext4 block and inode allocator improvements. In *Linux Symposium*, volume 1, 2008.

[36] Young Jin Yu, Dong In Shin, Hyeonsang Eom, and Heon Young Yeom. NCQ vs. I/O scheduler: Preventing unexpected misbehaviors. *ACM Transactions on Storage (TOS)*, 6(1):1–37, 2010.

[37] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 228–243, 2013.

[38] Dai Qin, Angela Demke Brown, and Ashvin Goel. Reliable writeback for client-side flash caches. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 451–462, 2014.

[39] Manh Dat Tran, Lan Anh Nguyen, Hyung Tae Lee, Jeongyeup Paek, Sungrae Cho, and Yongseok Son. A survey on copy-on-write file systems. In *2024 15th International Conference on Information and Communication Technology Convergence (ICTC)*, pages 436–441, 2024.

[40] Kevin M Greenan, Darrell Long, Ethan L Miller, Thomas JE Schwarz, and Avani Wildani. Building flexible, fault-tolerant flash-based storage systems. 2009.

[41] Sandip Agarwala, Arnab Paul, Umakishore Ramachandran, and Karsten Schwan. e-SAFE: An extensible, secure and fault tolerant storage system. In *First International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007)*, pages 257–268. IEEE, 2007.

[42] Benny Van Houdt. Performance of Garbage Collection Algorithms for Flash-Based Solid State Drives with Hot/Cold Data. *Performance Evaluation*, 70(10):692–703, 2013.

[43] XYHR Haas and X Hu. The fundamental limit of flash random write performance: Understanding, analysis and performance modelling. *IBM Research Report, 2010/3/31, Tech. Rep*, 2010.

[44] Western Digital. WD Blue PC HDD Product Brief, 2024. Available online.

[45] Seagate Technology. Seagate Archive HDD Product Manual, 2015. Available online.

[46] Jiaxin Ou, Jiwu Shu, and Youyou Lu. A High Performance File System for Non-Volatile Main Memory. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 1–16, 2016.

[47] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M Swift. Aerie: Flexible File-System Interfaces to Storage-Class Memory. In *Proceedings of the Ninth*

European Conference on Computer Systems, pages 1–14, 2014.

[48] Yue Yang and Jianwen Zhu. Write Skew and Zipf Distribution: Evidence and Implications. *ACM transactions on Storage (TOS)*, 12(4):1–19, 2016.

[49] Jiacheng Zhang, Jiwu Shu, and Youyou Lu. ParaFS: A Log-Structured file system to exploit the internal parallelism of flash devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 87–100, Denver, CO, June 2016. USENIX Association.

[50] Jingpei Yang, Ned Plasson, Greg Gillis, Nisha Talagala, and Swaminathan Sundararaman. Don't Stack your Log on My Log. In *2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads ({INFLOW} 14)*, 2014.

[51] Nicholas Malone and Others v. Western Digital Corporation. Malone v. western digital corporation, case no. 5:20-cv-03584-nc. [Link], 2020. Class action complaint alleging that Western Digital's WD Red NAS hard drives with SMR technology were misrepresented and unsuitable for NAS/RAID use; filed May 29, 2020 in U.S. District Court for the Northern District of California.

[52] Sheldon Irving and Others. Notice of civil claim, sheldon irving v. western digital corporation and western digital canada corporation, court file no. vlc-s-s-205402. [Link], 2020. Filed May 22, 2020 in the Supreme Court of British Columbia, alleging that Western Digital misled Canadian purchasers by using SMR technology in WD Red NAS drives without proper disclosure.

[53] TrueNAS (iXsystems). WD red SMR drive compatibility with ZFS, August 2024. [Link].

[54] Finished resilvering SMR disk RAID-Z2 pool after ..., 2022. Reddit, r/DataHoarder. [Link].

[55] Aah please help my pool is resilvering for the ..., 2023. Reddit, r/TrueNAS. [Link].

[56] Mistakenly bought SMR... need advice on best path to swap out, 2022. Reddit, r/TrueNAS. [Link].

[57] SMR drives aka "archive drives" – a word of caution, 2016. Reddit, r/DataHoarder. [Link].

[58] Will Taillac. WD red SMR vs CMR tested—avoid red SMR, May 2020. ServeTheHome. [Link].

[59] Ole Tange. GNU Parallel 20150322 ('Hellwig'). *Zenodo*, 2015.

[60] Changwoo Min, Sang-Won Lee, and Young Ik Eom. Design and implementation of a log-structured file system for flash-based solid state drives. *IEEE Transactions on Computers*, 63(9):2215–2227, 2013.

[61] Richard MH New and Mason Lamar Williams. Log-Structured File System for Disk Drives with Shingled Writing, August 9 2011. US Patent 7,996,645.

[62] Peter Macko, Xiongzi Ge, J Kelley, D Slik, et al. SMORE: A Cold Data Object Store for SMR Drives. In *Proc. 34th Symp. Mass Storage Syst. Technol.(MSST)*, 2017.

[63] Abutalib Aghayev, Theodore Ts'o, Garth Gibson, and Peter Desnoyers. Evolving Ext4 for Shingled Disks. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 105–120, Santa Clara, CA, 2017. USENIX Association.

[64] Ting Yao, Jiguang Wan, Ping Huang, Yiwen Zhang, Zhiwen Liu, Changsheng Xie, and Xubin He. GearDB: a GC-free Key-Value Store on HM-SMR Drives with Gear Compaction. In *17th USENIX Conference on File and Storage Technologies (FAST) 19)*, pages 159–171, 2019.

[65] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R Ganger, and George Amvrosiadis. File Systems Unfit as Distributed Storage Backends: Lessons from 10 Years of Ceph Evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 353–369. ACM, 2019.

[66] Su Zhou, Erci Xu, Hao Wu, Yu Du, Jiacheng Cui, Wanyu Fu, Chang Liu, Yingni Wang, Wenbo Wang, Shouqu Sun, et al. {SMRSTORE}: A storage engine for cloud object storage on {HM-SMR} drives. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*, pages 395–408, 2023.

[67] Chao Jin, Wei-Ya Xi, Zhi-Yong Ching, Feng Huo, and Chun-Teck Lim. HiSMRfs: A high performance file system for shingled storage array. In *2014 30th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–6. IEEE, 2014.

[68] Christoph Hellwig, Hans Holmberg, and Damien Le Moal. Staying in the zone-retrofitting zoned storage into a scalable enterprise file system. In *Proceedings of the 16th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 30–37, 2025.

[69] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. LAST: locality-aware sector translation for NAND flash memory-based storage systems. *ACM SIGOPS Operating Systems Review*, 42(6):36–42, 2008.

[70] Hyunjin Cho, Dongkun Shin, and Young Ik Eom. KAST: K-associative sector translation for NAND flash memory in real-time systems. In *2009 Design, Automation & Test in Europe Conference & Exhibition*, pages 507–512. IEEE, 2009.

[71] Hyojun Kim and Seongjun Ahn. BPLRU: A buffer management scheme for improving random writes in flash storage. In *FAST*, volume 8, pages 1–14, 2008.

[72] Aayush Gupta, Youngjae Kim, and Bhuvan Urgaonkar. DFTL: a Flash Translation Layer Employing Demand-Based Selective Caching of Page-Level Address Mappings. *Acm Sigplan Notices*, 44(3):229–240, 2009.

[73] Sang-Phil Lim, Sang-Won Lee, and Bongki Moon. FASTer FTL for enterprise-class flash memory SSDs. In *2010 International Workshop on Storage Network Architecture and Parallel I/Os*, pages 3–12, 2010.

[74] Shengzhe Wang, Zihang Lin, Suzhen Wu, Hong Jiang, Jie Zhang, and Bo Mao. LearnedFTL: A learning-based page-level FTL for reducing double reads in flash-based SSDs. In *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 616–629. IEEE, 2024.

[75] Tae-Sun Chung, Dong-Joo Park, Sangwon Park, Dong-Ho Lee, Sang-Won Lee, and Ha-Joo Song. A survey of flash translation layer. *Journal of Systems Architecture*, 55(5-6):332–343, 2009.

[76] The Linux Kernel Documentation. `dm-zoned` — the linux kernel documentation, 2025. [Link].

[77] Shiyu Wang, Zhihao Zhang, and Yiming Zhang. OnionDisk: A log-structured write-optimal virtual block device. In *Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 136–143, 2024.

[78] Ceph Documentation. Zoned storage, 2024. [Link].

[79] Pure Storage, Inc. The Meta Advantage: Scaling I/O Performance with FlashBlade//S and FlashBlade//E Architecture. White paper, Pure Storage, Inc., 2024. [Link].

[80] Cohesity, Inc. Cohesity SpanFS and SnapTree: Next-Generation Distributed File System Architecture. White paper, Cohesity, Inc., 2024. [Link].

[81] Ming-Chang Yang, Yuan-Hao Chang, Fenggang Wu, Tei-Wei Kuo, and David HC Du. Virtual persistent cache: Remedy the long latency behavior of host-aware shingled magnetic recording drives. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 17–24. IEEE, 2017.

[82] Babar Khan and Andreas Koch. Reflecting on the Past 17 Years of Shingled Magnetic Recording for Insights Into Future Disk Transitions: A Survey. *ACM Transactions on Storage*, 21(3):1–50, 2025.