Research Article

Host-Guided Data Placement—Whose Job Is It Anyway?

Devashish R. Purandare[1], Peter Alvaro[1], Avani Wildani[1], Darrell D. E. Long[1], Ethan L. Miller[1]

1. University of California, Santa Cruz, United States

The increasing demand for SSDs coupled with scaling difficulties have left manufacturers scrambling for newer SSD interfaces which promise better performance and durability. While these interfaces reduce the rigidity of traditional abstractions, they require application or system-level changes that can impact the stability, security, and portability of systems. To make matters worse, such changes are rendered futile with introduction of next-generation interfaces. Further, there is little guidance on data placement and hardware specifics are often abstracted from the application layer. It is no surprise therefore that such interfaces have seen limited adoption, leaving behind a graveyard of experimental interfaces ranging from open-channel SSDs to zoned namespaces.

In this paper, we show how *shim layers* can to shield systems from changing hardware interfaces while benefiting from them. We present Reshim, an *all-userspace* shim layer that performs affinity and lifetime based data placement with *no change* to the operating system or the application. We demonstrate Reshim's ease of adoption with host-device coordination for three widely-used data-intensive systems: RocksDB, MongoDB, and CacheLib. With Reshim, these systems see 2–6 times higher write throughput, up to 6 times lower latency, and reduced write amplification compared to filesystems like f2fs. Reshim performs on par with application-specific backends like zenfs while offering more generality, lower latency, and richer data placement. With Reshim we demonstrate the value of isolating the complexity of the placement logic, allowing easy deployment of dynamic placement rules across several applications and storage interfaces.

1. Introduction

As the demand for datacenter SSDs soars, scaling flash keeps getting harder, with density increases adversely impacting the performance and device lifetime [1]. To make matters worse, nand-flash SSDs cannot perform in-place updates and have large erase units, requiring valid data relocation to free up space. These garbage collection operations impact performance and device lifetime due to internal data movement

 doi.org/10.32388/OJQTZ1

and write amplification. Even log-structured, append-only systems, which are designed to match device characteristics, are limited by traditional interfaces and fail to reach their full potential. With application logs, misaligned on filesystem logs, further misaligned on device logs, the "log-on-log problem" [2] causes performance and lifetime degradation with redundant garbage collection at multiple levels.

Two key placement strategies can help log-structured systems take full advantage of nand flash. First, a *lifetime-aware placement* that clusters data with a similar lifetime (i.e. whose creation and deletion have temporal locality) can minimize data relocation on erase, improving overall performance as well as device endurance. Second, an *affinity-aware placement*that clusters data produced by a single application together and places independent data streams on separate device resources can provide performance isolation, reducing tail latency. These strategies can (in principle) be realized in the current state of the art: flash manufacturers have introduced storage interfaces such as Zoned Namespace SSDs (zns) [3], and Flexible Data Placement (fdp) [4], which allow the host to direct data placement on the ssd via *placement directives* of a variety of forms, ranging from placement hints at one extreme to assuming responsibility for placement and garbage collection at the other.

But exactly which layer should handle such abstraction-breaking host-device coordination features? One answer is the application itself, which knows best about its own data access patterns. This approach is fragile and risky. Implementing device-specific optimization in application space requires specialized expertise, and these efforts are likely to be made obsolete in the face of API changes. The other answer is the filesystem, which suffers from the opposite problem. While implementing support for device-specific placement directives in the operating system would shield applications from complexity and provide much better generality and reuse, filesystems provide interfaces that are too narrow to take advantage of application-level semantics with regard to lifetime and affinity. The cost of generality means failing to take advantage of device characteristics.

In this work, we argue that the responsibility for exploiting emerging device-side placement directives should fall neither on the application programmer nor on the kernel developer, but rather on the users or systems that understand the *application-device mapping*. Typically, neither application developers nor filesystem developers are aware of the exact storage architecture their work will be deployed on, and hence cannot make effective optimization decisions. For every application, for every device, there is a space of possible placement directives that may be factored apart from the application and filesystem. Strategies to optimize placement based on lifetime and affinity (application properties) utilizing the features provided by devices (from hints to full management of regions) can be expressed in a separate *shim layer* that interposes between application and OS. Many configurations (indeed, the product of devices and applications) must be implemented, but these are easier to implement in isolation than within applications or operating systems.

By isolating complexity of each interface in a module decoupled from applications and fileystems, we can shield all other layers of the system from change.

Our shim layer approach opens up a generalizable interface that is application-agnostic, but can be optimized per application. We can isolate the complexity of varying interfaces and hint generation in pluggable userspace modules, allowing quick and easy changes to work with changing interfaces. Our library, Reshim presents a blueprint for dealing with the complexity of host-device coordination, allowing dynamic placement decisions without requiring application or filesystem rewrites. Reshim is a dynamic library that unlocks the benefits of host-device coordination without requiring application rewrites or custom filesystems. Our key insight is that shim layers can offer the *performance* of custom solutions, the *compatibility* of filesystems, while reducing the *complexity* of using modern interfaces.

Our contributions:

- We demonstrate how shim layers can provide both performance and compatibility while reducing application and filesystem complexity.
- We evaluate Reshim on three widely used applications (RocksDB, MongoDB, and CacheLib) across two types of interfaces (zns and kernel hints); more applications and interfaces than any previous effort.
- To our knowledge this is the first work to present a generalized theory of data placement, showcasing *affinity* and *lifetime* as the important parameters over temperature-based approaches of the past.
- We deploy Reshim with heuristic and learning-based hints, showcasing extensibility which is difficult to achieve in filesystems or applications.
- Reshim is fast: we see 2–6 times higher write throughput, up to 6 times lower latency, and reduced garbage overhead over filesystems and application backends.

Type of SSD	**Multi-Stream**	**Open-Channel**	**Zoned Namespaces**	**Flexible Data Placement**
Introduced	2016	2017	2021	2022
Linux Kernel Support	Deprecated	Deprecated	✓	—
Hint Interface	fcntl()	liblightnvm	libzbd,libnvme	libnvme
Filesystems	—	—	f2fs, btrfs	—
Applications	AutoStream	RocksDB	RocksDB	Cachelib, RocksDB

Table 1. The history of interfaces for host-managed data placements provides a bleak picture, with new interfaces demonstrating 1–2 applications before being deprecated for lack of use.

2. Benefits of host-guided data placement

To motivate the need for techniques that offer wider compatibility than application-specific approaches while maintaining their performance, we show a simple experiment:

On a 4TB Western Digital Ultrastar DC ZN540 SSD [5], we performed sequential write tests with flexible IO tester (fio) [6] scaling up to 14 threads (the maximum open zones supported by the drive). We ran the tests on zonefs [7], a block-layer representation of the zns interface, and f2fs [8], a flash-optimized filesystem. We made sure that the writes on zonefs went to different zones, while on f2fs we provided the *sequential* write and the *extreme* lifetime hints. To ensure parity, we used Direct IO, instructing f2fs to skip the buffer cache (zonefs does not support write buffering).

As we see in Figure 1, a lightweight mapping layer with the right hints (map each file to a separate zone) can provide full device throughput, while f2fs is limited to 30–50% of the bandwidth. We analyzed the results in perf[9] and break them down by the cpu cycles spent by our test in each scenario. Despite O_DIRECT, f2fs needs to cache writes to map them to various segments. Such caches result in frequent internal data structure updates and syncs. This overhead adds up with in-kernel locking operations resulting in f2fs spending more time in sync (34.43%) than in write calls (12.72%). The added overhead results in 2–3 $\times$ higher latency and lower throughput. While in zonefs, since the filesystem is aware that these are writes to separate zones, it does not need to cache or sync to the device, utilizing the full bandwidth of the device buffer.

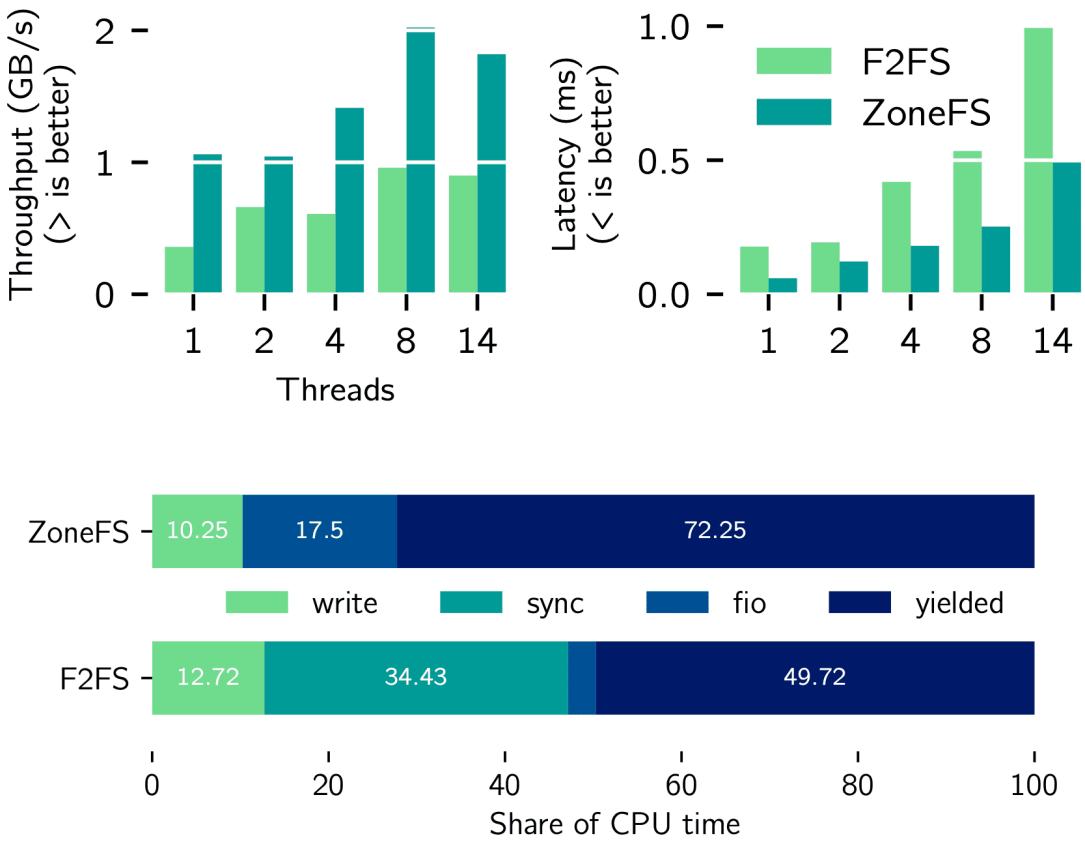

Figure 1. Writes to zonefs can get the full bandwidth while f2fs sees degradation in both latency and throughput.

However, it is important to note that f2fs is a full-fledged filesystem while zonefs is closer to a raw block device. Such overhead imposed by filesystems can be greatly reduced by designs which are aware of the log-structured nature of incoming data as well as the underlying device.

So how can a host help in such coordination?

2.1. Picking the right layer for coordination

Traditionally, support for host involvement has been performed either at the application or the filesystem. We argue that both these approaches have major limitations.

Why not rewrite the application?

1. *Rewriting applications is expensive:* Rewriting applications for a specific architecture requires significant engineering effort. For instance, the zenfs[10] project, which optimizes RocksDB for zns, is a multi-year

doi.org/10.32388/OJQTZ1

effort with thousands of lines of code. It has been unusable since February 2024 due to RocksDB updates. Such efforts need to be redone with every new interface.

2. *Applications are storage-unaware:* Applications typically use the file interface and are abstracted from the nature of storage. They are unaware of system usage or other applications, resulting in any efforts of resource acquisition and hinting being ineffective.
3. *Modifying applications limits layering and portability:* Even if an application is customized end-to-end to use custom interfaces, it cannot then effectively address multiple types of devices. Breaking away from the file abstraction can hurt tooling operating on the data like replication and backup utilities.

What about a filesystem?

A filesystem can shield application from hardware interface changes. However, filesystems present their own set of limitations:

1. *Filesystems are application-unaware:* Mainstream filesystems are designed to be independent of the applications running on top of them and find it difficult to generate useful hints unless communicated by the application. Currently, no standard interfaces exist: for example, f2fs can utilize hints from the multi-stream SSD interface and map it to zones, but with three hint levels across all applications, it is insufficient.
2. *Filesystems are hard to modify:* As filesystems reside in the kernel, they are hard to modify and upgrade. Fixing zns-related bugs in f2fs, for instance, requires upgrading to a newer version of the Linux kernel, which is impractical for data centers as it requires migration and downtime and may cause issues with other applications. Adding complexity to the kernel can give rise to crashes and security vulnerabilities.
3. *Filesystems require broad compatibility:* Adapting a filesystem for zns, for instance, should not limit support for other types of SSDs. As we see in Figure 2, the increasing complexity can result in increased bugs in the kernel, reduced performance, and an increased attack surface.

While some of these issues could be addressed, for example, with a FUSE[11] filesystem, it would still require a per-architecture per-application filesystem to utilize host-device hint mechanisms fully. Such usage of FUSE would be similar to our proposed shim layer but with the added complexity of managing per-application filesystems. Further, repeated kernel-crossings communicating between modules, mappings, and hint generation could negate any performance benefits from modern storage interfaces.

Not only do extra interfaces to filesystems increase bugs, due to their in-kernel nature they add security vulnerabilities. Rather than adding bloat to the kernel, as we see in Figure 2, we can isolate the complexity in a small audit-able layer, greatly reducing attack surface while improving performance.

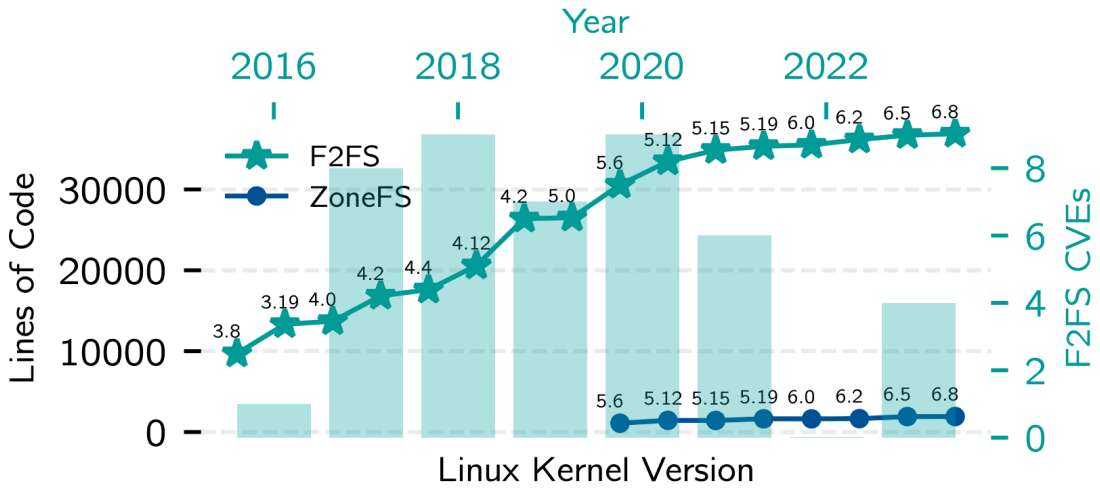

Figure 2. With diversifying hardware, the complexity of filesystems keeps increasing, increasing the attack surface in the Linux Kernel. Here, Reshim's approach of using zonefs can minimize the complexity, and hence the potential for bugs by using a significantly simpler filesystem.

The Middle Road: Shim Layers

A shim layer can abstract interface changes from the applications and filesystems while enabling low-overhead reconfiguration to get the benefits of modern SSDs. In this architecture, simplicity is maintained in the application and the filesystem, while the added complexity of hinting is isolated in a configurable layer—enabling low-cost, relatively-low-effort adoption.

With the goal to allow low-cost adoption of modern SSDs, an ideal shim layer should require:

1. *No changes the applications or operating system:* To simplify adoption, a shim layer should not need any changes to the application, any kernel modules, or reconfiguration of the system.
2. *Broad compatibility:* The shim layer should be able to work across different applications and utilities.
3. *Efficiency and effectiveness:* A shim layer should unlock performance benefits without adding more overhead than an tuned application or filesystem.
4. *Extensible:* The hint generation should be configurable, adding the ability to add custom logic, including systems that learn dynamically.

We built Reshim to stay true to these principles, and we demonstrate that such a layer is not only feasible; it can outperform other techniques. With Reshim, we propose the *addition of a layer* to break traditional layering abstractions, as it can *isolate changes across layers* without impacting the compatibility and portability of the application.

3. Reshim Architecture

Reshim is a dynamic library that allows interposition on application calls, modifying them if necessary, to embed placement directives or redirect them to different regions on an SSD. Reshim allows transparent hint injection or redirection of data based on the decision made by the placement engine. As Reshim has insight into the application and the storage architecture, it can generate more effective hints than applications or filesystems in isolation. Further, users can modify the hint generation easily without having to rewrite, reconfigure, or recompile the application or the filesystem.

Reshim performs three tasks necessary for host-device coordination, which act at the different layers of the modern stack—the application, the storage engine, and the device. Due to the differing philosophies in hardware, we designed each of these components to be customizable and replaceable with configuration changes. For instance, in the zns protocol, the device is partitioned into equal-sized append-only zones. The host is responsible for picking zones, managing buffers and garbage collection when it needs to free up space. This approach requires the *device manager* which is implemented in *reshim-engine*. Approaches like multi-stream and fdp on the other hand, simply require a directive on where to place the data, the device manager can be skipped for those devices.

For Reshim we present implementations for both of these approaches with demonstration on zns drives with the Western Digital Ultrastar DC ZN540 SSD and the ability to use kernel hint interfaces for multi-stream SSDs on filesystems that support these interfaces. Currently, fdp directives are only supported through NVMe commands, and since most applications do not implement direct NVMe writes (outside of exceptions like CacheLib [10]), we plan to look at fdpsupport as future work.

As seen in 3, Reshim has 3 major components:

1. Application Call Interception.
2. Data Placement Engine.
3. Device Manager: Backend.

In this section, we will go over each of these components.

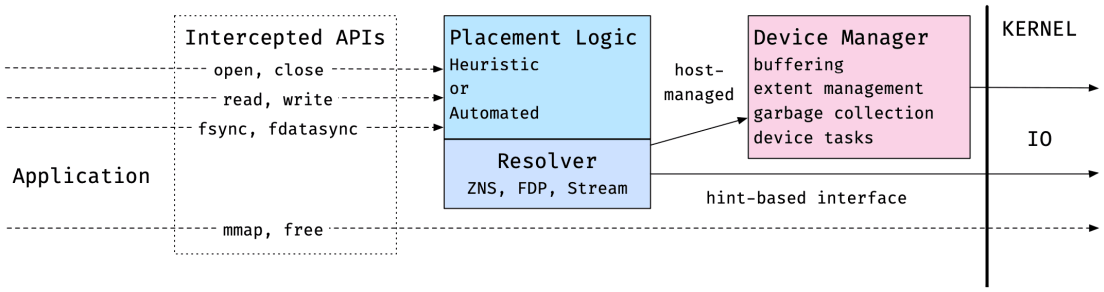

Figure 3. Simplified Reshim architecture: Reshim intercepts application calls, generates placement plans, resolves them to a particular protocol and finally manages data placement.

3.1. Application Call interception

Shim techniques have seen a renaissance as fast-changing hardware, special-purpose processing and changed memory hierarchies become common in our systems. It is no surprise therefore that we see a proliferation of techniques that allow shimming libc calls like eBPF[11], FUSE[12], WASM[13], virtual machines, and dynamic libraries[14] being widely deployed. While Reshim can be implemented with any of these techniques, we focused on LD_PRELOAD to eliminate added kernel crossings, allow us to implement an all-userspace system and minimize performance degradation. Dynamically linking with the application allows us to modify libc calls before they are sent through the kernel, allowing a simple IO path through the Kernel, while the complexity resides in userspace. LD_PRELOAD is used by projects ranging from custom allocators to debugging tools[15][16].

Table 2 presents the design space of shim techniques, and we chose the one with the best performance, and while its limitations could leave certain application ports out, the principles we discuss could be realized with other techniques or even a custom libc. Simplicity of implementation, userspace nature, and runtime linking make our approach easy to use, requiring *no recompilation, no changes to the application, and no changes to the system*. We discuss the limitations of our approach and alternatives in Section 3.4.

	Added Kernel Crossings	Static Linking	Kernel Changes	Per-Application Configuration	Shim Setup	Performance Degradation
LD_PRELOAD	–	–	–	Yes	Runtime	Minimal[1]
eBPF	Yes	Yes	Yes	Yes	In Advance	$\sim 10\times$ [17]
WASM	–	Yes	–	Yes	In Advance	$\sim 2.5\times$ [18]
FUSE	Yes	Yes	Yes	–	In Advance	$\sim 1.8\times$ [19]
PTrace	Yes	Yes	Yes	Yes	In Advance	$\sim 2.0\times$ [20]
syscall_intercept	–	–	–	Yes	Compile time	Significant[1]

Table 2. We chose the LD_PRELOAD approach for Reshim as it requires no change to the application, kernel, or special privileges, ensuring that data security and system stability is not affected.

[1] *LD_PRELOAD and syscall_intercept added* $\sim 2\,\mu s$ *and* $\sim 2\,ms$ *of userspace overhead per syscall in our tests respectively.*

Reshim is designed to be dynamically linked with the application at runtime using LD_PRELOAD, which allows libreshim to be loaded before other objects while executing a binary. Reshim uses this functionality to selectively override various libc calls, using Reshim calls instead. In hint-based interfaces Reshim injects placement directives received from the placement engine. Here hint-based approaches like fdp or multi-stream SSDs require just a placement identifier, while host-managed approaches like zns require the host to take over data placement, garbage collection, and resource management. For hint-based approaches, Reshim works on top of an existing storage system that supports hints, like f2fs. For host-managed approaches, Reshim implements its own device manager with *reshim-engine* and provides placement information per file. When used with *reshim-engine*, Reshim captures the content of calls the application issues, copying the data to its buffer cache before persisting it based on hints.

In both cases, Reshim needs to maintain a certain level of bookkeeping to supply directives per file. With System calls that use file descriptors which are ephemeral and issued per-session, Reshim needs to maintain a mapping of files to their descriptors to effectively understand which file a call is operating on. Since Reshim does not have a runtime independent of the application, it leverages specific calls to maintain state, spin up background tasks for garbage collection, and persist its metadata.

When working on top of a filesystem, Reshim issues the open() system call to the underlying filesystem, getting the kernel-issued file descriptor (fd), and then assigns a unique identifier (uuid) to the file, maintaining two mappings: path→uuid (the pathMap) and uuid→fd (the fdMap) in separate hash tables. The fdMap is automatically updated on each open() and close() call and never persisted, while the pathMap is updated on creates (seen through open(), rename(), and unlink()) and persisted on sync.

Data-intensive applications are often multi-threaded and can result in separate threads accessing the same file descriptor concurrently. This can quickly lead to contention on lookups, especially as Reshim is reactive, being invoked on an intercepted call. For this purpose, Reshim uses dashmap[21], a concurrent hashmap, for its data structures. On the first intercepted call, Reshim spins up multiple threads of its own for hint generation, metadata persistence, and placement logic depending on the configuration.

Reshim's interception varies depending on whether the interface is hint-based or host-managed. The logic is greatly simplified for hint-based interfaces:

3.1.1. Reshim in hint-based systems

In hint-based interfaces like multi-stream and fdp, Reshim does not need *reshim-engine*, so its interception requires limited state tracking, operating once per created file.

1. **First call:** On the first system call by the application, Reshim blocks and sets up necessary state, allocating memory for metadata tables, spinning up background threads for metadata persistence and hint generation. But first, Reshim checks for metadata to see if the application is resuming from a previous session with Reshim, and populates the necessary data structures.
2. **open():** Reshim checks if the path exists in the pathMap, if found, it forwards the call to the underlying filesystem, gets a fd, and adds an entry to the fdMap with the stored uuid before returning the fd. If not found, it provisions a new uuid and updates both maps and returns filesystem issued fd.

   If the file is opened in write mode, Reshim provides the path to the hint generation logic, and issues the necessary fcntl() and fadvise() calls with the hints returned by the placement engine.
3. **close():** removes the entry associated with the fd in fdMap and syncs persisted pathMap before forwarding close()to the filesystem.
4. **unlink():** Reshim updates both maps to remove the uuid on a successful return of the unlink call from the underlying filesystem.
5. **fsync():** With fsync and its variants (fdatasync, sync_file_range), Reshim syncs pathMap on a successful return from the underlying filesystem.
6. **rename():** On a successful rename, Reshim updates pathMap with the new path.

3.1.2. Reshim in host-managed systems

Host-managed interfaces like zns require deeper support than issuing just the required placement hint, and we implement *reshim-engine*, a lightweight device management engine to enable management of flash.

Some setup tasks are similar between hint-based and host-managed interfaces, but in the case of host-managed interfaces, Reshim needs to ensure a lot more than simply issuing the right hint advise system call. It needs to notify the *reshim-engine* to perform the needed task, not dissimilar to what a lightweight virtual filesystem would do.

1. **First call:** The first call is the same as hint-based systems, but in addition to restoring states, Reshim spins up *reshim-engine*, allowing it to allocate its metadata structures and write buffers for the buffering logic. While this can block for a few milliseconds, we observed that it provides significant performance benefits as no future calls invoke memory allocation.
2. **open():** Open performs the same tasks as described previously, however, in the host-managed case, instead of issuing a hint system call, Reshim requests a stream from the hint generation logic, mapping the current files to the stream as discussed in Sections 3.2 and 3.3.
3. **close():** removes the fd from fdMap and streamMap. Requests *reshim-engine* to flush data associated with the fd.
4. **unlink():** Removes all references to the file, requests *reshim-engine* to mark associated data for cleanup.
5. **write():** On the write call and its variants, Reshim forwards the buffer to *reshim-engine* with the uuid which handles writing the data to the device, on a successful return from *reshim-engine*, it returns success to the application.
6. **read():** On the read call and its variants, Reshim translates the read to uuid and offset before forwarding the request to *reshim-engine*, and returns the data it gets back.
7. **fsync():** On sync, Reshim persists its own mapping and forwards the request to *reshim-engine* to ensure the buffered data is persisted.
8. **rename():** is unchanged from hint-based logic.

In addition to the previously discussed calls Reshim needs to modify other calls to guarantee persistence, prevent extra allocation, and maintain a consistent state. Reshim performs the following actions on each of these calls:

- **fallocate():** In host-managed mode, these calls are suppressed as *reshim-engine* uses a *flush on sync* optimization.

- **ftruncate():** In host-managed mode, these calls only update the metadata as in-place updates on host-managed interfaces are not allowed unlike hint-based SSDs.
- **readahead():** Asks *reshim-engine* to readahead for the given range into its buffers.
- **mmap():** this call is unsupported for host-managed mode (outside read-only open), as host-managed devices cannot support the in-place updates made by the call. We forward these calls without change to a capable filesystem which runs alongside Reshim.

As we see with mmap(), Reshim allows specific files not to be intercepted using the hint mechanism. *reshim-engine* uses this for files that require in-place updates—manifest and configuration files, as well as special purpose files like LOCKfiles, device files, and procfs entries. This ensures that applications work with minimal modifications and do not get any unexpected errors, utilizing the default path for anything not implemented by *reshim-engine*. Since most log-structured systems need a small amount of in-place updatable files, *reshim-engine* maps them to in-place update-friendly conventional zones and puts the log-structured data, which makes up most of the data by volume, on sequential zones.

Once the calls are intercepted, Reshim requests a hint or a placement directive from the placement engine. Designed as a module this component is replaceable, allowing hand-tuned or automated hints across applications and interfaces.

3.2. Data Placement Engine

For a *good* placement plan, data relationships need to be considered based on two important properties:

- **Data Affinity:** Semantically-related writes grouped together maximize bandwidth through isolation.
- **Lifetime grouping:** Data that shares a common lifetime should be grouped together to minimize data movement to free up space.

For isolation, a good placement engine must separate independent write streams from across applications and within applications (such as data logs, write-ahead logs, checkpoints, and manifests) into separate groups. Such grouping will eliminate the interleaving of streams on device buffers and flash, improving performance due to device-level isolation and parallelism. Additionally, the system must group data by its *expected* lifetime, reducing the garbage collection overhead.

A major issue with implementing support for host-guided placement is the lack of usable Kernel abstractions. The RWH_HINT interface, leftover from multi-stream SSD days, allows four separate data streams: hot, cold, warm, and undefined. So far, only one application (RocksDB) uses them, and they are supported on a single filesystem (f2fs). These rigid interfaces mean effective placement requires custom filesystems or kernel bypass. With Reshim, we hope to change this, Reshim implements a flexible internal

hint representation, with resolvers for translating these hints into currently supported APIs, and ability to extend support to future APIs. *reshim-engine* demonstrates the utility of richer interfaces and can unlock the full potential of the placement logic (see Section 4).

3.2.1. Hints in Reshim

Recognizing that grouping of data needs to be more than just a handful of write-temperature streams, Reshim introduces two abstractions: *streams* and within them multiple *lifetime-groups*. A stream represents a single logical writer, which performs writes for a set of related data. This could be writing to a write-ahead log, a data log, or performing compaction of existing data. A lifetime-group on the other hand is a temporal subset of a stream, grouping blocks within a stream which are written together. Streams are application-specific, while lifetime groups are stream-specific.

As seen in Figure 4, an application could have different data streams—a write-ahead log, a data log in form of sorted string tables, and a separate writer for checkpoints. Thus, unlike traditional temperature-based hints, Reshim accounts for data for both, affinity and lifetime. We designed Reshim's internal hint representation to be simple and extensible, allowing easy translation into the available hint formats and any future changes.

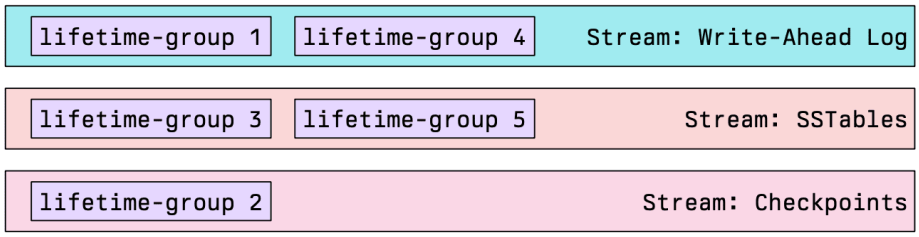

Figure 4. Reshim hints are organized into streams of writers based on affinity with groups of files based on lifetime.

Reshim's `get_hint()` API is called with a resolver, which can resolve hints into kernel hints, *reshim-engine* hints, or placement identifiers depending on the configuration (see Section 3.2.3) allowing it to work with existing hint systems as well as *reshim-engine*. Hint generation is the *only* part of Reshim that sees changes per application, these changes can be user-defined or automated. Streams can be added by users familiar with application semantics and workload or generated automatically based on observation. Resolvers are easy to write, they simply map this two-level hierarchy into what is suitable for the device interface.

doi.org/10.32388/OJQTZ1

3.2.2. Providing Hints: Users vs. Automation

For effective data placement, the placement engine needs to decide how to assign streams for newly created files, and how to assign a lifetime-group for each new block in the file. As we focus on log-structured applications, we can assign the lifetime group temporally, assigning a new group to blocks at fixed intervals of time or space. As cleanup of the log happens, older data is compacted and written to the tail of the log, allowing adjacent groups to be cleaned up together.

Distinguishing streams of incoming data from an application is more complex. We demonstrate two techniques, tapping into heuristics and automation. For the heuristic approach, we analyze the operation and the documentation of a particular application and provide rules for sorting files. This approach is particularly effective in applications like RocksDB which provide distinct locations and file extensions for longer lived sorted string tables, and short-lived write-ahead log. If manual tuning is not preferred, hints can be automated. Reshim captures data such as paths, opening flags, and observed workload, and can train a variety of algorithms on it for prediction of a stream. In Section 4 we demonstrate this with batch-based mini KMeans [22] to dynamically pick streams based on observed workload.

In traditional systems such as applications or filesystems, such flexibility is difficult, as the placement logic will need to be in the kernel or within the application, and cannot be changed without recompiling the entire system.

3.2.3. Resolving Reshim hints into interface hints

To effectively use Reshim's hints across multiple interfaces, we implement resolvers which can translate the internal representation into an interface-specific hint. Translation to other formats can be lossy, especially in cases like the Linux kernel hint interface (based on multi-stream ssd), which provides four values to from.

- **Multi-stream:** Here, we ignore the placement groups, assigning each Reshim stream to kernel stream (hot, cold, warm, undefined), and in case of more than four writers, we map multiple streams to each hint level, maintaining a diminished level of write isolation.
- **Zones:** For zns, we implement *reshim-engine* to demonstrate the full potential of our approach. Reshim maps new zones to each stream and ensures that placement-groups are laid out sequentially within a zone.
- **Placement identifiers:** Here we discard streams, using fdp placement identifiers with Reshim's lifetime-groups. Since there is limited virtualization support for fdp and we could not acquire fdp drives, we plan to evaluate this in future work.

The advantage of decoupling hint interface from the application or filesystem is that should a new hardware interface or software API be introduced, the only change this logic would need is a new resolver to map internal representation to the new format. Armed with placement directives and intercepted calls, we can now forward host-managed device support to *reshim-engine*, another pluggable module to demonstrate what such a system can be capable of.

3.3. Device Manager—the reshim-engine

As host-managed devices assign more responsibility (and hence, more power) to the data management system on the host, we decided to implement a lightweight storage engine to demonstrate the strengths of Reshim's approach.

With zns, Reshim's storage backend, *reshim-engine* takes over management of flash, presenting a virtual filesystem interface to support reads and writes. Rather than writing wrappers for device interfaces, we use zonefs [7], a filesystem wrapper in the kernel for zoned devices, which allows using system calls rather than NVMe commands to manage zns SSDs. *reshim-engine* performs three main tasks: data layout, buffering, and garbage collection.

3.3.1. Data Layout

Inspired by zns' design, *reshim-engine*'s data layout maps streams to individual zones. To support arbitrary file sizes, *reshim-engine* implements an extent-based logic, allowing user-configurable extents from $4\ k$ to hundreds of megabytes with the only restriction that they be powers of 2. These extents are flushed to the stream-mapped zone. Once the zone fills up, a new zone is fetched from a list of free zones. *reshim-engine* uses allocate-on-flush, deferring block reservation until a close or sync command to optimize grouping further.

reshim-engine does not allow files to share chunks and only needs to maintain the zone and offset to locate a particular extent. *reshim-engine* maintains two data structures: the pathMap to map paths to uuid, and a fileMap which maintains a file's extents as a list of (zone, offset) tuples as seen in Figure 5. Reshim maintains an additional structure, the zoneMap, to maintain per-zone metadata to simplify garbage collection. *reshim-engine* minimizes per-zone metadata by relying on the hardware write pointer to keep track of offset, only maintaining a bitmap of the deletion status of all extents on a zone. Effectively, the zoneMap requires 12–32 bits of memory per zone depending on the extent size. Both zoneMap and fileMap are synced to persistent storage on common operations like sync and close.

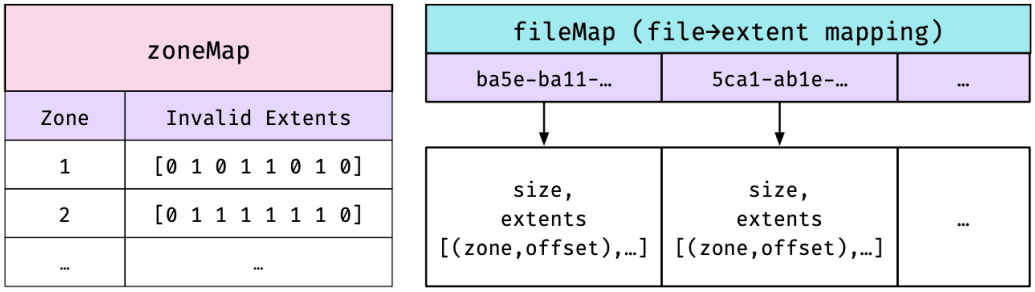

Figure 5.*reshim-engine* maintains two persistent data structures, a fileMap to maintain a mapping of file to extents, and a zoneMap to simplify garbage collection.

On zns devices, since the allocation of device buffers is not host-managed, *reshim-engine* needs to track active resources and keep them below the device limits by periodically finishing or closing zones. *reshim-engine* maintains a count of open zones and closes them as they fill up. Since there can be a dozen or more open zones at a time, this limit is rarely reached and typically happens when running multiple applications simultaneously.

3.3.2. Buffering

As zonefs does not support write buffering, we implement write buffering in *reshim-engine*. Due to the log-structured nature of applications handled by *reshim-engine*, we use a simplified write-through design. *reshim-engine* allocates a number of extent-sized buffers on boot, maintaining them in a free list and assigning a new buffer to each writable file. Once the buffer fills up, *reshim-engine* allocates blocks based on the corresponding stream and flushes the buffer to storage. While this approach adds a startup cost, we avoid allocating large amounts of memory during the rest of the execution improving overall performance.

3.3.3. Garbage Collection

reshim-engine implements lazy garbage collection, deferring data movement as long as possible. On each delete, *reshim-engine* updates the extent bitmap in the corresponding zone in the zoneMap, and if all extents are marked as deleted, it resets the zone, freeing up space without moving any data. Note that in most log-structured systems frequently updated streams like write-ahead logs see frequent deletes. Hence, *reshim-engine* practically doesn't need to move valid data as long as the device is not full. If available zones go below a user-configured threshold, Reshim iterates through zoneMapfiguring out the zone with the fewest valid extents and frees them up by moving the remaining extents to new zones in the same stream. Contiguous extents typically get invalidated together in compaction operations.

doi.org/10.32388/OJQTZ1

3.4. Limitations

Reshim inherits the limitations of LD_PRELOAD that we discussed in Section 3.1. So while Reshim cannot intercept statically-linked applications, in practice we observe that applications like RocksDB, Cachelib, and MongoDB still dynamically link with libc, and can be preloaded with Reshim. The bigger limitation comes with languages that do not use libc like Golang and Java, and hence systems written in these languages cannot be supported by LD_PRELOAD, and would need one of the other shim approaches. Further, it is not always easy to preload the library for complex client-server applications, as fork-execs and different coordinating processes may spawn processes that lose the intercepted functions. A filesystem approach or a custom C library can address this.

In this work we limited the scope of *reshim-engine* to log-structured append-only applications. Since all major data-intensive systems almost exclusively follow this pattern [23][24][25][26], we can adapt several applications to this interface, but applications with data structures that use in-place updates or mmap writes cannot utilize *reshim-engine* and will be passed through to the filesystem on conventional zones. Reshim can still issue hints for these writes if the underlying filesystem supports them.

Further, files stored by *reshim-engine* will not be visible to third party utilities like backup and copy unless they are preloaded with Reshim as well. *reshim-engine* focuses on data placement and is not a filesystem replacement as it does not implement filesystem operations like access control or locking. We support these to a limited extent on the random-write area on zns drives by using a conventional filesystem alongside *reshim-engine*. We could address these limitations by implementing Reshim using other shim techniques, however, as we discussed in Section 3.1, each comes with its own set of trade-offs. As we will see in Section 4, we prioritized simplicity and performance.

4. Evaluation

To demonstrate the benefits of Reshim, we present three different types of evaluation; we present three case studies with popular data management systems RocksDB [27], MongoDB [28], and CacheLib [10]. The first two are widely used log-structured storage backends, while CacheLib is a high-performance caching engine. These case studies demonstrate the ease of using Reshim and the performance benefits we get with each of the systems. Here we compare Reshim with filesystem approaches f2fs, and special-purpose systems (zenfs). Due to limited support for zns SSDs, we could not include other filesystems in our comparison.

To evaluate Reshim, we set up a test server with 64 core AMD EPYC 7452 system with 128 G of DRAM. We use Ubuntu 22.04 with Linux kernel 6.5, and 2 Western Digital Ultrastar DC ZN540 SSDs [5] each 4 T in size. We used the latest stable release of each system (with exceptions as seen in the box below) and used

unmodified bundled benchmarking tools. Between runs of each benchmark, we issued zone reset and NVMe format commands, and rebuilt the filesystems to ensure that each experiment had a fresh start. Since f2fs required an in-place updatable region for metadata, and Reshim uses it for LOCK files, we set up the region in the $4\ G$ random write space on the same drive.

zenfs was tested on RocksDB 8.9.1, while other systems were tested on RocksDB 9.6.1 as zenfs has been *effectively* abandoned and does not work on modern versions of RocksDB [29]. This may lead to small performance differences between versions. Incidentally, Reshim works with both versions of RocksDB, demonstrating further benefits of our approach.

A note about zenfs (*)

4.1. Case Study: RocksDB

Evaluating RocksDB provides several benefits: it is widely used, it supports a zns-specific backend and can provide write stream hints. As zenfs is specifically tuned for RocksDB on zns drives, it presents the gold standard for what performance highly tuned applications can have.

For our evaluation we ran 10 million operations, each with 20 keys and 400 values for fill workloads (sequential and random), reads (sequential and random), readwhilewriting, and overwrites. We used FIFO compaction as it provides natural temporal separation to each of the systems. For f2fs we forwarded the write hints provided by RocksDB.

We tested Reshim with two types of placement generation logic, the heuristic approach and automated approach. For the heuristic approach, we assigned separate streams to each of the logs—the Write-Ahead Log (wal) and the Sorted String Tables (sst). The placement groups were then based on the timestamps that the extents filled up in. For Automation (Reshim-Learn) we used batch-based mini KMeans clustering [22], a tuned online approach that predicts affiliation between a specified number of centroids. We then mapped the prediction of affinity to a particular centroid to a stream.

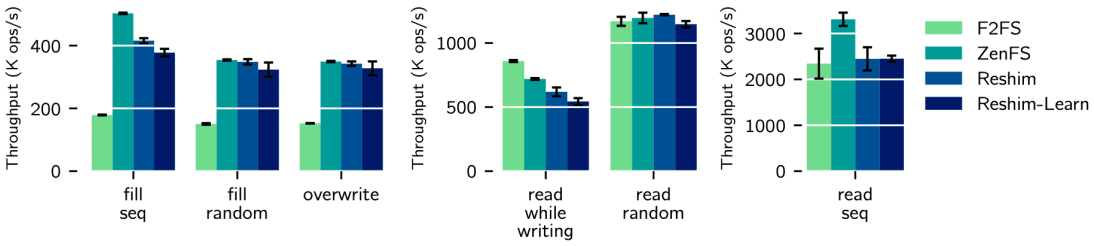

Figure 6. Throughput for fillrandom, overwrite, readwhilewriting and readrandom workloads of RocksDB's db_bench.

As seen in Figure 6, for inserts and updates, Reshim offers almost 2× improvement over f2fs and can match tailored approaches like zenfs. While zenfs offers slightly better performance in each case, it no longer works on modern RocksDB [29], and cannot be used with any other application. One of the main benefits of write isolation is the improvement in tail latency. Reshim's user-space nature ensures that compared to the high cost of kernel-crossings and persistence, the latency impact of the additional uReshim sees a small overhead on reads, and particularly readwhilewriting, due to the additional overhead of extent and address resolution in userspace.ser-space operations remains minimal.

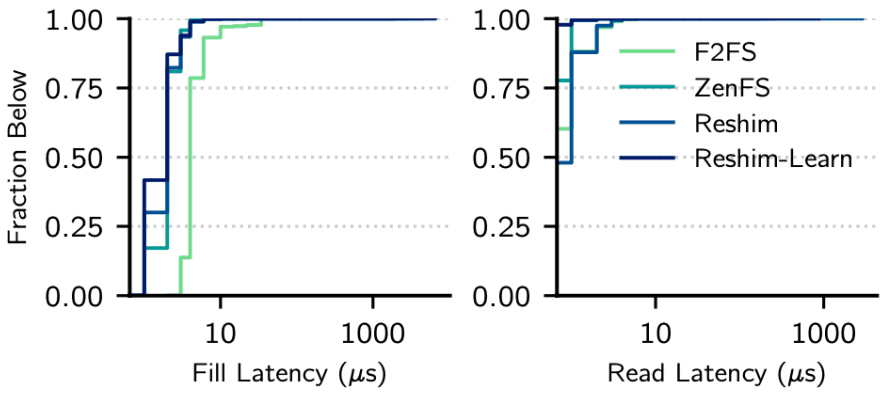

Figure 7. Reshim offers comparable latency to specifically tuned zenfs and improved latency over f2fs.

As seen in Figure 7, Reshim offers better tail latency over both f2fs and zenfs for random inserts and random reads. This is particularly evident at p99.99 (Reshim: ∼20 μs, zenfs: ∼100 μs), where Reshim can benefit from the filesystem optimizations in RocksDB, intercepting readahead() and pre-buffering data, a

doi.org/10.32388/OJQTZ1

filesystem optimization unavailable to zenfs. With automation, in Reshim-Learn we observe that it can match the performance of hand-tuned approaches, however it can suffer tail latency spikes due to the prediction engine. A faster approach like decision trees [30] can address this.

4.1.1. Why is Reshim faster?

Reshim provides intelligent hints discriminating between the frequent small writes of the *wal* and the large writes of Sorted String Tables (*sst*), utilizing separate device buffers to provide isolation. Digging deeper into the performance metrics as seen in Figure 8 for *f2fs*, the benchmark spends dozens of milliseconds to persist the *wal*. *wal* triggers locking in *f2fs*, packing incoming streams across its six logs, performing metadata updates, and allocating and freeing memory in the cache to support these operations. On the other hand, in Reshim the *wal* and the insert make up a relatively small chunk of CPU time, simply copying the buffer and periodically persisting it. The userspace overhead is also limited, with these functions accounting for less than 27% of total program time, similar to *zenfs* as opposed to the 44% of *f2fs*. Random reads (get operation) in Reshim are more efficient than *zenfs*, but add slight overhead over *f2fs* due to the extra steps in logical block resolution, accounting for 17.5% of the time as opposed to 23% on *zenfs*, which explains the latency spikes seen in Figure 7.

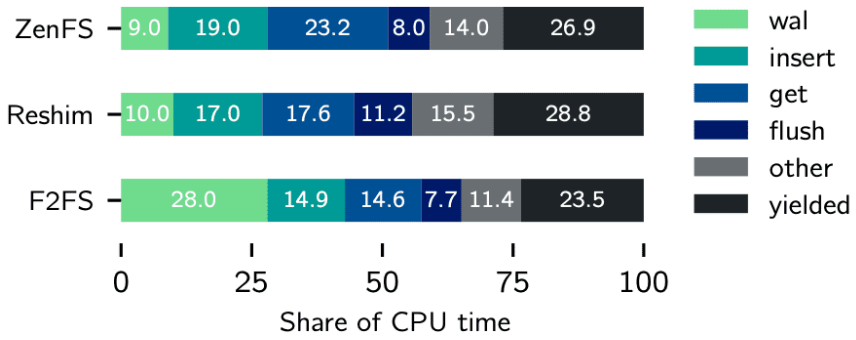

Figure 8. Breakdown of performance shows Reshim and zenfs with comparable write performance while f2fsstruggles to keep up with the frequent syncs of wal interfering with sst writes.

While RocksDB's composable nature makes it a great target for testing new interfaces, one of the main benefits of Reshim's approach is generality, where it works with more than just RocksDB, so we implemented placement for MongoDB.

doi.org/10.32388/OJQTZ1

4.2. Case Study: MongoDB

MongoDB's backend, WiredTiger gives the users a choice for its data structures: BTrees or Log-Structured Merge (LSM) trees. As BTrees require in-place updates, we focus on the LSM mode for Reshim. We tested WiredTiger's performance with a multithreaded insert and read test in wt_perf, to insert 10 Million entries in an LSM tree and then perform reads and updates in separate threads.

WiredTiger performs two streams of writes: logs and *ssts*. We reuse the logic from RocksDB for testing MongoDB, however a major limitation we ran into was supporting logs. WiredTiger uses *mmap()* writes for logging, which necessitate in-place and out-of-order updates that are not supported in *reshim-engine*. While *mmap()* databases are not recommended[31], this limitation handled by Reshim in its random-write region (with the lock files). For comparison, we look at *f2fs*, providing it filesystem hints through Reshim while *reshim-engine* uses Reshim's hints.

For WiredTiger, we observe a significant improvement in updates and reads in the LSM mode. As we see in Figure 9 Reshim with multi-threaded readers is more than 6$\times$ faster, with write-heavy workloads up to 3 $\times$ faster. These are enabled by the ability to separate logs that go to the random write area and streams of LSM files that stay on zones. Physical separation of mmap-writes in the dedicated random write area with its own buffers, while log-structured writes routed dedicated zone help improve the performance. We see similar dramatic improvements in latency in Figure 10, for read and update, Reshim's p50 is 31 μs and 36 μs respectively, compared to f2fs' 195 μs and 200 μs.

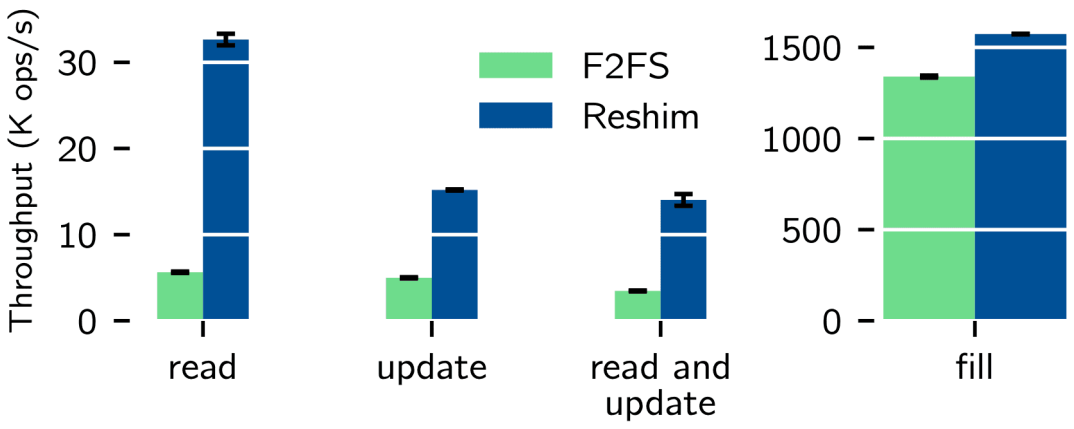

Figure 9. Due to physical separation of log and .lsm files to dedicated zones, Reshim can allow dramatically faster writes and updates in WiredTiger. The isolation across logs eliminates contention and improves both read and update latency.

doi.org/10.32388/OJQTZ1

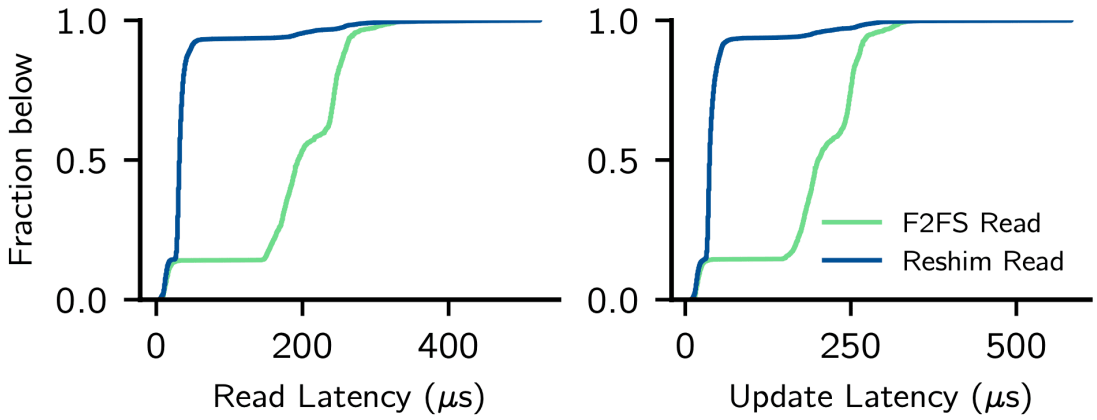

Figure 10. The isolation across logs eliminates contention and improves both read and update latency.

4.3. Case Study: CacheLib

Finally, to demonstrate caching workloads we adopted Meta's caching library CacheLib[32] to *reshim-engine*, which required no change in Reshim's logic, the only change required was configuration of CacheLib file format, which needs to be done at setup. As we see in Figure 11, Reshim offers lower latency for reads and writes (except maximum latency, a single spike required for our setup tasks), and improved throughput especially with increase in files. To perform this test we set up CacheLib with 128 M of cache in memory. We kept this cache minimal to accelerate spillover to flash, stress testing our systems. We then performed 10 million get and set operations on the cache. CacheLib supports two kinds of caches: one optimized for small objects and one for large. Again, we use a similar heuristic approach to map these to separate streams, relying on the log-structured nature to provide us temporal placement groups.

doi.org/10.32388/OJQTZ1

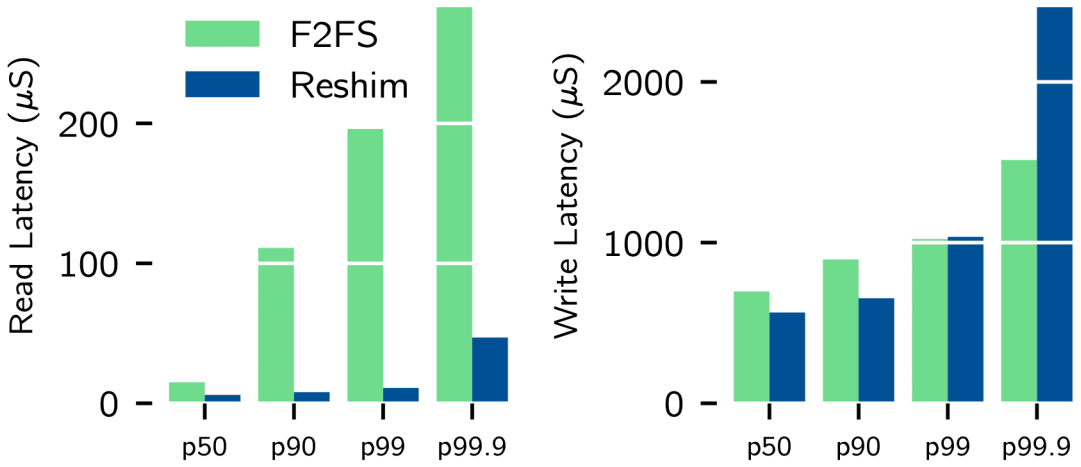

Figure 11. With CacheLib's cachebench tool, Reshim outperforms f2fs on throughput and read-write latency except the maximum write latency induced by initial setup

On CacheLib Reshim sees a small (8%) improvement in throughput $56.8\ K$Ops/s as compared to f2fs' $52.3\ K$Ops/s but a large reduction in read latency, particularly at p90 and above as seen in Figure 11. This is partially due to the highly optimized Navy engine, which uses optimizations io_uring[33], skipping a lot of filesystem overhead, and hence Reshim only sees a modest improvement. Reads are still improved, as incoming writes do not block read operations. Write latency on the other hand is on-par or higher (at extreme due to the setup cost).

4.4. Overhead

Finally, to show that Reshim maintains these performance properties while minimizing overhead, we measure the overhead in terms of data written and memory usage.

4.4.1. Write Amplification

Reduced garbage collection and improved grouping help reduce write amplification as data can be deleted together, avoiding the amplification caused by moving data. As there is no on-device garbage collection, *reshim-engine* sees a device write amplification factor of 1, similar to f2fs and zenfs. In the benchmarks we discussed, freeing up space in *reshim-engine* did not result in moving any data, as either a log or a sst zone was completely invalidated at the time of reclaim. As we see in Section 4.4.1, in our write-intensive experiments, Reshim frees up more space without moving any data as walblocks are freed up quickly,

doi.org/10.32388/OJQTZ1

allowing Reshim to erase without the need for relocating data. zenfs does not implement garbage collection, so it is not included in this table.

	Reshim	**f2fs**
Garbage Collection Calls	8	4
Data Moved	0	1059 MiB
Free Space at the End	48.2 GiB	33.2 GiB

Table 3. We ran an insert-heavy benchmark with updates, inserts, sequential, and random writes to compare f2fs and Reshim's garbage collection performance.

4.4.2. Memory Usage

Reshim in hint-only mode does not maintain any file data and uses no extra memory. However, *reshim-engine* needs to perform write buffering, which requires allocating memory depending on the number of open files. To speed up access, Reshim maintains all its maps in memory, however except fileMap, the rest are either fixed size or frequently cleaned up. In our experiments we used 32 M buffers with 2 write streams, totaling 64 M write buffers which made up most of the actively used memory. This size is smaller than the buffers maintained by filesystems in the kernel (f2fsmemory usage went up to 200 M in the same experiments). Profiling the memory using KDE heaptrack [34], *reshim-engine* used 73 M at its peak in the evaluation experiments.

$$\begin{aligned}&\text{The buffer cache (Extent Size} \times \text{Writeable Files)} +\\&\text{ZoneMap (32 bits} \times \text{Addressable Zones)} +\\&\text{Stream Hints (64 bits} \times \text{Hint Streams)} +\\&\text{File Data (64 bits} \times \text{Extents per File} \times \text{Open Files)}\end{aligned}$$

Memory in Reshim

5. Related Work

Prior work in using these interfaces involved modifying existing applications [35][36] or using custom file systems like zenfs and FStream [37][38]. Applications such as RocksDB, Percona server, and MyRocks have been adopted to the zns interface through the zenfs [39] plugin, the approach we evaluate against. WALTZ [40] optimizes further over zenfs, reducing the tail latency with the help of the zone append command.

Interposition approaches outside the file system have used either eBPF [41] or SPDK [42] as kernel-bypass mechanisms. Other approaches have used syscall_intercept [14] to overload system calls for persistent-memory programming by disassembling and patching binaries. The Reshim approach is similar to syscall_intercept but does not have the overhead of disassembling and patching binaries.

A few LD_PRELOAD-based filesystem prototypes exist, like PlasticFS [43] and AVFS [44], which allows peeking into compressed files. Goanna [45] implemented a filesystem through ptrace extensions, similar to the LD_PRELOAD technique in spirit. More recently, zIO [46] used user-space libraries using LD_PRELOAD to eliminate unnecessary copies of data, Reshim uses a similar interposition to redirect data. Several FUSE [19] filesystems have been implemented for optimized data placement. For instance, PLFS [47] optimizes for parallel checkpoints to map it optimally to underlying filesystems, similar to Reshim's mapping.

ZenFS vs Reshim

ZenFS presents an application-specific backend for zns SSDs, which is available as a RocksDB plugin. zenfs showcases how applications specifically tuned for new interfaces can perform. *reshim-engine* uses an extent-based design, much like zenfs but is more generalized as it can work with applications beyond RocksDB. Reshim utilizes a novel hint system that can be tuned per application, while zenfs depends on RocksDB for hints. Unfortunately zenfsseems largely abandoned with no updates since October 2023, and broken builds since February 2024[29], reemphasizing the need to separate data placement from application logic.

Reshim vs Filesystems

Filesystems like f2fs[8], and btrfs[48] offer zns-specific improvements and are similar to Reshim and an even broader application compatibility: the ability to perform in-place updates. f2fs is perhaps the best example of a zns-supporting filesystem, and we compare our approach to theirs throughout this work.

Persimmon[49] is an append-only fork of f2fs that requires no random-write area. Persimmon's performance is similar to f2fs, and it is tied to an older kernel version (5.18), and hence, we were unable to evaluate it. Performance in both Persimmon and f2fsis limited by their in-kernel nature, lack of application hints, and inability to use more than 3 data logs. Support for btrfson zns is limited, and we ran into several issues trying to run tests. *reshim-engine*, despite lacking filesystem features, can be used with applications that rely on a filesystem interface while offering a much better write performance and a comparable read performance.

Reshim uses interposition to inject hints and remap data if needed, no other system interposes between the application and the filesystem in such a way. Similar techniques have been implemented in different layers of the stack. Cloud Storage Acceleration Layer[50] enables the adoption of zns with clusters of varied storage and a host-based flash translation layer. However, such an approach is intended to be a solution with no application input and uses various types of storage to balance out random vs. sequential accesses. Reshim presents a filesystem-like solution that can scale from smartphones to large servers and be tuned per application.

6. Conclusion

The various efforts to introduce host-guided data placement present a sorry picture, we see a cycle of new interfaces being introduced, demonstrated, and deprecated within a couple of years. To change this paradigm, we need to decouple the complexity of data placement from the applications and filesystems, and make it easy to use these interfaces. Reshim represents an approach in this space, allowing filesystem and application development be unimpeded by changing interfaces, taking up the mantle to decide data placement and device management. It does this with rich hint interfaces and a composable structure that allows flexibility across applications, operating systems, and hardware protocols.

Hardware is in a turmoil. Compute, memory, and storage are moving over the stack making it challenging to adapt to these changes while maintaining compatibility. Ultimately, as is the case with Reshim, we believe that the way address the complexity is to isolate it in a dedicated layer that can be modified independent of other parts. This approach can speed up adoption of modern interfaces, simplify programming, offer improved performance, and allow broad application compatibility.

References

1. ^ *Grupp LM, Davis JD, Swanson S. "The Bleak Future of NAND Flash Memory." In: Proceedings of the 10th USE NIX Conference on File and Storage Technologies, FAST'12. San Jose, CA, USA: USENIX Association; 2012. p. 2.*

2. ^Yang J, Plasson N, Gillis G, Talagala N, Sundararaman S. "Don't Stack Your Log On My Log." In: 2nd Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW 14); 2014 Oct; Broomfield, CO. USENIX Association. Available from: https://www.usenix.org/conference/inflow14/workshop-program/presentation/yang.
3. ^Bjørling M. From Open-Channel SSDs to Zoned Namespaces. In: Vault '19, Boston, MA, February 2019. USENIX Association.
4. ^Sabol C, Stenfort R (2022). "Hyperscale Innovation: Flexible Data Placement Mode (FDP)". NVMe Flexible Data Placement. Available from: https://nvmexpress.org/wp-content/uploads/Hyperscale-Innovation-Flexible-Data-Placement-Mode-FDP.pdf.
5. a, bWestern Digital Corporation. Ultrastar DC ZN540 NVMe SSD [Internet]. 2024. Available from: https://www.westerndigital.com/en-ae/products/internal-drives/ultrastar-dc-zn540-nvme-ssd?sku=0TS2096. Accessed: 2024-10-22.
6. ^Axboe J. fio - flexible I/O tester rev. 3.30. Available from: https://fio.readthedocs.io/en/latest/fio_doc.html.
7. a, bLe Moal D, Yao T. "Zonefs: Mapping POSIX File System Interface to Raw Zoned Block Device Accesses." In: Vault '20; 2020 Feb; Santa Clara, CA. USENIX Association.
8. a, bLee C, Sim D, Hwang J, Cho S (2015). "F2FS: A New File System for Flash Storage". In: 13th USENIX Conference on File and Storage Technologies (FAST 15), pages 273–286.
9. ^De Melo AC. "The new Linux perf tools". In: Slides from Linux Kongress. 2010;18:1–42. Available from: http://oldvger.kernel.org/~acme/perf/lk2010-perf-acme.pdf.
10. a, bMeta Platforms, Inc. Cachelib. 2020. Available from: https://github.com/facebook/CacheLib. Available on GitHub.
11. ^Bijlani A, Ramachandran U. "Extension Framework for File Systems in User space". In: 2019 USENIX Annual Technical Conference (USENIX ATC 19). Renton, WA: USENIX Association; 2019. p. 121–134. Available from: https://www.usenix.org/conference/atc19/presentation/bijlani.
12. ^Mizusawa N, Nakazima K, Yamaguchi S. "Performance evaluation of file operations on OverlayFS". In: 2017 Fifth International Symposium on Computing and Networking (CANDAR). IEEE; 2017. p. 597–599.
13. ^Bytecode Alliance (2022). "WASI: WebAssembly System Interface". Online Documentation. Available from: https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-overview.md.
14. a, bIntel Corp. syscall_intercept. 2023. Available from: https://github.com/pmem/syscall_intercept. Retrieved: 2017-03-20.
15. ^Evans J. jemalloc [Internet]. 2005. Available from: https://github.com/jemalloc/jemalloc.

16. ^Seward J, Nethercote N, et al. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. 200 7. Available from: https://valgrind.org/.
17. ^Zheng Y, Yu T, Yang Y, Hu Y, Lai X, Quinn A. bpftime: userspace eBPF Runtime for Uprobe, Syscall and Kernel -User Interactions. December 2023. arXiv:2311.07923 [cs]. doi:10.48550/arXiv.2311.07923. Available from: http://arxiv.org/abs/2311.07923.
18. ^Jangda A, Powers B, Berger ED, Guha A. "Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code." In: 2019 USENIX Annual Technical Conference (USENIX ATC 19), Renton, WA: USENIX Association; 201 9. p. 107–120. Available from: https://www.usenix.org/conference/atc19/presentation/jangda.
19. a, bVangoor BKR, Tarasov V, Zadok E (2017). "To FUSE or not to FUSE: Performance of User-Space file system s". In: 15th USENIX Conference on File and Storage Technologies (FAST 17), pages 59–72.
20. ^Spillane RP, Wright CP, Sivathanu G, Zadok E. Rapid file system development using ptrace. In: Proceedings of the 2007 workshop on Experimental computer science. San Diego California: ACM; 2007. p. 22. doi:10.1145/128 1700.1281722.
21. ^Wejdenstal J. dashmap: A High-Performance Concurrent Hash Map for Rust [Internet]. 2023. Available from: https://github.com/xacrimon/dashmap. Accessed: September 21, 2023.
22. a, bSculley D. Web-scale k-means clustering. In: Proceedings of the 19th International Conference on World Wi de Web, WWW '10. New York, NY, USA: Association for Computing Machinery; 2010. p. 1177–1178. doi:10.1145/1 772690.1772862.
23. ^Calder B, Wang J, Ogus A, Nilakantan N, Skjolsvold A, McKelvie S, Xu Y, Srivastav S, Wu J, Simitci H, Haridas J, Uddaraju C, Khatri H, Edwards A, Bedekar V, Mainali S, Abbasi R, Agarwal A, Haq MF, Haq MI, Bhardwaj D, D ayanand S, Adusumilli A, McNett M, Sankaran S, Manivannan K, Rigas L. "Windows Azure Storage: a highly a vailable cloud storage service with strong consistency." In: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, SOSP '11, New York, NY, USA: Association for Computing Machinery; 2011. p. 1 43–157. doi:10.1145/2043556.2043571.
24. ^Bornholt J, Joshi R, Astrauskas V, Cully B, Kragl B, Markle S, Sauri K, Schleit D, Slatton G, Tasiran S, Van Geffe n J, Warfield A. Using lightweight formal methods to validate a key-value storage node in Amazon S3. In: SOS P 2021; 2021. Available from: https://www.amazon.science/publications/using-lightweight-formal-methods -to-validate-a-key-value-storage-node-in-amazon-s3.
25. ^Edara P, Forbes J, Li B. "Vortex: A Stream-oriented Storage Engine For Big Data Analytics". In: SIGMOD; 202 4.
26. ^Matsunobu Y, Dong S, Lee H (2020). "MyRocks: LSM-tree database storage engine serving Facebook's social graph". Proc. VLDB Endow.. **13** (12): 3217–3230. doi:10.14778/3415478.3415546.
27. ^Facebook (2013). "RocksDB". Available on GitHub. https://github.com/facebook/rocksdb.

28. ^MongoDB, Inc. MongoDB [Internet]. 2009 [cited 2023]. Available from: https://www.mongodb.com/.

29. a, b, cbpan2020 (2024). "There is an error when i am compilling rocksDB version above 8.10.0 with zenFS 2.1.4". ZenFS GitHub Issue \#288. Available from: https://github.com/westerndigitalcorporation/zenfs/issues/288.

30. ^Cook B. "An unexpected discovery: Automated reasoning often makes systems more efficient and easier to maintain". AWS Security Blog. Available from: https://aws.amazon.com/blogs/security/an-unexpected-discovery-automated-reasoning-often-makes-systems-more-efficient-and-easier-to-maintain/. Accessed: 2024-10-22.

31. ^Crotty A, Leis V, Pavlo A (2022). "Are You Sure You Want to Use MMAP in Your Database Management System?" In: {CIDR} 2022, Conference on Innovative Data Systems Research.

32. ^Berg B, Berger DS, McAllister S, Grosof I, Gunasekar S, Lu J, Uhlar M, Carrig J, Beckmann N, Harchol-Balter M, Ganger GR. "The CacheLib Caching Engine: Design and Experiences at Scale." In: 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). USENIX Association; 2020 Nov. p. 753-768. Available from: https://www.usenix.org/conference/osdi20/presentation/berg.

33. ^Axboe J. "Efficient IO with io_uring". Available from: https://kernel.dk/io_uring.pdf.

34. ^**KDE**. KDE heaptrack [Internet]. 2024. Available from: https://github.com/KDE/heaptrack. Last Accessed: January 15 2024.

35. ^Wang P, Sun G, Jiang S, Ouyang J, Lin S, Zhang C, Cong J (2014). "An efficient design and implementation of LSM-tree based key-value store on open-channel SSD". Proceedings of the Ninth European Conference on Computer Systems. 2014: 1–14.

36. ^Zhang J, Lu Y, Shu J, Qin X (2017). "FlashKV: Accelerating KV performance with open-channel SSDs". ACM Transactions on Embedded Computing Systems (TECS). **16** (5s): 1–19.

37. ^Oh M, Yoo S, Choi J, Park J, Choi C-E (2023). "ZenFS+: Nurturing Performance and Isolation to ZenFS". IEEE Access. **11**: 26344–26357. doi:10.1109/ACCESS.2023.3257354. Link.

38. ^Rho E, Joshi K, Shin SU, Shetty NJ, Hwang J, Cho S, Lee DD, Jeong J. "FStream: Managing Flash Streams in the File System." In: 16th USENIX Conference on File and Storage Technologies (FAST 18), Oakland, CA: USENIX Association; 2018. p. 257-264. Available from: https://www.usenix.org/conference/fast18/presentation/rho.

39. ^Bjørling M, Aghayev A, Holmberg H, Ramesh A, Le Moal D, Ganger GR, Amvrosiadis G. "ZNS: Avoiding the Block Interface Tax for Flash-based SSDs." In: 2021 USENIX Annual Technical Conference (USENIX ATC 21); 2021. p. 689-703.

40. ^Lee J, Kim D, Lee JW. "WALTZ: Leveraging Zone Append to Tighten the Tail Latency of LSM Tree on ZNS SSD". Proc. VLDB Endow.. **16** (11): 2884–2896, 2023. doi:10.14778/3611479.3611495.

41. ^Zhong Y, Li H, Wu YJ, Zarkadas I, Tao J, Mesterhazy E, Makris M, Yang J, Tai A, Stutsman R, Cidon A. "XRP: In-Kernel Storage Functions with eBPF." In: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), Carlsbad, CA: USENIX Association; 2022. p. 375-393. Available from: https://www.usenix.org/conference/osdi22/presentation/zhong.

42. ^Yang Z, Harris JR, Walker B, Verkamp D, Liu C, Chang C, Cao G, Stern J, Verma V, Paul LE. "SPDK: A development kit to build high performance storage applications." In: 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom). IEEE; 2017. p. 154-161.

43. ^Miller P. PlasticFS: A GNU Project [Internet]. 2012 [cited 2023]. Available from: https://plasticfs.sourceforge.net/. Last updated: 2012.

44. ^Hoffmann R, Szeredi M. AVFS: A Virtual Filesystem [Internet]. 2001. Available from: https://avf.sourceforge.net/. Last accessed: August 29, 2023.

45. ^Spillane RP, Gaikwad S, Zadok E, Wright CP, Chinni M. Enabling transactional file access via lightweight kernel extensions. In: Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09). San Francisco, CA: USENIX Association; 2009. p. 29-42.

46. ^Stamler T, Hwang D, Raybuck A, Zhang W, Peter S. "zIO: Accelerating IO-Intensive Applications with Transparent Zero-Copy IO." In: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22), Carlsbad, CA: USENIX Association; 2022. p. 431-445. Available from: https://www.usenix.org/conference/osdi22/presentation/stamler.

47. ^Bent J, Gibson G, Grider G, McClelland B, Nowoczynski P, Nunez J, Polte M, Wingate M. "PLFS: a checkpoint filesystem for parallel applications." Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis. 2009: 1-12. doi:10.1145/1654059.1654081.

48. ^Rodeh O, Bacik J, Mason C (2013). "BTRFS: The Linux B-tree filesystem". ACM Transactions on Storage (TOS). 9 (3): 1–32.

49. ^Purandare DR, Schmidt S, Miller EL. "Persimmon: an append-only ZNS-first filesystem." In: 2023 IEEE 41st International Conference on Computer Design (ICCD); 2023. p. 308-315. doi:10.1109/ICCD58817.2023.00054.

50. ^Ye Q, Karkra K. Cloud Storage Acceleration Layer (CSAL): An Open-Source, Host-Based Flash Translation Layer (FTL) [Internet]. 2023. Available from: https://www.snia.org/educational-library/cloud-storage-acceleration-layer-csal-enabling-unprecedented-performance-and. Last accessed: August 29, 2023.

Declarations

Funding: No specific funding was received for this work.

Potential competing interests: No potential competing interests to declare.