Valet: Efficient Data Placement on Modern SSDs

Devashish R. Purandare
dpuranda@ucsc.edu
UC Santa Cruz
Santa Cruz, CA, USA

Peter Alvaro
palvaro@ucsc.edu
UC Santa Cruz
Santa Cruz, CA, USA

Avani Wildani
agadani@gmail.com
Emory University / Cloudflare
San Francisco, CA, USA

Darrell D. E. Long
darrell@ucsc.edu
UC Santa Cruz
Santa Cruz, CA, USA

Ethan L. Miller
elm@ucsc.edu
Pure Storage / UC Santa Cruz
Santa Cruz, CA, USA

Abstract

The increasing demand for SSDs coupled with scaling difficulties has 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 the introduction of next-generation interfaces. It is therefore no surprise that such interfaces have seen limited adoption, leaving behind a graveyard of experimental interfaces ranging from open-channel SSDs to stream SSDs.

Our solution, Valet, leverages userspace *shim layers* to add placement hints for application data, delivering up to 2–4× write throughput over filesystems and comparable or better performance than application-specific solutions, with up to 6× lower tail latency. Valet generates dynamic placement hints, remapping application data to modern SSDs with *zero modifications* to the application, the filesystem, or the kernel. We demonstrate performance, efficiency, and multi-tenancy benefits of Valet across a set of widely-used applications: RocksDB, MongoDB, and CacheLib, presenting a solution that combines the performance of application-specific solutions with wide applicability to log-structured data-intensive applications.

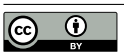
CCS Concepts

• **Computer systems organization** → **Secondary storage organization**; • **Software and its engineering** → *Cloud computing*.

Keywords

Solid-state drives, Data placement, Zoned namespaces, Storage systems, Shim layers, Write amplification, Log-structured systems

ACM Reference Format:
Devashish R. Purandare, Peter Alvaro, Avani Wildani, Darrell D. E. Long, and Ethan L. Miller. 2025. Valet: Efficient Data Placement on Modern SSDs. In *ACM Symposium on Cloud Computing (SoCC '25), November 19–21, 2025, Online, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3772052.3772256



SoCC '25, Online, USA

ACM ISBN 979-8-4007-2276-9/25/11
https://doi.org/10.1145/3772052.3772256

1 Introduction

As the demand for datacenter SSDs soars, scaling flash has become increasingly challenging, with density increases adversely impacting performance and device lifetime [24]. 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 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" [56] causes performance and lifetime degradation with redundant garbage collection at multiple levels.

In this paper, we focus on log-structured storage patterns as they make up almost all modern data-intensive systems, owing mainly to the performance benefits on hard disks (reduced seek), and efficiency benefits on SSDs (reduced garbage collection). Append-only log-structured storage backends are used by all major cloud storage providers [14, 25, 51], universally used in data-intensive applications [2–4, 23], and ubiquitous in modern filesystems [11, 30, 44].

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 latency spikes, which have a major impact on cloud storage performance [19].

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) [9, 43], and Flexible Data Placement (FDP) [45], 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 the responsibility for placement and garbage collection at the other. However, such changes require using newer abstractions or retrofitting current interfaces.

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-device combination, 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 the complexity of each interface in a module decoupled from applications and filesystems, 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. It isolates the complexity of varying interfaces and hint generation in pluggable userspace modules, allowing quick and easy changes to work with changing interfaces. Our library, Valet, presents a blueprint for dealing with the complexity of host-device coordination, allowing dynamic placement decisions without requiring application or filesystem rewrites. Our key insight is that shim layers can offer the *performance* of custom solutions, and *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 Valet 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 Valet with heuristic and learning-based hints, showcasing extensibility which is difficult to achieve in filesystems or applications. Valet 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.

2 Benefits of host-guided data placement

Hosts can greatly alleviate the need for garbage collection on SSDs and provide performance isolation for data streams if they can communicate data grouping to the device. This prevents interleaving of unrelated data on flash and reduces drive fragmentation. However, as we see in Table 1, manufacturer demonstrations of these interfaces have focused on 1–2 applications due to the complexity of adapting data-intensive applications to abstraction-breaking changes. To motivate the need for techniques that offer wider compatibility than application-specific approaches while maintaining their performance, we show a simple experiment:

Using a 4 TB Western Digital Ultrastar DC ZN540 SSD [54], we performed sequential write tests with flexible I/O tester (`fio`) [6] scaling up to 14 threads (the maximum open zones supported by the drive). We ran the tests on ZoneFS [37], a block-layer representation of the ZNS interface, and F2FS [30], 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 I/O, instructing F2FS to skip the buffer cache (ZoneFS does not support write buffering).

The results Fig. 1, show that 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` [18] and break them down by the CPU cycles spent by our test in each scenario. Even with `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× 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.

However, 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.

Picking the right layer for coordination

Traditionally, developers have implemented support for host involvement 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 effort with thousands of lines of code, which has been largely abandoned, with only a single commit since October 2023 [13].
(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 at 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,

Table 1: Data placement abstractions over the years provide a sobering reality: with new interfaces demonstrating a few applications before being deprecated for lack of use.

Type of SSD	Multi-Stream	Open-Channel	Zoned Namespaces	Flexible Data Placement
Introduced	2016	2017	2021	2022
Linux Kernel Support	Deprecated	Deprecated	Yes	—
Hint Interface	`fcntl()`	`liblightnvm`	`libzbd,libnvme`	`libnvme`
Filesystems	—	—	F2FS, BTRFS	—
Applications	AutoStream	RocksDB	RocksDB	Cachelib, RocksDB

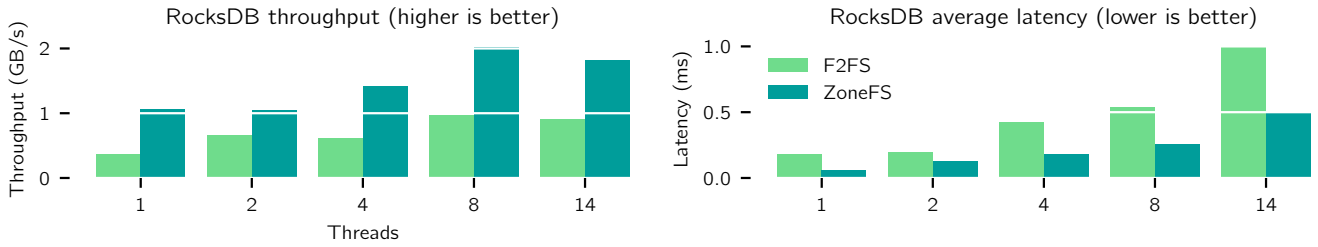


Figure 1: Writes to zoneFS can get the full bandwidth while F2FS sees degradation in both latency and throughput.

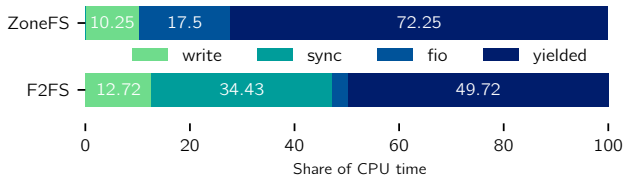


Figure 2: CPU time breakdown shows zoneFS yields 72% of CPU time back while F2FS only yields 49%, with 34% spent on synchronization overhead.

it cannot then effectively address multiple types of devices. Breaking away from the file abstraction can hurt tooling for operating on the data like replication and backup utilities.

What about a filesystem?

(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 just three hint levels are insufficient across all applications.
(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. 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 [50] 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, we can isolate the complexity in a small audit-able layer, greatly reducing attack surface while improving performance.

The Middle Road: Shim Layers. A shim layer can abstract interface changes from the applications and filesystems while enabling low-overhead reconfiguration to exploit 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 efficient adoption of modern SSDs, an ideal shim layer should require:

(1) *No changes to 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 a tuned application or filesystem.
(4) *Extensibility:* The hint generation should be configurable, adding the ability to add custom logic, including systems that learn dynamically.

We built Valet to stay true to these principles, and we demonstrate that such a layer is not only feasible; it can outperform other approaches. With Valet, 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 Valet Architecture

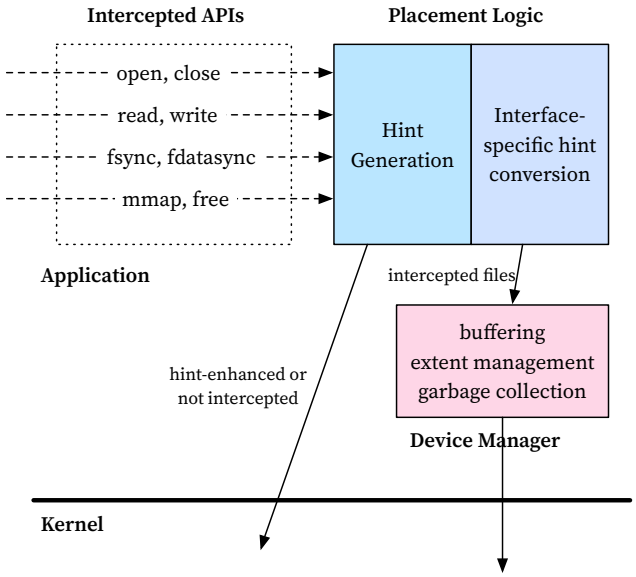


Figure 3: Simplified Valet architecture: Valet intercepts application calls, generates placement plans, resolves them to a particular protocol and finally manages data placement.

Valet 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.

Valet allows transparent hint injection or redirection of data based on the decision made by the placement engine. Since Valet 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.

Valet performs three tasks necessary for host-device coordination, which act at different layers of the modern stack—the application, the storage engine, and the device. Due to the differing hardware design approaches, we designed each of these components to be customizable and pluggable 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 *valet-mapper*.

Approaches like multi-stream and Flexible Data Placement (FDP) on the other hand, simply require a directive on where to place the data, hence the device manager can be skipped for those devices.

For Valet we present implementations for both of these approaches with demonstration using ZNS drives with Western Digital Ultrastar DC ZN540 and the ability to use kernel hint interfaces for multi-stream SSDs. Currently, FDP support is limited: hardware is not generally available and the directives (by design) are only supported through NVMe I/O. Since most applications do not use raw NVMe I/O (outside of exceptions like CacheLib [34]), we plan to look at FDP support as future work. This could be achieved by extending *valet-mapper* to use kernel NVMe device support and adding an extent mapping layer.
As seen in Fig. 3, Valet has 3 major components:

(1) Application call interception: Valet intercepts system calls issued by the application.
(2) Data Placement Engine: Valet uses automated or heuristic placement logic to group data and generate placement hints.
(3) Device Manager: Depending on the device type, Valet can either forward the hints or perform the physical 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 [8], FUSE [36], WASM [1], virtual machines, and dynamic libraries [16] being widely deployed. While Valet can be implemented with any of these techniques, we focused on `LD_PRELOAD` to avoid 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 I/O path through the kernel, while the complexity resides in userspace. `LD_PRELOAD` is used by projects ranging from custom allocators to debugging tools [21, 47].

Table 2 presents the design space of shim techniques. We chose the one with the best performance, and while its limitations may exclude certain applications, 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.

Valet is designed to be dynamically linked with the application at runtime using `LD_PRELOAD`, which allows the library, `libvalet`, to be loaded before other objects while executing a binary. Valet uses this functionality to selectively override various `libc` calls, using Valet calls instead. In hint-based interfaces, Valet 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

Table 2: `LD_PRELOAD` provides a valuable compromise, requiring no changes to the application or the kernel.

	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	~10× [60]
WASM	–	Yes	–	Yes	In Advance	~2.5× [27]
FUSE	Yes	Yes	Yes	–	In Advance	~1.8× [50]

require the host to take over data placement, garbage collection, and resource management. For hint-based approaches, Valet works on top of an existing storage system that supports hints like F2FS. For host-managed approaches, Valet implements its own device manager with *valet-mapper* and provides placement information per file. When used with *valet-mapper*, Valet captures the content of calls the application issues, copying the data to its buffer cache before persisting it based on hints.

In both cases, Valet needs to maintain a certain level of bookkeeping to supply directives per file. With the system calls that use file descriptors which are ephemeral and issued per-session, Valet needs to maintain a mapping of files to their descriptors to effectively understand which file a call is operating on. Since Valet does not have a runtime independent of the application, it leverages library load to set up a multi-threaded state, and specific conditions to request the read, write, garbage-collection or metadata threads to perform a task.

When working on top of a filesystem, Valet 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 operations. Data-intensive applications are often multithreaded and can result in separate threads accessing the same file descriptor concurrently. This can quickly lead to contention on lookups, especially as Valet is invoked on an intercepted call. To avoid frequent locking, Valet uses `dashmap` [53], a concurrent hashmap, for its state.

Valet's interception varies depending on whether the interface is hint-based or host-managed. Hint-based approaches are much simpler, but they lose out on some efficiency gains:

3.1.1 Valet in hint-based systems. In hint-based interfaces like multi-stream and FDP, Valet does not need *valet-mapper*, so its interception requires limited state tracking, operating once per created file.

(1) `library-load`: On library load, Valet blocks and sets up necessary state including loading previous metadata, updating internal data structures, read, write, garbage collect, and metadata threads, and loads the hint-generation configuration or model. This allows us to avoid allocation and initialization in hot data path.
(2) `open()`: If a new file is opened in write mode, Valet provides the path and open flags to the hint generation logic, adds the file to tracked files (`fdMap` and `pathMap`), 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()`: Valet 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`), Valet syncs `pathMap` on a successful return from the underlying filesystem.
(6) `rename()`: On a successful rename, Valet updates `pathMap` with the new path.

3.1.2 Valet in host-managed systems. Host-managed interfaces like ZNS require deeper support than issuing just the required placement hint, and we implement *valet-mapper*, 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, Valet needs to ensure more than simply issuing the right hint. It needs to notify *valet-mapper* to perform the needed task, not dissimilar to a lightweight virtual filesystem.

(1) `library-load`: The startup is similar to hint-based logic, but in addition to restoring states, Valet spins up *valet-mapper*, allowing it to allocate its metadata structures and pre-allocate write buffers for the buffering logic.
(2) `open()`: Open performs the same tasks as described previously, however, in the host-managed case, instead of issuing a hint system call, Valet 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 *valet-mapper* to flush data associated with the `fd`. *valet-mapper* further syncs its own metadata on every close (to uphold the POSIX contract on data persistence).
(4) `unlink()`: Removes all references to the file, requests *valet-mapper* to mark associated data for cleanup. Sends a message to the garbage collection thread to check if it can free up space.
(5) `write()`: On the write call and its variants, Valet forwards the buffer to *valet-mapper* with the UUID which handles writing the data to the device, on a successful return from *valet-mapper*, it returns success to the application.

[1] `LD_PRELOAD` and `syscall_intercept` added ~2 µs and ~2 ms of userspace overhead per syscall in our tests respectively.

(6) `read()`: On the read call and its variants, Valet translates the read to UUID and offset before forwarding the request to *valet-mapper*, and returns the data it gets back.
(7) `fsync()`: On sync, Valet persists its own mapping and forwards the request to *valet-mapper* to ensure the buffered data associated with the file is persisted.
(8) `rename()`: is unchanged from hint-based logic.

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

- `fallocate()`: In host-managed mode, these calls are suppressed as *valet-mapper* 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 *valet-mapper* 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. Valet forwards these calls to an in-place update friendly filesystem in the random-write area of the device.

As we see with `mmap()`, Valet allows specific files not to be intercepted using the hint mechanism. *valet-mapper* uses this for files that require in-place updates—manifest and configuration files, as well as special purpose files like `LOCK` files, 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 *valet-mapper* (e.g., `dup()`, `rmdir()`, etc.). Since most log-structured systems need a small amount of in-place updatable files, *valet-mapper* 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. Typically, these files make up less than 1% of the operations in log-structured systems.

Once the calls are intercepted, Valet requests hints from the placement engine. This pluggable module can either generate hints based on a heuristic or an automated model.

3.2 Data Placement Engine

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

(1) Data Affinity: Semantically-related writes grouped together maximize bandwidth through isolation.
(2) Lifetime grouping: Data that shares a common lifetime should be grouped together to minimize free space fragmentation.

For best results, 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). There are several issues with this approach: (1) three hint levels are insufficient with multiple applications, and different data streams, and (2) temperature of data may not always be correlated with lifetime. These rigid interfaces mean effective placement requires custom filesystems or kernel bypass. Valet, on the other hand, implements a flexible internal hint representation, with resolvers for translating these hints into currently supported APIs, and ability to extend support to future APIs. *valet-mapper* demonstrates the utility of richer interfaces and can unlock the full potential of the placement logic (see Section 4).

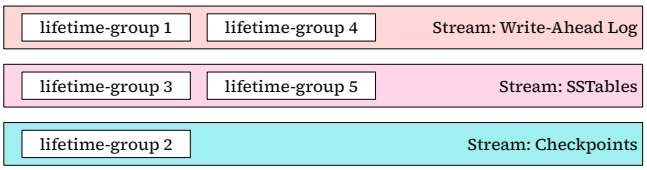


Figure 4: Valet hints are organized based on affinity (streams) and lifetime (groups).

3.2.1 Hints in Valet. Recognizing that grouping of data needs to be more than just a handful of write-temperature streams, Valet 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 Fig. 4, an application could have different data streams: a write-ahead log, data log (sorted string tables), and checkpoints. Thus, unlike traditional temperature-based hints, Valet accounts for both affinity and lifetime. We designed Valet's internal hint representation to be simple and extensible, allowing easy translation into the available hint formats and any future changes.

Valet's `get_hint()` API is called with a resolver, which can resolve hints into kernel hints, *valet-mapper* hints, or placement identifiers depending on the configuration (see Section 3.2.3) allowing it to work with existing hint systems as well as *valet-mapper*. Hint generation can be tuned 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.

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 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. Valet captures data such as paths, opening flags, and observed workload, and can train on this information to dynamically predict streams. In Section 4 we demonstrate this with batch-based mini KMeans [46] to dynamically pick streams based on observed workload. Valet-Learn picks the number of active streams that *valet-mapper* prefers, and uses a kmeans clustering model to fit file open characteristics to this set of streams. In the future, we plan to explore more dynamic features, as well as splitting individual files across streams depending on the observed read and write characteristics.

A placement algorithm that is independent of the application, filesystem, and the architecture allows a lot of flexibility in tuning placement, where the operators can use their expertise to tune hints based on their specific architecture and workload. In traditional systems such as applications or filesystems, such flexibility is almost impossible, as the placement logic will need to be in the kernel or within the application, and cannot be changed without recompiling the entire system. If a filesystem allows dynamic placement generation, it would need frequent kernel crossings to get placement information, or worse, put the model in kernel-space.

3.2.3 Resolving Valet hints into interface hints. To effectively use Valet'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 choose from.

- **Multi-stream**: Here, we ignore the placement groups, assigning each Valet 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 *valet-mapper* to demonstrate the full potential of our approach. Valet maps new zones to each stream and ensures that placement-groups are laid out sequentially within a zone.
- **Placement identifiers**: Here, Valet discard the streams, using FDP placement identifiers with Valet's lifetime-groups. As FDP implementation is planned as future work, this resolver has not been implemented.

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 *valet-mapper*, another pluggable module to demonstrate Valet's capabilities.

3.3 Device Manager—the *valet-mapper*

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 Valet's approach. With ZNS, Valet's storage backend, *valet-mapper* 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 [37], a filesystem wrapper in the kernel for zoned devices which allows using system calls rather than NVMe commands to manage ZNS SSDs. *valet-mapper* performs three main tasks: data layout, buffering, and garbage collection.

3.3.1 Data Layout. Inspired by ZNS' design, *valet-mapper*'s data layout maps streams to individual zones. To support arbitrary file sizes, *valet-mapper* implements an extent-based logic, allowing block-aligned extents ranging from 4 KiB to 512 KiB. On sync, the extent is padded to the nearest block-boundary and flushed to the stream-mapped zone. Once the zone fills up, a new zone is fetched from a list of free zones. *valet-mapper* uses *allocate-on-flush*, deferring block reservation until a close or sync command to further optimize grouping.

valet-mapper does not allow files to share chunks. This means that it only needs to maintain the zone, the offset, and the size, to locate a particular extent. *valet-mapper* 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, extent-size)` structures. Valet maintains an additional structure, the `zoneMap`, to maintain per-zone metadata to simplify garbage collection. *valet-mapper* 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.

On ZNS devices, since the allocation of device buffers is host-managed, *valet-mapper* needs to track active resources and keep them below the device limits by periodically finishing or closing zones. *valet-mapper* 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 in parallel.

3.3.2 Buffering. As zoneFS does not support write buffering, we implement write buffering in *valet-mapper*. Due to the log-structured nature of applications handled by *valet-mapper*, we use a simplified write-through buffer design. *valet-mapper* 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, *valet-mapper* allocates blocks based on the corresponding stream and flushes the buffer to storage. While this approach adds a startup cost, we avoid allocating memory in the hot data path, improving overall performance.

3.3.3 Garbage Collection. *valet-mapper* implements lazy garbage collection, deferring data movement as long as possible. On each delete, *valet-mapper* 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. In most log-structured systems, frequently updated streams like write-ahead logs see frequent deletes, allowing *valet-mapper* to reset fully-invalidated zones frequently. If available zones go below a user-configured threshold, Valet iterates through `zoneMap` identifying 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.

3.3.4 Crash Consistency. Valet offers similar guarantees as POSIX filesystems on a crash. Throughout execution, it performs careful metadata synchronization, updating various persistent structures on calls to `fsync(), fdatasync()` or `close()`, similar to filesystems. These updates write to a backup region before an atomic commit, which swaps the current metadata with the backup. This ensures that metadata persistence failures do not corrupt the whole system. Currently, we store the metadata in a human-readable JSON format on disk, as it is relatively small (a few KiB) and helpful for offline data analysis.

We made these decisions based on the general systems we want to replace with Valet, if there is a need for stronger guarantees or more efficiency, the metadata module will require small updates to support write-ahead logging or more efficient storage formats. In the current design, a crash will cause data between the crash and the last commit operation (triggered by sync or close operations) to be lost. On a new boot, the constructor checks for metadata in a known location on disk and reconstructs its state before allowing future operations to proceed.

3.4 Limitations

Valet inherits the limitations of `LD_PRELOAD` that we discussed in Section 3.1. While Valet cannot intercept statically-linked applications, in practice we observe that statically compiled applications like RocksDB, Cachelib, and MongoDB still dynamically link with `libc`, and can be preloaded with Valet. The bigger limitation comes with languages that do not use `libc` like Golang and Java, systems written in these languages cannot be intercepted by `libc` replacements, 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 different implementation of Valet could use approaches that replace `libc` like a custom library, or webassembly system interface to address these applications.

We limit the scope of *valet-mapper* to log-structured append-only applications. Since all major data-intensive systems almost exclusively follow this pattern [12, 15, 20, 33], we can adapt several applications to this interface, but applications with data structures that use in-place updates or mmap writes cannot utilize *valet-mapper* and will be passed through to the filesystem on conventional zones. Valet can still issue hints for these writes if the underlying filesystem supports them.

Finally, files stored by *valet-mapper* will not be visible to third party utilities like backup and copy unless they are preloaded with Valet as well. *valet-mapper* 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 *valet-mapper*. We could address these limitations by implementing Valet using other shim techniques, however, as we discussed in Section 3.1, each technique comes with its own set of trade-offs. As we will see in Section 4, we prioritized performance in our design, and the system can be ported to other techniques for wider-applicability.

4 Evaluation

To demonstrate the benefits of Valet, we present three different types of evaluation; we present three case studies with popular data management systems RocksDB [22], MongoDB [38], and CacheLib [34]. 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 Valet and the performance benefits we get with each of the systems. Here we compare Valet with filesystem approaches like F2FS, and special-purpose systems (ZenFS). Due to limited support for ZNS SSDs, we could not include other filesystems in our comparison.

To evaluate Valet, we set up a test server with 64-core AMD EPYC 7452 system with 128 GB of DRAM. We use Ubuntu 22.04 with Linux kernel 6.5, and 2 Western Digital Ultrastar DC ZN540 [54] each 4 TB in size. We used the latest stable release of each system 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 Valet uses it for `LOCK` files, we set up the region in the 4 GB random write space on the same drive.

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 the performance that highly tuned applications can achieve. For our evaluation we ran 100 million operations, each with 20 B keys and 400 B 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 Valet with two types of placement generation logic, the heuristic approach and the 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, (Valet-Learn) we used batch-based mini KMeans clustering [46], 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.

As seen in Fig. 5, for inserts and updates, Valet offers over 2× improvement over F2FS and is comparable to tailored approaches like ZenFS. While ZenFS offers higher throughput in each write case, as we will see in Fig. 6, it suffers from high tail latencies. For reads, Valet equals any other approach: whether it be F2FS using

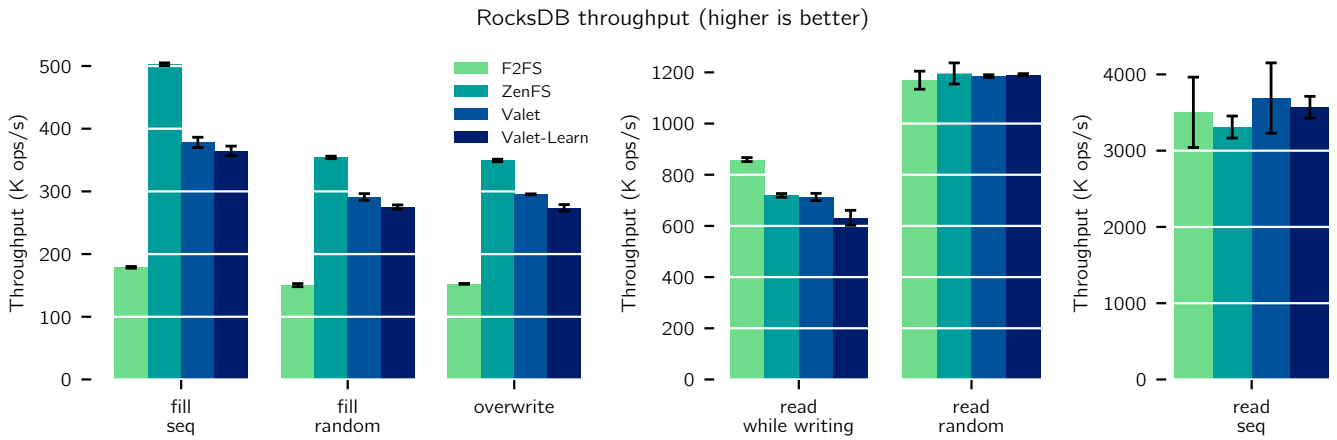


Figure 5: Throughput for `db_bench` workloads.

the kernel cache or zenFS largely avoiding the kernel. In each case Valet-Learn suffers from a slight overhead as it generates hints on the fly. However, unlike zenFS and baseline Valet, Valet-Learn can be used with other applications without any change.

One of the main benefits of write isolation is the improvement in tail latency. Valet's user-space nature ensures that compared to the high cost of kernel crossings and persistence, its impact is minimal. Here, Valet is comparable to zenFS and offers a lower read latency than F2FS. Valet-Learn adds a small overhead, but it is still only about a dozen microseconds at 99.9 percentile, while still outperforming F2FS and zenFS.

As seen in Fig. 6, Valet offers better tail latency over both F2FS and zenFS for random inserts and random reads. This is particularly evident at p99.99 (Valet: ~20 µs, zenFS: ~555 µs), where Valet benefits from the filesystem optimizations in RocksDB, intercepting `readahead()` and pre-buffering data, a filesystem optimization unavailable to zenFS. Valet sees a 300 ns impact on read latency due to differences between F2FS and zoneFS, however since reads take several microseconds, it is unlikely to have a real-world impact. With automation, in Valet-Learn we observe that it can match the performance of hand-tuned approaches, however it can suffer small tail latency spikes due to the prediction engine.

4.1.1 Why is Valet faster? Valet 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 (from `perf` [32]) as seen in Fig. 7 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 Valet 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 Valet 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 Fig. 6.

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

4.2 Case Study: MongoDB

MongoDB's backend, WiredTiger [39] gives 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 Valet. We tested WiredTiger's performance with the medium-sized lsm tree test in `wt-perf`. This included multithreaded reads, multithreaded updates, and simultaneous multithreaded updates and reads.

WiredTiger performs two streams of writes: logs and SSTs. Valet does not need any changes to support the data, but we ran into issues supporting the write-ahead logs. WiredTiger uses `mmap()` writes for logging, which necessitate in-place and out-of-order updates that are not supported in *valet-mapper*. While `mmap()` databases are not recommended [17], this limitation is handled by Valet in its random-write region (with the lock files). Fundamentally, mmap-writes do not guarantee ordering and are a violation of the log-structured contract.

For comparison, we looked at F2FS, providing it stream hints through Valet while *valet-mapper* uses Valet's richer hints.

In WiredTiger, we observe a dramatic improvement in updates and reads in the LSM mode. As we see in Fig. 8 Valet with multithreaded readers is more than 4× the throughput of F2FS, with write-heavy workloads offering a 3× throughput. Again, the benefits come from the simpler structure, offering dedicated write streams to different files eliminates contention on the SSD. mmap-writes when mixed with regular writes can cause contention on filesystem resources in addition to the device resources. Valet still performs them, but assigning different device regions reduces the contention. Valet-Learn again presents a minimal overhead, making a great case for exclusively using automated hints.

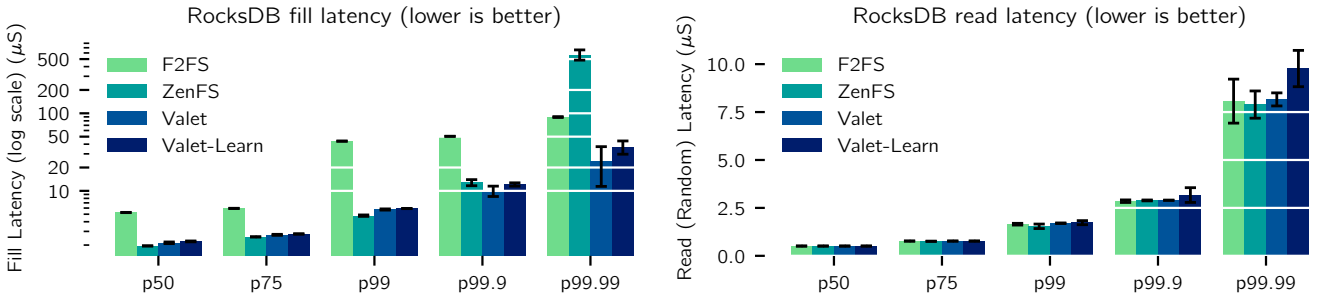


Figure 6: Valet offers comparable latency to specifically tuned zenFS and improved latency over F2FS.

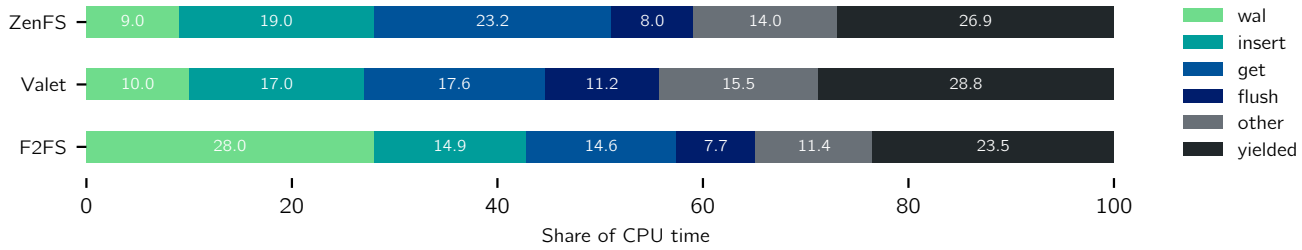


Figure 7: Analyzing each system, we see how a filesystem like F2FS gets consumed by frequent flush operations.

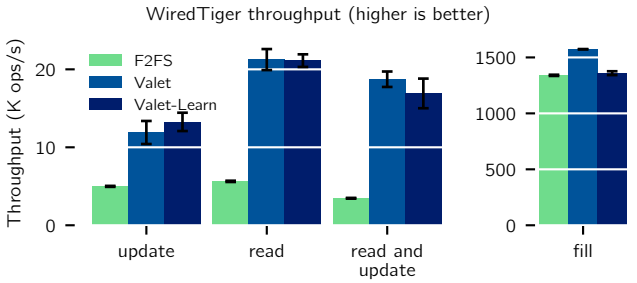


Figure 8: Due to physical separation of `log` and `.lsm` files to dedicated zones, Valet can allow dramatically faster writes and updates in WiredTiger.

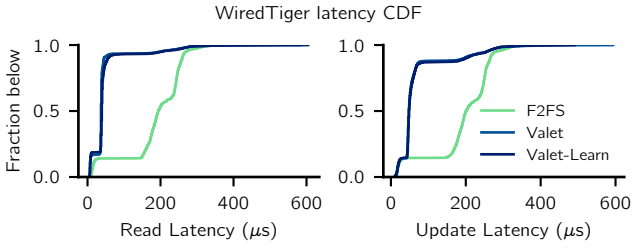


Figure 9: The isolation across logs eliminates contention and improves both read and update latency.

In terms of latency, both instances of Valet vastly outperform F2FS, offering <100 µs latency in almost all operations even as F2FS exceeds 200 µs.

4.3 Case Study: CacheLib

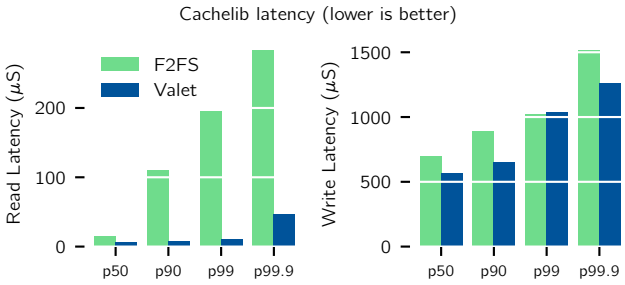


Figure 10: With CacheLib's `cachebench` tool, Valet outperforms F2FS on throughput and read-write latency

Finally, to demonstrate caching workloads we adopted Meta's caching library CacheLib [7] to *valet-mapper*, which required no change in Valet's logic. As we see in Fig. 10, Valet offers lower latency for reads and writes. To perform this test we set up CacheLib with 128 MB 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 used heuristics to map these to different Streams. This is one of the cases where automated approaches were identical, as the write streams are identical.

On CacheLib Valet sees a small (8%) improvement in throughput 56.8 KOps/s as compared to F2FS' 52.3 KOps/s but a large reduction

in read latency, particularly at p90 and above as seen in Fig. 10. This small difference is partly due to the highly optimized Navy engine, which uses optimizations `io_uring` [5], skipping a lot of filesystem overhead, and hence Valet only sees a modest improvement. Reads are still improved, as incoming writes do not block read operations.

4.4 Multi-tenancy

One of the main benefits of ZNS SSDs is the ability to minimize degradation caused by write interference from multiple streams of incoming writes. With Valet, multiple applications can be run in parallel on the same SSD with a significantly lower degradation in performance as their writes are isolated to separate write resources. To evaluate degradation, we ran Valet in a multi-tenant environment, addressing MongoDB and RocksDB simultaneously on the same device. This results in 4 concurrent writers, each multithreaded: RocksDB, RocksDB-WAL, WiredTiger-WAL and WiredTiger. We measured the throughput and latency on both WiredTiger and RocksDB, focusing on tail latency, which is typically adversely affected by such workloads. We observed that throughput was not greatly affected on either system, but latency numbers show interesting trends.

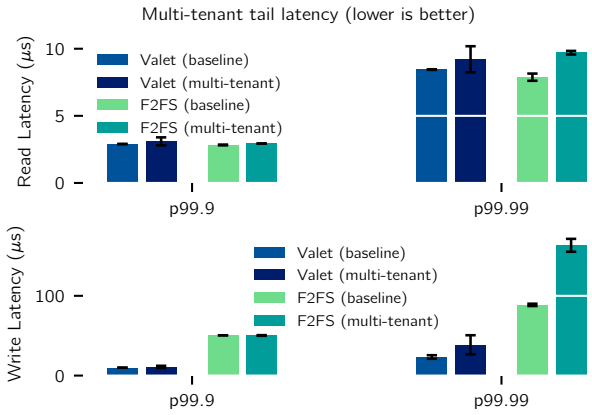


Figure 11: Valet can exploit device-level isolation as F2FS gets bogged down by concurrent writers.

As we see in Fig. 11, tail latencies for writes see dramatic spikes in filesystems like F2FS, owing to additional locking, and rigid adherence to specific open zones based on the filesystem structure. This causes the worst-case tail latency to spike almost 2×. On the other hand, Valet can simply map writes to different device buffers and regions and they see only a small degradation in latency. Despite being tailored for ZNS SSDs, systems like ZenFS inherently cannot allow multiple tenants or applications, significantly limiting their data center applicability.

4.5 Overhead

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

4.5.1 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, *valet-mapper* sees a device write amplification factor of 1, similar to F2FS and ZenFS. We ran an insert-heavy benchmark (100M sequential writes, random writes, and updates). As we see in Table 3, in our write-intensive experiments, Valet frees up more space without moving any data as WAL blocks are freed up quickly, allowing Valet to erase without the need for relocating data. We excluded ZenFS from this comparison as it does not implement garbage collection.

Table 3: Garbage collection performance

	Valet	F2FS
Garbage Collection Calls	8	4
Data Moved	0	1059 MiB
Free Space at the End	48.2 GiB	33.2 GiB

As we see in Table 3, freeing up space in *valet-mapper* was more frequent and more efficient, moving no data as entire zones were invalidated due to the better placement.

4.5.2 Memory Usage. Valet in hint-only mode does not maintain any file data and uses no extra memory. However, *valet-mapper* needs to perform write buffering, which requires allocating memory depending on the number of open files. To speed up access, Valet maintains all its maps in memory, however except for `fileMap`, the rest are either fixed size or frequently cleaned up. In our experiments we used 32 MB buffers with 2 write streams, totaling 64 MB write buffers which made up most of the actively used memory. This size is smaller than the buffers maintained by filesystems in the kernel (F2FS memory usage went up to 200 MB in the same experiments). Profiling the memory using KDE heaptrack [28], *valet-mapper* used 73 MB at its peak in the evaluation experiments.

Table 4: Added lines of code

System	Application	Kernel	Userspace
ZenFS	4017	0	988
F2FS (ZNS)	0	38,188	1252
Valet	0	0	1700

4.5.3 Simplifying the storage stack. Valet greatly simplifies the metadata and placement logic, replacing large amounts of unnecessary kernel code with a simplified mapper that is a better fit for modern applications and devices. As we discussed in Section 2, added complexity in the kernel comes at a cost of security, and modifying applications can make them hard to maintain. For instance, ZenFS already does not work with the latest RocksDB.

5 Related Work

Early approaches to leveraging new SSD interfaces involved modifying applications [52, 59] or using application-specific backends.

zenFS [10, 40] is a RocksDB plugin that maps RocksDB's files to ZNS zones. WALTZ [31] further optimizes zenFS, reducing tail latency using the zone append command.

Application-specific backends. Our work includes comparison of our approach with zenFS, while it offers better performance in certain conditions, Valet is close behind and can allow non-RocksDB applications. Additionally, zenFS development has stalled, highlighting challenges in maintaining application-specific backends.

Similarly, we attempted to compare Valet's CacheLib performance with Region-Cache [55], another ZNS-based CacheLib implementation that demonstrates the filesystem overhead for data-intensive applications. However, Region-Cache targets a CacheLib version from over a year ago, with different kernel requirements, and dependencies, preventing direct comparison in our test environment. In contrast, Valet works with unmodified CacheLib, avoiding version lock-in. Overall application-specific tuning will offer the best performance, but it requires a deep understanding of the application as well as the storage medium, and significant engineering effort. Even with these, it will suffer from version lock-in, and limited applicability.

Filesystems. Filesystems like F2FS [30], and BTRFS [44] offer ZNS-specific improvements and are similar to Valet with 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 [42] is an append-only fork of F2FS that requires no random-write area. Persimmon's performance is similar to F2FS. However, it is tied to an older kernel version (5.18), and hence, we were unable to evaluate it. BTRFS ZNS support is extremely limited and error-prone, and our benchmarks did not work on BTRFS.

Filesystem approaches face three key limitations that Valet addresses: (1) in-kernel execution adds synchronization overhead, (2) lack of application-specific hint information limits data placement optimization, and (3) architectural constraints like F2FS's 3-data-log limit restrict parallelism. *valet-mapper*, 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.

Interposition. Interposition approaches outside the filesystem have used either eBPF [61] or SPDK [57] as kernel-bypass mechanisms. Other approaches have used syscall_intercept [16] to overload system calls for persistent-memory programming by disassembling and patching binaries. The A few `LD_PRELOAD`-based filesystem prototypes exist, like PlasticFS [35] and AVFS [26], which allow peeking into compressed files. Goanna [48] implemented a filesystem through ptrace extensions, similar in spirit to `LD_PRELOAD`. More recently, zIO [49] used user-space libraries using `LD_PRELOAD` to eliminate unnecessary copies of data. Valet uses a similar interposition to redirect data.

Hint generation. While there are many proposals for hint formats, hint generation is largely an overlooked area in SSD research. Since workloads can be unpredictable, file open provides few distinguishing features, and abstractions ensure that the systems remain unaware of what they are storing, hint generation is an important challenge [41]. We address it in Valet with affinity and lifetime, while traditional approaches such as the kernel hints have focused on temperature.

Valet 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 [58] enables the adoption of ZNS with clusters of varied storage and a host-based flash translation layer. More recently, Google demonstrated the use of filenames and open flags to decide data placement, an approach similar to what Valet-Learn does [29]. These approaches rely on diverse tiered storage types with caches to balance random vs. sequential writes. Valet operates at the device interface level and could be the storage backend for such services.

Valet's modular architecture allows swapping hint generation strategies through pluggable modules (Section 3.2) without affecting other system components. This extensibility enables experimentation with machine learning models, LLM-based classification, or domain-specific heuristics as hint generation techniques evolve.

6 Conclusion

The various efforts to introduce host-guided data placement present a sorry picture: we see a recurring pattern 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. Valet represents an approach in this space, allowing filesystem and application development to 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. In this paper, we demonstrated how application-specific approaches were insufficient, and one-size-fits-all filesystems were inefficient.

Hardware is in turmoil. Compute, memory, and storage are dramatically changing abstractions, making adoption challenging while maintaining compatibility. Ultimately, as is the case with Valet, we believe that the way to address the complexity is to isolate it in a dedicated layer that can be modified independently of other parts. This approach can speed up adoption of modern interfaces, simplify programming, offer improved performance, and allow broad application compatibility. Valet can park your data more efficiently for you.

Acknowledgements

We are grateful to Matias Bjørling and Western Digital for the ZN540 drives and feedback on this work. We would like to thank the industrial sponsors of the Center for Research in Storage and Systems (CRSS), and the Kumar Malavalli endowment at UC Santa Cruz for supporting this work. Feedback from Daniel Bittman, Achilles Benetopoulos, Eugene Chou, and other CRSS researchers helped shape Valet, and we are grateful for their support. We would also like to thank the anonymous reviewers for their feedback and suggestions, which made this paper stronger.

The source code for the work presented in this paper is available at https://github.com/shimplify/valet.

References

[1] Bytecode Alliance. WASI: WebAssembly System Interface. https://github.com/bytecodealliance/wasmtime/blob/main/docs/WASI-overview.md, 2022.

[2] Apache Software Foundation. Apache BookKeeper Documentation. https://bookkeeper.apache.org/docs/, 2024. Accessed: 2025-01-06.

[3] Apache Software Foundation. Apache Cassandra Architecture Documentation. https://cassandra.apache.org/doc/latest/architecture/, 2024. Accessed: 2025-01-06.

[4] Apache Software Foundation. Apache Pulsar Documentation. https://pulsar.apache.org/docs/, 2024. Accessed: 2025-01-06.

[5] Jens Axboe. Efficient IO with io_uring. https://kernel.dk/io_uring.pdf.

[6] Jens Axboe. fio - flexible I/O tester rev. 3.30.

[7] Benjamin Berg, Daniel S. Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, and Gregory R. Ganger. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768. USENIX Association, November 2020.

[8] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 121–134, Renton, WA, July 2019. USENIX Association.

[9] Matias Bjørling. From Open-Channel SSDs to Zoned Namespaces. In *Vault '19*, Boston, MA, February 2019. USENIX Association.

[10] 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.

[11] Jeff Bonwick and Bill Moore. ZFS: The Last Word in File Systems. In *Sun Microsystems*, 2003. ZFS was announced on September 14, 2004, integrated into Solaris on October 31, 2005.

[12] James Bornholt, Rajeev Joshi, Vytautas Astrauskas, Brendan Cully, Bernhard Kragl, Seth Markle, Kyle Sauri, Drew Schleit, Grant Slatton, Serdar Tasiran, Jacob Van Geffen, and Andrew Warfield. Using lightweight formal methods to validate a key-value storage node in amazon s3. In *SOSP 2021*, 2021.

[13] bpan2020. "there is an error when i am compilling rocksdb version above 8.10.0 with zenfs 2.1.4". https://github.com/westerndigitalcorporation/zenfs/issues/288, 2024. ZenFS GitHub Issue #288.

[14] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, et al. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 143–157, 2011.

[15] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: a highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, page 143–157, New York, NY, USA, 2011. Association for Computing Machinery.

[16] Intel Corp. syscall_intercept. https://github.com/pmem/syscall_intercept, 2023. Retreived: 2017-03-20.

[17] Andrew Crotty, Viktor Leis, and Andrew Pavlo. Are you sure you want to use mmap in your database management system? In *CIDR 2022, Conference on Innovative Data Systems Research*, 2022.

[18] Arnaldo Carvalho De Melo. The new Linux perf tools. In *Slides from Linux Kongress*, volume 18, pages 1–42, 2010.

[19] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Commun. ACM*, 56(2):74–80, February 2013.

[20] Pavan Edara, Jonathan Forbes, and Bigang Li. Vortex: A stream-oriented storage engine for big data analytics. In *SIGMOD*, 2024.

[21] Jason Evans. jemalloc. https://github.com/jemalloc/jemalloc, 2005. Available on GitHub.

[22] Facebook. Rocksdb. https://github.com/facebook/rocksdb, 2013. Available on GitHub.

[23] Google Cloud. Google Cloud Spanner Architecture Documentation. https://cloud.google.com/spanner/docs/architecture, 2024. Accessed: 2025-01-06.

[24] Laura M. Grupp, John D. Davis, and Steven Swanson. The bleak future of nand flash memory. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST'12, page 2, USA, 2012. USENIX Association.

[25] Dean Hildebrand and Denis Serenyi. Colossus under the hood: a peek into Google's scalable storage system. Google Cloud Blog, April 2021. Accessed: 2025-01-06.

[26] Ralf Hoffmann and Miklos Szeredi. AVFS: A virtual filesystem. https://avf.sourceforge.net/, 2001. Last accessed: August 29, 2023.

[27] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, Renton, WA, July 2019. USENIX Association.

[28] KDE. KDE heaptrack. https://github.com/KDE/heaptrack, 2024. Last Accessed: January 15 2024.

[29] Larry Greenfield and Seth Pollen. How colossus optimizes data placement for performance. https://cloud.google.com/blog/products/storage-data-transfer/how-colossus-optimizes-data-placement-for-performance, March 2025. Google Cloud Blog.

[30] 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, 2015.

[31] Jongsung Lee, Donguk Kim, and Jae W. Lee. WALTZ: Leveraging Zone Append to Tighten the Tail Latency of LSM Tree on ZNS SSD. *Proc. VLDB Endow.*, 16(11):2884–2896, aug 2023.

[32] Linux Kernel Community. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/, 2009. Linux performance analysis tool.

[33] Yoshinori Matsunobu, Siying Dong, and Herman Lee. Myrocks: Lsm-tree database storage engine serving facebook's social graph. *Proc. VLDB Endow.*, 13(12):3217–3230, August 2020.

[34] Inc. Meta Platforms. Cachelib. https://github.com/facebook/CacheLib, 2020. Available on GitHub.

[35] Peter Miller. PlasticFS: A gnu project. https://plasticfs.sourceforge.net/, 2012. Last updated: 2012.

[36] Naoki Mizusawa, Kenji Nakazima, and Saneyasu Yamaguchi. Performance evaluation of file operations on OverlayFS. In *2017 Fifth International Symposium on Computing and Networking (CANDAR)*, pages 597–599. IEEE, 2017.

[37] Damien Le Moal and Ting Yao. Zonefs: Mapping POSIX file system interface to raw zoned block device accesses. In *Vault '20*, Santa Clara, CA, February 2020. USENIX Association.

[38] Inc. MongoDB. Mongodb. https://www.mongodb.com/, 2009. Available at https://www.mongodb.com/.

[39] MongoDB, Inc. Wiredtiger storage engine. https://github.com/wiredtiger/wiredtiger, 2012. Open source storage engine.

[40] Myounghoon Oh, Seehwan Yoo, Jongmoo Choi, Jeongsu Park, and Chang-Eun Choi. ZenFS+: Nurturing Performance and Isolation to ZenFS. *IEEE Access*, 11:26344–26357, 2023.

[41] Devashish R. Purandare. *Enhancing Flash Storage Performance and Lifetime with Host-Guided Data Placement*. PhD thesis, UC Santa Cruz, 2024.

[42] Devashish R. Purandare, Sam Schmidt, and Ethan L. Miller. Persimmon: an append-only zns-first filesystem. In *2023 IEEE 41st International Conference on Computer Design (ICCD)*, pages 308–315, 2023.

[43] Devashish R. Purandare, Pete Wilcox, Heiner Litz, and Shel Finkelstein. Append is near: Log-based data management on ZNS SSDs. In *Conference on Innovative Data Systems Research 2022 (CIDR '22)*, January 2022.

[44] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The linux b-tree filesystem. *ACM Transactions on Storage (TOS)*, 9(3):1–32, 2013.

[45] Chris Sabol and Ross Stenfort. Hyperscale innovation: Flexible data placement mode (FDP), 2022.

[46] D. Sculley. Web-scale k-means clustering. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, page 1177–1178, New York, NY, USA, 2010. Association for Computing Machinery.

[47] Julian Seward, Nicholas Nethercote, et al. Valgrind: A framework for heavyweight dynamic binary instrumentation. https://valgrind.org/, 2007. Available at https://valgrind.org/.

[48] R. P. Spillane, S. Gaikwad, E. Zadok, C. P. Wright, and M. Chinni. Enabling transactional file access via lightweight kernel extensions. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, pages 29–42, San Francisco, CA, February 2009. USENIX Association.

[49] Timothy Stamler, Deukyeon Hwang, Amanda Raybuck, Wei Zhang, and Simon Peter. zIO: Accelerating IO-Intensive applications with transparent Zero-Copy IO. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 431–445, Carlsbad, CA, July 2022. USENIX Association.

[50] Bharath Kumar Reddy Vangoor, Vasily Tarasov, and Erez Zadok. To FUSE or not to FUSE: Performance of User-Space file systems. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 59–72, 2017.

[51] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052, 2017.

[52] 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.

[53] Joel Wejdenstal. dashmap: A high-performance concurrent hash map for rust. https://github.com/xacrimon/dashmap, 2023. Accessed: September 21, 2023.

[54] Western Digital Corporation. Ultrastar DC ZN540 NVMe SSD. https://www.westerndigital.com/en-ae/products/internal-drives/ultrastar-dc-zn540-nvme-ssd?sku=0TS2096, 2024. Accessed: 2024-10-22.

[55] Chongzhuo Yang, Zhang Cao, Chang Guo, Ming Zhao, and Zhichao Cao. Can zns ssds be better storage devices for persistent cache? In *Proceedings of the 16th ACM Workshop on Hot Topics in Storage and File Systems*, HotStorage '24, page 55–62, New York, NY, USA, 2024. Association for Computing Machinery.

[56] 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)*, Broomfield, CO, October 2014. USENIX Association.

[57] Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu, Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, and Luse E Paul. SPDK: a development kit to build high performance storage applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 154–161. IEEE, 2017.

[58] Qinghua Ye and Kapil Karkra. Cloud Storage Acceleration Layer (CSAL): An open-source, host-based flash translation layer (FTL). https://www.snia.org/educational-library/cloud-storage-acceleration-layer-csal-enabling-unprecedented-performance-and, 2023. Last accessed: August 29, 2023.

[59] Jiacheng Zhang, Youyou Lu, Jiwu Shu, and Xiongjun Qin. Flashkv: Accelerating KV performance with open-channel SSDs. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(5s):1–19, 2017.

[60] Yusheng Zheng, Tong Yu, Yiwei Yang, Yanpeng Hu, Xiaozheng Lai, and Andrew Quinn. bpftime: userspace eBPF Runtime for Uprobe, Syscall and Kernel-User Interactions, December 2023. arXiv:2311.07923 [cs].

[61] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, Carlsbad, CA, July 2022. USENIX Association.