Elevating commodity storage with the SALSA host translation layer

Nikolas Ioannou Kornilios Kourtis Ioannis Koltsidas
IBM Research, Zurich
{*nio,kou,iko*}*@zurich.ibm.com*

***Abstract*—To satisfy increasing storage demands in both capacity and performance, industry has turned to multiple storage technologies, including Flash SSDs and SMR disks. These devices employ a translation layer that conceals the idiosyncrasies of their mediums and enables random access. Device translation layers are, however, inherently constrained: resources on the drive are scarce, they cannot be adapted to application requirements, and lack visibility across multiple devices. As a result, performance and durability of many storage devices is severely degraded.**

In this paper, we present SALSA: a translation layer that executes on the host and allows unmodified applications to better utilize commodity storage. SALSA supports a wide range of single- and multi-device optimizations and, because is implemented in software, can adapt to specific workloads. We describe SALSA's design, and demonstrate its significant benefits using microbenchmarks and case studies based on three applications: MySQL, the Swift object store, and a video server.

I. Introduction

The storage landscape is increasingly diverse. The market is dominated by spinning magnetic disks (HDDs) and Solid State Drives (SSDs) based on NAND Flash. Broadly speaking, HDDs offer lower cost per GB, while SSDs offer better performance, especially for read-dominated workloads. Furthermore, emerging technologies provide new tradeoffs: Shingled Magnetic Recording (SMR) disks [1] offer increased capacity compared to HDDs, while Non-Volatile Memories (NVM) [2] offer persistence with performance characteristics close to that of DRAM. At the same time, applications have different requirements and access patterns, and no one-size-fits-all storage solution exists. Choosing between SSDs, HDDs, and SMRs, for example, depends on the capacity, performance and cost requirements, as well as on the workload. To complicate things further, many applications (e.g., media services [3], [4]) require multiple storage media to meet their requirements.

Storage devices are also frequently idiosyncratic. NAND Flash, for example, has a different access and erase granularity, while SMR disks preclude in-place updates, allowing only appends. Because upper layers (e.g., databases and filesystems) are often not equipped to deal with these idiosyncrasies, translation layers [5] are introduced to enable applications to access idiosyncratic storage transparently. Translation layers (TLs) implement an indirection between the logical space as seen by the application, and the physical storage as exposed by the device. A TL can either be implemented on the host (host TL) or on the device controller (drive TL). It is well established that, for many workloads, drive TLs lead to suboptimal use of the storage medium. Many works identify these performance problems, and try to address them by improving the controller translation layer [6], [7], or adapting various layers of the I/O software stack: filesystems [8]–[10], caches [11], [12], paging [13], and key-value stores [14]–[18].

In agreement with a number of recent works [19]–[21], we argue that these shortcomings are inherent to drive TLs, and advocate placing the TL on the host. While a host TL is not a new idea [22], [23], our approach is different from previous works in a number of ways. First, we focus on commodity drives, without dependencies on specific vendors. Our goal is to enable datacenter applications to use cost-effective storage, while maintaining acceptable performance. Second, we propose a unified TL framework that supports different storage technologies (e.g., SSDs, SMRs). Third, we argue for a host TL that can be adapted to different application requirements and realize different tradeoffs. Finally, we propose a TL that can virtualize multiple devices, potentially of different types. The latter allows optimizing TL functions such as load balancing and wear leveling across devices, while also addressing storage diversity by enabling hybrid systems that utilize different media.

SALSA (SoftwAre Log Structured Array) implements the above ideas, following a log-structured architecture [24], [25]. We envision SALSA as the backend of a software-defined storage system, where it manages a shared storage pool, and can be configured to use workload-specific policies for each application using the storage. In this paper, we focus on the case where SALSA is used to run unmodified applications by exposing a Linux block device that can be either used directly, or mounted by a traditional Linux filesystem. The contribution of our work is a novel host TL architecture and implementation that supports different media and allows optimizing for different objectives. Specifically:

- SALSA achieves substantial performance and durability benefits by implementing the TL on the host for single- and multi-device setups. When deploying MySQL database containers on commodity SSDs, SALSA outperforms the raw device by $1.7\times$ on one SSD and by $35.4\times$ on a software RAID-5 array.
- SALSA makes efficient use of storage by allowing

application-specific policies. We present a SALSA policy tailored to the Swift object store [26] on SMRs that outperforms the raw device by up to a factor of 6.3×.
- SALSA decouples space management from storage policy. This enables SALSA to accommodate different applications, each with its own policy, using the same storage pool. This allows running MySQL and Swift on the same storage with high performance and a low overhead.
- SALSA embraces storage diversity by supporting multiple types of devices. We present how we combine SMRs and SSDs to speedup file retrieval for a video server workload (where adding an SSD improves read performance by 19.1×), without modifying the application.

The remaining of the paper is organized as follows. We start with our a brief overview of idiosyncratic storage and our motivation behind SALSA (§II). We continue with a description of the design of SALSA (§III), discuss how we satisfy specific application workload requirements (§IV), and evaluate our approach (§V). Finally, we discuss related work (§VI) and conclude (§VII).

II. Background and Motivation

In this section, we provide a brief background on Flash-based SSDs (§II-A) and Shingled Magnetic Recording (SMR) disks (§II-B), analyze the limitations of commodity drive TLs (§II-C), and argue for a unified host TL architecture (§II-D).

A. Flash-based SSDs

Flash memory fits well in the gap between DRAM and spinning disks: it offers low-latency random accesses compared to disks at a significantly lower cost than DRAM. As a result, its adoption is constantly increasing in the data center [27]–[29], where it is primarily deployed in the form of SSDs. Nevertheless, Flash has unique characteristics that complicate its use [30]. First, writes are significantly more involved than reads. NAND Flash memory is organized in pages, and a page needs to be *erased* before it can be *programmed* (i.e., set to a new value). Not only programming a page is much slower than reading it, but the erase operation needs to be performed in blocks of (hundreds or even thousands of) pages. Therefore, writes cannot be done in-place, and also involve a high cost as block erasures are two orders of magnitude slower than reading or programming a page. Second, each Flash cell can only sustain a finite number of erase cycles before it wears out and becomes unusable.

Flash translation layers (FTLs) [6] are introduced to address the above issues. In general, an FTL performs writes out-of-place, and maintains a mapping between logical and physical addresses. When space runs out, invalid data are garbage collected, and valid data are relocated to free blocks. To aid the garbage collection (GC) process, controllers keep a part of the drive's capacity hidden from the user (*overprovisioning*). The more space that is overprovisioned, the better the GC performs.

B. SMR disks

Magnetic disks remain the medium of choice for many applications [4], mainly due to low cost. However, magnetic recording is reaching its density scaling limits. To increase disk capacity, a number of techniques have been proposed, one of which, Shingled Magnetic Recording (SMR) [1], [31], has recently become widely available [32], [33]. SMRs gain density by precluding random updates. The density improvement is achieved by reducing the track width, thereby fitting more tracks on the surface of a platter, without reducing the write head size. As a result, while a track can be read as in conventional disks, it cannot be re-written without damaging adjacent tracks.

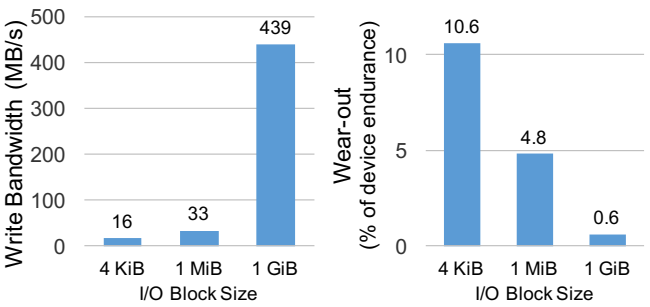
SMR disks are typically organized into zones. Zones are isolated from each other by guards, so that writes to a zone do not interfere with tracks on other zones. All the writes within a zone must be done in strict sequential order; however, multiple zones can be written to independently. These zones are called *sequential zones*. Drives, also, typically include a small number of *conventional zones*, where random updates are allowed.

Three categories of SMR drives exist based on where the TL is placed: drive-managed (DM), host-managed (HM), and host-aware (HA) SMRs [34]. In drive-managed disks, SMR complexities are fully hidden by a drive TL. On the other extreme, HM drives *require* a host TL to guarantee that writes within a zone will be sequential. HM drives provide a number of commands to the host, e.g., to reset a zone so that it can be re-written and to retrieve the drive's zone information. HA SMRs offer a compromise between DM and HM: they expose control commands, but can operate without a host TL.

C. Limitations of drive TLs

Demand to reduce costs per GB raises barriers to drive TL performance. SSD vendors increasingly offer commodity drives of higher densities at lower prices, without adding hardware resources (e.g., memory and compute) on the controller to deal with the extra capacity. Performance degrades further due to the use of consumer-grade Flash and low overprovisioning, as is typical in commodity SSDs. Furthermore, drive TLs are required to support a number of different workloads, and end up making compromises. Communicating application-specific hints to the drive is hard, if not impossible.

We illustrate the limitations of drive TLs on commodity SSDs by applying a random workload on a widely-used drive (after low-level formatting it) for tens of hours until performance stabilizes (see §V-A1 for details). We perform



(a) Write Bandwidth (b) Wear-out after 10 device writes

Figure 1: Random writes with a block size of 4KiB, 1MiB, and 1GiB

the experiment for three block sizes: 4KiB, 1MiB, and 1GiB. Fig. 1a shows the resulting (stable) write throughput, and Fig. 1b shows the wear induced to the device (as reported by SMART) after 10 full device writes. Our experiment, effectively, compares the drive TL performance under a random workload (4KiB) versus the ideal performance (1GiB), as well as an intermediate performance point (1MiB). A larger block size minimizes the need for the drive TL to perform GC: as the I/O size increases so does the probability that a write will entirely invalidate the Flash blocks it touches, eliminating the need for relocations. The drive TL fails to achieve high write bandwidth under unfavourable access patterns, only sustaining about 16MiB/s for 4KiB blocks, and 33MiB/s for 1MiB blocks. Interestingly, a block size of 1MiB is not large enough to bring the write performance of the drive to its ideal level; block sizes closer to the GiB level are required instead, which better reflects the native block size of modern dense Flash SSDs [35]. Furthermore, according to the SSD's SMART attributes, the write amplification for the 1GiB writes was $1.03\times$, whereas for the 4KiB writes it was $18.24\times$, and $8.26\times$ for 1MiB writes. We found that other commodity SSDs exhibit similar behavior, with write amplification factors as high as $50\times$. SMR drive TLs suffer from the same limitations as FTLs. As an example, we measured less than 200 KiB/s of random write bandwidth for 64KiB random writes to a drive-managed SMR disk (§V-A2). Overall, there seems to be significant room for improvement even for a single drive by employing a host TL that does its own relocations (additional reads and writes), but always writes sequentially to the device.

D. Why a host TL?

Vendors prioritize cost over performance for commodity drives, resulting in drives that are unfit for many applications that require high performance in terms of throughput and latency. Even simple techniques to alleviate this problem (e.g., configurable overprovisioning) are non-trivial to implement and rarely applied in practice.

We argue that a host TL can address these issues and improve efficiency. By transforming the user access pattern to be sequential, a host TL can realize significant performance and endurance benefits, enabling commodity drives to be used for datacenter applications even under demanding performance requirements. Furthermore, having visibility across multiple devices enables optimizations that are not possible from within a single drive. An evaluation of the Aerospike NoSQL store [36], for example, has shown the advantages of managing arrays of Flash SSDs as opposed to individual drives (e.g., by coordinating GC cycles across multiple devices).

Moreover, maximizing I/O performance for many application depends on exploiting workload properties. While this is difficult to do in a device TL, a host TL offers many such opportunities (e.g., improving performance by reducing persistence guarantees or sacrificing space). A host TL should be built as a framework that supports multiple types of storage, different policies and algorithms, and a wide range of configuration options. A host TL can, also, be extended and improved over time, allowing incremental adoption of advanced techniques, new storage technologies, and different tradeoffs.

Finally, and perhaps more importantly, a host TL allows combining multiple devices to build hybrid storage systems. By managing arrays of devices at the host, as opposed to a single device in the case of a drive TL, the TL can offer additional improvements by making global decisions, e.g., about balancing load and wear across devices. As the storage landscape increasingly diversifies, and applications require the use of different storage technologies, the existing TL indirection can be used for implementing hybrid storage policies. Under this scenario, a host TL is also applicable to technologies that are not, necessarily, idiosyncratic.

III. SALSA DESIGN

SALSA makes three principal design choices. First, it is *log-structured* [24], [25]. Among other benefits, this allows it to deal with storage idiosyncrasies. By only writing big sequential segments, SALSA renders the drive's GC irrelevant, as its task becomes trivial. When space runs out, SALSA does its own GC. Second, SALSA supports multiple storage types, and can combine them to build hybrid systems. Finally, SALSA follows a modular design so that it can be used as a framework for implementing a variety of policies and algorithms, enabling adaptability to different requirements.

From the perspective of the user, SALSA exposes block devices that can be used transparently by applications, e.g., by running a database directly on the device, or by creating a filesystem and running an application on top. An important benefit of SALSA is that it does not require any application modifications.

There are two main components in SALSA: the storage capacity manager (SCM), which is responsible for managing

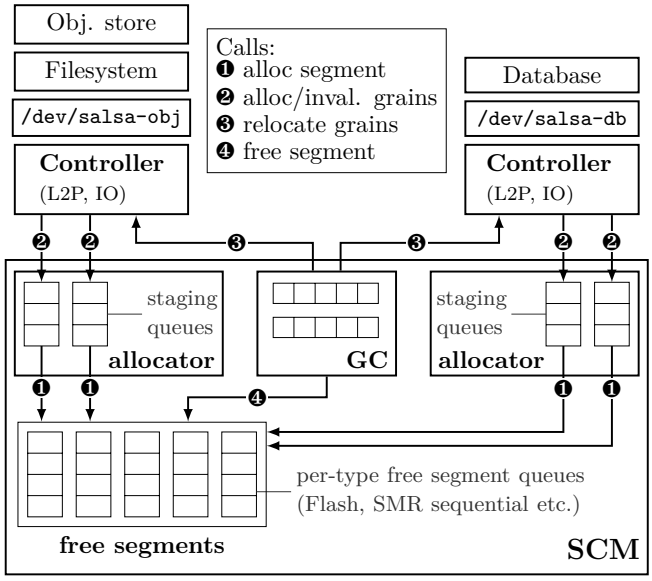


Figure 2: SALSA Architecture and allocation calls.

the underlying storage, and one or more controllers that operate on the SCM (Fig. 2). SCM is a common substrate that implements storage provisioning to controllers, GC, and other common functions. Controllers are responsible for implementing the storage policy, performing I/O, and mapping the logical (application) space to the physical space (*L2P*).

A. Storage Capacity Manager

Generally, the SCM manages multiple storage devices. To capture different device properties, SALSA defines appropriate storage types: NVM, Flash, HDD, SMR conventional, SMR sequential. At initialization, SCM identifies the type of its devices, and creates an address space that combines them. This address space is split into different areas, each characterized by a storage type. The boundaries of these areas are not necessary device boundaries: an SMR drive with both conventional and sequential zones, for example, is split into two different areas, one for each type of zone. The SCM address space is not exposed to the controllers.

SALSA can combine multiple devices linearly (i.e., appending one after another) or in a RAID-0, -1, or -5 configuration. Based on the configuration, a set of appropriate functions for performing I/O is provided to the storage controllers. For RAID-5, SALSA requires a small non-volatile memory buffer which is used to store the parity for the currently open stripe. For instance, the Persistent Memory Region of an NVMe SSD can be used for that purpose [37]. Parity is accumulated into that buffer as the stripe is being filled, and is committed to storage when it gets full. Thereby, SALSA avoids expensive read-modify-write operations, which are required with traditional (i.e., non log-structured) RAID-5.

The SCM physical storage space is divided into *segments*, large (e.g., 1GiB) contiguous regions of a single storage type. A segment can be in one of the following states: *free*: owned by the SCM, *staged*: used for storage allocation for a controller, or *allocated*: fully used, owned by a controller, and available for GC. Once allocated, a segment can only be used by a single controller.

Allocators allocate segments on behalf of the controllers via an interface (Fig. 2, ❶) that allows for specification of (hard and soft) constraints on the segment's storage type. To support this interface, SCM maintains multiple allocation queues that segregate segments based on the backing storage type. Each segment is divided into *grains*, the configurable smallest unit of allocation (e.g., 4KiB, 8KiB, etc.). Controllers allocate and free storage in grains (❷). Allocators maintain a number of *staged* segments and allocate space *sequentially* within them. We call this mechanism an *allocation stream*. Controllers can have multiple allocation streams, allowing for data segregation. SALSA can, for example, segregate writes by their update-frequency ("heat"), as well as segregate user and GC (i.e., relocation) writes. Each stream has its own constraints for segment allocation.

When a segment becomes full, its state transits to *allocated* and becomes a relocation candidate for the GC. For each segment, SALSA tracks the number of valid grains. Initially, all grains in a segment are valid. As data become invalid, controllers decrement the number of valid grains. When no valid grains remain, the segment is freed and returns to the SCM. Internal fragmentation can lead to inability to allocate new segments, even if there is enough free space. As is common in log-structured systems, we free space with a background GC process.

B. Garbage Collection (GC)

GC is responsible for relocating fragmented data to provide free segments. The SCM executes the GC algorithm that selects the best segments to relocate; GC operates across all devices but independently for each storage type. When a segment is selected, GC (up)calls the owning controller to relocate the valid data of this segment to a new one (❸). The GC is not aware of which grains are valid and which are not, nor the segment geometry in terms of page size, metadata, etc. This is left to the controller. Once the controller relocates data, it frees the corresponding grains, and, eventually, segments are freed and returned to their corresponding free queues (❹).

GC maintains a number of spare segments for relocation, because otherwise it will not be able to provide free segments for allocation. As with most TLs, SALSA overprovisions storage: it exposes only part of the device total capacity to the user, and uses the rest for GC.

Initially, all the device segments are free, and SALSA redirects user writes to free segments. When free segments run out, however, SALSA GC needs to perform relocations

to clean up segments. Relocations cause I/O amplification and the underlying devices serve both user and relocation traffic. SALSA uses two (configurable) watermarks: a low (high) watermark to start (stop) GC. For SMR sequential segments of host-managed drives, we reset the zone write pointers of a segment before placing it in the allocation queue. SALSA uses a generalized variant of the greedy [38] and circular buffer (CB) [24] algorithms, which augments a greedy policy with the aging factor of the CB. This aging factor improves the performance of the algorithm under a skewed write workload without hindering its performance under random writes.

C. LSA controller

SALSA supports multiple frontends, but in this paper we focus on the Linux kernel frontend where each controller exposes a block device. These controllers maintain a mapping between user-visible logical block addresses (LBAs), and backend physical block addresses (PBAs). We refer to them as Log Structured Array (LSA) [25] controllers. LSA controllers map LBAs to PBAs, with a flat array of 32 (default) or 64 bits for each entry (compile-time parameter). Larger blocks (or pages) require less space for the table, but lead to I/O amplification for writes smaller than the page size (e.g., read-modify-write operations for writes). For SSDs, the default page size is 4KiB, allowing us to address 16TiB (64ZiB for 64 bit entries) storage; for SMR drives, the default page size is 64KiB. Note that the page size has to be a multiple of the SCM grain size, in order to maintain interoperability with the GC and allocators. The mapping table is maintained in-memory, with an overhead of 4B per LBA (e.g., 1GiB DRAM per 1TiB of storage for 4KiB pages, 512MiB for 8KiB, etc.). A back-pointer table of PBA-to-LBA mappings is maintained per segment for GC and restore operations, and it is either always in-memory or is constructed on-demand by scanning the LBA-to-PBA table, based on a run-time configuration parameter.

Accesses and updates to the mapping table are done in a thread-safe lock-free manner using compare-and-swap. A read operation will typically read the table, perform a read I/O operation to fetch the necessary data, and return them to the user. A write operation will allocate new space, perform a write I/O operation to write user data to this space, and update the table entry. A relocation operation on a PBA will read the PBA-to-LBA back-pointer, check that the LBA stills maps to the PBA in question, read the data, allocate new space, write the valid data to a new location, and update the table only if the LBA still maps to the relocated PBA.

For sequential segments of host-managed SMRs we *force* the allocated pages to be written sequentially to the drive, to avoid drive errors. We do so via a thread that ensures that all writes to these segments happen in-order. This is not required for other storage types (e.g., SSDs), and we do not use the I/O thread for them.

D. Persisting metadata

The LSA controller we described so far maintains the LBA-to-PBA mapping in memory and dumps it to storage upon shutdown. To protect against crashes, controllers log updates to the mapping table. Under this scheme, a segment contains two types of pages: pages written by the user, and metadata pages that contain mapping updates. In the absence of flush operations (e.g., `fsync`), one metadata page is written for every m data pages (m is configurable at run-time for each controller). In case of a flush, a metadata page is written immediately. Therefore, SALSA provides the same semantics as traditional block devices that use internal write buffers. The metadata flush is handled differently for different storage *types*. For SMR storage, we pad segments so we adhere to the sequential pattern. For Flash, we update the metadata page in-place; although this might break the strict sequentiality of writes at the SSD level, flush operations are rare, and did not noticeably affect performance in any of our experiments. SALSA also maintains a configuration metadata page at each device, and a configuration metadata page per segment. The metadata overhead depends on the value of the m, on the segment size, and on the grain size. For 1GiB segments, $m = 512$ (default value), and the smallest possible grain size (4KiB), it amounts to 0.2% of the total capacity.

Upon initialization, we first check whether SALSA was cleanly stopped using checksums and unique session identifiers written during the LBA-to-PBA dumps. If a clean shutdown is detected, the mapping of each controller is restored. Otherwise, SALSA scans for metadata pages across all valid segments belonging to the controller, and restores LBA-to-PBA mappings based on back-pointers and timestamps. The SCM coordinates the restore process: it iterates over segments in parallel and upcalls owning controllers to restore their mapping.

E. Implementation notes

The core of SALSA is implemented as a library that can run in kernel- or user-space. Different front-ends provide different interfaces (e.g., a block device, or a key-value store) to the user. The Linux kernel block device interface is implemented on top of the device-mapper (DM) framework. SALSA controllers are exposed as DM block devices. Any I/O to these devices is intercepted by the DM and forwarded to the SALSA kernel module, which in turn remaps the I/O appropriately and forwards it to the underlying physical devices. Devices can be created, managed and destroyed, using the SALSA user interface tool (UI).

IV. Adapting to application workloads

Host TLs can be adapted to different application workloads, which we fully embrace in SALSA. At a first level, SALSA offers a large number of parameters for run-time configuration. Controllers parameters include: page size,

number of streams for user/GC writes, metadata stripe size, sets to specify storage types each controller can use, etc. Furthermore, SALSA includes multiple controller implementations, each with their own specific parameters. There are also global parameters: grain size, GC implementation (each with its own parameters), GC watermarks, etc. Users are not expected to understand these details: the UI provides sane default values. In practice, we have found this rich set of options extremely useful. Moreover, SALSA can be extended to adapt to specific workloads and meet different application requirements by implementing different controllers. For example, an RDMA interface to NVM storage has been implemented as a SALSA controller in FlashNet [39]. Next, we discuss two controller designs that we developed to address application-specific workloads.

A. Dual mapping controller

Our use-case is running an object store service on SMR drives. Because object stores perform fault management and load distribution, we run one SALSA instance per drive and let the upper layer balance load and deal with faulty drives. For object stores, 128KiB is considered a small object size [40]. Therefore, we can set the page size to 64KiB, leading to an overhead of 64MiB RAM per 1TiB of storage, making SALSA feasible even for servers that can contain close to a hundred drives [41].

During an initial evaluation, we observed a number of read-modify-write operations that degraded performance. We found two sources for this: writes that are not aligned to 64KiB, and filesystem metadata that are smaller than 64KiB. Even though the number of these operations is relatively small, they lead to a noticeable performance hit. We can avoid read-modify-write operations with a controller that supports a small number of sub-64KiB mappings, while using 64K pages for everything else. To that end, we develop a *dual mapping controller* that maintains two mappings: a sparse mapping for 4KiB pages and a full mapping for 64KiB. A read operation checks whether the pages exist in the sparse mapping *first* and if they do not, checks the full mapping. A write operation will use the full mapping for 64KiB pages, and the sparse mapping for smaller pages. If the sparse mapping does not contain a page during a write and has no free locations, we perform a read-modify-operation and update the full mapping.

B. Hybrid controller

Hybrid systems that make use of multiple storage types allow tradeoffs that are not possible otherwise, offering great opportunities for maximizing the system's utility [3], [42]. As storage diversity increases, we expect the importance of hybrid systems to rise. For example, vendors offer hybrid drives (Solid State Hybrid Drives – SSHDs) that combine a disk and NAND Flash within a single drive [43], [44]. These drives, however, have hard-coded policies and cannot be repurposed.

Multi-device host TLs enable building better hybrid systems. In contrast to a device TL, a host TL can support multiple devices from different vendors. Moreover, the indirection and mechanisms employed by a TL like SALSA can be reused, enabling transparent data relocation between different media. Finally, as we argued throughout this paper, a host implementation offers additional flexibility, as well as co-design potential [45], compared to a drive implementation.

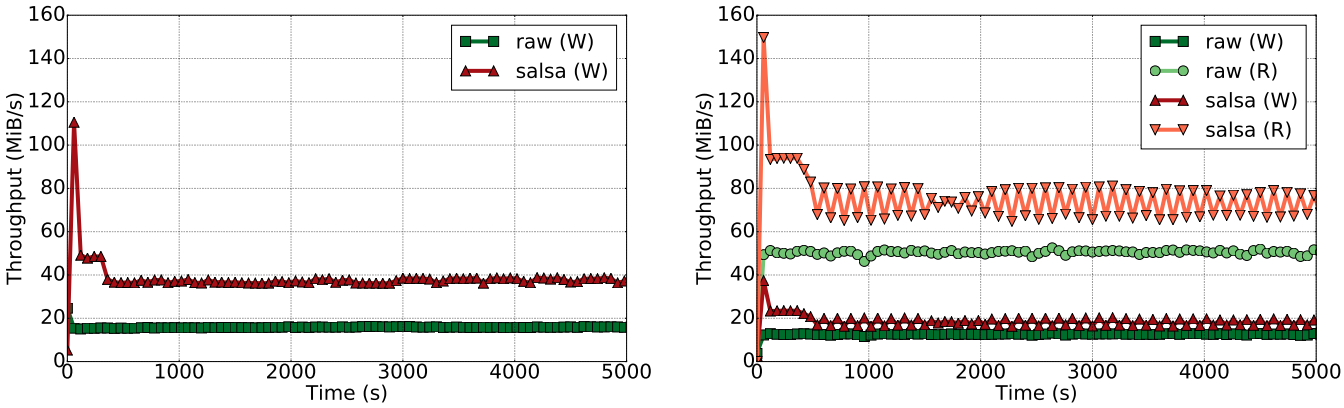
We consider a video service for user-generated content (e.g., YouTube). Because user-generated videos are typically short [46], and will get striped across a large number of disks, reading them from a disk will result in reduced throughput due to seeks. Because most views are performed on a relatively small subset of the stored videos, there is an opportunity to optimize the read throughput by moving them into a faster medium. If the read working set does not fit to DRAM, moving data to an SSD is the best solution. The next section presents a SALSA controller implementing this functionality.

For our hybrid controller, we configure two allocation streams: one for fast storage (Flash) and one for slow storage (disks). User and GC relocation writes always allocate from the slow storage stream, while the fast storage stream is used for relocating "hot" pages that are frequently accessed. To determine the "hot" pages we maintain a data structure with a "temperature" value of each (logical) page. We use an array of 64-bit values sized at the number of logical pages divided by 256 (configurable) to reduce memory overhead. Because most files are stored sequentially on the block device, we map consecutive pages to the same temperature.

At each read, we increase the temperature by one. Because we use 64-bit values, overflowing is not an issue. If a page is discarded, we set the temperature to zero. We also periodically (once a day) halve the temperature of all values. When we increase the temperature, we check the new value against a configurable threshold. If the threshold is reached and the page is not already located into Flash storage, we schedule a relocation. The relocation happens asynchronously on a different (kernel) thread to avoid inducing overhead to the read operation. If at any point something goes wrong (e.g., there are no available Flash physical pages, the mapping or temperature changed in the meantime) the operation is aborted.

V. Evaluation

We start our evaluation (§V-A) discussing the performance and durability benefits of SALSA using a random workload. Subsequently, we show how SALSA features can benefit real-world applications. In §V-B, we evaluate the benefits of SALSA's single- and multi-device optimizations using MySQL containers on SSDs. In §V-C, we evaluate the



(a) 100% writes

(b) 20% writes - 80% reads

Figure 3: 4KiB uniform random workload on an SSD

dual-mapping controller (§IV-A) using Swift. We use both MySQL and Swift to evaluate the benefits of supporting multiple application policies in §V-D. Finally, in §V-E, we evaluate the hybrid controller (§IV-B) for a user-generated video service.

SSD experiments (§V-A1, §V-B, §V-D) are performed on a 16 core dual-node x86-64 server with 128GiB RAM running RHEL 7.2 with a 3.10 Linux kernel, using a widely-used off-the-shelf 2.5" 1TB SATA NAND Flash SSD. The SMR experiments (§V-A2, §V-C, §V-E) are performed on a 4 core x86-64 server with 20GiB RAM running RHEL 6.6 with a 4.1 kernel, with a drive-managed 8TB SMR drive.

A. Microbenchmarks

To evaluate the benefits of placing the TL on the host, we compare the performance and endurance of SALSA against raw SSD and SMR drives under a sustained random workload. We use this workload because random writes are highly problematic for SSDs and SMRs for two reasons. First, GC runs concurrently with user operations and causes maximum disruption. Contrarily, in a bursty workload, GC would have time to collect between bursts. Second, random writes across the whole device maximize write amplification. We use a microbenchmark that applies uniformly random read and writes directly (using `O_DIRECT`) to the device. We measure device throughput using `iostat` [47].

1) SSDs: We low-level format the drive before our experiments. We overprovision SALSA with 20% of the SSD's capacity. We facilitate a fair comparison by performing all measurements on the raw device on a partition with equal size to the SALSA device. That is, we reserve 20% space on the raw device which is never used after low-level formatting the drive. The 20% overprovision was chosen to offer a good compromise between GC overhead and capacity utilization [48]. To measure stable state, we precondition the device (both for raw and SALSA) by writing all its capacity once in a sequential pattern, and once in a uniformly random pattern. Subsequent random writes use *different* patterns.

	write-only	read-mostly	
throughput	W:100%	R:80%	W:20%
raw	15.9 ± 0.2	50.6 ± 0.9	12.6 ± 0.2
salsa	37.7 ± 0.9	72.5 ± 6.2	18.1 ± 1.5

Table I: Average throughput (MiB/s) and standard deviation for two random workloads on an SSD: 100% writes and 80%/20% reads/writes. Block size is 4KiB.

We consider two workloads: *write-only* (100% writes) and *read-mostly* (80% reads, 20% writes), both with 4KiB blocks and queue depth (QD) of 32. The benefits of SALSA in a *read-mostly* workload are smaller because read operations do not directly benefit from SALSA and write amplification due to GC having a smaller impact when writes are infrequent.

Stable state throughput over time is shown in Fig. 3, and the average throughput is reported in Table I. SALSA achieves 2.37× better average throughput than the raw device for a write-only workload. For a read-mostly workload, SALSA improves both read and write throughput by 1.43×. We attribute the worse read throughput of the raw device to obstruction caused by the drive GC that stalls reads. Moreover, we have extensively experimented with more than 20 commodity SSDs. Among those, using 20% overprovisioning, SALSA improves throughput on a sustained random write workload by a factor of 1.5×-3×.

Next, we compare endurance when using SALSA against using the raw drive. We measure wear via a SMART attribute that, according to the device manufacturer, increases linearly with the wear (Program/Erase cycles) of the Flash cells. We low-level format the drive and fill it up once sequentially. Subsequently, we perform 10 full device random writes with 4KiB. We record the wear of the device after each full device write (11 data points including the initial sequential

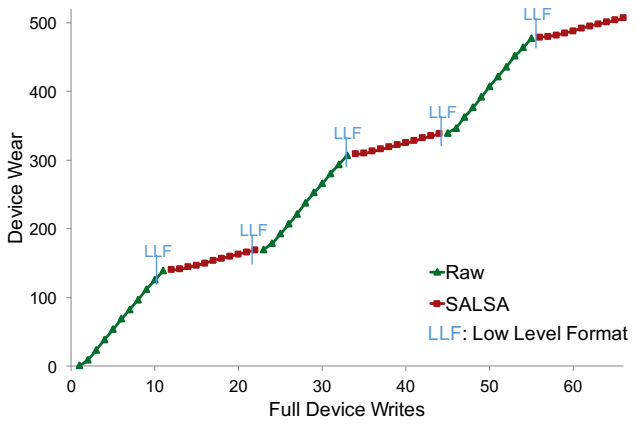


Figure 4: SSD wear with and without SALSA.

write). We repeat the experiment 6 times alternating between runs on the raw device and runs on SALSA. As before, experiments on the raw device were performed on a partition equal to the SALSA device size, so a full device write amounts to the same amount of data in both cases.

Results are shown in Fig. 4. Overall, the same workload incurs $4.6\times$ less wear to the device when running on SALSA compared to the raw device. In this experiment, we measured a write amplification of 2.5 on average for SALSA (which is very close to the theoretically expected 2.7 for random writes and chosen overprovision [48]), which implies that the internal drive write amplification was $11\times$ less compared to the raw device experiment; SALSA wrote $2.5\times$ the user data and still induced $4.6\times$ less total device writes compared to the raw device, suggesting that the total device writes for the raw device was $2.5 \times 4.6 \approx 11\times$ the user data. Note that the internal drive write amplification typically includes metadata (and possibly data) caching on top of GC traffic; in fact, since the GC traffic should be similar between the two experiments for random writes, we attribute most of the extra amplification to this cache traffic. Results were repeatable over multiple executions of the experiment, and other commodity SSDs we examined behaved similarly.

2) SMRs: We now turn to SMR drives, comparing the performance of SALSA against the raw device using 64KiB uniform random writes with QD1 across the whole device.

We use SALSA with all SMR variants (drive-managed, host-aware, and host-managed) across multiple vendors. Here, we present results for a drive-managed SMR, because we can directly compare against the drive's TL by applying the same workload on the raw device.[1] A drive-managed SMR, however, limits SALSA because it does not expose drive information (e.g., zones) and cannot be directly controlled (e.g., does not allow resetting write pointers). Instead,

[1]The host-aware SMR drives that we tested were vendor samples, and therefore might not be representative of the final products. For this reason we opted to present results for widely-available drive-managed SMRs.

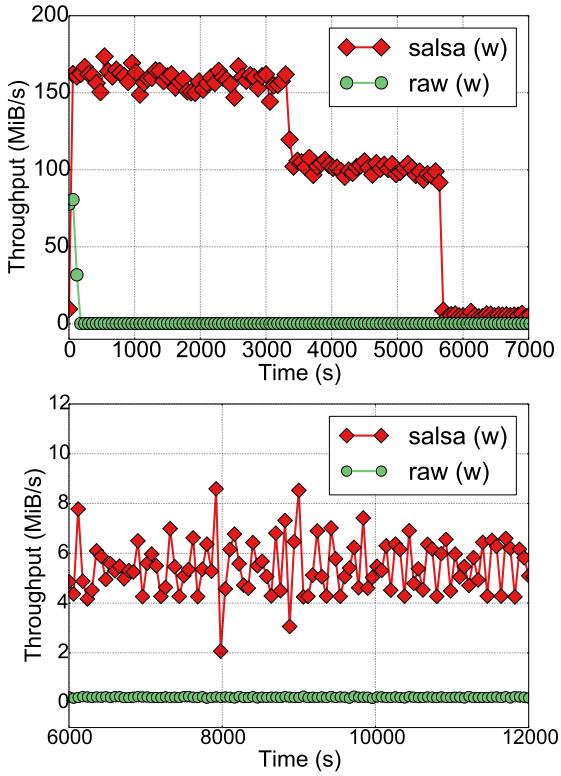


Figure 5: 64KiB random writes on a host-managed SMR with and without SALSA. The raw results are after the first write on the device, while the SALSA results are after the whole device was randomly written once. The top plot shows the 0-7 Ksecs area, while the bottom focuses on the 6-12 Ksecs area.

similarly to SSDs, SALSA writes sequentially to minimize the drive's TL interference. We overprovision SALSA by 10%, and low-level format the drive before the experiments. We select this value with a steady-state random workload in mind; for other workloads (e.g., read-mostly or sequential) smaller values might offer a better tradeoff.

The results are shown in Fig. 5. The raw device throughput starts close to 80MiB/s but drops to 200 KiB/sec after about 5 minutes, which renders the device effectively unusable for many applications. We attribute the drop in performance to the persistent cache of the drive, as identified by prior work [49], [50]: after the persistent cache is filled ($\sim 1.4GiB$ of random 64KiB writes [49]), then the drive starts its cleanup process, which entails read-modify-writes on MiBs of data.

Contrarily, SALSA's performance does not degrade that quickly. Hence, to facilitate an easier comparison Fig. 5 presents SALSA throughput results *after* a full device (random) write. We observe three phases in SALSA performance. During the first and second phase, no GC is performed. Initially, caching on the drive allows an initial

throughput of roughly 160MiB/s, which drops to 100MiB/s after about 3K seconds. This designates the SALSA performance for bursts up to an amount of data equal to the difference between the high and low GC watermarks. In the third phase, GC starts and the throughput of the SALSA device becomes roughly 5 MiB/s, 25× better than the throughput of the raw drive.

B. Containerized MySQL on SSDs

In this section, we evaluate the effect of SALSA's single- and multi-device optimizations on the performance of a real-world database. Specifically, we deploy multiple MySQL Docker containers on commodity SSDs in a single- and a multi-device (RAID-5) setup, and execute an OLTP workload generated by sysbench [51].

We evaluate 5 container storage configurations: three with 1 SSD (raw device, F2FS [10], and SALSA), and two with 4 SSDs using RAID-5 (Linux MD [52] and SALSA equivalent). We use the same hardware and setup (formatting, partitioning, preconditioning) as in V-A1. For F2FS, we also allocate the same over-provisioning as the other deployments: 20% using the `-o 20` option when creating the filesystem with `mkfs.f2fs`. We only use F2FS in the single device deployment, since it did not provide a native RAID-5 equivalent multi-device deployment option. The only difference across our experiments is the device we use (a raw device partition, a SALSA device, or an MD device). In this device we create one (logical) volume per MySQL instance to store database data. We use the Ubuntu 14.04 image provided by Docker, adding the necessary packages for our experiment. We deploy four containers with one multi-threaded MySQL server per container. Each server uses a 160GiB database image which we place on the corresponding volume. On each container, we run 4 sysbench threads to maximize IO throughput. We use the default LSA controller (§III-C) for SALSA.

raw			F2FS			SALSA		
tps	avg	95%	tps	avg	95%	tps	avg	95%
22.2	180ms	651ms	25.6	157ms	599ms	37.4	107ms	266ms
21.3	188ms	655ms	25.6	156ms	599ms	37.6	106ms	264ms
21.2	188ms	656ms	25.5	157ms	596ms	37.7	106ms	264ms
21.2	188ms	654ms	25.6	157ms	603ms	39.1	102ms	258ms

(a) 1 SSD

Linux MD			SALSA		
tps	avg	95%	tps	avg	95%
8.1	2.0s	5.3s	287.2	55.7ms	99.5ms
8.1	2.0s	5.3s	290.5	55.1ms	98.4ms
8.3	1.9s	5.2s	286.5	55.9ms	99.9ms
7.8	2.1s	5.6s	291.1	55.0ms	98.2ms

(b) 3+1 SSDs RAID-5: Linux MD and SALSA

Table II: Sysbench results for each MySQL instance: throughput in transactions per second (*tps*), average (*avg*) and 95th percentile (*95%*) response times.

Results for one SSD, as reported by each sysbench instance are shown in Table IIa. Fig. 6a depicts sysbench throughput over time for each instance. SALSA improves throughput by 1.68×, and the average latency by 1.69× compared to raw device, illustrating the benefits of implementing a TL on the host, instead of the device where resources are limited. Also, SALSA provides an improved throughput by 1.47× compared to F2FS, at a reduced tail latency (95% percentile) of 2.45×. We attribute the improvement against F2FS mainly to two reasons: (i) F2FS uses a 2MiB segment size which is not optimal for modern commodity SSDs II-C, compared to segments at the GiB level for SALSA, and (ii) F2FS updates its metadata in separate write-logs and at eventually in-place [10] which further reduce the effective sequential I/O size as received at the drive TL level; large, uninterrupted sequential overwrites are essential to achieve the ideal write performance of non-enterprise grade SSDs [35].

Fig. 6b and Table IIb show results for four SSDs in a RAID-5 configuration, using Linux MD and SALSA. SALSA increases throughput by 35.4× and improves the average response time by 36.8×. These results showcase the significant benefits of a TL that is multi-device aware. While SALSA can guarantee full-stripe writes with a small persistent buffer, in-place update approaches such as Linux MD cannot, because that would require a buffer with size comparable to device capacity. Hence, in-place updates in Linux MD trigger read-modify-write operations that lead to response times in the order of seconds, rendering this setup unsuitable for many applications.

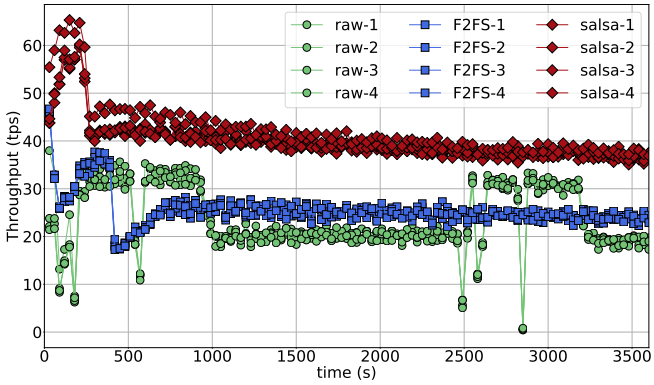
We also note that the performance difference between SALSA for one device and RAID-5 is due to the lower GC pressure in the latter case, since the RAID-5 configuration has 3 times the capacity of the single device configuration while the working set size does not change across the two tests. Contrarily, the Linux RAID-5 implementation has lower throughput than the single device, due to the parity updates and read-modify-write operations, which also slow down dependent reads.

Finally, the CPU overhead is negligible. In the RAID-5 configuration, we measured an overhead of less than 6% in normalized CPU utilization (CPU utilization / TPS) compared to the raw Linux MD configuration.

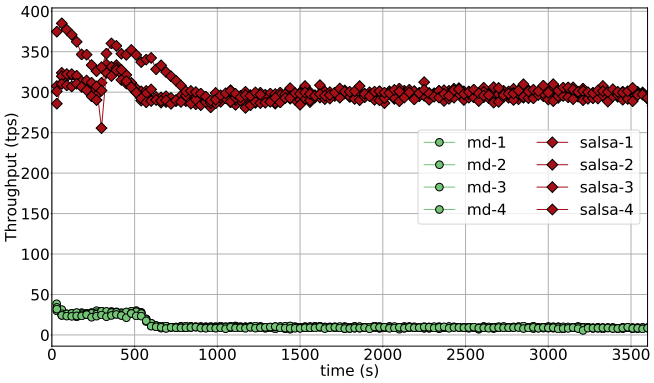
C. Object store using SMR drives

A host TL enables workload-specific optimizations. We evaluate the benefits of such an approach by running an object store on SMR disks, comparing the SALSA dual-mapping controller (§IV-A) against the raw device.

We use Openstack Swift, a popular open-source eventually-consistent object store [26]. Swift is written in Python and includes mutliple services. For our evaluation we focus on the object server [53], the component that stores, retrieves, and deletes objects on local devices. Objects are



(a) 1 SSD: raw device, F2FS, and SALSA



(b) 4 SSDs: Linux md software-RAID 5 and SALSA RAID-5 equivalent

Figure 6: Throughput of sysbench during execution

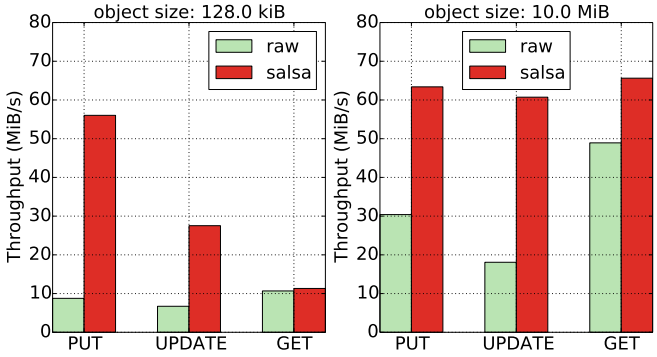


Figure 7: Swift storage server throughput for different operations, comparing the raw device and SALSA.

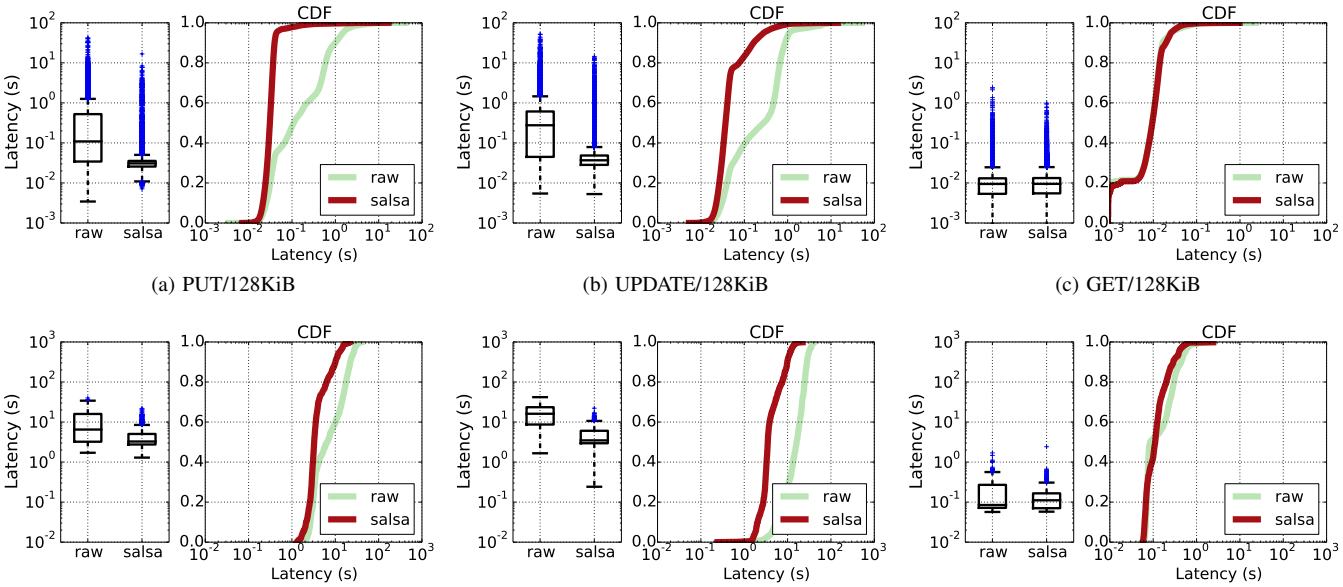
stored as files on the filesystem, while object metadata are stored in the file's extended attributes. For both experiments, we use an XFS filesystem configured per the Swift documentation [54] on the same SMR drive as §V-A2. To isolate storage performance, we wrote a Python program that issues requests directly to the object server. Swift uses "green threads", i.e., collaborative tasks, to enable concurrency. We do the same, using a 32 green thread pool for having multiple requests in flight.

We initialize the object store via PUT operations, so that the total data size is 64GiB. We subsequently update (UPDATE), and finally retrieve (GET) all the objects. The last two operations are performed in different random orders. We clear the filesystem and block caches before starting each series of operations.

Fig. 7 shows the throughput of the raw drive and SALSA for 128KiB and 10MiB objects for each different operation. (These sizes were found to represent small and large object sizes in the literature [40].) For small objects, using the raw device leads to low throughput for both PUTs and UPDATEs: 8.8 and 6.7 MiB/s. We attribute the poor performance to the drive having to write different files, located at different extents, potentially triggering relocation. SALSA, on the other hand, achieves higher throughput: 56 MiB/s for PUTs (6.36×) and 27.5 MiB/s for UPDATEs (4.1×). UPDATEs exhibit lower performance for both systems since the file that represents the object needs to be modified. GET performance is similar for both systems: 10.7 for raw and 11.3 MiB/s for SALSA. For large objects the behaviour for PUTs and UPDATEs is similar, but the difference between the raw device and SALSA is smaller. For PUTs SALSA achieves 63.4 MiB/s, 2× higher than the raw device (30.4 MiB/s); for UPDATEs the respective numbers are 60.7 MiB/s and 18.1 MiB/s, a 3.35× improvement for SALSA. SALSA results in better throughput for the GET operation of large objects at 65.6 MiB/s, while the raw device is at 48.9 MiB/s. We believe this is because XFS uses multiple extents for large files. In SALSA, these extents end up being close together even if they have different logical addresses, thus minimizing seek time when accessing them.

In addition to throughput, we sample the operation latency every ten operations and summarize the results in Fig. 8, using a box plot and a CDF diagram for each operation type. Because latency has a wide range of values, we use a logarithmic scale. For small objects, SALSA results in a lower median latency for both PUT and UPDATE operations: 30.8ms and 36.8ms. Using the raw device leads to much higher latencies: 109ms for PUT (3.5× higher) and 276ms for UPDATE (7.5× higher). Both SALSA and raw have a similar median latency for GET: 9.5ms. For large objects, SALSA still achieves a significantly lower median latency that the raw device. The median latency for a PUT on the raw device is close to 2× higher than SALSA (6.5s versus 3.3s), while for UPDATEs raw is 4.6× higher than SALSA (16.1s versus 3.5s). The raw device achieves an improved latency of 84.8ms for GET compared to SALSA that achieves 111.1ms, but as shown in Fig. 8, the raw device has a wider spread.

The relation between latency and throughput is different



(a) PUT/128KiB (b) UPDATE/128KiB (c) GET/128KiB

(d) PUT/10MiB (e) UPDATE/10MiB (f) GET/10MiB

Figure 8: Swift storage server latency for different operations, comparing the raw device and SALSA. The box is placed in the first and third quartiles, the line inside the box is the median, and the whiskers are at 1.5 IQR.

for `GET`s and write operations (`PUT`s and `UPDATE`s). In small objects, for example, `GET`s have lower throughput even though they have lower latency. This is because write operations allow higher concurrency. Swift performs writes by placing the data into a temporary file, updating metadata, calling `fsync`, and finally moving the file in its proper location using `rename`. The final steps are offloaded to another thread and execution continues with the next request. Only after a number of subsequent requests are serviced, the initial requests will be allowed to continue execution and complete, even if the `rename` call was completed before that. This approach enables high-throughput but can significantly hurt latency.

D. Multiple TLs for mixed workloads

SALSA enables different policies over a storage pool by decoupling storage policy and space management. Each policy is implemented as a different controller (TL) that exposes a different device to the user. In this section, we evaluate the benefits of this approach by deploying two different controllers on an SSD. Specifically, we run a containerized MySQL database on an LSA controller (§III-C) with an 8KiB page size to match the database page size, and a Swift object storage system on a dual-mapping controller (§IV-A) on the same SSD. We compare this approach against two others that use traditional partitions (one for each application): the raw device, and the LSA controller (configured with the default 4KiB page size). We run the mixed workload comprising sysbench OLTP and a object store `PUT` workload with 128KiB objects for 30 minutes and evaluate the different configurations based on memory footprint and application performance. We further compare the observed relocation traffic for SALSA under the two configurations.

Table III summarizes the results. For F2FS, we include results with 1MiB objects since under 128KiB objects its performance was low (17.46 sysbench tps, and 6.7MiB/s object write throughput), due to stressing the file creation scalability of the filesystem at hundreds of thousands of files [55], which was not the aim of this evaluation. Both SALSA configurations maintain a performance improvement similar to the single-SSD experiments presented in Sections V-A1 and V-B, both against the raw device and against F2FS. By using a separate controller tailored to each application, the dual controller setup realizes slightly higher performance than the default single LSA controller setup with 4KiB page size. More importantly, it does so at a significantly lower overhead, both in terms of DRAM (60%) and storage capacity (71%).

Moreover, the dual controller configuration provides segregation of the different applications' data, as each controller appends data to separate segments (by using separate allocators Fig. 2). This data segregation allows the dual-controller configuration to perfectly separate the data of the object store from the data of the MySQL database. The result is a relocation traffic that is reduced by 28% compared to the single-controller configuration. In this particular case, this reduction does not translate to significant bottom line performance improvement, because relocations comprise a

	raw	F2FS	salsa-single	salsa-dual
sysbench (tps)	20.3	20.4	34.05	35.90
Swift PUT (MiB/s)	25.5	34.28	37.29	38.19
DRAM overhead (GiB)	NA	0.85	1.66	0.68
MD overhead (GiB)	NA	2.11	1.82	0.53
Relocations (MiB/s)	NA	NA	2.48	1.78

Table III: Mixed workload results over raw device (*raw*), over the F2FS filesystem (*F2FS*), SALSA with 1 controller (*salsa-single*) and SALSA with 2 controllers (*salsa-dual*): sysbench throughput in transactions per second (*tps*), Swift object server PUT throughput, DRAM overhead, Metadata (*MD*) capacity overhead, and relocation traffic.

small component of the total write workload (7% for the single controller setup) which is expected considering that most of the write workload is sequential (object PUTs). Furthermore, the SSD we use does not offer control over the write streams to the host. Such control, e.g., in the form of the Write Streams Directive introduced in the latest version of the NVMe interface [37], would substantially increase the benefit from stream separation at the host TL level.

E. Video server with SMRs and SSDs

SALSA also supports controllers that combine different storage media. Here, we evaluate the benefits of running a server for user-generated videos on a SALSA hybrid controller that combines SSD and SMR drives. We compare three configurations: using the raw SMR device, using the SMR device over SALSA, and using a SALSA hybrid controller that employs both a Flash drive and an SMR drive as described in §IV-B.

We use an XFS filesystem on the device, and we generate the workload using Filebench [56]. Filebench includes a video-server macro-workload that splits files into two sets: an active and a passive set. Active files are read from 48 threads, and a file from the passive set is replaced every 10 seconds. User-generated videos are typically short, with an average size close to 10MiB [46]. Furthermore, videos are virtually never deleted, and most views happen on a relatively small subset of the stored videos. Subsequently, we modify the workload to use smaller files (10MiB), create new files instead of replacing files from the passive set every 1 second, and use direct IO for reads to avoid cache effects.

We run the benchmark for 20 min and show the throughput as reported by Filebench on Table IV. The write throughput remains at 10 MiB/s for all cases since we are writing a 10MiB file every second. Using SALSA over the SMR drive delivers a higher read throughput (6.2 MiB/s versus 4.8 MiB/s) because the periodical writes are less obstructive to the reads. The hybrid controller achieves a much higher read throughput of 118.5 MiB/s by using an SSD to hold the "hot" files.

Fig. 9 gives more insight on the operation of the hybrid controller by showing the read and write throughput of the

throughput	R (MiB/s)	W (MiB/s)
raw	4.8	10.1
salsa	6.2	10.1
salsa-hybrid	118.5	10.0

Table IV: Read (*R*) and write (*W*) throughput of video server macro-benchmark workload.

SSD and SMR drives as reported by iostat for the duration of the benchmark (we use a logarithmic scale on the y axis for clarity). Initially, all files are on the SMR drive. As the active videos are accessed by the reader threads, data migrates to the SSD and we observe SSD writes. After about 200 seconds, we reach stable state where all active videos are in the SSD. At this point, writes are served by the SMR and reads by the SSD.

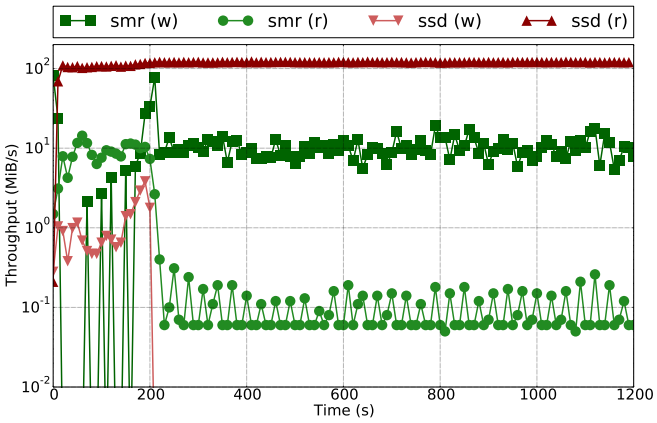


Figure 9: Read and write throughput of the SSD and SMR drive for the video server macro-benchmark when using the SALSA hybrid controller.

VI. Related work

The log-structured filesystem design was proposed independently of SSDs, as a way to increase write bandwidth [24]. Subsequent research work has proposed Flash-tailored log-structured filesystems to increase performance either on top of an FTL [9], [10], [57] or by accessing Flash memory directly [8], [58]. Menon introduces log-structured arrays implemented in the storage controller, as an alternative to RAID [25]. The same approach is followed in Purity for an enterprise all-Flash array [59]. All the above systems adopt append-only writes as a way to minimize random writes on Flash and increase performance. In our work, we follow a similar approach, but we tailor it to low-cost commodity devices, while also supporting multiple storage types.

A number of works have identified the limitations of SSD drive TLs, proposing offloading functionality to the host. Jeong et al. [60] propose caching the address mapping table in host memory, illustrating the problems of

limited drive controller resources. The Virtual Flash Storage Layer (VFSL) [8], [22], [61] is an attempt to place the TL on the host, exporting a large, virtual block address space that enables building Flash-friendly applications [16]. LSDM [62] is a host log-structured TL that targets low-cost SSDs. Recently, Linux introduced a TL for zoned drives [63] that exclusively targets zoned storage types (e.g., HA or HM SMR). While our motivation is common with these host TLs, SALSA is fundamentally different from in two ways. First, SALSA is designed to support multiple storage types and devices, using a common codebase to operate on them. Second, the aforementioned works implement a single TL layer which all applications use. In SALSA, contrarily, we concede that no single TL implementation is best for all cases. Instead, SALSA allows for multiple TL implementation instances (resulting in multiple volumes, for example) on top of a common SCM layer.

In a similar spirit, recent attempts expose the internal storage complexities (e.g., Flash channels [19], [20], or GC controls [64]) to enable host software to make more intelligent decisions and reduce controller costs. We view these efforts as orthogonal to ours: SALSA can operate on and benefit from these drives, but does not depend on them. Similarly, we view attempts to redefine the interface between applications and idiosyncratic storage [21], [23], [58], [65], [66] also as orthogonal. Currently, SALSA controllers offer a traditional interface because we target unmodified applications. Improved interfaces can be implemented (and co-exist) by individual controllers.

A hybrid system with Flash and disk is presented in [67] for database storage, where a cost-based model is used to decide which pages to store on Flash and which pages to store on disk. SALSA is different in that it focuses on actively transforming the workload to achieve higher performance (and, thus, lower cost) from the devices using a log-structured approach. A hybrid approach that we have not investigated is realized by Griffin [68] that uses HDDs as a write-cache for SSDs. Another hybrid approach is taken by Gecko [69], where a log-structured array on top of HDDs in a single TL layer is implemented, augmented by RAM- and SSD-based caching. SALSA, on the other hand, operates on SSDs and SMRs, does not rely on data caching, and supports multiple TL implementation instances.

VII. Conclusion and Future Work

In this paper we presented SALSA, a log-structured host TL that that can be used transparently from applications and offers significant performance and durability benefits for SSDs and SMR drives.

While we focus on SSDs and SMRs due to their idiosyncrasies in this paper, we believe that SALSA is also useful for other types of storage. On one hand, a log-structured TL has significant benefits even in non-idiosyncratic storage like DRAM [70] or non-volatile memory [71], [72]. On the other hand, coupled with proper policies, a host TL like SALSA can enable smart data movement between different storage types. We plan to expand on these ideas in future work. Moreover, in ongoing work we explore buidling SALSA-native applications that execute SALSA as a library in user-space. Among other benefits, this allows avoiding kernel overheads by utilizing user-space I/O drivers such as the Storage Performance Development Kit (SPDK) [73].

References

[1] R. Wood, M. Williams, A. Kavcic, and J. Miles, "The feasibility of magnetic recording at 10 terabits per square inch on conventional media," *Magnetics, IEEE Transactions on*, vol. 45, no. 2, pp. 917–923, Feb 2009.

[2] M. Nanavati, M. Schwarzkopf, J. Wires, and A. Warfield, "Non-volatile storage," *Commun. ACM*, vol. 59, no. 1, pp. 56–63, Dec. 2015. [Online]. Available: http://doi.acm.org/10.1145/2814342

[3] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li, "RIPQ: Advanced photo caching on flash for facebook," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, Feb. 2015, pp. 373–386. [Online]. Available: https://www.usenix.org/conference/fast15/technical-sessions/presentation/tang

[4] E. Brewer, L. Ying, L. Greenfield, R. Cypher, and T. T'so, "Disks for data centers," Google, Tech. Rep., 2016.

[5] T.-S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, "A survey of flash translation layer," *J. Syst. Archit.*, vol. 55, no. 5-6, pp. 332–343, May 2009. [Online]. Available: http://dx.doi.org/10.1016/j.sysarc.2009.03.005

[6] D. Ma, J. Feng, and G. Li, "A survey of address translation technologies for flash memories," *ACM Computing Surveys (CSUR)*, vol. 46, no. 3, p. 36, 2014.

[7] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV, 2009, pp. 229–240. [Online]. Available: http://doi.acm.org/10.1145/1508244.1508271

[8] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn, "DFS: A file system for virtualized flash storage," *Trans. Storage*, vol. 6, no. 3, pp. 14:1–14:25, Sep. 2010. [Online]. Available: http://doi.acm.org/10.1145/1837915.1837922

[9] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom, "Sfs: Random write considered harmful in solid state drives," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST'12, 2012, pp. 12–12. [Online]. Available: http://dl.acm.org/citation.cfm?id=2208461.2208473

[10] C. Lee, D. Sim, J. Hwang, and S. Cho, "F2FS: A new file system for flash storage," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, Feb. 2015, pp. 273–286. [Online]. Available: https://www.usenix.org/conference/fast15/technical-sessions/presentation/lee

[11] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee, "CFLRU: A replacement algorithm for flash memory," in *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, ser. CASES '06, 2006, pp. 234–241. [Online]. Available: http://doi.acm.org/10.1145/1176760.1176789

[12] H. Kim and S. Ahn, "BPLRU: A buffer management scheme for improving random writes in flash storage," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, ser. FAST'08, 2008, pp. 16:1–16:14. [Online]. Available: http://dl.acm.org/citation.cfm?id=1364813.1364829

[13] M. Saxena and M. M. Swift, "FlashVM: Virtual memory management on flash," in *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, ser. USENIXATC'10, 2010, pp. 14–14. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855840.1855854

[14] B. Debnath, S. Sengupta, and J. Li, "Flashstore: High throughput persistent key-value store," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 1414–1425, Sep. 2010. [Online]. Available: http://dx.doi.org/10.14778/1920841.1921015

[15] ——, "Skimpystash: Ram space skimpy key-value store on flash-based storage," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '11, 2011, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/1989323.1989327

[16] L. Marmol, S. Sundararaman, N. Talagala, R. Rangaswami, S. Devendrappa, B. Ramsundar, and S. Ganesan, "NVMKV: A scalable and lightweight flash aware key-value store," in *6th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 14)*, Jun. 2014. [Online]. Available: https://www.usenix.org/conference/hotstorage14/workshop-program/presentation/marmol

[17] P. Wang, G. Sun, S. Jiang, J. Ouyang, S. Lin, C. Zhang, and J. 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*, ser. EuroSys '14, 2014, pp. 16:1–16:14. [Online]. Available: http://doi.acm.org/10.1145/2592798.2592804

[18] R. Pitchumani, J. Hughes, and E. L. Miller, "SMRDB: key-value data store for shingled magnetic recording disks," in *Proceedings of the 8th ACM International Systems and Storage Conference*, ser. SYSTOR '15, 2015, pp. 18:1–18:11. [Online]. Available: http://doi.acm.org/10.1145/2757667.2757680

[19] J. Ouyang, S. Lin, S. Jiang, Z. Hou, Y. Wang, and Y. Wang, "SDF: Software-defined flash for web-scale internet storage systems," in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '14, 2014, pp. 471–484. [Online]. Available: http://doi.acm.org/10.1145/2541940.2541959

[20] M. Bjørling, J. Gonzalez, and P. Bonnet, "Lightnvm: The linux open-channel SSD subsystem," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 359–374. [Online]. Available: https://www.usenix.org/conference/fast17/technical-sessions/presentation/bjorling

[21] S. Lee, M. Liu, S. Jun, S. Xu, J. Kim, and Arvind, "Application-managed flash," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016, pp. 339–353. [Online]. Available: http://usenix.org/conference/fast16/technical-sessions/presentation/lee

[22] Gary Orenstein, "Optimizing I/O operations via the flash translation layer," Flash Memory Summit, 2011, http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2011/20110809_F1B_Orenstein.pdf.

[23] Y. Lu, J. Shu, and W. Zheng, "Extending the lifetime of flash-based storage through reducing write amplification from file systems," in *11th USENIX Conference on File and Storage Technologies (FAST 13)*, 2013, pp. 257–270. [Online]. Available: https://www.usenix.org/conference/fast13/technical-sessions/presentation/lu_youyou

[24] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 26–52, 1992.

[25] J. Menon, "A performance comparison of raid-5 and log-structured arrays," in *High Performance Distributed Computing, 1995., Proceedings of the Fourth IEEE International Symposium on*, 1995, pp. 167–178.

[26] "OpenStack Swift," http://swift.openstack.org/.

[27] B. Schroeder, R. Lagisetty, and A. Merchant, "Flash reliability in production: The expected and the unexpected," in *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, 2016, pp. 67–80. [Online]. Available: http://usenix.org/conference/fast16/technical-sessions/presentation/schroeder

[28] R. L. Villars and E. Burgener, "IDC: Building data centers for todays data driven economy: The role of flash," https://www.sandisk.com/business/datacenter/resources/white-papers/flash-in-the-data-center-idc, July 2014.

[29] S. Knipple, "Leveraging the latest flash in the data center," Flash Memory Summit, 2017, https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2017/20170809_FG21_Knipple.pdf.

[30] D. G. Andersen and S. Swanson, "Rethinking flash in the data center," *IEEE Micro*, vol. 30, no. 4, pp. 52–54, Jul. 2010. [Online]. Available: http://dx.doi.org/10.1109/MM.2010.71

[31] T. Feldman and G. Gibson, "Shingled magnetic recording areal density increase requires new data management," *USENIX; login: Magazine*, vol. 38, no. 3, 2013.

[32] *Archive HDD: v2 SATA Product Manual: ST8000AS0022, ST6000AS0022*, Seagate, Nov. 2015, revision F.

[33] *HGST Ultrastar Archive Ha10, Hard disk drive specifications*, HGST, Jun. 2015, revision 1.0.

[34] *Information technology – Zoned Block Commands (ZBC)*, INCITS T10 Technical Committee, Nov. 2014, working Draft, Revision 3. Available from http://www.t10.org/drafts.htm.

[35] L. Caulfield, M. Xing, Z. Tan, and R. Alexander, "Andromeda: Building the next-generation high-density storage interface for successful adoption," http://nvmw.eng.ucsd.edu/2017/assets/slides/51/, 2017.

[36] OCZ, "Saber 1000 HMS series: Performance test report using Aerospike db and the YCSB benchmark tool."

[37] "Non-Volatile Memory Express (NVMe) 1.3," http://nvmexpress.org/.

[38] L.-P. Chang, T.-W. Kuo, and S.-W. Lo, "Real-time garbage collection for flash-memory storage systems of real-time embedded systems," *ACM Trans. Embed. Comput. Syst.*, vol. 3, no. 4, pp. 837–863, Nov. 2004.

[39] A. Trivedi, N. Ioannou, B. Metzler, P. Stuedi, J. Pfefferle, I. Koltsidas, K. Kourtis, and T. R. Gross, "Flashnet: Flash/network stack co-design," in *Proceedings of the 10th ACM International Systems and Storage Conference*, ser. SYSTOR '17. New York, NY, USA: ACM, 2017, pp. 15:1–15:14. [Online]. Available: http://doi.acm.org/10.1145/3078468.3078477

[40] Q. Zheng, H. Chen, Y. Wang, J. Zhang, and J. Duan, "Cosbench: Cloud object storage benchmark," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '13, 2013, pp. 199–210. [Online]. Available: http://doi.acm.org/10.1145/2479871.2479900

[41] Seagate Corporation, *OneStor AP-2584 Datasheet*. [Online]. Available: http://www.seagate.com/www-content/product-content/xyratex-branded/embedded-storage-platforms/en-us/one-stor-ap2584-datasheet.pdf

[42] R. Mack, "Building timeline: Scaling up to hold your life story," https://code.facebook.com/posts/371094539682814/building-timeline-scaling-up-to-hold-your-life-story/.

[43] "Seagate: Solid state hybrid technology," http://www.seagate.com/solutions/solid-state-hybrid/products/.

[44] Western Digital Technologies, *WD Black2 Dual Drive User Manual*, Nov 2013.

[45] M. Saxena, M. M. Swift, and Y. Zhang, "FlashTier: a lightweight, consistent and durable storage cache," in *Proceedings of the 7th ACM European Conference on Computer Systems*, ser. EuroSys '12, 2012, pp. 267–280. [Online]. Available: http://doi.acm.org/10.1145/2168836.2168863

[46] X. Cheng, J. Liu, and C. Dale, "Understanding the characteristics of internet short video sharing: A youtube-based measurement study," *Trans. Multi.*, vol. 15, no. 5, pp. 1184–1194, Aug. 2013. [Online]. Available: http://dx.doi.org/10.1109/TMM.2013.2265531

[47] S. Godard, *iostat(1) Linux User's Manual*, July 2013.

[48] R. Stoica and A. Ailamaki, "Improving flash write performance by using update frequency," *Proc. VLDB Endow.*, vol. 6, no. 9, pp. 733–744, Jul. 2013. [Online]. Available: http://dx.doi.org/10.14778/2536360.2536372

[49] A. Aghayev and P. Desnoyers, "Skylight—a window on shingled disk operation," in *13th USENIX Conference on File and Storage Technologies (FAST 15)*, Feb. 2015, pp. 135–149. [Online]. Available: https://www.usenix.org/conference/fast15/technical-sessions/presentation/aghayev

[50] A. Aghayev, T. Ts'o, G. Gibson, and P. Desnoyers, "Evolving ext4 for shingled disks," in *15th USENIX Conference on File and Storage Technologies (FAST 17)*. Santa Clara, CA: USENIX Association, 2017, pp. 105–120. [Online]. Available: https://www.usenix.org/conference/fast17/technical-sessions/presentation/aghayev

[51] A. Kopytov, "SysBench: a system performance benchmark 0.5," https://code.launchpad.net/~sysbench-developers/sysbench/0.5.

[52] *md: Multiple Device driver aka Linux Software RAID.*

[53] "Swift architectural overview," http://docs.openstack.org/developer/swift/overview_architecture.html.

[54] "Swift software configuration procedures," http://docs.openstack.org/developer/swift/ops_runbook/procedures.html.

[55] C. Min, S. Kashyap, S. Maass, and T. Kim, "Understanding manycore scalability of file systems," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. Denver, CO: USENIX Association, 2016, pp. 71–85. [Online]. Available: https://www.usenix.org/conference/atc16/technical-sessions/presentation/min

[56] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," *;login*, vol. 41, no. 1, 2016.

[57] R. Konishi, Y. Amagai, K. Sato, H. Hifumi, S. Kihara, and S. Moriai, "The linux implementation of a log-structured file system," *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 3, pp. 102–107, Jul. 2006. [Online]. Available: http://doi.acm.org/10.1145/1151374.1151375

[58] J. Zhang, J. Shu, and Y. Lu, "Parafs: A log-structured file system to exploit the internal parallelism of flash devices," in *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '16, 2016, pp. 87–100. [Online]. Available: http://dl.acm.org/citation.cfm?id=3026959.3026968

[59] J. Colgrove, J. D. Davis, J. Hayes, E. L. Miller, C. Sandvig, R. Sears, A. Tamches, N. Vachharajani, and F. Wang, "Purity: Building fast, highly-available enterprise flash storage from commodity components," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1683–1694. [Online]. Available: http://doi.acm.org/10.1145/2723372.2742798

[60] W. Jeong, H. Cho, Y. Lee, J. Lee, S. Yoon, J. Hwang, and D. Lee, "Improving flash storage performance by caching address mapping table in host memory," in *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*. Santa Clara, CA: USENIX Association, 2017. [Online]. Available: https://www.usenix.org/conference/hotstorage17/program/presentation/jeong

[61] A. Batwara, "Leveraging flash translation layers for application acceleration," Flash Memory Summit, 2012, http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2012/20120821_TB11_Batwara.pdf.

[62] A. Zuck, O. Kishon, and S. Toledo, "LSDM: improving the performance of mobile storage with a log-structured address remapping device driver," in *Proceedings of the 2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies*, ser. NGMAST '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 221–228. [Online]. Available: http://dx.doi.org/10.1109/NGMAST.2014.9

[63] "dm-zoned: Zoned block device support," https://www.kernel.org/doc/Documentation/device-mapper/dm-zoned.txt.

[64] "OCZ announces first SATA host managed SSD: Saber 1000 HMS," http://www.anandtech.com/show/9720/ocz-announces-first-sata-host-managed-ssd-saber-1000-hms.

[65] X. Ouyang, D. Nellans, R. Wipfel, D. Flynn, and D. K. Panda, "Beyond block I/O: Rethinking traditional storage primitives," in *2011 IEEE 17th International Symposium on High Performance Computer Architecture*, Feb 2011, pp. 301–311.

[66] Y. Zhang, L. P. Arulraj, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "De-indirection for flash-based ssds with nameless writes," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, ser. FAST'12, 2012, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=2208461.2208462

[67] I. Koltsidas and S. D. Viglas, "Flashing up the storage layer," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 514–525, Aug. 2008. [Online]. Available: http://dx.doi.org/10.14778/1453856.1453913

[68] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending ssd lifetimes with disk-based write caches," in *8th USENIX Conference on File and Storage Technologies*, ser. FAST'10, 2010, pp. 8–8. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855511.1855519

[69] J. Y. Shin, M. Balakrishnan, T. Marian, and H. Weatherspoon, "Gecko: Contention-oblivious disk arrays for cloud storage," in *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*. San Jose, CA: USENIX, 2013, pp. 285–297. [Online]. Available: https://www.usenix.org/conference/fast13/technical-sessions/presentation/shin

[70] S. M. Rumble, A. Kejriwal, and J. Ousterhout, "Log-structured memory for DRAM-based storage," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*. Santa Clara, CA: USENIX, 2014, pp. 1–16. [Online]. Available: https://www.usenix.org/conference/fast14/technical-sessions/presentation/rumble

[71] "Btt - block translation table," https://www.kernel.org/doc/Documentation/nvdimm/btt.txt.

[72] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, 2011, pp. 91–104. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950379

[73] "Storage performance development kit," http://www.spdk.io/.