Beating the I/O bottleneck: A case for log-structured virtual disks

Anonymous Author(s)
Submission Id: 297

Abstract

With the increasing dominance of SSDs for local storage, today's network mounted virtual disks can no longer offer competitive performance. We propose a log-structured virtual disk (LSVD), using a locally attached SSD as a journaled cache, and persisting data in a log-structured format in generally available object storage. Our prototype demonstrates that the approach preserves all the advantages of virtual disks, while offering dramatic performance improvements over not only commonly used remote disks, but the same disks combined with inconsistent (i.e. unsafe) local caching

We describe and evaluate our prototype, discuss the lessons learned, and describe research opportunities opened up by this approach. We are working as part of an industry parnership to "upstream" a revised version of the prototype in an active open source community.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **Information systems → Distributed storage**.

1 Introduction

Network mounted virtual disks [10, 20, 28, 29] where the disk blocks are redundantly stored across physical disks attached to remote servers, have transformed the data center and are a fundamental underpinning of the cloud and many hyper-scale services. While supporting similar consistency to locally attached disks, by providing reliable storage that is de-coupled from computers, virtual disks enable resilience to compute and storage failures, flexible allocation of resources, virtual machine migration, and provide widely used volume management features such as snapshots and efficient cloning from pre-existing disk images.

In the past, virtual disks could also provide both price and performance advantages over local hard drives (HDDs), using statistical multiplexing to achieve higher peak performance with fewer total spindles. Today it is difficult for virtual disks to compete with local storage, with typical local SSDs achieving speeds that would saturate most network connections. One can employ SSDs for a client-side cache on top of a virtual disk [3, 15, 24]; however, while a large cache can eliminate all reads, scale-out storage backends perform poorly for the remaining typically random write-back traffic of (small) disk blocks(§2).

In the late 80s the LFS team [23, 26] hypothesized that write I/O would become a key computing bottleneck. They considered an approach using a durable cache in battery-backed RAM, but rejected it in favor of a log-structured file system [26] based on mass-market hardware of the day, i.e. disk and volatile RAM. For years later, durable caches remained niche solutions in proprietary NAS file servers [14] and RAID hardware [18].

There have been massive changes in the hardware landscape since then. SSDs are commonplace, large enough to hold entire working sets, and are naturally durable without specialized hardware. Object stores such as S3, provide very high throughput remote storage, and their reliability and atomicity greatly simplifies implementation of a reliable log structured system.

We propose the idea of a *log-structured virtual disk* (LSVD), building a durable cache and log-structured backend on modern technology to solve today's I/O bottleneck for virtual disks on scale-out infrastructure. We employ SSDs as a local journaled cache, and use a log-structured approach to convert writes into operations that can be efficiently supported by today's throughput optimized object storage.

We have implemented a simple prototype to explore these ideas, comprised of a Linux kernel module implementing the journaled cache and a user-level service to log data to an object store and perform garbage collection. We compare performance of our prototype against a popular open source virtual disk implementation (Ceph RBD) and show that an LSVD approach can:

1. **Achieve dramatically higher performance.** Converting block writes into a log enables them to be efficiently combined into large writes to throughput-optimized object stores; as a result, LSVD is able to sustain a very high rate of writes to the local SSD cache with dramatically reduced load on the backend storage devices. For example, (§4.5) 16 LSVD volumes with a total of 50K IOPS client traffic resulted in only 10% utilization of the backend drives, while RBD in the same configuration achieved 13K client IOPS with 70% backend utilization. In addition garbage collection overhead—of concern in any log-structured system—is low even for extreme random-write workloads.
2. **Preserve key properties of today's virtual disks.** LSVD recovers all committed data in the case of a client crash and recovery, and supports the same prefix consistency guarantee as other virtual disk designs [19, 36] in the case of a catastrophic failure. It makes efficient use of erasure-coded backends, with potentially higher levels of resilience to failure than replicated storage. The efficiency of writes enables

the cache to be cleaned in seconds, enabling flexible allocation of resources and virtual machine migration. Finally, the LSVD log structure and garbage collection algorithm naturally supports snapshots and cloned volumes.

3. **Naturally enables functionality that is challenging with today's virtual disk implementations.** By using immutable garbage-collected objects for the log we are able to asynchronously replicate a volume over geographic distances. In addition since the entire implementation is done in the client, our virtual disk can be employed in existing clouds with no support from the cloud vendor. For example, LSVD on AWS (§4.8) is able to achieve an IOPS rate for a few dollars a month that would have cost over $3000/mo using AWS EBS.

We compare performance of the journaled against bcache, a popular open source client side cache that is often layered on top of RBD for improved performance. To preserve the virtual drive advantages of migration and durability, LSVD aggressively writes back data (to S3) and offers prefix consistency in the case of a full client failure. In contrast, bcache prioritizes local access over write-back unless the cache is full, and offers no consistency guarantee if the cache is lost.

When we minimize backend traffic, we found LSVD and bcache have generally similar performance advantages over base RBD (16x §4.2), while LSVD performs better for sync intensive workloads (achieving 4x better performance for a data base benchmark §4.2.2). While bcache needs to write its map on each sync operation, LSVD's journaled cache allows map persistence to be moved off the critical path.

For sustained load, where both LSVD and bcache write back data aggressively, we find that LSVD has a dramatic performance advantage (e.g. 8x §4.3). Although bcache can handle bursts of writes at high speed, it can spend 10s of minutes writing data back, during which time a failure of the client can result in file system corruption (§4.4). In contrast, LSVD needs just 10s of seconds to clean the cache, and even a catastrophic failure during that period will still leave a prefix consistent image on backend storage.

The main contributions of this paper are:

- We show that close coupling of a durable local cache and an out-of-place log-structured design offers enormous advantages with today's technologies. While there has been much research on both client side caching [3, 15, 24, 32] and novel virtual disk implementations based on out-of-place writes [17, 19, 36], to the best of our knowledge no work has closely coupled the two.
- We develop and evaluate a prototype that demonstrates the practicality of the idea and demonstrates that the approach can: 1) achieve enormous performance improvements, 2) retain the advantages of today's virtual disks, and 3) offer advantages in mobility and deployability over many previous approaches.
- Given the novelty of an LSVD approach, there are numerous alternatives on how one should, for example: 1) maintain a journaled cache, 2) support snapshots and clones, 3) perform garbage collection, 4) split implementation between kernel and user level, etc. We describe a viable set of design decisions we used in our prototype, then based on this experience, discuss opportunities for improvement and research enabled by an LSVD approach.

In the remainder of the paper we discuss motivation and survey related work (§2), describe the design and implementation of our prototype (§3), evaluate our prototype (§4), discuss lessons learned from the prototype (§5) and conclude.

2 Background and Motivation

A large portion of the computation performed today runs on virtual machines, and almost all of these virtual machines run on remote virtual disks on scale-out storage. In the cloud, users may interact with these disks directly—e.g. AWS EBS [28], Google Persistent Disk [10], Azure Managed Disks [20]—or indirectly via VM-isolated containers or functions; in open source-based clouds Ceph RADOS Block Device (RBD) [29] is common. Non-cloud hyperscale deployments deploy VMs for flexibility and efficiency, with remote disks allowing not only resilience to failure, but flexible deployment of resources.

Any possible improvements in virtual disk efficiency for these untold millions of virtual disks would result in significant savings. As we will see, such improvements are not only possible, but are potentially quite large.

2.1 Virtual Disk Overhead

Consider a large cluster with storage on IOPS-limited devices—today this includes not only HDDs but higher-capacity flash-based SSDs [33]. Total storage performance (as seen by clients) is roughly determined by (a) how efficiently all devices can be utilized, and (b) the degree of I/O amplification between client and devices.

Remote shared storage has performance advantages and disadvantages vs. local storage. Shared devices can be utilized more effiently, making use of IOPS that would be lost on idle local drives with bursty workloads. Yet remote storage also pays a significant cost in I/O amplification. Reads are not the problem—with cached metadata, most scale-out storage systems can translate a small read operation (typical of client block I/O) to a single operation on the underlying device. Writes, however, are a different issue.

There are two main sources of write amplification. Expansion due to redundancy is unavoidable, ranging from roughly 1.5x for erasure coding to 3x for triple replication. Consistency adds additional costs—writes must either succeed or fail, in the latter case leaving the target blocks unmodified. These issues are especially pronounced when erasure coding is used, as small modifications may require re-writing all parity blocks for the stripe, and care must be taken to ensure parity and

data are always consistent[1]. These overheads are not small, e.g. one study of the RBD virtual disk [16] has shown a write amplification of 13x.

This overhead is especially problematic on HDD-based backends. Although providing far fewer operations per second than even high-capacity SSD, HDDs are often favored for their much lower cost per gigabyte; thus e.g. at the time of submission, the default for Google Persistent Disk is HDD-based.

2.2 Caching and Consistency

Various prior work has sought to accelerate remote virtual disks via local caching, typically in SSD [3, 15, 24, 32][2]. These efforts show great success in speeding up read operations; however although reads are more numerous than writes in most workloads, the relative efficiency of backend reads limits its potential gains. Caches to date which provide the barrier semantics needed to prevent file system corruption on cache failure (e.g. Mercury [3], Write-Back Persist [24]) do little to reduce the overall number of backend writes, at most smoothing out short bursts and coalescing small numbers of writes to the same blocks.

Even "unsafe" write-back caches — ones such as Linux bcache which order write-backs for efficiency rather than consistency — are limited in the degree which they can reduce write overhead. In our evaluation (§4.3, Figure 7) we see an example of this: using bcache over a remote RBD volume, a 2-minute random write burst completes at local SSD speed, however despite having full use of a 10-machine 70-device backend cluster with no competing usage, the write-back "backlog" takes 20 minutes to complete. Sustained performance would be limited to $1/10^{th}$ that of local, while monopolizing a backend shared by many clients.

Block devices may provide varying levels of consistency, which is described in terms of *write acknowledgements* and *commit barriers*. Ordering in block devices is not guaranteed for outstanding writes, however under normal conditions, a read or write operation must "see" any writes which were acknowledged before the I/O is submitted. Under failure conditions, however, acknowledged writes may be lost; typical storage devices (SSDs, HDDs) only guarantee durability of a write after a commit barrier instruction (e.g. SATA FLUSH CACHE) is completed.

Most virtual disks [19, 37] and write-back caches [15, 24] provide a weaker guarantee, *prefix consistency* [9]. Acknowledged writes may be lost, but visible writes will form a consistent prefix; i.e. if any write after a commit barrier is visible, all writes before that barrier will be as well. Prefix consistency preserves the crash consistency guarantees of the file system, preventing file system corruption on failure. Such failures will in effect "roll back the clock" on the storage device by a short length of time, much like failures in early journaled file systems with batch commit [11]. This poses few problems for many virtual machine uses, such as "stateless servers" where critical state is persisted in a remote database.

Many demanding uses are tolerant of prefix consistency as well, due to mechanisms used to recover from temporary failure; consider Dynamo [5] as an example. Updates at each node are versioned, and queries requiring consistency guarantees are sent to more than one replica; if a node missed an update while it was down, its reply will be filtered in favor of a more up-to-date reply, and the node will eventually recover the most recent value via the anti-entropy protocol. Prefix consistency will in effect result in the need to recover from a perceived outage lasting a fraction of a second longer than the real failure, with negligible effect[3].

2.3 Remote Disks

The virtual disk systems described in the literature which most resemble LSVD are Salus [36], Blizzard [19]), and Ursa [17]. Unlike e.g. Ceph RBD [29], which translates individual disk writes into writes to an underlying rewritable storage system, each of these systems is based on out-of-place writes in a log-structured format, using sequencing mechanisms to provide prefix consistency under worst-case failure scenarios.

Salus [36] is an uncached remote disk modeled after HBase [1] but optimized for block storage. Client writes are sent to the current leader, where they are logged to an HDFS write-ahead log and accumulated in memory into large batches, which are then written to large *checkpoint* files. A distributed, consistent map is kept from LBAs to locations in the HDFS checkpoints, and a garbage collection process periodically compacts these checkpoints to minimize storage overhead. On leader failure or reconfiguration, client-provided sequence numbers are used to preserve prefix consistency.

Ursa [17] uses similar replicated and sequenced journaling mechanisms, but is (a) adapted for a hybrid environment with SSD primary replicas log-shipping to HDD secondaries, and (b) taking advantage of its out-of-place write mechanism to provide durability if even only one replica has been written before a failure occurs.

Blizzard [19] is an uncached client-side disk over Flat Datacenter Storage [21], like LSVD requiring only modest consistency guarantees from the backend service to provide prefix consistency in its *fast ack* mode. Although Blizzard is log-structured, it cannot perform batching, resulting in a large block size (64 or 128 KB) and significant read/modify/write overhead for emulation of smaller writes.

LSVD fundamentally differs from these previous systems in the coordinated combination of a journaled write-back cache

[1] That is, the "RAID small write and "RAID write hole" problems.

[2] DRAM-based caching is of limited value for virtual disks, as such a cache is in effect an level 2 cache with the guest OS buffer cache acting as the first level [38]; few hits will be seen until the buffer cache has been duplicated in the disk cache.

[3] We note that Raft's AppendEntries mechanism [22] will tolerate prefix-consistent failures in a similar fashion, and expect many other systems to do so as well.

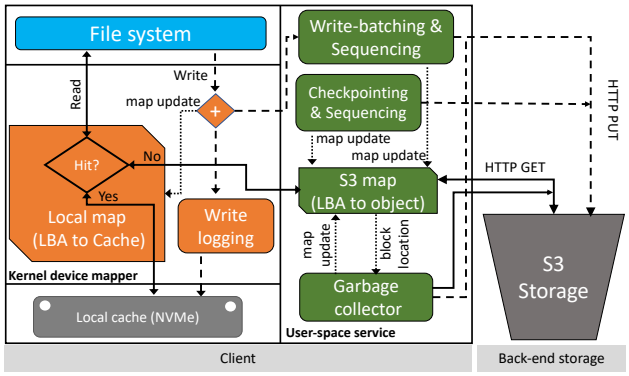

Figure 1. LSVD architecture.

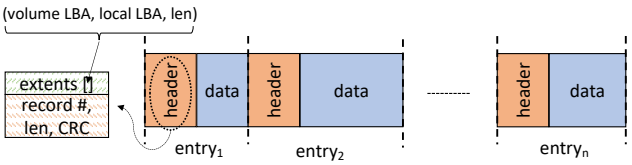

Figure 2. LSVD Local cache journal format.

with the out-of place writes, enabling much larger batching of writes than any of these systems. As a result, LSVD can layer on top of existing generally available object storage services, avoiding the fixed block size issues of Blizzard.

We note that there are a number of proprietary systems which may be related to LSVD, including cloud provider-specific virtual disks (Amazon EBS [28], Google Persistent Disk [8, 13], and Azure Managed Disk [27]); we are unable to contrast their approach to that of LSVD as this information is not available to the authors or readers. The only proprietary third-party system known to the authors is the S3-based CloudBD [4] block device, however experiments indicate that it is not log-structured.

3 Prototype

To evaluate LSVD we constructed a Linux-based prototype, described below.

3.1 Implementation

The prototype is implemented in two components, which may be seen in Figure 1: a kernel-mode *device mapper* [7] and a user-space service, with the two communicating via an `ioctl` interface. The device mapper is implemented in ~1000 lines of C, while the user-space service is ~1500 lines of Go[4].

[4]Source code is available under an open source license at [redacted]; in addition we are working with the open source community to contribute a modified version of this code to the Ceph project.

3.2 Operation

At a very high level the strategy used for the local cache and remote data is similar: (a) write data with a prepended header holding mapping information (i.e. LBAs), (b) lazily persist a full translation map, used as a base for recovery, and (c) on restart, load the most recent checkpointed map and update it with mappings from any journal headers written after the checkpoint. We explain this in more detail below.

Writes are tracked using in-memory maps; the kernel map translates volume LBAs to physical LBAs in local storage, while the user-space map translates volume LBAs to locations in remote objects, identified by object sequence number (from which an object name may be derived) and offset within the object. Write operations are forwarded to the user-space service, as well as being logged to local storage with a prepended journal header, shown in Figure 2; the in-kernel map is updated to record the location of the newly-written data.

Writes are acknowledged after replies are received from the journal device, and commit barriers are forwarded on the same path, ensuring that all proceeding writes are durable after completion of a commit barrier. In addition to mapping information (i.e. LBAs) the journal header contains a sequence number and CRC over the header and data, allowing incomplete writes to be detected during journal recovery. The journal header is allocated a single 4 KB block; although this increases the volume of data written for small writes, the number of operations remains the same and the efffect on throughput is negligible. In our evaluation (§4.2.2) we compare with bcache [32], which caches an on-SSD map in memory and flushes it on sync operations; for sync-heavy workloads our journaled cache is significantly faster.

The map from volume LBA to physical journal location is kept in memory; the user-space service periodically queries the kernel driver to retrieve this map, along with the current journal head position, and writes it to an on-SSD checkpoint. The performance impact of checkpointing the map is low, as it is performed relatively infrequently and in parallel with user I/O. On restart the service retrieves this checkpoint, "rolls" the journal forward to the end to retrieve the most recent map of cache contents, and initializes the kernel cache map.

The user-space service receives writes from the kernel and buffers them in memory until a configured size is reached (32 MB in our experiments) or a timeout is triggered. Each batch is assigned a sequence number and written to object storage, with a name composed of a prefix (indicating the disk image) and the sequence number as a numeric suffix. (see Figure 1) As with local writes to the journaled cache, a prepended header records the volume LBAs of the data blocks, as well as additional information described below.

The user-space service keeps a map from LBA to locations in the object store, i.e. objects and offsets within them, updated

with each write. Again we persist the map periodically, amortizing overhead while bounding the amount of log recovery needed at restart.

Map checkpoints are stored in sequenced objects in the object stream, precisely ordering each snapshot with respect to preceding and following writes. The most recent map checkpoint may be found by locating the log head (e.g. via the S3 ListObjects operation) and searching backwards; the map may then be loaded and the log rolled forward to incorporate updates after it was persisted[5].

Reads are checked against the kernel map, and forwarded directly to the journaled cache device if present locally. Read misses are forwarded to the user-space service, which consults its map to determine the corresponding location (i.e. object number and offset) in the object stream, and issues a range read request. (see Figure 1) When this completes the result is forwarded to the kernel module, which completes the read operation, as well as writing the data to SSD and entering it (as clean rather than dirty data) in the kernel map.

3.3 Recovery and Consistency

Recovery from a temporary client failure or crash is done by reading the most recent map checkpoint and rolling the log forward, akin to the checkpoint / roll-forward procedures in the original log-structured file system [26]. Since commit barriers are forwarded to the journaled cache device itself, any committed writes will have been made durable along with their journal headers, allowing full recovery.

Permanent failures of the cache device or client require recovery from the remote object stream. LSVD writes objects concurrently, not waiting until one completes before issuing another write; this may result in inconsistent data in the log after a crash. For example, creation of an object holding writes following a particular commit barrier may have succeeded, while a prior object holding writes from before that barrier may be missing.

This is solved by taking the longest consecutive prefix of sequence numbers[6]: if object n is missing, then objects $n+1$,... are ignored. Since writes are batched into objects in the order they are received, the result is a set of visible writes which is consistent with respect to any commit barrier separating writes within that set, guaranteeing prefix consistency.

3.4 Garbage Collection

When a virtual disk location is overwritten, any data from previous writes to that location becomes out-dated or *stale*. This space cannot be re-used, as LSVD is designed to be layered on S3, where objects are immutable; however such stale data is typically intermingled with "live" data, preventing deletion of the object. If not addressed, this would result in unbounded growth of the storage consumed by the object stream.

A greedy garbage collection algorithm chooses the objects with the smallest valid data ratios, copies the live sectors from these objects to new ones, and deletes the now-unneeded objects. After sufficient wasted space has been reclaimed, the garbage collection process stops and a checkpoint is written. The garbage collection algorithm tracks the size and utilization of each "live" (i.e not yet deleted) object in the object stream, updating this information with each write.

Although the table allows identification of candidate objects for cleaning, it does not identify the location of any live data to be copied. Since any remaining live data is a subset of the data originally stored in the object, we retrieve the object header and look up each extent in the current map, locating ranges which still map to the to-be-deleted object.

We note that garbage collection introduces gaps in the object sequence, complicating recovery. This is addressed by deferring deletion until a batch of garbage collection writes have all completed, along with a map checkpoint; recovery will proceed forward from that checkpoint and ignore sequence number gaps preceeding it.

Garbage collection is performed in batches, writing copied data in order of logical address. A small amount of defragmentation is performed during this process, copying additional data to plug small "holes" in the garbage-collected data; this improves large read performance as well as reducing memory usage of the maps. Copied data is written in separate batches from incoming data, creating a "generational" garbage collector which implicitly separates hot and cold data, reducing write amplification [6].

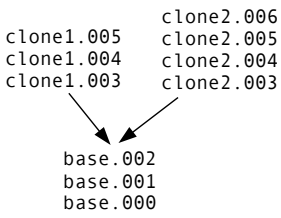

Figure 3. Object streams for two clone volumes from a single base image.

3.5 Volume Clones and Snapshots

LSVD allows multiple virtual disks to be "cloned" from a single base image, which is not modified. Conceptually this is quite simple, as illustrated in Figure 3—each clone branches its object stream off from the object stream of the precursor volume. Implementation requires some additional rules for mapping sequence numbers to object names; in addition the

[5]This may be further optimized by writing a root object under a well-known name (e.g. omitting the sequence number) with a pointer to the most recent map.

[6]Actually, the longest consecutive sequence of sequence numbers higher than the most recent checkpoint. The garbage collector deletes objects, leading to "holes" in the object sequence, but it persists a copy of the map before performing any deletes.

client	backend	
	✓	Ceph Octopus 15.2.3
✓	✓	Ubuntu 20.04, kernel 5.4/5.0
✓	✓	40 cores (2x E5-2660)
✓	✓	128 GB RAM
✓	✓	10Gbit ethernet
	✓	8x SATA SSD (mixed 256, 512GB)
✓		800 GB Intel DC P3700 NVMe

Table 1. Experimental setup.

base image stream must be protected from garbage collection by any of its clones.

A volume snapshot is simple, as well—any object i in the object stream can serve as a snapshot. After loading the newest snapshot previous to i, and then rolling forward to i, LSVD will present a point-in-time view of the volume at the precise point when object i was written. That is, it will do so until the garbage collector deletes some essential parts of it—the fundamental LSVD snapshot algorithm includes a modified deletion mechanism to preserve objects "pinned" by snapshots, and delete them when the snapshots are moved.

The key idea is that a checkpoint at sequence k depends on all objects, and only those objects, which were "live" (i.e. had been created and not yet deleted) at sequence k. Thus if an object is created at sequence i, cleaned at sequence j, and there are no checkpoints in the sequence number range $i \ldots j$, then it may be safely deleted: any checkpoint before i cannot depend on it, nor can any created after j as it no longer contains current data. However if such a checkpoint k exists, $i < k < j$, then the object is added to a list of deferred deletions. When a checkpoint is deleted the deferred delete list is scanned, and any objects freed up are deleted.

Both volume clones and snapshots require additional bookkeeping information to be persisted in checkpoints. For clones this includes the prefix and sequence numbers of all base volumes; for snapshots it includes lists of both current snapshots and deferred deletions.

4 Evaluation

After (§4.1) describing our experimental setup, we compare LSVD to an optimized SSD cache with minimal backend traffic (§4.2), and examine sustainable performance when data is continuously written to the backend (§4.3). Section 4.4 evaluates the time to clean the cache, a critical property for VM migration. Subsequent sections (§4.5 and §4.6) conclude our evaluation by examining load on the shared backend storage and the impact of garbage collection. Finally, we explore new functionality (§4.7 & §4.8) which is challenging in previous virtual disks: geographical replication and deployment on existing clouds without provider support.

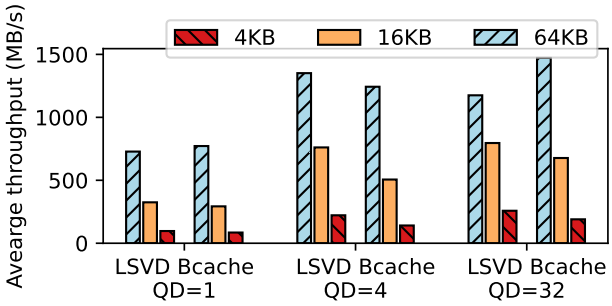

Figure 4. Write performance: LSVD vs RBD+bcache for varying queue depth (QD, i.e. concurrent operations) and I/O sizes. LSVD is equivalent or faster for all but 64 KB QD=32.

4.1 Experimental Setup

The primary test environment (Table 1) is a Ceph cluster with 4 storage (OSD) servers with a total of 32 capacity SSDs, one S3 server (Rados Gateway, RGW), and a single client. On the client we compare LSVD to Rados Block Device (RBD) [29], and to RBD with bcache [32]. RBD is a virtual disk implementation using consistent hashing to stripe and replicate the disk image across an array of storage servers (OSDs), while bcache is a linux kernel module often used with RBD, which implements a cache on top of another block device. Additional tests were performed on an HDD-based Ceph cluster with 9 OSD servers and 62 10K RPM HDDs. RGW was configured to use a 4,2 erasure-coded pool, while the RBD pool was triple-replicated.

Both LSVD and bcache were configured with 700 GB of NVMe storage; bcache was set to write-back mode with default parameters except for the write-back speed throttle, which was disabled. The LSVD garbage collection threshold was set to 70%, with cleaning triggered when average utilization dropped below that point.

4.2 Cache Performance

We compare LSVD's journaled cache against bcache using block (§4.2.1) and file system (§4.2.2) micro-benchmarks while selecting cache sizes and parameters to ensure that all operations are satisfied in local cache[7]. Although the LSVD cache has not been optimized for performance, we find that it is competitive and can even have significant advantages for sync-intensive workloads with many commit barriers. The advantage for LSVD stems from its journaled cache, allowing lazy persistence of the full cache map. In contrast, bcache must aggressively write its map, as writes will be lost after a crash if their corresponding map entries have not been persisted.

4.2.1 Block benchmarks: We use `fio` [2] with the `libaio` backend and direct I/O to the raw block device to exercise LSVD and bcache. Figures 4 and 5 show the performance of

[7] In particular, pre-loading the cache to so all reads hit, providing bcache with a large enough cache that it will not write back any data during tests, and configuring LSVD with a large write-back window to produce similar behavior.

	file count	mean file size	IO size	thread count	mean append size	log file size
fileserver	200,000	128 KiB	n/a	50	16 KiB	n/a
oltp	250	100 MiB	2000	50	n/a	100 MiB
varmail	900,000	32 KiB	n/a	16	16 KiB	n/a

Table 2. Filebench workload parameters.

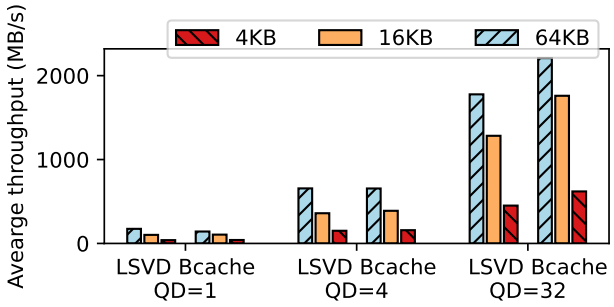

Figure 5. LSVD vs. RBD+bcache: random read, 100% cache hits: LSVD is equivalent at low queue depths, but somewhat slower at QD=32.

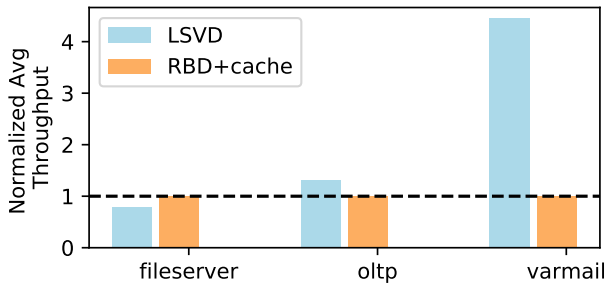

Figure 6. Filebench: LSVD vs RBD+bcache throughput normalized to bcache.

	between syncs		mean
workload	writes	bytes written	write size*
fileserver	12865	579 MiB	94 KiB
oltp	42.7	199 KiB	4.7 KiB
varmail	7.6	131 KiB	27 KiB

*after merging consecutive sequential writes

Table 3. Filebench workloads: write operations, number of sectors between commit barriers; mean merged write size.

random read and write requests with different concurrency (queue depth, QD) and block sizes from 4 KB to 64 KB. LSVD offers very competitive performance for operations that are satisfied in the cache. Reads are equivalent to bcache at low queue depths, but slower with 32 outstanding operations; writes are similar or faster for all but the 64 KB QD=32 case.

As expected both caches achieve dramatically higher performance than uncached RBD, which is omitted from the graphs as it would not be visible. In the best case (64 KB, QD=32) RBD write reached roughly 100MB/sec, while LSVD and bcache speeds were more than 16 times faster.

4.2.2 File system benchmarks. Figure 6 uses workloads from *Filebench* [31] to explore the impact of more complex file system benchmarks on the performance of LSVD versus bcache. These benchmarks include rich mixtures of file system operations that result in meta-data operations and commit barriers to the disk. Specific workloads we emulate are a network file server (*fileserver*), a database (*oltp*) and a mailserver (*varmail*) with parameters seen in Table 2. A fresh `ext4` file system was created for each run, with default `mkfs` and mount options.

As in the case of the block cache we were surprised that LSVD outperformed bcache for two of the three workloads; achieving more than 4x the throughput in the case of *varmail*. Collecting information at the SCSI level (Table 3) we find that *varmail* has the least number of bytes written between sync operations. In retrospect it is clear that bcache has to persist its translation map for each sync operation, and sync operations hence exacerbate the overhead we saw for block operations (§4.2) of persisting that map. LSVD journaling increases the volume of data written to SSD, but does not add any additional operations as a result of syncs.

4.3 Sustained Performance

While it is intriguing that the LSVD design can have advantages when all operations can be satisfied in the cache, a major goal of LSVD is to write back data aggressively in order to enable the advantages of existing network mounted virtual disks (e.g., VM migration). Figures 8 and 9 show the performance for sequential and random writes when we restrict the size of the cache to 5 GB, forcing both LSVD and bcache to aggressively write back data to the backend.

LSVDs performance does degrade substantially with a small cache; for example, throughput dropped from 1500MB/s to 600MB/s for large random writes with high concurrency. This degradation occurs in part as a result of the limited backend bandwidth; if data cannot be written back fast enough, eventually all blocks become dirty and subsequent writes to the cache are blocked.

In all cases LSVD performance was higher than bcache, by factors ranging from 2x to 8x. While with bcache sequential writes have better performance than random ones, it is still not able to achieve close to the same performance as LSVD that aggregates all writes into large operations that are erasure coded across many disks. LSVD is, in contrast, largely insensitive to the access pattern.

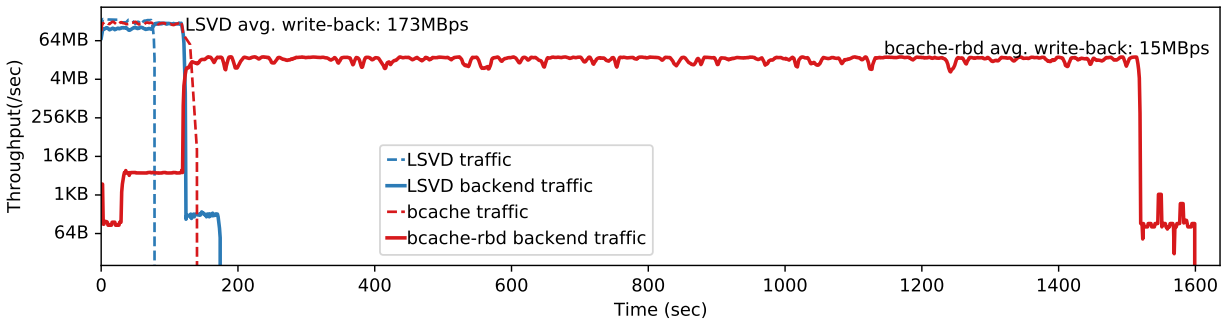

Figure 7. 20 GB of 4 KB random writes and recovery: LSVD and rbd+bcache. Solid lines are virtual disk throughput, dashed lines are backend traffic. LSVD writes complete at 75s; write-back is complete at 125s. Bcache+RBD writes complete at ~125s, while write-back completes at 1510s.

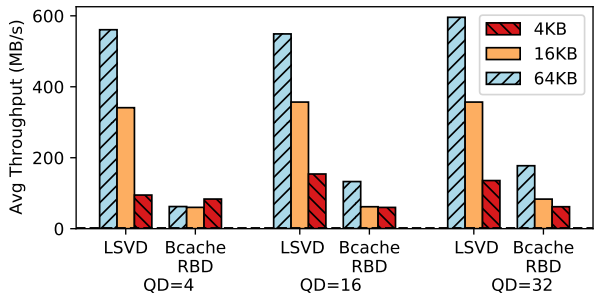

Figure 8. Sequential block write performance, 5 GB cache.

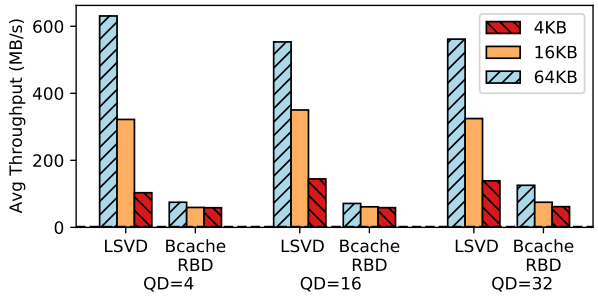

Figure 9. Random block write performance, 5 GB cache.

4.4 Write-back and Failures

To understand how long it takes for the state in a cache to be fully written to the backend (e.g. enabling a VM to be migrated) we wrote 20 GB of random 4 KB writes to both bcache and LSVD, and tracked virtual disk throughput and back end traffic until the backlog cleared; results are shown in Figure 7. The HDD-based backend was used, with 62 10K RPM drives across 9 machines. Bcache incorporates a write-back throttling mechanism; for these tests it was effectively disabled by setting the throttle limit to 5 TB/s.

Virtual disk throughput is high for both RBD+bcache and LSVD, dropping to zero when the workload completes at about 80s (LSVD) and 125s (RBD+bcache); the difference in throughput is not visible on the log scale. LSVD writes data to the backend at high speed during the random write phase, then continues at the same rate for a few tens of seconds before finishing at 125s. In contrast, bcache pauses write-back under high load, then spends over 20 minutes performing random writes to the backend, as few of the random writes were able to be merged by the cache into larger.

		Mounted after crash?	Required FSCK
Bcache	1	Yes	No
	2	No	Yes
	3	Yes	No
DIS	1	Yes	No
	2	Yes	No
	3	Yes	No

Table 4. Crash tests of LSVD and RBD+bcache: recursive copy of 74K-file directory interrupted by VM reset. In bcache test 2 the file system was recoverable via fsck, but all copied files were lost.

The experiment points out several significant costs to the use of unsafe write-back caches in a shared virtualized environment. The single virtual disk used in the experiment was given exclusive access to the back-end cluster, with no competing use; write-back would take longer in the presence of other users, who would experience significant performance degradation during that time. In addition we lose several of the advantages of a remote virtual disk: durability, as failure during this window would result in massive file system corruption, and mobility, as any attempt to migrate the virtual machine would need to wait for 20+ minutes for pending write-backs to be flushed.

The LSVD consistency guarantee is a key advantage over inconsistent write-back cache solutions such as bcache. To evaluate this we perform a recursive copy of a directory tree of 74,000 files to a fresh ext4 file system, performing a virtual machine reset just before or after completion of the copy, and then

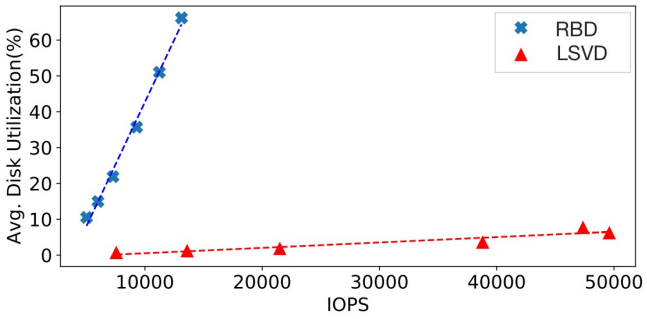

Figure 10. Write efficiency: IOPS vs backend disk busy time, LSVD vs RBD. Total virtual disk IOPS are plotted against mean backend device utilization for tests with 1, 2, 4, 8, 16 and 32 virtual disks and random 16 KB writes.

simulating client failure by deleting the cache. Results may be seen in Table 4. In all cases the LSVD disk image mounted without errors; in contrast the RBD+`bcache` image was unmountable in one case, with no files recovered after `fsck`.

4.5 Back-end Load

How well does LSVD succeed in its goal of increasing the efficiency of writes to the back-end? To measure this we run a series of tests with different numbers of virtual disks, measuring back-end load; for a workload we use random write(fio 16 KB writes with queue depth 32) as it presents a worst case for garbage collection. Tests were run with the HDD-based backend, with 9 OSD servers and a total of 62 10K RPM HDDs, and measure mean disk utilization (i.e. fraction of time busy, from `/proc/diskstats`).

Results are seen in Figure 10. Load is measured as disk busy time, in percent, as reported by `/proc/diskstats`, averaged across all disks in the storage backend. With RBD the backend load grows quickly, reaching 70% with 32 virtual disks and ~13,000 IOPS. In contrast, with LSVD the backend utilization never reaches 10%, while aggregate client IOPS reach roughly 50,000 in experiments with either 16 or 32 virtual disks, approaching the speed of the 10 Gbit/s client network link.

In this experiment, LSVD provides a 25x advantage in efficiency over RBD for small writes, a particularly difficult workload for RBD and similar systems[8].

4.6 Garbage Collection

Garbage collection is a key factor in the performance of any translation layer. We evaluate its performance in both simulation and file system-based experiments.

Simulation: Garbage collection behavior is highly dependent on workload characteristics, and differing (real-world) workloads may show very different performance. To explore

[8]And a not-uncommon one: this work was originally motivated by poor performance seen running Kubernetes on RBD-based VMs, due to write traffic from `etcd`.

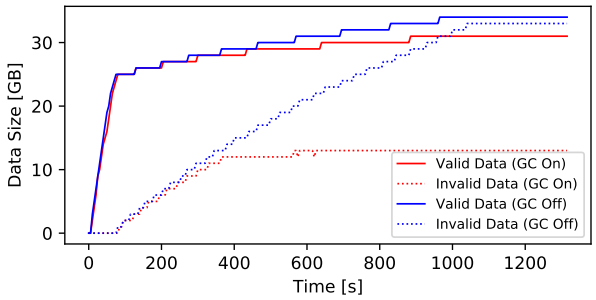

Figure 11. Garbage collection for varmail workload: GC on (red) vs off (blue); solid line indicates valid data, dashed invalid.

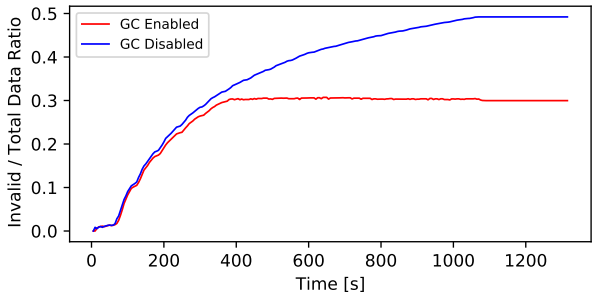

Figure 12. Garbage collection for Filebench *varmail* workload: invalid data as fraction of total.

this we simulate the LSVD write batching and garbage collection algorithms on traces from the CloudPhysics corpus [35]. This is a set of week-long block traces from 106 Linux and Windows virtual machines, representing a wide range of workloads. All simulations used a 32 MB write batch size, with utilization thresholds for starting and stopping garbage collection set to 70% and 75% respectively.

We report the following measures: *Write amplification*, or the ratio of total back-end writes to client writes. *Merge ratio*: the fraction of write data eliminated when writes are coalesced within batches[9]. *Extent count*: LSVD is an extent-based system (like e.g. NTFS or ext4), and the extent count determines map memory usage and measures back-end storage fragmentation; map size is reported at the end of the simulation.

Table 5 shows results for a selection traces, including those with highest and lowest results for both write amplification and extent map size. Write amplification is modest, in almost all cases well under 1.5x; the two exceptions are some of the lowest-speed traces, with 60 and 85 GB written over the period of a week. Write coalescing is highly effective for a limited set of workloads—e.g. in w66 and w41 a majority of bytes are overwritten while batching, and never sent to the backend. This results in significant improvements in write amplification, e.g. from 1.97 to 1.35 for w66 and 1.44 to 1.14 for w41. Extent map size is quite modest for most workloads, and improves with write coalescing, but remains high for w01. We evaluate a modified algorithm which performs extra reads to plug "holes"

[9]Cross-batch write coalescing would break the LSVD consistency guarantees.

	writes	extent count (M)			WAF			merge
	GB	no merge	merge	defrag	no merge	merge	defrag	ratio
w10	484	3.88	3.51	3.51	1.11	1.1	1.10	0.01
w04	1786	1.93	1.91	1.91	1.52	1.44	1.44	0.21
w66	49	0.02	0.02	0.02	1.97	1.35	1.36	0.55
w01	272	5.67	5.47	2.78	1.2	1.18	1.20	0.11
w07	85	0.7	0.69	0.55	1.82	1.76	1.83	0.06
w31	321	0.9	0.61	0.61	1.03	1.02	1.02	0.02
w59	60	0.26	0.26	0.26	1.75	1.65	1.64	0.14
w41	127	0.59	0.58	0.05	1.44	1.14	1.14	0.71
w05	389	6.8	3.06	3.06	1.08	1.08	1.08	0.0

Table 5. Simulated LSVD garbage collection on representative traces, showing volume of data written, final extent map size, write amplification factor (WAF) and write coalescing (i.e. merging) performance.

in copied data of 8 KB or less; this reduces w01 map size by over a factor of 2 at a negligible cost in write amplification.

Physical experiments: Here we (a) examine the effectiveness of the garbage collector in eliminating stale data, and (b) measure its impact on performance. We use the *varmail* workload to evaluate cleaning effectiveness, as after populating its test directory it repeatedly re-writes the same blocks by creating and deleting small files, generating large amounts of stale data.

In Figure 11 we see the volume of live and stale data graphed over time for two benchmark runs, with garbage collection enabled in one and disabled in the other. With garbage collection disabled, the volume of invalid data grows nearly linearly, leveling off only after the workload completes at 1000s. With it enabled, cleaning begins when valid data drops to 70%; as seen in Figure 12 which plots overall utilization, this ratio is maintained for the rest of the experiment.

The performance impact of garbage collection may also be seen: after 1000 seconds, slightly more valid data has been written in the GC-disabled run (blue) than in the GC-enabled one (red). Additional experiments show a slowdown of roughly 8% for *fileserver*, 5% for *varmail* and 2% for *oltp*.

4.7 Replication

LSVD volumes may be asynchronously replicated by the simple mechanism of lazily copying the object stream, applying the same recovery rules when mounting the replicated volume[10]. We evaluate this on a pair of object stores: directly writing to RGW running over a 5-host NVMe-based cluster, replicating to a second RGW instance over a 9-node cluster with 62 10K SAS hard drives. The client ran three copies of Filebench *fileserver*, generating hot, medium, and cold data, with file set sizes of 50 K, 200 K and 800 K respectively. Objects older than 60 seconds were copied from the primary to the replica.

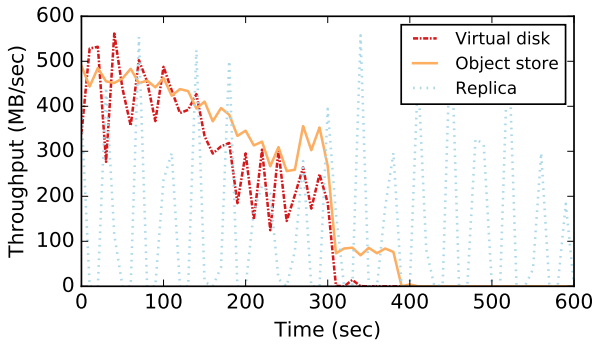

Figure 13. Data transfer during asynchronous replication.

Data transfer during the experiment is shown in Figure 13. I/O writes to the virtual volume (as measured by `iostat`) track object writes to the primary store for the first 200 seconds, after which the two diverge somewhat due to garbage collection. The asynchronous replication process starts almost immediately, periodically copying objects to the secondary store. Over the course of the experiment 103 GB of data are written to the LSVD virtual disk, while as a result of garbage collection deleting some fraction of objects before being replicated, only 85 GB of data are copied to the remote replica.

While at times objects appeared in the second cluster out of order, the standard LSVD recovery strategy was sufficient; a consistent disk could be created on the second RGW cluster based on the available sequence of objects. This experiment provides strong initial evidence that LSVD provides a natural model for replicating virtual disks across geographies.

4.8 Deployability

LSVD performs all its work in the client, and can exploit any S3 implementation as a back-end. As a result a user can deploy a LSVD-enabled client in a cloud, using it as an alternative to virtual disks provided by that cloud.

[10]This description ignores several subtleties: the garbage collector ensures map checkpoints are persisted before objects are deleted, but an independent replicator cannot maintain this guarantee. Signaling between the two (e.g. via a well-known object name) could address this; more possibilities are discussed in §5.

To validate this, we ran experiments on Amazon AWS using S3 for the backend and a (m5d.xlarge) EC2 instance as a client with 4 vCPUs, 16GB RAM, 10G NIC and a dedicated 150GB NVMe drive, with measured read and write bandwidth of 230MB/s and 128MB/s for large I/O size and queue depth. Both S3 and EC2 instance are in the same datacenter (us-east-1) and EC2 instance is running Ubuntu Linux 18.04.5 LTS (kernel 5.0.0-1021-aws).

We note that peak LSVD I/O rates for random read are close to the maximum available provisioned IOPS level for EBS (64,000); however while the local NVMe and remote S3 needed by LSVD cost a few dollars per month at on-demand prices, at the on-demand price of $0.065/mo per IOPS above 1000, a 50,000 IOPS EBS volume would cost over $3000 per month.

5 Concluding Remarks

LSVD combines durable cache and log-structured storage; prior work has primarily considered one of these or the other, but not the potential of combining both. Our prototype has demonstrated that with this strategy we can: 1) get massive gains in performance over existing network-mounted virtual drives (16x), with dramatically reduced demand on the storage service (25x), 2) preserve the reliability & functionality (e.g, migration, snapshots) advantages of today's virtual disks, and 3) naturally support new functionality such as asynchronous replication over geographic distances and user deployment. These compelling results have resulted in a new open-source effort with industry collaborators.

In this section we discuss the lessons learned from this prototype: which decisions were validated, and which should be revisited. In addition we describe new directions of potential research which have been opened up by this approach.

5.1 The Good

There are a number of key design decisions in the prototype that we are moving forward with in the open source project.

Client-side Implementation. LSVD performs out-of-place writes, mapping, and volume management on the client, relying on the back-end for simple immutable storage. This client-side approach is of course shared with prior client-side caches, but differs from all virtual disks in the literature with the exception of Blizzard [19].

Our experience appears to be a strong validation of the Blizzard and LSVD approach. A virtual disk with out-of-place writes requires a strongly-consistent translation map, updated on every write operation, with client I/O stalled any time the map is unavailable. Achieving this in a distributed system is complex; in contrast a simple non-replicated local map will be available at all times a client can issue write operations, as well as providing a modest gain in performance.

In-memory Map. Rather than using an on-SSD map with in-memory caching, the LSVD prototype relies on an extent map maintained exclusively in memory, with updates journaled to SSD and the remote log. The resulting map is simple and fast: a tuned version takes about 0.2 μs for lookups and 0.5 μs for updates on a 10M-entry map, resulting in negligible performance overhead in our target scenario. At the same time its memory requirements are modest: the same tuned map requires about 24 bytes per entry, or 0.5 GB for a 20M-entry map[11]. An excessively fragmented map could result in higher memory use (as well as reduced read performance due to fragmenting of operations [12]) but our simulations show that modest changes to the garbage collection algorithm are able to bound this fragmentation.

Journaled Cache. This was used in our earliest prototype to ensure consistency while batching writes, however it has many other advantages even when batching is moved into memory. It incurs a modest increase in the volume of data written, but requires no additional write operations on commit barriers (e.g. to persist dirty map entries); full metadata checkpoints are performed lazily and do not block user I/O. For sync-heavy workloads such as Filebench "oltp" (§4.2.2) this results in substantially higher local cache performance than that achieved by `bcache`, a non-journaled cache.

Atomic Batching of Writes. As argued by Koller [15], persisting writes in the order received will guarantee prefix consistency, even if commit barriers are ignored. Yet ensuring this ordering is difficult over block interfaces, where the only transaction-like mechanism is the commit barrier. As a result, prefix-consistent caches over block back-ends [24] must flush their "pipeline" of outstanding writes at every commit barrier, greatly decreasing throughput on sync-heavy workloads. In contrast the combination of out-of-place writes and atomic object creation maintains the ordering needed for prefix consistency, even while supporting high levels of concurrency.

5.2 The Bad

In working towards a larger open source project, we are moving away from a number of the design decisions described in the prototype.

Kernel/User Implementation. The local journal was implemented in the kernel for highest performance. Yet in many hypervisors—in particular our target, KVM/QEMU—the I/O stack is in user space, and our design increases rather than decreases the number of kernel/user boundary crossings. In addition the split implementation places significant constraints on design of the cache, due to the difficulty of locking data structures across this boundary, and loses access to the

[11] If we adopt a rule of thumb that the cost of RAM for a virtual disk should be less than the cost of the managed local storage, this would limit us to perhaps 100 small virtual disks sharing a several-TB SSD. For large disk images we note that the cost of RAM for a map is trivial compared to the market price of the volume; e.g. a 10 TB volume costs $300-$800/mo from public cloud providers.

hypervisor buffer cache. For these reasons our open-source collaboration is focused on a pure user-space implementation.

Multiple Translation Maps. Our prototype maintains multiple translation maps and block allocators to avoid kernel/user locking issues or additional system calls. This results in poor utilization of the cache device, and prevents implementation of several research ideas discussed below.

Lack of Ecosystem. By focusing on the implementation of a local block device, we missed the signficance of the broader virtual disk ecosystems of file systems and tools. In retrospect this was a mistake, complicating development, testing, and evaluation. It now appears that we can adapt existing library interfaces for a user-space virtual disk, enabling simple integration into these ecosystems, and providing access to existing test and deployment systems as well as a community of users.

5.3 The Research

This work proposes to directly link client-side caching and remote virtual disk storage in a way which has not been done before; in doing so it raises research questions beyond those addressed in this work.

Garbage Collection. The object-based LSVD backend differs in many ways from the flash and shingled disk media of previous translation layers. Its capacity is elastic, and erase units are of variable size. Per-operation overhead is intermediate between that of flash (i.e. negligible) and shingled disk (high), read costs are non-negligible (unlike flash), and unlike either flash or disk there is little or no interference between concurrent operations.

One area of future investigation is the use of cached data in garbage collection: in which cases are we better off performing a "cheaper" copy using cached data, rather than an optimal one requiring remote reads? An additional question is that of cleaning within an elastic "arena", rather than one of fixed size as in an SSD or SMR disk.

Defragmentation. We have investigated defragmentation primarily as a mechanism for allowing the translation map to be maintained in RAM; however there are also performance impacts of fragmentation when storage has non-negligible per-operation overhead [12]. We have prototyped simple implicit and explicit strategies for reducing fragmemtation during garbage collection, but have explored only a fraction of the design space and done little to compare the performance gains or impacts of different approaches.

Cache Sharing. It is common for a single virtualization host to run multiple virtual machines, each with disks cloned from the same base image, comprised of the same objects on back-end storage. If we modify the LSVD mapping strategy to tag cache entries by "physical" addresses (i.e. back-end object and offset) rather than virtual addresses (i.e. LBAs) we believe we need only retrieve and cache this shared data once, eliminating network I/O as as successive VMs access shared blocks.

This in fact points to a more general-purpose ability to leverage the copy-on-write nature of LSVD for de-duplication. By pre-processing the base image we can eliminate duplicate blocks, pointing multiple LBA extents to the same back-end object data, similar to VMAR's de-duplication translation maps [30] but simpler to implement. Speculating further, it may be possible for the hypervisor to use this information for efficient transparent page sharing [34] based on page origin rather than exhaustive comparison.

Cache Placement and Pre-fetching. LSVD batches data *temporally*, i.e. in the order it is written, rather than spatially. We have not yet explored the potential of "temporal read-ahead" based on this structure, or the impact of restoring spatial ordering during garbage collection. Additional areas of investigation include persisting the state of the cache replacement algorithm across shutdown and restart, and potential pre-population of the cache [25] on VM migration.

Asynchronous Replication. The use of an immutable object stream enables asynchronous replication; as noted in §4.7 this requires some synchronization between the replication and garbage collection processes. An explicit process is described, but implicit synchronization could be performed by e.g. setting a minimum age before which objects would not be deleted. For many workloads the replication bandwidth would be significantly reduced (as shown in §4.7) by deferring replication until one or more cycles of garbage collection have been performed, but more work is needed to determine how these savings may be achieved without compromising consistency or *recovery point objective* (RPO, or "freshness" of the replica).

References

[1] Apache Foundation. 2021. Apache HBase. http://hbase.apache.org.

[2] Jens Axboe. 2021. fio. https://github.com/axboe/fio.

[3] Steve Byan, James Lentini, Anshul Madan, and Luis Pabón. 2012. Mercury: Host-side flash caching for the data center. In *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. 1–12.

[4] CloudBD. 2021. CloudBD. https://www.cloudbd.io.

[5] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP '07)*. ACM, New York, NY, USA, 205–220. https://doi.org/10.1145/1294261.1294281

[6] Peter Desnoyers. 2014. Analytic Models of SSD Write Performance. *ACM Transactions on Storage* 10, 2 (March 2014), 8:1–8:25.

[7] Linux Device-Mapper. 2001. Device-Mapper Resource Page. https://sourceware.org/dm/.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. 2003. The Google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*. ACM, Bolton Landing, NY, USA, 29–43. https://doi.org/10.1145/1165389.945450

[9] Alain Girault, Gregor Gössler, Rachid Guerraoui, Jad Hamza, and Dragos-Adrian Seredinschi. 2018. Monotonic Prefix Consistency in Distributed Systems. In *Formal Techniques for Distributed Objects,*

Components, and Systems, Christel Baier and Luís Caires (Eds.). Springer International Publishing, Cham, 41–57.

[10] Google. [n.d.]. Persistent Disk. https://cloud.google.com/persistent-disk/.

[11] R. Hagmann. 1987. Reimplementing the Cedar file system using logging and group commit. In *ACM SIGOPS Operating Systems Review (SOSP '87)*. ACM, New York, NY, USA, 155–162. https://doi.org/10.1145/41457.37518 ACM ID: 37518.

[12] Mohammad Hossein Hajkazemi, Mania Abdi, and Peter Desnoyers. 2018. Minimizing read seeks for SMR disk. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 146–155.

[13] Dean Hildebrand and Denis Serenyi. 2020. A peek behind the VM at the Google Storage infrastructure. cloud.google.com/blog/products/infrastructure/google-cloud-next20-onair-infrastructure-week-session-list.

[14] Norman C. Hutchinson, Stephen Manley, Mike Federwisch, Guy Harris, Dave Hitz, Steven Kleiman, and Sean O'Malley. 1999. Logical vs. Physical File System Backup. In *Proceedings of the Third Symposium on Operating Syst ems Design and Implementation (OSDI '99)*. USENIX Association, USA, 239–249.

[15] Ricardo Koller, Leonardo Marmol, Raju Rangaswami, Swaminathan Sundararaman, Nisha Talagala, and Ming Zhao. 2013. Write Policies for Host-side Flash Caches. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*. USENIX, San Jose, CA, 45–58.

[16] Dong-Yun Lee, Kisik Jeong, Sang-Hoon Han, Jin-Soo Kim, Joo-Young Hwang, and Sangyeun Cho. 2017. Understanding Write Behaviors of Storage Backends in Ceph Object Store. In *Proceedings of the IEEE International Conference on Massive Storage Systems and Technology*. IEEE, Santa Clara, CA.

[17] Huiba Li, Yiming Zhang, Dongsheng Li, Zhiming Zhang, Shengyun Liu, Peng Huang, Zheng Qin, Kai Chen, and Yongqiang Xiong. 2019. Ursa: Hybrid block storage for cloud-scale virtual disks. In *Proceedings of the Fourteenth EuroSys Conference 2019*. 1–17.

[18] Jai Menon and Jim Cortney. 1993. The architecture of a fault-tolerant cached RAID controller. In *Proceedings of the 20th annual international symposium on Computer architecture - ISCA '93*. ACM Press, San Diego, California, United States, 76–87. https://doi.org/10.1145/165123.165144

[19] James Mickens, Edmund B. Nightingale, Jeremy Elson, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan, and Krishna Nareddy. 2014. Blizzard: Fast, Cloud-scale Block Storage for Cloud-oblivious Applications. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. USENIX Association, Seattle, WA, 257–273.

[20] Microsoft Corp. 2019. Disk Storage – HDD/SSD on Azure | Microsoft Azure. https://azure.microsoft.com/en-us/services/storage/disks/.

[21] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. 2012. Flat Datacenter Storage. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, Berkeley, CA, USA, 1–15.

[22] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 305–319.

[23] John Ousterhout and Fred Douglis. 1989. Beating the I/O Bottleneck: A Case for Log-structured File Systems. *SIGOPS Oper. Syst. Rev.* 23, 1 (Jan. 1989), 11–28. https://doi.org/10.1145/65762.65765

[24] Dai Qin, Angela Demke Brown, and Ashvin Goel. 2014. Reliable Writeback for Client-side Flash Caches. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. USENIX Association, Philadelphia, PA, 451–462.

[25] Francisco Romero, Gohar Irfan Chaudhry, Iñigo Goiri, Pragna Gopa, Paul Batum, Neeraja J. Yadwadkar, Rodrigo Fonseca, Christos Kozyrakis, and Ricardo Bianchini. 2021. Faa$T: A Transparent Auto-Scaling Cache for Serverless Applications *(SoCC'21)*. Seattle, Washington, USA. https://doi.org/10.1145/3472883.3486974

[26] Mendel Rosenblum and John K. Ousterhout. 1991. The design and implementation of a log-structured file system. In *13th ACM symposium on Operating systems principles*. ACM, Pacific Grove, California, United States, 1–15.

[27] Corey Sanders. 2017. Announcing general availability of Managed Disks and larger Scale Sets.

[28] Amazon Web Services. 2021. Amazon Elastic Block Store. https://aws.amazon.com/ebs/.

[29] Venky Shankar. 2017. Deep Dive into Ceph RBD. www.snia.org/events/sdc-india/archive/sdc-india-2017-presentations.

[30] Zhiming Shen, Zhe Zhang, Andrzej Kochut, Alexei Karve, Han Chen, Minkyong Kim, Hui Lei, and Nicholas Fuller. 2013. VMAR: Optimizing I/O Performance and Resource Utilization in the Cloud. In *Middleware 2013*. Vol. 8275. Springer Berlin Heidelberg, Berlin, Heidelberg, 183–203. https://doi.org/10.1007/978-3-642-45065-5_10 Series Title: Lecture Notes in Computer Science.

[31] Spencer Shepler, Eric Kustarz, and Andrew Wilson. 2008. The New and Improved FileBench File System Benchmarking Framework.

[32] William Stearns and Kent Overstreet. 2010. Bcache: Caching beyond just RAM.

[33] Ross Stenfort, Lee Prewitt, and Paul Kaler. 2021. Challenges in Hyperscale: What Hyperscalers Care About. https://searchstorage.techtarget.com/post/Challenges-in-Hyperscale-What-Hyperscalers-Care-About.

[34] Carl A. Waldspurger. 2002. Memory resource management in VMware ESX server. In *Proceedings of the 5th symposium on Operating systems design and implementation*. USENIX Association, Boston, Massachusetts, 181–194.

[35] Carl A. Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. 2015. Efficient MRC Construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*. USENIX Association, Santa Clara, CA, 95–110.

[36] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. 2013. Robustness in the Salus Scalable Block Store. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 357–370.

[37] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. 2013. Robustness in the Salus Scalable Block Store. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 357–370.

[38] D.L. Willick, D.L. Eager, and R.B. Bunt. 1993. Disk cache replacement policies for network fileservers. In *[1993] Proceedings. The 13th International Conference on Distributed Computing Systems*. IEEE Comput. Soc. Press, Pittsburgh, PA, USA, 2–11. https://doi.org/10.1109/ICDCS.1993.287729