Beating the I/O Bottleneck: A Case for Log-Structured Virtual Disks

Mohammad Hossein Hajkazemi[*][†]
NetApp Inc.
Boston, Massachusetts, USA
mhajkaze@netapp.com

Vojtech Aschenbrenner[*][‡]
EPFL
Lausanne, Switzerland
vojtech.aschenbrenner@epfl.ch

Mania Abdi
Northeastern University
Boston, Massachusetts, USA
abdi.ma@northeastern.edu

Emine Ugur Kaynar
Boston University
Boston, Massachusetts, USA
ukaynar@bu.edu

Amin Mossayebzadeh
Boston University
Boston, Massachusetts, USA
mosayyeb@bu.edu

Orran Krieger
Boston University
Boston, Massachusetts, USA
okrieg@bu.edu

Peter Desnoyers
Northeastern University
Boston, Massachusetts, USA
p.desnoyers@northeastern.edu

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) that couples log-structured approaches at both the cache and storage layer to provide a virtual disk on top of S3-like storage. Both cache and backend store are order-preserving, enabling LSVD to provide strong consistency guarantees in case of failure. Our prototype demonstrates that the approach preserves all the advantages of virtual disks, while offering dramatic performance improvements over not only commonly used virtual disks, but the same disks combined with inconsistent (i.e. unsafe) local caching.

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

Keywords: cloud computing, virtual disk, block storage

ACM Reference Format:
Mohammad Hossein Hajkazemi, Vojtech Aschenbrenner, Mania Abdi, Emine Ugur Kaynar, Amin Mossayebzadeh, Orran Krieger, and Peter Desnoyers. 2022. Beating the I/O Bottleneck: A Case for Log-Structured Virtual Disks. In *Seventeenth European Conference on Computer Systems (EuroSys '22), April 5–8, 2022, RENNES, France.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3492321.3524271

1 Introduction

Network mounted virtual disks [8, 18, 26, 27] 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 cloning of disk images.

In the past, virtual disks could also provide both price and performance advantages over local hard drives (HDDs), using statistical multiplexing for higher peak performance with fewer total spindles. This statistical multiplexing, however, relied on network bandwidth being much greater than the speed of the aggregate HDDs. Today, local SSDs achieve speeds that would saturate most network connections, and virtual disk performance is no longer competitive with local storage,

One can employ SSDs for a client-side cache on top of a virtual disk [2, 13, 22]; however, while a large cache can eliminate all reads, existing write-back caches either risk corruption and loss in the case of failure [21] or limit the number of concurrent writes to the backend [2, 22]. Moreover, only concurrent writes that are adjacent in the virtual disk can be combined into larger writes and (see §2.1) today's scale-out storage backends perform poorly for small writes.

We propose a *Log-Structured Virtual Disk* (LSVD) a new way to provide the abstraction of a virtual disk based on log-structured approaches for both caching and storage. All LSVD

[*] Both authors contributed equally to this research.
[†] Work performed while at Northeastern University
[‡] Work performed while at Charles University, Prague

EuroSys '22, April 5–8, 2022, RENNES, France
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9162-7/22/04.
https://doi.org/10.1145/3492321.3524271

functionality runs at the client system on top of a locally attached SSD for the cache, and writing to immutable object storage (e.g. S3) for long term durability. Writes to virtual disk blocks are logged in the cache along with the meta-data that describes them. When a large batch is accumulated, it is written as an object, where the object name identifies the order in the storage log. The client maintains in-memory maps to identify live data in the logs for reads, and when the live data in an object drops below a threshold uses garbage collection to write the data to a new object.

We have built a simple prototype, split across a kernel module exposing a block device and implementing the cache, and a user-level service to perform reads and writes to the object store and perform garbage collection. We compare performance of our prototype against a popular open source virtual disk implementation (Ceph RADOS Block Device [27], or RBD) coupled with a popular writeback cache (bcache [30]). We show that an LSVD approach can:

1. **Dramatically boost performance.** For example, with a random write workload (§4.5) a single 16 KB client write to LSVD generates 0.25 back-end disk I/Os, compared to 6 for RBD. This 24x improvement in I/O efficiency has corresponding effects on performance: in a load test LSVD achieved 50,000 client IOPS while leaving the back-end disks over 90% idle while RBD achieved only 12,000 IOPS. For sync-heavy workloads, LSVD's cache out-performs bcache (e.g., 4x §4.2.2), due to the elimination of additional I/Os for metadata persistence.
2. **Preserve key properties of today's virtual disks.** LSVD recovers all committed writes (see §2.2) in the case of a client crash and recovery, and guarantees *prefix consistency* in the case where the local cache is lost—the same guarantee provided by prior work such as Blizzard [17] and Salus [33].[1] The efficiency of LSVD writes enables it to more efficiently synchronize the cache and backend than caches layered over virtual disks (e.g. 10x faster than bcache §4.4); 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 (§4.8) through simple object copy commands. In addition since all functionality runs at the client, our virtual disk can be employed in existing clouds with no support from the cloud vendor. LSVD on AWS (§4.9) is able to achieve an IOPS rate for a few dollars a month that would have cost over $3000/mo using AWS EBS.

While there has been much research on both client side caching [2, 13, 22, 30] and novel virtual disk implementations based on out-of-place writes [16, 17, 33], LSVD differs from previous work both in coupling a client cache with a log-structured backend, and in introducing the idea of a log-structured cache. Given the novelty of an LSVD approach, there are numerous alternatives on how one should, for example: 1) maintain a logging cache, 2) support snapshots and clones, 3) perform garbage collection, 4) split implementation between kernel and user level, etc. We describe the set of design decisions we used in our prototype, then based on this experience, discuss opportunities for improvement and research directions which LSVD makes possible.

After motivation (§2), we describe the architecture and implementation of our prototype (§3), evaluate our prototype (§4), survey related work (§5) discuss lessons learned from the prototype (§6) and conclude.

2 Background and Motivation

A large fraction of the computation performed today runs on virtual machines, with these virtual machines typically running on remote virtual disks implemented on scale-out storage systems. Cloud users may use these virtual disks directly, as e.g. Amazon EBS [26], Google Persistent Disk [8], or Azure Managed Disks [18]—or indirectly in higher-level services such as containers or serverless functions. Additional virtual machines and disks underly the hyper-scale services we use every day, from email to e-commerce and social media. Any possible improvements in the efficiency of 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

Given a storage cluster composed of some number of IOPS-limited devices (HDDs or capacity SSDs), the total rate at which operations may be performed on the pool of devices is fixed, but the achievable user operation rate is determined by how efficiently the storage system translates client operations to reads and writes of the back-end devices. Individual client reads typically translate into a single read to a back-end device, but writes suffer *write amplification*: a single client write results in multiple writes to back-end devices, for reasons of resiliency and consistency. In such a system, the available client write IOPS are equal to the aggregate write IOPS of the storage pool, divided by this write amplification.

Where does this amplification come from? The first source is to provide redundancy for recovery in the face of device failure. For each data byte written by the client, three bytes must be written to separate disks for triple replication, and ~1.5 bytes for typical erasure coding configurations. The second source is due to the architecture of traditional virtual disks.

[1] We argue (§2.2) that this is sufficient for some of the most demanding storage-based services.

With a block-style interface they are rarely able to batch multiple writes together[2]; as a result the small client-side writes typical of virtual disk workloads [17] result in small writes at the backend. In the best case (e.g. server-side logging with deferred batch write [33]) these small writes must be replicated and performed immediately; in other designs additional writes may needed to ensure consistency if a replica fails, and yet more if erasure-coded parity blocks must be recomputed and rewritten. This write I/O amplification can be large: 6x in our experiments (§4.4) with Ceph RADOS Block Device (RBD) [3], while others report up to 13x [15]. This overhead is especially problematic for HDD-based backends, which remain in common use[3].

Rather than amplifying client writes, LSVD *reduces* them, adding a cache and refactoring the client-to-remote interface to keep small writes local, and send large contiguous writes to the backend. This (a) reduces the aggregate IOPS required of the storage backend, and (b) by avoiding small writes allows efficient use of erasure coding, with its higher capacity and write throughput.

2.2 Consistency

Block devices may provide various levels of consistency, which we describe 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. Disk clients (e.g. file systems) must pause writes during a commit barrier operation; any write is thus ordered with respect to a commit barrier, occurring either before or after it.

By making writes before a commit barrier durable, a disk or SSD guarantees that *all committed writes* will be reflected in the device contents after a crash. LSVD, like most virtual disks [17, 33] and consistent write-back caches [22], provides a weaker guarantee in the worst case: *prefix consistency.* A prefix-consistent system may lose some committed writes due to a crash, but guarantees that the recovered state reflects a *consistent prefix* of these writes. If the system crashes at time t, the recovered state will reflect (a) all committed writes at some $t' < t$, and (b) no writes which were issued after t'.

The potential loss of committed data in a prefix-consistent system is readily tolerated by many applications, e.g. "stateless" servers which are restarted from a fresh image after any failure or shutdown. More demanding applications often tolerate it, as well, via mechanisms for recovery from temporary failure. Consider Dynamo [5] as an example, where updates at each node are versioned and queries requiring consistency guarantees are sent to multiple replicas. If a node missed an update when it was down, its reply will be ignored and it will eventually recover the most recent value via the Dynamo anti-entropy protocol; the same action would be taken if a prefix-consistent disk lost an update received just prior to crashing. Similarly, an examination of Raft [20] shows that it will tolerate prefix-consistent failures, using its AppendEntries mechanism. We believe that most distributed, replicated systems will tolerate prefix-consistent data loss, if they (a) include mechanisms to recover from temporary node failures, and (b) use only local state to determine the recovery point. For such a system, prefix-consistent data loss will have the same effect as a slight increase in failure duration.

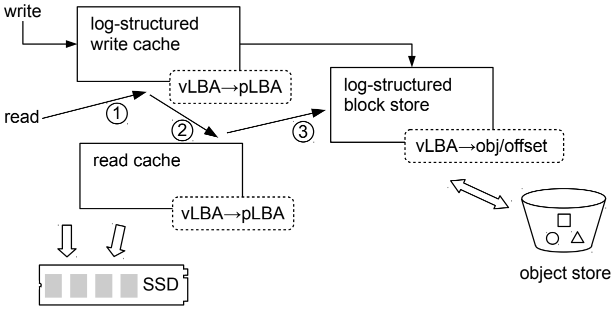


Figure 1. LSVD architecture: log-structured write cache, read cache, and log-structured block store. In-memory maps translate from virtual disk LBA (vLBA) to physical SSD location (pLBA) and remote object location (object/offset). Reads check in turn (1) the write cache, (2) read cache, and then (3) remote storage.

3 Architecture and implementation

The LSVD virtual disk is implemented at the client, using a standard S3-compatible object store as its backend. It sends incoming writes to a log-structured writeback cache, then aggregates them and stores them in a log-structured sequence of named objects, maintaining end-to-end write ordering for consistency. For reads, in-memory maps identify the local of live disk blocks, and blocks are cached in a separate large read cache.

These components may be seen in Figure 1; we describe them (§3.1), their behavior for write, read, and commit barrier operations (§3.2), explain how metadata is persisted and recovered on failure (§3.3), consistency (§3.4), garbage collection (§3.5), and volume management features (i.e. snapshot and clone) (§3.6), and finally our implementation (§3.7).

3.1 Basic components

Log-structured writeback cache: The write-back cache is designed to handle writes at high speed, persisting them locally while they are written to the backend in large batches

[2] Small write sizes measured in previous work are seen *after* write merging in a guest operating system.

[3] We note that at the time of publication, Google Persistent Disk defaults to HDD-based volumes.

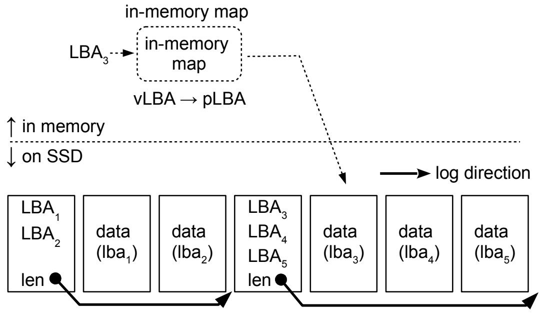


Figure 2. Records in log-structured on-SSD write cache and associated in-memory map. The map may be recovered from log record headers; additional header fields (sequence number, CRC) ensure atomicity of log writes.

by the object store. The on-SSD log structure is shown in Figure 2, and is similar to the physical journal in ext3/ext4: a log record contains a series of data blocks, with a header indicating the logical block address (LBA) in the virtual disk corresponding to each block, with a header sequence number and CRC (covering both header and data) ensuring that only complete log records are used in recovery.

Implementing the cache as a log means: 1) write ordering is maintained, in turn allowing ordering to be maintained in the log-structured block store, 2) small and random writes are fast, as they are translated into contiguous log records, merged by lower I/O layers, and written sequentially, and 3) commit barriers are fast (§4.2.2), requiring only a commit operation to the cache SSD to ensure all prior log records are durable.[4] However, a log has several disadvantages: it is space-inefficient, using 4 KB alignment and expanding small writes by as much as 100%; its map is memory-inefficient (see below) and it is difficult to perform non-FIFO eviction.

Read cache: We devote much of the SSD to a standard read cache, which uses SSD space more efficiently, can use more efficient mapping (see below) and can provide LRU or similar eviction policies. Maintaining a separate read cache also greatly simplifies dealing with write-after-read hazards, where newer writes risk being replaced by with stale data from the backend; LSVD simply prioritizes serving reads from the write cache.

Log-structured block store: The log-structured block store (Figure 3) collects batches of writes and stores them in an ordered stream of objects, using object names to indicate order, e.g. vdisk.001, vdisk.002, etc. The block store writes data to new objects, maintaining a map of the current location of each logical block, and using garbage collection to reclaim space by copying live data from old objects and deleting them. As with the write-back cache, objects in the block store include headers with LBA information (see Figure 4, allowing the in-memory map to be reconstructed on startup or after failure.

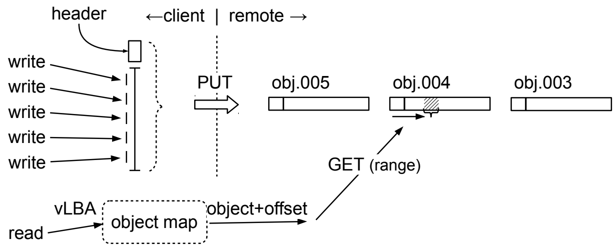


Figure 3. Log-structured block store. Writes are batched and written to immutable objects; reads are routed by an in-memory map. Garbage collection is not shown.

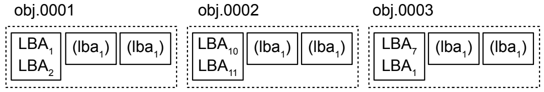


Figure 4. Backend objects and metadata headers. Again the header indicates LBAs of following data blocks; typical object aggregates 4-32 MB of writes.

Since objects are written atomically, individual writes within an object may be re-ordered without compromising the ordering guarantee: if the effect of some write w_j is seen in the object stream at any point, then all $w_i | i < j$ will be reflected in the state as well. Writes may thus be coalesced within a single batch, although not across batches.

Maps: For read performance, LSVD maintains in-memory maps, for the write-back cache, the read cache, and the block store to identify the location of live data. One important issue is bounding the amount of memory used for these maps.

Memory usage for the caches are bounded: no matter how many virtual disks are deployed on a host, the amount of cache SSD to be mapped is constant. The majority of the SSD is used for the read-cache, and standard techniques can be used to arbitrarily limit the map. For example, with a block size of 128 KB, at 24 bytes per entry the in-memory map per TB of read cache will be less than 200 MB. The write-back cache is mapped at much finer granularity; assuming a mean write size of 16 KB plus 4 KB overhead it would require 1.2 GB per TB; if 20% of the SSD is dedicated to write cache, total memory used will be less than 300 MB per TB of cache SSD.

Memory usage for the object map, however, scales with the number of virtual disks. As we will see, extents in this map can be much larger than in the write-back cache, as writes can be coalesced before objects are stored; in addition we show in §4.6 that the garbage collection algorithm can defragment the map further. At a mean extent size of 128 to 256 KB, which appears reasonable based on our trace-driven simulations, a 1 TB virtual disk image would require 100 to 200 MB of memory.

[4] In contrast, other write-back caches [4, 21] must write multiple pages to SSD to make preceding writes durable.

3.2 Basic operations

As a block device, LSVD implements three operations: write, read, and commit barrier. For simplicity we describe LSVD behavior for single-block read and write operations, to an address identified by a virtual disk LBA (vLBA).

Write: Writes are directed to the write-back cache, where a log header is prepended and the record is written to SSD. On completion of this write, (a) the write cache map is updated with the new vLBA to physical SSD LBA (pLBA) mapping, (b) the caller (e.g. file system) is notified of I/O completion, and (c) a copy of the write is sent to the block store. The block store aggregates writes, storing a batch when it reaches a configured size. (e.g. 8 or 32 MB) Each batch is assigned a sequence number and written to an object named with that number, then mappings for data contained in the object (i.e. vLBA to object/offset) are entered in the object store map.

Commit barrier: These are handled by the write-back cache: a commit barrier operation is sent to the cache SSD; when it completes, all preceding LSVD writes will be durable.

Read: This may be seen in Figure 1: reads check in turn (1) the write-back cache, (2) the read cache, and (3) the log-structured block-store, returning the first match found. If the vLBA is present in either of the cache maps, the data is read from SSD and returned immediately. If not, and if the vLBA is present in the object map, a range read request is used to fetch a block containing the requested data. The requested data is then returned to the caller, and the block is entered into the read cache. Unitialized disk blocks will not be present in the object map; per standard disk semantics, they will read as zeros.

Read prefetching is performed by reading a larger range than requested, and entering all retrieved data into the read cache, making the additional data available for subsequent read operations. Due to the order-preserving nature of the LSVD block store, this strategy will prefetch based on *temporal locality* rather than *spatial locality*—i.e. it will prefetch data written at the same time as the triggering read, whether or not it was written to nearby addresses.

Finally, since loss of data in the read cache does not affect correctness, there is no need to protect its metadata with logging. To avoid the need to re-fetch data after restart or failure, the read cache map is periodically persisted to SSD.

3.3 Metadata persistence and recovery

All the information needed to recover the in-memory maps for the (a) write-back cache and (b) block store may be recovered from the corresponding log headers, applying vLBA to location mappings in the order seen in the log. To bound the time needed for map recovery from its associated log, we periodically *checkpoint* each map, writing a full copy of the writeback cache map to a fixed location on SSD, and the object map to a numbered object in the object store. At startup we locate the most recent checkpoints and load them into their respective maps, and then replay map updates from the checkpoint to the end of the log.

A key consideration in replaying the log is to determine where it ends. Both caches issue multiple overlapped writes for higher throughput; however typically neither the SSD nor the backend (e.g. NVMe and S3, respectively) guarantee in-order completion of requests. Log recovery stops at the end of the consecutive sequence of records, e.g. if objects 99, 100, and 102 are seen on the backend, we take only 99 and 100. This is done implicitly in an on-disk log, where recovery stops at the first invalid log header, detected by sequence number and CRC. For the object log these "stranded" writes are visible, and are deleted during recovery. To avoid interference with recovery, the garbage collector only deletes objects that are older than the most recent checkpoint; the "holes" created will not be seen during the recovery process.

When recovering from a client crash, there will typically be writes present in the local cache which are not reflected in the backend store. Before making the virtual disk available we (1) delete any backend objects ignored by the above prefix rule, (2) "rewind" the cache log to a point at or before the end of the most recent backend object, and (c) replay all writes after this point, bringing the backend image up to date with respect to the cache. In the case of further failure, the steps may be repeated without risk of inconsistency.

3.4 Consistency

Recovery from local cache preserves the same consistency guarantees as provided by the cache device itself. In particular, write completion is reported when the device acknowledges the corresponding log write, and commit barriers ensure that all log records for preceding writes have been made durable.

More serious failures, e.g. permanent device or machine failures, may result in loss of all cached data. In this case LSVD will lose any writes being batched for writing to the backend, as well as any data in flight to the object store. In this case the virtual disk will be *prefix consistent* [31]: since we recover a consistent prefix of objects, and preserve write ordering, the virtual disk will reflect a consistent prefix of committed writes. As described above (§2) this worst-case consistency guarantee is the same as provided by prior work such as Qin et al. [22], Salus [33] or Blizzard [17], and is sufficient for many distributed systems such as Dynamo [5] or Raft [20].

3.5 Garbage collection

Like other log-structured systems [10, 23] , the log-structured block store performs garbage collection to reclaim storage used by data blocks which have been overwritten. Garbage collection is triggered when overall utilization (i.e. ratio of "live" data to total object size) drops below a fixed threshold, 70% in our experiments. The Greedy cleaning algorithm [23] is used, selecting the least-utilized objects for collection. Live data from the selected objects is (a) fetched using a combination of range and whole-object read operations, and (b) written to

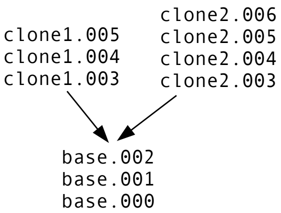


Figure 5. Object streams for two clone volumes from a single base image. The "clone1" and "clone2" object sequences each contain copy-on-write changes to the data in the "base" objects.

new objects, after which (c) the object map is updated, and (d) the old objects are deleted.

Several steps are taken to optimize this process. An in-memory table tracks total object sizes and remaining live data, allowing efficient selection of cleaning candidates; this table is persisted with map checkpoints, then loaded and updated during the recovery process. To efficiently identify the remaining live data in an object we retrieve the object header, which lists the live extents held in that object at the time of its creation; only these ranges need be examined in the object map to find any remaining live data. Finally, we note that in many cases the data needed for garbage collection may be found in the local cache, eliminating the need for it to be fetched during the cleaning process.

3.6 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 5—each clone shares a common prefix of objects, with that prefix making up the clone base. This may be done by using a different string to form object names, as in the example, with objects in the base image having the form "base.N" while the two clones have object streams "clone1.N" and "clone2.N", where N is the object sequence number. In our implementation the object map stores only sequence numbers, not full object names; for clones the read logic uses the base image name ("base" in Figure 5) plus a sequence number for objects in the clone base ($N \leq 2$ in figure), and the name of the cloned volume (e.g. "clone1") plus a sequence number for more recent ones. The garbage collector is aware of clones, and only cleans and deletes objects within the clone image, i.e. $N \geq 3$. Finally, since the clone base remains unchanged even after all clones are deleted, there is no need for reference counting mechanisms.

LSVD supports snapshots in the same fashion as many other log-structured systems [12, 14], by (a) saving a pointer to the log head at the time of the snapshot, and (b) preventing the garbage collector from removing data referenced by a snapshot. Any object in the object stream can be designated as a snapshot, and can be mounted read-only by backtracking to the last map checkpoint before that point, and recovering up to the snapshot point but no farther. To create a writable snapshot, a clone can be created based on the snapshot point.

The snapshot mechanism makes minimal changes to the garbage collector, which continues to run as described above, but defers deletion of garbage-collected objects. To explain this deferral, we first note that a snapshot at sequence N depends on all objects prior to N which had not been deleted at the time of the snapshot. Given a set of volume snapshots pointing to object numbers $N_1 > N_2 > \ldots$, the garbage collector is free to delete any objects numbered $N_x > N_1$, as those objects are newer than any snapshot. For any other objects the collector records the object number (N_0) and the newest object number at the time of garbage collection, N_{gc}. If there is no intervening snapshot N_x, i.e. $N_0 < N_x < N_{gc}$, then the object may be deleted; otherwise the pair N_0,N_{gc} is added to the list of deferred deletes, which is persisted to the backend as part of volume metadata, i.e. in object headers and map checkpoints. When a snapshot is deleted the list is reexamined, and we perform any deferred delete which has now become allowable.

3.7 Implementation

The LSVD prototype evaluated in this paper is implemented as a Linux *device mapper* [6], providing a native block device to file systems and other applications. The kernel subsystem implements the log-structured cache, and is coupled (via an `ioctl` interface) to a user-space daemon implementing the log-structured block store.

The kernel device mapper is implemented in $\sim$1000 lines of C, while the user-space service is $\sim$1500 lines of Go. Source code is available under an open source license at https://github.com/asch/dis. We are developing a new version targeting the KVM/QEMU hypervisor, and are working with the Ceph community to contribute it to the Ceph project as it is completed.

The prototype implements a simplified version of the full LSVD described above. Snapshot and clone mechanisms have not yet been implemented. We re-use the write-back cache implementation for the read cache, with static partitioning and FIFO replacement for both. Data blocks are not copied across the kernel/user interface, but instead pass through the SSD: the userspace write path reads outgoing data from the writeback cache, while the read path stores data in the read cache before replying to the kernel. Although simplifying the implementation, this results in significant overhead (§4.7).

The three maps are implemented as *extent maps*: ordered search trees of (start,length) pairs. The prototype uses red-black trees for its maps, at a memory cost of 40 bytes per entry; the new version uses an in-memory B+-tree averaging 24 bytes/entry.

4 Evaluation

After describing the experimental setup (§4.1) we compare the in-cache performance of LSVD to an optimized SSD cache

client	backend	
	✓	Ceph Octopus 15.2.3
✓	✓	Ubuntu 20.04, kernel 5.0, 5.4
✓	✓	20 core / 40 thread (2x E5-2660 v2)
✓	✓	128 GB RAM
✓	✓	10Gbit ethernet
✓		800 GB Intel DC P3700 NVMe
	✓	8x 250 GB SATA SSD (config #1)
	✓	7x 10K RPM SAS HDD (config #2)

Table 1. Experimental Setup. Two Ceph backends were used, a 4-node 32-SSD cluster (config 1) and a 9-node 62-HDD one (config 2); all tests are performed on config 1 unless noted otherwise.

(bcache [30]), (§4.2) for both block-level and filesystem-based benchmarks. For block-level writes LSVD is up to 30% faster and outperforms bcache by up to 4x on sync heavy filesystem-based workloads.

LSVD writes data back aggressively, minimizing the amount of "dirty" data in cache, thereby both minimizing data loss in the case of catastrophic failure and simplifying VM migration. Section 4.3 compares sustained performance when data needs to be written back to storage, and Section 4.4 examines how long each system requires to clean the cache after write bursts. We find that LSVD is more than 11x faster than bcache+RBD. Section 4.5 examines the load imposed by LSVD volumes on the backend storage (a 25x advantage over bcache+RBD), and present detailed trace results to investigate the source of this improvement.

The remainder of this evaluation examines garbage collection overhead (§4.6) the execution overhead of our prototype (§4.7), and finally demonstrates the potential for geographic replication (§4.8) and cloud deployment without provider support (§4.9).

4.1 Experimental Setup

The experimental setup is shown in Table 1; the same client and server hardware is used for both configurations. We compare LSVD to the Ceph RADOS block device, RBD [27], coupled with Linux bcache [30] configured as a write-back cache. RBD connects directly to the Ceph storage pool, and uses triple replication; for LSVD we used Ceph RADOS Gateway (RGW), an S3-compatible object store, which we configured to use a 4,2 erasure code on the same Ceph storage pool[5].

The storage backend in configuration 1 uses consumer-grade SSDs, with a sustained random write speed of ∼10,000 IOPS per device. The client cache device is rated at 2.8 and 1.9 GB/s sequential read and write, respectively, and 460K/90K read/write IOPS.

Both LSVD and bcache were configured with 700 GiB 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 the ratio of live data to the total sum of object sizes dropped below that point. All virtual disks were 80 GiB unless specified otherwise, and were preconditioned to fill them with data before beginning each experiment. Garbage collection activity is reported for all experiments in which it was observed.

4.2 In-cache performance

In our first experiments we compare LSVD to bcache for in-cache operations, configuring a cache larger than the volume itself and pre-loading the cache before each test. We perform (a) micro-benchmarks using the `fio` [1] I/O tester and writing directly to the block device, and (b) filesystem-based tests using Filebench [29] over ext4, generating a more complex workload of reads, writes, and commit barriers.

Although the LSVD cache is an unoptimized prototype, we find that it is competitive with bcache: random writes are faster in most cases, and sync-heavy filesystem-based workloads are up to 4x faster. The LSVD advantage is due to its log-structured cache, generating faster sequential writes and eliminating the need to persist cached mapping metadata at each commit barrier.

4.2.1 Micro-benchmarks.

Random write and read tests were performed on 80 GB virtual disks, with a range of block sizes and queue depths (i.e. concurrency); each experiment was run for 120s. Random write performance is shown in Figure 6: LSVD is faster by 20 to 30% for small writes, only falling behind for 64 KB writes with a queue depth of 32. For 4 KB and 16 KB writes LSVD reaches ∼60K and ∼50K IOPS respectively, approaching the rated speed (90K IOPS) of the device. We believe the small-write performance advantage over bcache due to the sequential writes generated by the log-structured cache and lack of extra metadata writes, as the device used is moderately faster for sequential writes than random ones.

Random read performance is seen in Figure 7. The effect of our unoptimized read cache may be seen here: its speed is equivalent to bcache at lower queue depths, but falls behind by up to 30% at high queue depths.

Graphs of sequential performance are omitted for reasons of space. Sequential read performance was similar for LSVD and bcache in all cases, with LSVD ranging from 25% faster (16 KB depth 4) to 25% slower (64 KB depth 32).

4.2.2 File system benchmarks.

We use *Filebench* [29] to simulate realistic file system-based workloads, with a mixture of writes, reads, and commit barriers, and significant overwriting of data to trigger garbage collection. The Filebench workload models we use are *fileserver*, emulating a network file server, *oltp* (database), and *varmail*, a file create/delete test emulating a message transfer agent. Specific parameters used

[5] This is the optimal choice for each: LSVD makes use of the higher large-write throughput provided by erasure coding, while by default RBD cannot be configured on erasure-coded storage, due to their poor small-write performance.

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

Table 2. Filebench workload parameters; omitted values are not applicable to the specific workload.

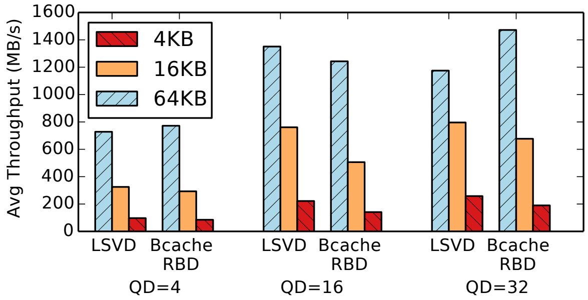


Figure 6. Random write performance: 80 GiB volume, large cache.

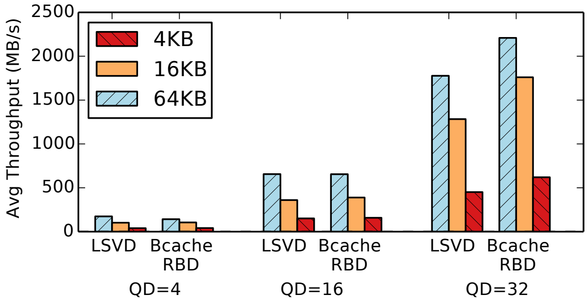


Figure 7. LSVD vs. RBD+bcache: 80 GiB volume, random read, large cache (100% cache hits).

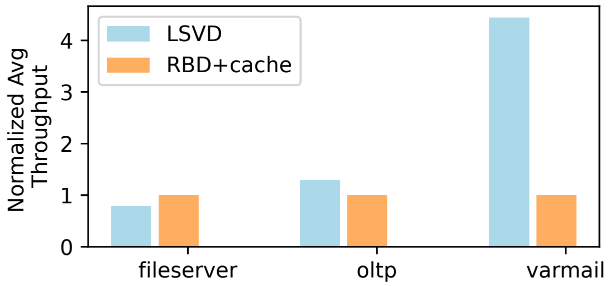


Figure 8. Filebench normalized throughput, LSVD vs RBD+bcache. LSVD performance is similar to RBD for *fileserver* (0.8x) and *oltp* (1.25x), but 4x higher for *varmail*.

workload	between syncs		mean
	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 block-level behavior on ext4: write size, commit barrier distance (measured in both write operations and MiB written).

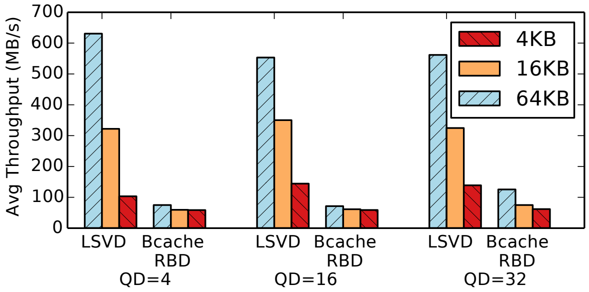


Figure 9. Random writes, small (5 GB) cache. Test duration: 120s.

for these models are given in Table 2. A fresh `ext4` file system was created for each test run, with default `mkfs` and mount options, and tests ran for 300 seconds after initialization phases were complete.

LSVD outperforms bcache+RBD for two of three workloads: varmail (by 4x), and oltp (by 1.25x). This is due in part to the efficiency of handling commit barriers in LSVD, which incur no additional operations due to the log structure. In contrast, bcache caches its B-tree-based map in memory, and only writes it out when a commit barrier is received. Statistics from a block-level trace in Table 3 show that oltp and varmail benchmarks are *sync-heavy*, with little data and few write operations between successive commit barriers.

While bcache pauses writeback under load, writing back only after test completion, LSVD not only aggressively writes data to the backend but performs garbage collection, which is triggered during these tests. We measure write amplification factor, i.e. the ratio of total backend bytes written to client-written bytes; the observed values are 1.046 for fileserver, 1.22 for varmail, and 1.75 for oltp. In other words, while writing not only all client data but an additional 22% or 75%, LSVD is still able to out-perform bcache with no writeback, by factors of 4x and 1.25x respectively.

4.3 Sustained performance

LSVD's aggressive writeback not only reduces the chance of data loss due to failure, but is important for virtual machine

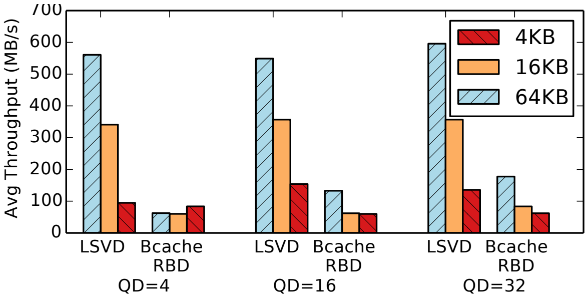


Figure 10. Sequential writes, small(5 GB) cache. Test duration: 120s.

migration: all cached data must be written back before a VM may be migrated, potentially resulting in significant delays. We measure writeback speed, showing improvements of 2x to 8x over bcache+RBD.

Write tests were performed with a 5 GB SSD cache, forcing writeback for most of the 120s duration of the test; random and sequential results are seen in Figures 9 and 10. LSVD writeback to the remote store is nearly as fast as a medium-performance local SSD, reaching speeds of over 600 MB/s. This is 2x to 8x faster than bcache+RBD; bcache provides little or no performance advantage here, with uncached RBD achieving nearly the same performance. RBD performance improves modestly with sequential operations, while LSVD is largely insensitive to the access pattern.

4.4 Write-back behavior

LSVD's writeback performance allows it to rapidly synchronize the cache and backend; we explore this behavior further, contrasting it to bcache+RBD. Using the HDD-based configuration 2, we perform 20 GB of random writes on an 80 GB volume, measuring client throughput and writeback speed, and waiting until the remote image is synchronized with the cache.

Results may be seen in Figure 11. LSVD completed the client writes in 77 seconds (dashed blue line); writeback (solid blue line) ran at full speed during this time, and completed at 120 seconds. In contrast, bcache disables writeback under heavy load[6], performing none during the 120 seconds needed for the client to write 20 GB (dashed red); writeback (solid red) begins after client writes complete, and continues until after the 1500 second mark. During these 25 minutes the backend image was not consistent, and a file system on the volume could be corrupted by SSD failure.

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 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`.

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

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 test random 16 KB write performance (queue depth 32) on multiple virtual disks in parallel, increasing the number of disks from 1 to 32. Tests were run in configuration 2 (9 servers, 62 10K RPM HDDs), measuring mean disk utilization (i.e. fraction of time busy, from `/proc/diskstats`) averaged across all disks in the storage backend.

Results are seen in Figure 12. LSVD reaches 47,000 IOPS with 16 virtual disks, and 50,000 IOPS with 32; at this point backend disks are 10% busy and throughput is limited by the single client machine and its single SSD. In contrast, with RBD the backend load grows quickly, reaching 70% with 32 virtual disks and ~13,000 IOPS; at this point the single client machine is consuming the entire I/O bandwidth of a 9-server 62-disk storage pool.

In this experiment, LSVD provides a 25x advantage in efficiency over RBD for small writes, a particularly difficult workload for RBD and similar systems. To investigate further we collect block traces of disk I/O at each device in the storage pool. In Figure 13 we see collected client and server-side statistics for (a) write operations and (b) total bytes, with server values summed over all disks in the storage pool. RBD writes suffer an amplification of 6× for both operations and bytes: one data write and one metadata/write-ahead log write at each of three replicas. LSVD, in contrast, has an I/O amplification of 0.25, generating one backend write for every 4 client writes.

This may be seen in more detail in histograms of backend write sizes: Figure 14 shows bytes written vs write operation size[7]. Almost all RBD writes are roughly 16 KB; if examined in more detail we see that half are exactly 16 KB, while the other

[6] There does not appear to be a setting to disable this behavior.

[7] Sequential writes were merged for this analysis.

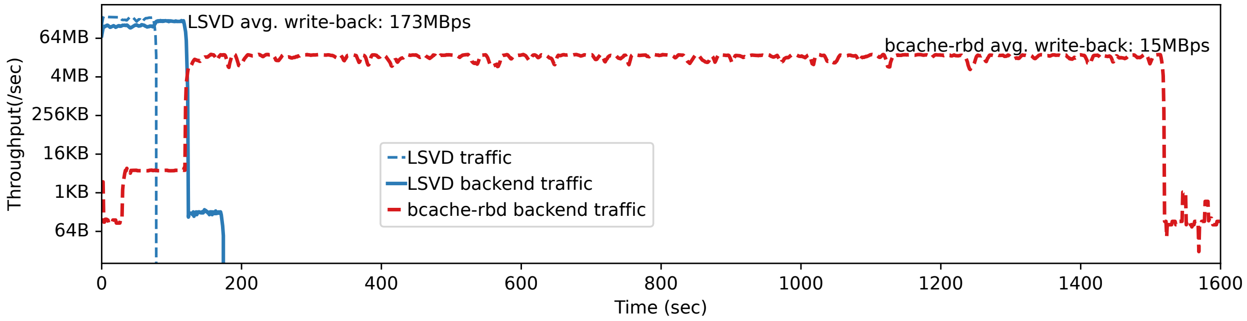


Figure 11. Writeback behavior. Client side performs 20 GB of 4 KB random writes (dashed lines); solid lines indicate writeback traffic. Note log scale—LSVD writeback is 11.5x faster than bcache+RBD.

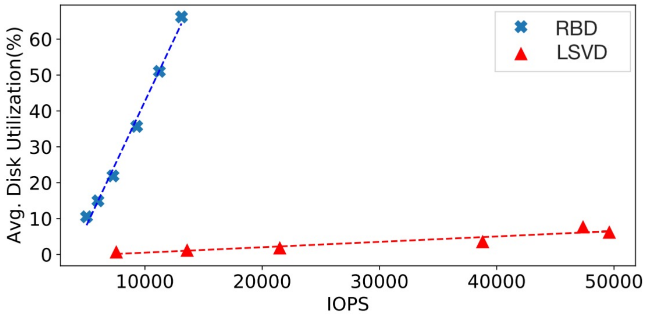
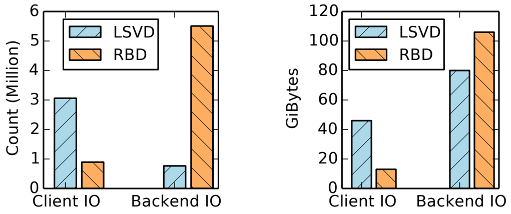


Figure 12. Write efficiency: IOPS vs backend disk busy time, random 16 KB write tests running simultaneously on 1, 2, 4, 8, 16 and 32 virtual disks. X axis: total virtual disk IOPS, Y axis: mean backend device utilization.



(a) I/O amplification

(b) Byte amplification

Figure 13. I/O and byte amplification: 16 KiB random write load test.

half are 20 or 24 KB, no doubt representing write-ahead log entries for the two-phase commit used for atomic updates of small ranges. In contrast LSVD writes cluster around 1 MB, i.e. the data and parity chunk size when splitting a 4 MB RADOS object with a 4,2 code. A significant number of small metadata writes are needed to create a 4 MB object; in all Ceph issues 64 writes across 3 disks to create a 4 MB object, which in turn (neglecting a small amount of metadata) holds 256 16 KB client writes.

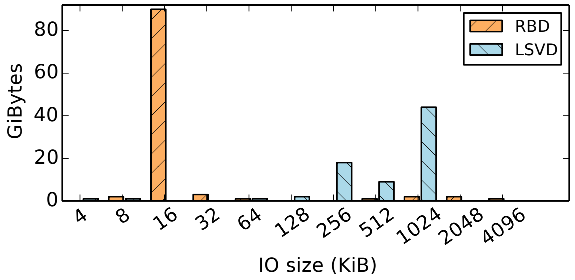


Figure 14. Bytes written vs I/O size for 16 KB random write test. X axis labels indicate lower bounds of histogram bins.

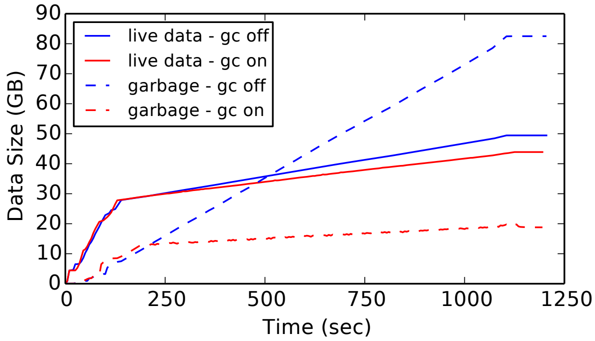


Figure 15. Garbage collection performance: varmail 1000s, 5 GB cache. Solid lines show valid data, dashed lines invalid. With garbage collection (red) the workload runs slightly slower, but invalid data is limited to 30% of total.

Further analysis shows that the RBD-generated writes cluster in streams, and with re-ordering only 18% of writes require seeks of more than 128 KB; 18% of 6×13,000 IOPS gives 226 IOPS per drive, on 10K RPM drives with rated speeds of ~370 write IOPS. At 50,000 16 KiB writes per second and 1.5x expansion for erasure coding, LSVD is writing 20 MB/s of data to each drive, or 10% of its rated transfer speed, plus roughly 32 IOPS per drive in small writes.

	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.

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 this we simulate the LSVD write batching and garbage collection algorithms on traces from the CloudPhysics corpus [32]. 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[8]. *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" 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.

k/u		Read miss operation	μS
1	k	map lookup	3
2	k	context switch	50
3	k	return to user space	22
4	u	golang overhead	34
5	u	S3 range request	5920
6	u	write to NVMe	136
7	k	return to kernel	27
8	k	read from NVMe	*

k/u		Write breakdown operation	μS
1	k	write to NVMe	64
2	k	map update	3
3	k	context switch	50
4	k	return to userspace	20
5	u	golang overhead	63
6	u	read from NVMe	110
7	k	return to kernel	27

Table 6. Single read and write fine-grained measurements; k=kernel mode, u=userspace. *see text.

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. The workload parameters from Table 2 are used, with a post-initialization test duration of 1000 seconds, and a 5 GB write-back cache.

In Figure 15 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. With it enabled, cleaning begins when valid data drops to 70%; this ratio is maintained for the rest of the experiment, with an overall write amplification factor 1.176.

The performance impact of garbage collection may also be seen: after the execution phase is complete, slightly more valid data has been written in the GC-disabled run (blue) than in the GC-enabled one (red). Further experiments show a slowdown of ~8% for *fileserver*, 10% for *varmail* and 2% for *oltp*.

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

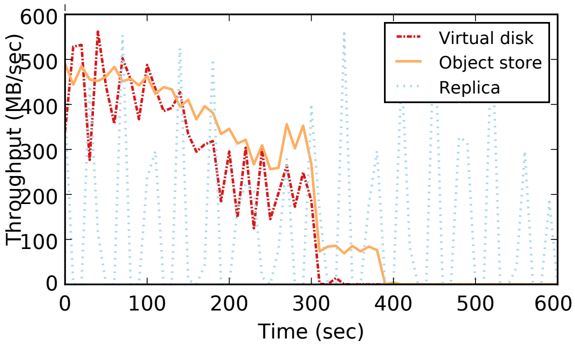


Figure 16. Data transfer during asynchronous replication.

4.7 Overhead analysis

To explore the performance overhead in our prototype we instrument both the kernel and userspace components, logging timestamp counters in memory to minimize the effect of measurement. Logs were collected for periodic but isolated reads and writes; the analysis may be seen in Table 6. Note that data is passed in the SSD, rather than across the `ioctl` interface; thus e.g. the userspace write logic reads data from the SSD. We were unable to measure step 8 in the read path, as the LSVD kernel device does not receive the completion event.

The most time-consuming operation is an S3 GET operation in response to a read miss, taking nearly 6 ms in our tests. We note that context switching—i.e. the delay between calling `wake_up` from an event handler until the corresponding `sleep_on` returns—is significantly more expensive than entering or leaving the kernel. The entries labeled "golang overhead" (read 4, write 5) are between points separated by a channel send and receive; the time appears to be spent on thread switching and buffer allocation / deallocation. The use of the SSD to pass data between kernel and user space adds overhead, reducing maximum throughput due to additional SSD operations, but does not significantly impact latency: steps 3 and later in the write path are in the background, and the read path is dominated by the S3 request.

4.8 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[9]. We evaluate this on a pair of object stores: directly writing to Ceph S3 on a 5-host NVMe-based cluster, replicating to a second Ceph S3 instance on the configuration 2 backend from Table 1. 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.

Data transfer during the experiment is shown in Figure 16. 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.

4.9 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.

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). Yet while a 50,000 provisioned IOPS EBS volume would cost over $3000 per month (current on-demand price), the local NVMe and remote S3 needed by LSVD would in contrast cost only a few dollars per month.

5 Related work

In surveying related work, we first note that many of the most widely-used virtual disk implementations are proprietary: Amazon EBS [26], Google Persistent Disk [7, 11], and Azure Managed Disk [25]. No details have been disclosed describing the operation of these systems, and we are thus unable to contrast them to LSVD.

The most widely-used publicly-disclosed virtual disk appears to be Ceph RADOS Block Device [27], or RBD. RBD splits a virtual disk image into smaller named objects distributed across the storage pool using consistent hashing. Objects are mutable and triple-replicated, resulting in the write amplification described in previous sections.

In the literature, the virtual disk systems most resembling LSVD are Salus [33], Blizzard [17]), and Ursa [16]: each uses out-of-place writes in a log-structured format, using sequencing mechanisms to provide prefix consistency under worst-case failure scenarios. None of these include client-side caching:

[9] This description ignores several subtleties, omitted for reasons of space.

Salus and Ursa use standard block interfaces between the client and remote systems, while Blizzard and LSVD are able to perform much larger writes atomically using Flat Datacenter Storage and S3 object storage respectively.

In Salus [33], clients send writes to a lead server, which (a) logs them to an HDFS write-ahead log, and (b) accumulates them in memory into large batches, then writes batches to large *checkpoint* files. Salus must maintain a distributed, consistent map from block addresses to locations in these checkpoint files, involving significant complexity. And in step (a) above it must persist individual writes (or small batches) in replicated storage; this is a limitation of any purely server-side log-structured virtual disk, and limits the degree to which backend I/O rate may be reduced. Ursa [16] is a replicated remote disk with a primary SSD-based replica and two secondary HDD-based ones; client writes update the primary, and logs are shipped to the replicas, which are lazily replicated. Blizzard [17] is an uncached client-side disk over Flat Datacenter Storage [19], providing prefix consistency in its *fast ack* mode. Although Blizzard is log-structured, it cannot perform batching, and thus uses a very large block size (64 or 128 KB), incurring read/modify/write overhead for small writes.

The other area of related work focuses on client-side caching of remote disk images. Some of these caches (e.g. Hystor [4] and bcache [21]) were originally designed as SSD caches for local hard drives; such caches may not preserve consistency in the face of independent failures of the client and server, as they were designed for a case where client and server are the same machine. At the other end of the spectrum, Mercury [2] uses a write-through policy, guaranteeing full consistency but offering no performance benefit for write-heavy workloads.

Koller et al [13] propose *journaled writeback*, which preserves write ordering of cached data[10], but cannot guarantee prefix consistency when coalescing data, as block device writes cannot be batched. Qin et al [22] use write barriers to define epochs, allowing multiple epochs of writes to be cache locally, but ensuring that an epoch is written completely to the backend before any following epoch is destaged. In other words, since the interface between their cache and backend storage is unable to write atomic batches or preserve ordering, they insert runtime delays to ensure consistency, with a resulting loss in performance.

Note that these caches do little to reduce the load imposed on remote storage systems by small write workloads. Although these write-back caches allow bursts of writes to spread over longer time periods, they do little to reduce the total write burden on the backend. The only actual reduction in writes they provide is due to coalescing, and to maintain consistency, writes may only be coalesced within a single epoch.

In contrast LSVD is able to coalesce writes within a single batch, even if separated by multiple commit barriers, and uses an object store interface to send the entire batch to the backend, where it can be efficiently stored using small numbers of large write operations. The log-structured write-back cache is conceptually simple, but does not appear to have been described previously in the literature; in particular, Koller et al. [13] do not log metadata, but rather propose using NVRAM for this purpose. We speculate that this approach may have been dismissed in the past due to the high space amplification (2x) for small writes; as the cost of high-speed SSDs drops, trading space for performance becomes more attractive. The split between write-back and read caches is similar to that found in Hystor [4], for similar motivations, and the read cache itself is of standard design.

6 Concluding Remarks

LSVD combines a log-structured local cache with a log-structured remote store, preserving write ordering in each and using object store semantics to perform large batched writes to the backend. Our prototype has demonstrated that this strategy can: 1) achieve massive performance gains over existing remote virtual disks (16x), with dramatically reduced demand on the storage service (25x), 2) preserve the reliability & functionality advantages (e.g, migration, snapshots) of today's virtual disks, and 3) naturally support new functionality such as user deployment and asynchronous replication over geographic distances. 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.

6.1 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 [17].

Our experience appears to validate the Blizzard and LSVD approach. A virtual disk with out-of-place writes requires complex, strongly-consistent, distributed translation maps, with rapid updates. In contrast an in-memory client-side map will be available any time it is needed by the client.

In-memory Map. Rather than using an on-SSD map with in-memory caching, the LSVD prototype relies on an extent map, mapping ranges rather than single blocks, which is maintained exclusively in memory, with updates journaled to SSD and the remote log. Its memory requirements are modest: about 24 bytes per entry, or 0.5 GB for a 20M-entry

[10] While journaled write-back logs data to cache in the order it is written, unlike LSVD it does not write metadata to the log.

map. Our simulations show that modest changes to the garbage collection algorithm are able to bound the size of the map to well under this size for measured workloads on virtual disks of up to 1 TB.

Log-structured Writeback Cache. This was introduced to preserve write ordering for consistency of the remote image; however it provides significant performance gains as well, avoiding the need for extra write operations on commit barriers. For sync-heavy workloads such as Filebench "varmail" (§4.2.2) this results in up to 4x higher performance than that achieved by bcache.

Atomic Batching of Writes. As argued by Koller [13], 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, which requires stalling until all outstanding writes have completed. As a result, prefix-consistent caches over block back-ends [22] 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 concurrent writes.

6.2 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. When the LSVD prototype is used on a "bare-metal" machine, the kernel driver receives I/O requests directly from the in-kernel file system. But our ultimate target is not bare metal, but the KVM/QEMU hypervisor, allowing LSVD disks to be accessed through the standard *virtio* [24] block interface. In this case the I/O stack runs in the user-space QEMU process, and our prototype incurs additional overhead on each I/O. The open source implementation in progress is a userspace QEMU plugin, eliminating boundary crossings and simplifying development.

Lack of Ecosystem. There are a wide range of tools and platforms for working with virtual disks for virtual machines; however we implemented a virtual disk for physical machines. We were thus unable to use much of the development, testing, and evaluation infrastructure for virtual disks which fit into the KVM/QEMU framework, and were hampered by a lack of user and developer communities.

6.3 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 backend used by LSVD offers different tradeoffs than seen in prior log-structured systems, with variable-sized units and elastic capacity. 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?

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 [9]. We have prototyped simple implicit and explicit strategies for reducing fragmentation 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. A single host may run many virtual machines, each with disks cloned from the same image, using the same objects on backend storage. We are looking at mechanisms to cache and share this data across multiple virtual disks, eliminating network I/O as as successive VMs access shared blocks. Block sharing should be possible within a virtual disk, as well, by pre-processing a disk image to eliminate duplicate blocks, pointing multiple LBA extents to the same back-end object data, similar to VMAR's de-duplication translation maps [28] but simpler in implementation.

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.

Asynchronous Replication. The use of an immutable object stream enables asynchronous replication; as noted in §4.8 this requires some synchronization between the replication and garbage collection processes. Further work is needed to allow write coalescing before replication, for reduced bandwidth, while maintaining consistency of the replica.

Acknowledgments

Special thanks go to the partners of the Mass Open Cloud (MOC) Alliance, and Red Hat and Two Sigma in particular for their support, as well as to the entire MOC team that exposed us to the real world problems that inspired this work. We would also like to thank the EuroSys reviewers for their insightful comments and questions, which greatly improved the paper. This work was supported in part by the National Science Foundation awards 1910327 and 1931714.

References

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

[2] 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.

[3] Ceph project. 2022. Ceph documentation: Ceph Block Device. Available from docs.ceph.com. Last accessed March 2022.

[4] Feng Chen, David A. Koufaty, and Xiaodong Zhang. 2011. Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems. In *Proceedings of the International Conference on Supercomputing (ICS '11)*. ACM, New York, NY, USA, 22–32.

[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.

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

[7] 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.

[8] Google. 2019. Persistent Disk. https://cloud.google.com/persistent-disk/.

[9] 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.

[10] Mohammad Hossein Hajkazemi, Mania Abdi, Mansour Shafaei, and Peter Desnoyers. 2018. FSTL: A Framework to Design and Explore Shingled Magnetic Recording Translation Layers. In *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 40–52. https://doi.org/10.1109/MASCOTS.2018.00012

[11] Dean Hildebrand and Denis Serenyi. 2020. STO300: A peek behind the VM at the Google Storage infrastructure. available from https://cloud.google.com/blog.

[12] Dave Hitz, James Lau, and Michael Malcolm. 1994. File system design for an NFS file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*. USENIX Association, San Francisco, California, 19–19.

[13] 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.

[14] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. 2006. The Linux Implementation of a Log-structured File System. *SIGOPS Operating Systems Review* 40, 3 (July 2006), 102–107.

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] Kent Overstreet. 2017. bcache. Available from https://bcache.evilpiepirate.org.

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

[23] 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.

[24] Rusty Russell. 2008. virtio: towards a de-facto standard for virtual I/O devices. *SIGOPS Operating Systems Review* 42, 5 (2008), 95–103.

[25] Corey Sanders. 2017. Announcing general availability of Managed Disks and larger Scale Sets. Available from azure.microsoft.com/en-us/blog.

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

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

[28] 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.

[29] Spencer Shepler, Eric Kustarz, and Andrew Wilson. 2008. The New and Improved FileBench File System Benchmarking Framework. In *USENIX File and Storage Technology (FAST '08) Work in Progress session*. San Jose, California.

[30] William Stearns and Kent Overstreet. 2010. Bcache: Caching beyond just RAM. *LWN.net* (July 2010).

[31] Doug Terry. 2011. *Replicated Data Consistency Explained Through Baseball*. Technical Report MSR-TR-2011-137. Microsoft Research.

[32] 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.

[33] 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.

A Artifact Appendix

A.1 Abstract

This artifact provides the source code of the LSVD prototype, as well as scripts to run the main experiments in the paper which compare LSVD to RBD and bcache. LSVD is a log-structured virtual disk, using a log-structured local SSD cache and a log-structured remote object-based store. The prototype provides a Linux block device which may be formatted and used as a normal disk.

This artifact does not include scripts for the the writeback behavior (Section 4.4), backend load (Section 4.5) and AWS deployability (Section 4.9) experiments; these should be straightforward to implement. The simulations in Section 4.6 (Garbage Collection) and replication in Section 4.8 (Async Replication) require additional components which are not provided.

A.2 Description & Requirements

A.2.1 How to access. Source code and scripts may be accessed at https://github.com/asch/dis.

A.2.2 Hardware dependencies. A local (preferably NVMe) SSD is needed on the test machine.

A.2.3 Software dependencies. At present the kernel component must be compiled with Linux kernel 5.0.0, and has not been ported to other versions. Kernel headers must be installed for kernel module compilation.

The primary experiments require an S3 object store endpoint; the provided code has been tested against RADOS Gateway (RGW) from the Ceph 15.2.16 Octopus distribution. Additional experiments require direct client access to Ceph RBD images and a RADOS storage pool.

A.2.4 Benchmarks. You will need the *fio* and *Filebench* benchmark programs: Tests in the paper were performed with fio version 3.16 and Filebench 1.5-alpha3.

A.3 Set-up

1. compile kernel headers: cd to `kernel` and run `make`. You may need to edit the kernel header include path in `Makefile`.
2. compile userspace code: in `userspace`, run `go build`.
3. edit benchmark configuration: in `benchmarks/helpers/dis_on.sh` edit AWS access and secret keys. In `benchmarks/config.toml` edit the S3 endpoints (`DIS_BACKEND_OBJECT_S3_REMOTE`), Ceph RADOS pools (if desired) and list of experiments. (see README.md for more details)

A.4 Evaluation workflow

A.4.1 Major Claims.

- *(C1): LSVD over S3 achieves comparable or better in-cache random write performance as* bcache *over RBD when using the same backend store, shown by experiment (E1) in section 4.2.1, Figure 6.*
- *(C2): LSVD over S3 achieves similar in-cache random read performance as* bcache *over RBD when using the same backend store, shown by experiment (E2) in section 4.2.1, Figure 7.*
- *(C3): LSVD over S3 achieves better Filebench throughput for the "oltp" and "varmail" workloads than* bcache *over RBD when using the same backend store, shown by experiment (E3) in section 4.2.2, Figure 8.*
- *(C4): for small cache sizes, LSVD over S3 achieves significantly better random-write throughput than* bcache *over RBD , shown by experiment (E4) in section 4.3, Figures 9 and 10.*

A.4.2 Experiments. Experiments E1 through E4 may be conducted in a single run of the `benchmarks/run.py` script. In ‘fio.toml‘ set "rw" to "randwrite" and "randread", in ‘config.fio‘ set "enabled" to "bcache_rbd_replicated" and "dis_rgw_ec", and "benchmarks" to "fio".

Runtime will be roughly 30 hours using default parameters: 9 iterations (config.toml, "iterations"), two cache sizes (config.toml, "cache_size_M" parameters), two test configurations (config.toml, "enabled" = "bcache_rbd_replicated", "dis_rgw_ec"), random read / random write (fio.toml, "rw"), three block sizes and four queue depths (fio.toml, "bs" and "iodepth"), and a runtime of 120s (fio.toml).

Runtime may be decreased by reducing the number of iterations or test cases.

Run statistics will be saved to `fb.csv` and `fio.csv`; additional information will go to standard output.

Human time to configure LSVD and the tests should be fairly short, e.g. 1 hour. Configuring a Ceph backend for testing will take much longer.