FreEBS - A Distributed Virtual Block Device

Igor Canadi Rebecca Lam James Paton
University of Wisconsin-Madison
{canadi,rjlam,paton}@cs.wisc.edu

Abstract

The primary goal of this project was to create a free, open-source implementation of Amazon EBS. We present FreEBS, a distributed virtual block device that uses chained replication and log-based virtual disks with the goals of availability, durability, and reliability in mind. Our backing storage also provides the framework for snapshotting, one of the main features of Amazon EBS that makes it more robust and resilient to failure.

We compare our system's performance to the performance of native I/O on a local disk. We find that, for random and mixed random-sequential workloads, FreEBS performs with much higher throughput than local, while for purely sequential workloads it performs much more poorly.

Finally, we argue that FreEBS points the way toward a viable commoditization of replicated block storage, exemplifying a novel model as compared to DRBD or SAN. This model has the potential to lower the cost and increase performance.

1. Introduction

FreEBS is a distributed virtual block device that utilizes replication and log-based storage to provide reliability and durability. The major goal of FreEBS is to create a free and open version of Amazon Elastic Block Store (EBS)[1], a virtual mountable block device used by EC2 instances. Many implementation details about EBS have not been publicized, but many speculate that EBS is based on DRBD®[2].

There are two main features that are publicly known about Amazon EBS — replication and snapshots. EBS mirrors volume data across different servers within the same Availability Zone. This provides some increased reliability over a single disk, but EBS further adds durability using snapshots. Snapshots are customizable incremental backups stored on Amazon S3. Each snapshot only saves the blocks that have been modified since the last snapshot. For each snapshot there is a table of contents that maps each EBS volume block to the valid corresponding block in S3. When a snapshot is deleted, only blocks that are not pointed to by any other snapshot are removed from S3. Because snapshots only keep track of modified blocks, they take less time to perform than a full-volume backup, and require less space to store. The frequency at which they are performed determines the durability of the volume in the case that the EBS volume fails. [1]

Due to the lack of concrete, detailed information about Amazon EBS, we draw many of our ideas about how EBS is implemented from Distributed Replicated Block Device® (DRBD)[2], an open source virtual RAID-1 block device. We discuss DRBD® in more detail in Section 2.

Given what we know of EBS and DRBD, FreEBS seeks to provide the following:

Availability and Reliability The system should continue to operate even when a certain number of replicas fail;

Durability All written data should exist on the volume even when a certain number of replicas fail;

Consistency The system should serve correct versions of the data at all times; fail if not possible to satisfy consistency;

Snapshots The system should support snapshots and incremental backups.

We address these goals by using replica servers that coordinate using a userspace process on each machine. These replicas propagate writes and perform synchronization. They also interact with a log-structured virtual disk that provides checkpointing, mechanism for recovery, and the framework for snapshotting.

The rest of the paper is organized as follows. In Section 2 we discuss related work, and in Section 3 we explain the overall design of the system. We go into more detail about our system in Section 4. Finally, we evaluate our design and draw appropriate conclusions in Section 5 and Section 7, respectively. Section 6 describes future work.

2. Related Work

We drew many of our ideas about EBS on past work, especially DRBD®. Here we discuss related network and cloud storage systems and compare our work to them.

2.1 DRBD

DRBD® is a virtual RAID-1 block storage device consisting of two shared-nothing nodes that form a highly available *cluster*. The DRBD® interface is implemented as a kernel module that can be mounted like a normal block device. Since it is an open-source project, many aspects of its implementation are available; however, in this section we focus on the details of replication and synchronization.

2.1.1 Replication

DRBD® uses a RAID-1 method of replication, meaning there are two nodes in a basic DRBD® cluster — one *primary* and one *secondary* — that each contain typical Linux kernel components[2, 9]. The primary node services all reads and writes, and the secondary node fully mirrors the primary node by propagation of writes across the network. If the primary node fails, then service migrates to the secondary node. There are two main modes for replication — fully synchronous and asynchronous. The former means that the primary node reports a completed write only after the write has been committed to both nodes in the cluster. The latter means that each node reports a successful write as soon as the data is written to its local disk.

FreEBS uses a technique similar to the asynchronous mode of replication; we utilize a primary replica that services reads and propagates write requests. However, FreEBS waits for a majority of replicas to respond with a successful write completion. Our system also offers much more flexibility in terms of the number of configurable replicas. For instance, DRBD® must use stacked DRBD volumes in order to create more than two replicas, which can get cumbersome as we add more replicas.

2.1.2 Synchronization

DRBD® offers variable-rate and fixed-rate synchronization [9]. For the first method, DRBD® selects a synchronization rate based on the network bandwidth. For the fixed-rate case, synchronization is performed periodically at some constant time interval. Synchronization can be made more efficient by using checksums to identify blocks that have changed since the last synchronization. This eliminates the need to synchronize blocks that were overwritten with identical contents. Changes are tracked using the activity log (AL) that records *hot extents*, the blocks that have been modified between synchronization points. The activity log uses a quick-sync bitmap to keep track of modified blocks.

FreEBS uses a fixed-rate synchronization scheme that allows nodes to request all writes since a particular version. This is done by backward traversal through our log-based backing file until the requested version is found, eliminating the need to have a separate activity log.

2.2 Cloud Storage

Rackspace Cloud Block Storage (CBS) and HP Cloud Block Storage provide persistent storage for their cloud instances[3, 4]. Both also support snapshots. However, unlike EBS snapshots, Rackspace CBS snapshots contain the full directory structure of a volume. Both HP CBS and Rackspace CBS are built on the OpenStack storage platform. Another commercially available cloud block storage is Zadara™Virtual Private Storage Arrays[5], which is a cloud-based NAS service that uses iSCSI and NFS protocols.

2.3 Network Block Level Storage Systems

There is a fair amount of work on the subject of network block level storage. For instance, in 1996 Lee and Thekkath published a paper on Petal, a distributed virtual disk system[10]. In this system, a cluster of servers manage a shared pool of physical disks and present the client with a virtual disk. Virtual disks are globally accessible to all Petal clients, not one-to-one like FreEBS. Petal handles failure by chain-replicating blocks with one neighboring server. If one disk goes down, then the server with the replica block will serve the request.

In [7] Dabek et al. discuss Cooperative File System (CFS), a peer-to-peer read-only block storage system with robustness and scalability in mind. CFS uses a

distributed hash table to map blocks to a CFS server and utilizes block-level replication to increase reliability. Block replicas exist on the subsequent k servers based on the corresponding hash value for that block; thus, the next server in the hash takes up responsibility for serving that block.

Conversely, Quinlan and Dorward present Venti, a network storage system that uses a write-once only policy for archival data[11]. Venti is a userspace daemon that performs block level reads and permanent writes. Venti provides reliability and recovery by relying on RAID-5 disk arrays rather than mirroring.

The main difference between these works and FreEBS is that they are built for a shared virtual disk system and thus consider scalability and load balancing. FreEBS volumes, by contrast, only support being attached to one machine at a time. FreEBS also utilizes a log-based backing file to keep track of changes, whereas the above systems do not.

3. Design

FreEBS is a virtual block device backed by replication and log-structured virtual disks for high reliability and availability. For all intents and purposes, the FreEBS storage system appears to the kernel as a normal block device and thus reads and writes are issued to FreEBS without knowledge of its internals. The architecture of FreEBS is depicted in Figure 1. The most important component in the system is the *driver*, which is a kernel module that implements kernel block device interface

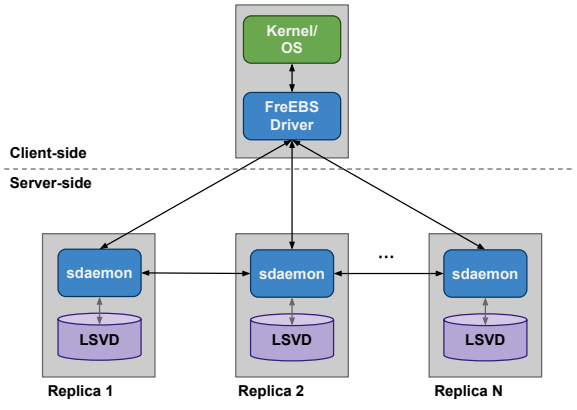


Figure 1: The system architecture for FreEBS. A driver on the client machine communicates with processes on the replicas on server machines to read and write data.

and exports a block device that can be mounted on a system. The kernel forwards all device's read and write requests to the driver. The driver forwards the requests appropriately to the *replicas*. Our current configuration uses three replicas, but our design is flexible enough that we can arbitrarily add more replicas. We discuss replication in more detail in Section 4.2.

Replicas are essentially server machines that communicate with each other and the driver across the network using a custom TCP protocol. Each replica runs a userspace process called `sdaemon` that is in charge of propagating write requests and synchronizing replicas. It is also responsible for interfacing with its private copy of the underlying storage that actually holds the device data, *Log Structured Virtual Disk* (LSVD). LSVD is backed by a file on the server file system that uses a dynamically growing file format. It provides checkpointing and versioning for high data integrity. LSVD borrows ideas from Log Structured File System [12]. A more in-depth discussion about LSVD is in Section 4.3.

4. Implementation

Based on our design and ideas, we implemented FreEBS prototype. We have successfully implemented the core of the system. We have not, however, implemented snapshotting or recovery from failures.

In the following sections we present implementation details of our prototype.

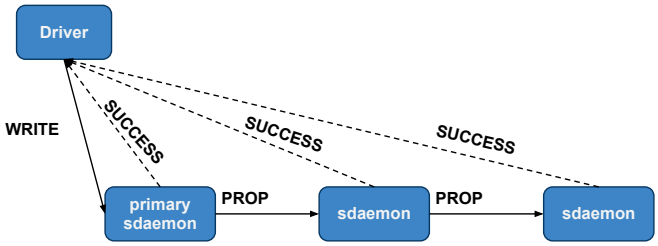
4.1 Device Operations

There are two operations that are issued by the driver — read and write. In order to perform these commands, the driver keeps track of the most recent version for each replica in a list, essentially acting as the master for the system. This replica versioning structure is updated on a write and used to choose which replica to service a read request. In order to accurately populate the version list, the driver must perform a handshaking protocol on initial connection to a replica, which reports its current LSVD version. We explain versioning in more detail in Section 4.3. For now, it suffices to note that versions are at the granularity of LSVD volumes rather than blocks. This means that the entire disk has the same version number.

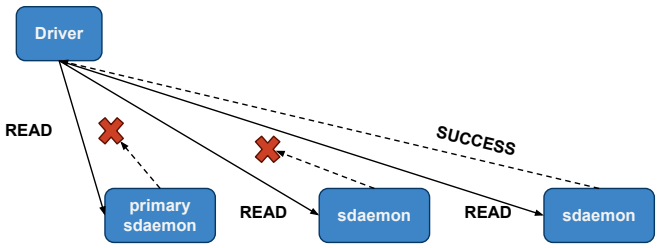
Figure 2a shows the message flow on a write request. Notice that, in a similar to fashion to Google File System [8], we utilize chained replication, where the closer

the replica is to the head of the chain the more more up-to-date it is. The goal of this chained scheme is to fully utilize each machine's network bandwidth. Thus, when the driver sends a write request, `sdaemon` issues the write to its local LSVD copy and then propagates the modified blocks to the next replica in the chain. Propagation is done in a streaming fashion — as soon as a chunk of data is received, it is sent to the next replica. After writing to its own LSVD, each replica sends back a response to the driver with a status and its current version number. The driver then updates its replica version list with the value in the response. If a majority of replicas responds to the driver with a SUCCESS message, then the driver reports a success back to the kernel. Otherwise, it will report a failure.

In contrast, a read only needs one replica to respond with a success. In Figure 2b we observe that when the driver issues a read request it issues the request to the replica with the most recent version. Typically this is the primary replica. If the replica responds with a SUCCESS, then the driver serves the request back to the kernel. Otherwise, it will issue the request to the next replica in the chain with the most recent version. If no replicas respond, then the driver reports a failure to the kernel.



(a) The message flow for a write operation. A quorum of SUCCESS responses must be received for a successful write.



(b) The message flow for a read operation. A read request is issued to the replica with the most recent version until a SUCCESS is received or if no replicas with the most recent version responds with SUCCESS.

Figure 2: Message flows for read and write operations

4.2 Synchronization and Failure

Replication is managed by `sdaemon` using write propagation and synchronization. Propagation was discussed in Section 4.1. Synchronization is performed whenever a replica joins the chain and also periodically to keep replicas near the tail from becoming increasingly out of date. Figure 3 shows how replication is performed. Replica C sends a SYNC request with the version number of its local LSVD copy to the previous replica in the chain, replica B. Replica B then responds with all the writes from C.version to B.version. Replica C then applies all the writes to its LSVD. After the synchronization operation, Replica C's LSVD version is equal to B.version. Similarly, replica B initiates the synchronization procedure with replica A.

To handle failure, a replica controller is used to inform the driver and neighboring replicas that a failure has occurred. This controller keeps track of the liveliness of each replica by recording the elapsed time since the last heartbeat message, which is sent every T seconds by each replica. After $4T$ seconds the replica is marked as failed. Upon failure, the controller informs the driver and the next replica server in the chain using update messages. Upon receipt of the update message, the successor replica connects to the replica ahead of the faulting server, which updates its next server information. The successor replica then sends a SYNC request, and then normal operation resumes. Meanwhile, the controller spins off a new `sdaemon` process that is appended to the end of the chain. The driver appropriately modifies its replica version list by deleting the faulted server and adding the new server and then connects to the new server to retrieve its current version.

In order to minimize the amount of synchronization that needs to be performed on startup, we use a copy of the LSVD of the tail replica on initialization. In Section 4.3 we discuss how we handle LSVD failure. Note that the replica controller and failure handling are currently not implemented in our prototype.

4.3 Log Structured Virtual Disk

When designing our file format to back virtual disk, we had few goals in mind. First, the file should be dynamically growing. If a disk is defined with D bytes free space and the user only uses C bytes for her content, the backing file size should use on the order of $O(C)$ bytes.

Figure 3: The message flow for a synchronization operation. Replica C sends a SYNC request to its predecessor, which responds with all the writes since C's version.

Second, we must support versioning. We define *version* as a monotonically increasing update sequence number. Each update increments the whole disk version by one. An *update* is a request originating from the driver that consists of one or more sequential sectors on the disk. Moreover, versions are required to enable synchronization; `sdaemon` must be aware of the current local disk version and must be able to send arbitrary version updates to other replicas for synchronization. The third goals for LSVD was consistency. If the local data has version V, it should consist of all updates with versions less than or equal to V and no updates with versions greater than V. Finally, the fourth goal was the ability to easily support snapshots and incremental backups.

To address all of these requirements, we developed Log Structured Virtual Disk (LSVD). When LSVD receives updates, it writes them sequentially on the backing file and increments the LSVD version by one. This append-on-write behavior, along with the LSVD cleanup procedure, satisfies our dynamically growing file goal. We also support versioning and consistency by recording all modifications and their corresponding versions. When designing the LSVD, we took special care to ensure that the backing file remains consistent and recoverable up to some recent version, even if the replica server fails at any arbitrary point in time. In other words, if the replica fails, it will always be able to recover to some version V and will include all updates up to V. Additionally, version V is not too far behind, which makes synchronization faster. We call this property *on-disk data consistency*. Finally, the design of LSVD enables snapshotting and incremental backups; however, this is not currently implemented. We discuss possible implementations of snapshotting later.

Figure 4 shows the layout of an LSVD file. The first few bytes define the superblock, which contains disk size, unique identifier, magic number and pointer to the current checkpoint. Following this there are two checkpointing placeholders. After these we store the updates. Each update consists of a data block and a commit block. The data block is comprised of one or more sectors of write data. The commit block verifies the integrity of the data block by storing its checksum. To guarantee consistency, we require atomic version writes; thus if a commit block is not written to disk or its checksum is invalid, we define the update as not committed and that update is skipped. We use checksums to guarantee on-disk data consistency, even if the disk or the kernel reorder our writes (for example writing commit blocks before data blocks).

Figure 4: The LSVD file structure.

When opening a disk after recovery, LSVD could rebuild the sector to offset map by reading all the updates from the beginning of time; however, this method is slow. To keep recovery time from growing linearly with time, we perform *checkpointing*. Every 60 seconds, we write the sector to offset map to disk into one of the checkpointing placeholders at the beginning of the file. In order to negate the impact of checkpoints on normal write or read operations, they are performed slowly. After a checkpoint is fully written to one of the placeholders, LSVD updates the superblock checkpoint pointer and calls `fsync` to flush the write to the physical disk. During recovery, the checkpoint data (sector to offset map) is loaded into memory. The checkpoint also stores the pointer `last_offset` to the offset in the file when we started writing the checkpoint. Any updates after `last_offset` may or may not be updated in the sector to offset map. Note, however, that some of the updates may exist in the checkpoint since we perform checkpointing slowly and allow concurrent updates. To rebuild the sector to offset map, we load in all the updates after `last_offset` and use them to update the map, similar to normal operation. By checkpointing periodically, recovery time is linear to the rate of updates rather than the total number of updates over time.

The problem with storing data in an append-on-write, log-structured format is data redundancy, or bloat. Even if a sector has been overwritten by newer data, LSVD still keeps its old versions around. This is both advantageous and disadvantageous. For instance,

we can use it to enable users to recover the disk state from an arbitrary point in time. However, the file may grow to an unmanageable size. We therefore implement garbage collection with a goal of reclaiming state from sectors that have been overwritten with newer data. To do this, we developed a separate `cleanup` function that reads in an LSVD file and essentially flattens all updates into the latest version. The `cleanup` function does this by reading the file and reconstructing the sector to offset map. It then starts at the beginning of the file and considers all updates on the disk sequentially. If any of the sectors contained in the update is up-to-date, the update is appended to the new file. The process skips the update if all sectors were overwritten by newer updates. Since the LSVD file is guaranteed to be consistent at any point in time, we can execute cleanup during live operation. Once the new file is constructed, it can be brought up-to-date with the replica synchronization protocol. After it has been synchronized, we can transparently change the backing file used by LSVD to the new, compacted version and delete the old file. We have not implemented the hot swap, however. Note that there are two drawbacks of this garbage collection approach. First, it must read and process the entire file, which can be a slow operation. Second, it requires enough free space on the current drive to store the new file. However, if we store $O(n)$ files on the single physical disk and run cleanup on them in round robin fashion, we need free space for only $O(1)$ disks.

We have not implemented snapshotting or incremental backups, but present here one possible design. On a snapshot, a disk contents are supposed to be saved and new updates are supposed to be written to a newly created delta file. One important property is that the delta file's size does not depend on the original disk size but only on the size of updates. To support snapshotting, LSVD files would be augmented with an additional pointer to the base disk in their superblock. On a snapshot, all updates to the current file is halted and a new delta file is created with a pointer to the current file. All new updates are performed on the new file. LSVD then manages both files by keeping additional information in the checkpoint on which file contains the sector. If the base file contains a pointer to another base file, the approach could extend to multiple files, easily enabling incremental backups.

```
struct lsvd_disk *create_lsvd(
       const char *pathname, uint64_t size);
struct lsvd_disk *open_lsvd(
       const char *pathname);
int cleanup_lsvd(const char *old_pathname,
       const char *new_pathname);
int read_lsvd(struct lsvd_disk *lsvd,
       char *buf, uint64_t length,
       uint64_t offset, uint64_t version);
int write_lsvd(struct lsvd_disk *lsvd,
       const char *buf, uint64_t length,
       uint64_t offset, uint64_t version);
int fsync_lsvd(struct lsvd_disk *lsvd);
uint64_t get_version(struct lsvd_disk *lsvd);
char *get_writes_lsvd(struct lsvd_disk *lsvd,
       uint64_t first_version, size_t *size);
int put_writes_lsvd(struct lsvd_disk *lsvd,
       uint64_t first_version, char *buf,
       size_t size);
int close_lsvd(struct lsvd_disk *lsvd);
```

Figure 5: LSVD library interface

We implemented LSVD as a C library. Figure 5 shows the interface exposed by the library. Most of the functions are straight forward. We use functions `get_writes_lsvd` and `put_writes_lsvd` to support replica synchronization. `get_writes_lsvd` returns raw update data for all updates since `first_version`. We do not keep a version to offset map, so we must scan all the updates backwards until we encounter a version less than or equal to `first_version`. Once we find the requested starting version, we copy the raw update data into the buffer. The raw data is then transferred to the lagging replica and appended to its LSVD file using function `put_writes_lsvd`, bringing it up-to-date.

5. Evaluation

To evaluate our implementation, we executed three sets of benchmarks. First, we evaluated LSVD implementation using simple microbenchmarks. Next, we demonstrated usefulness of checkpointing in LSVD. Finally, we executed some known file system benchmarks on ext4 mounted on remote FreEBS device.

5.1 Methodology

We ran our local experiments on one of two machines. The first was a Dell Precision T7500n Dual Processor desktop computer running 64-bit RHEL6 built on the 2.6.32 kernel for CentOS6. This machine housed a

Dual Quad Core Intel® Xeon® E5620 2.40GHz Processor, 24GB of RAM, two 600GB 15K RPM hard drives and one 2TB hard drive. The second was a desktop machine in the mumble lab with an Intel Core 2 Quad Q9400 2.66GHz CPU and 8 GB of RAM running Red Hat Enterprise Linux Server 6.4. In both cases, the storage used was a local ext4 partition. In the following section, we will indicate which machine was used for each experiment.

For testing FreEBS, we used a VirtualBox virtual machine for the client and ran our storage daemons on mumble machines identical to the mumble machine described above. The machines in the mumble lab are connected via a 1 gbps switch.

5.2 LSVD microbenchmarks

First, we wanted to evaluate our LSVD implementation. Since LSVD exports interface very similar to a file, we can easily compare LSVD throughput to regular file throughput. These benchmarks were all run locally on the Dell.

We evaluated LSVD using five benchmarks, which we ran in specific order:

1. **Sequential write** - Sequential write of 5GB random data to a new file/LSVD with 40KB buffers,
2. **Sequential read** - Sequential read of 5GB of previously created file with 40KB buffers,
3. **Random read** - Randomly read 128MB using 4KB buffers,
4. **Random write** - Randomly written 128MB using 4KB buffers,
5. **Sequential read 2** - Sequential read of 5GB of the file using 40KB buffers. However, this time we are also reading changes made by fourth benchmark.

All read and write operations were performed sequentially, without concurrency. Before read benchmarks, we cleared the file cache. Benchmarks were ran with LSVD checkpointing turned off.

Table 1 shows benchmark results. As expected, we see very similar performances for sequential write, sequential read and random read benchmarks. Since LSVD turns all random writes into sequential writes, random writes executed on LSVD show throughput 52x larger than sequential writes on a normal file. However, LSVD pays the price of fast random writes on subsequent sequential reads of the data. Benchmark sequential read 2 read the whole file again, but now with 128MB of new writes at random places in the file. Benchmark executed on a normal file showed the same throughput as the first time. However, random writes decreased LSVD sequential read throughput by 2.3x.

Table 1: LSVD microbenchmark results

Benchmark	File (KB/s)	LSVD (KB/s)
Sequential write	121,297	120,143
Sequential read	194,323	194,407
Random read	1,202	1,189
Random write	2,097	109,493
Sequential read 2	195,421	85,183

The benchmarks show that LSVD trades off fast random writes to slower sequential reads in some cases. However, with growing cache sizes, most of the reads are satisfied from cache and we argue this is a good trade-off to make.

5.3 Checkpointing evaluation

One of the important aspects of LSVD is checkpointing. We use it to keep recovery time from growing indefinitely. We designed an experiment to prove that our recovery time is bounded. This experiment was run locally on the Dell.

The experiment started with a write operation, which kept sending new updates to LSVD file until we killed it with `SIGKILL`. We then opened LSVD file and measured how long does a recovery take.

Figure 6 shows the results of the experiment. The horizontal axis shows runtime of the write before we killed it. Blue line shows time it took LSVD to recover to the state where it's able to receive new updates. Red line shows total data written during each experiment. When we ran the experiment for 150 seconds, it took LSVD 40 seconds to recover. However, when we ran the experiment for 200 seconds, recovery time was less than 5 seconds, even though we wrote more data to LSVD in total. These results show that our recovery time is bounded and does not grow linearly with number of updates or total data written.

It is interesting to see how recovery time grew rapidly when we executed write operation for 130 seconds. We have not looked into this issue more closely and are not sure what might be causing this slowdown.

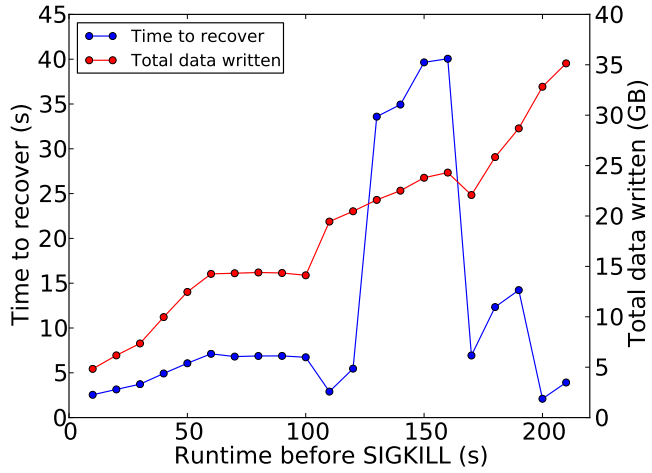


Figure 6: Recovery time after writing data to LSVD file for a specific amount of time.

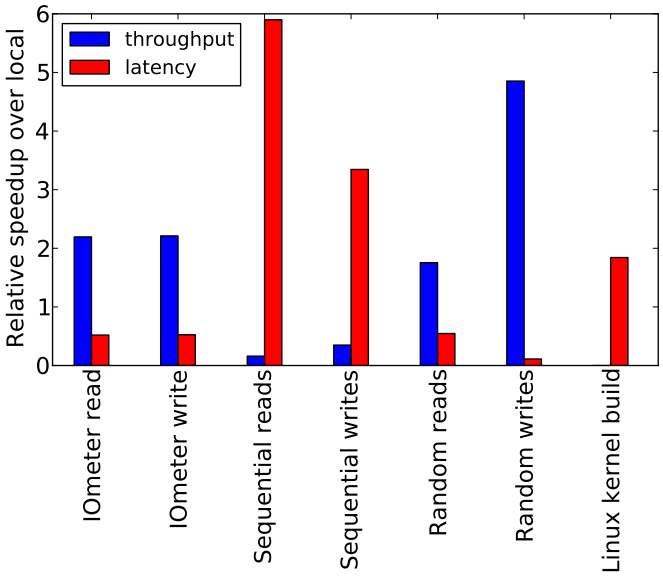


Figure 7: File system benchmarks.

5.4 File System Benchmarks

Finally, we measured FreEBS's performance as compared to local disk performance using both microbenchmarks and macrobenchmarks. Results for latency and throughput are presented in Figure 7.

The first six benchmarks were run using Flexible IO Tester (fio). The sequential write, sequential read, random write, and random read were run with the synchronous IO engine, which means that operations were conducted with blocking `read` and `write` calls. They all used a file size of 256 MB. The sequential benchmarks used block sizes of 4 KB for each operation while the random benchmarks used block sizes of 512 B. The write benchmarks were also configured to issue an `fsync` call every 128 operations.

The IOMeter benchmark was based on an example configuration included with the fio distribution designed to emulate Intel's IOMeter file server access pattern. It uses libaio (an asynchronous IO library for Linux), an 80% read/20% write mix, allows up to 64 requests to be in flight at a given time, and issues requests using a variety of block sizes. We configured it to use a 400 MB file.

Initially, direct I/O was enabled in our benchmarks, but we found that it inflated FreEBS's performance. This is because the replica managers use their local file caches. Since the network was extremely fast, this put the local machine at a disadvantage since it had no cache. Therefore, for all of our benchmarks, we do not use direct I/O.

A final benchmark we ran was to time how long it took to compile the Linux kernel on the volume. For this benchmark, we do not present throughput.

FreEBS excels at random writes due to the log-structured nature of its backing store. It also seems to have done better than local for random reads – this may be due to the relevant parts of the replica file remaining in cache after setup. Sequential read and write performance is quite bad. This is likely due to the synchronous nature of the tests; every request involves at least one round trip and multiple requests must be serviced serially. The Linux kernel build took twice as long on FreEBS, but this may also be affected by the fact that it was running inside a virtual machine. Further testing could mitigate these problems, but lack of time prevented us from accomplishing this.

After executing random write benchmark, we also evaluated usefulness of LSVD `cleanup` function. The size of LSVD backing file on primary replica after the benchmark was 2.4GB. Execution of `cleanup` function reduced the size to 290MB.

6. Future Work

The existence of a second-level cache in the file system of the replica managers is an opportunity for further research. In particular, we would like to explore the use of gray-boxing techniques to enhance cache exclusivity [6].

7. Conclusion

Presently, replicated block stores seem to be dominated by either DRBD-style or commercial NAS solutions. DRBD has the disadvantage that it is somewhat inflexible: it only allows two replicas (without stacking) and lacks dynamically resizing disk images. Further, DRBD puts all of its logic into a Linux kernel driver, making it more difficult to port to different systems. Commercial NAS solutions have the disadvantage of expense and opacity.

FreEBS shows that one can profitably move most of the replication logic into userspace programs. When the kernel driver becomes much simpler, it becomes more portable and reliable. Further, the replica managers themselves are more portable and maintainable. FreEBS is also a strong step in the direction of commoditizing replicated block storage for cloud computing since it enables such technology to be run on commodity hardware.

References

[1] Amazon elastic block store (EBS). `http://aws.amazon.com/ebs/`. Accessed: 2013-05-10.

[2] DRBD: Software develpment for high availability clusters. `http://www.drbd.org/`. Accessed: 2013-05-10.

[3] HP cloud block storage. `https://www.hpcloud.com/products/block-storage`. Accessed: 2013-05-12.

[4] Rackspace cloud block storage. `http://www.rackspace.com/cloud/block-storage/`. Accessed: 2013-05-12.

[5] Your private SAN and NAS in the cloud. `http://www.zadarastorage.com/features/`. Accessed: 2013-05-12.

[6] Lakshmi N Bairavasundaram, Muthian Sivathanu, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. X-ray: A non-invasive exclusive caching mechanism for raids. In *Computer Architecture, 2004. Proceedings. 31st Annual International Symposium on*, pages 176–187. IEEE, 2004.

[7] Frank Dabek, M Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. *ACM SIGOPS Operating Systems Review*, 35(5):202–215, 2001.

[8] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *ACM SIGOPS Operating Systems Review*, volume 37, pages 29–43. ACM, 2003.

[9] Brian Hellman, Florian Haas, Philipp Reisner, and Lars Ellenberg. The DRBD user's guide. `http://www.drbd.org/users-guide-8.4/`. Accessed: 2013-05-10.

[10] Edward K Lee and Chandramohan A Thekkath. Petal: Distributed virtual disks. *ACM SIGOPS Operating Systems Review*, 30(5):84–92, 1996.

[11] Sean Quinlan and Sean Dorward. Venti: a new approach to archival storage. In *Proceedings of the FAST 2002 Conference on File and Storage Technologies*, volume 4, 2002.

[12] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.*, 10(1):26–52, February 1992.