Silver: A scalable, distributed, multi-versioning, Always growing (Ag) File System

Michael Wei[⋆†], Amy Tai[⋆‡], Chris Rossbach[⋆]
Ittai Abraham[⋆], Udi Wieder[⋆], Steven Swanson[†], Dahlia Malkhi[⋆]
[⋆]VMware Research, [†]University of California, San Diego, [‡]Princeton University

Abstract

The storage needs of users have shifted from just needing to store data to requiring a rich interface which enables the efficient query of versions, snapshots and creation of clones. Providing these features in a distributed file system while maintaining scalability, strong consistency and performance remains a challenge. In this paper we introduce Silver, a file system which leverages the Corfu distributed logging system to not only store data, but to provide fast strongly consistent snapshots, clones and multi-versioning while preserving the scalability and performance of the distributed shared log. We describe and implement Silver using a FUSE prototype and show its performance characteristics.

1 Introduction

Storage capacity has steadily grown over the years, and with it, software workloads and user expectations increasingly shifted toward write-once, ever-growing stores. This paper introduces Silver, a distributed file-system designed as an ever-growing store.[1] Silver builds on the principles of a log-structure store, which were historically introduced in order to serialize IO [11, 6], not to expose the versions. It retains the lock-free read/write IO path of classical LFS, enhanced with with features of modern LFS file systems [10, 17, 15], such as "time-travel" versions, copy-on-write (CoW), snapshot and cloning. At the same time, it provides a clean-slate, efficient design for distributed logging, global snapshot, and unconstrained cloning with recursive-write avoidance.

The write-once substrate of Silver utilizes the Corfu [2] distributed logging system. Silver keeps track of all changes in the log, enabling users to go back to any point in time for almost free. It is built to be truly append-only and never overwrites data. While other append-only file systems exist, especially in the realm of optical media [1], Silver combines Corfu with Replex [12], a unique replication protocol which enables efficient *log virtualization* to support multiple writers, low-latency linearizable reads, and the ability to create fast copy-on-write clones. The Silver design has the following desirable properties:

- Data is sharded over a cluster for scalability, and at the same time, Silver provides read-after-write strict consistency semantics.
- At the foundation of the system is a log, which supports multi-versioning with continuous, consistent snapshots: every operation to Silver is logged and Silver efficiently supports "time-travel" on the log.
- Every directory in the file system hierarchy is mapped a virtualized log, called a "stream", serving as an indirect reference to the latest state of the directory. Uniquely, this allows copy-on-write snapshot cloning of files or sub-directories at any level while completely circumventing the recursive update problem [16] (Section 2.3).
- At any moment, taking a snapshot is done simply by capturing a prefix of the log, which allows for easy implementation of tiering.
- Resting on the LFS approach, concurrent readers and writers have separate IO paths and require no locking.

2 System

2.1 Distributed Log

Silver is built on top of a distributed, shared log [2, 3]. Our log is made of two components: a high throughput

[1] We name the system after silver, the element indicated by the symbol Ag, which stands for "Always Growing".

Operation	Description
`read(addresses)`	Get the data stored at a particular address or list of *addresses*.
`read(stream, address)`	Get the data stored at a particular address or list of *addresses* on a specific *stream*.
`read(address, offset, len)`	Partially read *len* bytes of an *extent entry* at *address* and *offset*.
`append(stream, address, data)`	Append *data* to a given *address* on a particular *stream*.
`check(stream)`	Get the last address written to on a particular stream.

Table 1: Operations supported by our distributed log.

sequencer which orders append operations and a write-once storage device which stores updates. This design has been shown to scale to more than half a million operations per second [2]. We have previously described an implementation of this design on a FPGA in hardware [14].

As described in Tango [3], in addition to the basic log `append` and random `read` operations, our log also supports *streams*, which are essentially virtualized logs within our log, identified by a unique 128-bit id. In Silver, we implement log virtualization using Replex [12], a unique replication protocol which enables fast, random access to streams by building a secondary index during replication. This avoids the overhead and complexity of traversing backpointers in Tango. Replex also has strong failure recovery characteristics which are outside the scope of this paper.

We describe the basic operations supported by our log in table 1. Silver is entry-oriented, not block oriented, and clients may only append and read entries. Entries in our log are variably-sized and we do not impose a size limit.

2.2 Distributed File System Design

On disk, Silver is stored as an ever-growing log as described in section 2.1. Figure 1 depicts the on-disk layout of Silver and compares it to other file systems in use today. Unlike these file systems, however, Silver is distributed and replicated so that there is a global log and stream replicas which provide locality and efficient random accesses. The log is divided into three different types of streams, which represent either file metadata, data or directories:

- *Metadata streams*, which contain file metadata and represent files, such as attributes. Small files $\leq$*4KB* also store data in the metadata stream.
- *Data streams*, which contain the actual file data.
- *Directory streams*, which contain directories that point to other directory or file streams.

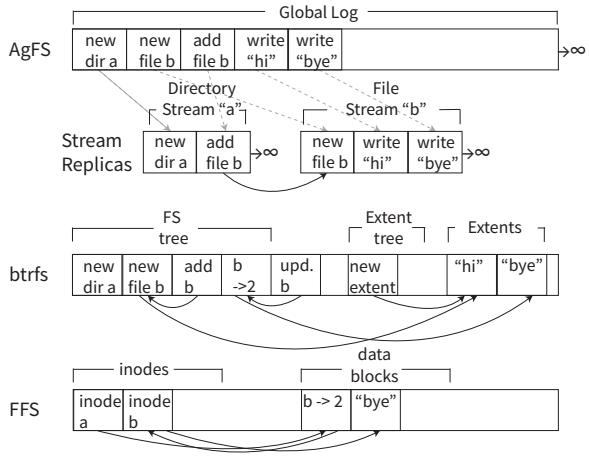

Figure 1: Simplified models of the on-disk layouts of Silver, btrfs [10] and FFS [8]. In each file system, a directory named `a` is created and a file `b` is created in it with the contents "hi". Finally, file `b`'s contents is overwritten with "bye'. Unlike the other file systems, Silver is replicated, and stream replicas provide efficient random access to streams.

Each stream consists of entries which record updates or changes. For example, when a file is added to a directory, an "add" entry is appended to the directory stream with a pointer to the file's metadata stream. Table 2 contains the basic operations supported by each stream. Even though users may delete directories or overwrite data, the log preserves the order in which changes are applied to the file system, so that a delete operation can easily be "undone". Every address in the log is a *version* in the file system, and "time-travel" can be done by restricting traversal to a specific set of addresses.

Every file system starts with a root directory. Clients use the stream ID of the root directory to find the file system. To open a file system, an uninitialized client first calls the `check` function to get the last update on the root directory from the sequencer. The client then must read the entire root directory stream and apply those updates in-memory to get the current state of the root directory.

Once the root directory is read, subsequent traversals of the file system selectively read the streams necessary to satisfy that request. For example, to perform a `ls` of a child directory, the client would only need to read the stream for that directory. All requests for metadata (metadata and directory streams) are served quickly and efficiently from in-memory state, and updating that state consistently involves merely contacting the sequencer, reading any updates on the stream and then applying it to the in-memory state. This permits fast data structures such as hash tables and skip-lists to be used rather than the traditional B-trees for the file map.

Operation	Description
All Streams	
`copy(srcid, dstid, ver)`	Copies a stream with the given *srcid* up to *ver* to *dstid*
File Stream	
`setAttr(attr, val)`	Sets an *attr* to a given *val*
`write(data, offset)`	Writes *data* to a given *offset*.
Data Stream	
`write(data, offset)`	Writes *data* to a given *offset*.
Directory Stream	
`addChild(child)`	Adds *child* to the directory if it does not exist.
`delChild(child)`	Remove *child* from the directory if it exists.
`setAttr(attr, val)`	Sets an *attr* to a given *val*

Table 2: Types of operations supported on streams.

However, keeping the data for large files in memory would be costly and impractical. To implement large files efficiently, we implement extent entries, which contain the entire file within a single entry, and we support partial reads of that entry. The single entry is written sequentially into the log and does not require multiple pointers or a tree structure to traverse, unlike traditional file systems which write many extents that must be traversed through a tree due to allocation and reallocation of the extent space. This allows fast random accesses to large chunks of data stored efficiently within the log.

2.3 Streams and Indirection

An important optimization is that pointers in Silver always refer to other streams by their id, rather than a physical address as in other file systems. For example, in figure 1, directory a points to metadata stream b, rather than physical address 2. This allows the pointer to b to remain valid even after it is updated and clients can quickly get the latest version by contacting the appropriate stream replica. In this manner, we eliminate the dependency on the equivalent of the inode map in LFS [11]. Other file systems, like btrfs, suffer from the recursive update problem [16] so updates must propagate to the root, as the pointer to physical address 2 is no longer valid, so a new root must be written pointing to the new update leading to significant write amplification.

Efficiently supporting streams has many benefits, which we describe in the next sections:

Caching - Read Path Separation. File systems today employ many reader-writer locks for mutual exclusion where read and write paths intersect, since writers may overwrite previously written data. In Silver, the read path and write path are separated since no overwriting occurs: any successful write is immutable, which obviates the need for mutual exclusion logic. Mutual exclusion is a even bigger problem in a distributed systems where many have resorted to relaxed consistency for performance, Silver is able to never compromise consistency by never overwriting.

This separation greatly simplifies caching in Silver. Silver currently implements a efficient LRU cache of log entries in DRAM for faster log traversal and to cache large files, while in-memory state serves as a cache for both metadata and small files. In a traditional distributed file system, eviction of stale data must be detected and often the entire file must be copied. In Silver, updating stale state simply involves contacting the sequencer, which informs the client of staleness immediately, and reading any updates, which are stored as deltas from the log, and applying the state in-memory.

Tiering - Write Path Separation. Isolating the write path also simplifies tiering, and allows Silver to easily take advantage of heterogeneous storage systems. For example, consider a storage system which consists of fast NVM, SSD and hard disks. A tiered Silver system could direct writes first to fast NVM. As the NVM fills up sequentially, writes can be evicted to the SSD in large sequential chunks, permitting the NVM to be reused. Before eviction, a small update to the read cache's map would allow reads to continue being serviced, and the read cache would make sure that popular old entries still have fast service times, despite being evicted out to archival storage.

Reclaiming Space. Even though Silver is designed on an ever-growing log, we recognize that practical storage considerations may limit the number of updates which a system can store. To mitigate this limitation, we offer two mechanisms: first, we offer a compress command which compresses prefixes of the log, and appends the compressed prefixes to the end of the log. Second, we offer a checkpoint mechanism which takes an address and compacts the history of each stream into a single entry to free space. However, once a checkpoint is performed, "time-travel" to history previous to the checkpoint is no longer permitted since history is lost. Users may choose to checkpoint only the histories they are interested in. As compression and checkpointing always free data from the start of the log and append to the end of the log, this allows a fixed storage space to be used as a circular buffer.

Snapshots. Since every update to Silver is logged, Silver supports full time travel by playing back the log. Snapshots are truly free in Silver, unlike other log-based

systems where snapshots must be explicitly created to freeze data blocks. In addition, Silver is always consistent because the log is an authoritative source for the ordering of writes. Shipping a snapshot of Silver is as simple as transferring a prefix of the log. Currently a user of Silver can read the file system version by reading a special entry in the file system - reading this entry queries the sequencer. To access the snapshot, a clone command is provided which takes the version number and directory (which may be the root) as a parameter and creates a new fully writeable clone of the directory using CoW, which is described in the next section.

Copy on Write. Copy on Write (CoW) is a feature supported by many logging file systems. It enables a file system to quickly create a diverging copy by referencing the source and only recording changes. Silver supports CoW of any stream. The copy command creates a *CoW entry* with the source stream and an address, which denotes the version where the new stream diverges from the source stream (we require that *source version* $<$ *destination version*). When a directory entry is copied, subsequent traversals to children of that directory create copies as well to ensure that the new history diverges from the source.

Other CoW file systems suffer from *fragmentation* under random writes because they must allocate data blocks or extents for a CoW file and update pointers for every write. This problem is so bad that most logging file systems recommend disabling CoW support, especially for large files. Silver does not suffer from this problem: random writes are sequentialized by virtue of the log, and the fast streaming support eliminates problems fragmentation may pose.

3 Evaluation

We have prototyped Silver in Java 8 over FUSE, through the use of Java Native Runtime (JNR) bindings. While the use of Java prevents our current prototype from performing as well as a native implementation, our prototype enables us to understand and evaluate the key benefits of Silver. Our implementation is scalable and distributed: following the design of Corfu [2], our log can be sharded and replicated across many nodes. However, while we aim to be as POSIX compliant as possible, due to limitations in FUSE, Silver is not fully POSIX-compliant. Furthermore, the use of FUSE adds significant overhead.

Our current implementation Silver is elegant and concise, taking a mere 3,186 SLOC in Java. Part of the simplicity of Silver owes to the fact that much of the streaming logic is implemented in the distributed log, which is about 15k SLOC but reusable by other applications (other applications may directly use the log simultaneously with Silver). We also envision that the distributed log may be implemented in hardware.

Operation	Latency
clone fs	0.8ms
clone dir	0.8ms
clone file	0.8ms
access fs (mount)	1.8ms
access cloned fs	2.2ms
access dir	0.9ms
access cloned dir	1.0ms
access (cat) 4KB file	0.6ms
access cloned 4KB file	0.8ms

Table 3: Silver performance on a 1Gbit network link.

In table 3, we show the basic performance of Silver with 3x replication, demonstrating fast cloning performance. No distributed file system that we are aware of can provide such a consistent global snapshot efficiently. Due to the different guarantees provided by other file systems and the overhead of FUSE, we are unable show meaningful comparisons with other file systems, which we leave to future work.

4 Related Work

Log-structured CoW File Systems. While Silver is distributed, single node log-structured file systems still serve as a useful comparison point for Silver's design. Historically, these stores were introduced mostly in order to serialize IO, and not in order to expose the history of versions. Systems like LFS [11] and Zebra [6] aggressively garbage-collected all by the latest updates to any block. Consequently, the metadata issues they tackle are quite different: the name space did not need to expose versions, nor support cloning and snapshots.

btrfs [10], nilfs [7], WAFL [5] and ZFS [17] are copy-on-write file systems with snapshot capability. These systems primarily log metadata, but data is typically stored in allocated regions called *extents* in btrfs and *slabs* in ZFS. Generating snapshots, as a result, is not automatic as the filesystem must lock data regions to create a snapshot. Furthermore, the allocated regions are at risk for inconsistency, whereas in Silver the log is the pristine source of ordering for all metadata and file data. Finally, allocators can suffer from fragmentation especially with a high number of random writes. Silver converts random writes into compact sequential writes while an efficient cache mechanism enables these writes to be read efficiently.

Append-only File Systems. Many append-only file systems exist today which are primary a result of limitations of physical media. For example, UDF [1] is an example of a system designed for optical media, and Quinlan [9] describes a file system for early WORM media. Unlike Silver, these file systems are mainly designed for archival content, which is reflected by their slow access times and mount times.

Distributed File Systems. Most distributed file systems such as the popular HDFS [4], Ceph [15] and CalvinFS [13] separate the storage of metadata and data. This separation makes it difficult to create true snapshots: for example, in HDFS and Ceph snapshots only capture the state of the metadata. Data, which is stored separately from metadata can continue to change after a snapshot, resulting in a inconsistent snapshot. Furthermore, these file systems are not copy-on-write, so creating a modifiable clone is an expensive operation.

Streams and Backpointers. Tango [3] presented the concept of a stream within a distributed log. In Tango, streams never referred to each other: rather they contained opaque objects on a single log so that transactions could be run against them. In Silver, our file system leverages many streams in order to support efficient accesses and copies.

In Tango, backpointers are used to enable streams, which imposes a burden on both the sequencer, which must persist the last token that was issued for each stream, and during failures, which require scanning the log to find the stream. Furthermore, bulk reads of a stream are not possible with backpointers. Silver uses the Replex [12] replication protocol to address these issues.

5 Conclusion

The storage needs of users have shifted from just needing to store data to requiring a rich interface which enables the efficient query of versions, snapshots and creation of clones. Silver leverages a distributed shared log to efficiently provide these components without sacrificing consistency, scalability or performance.

References

[1] Optical Storage Technology Association. *Universal Disk Format Specification 2.60*. 2005.

[2] Mahesh Balakrishnan, Dahlia Malkhi, John D Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. Corfu: A distributed shared log. *ACM Transactions on Computer Systems (TOCS)*, 31(4):10, 2013.

[3] Mahesh Balakrishnan, Dahlia Malkhi, Ted Wobber, Ming Wu, Vijayan Prabhakaran, Michael Wei, John D Davis, Sriram Rao, Tao Zou, and Aviad Zuck. Tango: Distributed data structures over a shared log. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 325–340. ACM, 2013.

[4] Dhruba Borthakur. Hdfs architecture guide. `http://hadoop.apache.org/common/docs/current/hdfsdesign.pdf`, 2008.

[5] John K Edwards, Daniel Ellard, Craig Everhart, Robert Fair, Eric Hamilton, Andy Kahn, Arkady Kanevsky, James Lentini, Ashish Prakash, Keith A Smith, et al. Flexvol: flexible, efficient file volume virtualization in wafl. In *Usenix 2008 Annual Technical Conference*, pages 129–142. USENIX Association, 2008.

[6] John H. Hartman and John K. Ousterhout. Zebra: A striped network file system. Technical Report UCB/CSD-92-683, EECS Department, University of California, Berkeley, Apr 1992.

[7] Ryusuke Konishi, Yoshiji Amagai, Koji Sato, Hisashi Hifumi, Seiji Kihara, and Satoshi Moriai. The linux implementation of a log-structured file system. *ACM SIGOPS Operating Systems Review*, 40(3):102–107, 2006.

[8] Marshall K McKusick, William N Joy, Samuel J Leffler, and Robert S Fabry. A fast file system for unix. *ACM Transactions on Computer Systems (TOCS)*, 2(3):181–197, 1984.

[9] Sean Quinlan. A cached worm file system. *Softw., Pract. Exper.*, 21(12):1289–1299, 1991.

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

[11] Mendel Rosenblum and John K Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[12] Amy Tai, Michael Wei, Michael J. Freedman, Ittai Abraham, and Dahlia Malkhi. Replex: A scalable, highly available multi-index data store. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, June 2016. USENIX Association.

[13] Alexander Thomson and Daniel J Abadi. Calvinfs: consistent wan replication and scalable metadata management for distributed file systems. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 1–14, 2015.

[14] Michael Wei, John D Davis, Ted Wobber, Mahesh Balakrishnan, and Dahlia Malkhi. Beyond block i/o: implementing a distributed shared log in hardware. In *Proceedings of the 6th International Systems and Storage Conference*, page 21. ACM, 2013.

[15] Sage A Weil, Scott A Brandt, Ethan L Miller, Darrell DE Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation*, pages 307–320. USENIX Association, 2006.

[16] Yiying Zhang, Leo Prasath Arulraj, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. De-indirection for flash-based ssds with nameless writes. In *FAST*, page 1, 2012.

[17] Yupu Zhang, Abhishek Rajimwale, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. End-to-end data integrity for file systems: A zfs case study. In *FAST*, pages 29–42, 2010.