Replication, History, and Grafting in the Ori File System

Ali José Mashtizadeh, Andrea Bittau, Yifeng Frank Huang, David Mazières

Stanford University

Abstract

Ori is a file system that manages user data in a modern setting where users have multiple devices and wish to access files everywhere, synchronize data, recover from disk failure, access old versions, and share data. The key to satisfying these needs is keeping and replicating file system history across devices, which is now practical as storage space has outpaced both wide-area network (WAN) bandwidth and the size of managed data. Replication provides access to files from multiple devices. History provides synchronization and offline access. Replication and history together subsume backup by providing snapshots and avoiding any single point of failure. In fact, Ori is fully peer-to-peer, offering opportunistic synchronization between user devices in close proximity and ensuring that the file system is usable so long as a single replica remains. Cross-file system data sharing with history is provided by a new mechanism called *grafting*. An evaluation shows that as a local file system, Ori has low overhead compared to a File system in User Space (FUSE) loopback driver; as a network file system, Ori over a WAN outperforms NFS over a LAN.

1 Introduction

The file system abstraction has remained unchanged for decades while both technology and user needs have evolved significantly. A good indication of users' needs comes from the products they choose. At $1/GB/year, cloud storage like Dropbox [1] commands a 25x premium over the cost of a local hard disk. This premium indicates that users value data *management*—backup, versioning, access from any device, and multi-user sharing—over storage capacity. But is the cloud really the best place to implement data management features, or could the file system itself directly implement them better?

SOSP'13, Nov. 3–6, 2013, Farmington, Pennsylvania, USA.
ACM 978-1-4503-2388-8/13/11.
http://dx.doi.org/10.1145/2517349.2522721

In terms of technological changes, disk space has increased dramatically and has outgrown the increase in wide-area bandwidth. In 1990, a typical desktop machine had a 60 MB hard disk, whose entire contents could transit a 9,600 baud modem in under 14 hours [2]. Today, $120 can buy a 3 TB disk, which requires 278 days to transfer over a 1 Mbps broadband connection! Clearly, cloud-based storage solutions have a tough time keeping up with disk capacity. But capacity is also outpacing the size of managed data—i.e., the documents on which users actually work (as opposed to large media files or virtual machine images that would not be stored on Dropbox anyway).

Ori is a new file system designed to leverage ever-growing storage capacity to meet user's data management needs. Ori further capitalizes on the increasing diversity of devices containing storage (e.g., home PC, work PC, laptop, cell phone, USB disks). Based on these trends, Ori achieves several design goals.

First, Ori subsumes backup. It records file histories, allowing easy recovery of accidentally deleted or corrupted files. It furthermore replicates file systems across devices, where device diversity makes correlated failure less likely. Recovering from a device failure is as simple as replicating an Ori file system to the replacement device. A new replica can be used immediately even as the full data transfer completes in the background.

Second, Ori targets a wide range of connectivity scenarios. Because new replicas are immediately available, replication feels light-weight, akin to mounting a network file system. However, once replication is complete, a replica can be used offline and later merged into other replicas. Ori also uses mobile devices to overcome limited network bandwidth. As an example, carrying a 256 GB laptop to work each day provides 20 times the bandwidth of a 1 Mbps broadband connection. Ori leverages this bandwidth to allow cross-site replication of file systems that would be too large or too write-active for a

traditional cloud provider. Several other techniques minimize wide area network (WAN) usage in Ori. These should prove increasingly important as disk capacity and managed data sizes outpace WAN bandwidth.

Finally, Ori is designed to facilitate file sharing. Using a novel feature called *grafts*, one can copy a subtree of one file system to another file system in such a way as to preserve the file history and relationship of the two directories. Grafts can be explicitly re-synchronized in either direction, providing a facility similar to a distributed version control system (DVCS) such as Git [3]. However, with one big difference: in a DVCS, one must decide ahead of time that a particular directory will be a repository; while in Ori, any directory can be grafted at any time. By grafting instead of copying, one can later determine whether one copy of a file contains all changes in another (a common question when files have been copied across file systems and edited in multiple places).

Ori's key contribution is to store whole-file-system history and leverage it for a host of features that improve data management. To store history, Ori adapts techniques from version control systems (VCS); doing so is now feasible because of how much bigger disks have become than typical home directories. Better data management boils down to improving durability, availability, latency over WANs, and sharing.

Ori's history mechanism improves durability by facilitating replication, repair, and recovery. It improves availability because all data transmission happens through pairwise history synchronization; hence, no distinction exists between online and offline access—any replica can be accessed regardless of the others' reachability. History improves WAN latency in several ways: by restricting data transfers to actual file system changes, by facilitating opportunistic data transfers from nearby peers (such as physically-transported mobile devices), and through a novel *background fetch* technique that makes large synchronizations appear to complete instantly. Finally, Ori improves sharing with its *graft* mechanism.

We have built Ori for Linux, FreeBSD, and Mac OS X as a File system in User Space (FUSE) driver. Our benchmarks show that Ori suffers little if any performance degradation compared to a FUSE loopback file system, while simultaneously providing many data management benefits.

2 Ori Overview

Figure 1 illustrates the main concepts in Ori. Each user owns one or more *file systems* replicated across his or her devices. Devices contain *repositories*, each of which is a replica of a file system. A repository is a collection of *objects* that represent files, directories, and snapshots

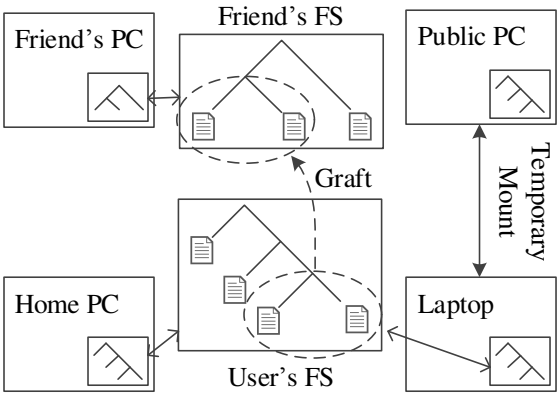

Figure 1: Example Ori usage. A file systems is replicated across a user's devices. The user has remotely mounted and instantly accessed the file system on a public workstation. A friend has *grafted* one of the user's directories into a different file system. Changes to grafted files can later be manually synchronized with their source.

in the file system. Both files and their history are replicated across devices. Ori targets collections of files much smaller than local storage, making it reasonable to replicate multiple file systems on each device and incur significant storage overhead for history.

The figure shows two ways of accessing remote data. The first is through replication and mounting of a file system, as shown with the Public PC. The second is through *grafts*, which share both files and their history across different file systems. A graft can be repeated to keep synchronizing a directory across file systems.

Ori does not assume constant connectivity between replicas of a given file system. Instead, through automatic device discovery, it synchronizes whatever replicas happen to be able to communicate at any given time. Such an approach necessarily prioritizes availability over consistency, and can give rise to update conflicts. If possible, Ori resolves such conflicts automatically; otherwise, it exposes them to the user, much like a VCS.

2.1 Challenges

Ori has to deal with several issues common to distributed file systems. First, devices may crash or suffer from silent data corruption. Naïve replication can actually compound such problems, as seen in cloud storage solutions [4]. Second, replication and other heavyweight operations must not interfere with user productivity through poor performance. In particular, it is critical for synchronization operations to minimize waiting time and return needed data quickly, even as they continue to transfer large histories in the background.

Ori faces two additional challenges that are not present in other distributed file systems. First, the *normal* case for Ori is that we expect intermittent connectivity; some pairs of nodes may never communicate directly. For example, a cell phone carried by a user has connectivity to different machines depending on the user's physical location, and a work and home machine may have no direct connectivity. Typical distributed file systems consider connectivity issues to be a failure case.

Ori must work offline, and as a consequence it must handle update conflicts or so-called "split-brain" issues. Conflicts need to be detected and resolved. Like cloud storage solutions, we depend on the user to resolve conflicts if it is not possible to do so automatically. Unlike the few previous file systems that allow update conflicts, Ori leverages file history to provide three-way merges.

2.2 Mechanisms

To achieve our vision and address these challenges, we made a few architectural choices. Ori borrows from Venti [5] the concept of a content addressable storage (CAS) system. It borrows Git's model for storing history [3]. For handling update conflicts, Ori borrows three-way merging from VCS [3,6]. Finally, we designed Ori to support two crucial optimizations: *background fetch* allows data transfer operations to complete asynchronously and thus hide the end-to-end time associated with replication; *distributed fetch* leverages nearby Ori file systems as a cache for objects.

Ori's CAS uses SHA-256 hashes to obtain a globally unique namespace for addressing objects. SHA-256's collision-resistance allows Ori to detect damaged files and recover them from another device. Ori's *distributed fetch* optimization also leverages collision-resistance to look for the same object in nearby peers (that may be replicating different file systems).

In our data model, all objects (i.e., files, directories, and snapshots) are stored in CAS and are immutable. Immutable objects make it easier to record history and create fast snapshots. This history is then used to detect and resolve update conflicts. Because of three-way merges, Ori can resolve many update conflicts automatically. Note that update conflicts also occur in cloud storage providers (e.g., Dropbox), when a file is modified on two machines without network connectivity.

The *background fetch* optimization allows long running tasks, such as replication and synchronization plus merging, to complete in the background while the user continues working. This improves perceived performance and enhances the usability of the system. By virtue of completing any user-visible synchronization first, these operations also reduce the window for update conflicts when connectivity becomes available.

The *distributed fetch* optimization also helps us achieve higher performance and reduce the impact to the user. When synchronizing over slower network links, Ori can use unrelated but nearby repositories as a cache to accelerate the replication operation.

2.3 User Interface

Ori is an ordinary file system, accessed through the customary POSIX system calls (e.g., open, close, read, write). However, a command-line interface (CLI) bypasses the POSIX API to provide Ori-specific functions. Examples include setting up replication between two hosts, using history to access an accidentally deleted file, and reverting the file system after an undesirable change. The CLI also allows cross-file-system sharing with history through our grafting mechanism. Manual synchronization and conflict resolution is also possible through the CLI.

Table 1 shows a list of commonly used commands. Some commands are rarely used by users; for instance, the *pull* command, used to initiate manual unidirectional synchronization, is unnecessary for users of *orisync*.

Command	Description
ori newfs	Create a new file system
ori removefs	Remove local repository
ori list	List local repositories
ori status	Show modified files
ori diff	Show diff-like output
ori snapshot	Create a snapshot
ori log	Show history
ori replicate	Replicate a remote repository
ori pull	Manually synchronize one way
ori merge	Manually merge two revisions
ori checkout	Checkout a previous revision
ori purgesnapshot	Purge a commit (reclaim space)
ori graft	Graft a file or directory
orifs	Mount repository as a file system
orisync init	Configure orisync

Table 1: A list of the basic commands available to Ori users through the command line. Included is the *ori* CLI that controls the file system, the FUSE driver (*orifs*), and the automatic repository management daemon (*orisync*).

To give a feel for the Ori user experience, we present several examples of tasks accomplished through the CLI.

▶ Our first example is configuring replication between two hosts. To enable *orisync*, our device discovery and synchronization agent, a user must interactively configure a *cluster name* and *cluster key* using the `orisync init` command. *orisync* detects changes within approx-

imately five seconds (based on the advertisement interval). The following commands create and mount an autosynchronized file system.

Listing 1: Configuration of a user's initial repository.

```
ori newfs MyRepo
orifs MyRepo
```

In the example above, a repository named `MyRepo` is created, and mounted onto a directory with the same name. By default *orisync* is enabled, but can be disabled with the `--nosync` flag. Next, in Listing 2, we replicate and mount the file system on a different computer from the first host over SSH.

Listing 2: Remote mounting a file system

```
ori replicate --shallow user@host:MyRepo
orifs MyRepo
```

In this example we use the `--shallow` flag to enable the background fetch optimization. Data is fetched from the source in the background and on-demand, and once enough data is replicated the machine will be ready for offline operation. We mount the `MyRepo` replica onto a directory with the same name.

► The second example, shown in Listing 3, shows how to use version history to recover a deleted file. The `.snapshot` directory in the file system's root allows users to access snapshots by name. Silent data corruption repair, which is not demonstrated in this example, is transparent to the user.

Listing 3: Snapshots and recovery

```
ori snapshot BEFORE
rm deleted_file
ori snapshot AFTER
cp .snapshot/BEFORE/deleted_file ./
```

► The third example reverts the whole file system to a previous state rather than recovering individual files. We can discard all changes and revert to the snapshot named `BEFORE`, with the use of the `ori checkout --force` command.

► In Listing 4, we show the fourth example of sharing files with history (grafting) between two locally mounted file systems. Once complete, the history of the graft is made available to the user. Diffs can be generated to see what has changed or determine what is the latest version. By re-running the `graft` command, new changes can be pulled from the source

Listing 4: Grafting files between two repositories.

```
ori graft src_repo/dir_a dst_repo/dir_b
```

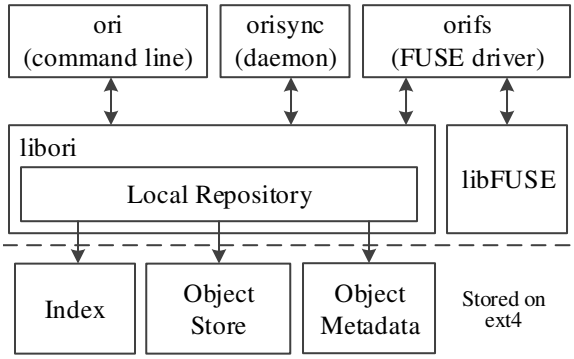

Figure 2: Ori system diagram showing the *ori* CLI, and *orifs* FUSE driver. These user interfaces are built on top of libOri, which implements local repository access and a client/server API for exchanging snapshots. In this figure we show the various storage files of a local file system stored on an existing file system, in this case ext4.

► Our final example uses the *pull* and *merge* commands to manually synchronize and merge the file system when *orisync* is disabled. For example this can be used to migrate changes from a test to production setup once testing is complete. Conflict resolution is done on a running file system, but if automatic resolution fails (or if a file is actively being modified) two files will be created next to the conflicting file with the suffixes `:base` and `:conflict`. A user will have to manually resolve the conflicts.

3 Design

Figure 2 shows a high level system diagram of Ori. It includes user facing tools: the *ori* CLI, the automatic synchronization tool *orisync*, and the FUSE driver *orifs*. Users invoke *ori* to manage the file system and conduct manual operations. *orisync* discovers and automatically synchronizes with other replicas. *orifs* implements some automated operations such as periodic snapshotting and provides a read-only view of all snapshots. These tools are built on top of libOri, our main library that implements the local and remote file system abstractions. It also implements higher-level operations including pulling and pushing snapshots, merging, and other operations discussed in the following sections.

Figure 2 also shows the three main structures that are stored on a backing file system (e.g., local ext4): the *index*, *object store*, and *object metadata*. The index forms an indirection layer necessary to locate objects on disk by hash rather than location. Objects in our file system,

e.g., files, directories and snapshots, are stored in the object store. Mutable metadata, such as reference counts, are stored in a separate metadata structure. The metadata is a per-object key-value store that is not explicitly synchronized between hosts.

In Section 3.1, we describe Ori's core data model. Next, the grafting mechanism that is used for file sharing is described in Section 3.2. Replication operations are described in Section 3.3, which also includes related distributed and background fetch optimizations. In Section 3.4 we describe the device discovery and management agent. Subsequently, the space reclamation policy is discussed in Section 3.5. Section 3.6 discusses replication and recovery. Finally, Section 3.7 describes the integrity facility for detecting data corruption or unauthorized tampering with a file system.

3.1 Data Model

Like a DVCS (e.g., Git), Ori names files, directories, and snapshots by the collision-resistant hash of their content (using SHA-256). Each snapshot captures a single point in the history of the file system. Each file system has a Universally Unique Identifier (UUID) associated with it, that is used to identify instances of the same file system.

Ori's basic model supports four types of object: *Commit* objects, which represent snapshots and their historical context; *Trees*, which represent directories; and both *Blobs* and *LargeBlobs*, which represent individual files. Throughout this paper, we use the term *snapshot* to refer to the collection of objects that are accessible from a single commit object and not just the commit itself.

The objects are stored in an append-only structure called a packfile that contains multiple objects. Objects are written in clusters, with headers grouped at the beginning of each cluster. This speeds up rebuilding the index on crashes or corruptions. The index and metadata files are log-structured and are updated after objects are stored in a packfile. Ori uses a transaction number to allow for rollback and to ensure the index and metadata are updated atomically. Deleted objects are marked for deletion in the index and reclaimed later by rewriting packfiles during garbage collection. Garbage collection is done rarely to avoid write amplification, a reasonable choice given Ori's premise of abundant free space.

Commit objects consist of a series of immutable fields that describe the snapshot. All snapshots have the following fields: root tree hash, parent commit hash(es), time, and user. Users may optionally give snapshots a name and description, which may help them when working with history-related operations. Other optional attributes are specific to particular features such as grafting and tamper resistance (see Sections 3.2 and 3.7).

Tree objects, which represent directories, are similar to Git's tree objects. Unlike directories in normal file systems, a tree object combines both the directory entry and inode fields into a single structure. Unlike traditional inodes, Ori directory entries point to file contents with a single content hash. Combining inodes with directory entries reduces IO overhead and file system complexity. However, Ori does not support hard links, as a separate data structure would be required to differentiate them from deduplication.

Blobs store data of files less than one megabyte in size (a configurable value). LargeBlobs split large files into multiple Blobs by storing a list of Blob hashes of individual file chunks. Ori uses a variable size chunking algorithm that creates chunks between 2KB and 8KB, with an average of 4KB. Variable size chunking uses Rabin-Karp fingerprinting to break chunks at unique boundaries, which is similar to the technique used by LBFS [7].

LargeBlobs exploit the clustering of data in packfiles by storing the large file contiguously in the packfile, while omitting any deduplicated chunks. When a read operation spans multiple chunks of a LargeBlob, Ori uses a vectored read, reordering and coalescing the vectored chunks to benefit from packfile locality. (Coalescing chunks reduces disk seeks, while vectored reads reduce IO overhead.) Ori maintains this optimization across replication operations, as discussed in Section 3.3.

LargeBlobs provide sub-file data deduplication, which saves space and minimizes bandwidth use on transfers. Other systems, such as Git, rely instead on delta compression for the same purpose. But delta-compression algorithms can be very CPU and memory intensive, too high a price to pay in a general purpose file system. Note that deduplication can occur between any two Blobs, even if they do not share or are not part of a LargeBlob. LargeBlob chunking (and thus deduplication) is done during snapshot creation.

Figure 3 illustrates how Ori's various objects are organized. Trees and Blobs mirror the structure of directories and files in the actual file system. The Commit objects correspond to snapshots. Objects may be referenced from multiple places in the same tree or across snapshots, thus enabling deduplication. The entire tree up to the latest commit forms a Merkle tree that encompasses the file system structure and full history.

The use of content hashes to name objects has two implications. First, objects are uniquely identified in a global namespace across devices and file system instances. Second, file system layout is independent of file system structure. However, this architecture in practice requires an index that simplifies locating objects by hash. The object store consists of one or more local files packed with objects, and thus the index contains file/offset pairs. All of Ori's main structures are stored as files in the local file system (e.g., ext4, UFS, or NTFS).

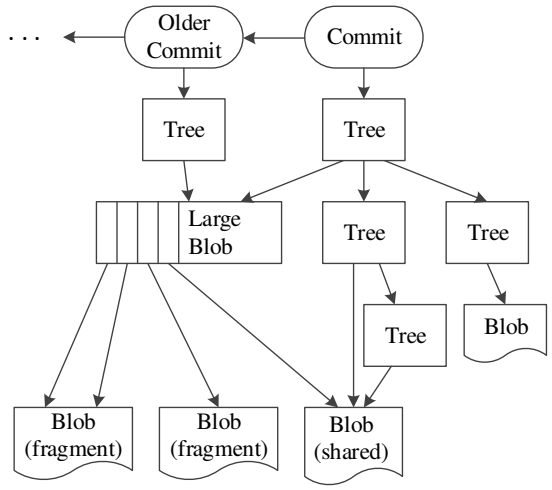

Figure 3: Object model in Ori including the chain of Commit objects corresponding to snapshots, Tree objects representing directories, and both LargeBlobs and Blobs representing files.

All objects in a repository are reference-counted for reclamation. Because directories are acyclic, unreachable objects always end up with a reference count of zero. To minimize reference-tracking overhead, Ori only counts unique, direct references from one object to another. For example, if the same Tree object is reachable from two snapshots, Ori only increments the reference count of the Tree object itself, not its children. Reference counts are local to a single repository and get updated incrementally as part of the synchronization process.

3.2 Grafting: Inter-file system copy

Grafting enables two-way synchronization between file systems. The operation copies a set of files or directories between two file systems (or possibly two locations in the same file system) while preserving the cross-file-system history. The history of the graft will be integrated with the history of the local file system and accessible through the *log* command. Many other commands work as usual, with a notable exception that automatic synchronization is not supported.

When a file/directory path S (in file system FS_S) is grafted onto a file/directory path D in another file system (FS_D), it creates a special graft commit record in FS_D. Specifically, such a graft commit contains the following extra fields, not present in regular commit records:

- *graft-fsid*: UUID of FS_S
- *graft-path*: Source pathname S
- *graft-commit*: Hash of the original commit in FS_S
- *graft-target*: Destination pathname D

The grafting algorithm takes a set of commits in FS_S and makes them a part of the history of FS_D. At a high level, these commits provide a series of changes to be applied to the head of FS_D at directory D. FS_D imports a full history of changes to FS_S, from the first commit before S existed to the last modification of FS_S. This imported history may indicate that S was previously grafted from a third file system $FS_{S'}$, in which case the graft commit will record $FS_{S'}$'s *graft-fsid* and *graft-path*, rather than that of FS_S (unless S' itself was grafted from somewhere else, and so forth).

Though FS_D incorporates the full history of commit objects from FS_S, it need not store the entire contents of every snapshot in that history. Only the contents of the grafted file or directory are stored. Each use of the grafting mechanism transfers data in a single direction so that users can control when they may need to merge with someone else's changes.

An important use case is when a group of users desire to share files by synchronizing a directory amongst themselves. Here a grafted snapshot might be grafted again multiple times as changes are propagated between machines. For this use case to work, all grafts must maintain the original source of the commit. This lets Ori correctly identify grafted snapshots and avoid accidentally creating branches depending on how changes are pulled.

By default, grafting requires full access to the source file system by the destination. To support remote grafting without giving access to the entire file system, the source must explicitly export a graft (similar to an NFS export). The source creates an exported history that others will see and import. The destination file system will graft the export onto the local repository.

We note that access control for remote grafts is performed at the granularity of exported file system mount points. If one has access to the exported file system, then one can access all files within it. Individual file permissions are ignored. Ori cannot be used directly in a traditional NFS-like multi-user setting, where a single export has different file permissions on a per file and per user basis. We currently have no mechanism to enforce such a policy. It is still possible, though clunkier, to create multiple grafts—one for each different user—to achieve similar use cases.

3.3 Replication

The *ori replicate* and *pull* commands are used to initiate replication and initiate a unidirectional synchronization between multiple repositories of the same file system. Initiating replication creates an empty local repository and then populates it with data mirroring the source

repository. Pull updates an existing repository and transfers only missing data, but does not merge new changes into a mounted file system. (Note this operation is closer to *git fetch* than *git pull*.) These commands support transport over SSH, which relies on existing authentication mechanisms, or HTTP. Automatic synchronization and management is left to *orisync*, described in Section 3.4, which synchronizes when devices are available.

The *replicate* operation creates a new replica of a file system when given a path to a source replica (local or remote). It works by first creating and configuring an empty repository. Next, it retrieves the hash of latest commit from the source and saves it for later use. It then scans the source for all relevant objects and transfers them to the destination. The set of relevant objects depends on whether the new instance is a full replica (including history) or shallow replica. Finally, when the transfer is complete, the new file system head is set to point to the latest commit object that was transfered.

Several important operations must occur during the process. First, metadata, object reference counts, and the file system index must be maintained. The index is updated as objects are stored in the local repository. Large portions of the index may be updated at once to reduce disk seeks. The computation of reference counts is done incrementally, as objects are received, and requires parsing Commit, Tree, and LargeBlob objects. Reference counts are updated on disk in a single write after all data has arrived.

A major performance concern is how we transfer and store objects on the disk. When the destination requests a set of objects, the source transmits the objects in the same order they are stored in packfiles, along with hints for when to break packfile boundaries. This is the network equivalent of the vectored read operation that orders and coalesces reads. This maintains spacial/temporal locality of snapshots by keeping objects of the same snapshot near one another. This heuristic improves source read performance during the operation, and maintains read performance for the destination. It also avoids breaking the optimizations on LargeBlob objects discussed in Section 3.1.

The unidirectional synchronization operation is used by users or *orisync* to propagate changed data and history between two file systems. The operation is similar initial replication and benefits from the same optimizations, with a few additions. Rather than scan all objects from the source, it scans the graph of Commit objects to identify the required file system snapshots that need to be migrated over to the destination. As part of the operation, we avoid asking for objects that the destination already has. In addition, metadata is updated incrementally, so as not to walk all objects in the file system.

After manually executing a synchronization through the *pull* command, either the *checkout* or *merge* commands must be used to make changes visible in the local file system. *orisync* does this automatically and thus these operations are only useful when manual synchronization is used.

3.3.1 Distributed Fetch

The *distributed fetch* optimization helps Ori transfer data by avoiding low bandwidth and high latency links, and uses nearby hosts as a cache for requested data. During replication operations Ori identifies the nearby instances using mDNS/Zeroconf and statically configured hosts. Ori contacts hosts in order of latency as an approximation of network quality. Before transferring objects from the original source, the algorithm attempts to use nearby hosts by asking for objects by hash.

There are a few security concerns that we address. Foremost, the current design uses cryptographically strong hashes and verifies them for all objects retrieved from nearby hosts as a safe-guard against tampering. Users can disable this feature or restrict it to static hosts to prevent any leakage of data hashes. This would prevent random hosts from attempting to identify what a user has based on known hashes of files (e.g., a known file downloaded from the Internet).

3.3.2 Background Fetch

Recall that Ori's background fetch optimization allows operations such as *replicate*, *pull*, and *graft*, to complete the bulk data transfer in the background while making the file system immediately available. If a process accesses files or directories that are not yet locally replicated, it blocks and *orifs* moves the needed contents to the head of queue, ahead of other less important remaining background operations. Any modified data can either be propagated manually with a *push* command or automatically at an interval configured with *orisync*.

Background fetch has three modes: background data transfer, on-demand with read caching, and without read caching. The *–ondemand* flag of *orifs* enables the background fetch feature and fetches objects on-demand without performing any background data transfers, useful for temporary access. The *–nocache* flag disables caching objects read from a remote host.

Supporting background operations requires splitting a task into two stages. First, completing any required synchronous tasks—e.g., checking for merge conflicts—then allowing users to proceed while pulling missing data on-demand.

To initiate replication with background fetching, Ori first connects to a remote host. It then creates an empty repository with the same file system ID on the local host.

Next, it adds the remote host to the local file system's peer list and marks it as an on-demand target. Finally, it sets the local file system's head version to match that of the remote. At this point, the file system can be mounted. Whenever an object that does not exist locally is accessed, Ori queries peers for the object. Objects received from peers are stored based on which of the three modes was selected. The mode also determines whether to run a background thread that proactively fetches all objects in the file system history.

Background fetch is generalized to support other operations such as *pull* and *merge*. In this case, Ori must complete a few tasks such as pulling all Tree objects in the latest commit and attempting a merge. Conflicting files will also be downloaded in order to attempt an auto-merge algorithm. Once complete, the background fetch optimization can continue to pull any missing objects (newly created files or modified files without merge conflicts) and previous history in the background or on-demand.

Background fetch exposes users to additional failure scenarios. When remote hosts become unavailable, then the current implementation will report that failure to the user if it fails to find an object locally. We do attempt to connect to all on-demand peers that are available. This situation does not occur without background fetch, as all normal Ori operations wait for completion to declare success or roll back. This failure scenario is comparable to that of NFS. One benefit over NFS is that files and directories that have already been accessed will continue to function without connectivity, and new files can be created without needing remote access.

3.4 Automatic Management

There are two automated tasks in Ori: automatic snapshots and automatic synchronization between devices, which includes their discovery. Automatic snapshots are needed so people can use Ori as a traditional file system, without having to run extra commands or explicitly take snapshots, but still get the benefits of versioning and being able to look at old files. These implicit snapshots are time-based and change-based, and are part of the permanent history. These snapshots may be reclaimed based on a policy, e.g., keep daily snapshots for a month and weekly snapshots for a year. Another policy is managing external drives as long-term archival stores and moving snapshots to these drives.

Ori supports automatic synchronization and discovery among devices. This functionality is delegated to the *orisync* agent. Users configure a cluster name and a shared key. The agent then periodically broadcasts encrypted announcements containing information about the repositories held on each machine. Agents running on other devices owned by the user can decrypt these announcements and initiate file system synchronizations. Users may also specify statically configured repositories/hosts so that devices can synchronize against a hosted peer over the Internet.

Each announcement contains a list of repositories that include the file system ID (UUID), path, and file system head. When receiving an announcement the agent checks against a list of locally registered repositories for matching file system IDs, which are the same across all instances of a file system. If the repositories have different revisions it can exchange snapshots and resolve conflicts to ensure repositories have caught up with each other. We assume that users are typically working on a single device at a time, so periodic synchronization between devices prevents stale data from being accessed on another machine. The maximum lag in synchronization is controlled by the announcement interval, which is about five seconds.

3.5 Space Reclamation Policy

Ori uses all the space allocated to it. In general Ori's space will have three parts: the latest version of the file system holding current data, historical versions, and free space. If the disk is large or the file system is new there will be plenty of free space on disk. Over time this will be filled with historical data. Eventually, the free space will be used up and historical data will need to be removed by Ori to create space for new files.

The eviction policy is to delete older snapshots, which makes the history sparser as it ages. The basic premise is that users care more about data in the past few weeks, e.g., recovering an accidentally deleted file, but care much less about the day-to-day changes they made a year ago. The policy is applied in order of the rules below, but in reverse historical order. That is Ori will look for the oldest daily snapshots past a week to delete before deleting any weekly snapshots. The deletions occur in the following order:

1. Temporary FUSE snapshots (never synchronized).
2. Daily snapshots past a week.
3. Weekly snapshots past a month.
4. Monthly snapshots past a year (keep quarterly).
5. Quarterly snapshots past a year (keep annual).
6. Annual snapshots.

3.6 Replication and Recovery

Ori focuses on a replication paradigm where data is present on all user devices. Devices can be configured in two modes: default or carrier. Carrier devices (like

mobile phones) will first replicate incremental snapshots needed to bring other devices up to speed, and then proceed to replicate the latest full snapshot (for availability) if space permits. The default policy instead is used on devices with ample space (like desktops) and it first replicates the latest snapshot (so one can start working) followed by historical data.

Using these replicas Ori can recover from silent disk corruption by connecting to other devices and retrieving objects by their hash. The data model uses object hashes for the dual purpose of addressing and verifying file system integrity. We can identify corrupted data and attempt recovery by restoring corrupted objects from other devices. The automatic repair functionality utilizes the same background fetch mechanism to pull objects from registered peers.

3.7 Tamper Resistance

Our data model, like that of Git, forms a Merkle tree in which the Commit forms the root of the tree pointing to the root Tree object and previous Commit object(s). The root Tree object points to Blobs and other Tree objects that are recursively encapsulated by a single hash. Ori supports signed commit objects, by signing the serialized Commit object with a public key. The Commit object is then serialized again with the key embedded into it, thus forcing subsequent signatures to encapsulate the full history with signatures. To verify the signature we recompute the hash of the Commit object omitting the signature and then verify it against the public key.

Note that Ori's design allows users to delete data reference by older snapshots, but always retains commit objects for verifying history. This means that the Blobs, LargeBlobs, and Tree objects will be reclaimed when a snapshot is deleted, but the commit object of a snapshot will remain. This allows us to poke holes in history and still be able to access older snapshots and verify them. This differs from Git's Merkle trees where previous states are always available.

4 Implementation

Ori currently runs on Linux, Mac OS X, and FreeBSD. It consists of approximately 21,000 lines of C++ code including 4,000 lines of headers. Ori is built in a modular way, making it simple to add functionality to the core file system. For example, *background fetch* is less than 100 lines of code. Another example is a standalone tool to backup to Amazon S3 (online storage) built using libs3, an open source Amazon S3 client. This entire example tool consists of less than 800 lines of code and headers.

4.1 FUSE Driver

We implemented Ori using FUSE, a portable API for writing file systems as user-space programs. FUSE forwards file system operations through the kernel and back into the user-space Ori driver. FUSE increases portability and makes development easier, but also adds latency to file system operations and makes performance sensitive to operating system scheduling behavior.

In response to file system read requests, Ori fetches data directly from packfiles. This is more space-efficient than Git, which provides access through a "checked-out" working directory containing a second copy of the latest repository tree. For write support, *orifs* maintains a temporary flat tree in memory, containing file and directory metadata, while the file data is stored in temporary files. We call this temporary file store the *staging area*.

The FUSE implementation provides convenient access to explicit snapshots and periodic snapshotting. Users have access to snapshots through a `.snapshot` directory located at the root mount point. Periodic snapshots are made in what can be thought of as a FUSE branch. We use Ori's metadata structure to flag FUSE commits for reclamation. When a garbage collection cycle is run, which occurs rarely, we can delete temporary FUSE commits. This may entail rewriting packfiles. Periodic snapshotting also speeds up permanent snapshots as we lack a true copy-on-write (COW) backend (because the staging area is outside of the packfiles).

Snapshotting works by copying data from the staging area to a packfile. When creating a permanent snapshot, we create a new commit object and any new or modified Trees and Blobs. This requires Ori to to read, compress, deduplicate, and write objects from the staging area into the repository. A COW file system would avoid some of the read and write overhead by placing data in a temporary area within the repository or in memory (rather than packfiles). This performance improvement could be addressed by implementing known techniques, but would require more frequent snapshotting and garbage collection. Our current design avoids a lot of the complexity of building a garbage collector, as we expect the current collector to run infrequently.

Currently the FUSE file system supports almost all standard POSIX functionality except hard links. Of course, a common usage of hard links is to deduplicate data, which Ori already handles transparently even across files. Ori does not store link counts for directories; we emulate them in *orifs*, which adds overhead when `stat`ting a directory (as the directory must be read to determine the number of subdirectories).

Benchmark	ext4	Ori	loopback
Create	8561±28%	6683±7%	6117±12%
Delete	4536±25%	1737±25%	3399±14%
Stat	14368	11099	10567
MakeDir	17732	10040	12197
DeleteDir	4402±8%	7229	3379±7%
ListDir	13597	6351	5717

Table 2: Filebench microbenchmark results for create, delete, and stat of files, as well as make, delete, and list of directories. Are results are in operations per second.

5 Evaluation

We evaluate Ori in two main settings: as a general purpose local file system and as a distributed file system.

The tests were run on an 8-core, 16-thread machine with two Intel Xeon E5620 processors clocked at 2.4 GHz and 48 GiB of RAM running Ubuntu Linux 3.8.0-26. Although Ori and FUSE are multithreaded, we limited the memory and CPU using Linux boot time flags to 4 GiB and a single processor. This serves the dual purpose of providing a realistic desktop configuration and reducing variability in the results (due to the uniprocessor configuration). The machine had a 3 TB 7200 RPM magnetic disk and a 120 GB Intel X25-M SSD, both connected via SATA. In addition, we used another machine with identical hardware that was connected with Gigabit Ethernet. For wide area network tests we used a host running FreeBSD 9.1 powered by an Intel Core 2 Duo E8400 processor running at 3 GHz with a ZFS RAIDZ1 (software RAID 5) spanning four 5400 RPM green drives.

We report SSD numbers for all local tests, as this demonstrated the file system bottlenecks better. The magnetic disk's latency hides most of the file system overhead and FUSE latency. We ran tests five times reporting the mean and state the standard deviation only when it exceeds 5% of the mean. For a few benchmarks, the SSD had a much higher variance than the magnetic disk, but the results are meaningful as only one or two runs were outliers.

5.1 Microbenchmarks

We start by examining the cost of common file system operations like creating and deleting files and directories. We use Filebench [8] for this and the results are shown in Table 2. We compare Ori (using FUSE) against ext4. As a baseline and to isolate the cost of FUSE itself, we also compare against the FUSE loopback driver that simply forwards all file system operations to the kernel.

Benchmark	ext4	Ori	loopback
16K read	284,078	237,399	236,762
16K write	108,685	106,938	107,053
16K rewrite	71,664	64,926	63,674

Table 3: Bonnie++ benchmark result averaged over five rounds taken on the SSD device. Read-/write/rewrite units are KiB/sec.

The file creation benchmark creates a file and appends data to the file. In this benchmark, Ori and the loopback perform about 27% and 21% slower than ext4, mostly due to the overhead of FUSE that requires crossing between user and kernel space multiple times. Ori is slower than FUSE loopback because it additionally needs to journal the file creation. Another reason Ori is slower is because the benchmark creates 400k files in a directory tree, but Ori creates all the files in one directory (the staging area), putting pressure on the ext4 directory code.

The file stat benchmark creates 100k files and then measures the performance of calling `fstat` on random files. Ori performs 5% better than the loopback. In all cases the data suggests `fstat` information, stored in the inodes, is cached. Thus, Ori has lower latency than the loopback driver because it caches metadata inside it's own structures, rather than relying on the underlying file system's metadata.

The directory creation and deletion operations in Ori are not directly comparable to ext4. These operations in Ori happen on data structures resident in memory and only a journal entry is written on the disk itself. Ori also outperforms ext4 in the directory list benchmark because it caches directory structures that contain modified files or directories.

We now examine the cost of file IO operations using the Bonnie++ [9] benchmark. Table 3 shows the results of file reads and writes. Compared to ext4, Ori's user-space implementation incurs a penalty of at most 20% on reads and 2% on writes. Ori performs nearly the same as loopback, however, since both perform similar actions and forward IOs to the underlying file system.

5.2 Filesystem macrobenchmarks

Filebench provides some synthetic macrobenchmarks, whose results we report in Table 4. A few benchmarks had high variance because of calls to `delete` and `fsync`. Filebench's macrobenchmarks function by populating a directory structure with files randomly and then operating on those files. It should immediately stand out that Ori and the FUSE loopback file system performed similarly except on the varmail benchmark.

One reason Ori performs well is that the directory

Benchmark	ext4	Ori	loopback
fileserver	2919±23%	2235±24%	2258±27%
webserver	10990	8250	8046
varmail	3840±10%	1961±17%	8376
webproxy	10689	6955	7281
networkfs	603	612	570±13%

Table 4: Filebench macrobenchmarks for ext4, Ori, and FUSE loopback driver. The numbers are listed in operations per second.

structure is largely cached in memory as a consequence of optimizing snapshot creation. Moreover, Ori's staging area consists of a single directory in which temporary files record writes since the previous snapshot. Such a flat structure incurs fewer expensive multi-level directory lookups in the underlying ext4 file system.

The varmail benchmark is unique in that it is the only benchmark that is calling `fsync` operations. The FUSE loopback ignores `fsync` operations, which is why the varmail benchamark is faster than even the ext4 runs. Ori instead `fsyncs` the corresponding file in the staging area to give the same guarantees as the local file system. As this operation is synchronous, performance is much worse when calling `fsync` through a FUSE file system.

5.3 Snapshot and Merge Performance

We examine the utility of the snapshot feature by comparing how quickly Ori can take snapshots. In this simple test, we extracted the zlib source tree into an Ori file system and took a snapshot. The snapshot time includes the time required to read all of the staging area. Then we compress, deduplicate, and store the data from the staging area into our file system store. We also have to flatten the in-memory metadata state and transform it into a series of Tree objects and one Commit object. This entire process for 3 MBs of source files took 62 ms on a SSD.

To put this number in perspective, we compare it to Git and ZFS. It took 121 ms to add and commit the files into Git using the same SSD. Taking a ZFS snapshot (on the FreeBSD host) took approximately 1.3 s. These tests are not ideal comparisons, but strongly suggest that Ori's performance is more than acceptable.

Merge performance is largely dependent on the number of files modified. We measured the total time required to merge two file systems containing one change in each (not including pull time). The merge completed in approximately 30 ms.

Data Set	Raw Size	Ori	Git
Linux Snapshot	537.6M	450.0M	253.0M
Zlib	3.044M	2.480M	2.284M
Wget	13.82M	12.46M	6.992M
User Documents	4.5G	4.6G	3.4G
Tarfiles	2.8G	2.3G	2.3G

Table 5: Repository size in Ori and Git.

5.4 Ori as a Version Control System

Ori's history mechanism is essentially a version control system, which can be used as an alternative to Git. The key differences between the two are that Git uses a combination of whole file deduplication, differential compression, and compression to save space (at the cost of using more CPU), whereas Ori simply relies on subfile deduplication and compression. We wished to compare the two approaches. However, Ori's FUSE interface gives it an unfair advantage (by avoiding the need to create a checked-out tree). Hence, to make the comparison fair, we disabled the FUSE driver and used Ori as a standalone check-in/check-out tool. In this setting, we found that Ori has faster adds and commits (by 64%). Similarly, clone operations in Ori are 70% faster than Git. However, Git's more expensive compression does save more space.

The space savings that occur thanks to deduplication and compression, where identical file chunks are stored only once, are shown in Table 5. The table shows the amount of disk space needed when storing files in Ori compared to their original size. Git saves even more space thanks to its differential compression, which came at the cost of slower commits and clones. Ori also has the ability to delete snapshots, which Git does not support. The lack of differential compression avoids the extra decompression and recompression steps that would otherwise be necessary when deleting snapshots.

5.5 Network Performance

We now examine some of the networked aspects of Ori. We took several measurements of Ori's remote file access both in a Gigabit local area network (LAN) setting and over a WAN. In the WAN scenario, we were uploading from a remote machine that had a 2 Mbps up and 20 Mbps down link with a 17 ms round-trip time (RTT). In all cases we used magnetic disks rather than SSDs, which also means disk seeks due to deduplication impacted performance.

5.5.1 Network Throughput

Table 6 shows how long it took to migrate a 468 MB home directory containing media, source code, and user

	LAN		WAN	
	rsync	ori	rsync	ori
Time	9.5s	15s±6%	1753s	1511s
Sent	3MiB	5.4MiB	12.3MiB	13.3MiB
Rcv.	469MiB	405MiB	469MiB	405MiB
BW	49MiB/s	27MiB/s	267KiB/s	268KiB/s

Table 6: Network performance comparison to rsync in the LAN and WAN. We include the total time, megabytes sent and recieved (Rcv.), and bandwidth (BW).

documents over the LAN (left columns) and WAN. On the LAN, Both rsync and Ori are bandwidth limited by the disk. Ori cannot perform as fast as rsync because subfile deduplication induces some fragmentation in the IO workload that causes additional disk seeks. The fragmentation combined with the recreation of the repository on each run caused the higher standard deviation of 6%.

In the WAN scenario, the systems are bandwidth limited (2 Mbps uplink), and Ori's compression and deduplication allowed it to outperform rsync. More generally, the compression and deduplication allows Ori to outperform rsync as bandwidth falls below 49 MiB/s, which is equivalent to a very high-end wireless connection. Many mobile devices today have 802.11n with a single antenna, which caps bandwidth at 150 Mbps, far below the 49 MiB/s disk throughput. Thus, Ori can synchronize data faster, and our WAN measurement confirms Ori's protocol is capable of handling higher latency connections. We note that rsync has compression built into the protocol and it can perform better or worse than Ori depending on the compression level and bandwidth. All of Ori's deduplication and compression is from what already is achieved in the object store.

5.5.2 Ori as a Network File System

As a sample workload, we built zlib over the network, showing any replication, configuration, build, snapshot, and push times over both a LAN and WAN network. The compile output (object files) is stored on the networked file system so both read and write operations happen. The WAN scenario was conducted by pulling or mounting from the remote FreeBSD machine.

Table 7 shows the results for our LAN and WAN configurations. We ran the benchmark on NFS version 3, NFS version 4, Ori with replication, and Ori with background fetch enabled in the on-demand mode (i.e., prefetching disabled). In both the NFSv3 and NFSv4 benchmarks we used TCP. Notice that the failure semantics of these two systems are very different: A server crash or network interruption would block the NFS workloads, whereas Ori does not make data available to others until after the push. Nonetheless, this comparison is still meaningful.

The most interesting result is that even with background fetching, which is slower, Ori over the WAN slightly outperforms NFSv3 and NFSv4 over the LAN. If we compare the WAN numbers, Ori runs more than twice as fast as NFSv3 and NFSv4. While compiles over a remote network are not a common use case they provide a nice mixture of file and directory operations. A major reason Ori outperforms NFS is that it batches the writes into a single push operation at the end of the build.

The results also give us a measure of the performance impact of background fetching. The building time increased by approximately a factor of 1.3 and 1.2 in the LAN and WAN cases respectively. The WAN case has less overhead because the test becomes bandwidth limited, thus background fetching lowers end-to-end time. If we had enabled prefetching objects we would expect to see even lower results, but with more variability.

We can see how Ori performs with respect to the increase in network latency and bandwidth constraints. Latency has a smaller impact on the results when Ori becomes bandwidth limited. This is because the vectored read and write operations effectively pipeline requests to hide network latency (in addition to disk latency). In this test the clone and pull operations take between three and five synchronous requests to transfer all changes. This can vary widely depending on the tree and history depth. Ori also suffers from the additional latency associated with SSH (i.e., encryption and protocol overhead).

5.5.3 Distributed Fetch

Recall the distributed fetch feature uses nearby replicas (possibly of unrelated file systems) to fetch data through faster networks. We benchmark this aspect of Ori comparing a standard, non-distributed fetch of the Python 3.2.3 sources (2,364 files, 60 MB total) against two distributed fetch variants shown in Table 8. The source for all three scenarios was across a long-distance link (average latency 110 ms, up/down throughput 290/530 KB/sec). In the scenario labeled "Remote," no local Ori peers were present and all objects were pulled from the source. In the "Distributed" scenario, an exact copy of the repository was available on a local peer. In the "Partial Distributed" scenario, a similar repository (in this case, the Python 2.7.3 sources) was available on a peer.

Our results show that distributed fetch can be effective even in the last scenario, where only a distantly-related repository was available as a distributed fetch source. Ori was still able to source 26% of the objects from the related repository, decreasing total time spent by 22.7%. In the case of an identical repository, distributed fetch was 22 times faster than when distributed fetch is disabled.

	NFSv3		NFSv4		Ori		Ori on-demand	
Benchmark	LAN	WAN	LAN	WAN*	LAN	WAN	LAN	WAN
Replicate					0.49 s	2.93 s		
Configure	8.14 s	21.52 s	7.25 s	15.54 s	0.66 s	0.66 s	1.01 s	1.33 s
Build	12.32 s	33.33 s	12.20 s	28.54 s	9.50 s	9.55 s	11.45 s	12.77 s
Snapshot					0.19 s	0.19 s	2.72 s	3.37 s
Push					0.49 s	1.58 s	0.85 s	1.89 s
Total Time	20.45 s	54.85 s	19.45 s	44.07 s	11.33 s	15.30 s	16.04 s	19.34 s

Table 7: The configure and build times for zlib 1.2.7 over a LAN and WAN network for NFS, Ori, and Ori on-demand enabled (i.e., no prefetching). (*) The NFSv4 WAN numbers were taken with a host running Linux, since the FreeBSD 9.1 NFSv4 stack performed worse than NFSv3.

	Distributed	Partial Distributed	Remote
Time	7.75 s	132.05 s	170.79 s

Table 8: Time to pull a repository with a local peer available, a local peer that has some of the data, and from a remote peer.

6 Related Work

Ori borrows many ideas from past work on archival file systems, version control, and network file systems.

Distributed version control systems such as Git [3] and mercurial provide offline use. However, they are inadequate as general-purpose file systems, as they lack mechanisms to reclaim space and deal with large files. Git Annex [10] is one attempt to use Git for storing general data by using Git to manage symbolic links pointing into a shared storage system. Of course, this just pushes a lot of data management issues onto the shared storage system, for which one option is Amazon S3.

Another limitation of Git and related tools is the lack of instant access to remote files or transparent synchronization, even with good network connectivity. We are aware of concurrent efforts by Facebook to speed up Mercurial's clone command. However, without a FUSE driver to provide the illusion of a complete tree and fetch chunks on demand, we do not believe mercurial can ever compete with the near instant access provided by Ori's background fetch feature.

Network File System (NFS) [11], Common Internet File System (CIFS) [12, 13], and Apple Filing Protocol (AFP) [14] are all examples of network file systems designed for LANs. These allow instant access to remote files but lack offline use, versioning, or integrated backup. They also perform poorly over WANs.

AFS [15], Coda [16] and InterMezzo [17] are distributed file systems that use caching. Coda provides Ficus-like [18] offline access, but users must manually configure hoarding policies or risk missing files. Unlike Ori's peer-to-peer architecture, clients do not synchronize with one another; nor can a client replace a failed server, since clients do not contain complete replicas of the file system. Coda's architecture results in separate client and server implementations and a vastly larger code base than Ori.

Another source of complexity is that Coda aggressively resolves update conflicts in application-specific ways, violating traditional abstraction layering. By contrast, Ori simply exposes unresolvable conflicts to the user. However, because Ori has history, the user can understand where and how files relate. In particular, Ori always supplies a "merge-base" ancestor for three-way merging, which is not possible without history in Coda.

JetFile [19] is a peer-to-peer distributed file system based around a multicast transport, though it does not fully support offline use. GlusterFS [20] is another distributed peer-to-peer file system, designed to span a cluster of machines. Like Ori, GlusterFS is peer-to-peer and avoids any central point of failure. Unlike Ori, GlusterFS targets a completely different use case—large data sets too large to store on any single node. This leads to a very different design. For example, GlusterFS does not store history and has more problems recovering from split-brain conditions.

6.1 Other relevant storage systems

Eyo [21] is an application API for file synchronization and versioning between mobile devices. It shares several of Ori's goals, such as device transparency and versioning, but focuses on space-constrained devices. Eyo lacks atomic snapshots and integrity protection. The authors modified several applications to support Eyo and reported less than 10% of application source code needed to change. Being a general-purpose file system, Ori does not *require* any modifications to applications, though it provides an extended API with similar features to Eyo.

Cimbiosys [22] and EnsemBlue [23] are both replicated storage systems that do not require application

modification. Both systems replicate individual files and focus on allowing users to subset files for different devices. Ori presumes that embedded device storage capacities will continue to grow relative to the size of managed data, and hence primarily targets complete replication. Ori users can, however, subset files either by replicating only recent history, grafting a subtree into a new repository, or accessing data on-demand through the background fetch feature.

PRACTI [24] is a replication system that supports partial replication, arbitrary consistency, and topology independence. One of the applications built on PRACTI is a peer-to-peer distributed file system for WANs, which could be viewed as a more latency-tolerant version GlusterFS. Ori has topology independence, but is not intended for partial replication and exclusively supports eventual consistency. Ori's primary contributions center around the practicality of totally replicated file histories, which only loosely relates to PRACTI's contributions.

Dropbox [1] is a cloud-based data management system. Like Ori, it allows synchronization between devices and supports file sharing. Unlike Ori, it is centralized, has a single point of failure, and generally does not benefit from fast LAN-based peer-to-peer transfers. Dropbox offers limited version control that cannot be controlled by the user. There is no equivalent of grafting for users to compare histories. Most importantly, Ori leverages existing unused space on user's devices, while Dropbox wants users to rent additional storage in the Cloud.

Archival File Systems The Write Anywhere File Layout (WAFL) and ZFS are both file systems based around implementing copy-on-write from the superblock down to the individual files [25, 26]. WAFL and ZFS operate at the block layer, whereas Ori works on top of an existing file system at a file granularity.

The Elephant file system [27] allowed versioning on individual files. Unlike Ori, which works on the entire file system, in Elephant it is difficult to snapshot the entire file system at once.

Plan 9's fossil file system [28] stores data in a simple tree format with the file system layered on top of it. This is similar to Ori, which stores information in a set of simple containers, except that Ori has tighter integration between the object interface and the file system. Moreover, fossil is a client-server system in which history is not replicated on clients.

Wayback [29] and Versionfs [30] are both archival file systems centered around enabling users to control and access versioning. Wayback is implemented as a user-level (FUSE) file system, while Versionfs is built as a kernel-level stackable file system. Both systems are designed for single machines. Neither provides the non-versioning functionality that Ori provides, and neither provides integrity protection.

Apple's Time Machine [31] and Microsoft Windows 8 File History [32] are backup solutions that provide users a way to visualize the history of directories and files. Unlike Ori, these systems are stored on external or network drives and are not mobile.

Network File Systems Ori borrows deduplication techniques from LBFS [7] and Shark [33]. In particular, Ori uses a distributed pull mechanism that searches for other nearby copies of file fragments in a manner similar to Shark.

SFSRO [34] and SUNDR [35] both use Merkle trees [36] to represent and verify file system data structures. To verify consistency in the presence of an untrusted server, SUNDR described a "straw-man" design that logged all file system operations, an impractical approach for SUNDR. Interestingly, because Ori supports compound updates, tolerates conflicts, and keeps a complete log anyway, it effectively implements SUNDR's straw man. This opens the possibility of using Ori to achieve a form of fork linearizability in a distributed setting. (Critical reads must be logged as empty commits to record where in the order of commits they occurred.)

7 Conclusion

Most people spend their time working on at most a few tens of Gigabytes of documents, yet the knee in the cost curve for disks is now at 3 TB. Hence, it is time to rethink the decades-old file system abstraction and ask how all this storage capacity can better serve user needs. Specifically, the popularity of cloud storage services demonstrates a demand for several management features—backup, versioning, remote data access, and sharing.

Ori is a new file system designed to meet these needs. Ori replicates file systems across devices, making it trivial to recover from the loss of a disk. It provides both explicit and implicit versioning, making it easy to undo changes or recover from accidentally deleted files. Ori provides the illusion of instant replication and synchronization, even as data transfers proceed in the background. Finally, Ori introduces grafting, a technique that preserves file histories across file systems, simplifying the process of reconciling changes after sharing files.

Benchmarks show Ori's performance to be acceptably close to other file systems, and that Ori significantly outperforms NFS over a WAN. The implementation currently supports Linux, Mac OS X, and FreeBSD. Source code is available at `http://ori.scs.stanford.edu/`.

Acknowledgements

We would like to thank the SOSP program committee. We thank Adam Belay, Emre Celebi, Tal Garfinkel, Daniel Jackoway, Ricardo Koller, Amit Levy, Deian Stefan, and Edward Yang for their helpful comments. Stan Polu and Brad Karp contributed to the inspiration for Ori through a masters project and several discussions. This work was funded by DARPA CRASH grants N66001-10-2-4088 and N66001-10-2-4089, as well as by a gift from Google.

References

[1] Dropbox Inc. Dropbox - simplify your life. `http://www.dropbox.com/`, September 2012.

[2] R. Rodrigues and C. Blake. When multi-hop peer-to-peer routing matters. In *3rd International Workshop on Peer-to-Peer Systems*, La Jolla, CA, 2004.

[3] Inc. Software Freedom Conservancy. Git. `http://www.git-scm.com/`, September 2012.

[4] Yupu Zhang, Chris Dragga, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. *-box: Towards reliability and consistency in dropbox-like file synchronization services. In *Proceedings of the The Fifth Workshop on Hot Topics in Storage and File Systems*, HotStorage '13, Berkeley, CA, USA, 2013. USENIX Association.

[5] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the Conference on File and Storage Technologies*, FAST '02, pages 89–101, Berkeley, CA, USA, 2002. USENIX Association.

[6] Mercurial scm. `http://mercurial.selenic.com/`, September 2012.

[7] Athicha Muthitacharoen, Benjie Chen, and David Mazières. A low-bandwidth network file system. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 174–187, New York, NY, USA, 2001. ACM.

[8] Filebench. `http://sourceforge.net/apps/mediawiki/filebench/index.php`, 2013.

[9] Russell Coker. Bonnie++ now at 1.03e. `http://www.coker.com.au/bonnie++/`, September 2012.

[10] Joey Hess. git-annex. `http://git-annex.branchable.com/`, September 2012.

[11] B. Nowicki. NFS: Network File System Protocol specification. RFC 1094 (Informational), March 1989.

[12] Microsoft Corporation. Common internet file system (cifs) protocol specification. `http://msdn.microsoft.com/en-us/library/ee442092.aspx`, September 2012.

[13] Jose Barreto. Smb2, a complete redesign of the main remote file protocol for windows. `http://blogs.technet.com/b/josebda/archive/2008/12/05/smb2-a-complete-redesign-of-the-main-remote-file-protocol-for-windows.aspx`, December 2008.

[14] Gursharan S. Sidhu, Richard F. Andrews, and Alan B. Oppenheimer. *Inside AppleTalk*. Addison-Wesley, Boston, 1990.

[15] Mahadev Satyanarayanan. Scalable, secure, and highly available distributed file access. *Computer*, 23(5):9–18, 20–21, May 1990.

[16] Mahadev Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Trans. Comput.*, 39(4):447–459, April 1990.

[17] P.J. Braam, M. Callahan, and P. Schwan. The intermezzo file system. *The Perl Conference 3*, August 1999.

[18] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, and Gerald Popek. Resolving file conflicts in the ficus file system. In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1*, USTC'94, pages 12–12, Berkeley, CA, USA, 1994. USENIX Association.

[19] Björn Grönvall, Ian Marsh, and Stephen Pink. A multicast-based distributed file system for the internet. In *Proceedings of the 7th workshop on ACM SIGOPS European workshop: Systems support for worldwide applications*, EW 7, pages 95–102, New York, NY, USA, 1996. ACM.

[20] GlusterFS. `http://www.gluster.org/`.

[21] Jacob Strauss, Justin Mazzola Paluska, Chris Lesniewski-Laas, Bryan Ford, Robert Morris, and Frans Kaashoek. Eyo: device-transparent personal storage. In *Proceedings of the 2011 USENIX*

conference on USENIX annual technical conference, USENIXATC'11, pages 35–35, Berkeley, CA, USA, 2011. USENIX Association.

[22] Venugopalan Ramasubramanian, Thomas L. Rodeheffer, Douglas B. Terry, Meg Walraed-Sullivan, Ted Wobber, Catherine C. Marshall, and Amin Vahdat. Cimbiosys: a platform for content-based partial replication. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 261–276, Berkeley, CA, USA, 2009. USENIX Association.

[23] Daniel Peek and Jason Flinn. Ensemblue: integrating distributed storage and consumer electronics. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 219–232, Berkeley, CA, USA, 2006. USENIX Association.

[24] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. Practi replication. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 5–5, Berkeley, CA, USA, 2006. USENIX Association.

[25] Dave Hitz, James Lau, and Michael Malcolm. File system design for an nfs file server appliance. In *Proceedings of the USENIX Winter 1994 Technical Conference on USENIX Winter 1994 Technical Conference*, WTEC'94, pages 19–19, Berkeley, CA, USA, 1994. USENIX Association.

[26] J. Bonwick and B. Moore. Zfs: The last word in file systems. `http://hub.opensolaris.org/bin/download/Community+Group+zfs/docs/zfslast.pdf`, September 2012.

[27] Douglas J. Santry, Michael J. Feeley, Norman C. Hutchinson, and Alistair C. Veitch. Elephant: The file system that never forgets. In *Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, HOTOS '99, pages 2–, Washington, DC, USA, 1999. IEEE Computer Society.

[28] S. Quinlan, J. McKie, and R. Cox. Fossil, an archival file server. `http://www.cs.bell-labs.com/sys/doc/fossil.pdf`, 2003.

[29] Brian Cornell, Peter A. Dinda, and Fabián E. Bustamante. Wayback: a user-level versioning file system for linux. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, ATEC '04, pages 27–27, Berkeley, CA, USA, 2004. USENIX Association.

[30] Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, and Erez Zadok. A versatile and user-oriented versioning file system. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 115–128, Berkeley, CA, USA, 2004. USENIX Association.

[31] Wikipedia. Time machine (mac os). `http://en.wikipedia.org/wiki/Time_Machine_(Mac_OS)`, March 2013.

[32] Steve Sinofsky. Protecting user files with file history. `http://blogs.msdn.com/b/b8/archive/2012/07/10/protecting-user-files-with-file-history.aspx`, March 2013.

[33] Siddhartha Annapureddy, Michael J. Freedman, and David Mazières. Shark: scaling file servers via cooperative caching. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 129–142, Berkeley, CA, USA, 2005. USENIX Association.

[34] Kevin Fu, M. Frans Kaashoek, and David Mazières. Fast and secure distributed read-only file system. *ACM Transactions on Computer Systems*, 20(1):1–24, February 2002.

[35] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (sundr). In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 9–9, Berkeley, CA, USA, 2004. USENIX Association.

[36] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology—CRYPTO '87*, volume 293 of *Lecture Notes in Computer Science*, pages 369–378, Berlin, 1987. Springer-Verlag.