CalvinFS: Consistent WAN Replication and Scalable Metadata Management for Distributed File Systems

Alexander Thomson
Google[†]
agt@google.com

Daniel J. Abadi
Yale University
dna@cs.yale.edu

Abstract

Existing file systems, even the most scalable systems that store hundreds of petabytes (or more) of data across thousands of machines, store file metadata on a single server or via a shared-disk architecture in order to ensure consistency and validity of the metadata.

This paper describes a completely different approach for the design of replicated, scalable file systems, which leverages a high-throughput distributed database system for metadata management. This results in improved scalability of the metadata layer of the file system, as file metadata can be partitioned (and replicated) across a (shared-nothing) cluster of independent servers, and operations on file metadata transformed into distributed transactions.

In addition, our file system is able to support standard file system semantics—including fully linearizable random writes by concurrent users to arbitrary byte offsets within the same file—across wide geographic areas. Such high performance, fully consistent, geographically distributed files systems do not exist today.

We demonstrate that our approach to file system design can scale to billions of files and handle hundreds of thousands of updates and millions of reads per second—while maintaining consistently low read latencies. Furthermore, such a deployment can survive entire datacenter outages with only small performance hiccups and no loss of availability.

1 Introduction

Today's web-scale applications store and process increasingly vast amounts of data, imposing high scalability requirements on cloud data storage infrastructure.

The most common mechanism for maximizing availability of data storage infrastructure is to replicate all data storage across many commodity machines within a datacenter, and then to keep hot backups of all critical system components on standby, ready to take over in case the main component fails.

However, natural disasters, configuration errors, hunters, and squirrels sometimes render entire datacenters unavailable for spans of time ranging from minutes to days [13, 15]. For applications with stringent availability requirements, replication across multiple geographically separated datacenters is therefore essential.

For certain classes of data storage infrastructure, significant strides have been made in providing vastly scalable solutions that also achieve high availability via WAN replication. For example, replicated block stores, where blocks are opaque, immutable, and entirely independent objects are fairly easy to scale and replicate across datacenters since they generally do not need to support multi-block operations or any kind of locality of access spanning multiple blocks. NoSQL systems such as Cassandra [12], Dynamo [8], and Riak [2] have also managed to achieve both scale and geographical replication, albeit through reduced replica consistency guarantees. Even some database systems, such as the F1 system [19] which Google built on top of Spanner [7] have managed to scalably process SQL queries and ACID transactions while replicating across datacenters.

Unfortunately, file systems have not achieved the same level of scalable, cross-datacenter implementation. While many distributed file systems have been developed to scale to clusters of thousands of machines, these systems do not provide WAN replication in a manner that allows continuous operation in the event of a full datacenter failure due to the difficulties of providing expected file system semantics and tools (linearizable operations, hierarchical access control, standard command-line tools, etc.) across geographical distances.

[†]This work was done while the author was at Yale.

In addition to lacking support for geographical replication, modern file systems—even those known for scalability—utilize a fundamentally unscalable design for metadata management in order to avoid high synchronization costs necessary to maintain traditional file system semantics for file and directory metadata, including hierarchical access control and linearizable writes. Hence, while they are able to store hundreds of petabytes of data (or more) by leveraging replicated block stores to store the *contents* of files, they rely on an assumption that the average file size is very large, while the number of unique files and directories are comparatively small. They therefore run into problems handling large numbers of small files, as the system becomes bottlenecked by the metadata management layer [20, 27].

In particular, most modern distributed file systems use one of two synchronization mechanisms to manage metadata access:

- A special machine dedicated to storing and managing all metadata. GFS, HDFS, Lustre, Gluster, Ursa Minor, Farsite, and XtreemFS are examples of file systems that take this approach [10, 21, 18, 1, 3, 4, 11]. The scalability of such systems are clearly fundamentally bottlenecked by the metadata management layer.
- A shared-disk abstraction that coordinates all concurrent access. File systems that rely on shared disk for synchronization include GPFS, PanFS, and xFS [17, 26, 22]. Such systems generally replicate data across multiple spindles for fault tolerance. However, these typically rely on extremely low (RAID-local or rack-local) synchronization latencies between replicated disks in order to efficiently expose a unified disk address space. Concurrent disk access by multiple clients are synchronized by locking, introducing performance limitations for hot files [17]. Introducing WAN latency synchronization times into lock-hold durations would significantly increase the severity of these limitations.

In this paper, we describe the design of a distributed file system that is substantially different from any of the above-cited file systems. Our system is most distinguished by the metadata management layer which horizontally partitions and replicates file system metadata across a shared-nothing cluster of servers, spanning multiple geographic regions. File system operations that potentially span multiple files or directories are transformed into distributed transactions, and processed via a transaction scheduling and replication management layer of an extensible distributed database system in order to ensure proper coordination of linearizable updates.

Due to the uniqueness of our design, our system, which we call CalvinFS, has a different set of advantages and disadvantages relative to traditional distributed file systems. In particular, our system can handle a nearly unlimited number of files, and can support fully linearizable random writes by concurrent users to arbitrary byte offsets within a file that is consistently replicated across wide geographic areas—neither of which is possible in the above-cited file system designs. However, our system is optimized for operations on single files. Multiple-file operations require distributed transactions, and while our underlying database system can handle such operations at high throughput, the latency of such operations tend to be larger than in traditional distributed file systems.

2 Background: Calvin

As described above, we horizontally partition metadata for our file system across multiple nodes, and file operations that need to atomically edit multiple metadata elements are run as distributed transactions. We extended the Calvin database system to implement our metadata layer, since Calvin has proven to be able to achieve consistent geo-replicated and linear distributed transaction scalability to hundreds of thousands of transactions per second across hundreds of machines per replica, even under relatively high levels of lock contention [24]. The remainder of this section will provide a brief overview of Calvin's architecture and execution protocol.

A Calvin deployment consists of three main components: a transaction request **log**, a **storage layer**, and a **scheduling layer**. Each of these components provides a clean interface and implementations that can be swapped in and out. The log stores a global totally-ordered sequence of transaction requests. Each transaction request in the log represents a read-modify-write operation on the contents of the storage layer; the particular implementation of the storage layer plus any arguments logged with the request define the semantics of the operation. The scheduling layer has the job of orchestrates the (concurrent) execution of logged transaction requests in a manner that is equivalent to a deterministic serial execution in exactly the order they appear in the log.

For each of these three components, we describe here the specific implementation of the component that we used in the metadata subsystem of CalvinFS.

Log

The log implementation we used consists of a large collection of "front-end" servers, an asynchronously- replicated distributed block store, and a small group of "meta-log" servers. Clients append requests to the log by sending them to a front-end server, which batches it with other incoming requests and writes the batch to the distributed

block store. Once it is sufficiently replicated in the block store (2 out of 3 replicas have acked the write, say), the front-end server sends the batch's unique block id to a meta- log server. The meta-log servers, which are typically distributed across multiple datacenters, maintain a Paxos-replicated "meta-log" containing a sequence of block ids referencing request batches. The state of the log at any time is considered to be the concatenation of batches in the order specified by the "meta- log".

Storage Layer The storage layer encapsulates all knowledge about physical datastore organization and actual transaction semantics. It consists of a collection of "storage nodes", each of which runs on a different machine in the cluster and maintains a shard of the data. Valid storage layer implementations must include (a) read and write primitives that execute locally at a single node, and (b) a placement manager that determines at which storage nodes these primitives must be run with given input arguments. Compound transaction types may also be defined that combine read/write primitives and arbitrary deterministic application logic. Each transaction request that appears in the log corresponds to a primitive operation or compound transaction. Primitives and transactions may return results to clients upon completion, but their behavior may not depend any inputs other than arguments that are logged with the request and the current state of the underlying data (as determined by read primitives) at execution time.

The storage layer for CalvinFS metadata consists of a multiversion key-value store at each storage node, plus a simple consistent hashing mechanism for determining data placement. The compound transactions implemented by the storage layer are described in Section 5.1.

Scheduler

Each storage node has a local scheduling layer component (called a "scheduler") associated with it which drives local transaction execution.

The scheduling layer takes an unusual approach to pessimistic concurrency control. Traditional database systems typically schedule concurrent transaction execution by checking the safety of each read and write performed by a transaction immediately before that operation occurs, pausing execution as needed (e.g., until an earlier transaction releases an already-held lock on the target record). Each Calvin scheduler, however, examines a transaction before it begins executing at all, decides when it is safe to execute the *whole* transaction based on its read-write sets (which can be discovered automatically or annotated by the client), and then hands the transaction request to the associated storage node, which is free to execute it with no additional oversight.

Calvin's scheduler implementation uses a protocol called deterministic locking, which resembles strict two-phase locking, except that transactions are required to request all locks that they will need in their lifetimes atomically, and in the relative order in which they appear in the log. This protocol is deadlock- free and serializable, and furthermore ensures that execution is equivalent not only to *some* serial order, but to a deterministic serial execution in log order.

All lock management is performed locally by a scheduler, and schedulers track lock requests only for data that resides at the associated storage node (according to the placement manager). When transactions access records spanning multiple machines, Calvin forwards the *entire* transaction request to all schedulers guarding relevant storage nodes. At each participating scheduler, once the transaction has locked all local records, the transaction proceeds to execute using the following protocol:

1. **Perform all local reads.** Read all records in the transaction's read-set that are stored at the local storage node.
2. **Serve remote reads.** Forward each local read result from step 1 to every other participant.
3. **Collect remote read results.** Wait to receive all messages sent by other participants in step 2[1].
4. **Execute transaction to completion.** Once all read results have been received, execute the transaction to completion, applying writes that affect records in the local storage node, and silently dropping writes to data that is not stored locally (since these writes will be applied by other participants).

Upon completion, the transaction's locks are released and the results are sent to the client that originally submitted the transaction request.

A key characteristic of the above protocol is the lack of a distributed commit protocol for distributed transactions. This is a result of the deterministic nature of processing transactions—any failed node can recover its state by loading a recent datat checkpoint and then replaying the log deterministically. Therefore, double checking that no node failed over the course of processing the transaction is unnecessary. The lack of distributed commit protocol, combined with the deadlock-free property of the scheduling algorithm greatly improves the scalability of the system and reduces latency.

[1] If all read results are not received within a specified timeframe, send additional requests to participants to get the results. If there is still no answer, also send requests to other replicas of the unresponsive participant.

2.1 OLLP

Certain file system operations—notably recursive moves, renames, deletes, and permission changes on non-empty directories—were implemented by bundling together many built-in transactions into a single compound transaction. It was not always possible to annotate these compound transaction requests with their full read- and write-sets (as required by Calvin's deterministic scheduler) at the time the recursive operation was initiated. In these cases, we made use of Calvin's Optimistic Lock Location Prediction (OLLP) mechanism [24] as we describe further in Section 5.2.

With OLLP, an additional step is added to the transaction execution pipeline: all transaction requests go through an `Analyze` phase before being appended to the log. The purpose of the `Analyze` phase is to determine the read- and write-sets of the transaction. Stores can implement custom `Analyze` logic for classes of transactions whose read- and write-sets can be statically computed from the arguments supplied by the client, or the `Analyze` function can simply do a "dry run" of the transaction execution, but not apply any writes. In general, this is done at no isolation, and only at a single replica, to make it as inexpensive as possible.

Once the `Analyze` phase is complete, the transaction is appended to the log, and it can then be scheduled and executed to completion. However, it is possible for a transaction's read- and write-sets to grow between the `Analyze` phase and the actual execution (called the `Run` phase) due to changes in the contents of the datastore. In this case, the worker executing the `Run` phase notices that the transaction is attempting to read or write a record that did not appear in its read- or write-set (and which was therefore not locked by the scheduler and cannot be safely accessed). It then aborts the transaction and returns an updated read-/write-set annotation to the client, who may then restart the transaction, this time skipping `Analyze` phase.

3 CalvinFS Architecture

CalvinFS was designed for deployments in which file data and metadata are both (a) replicated with strong consistency across geographically separated datacenters and (b) partitioned across many commodity servers within each datacenter. CalvinFS therefore simultaneously addresses the availability and scalability challenges described above—while providing standard, consistent file system semantics.

We engineered CalvinFS around certain additional goals and design principles:

Main-memory metadata store. Current metadata entries for *all* files and directories must be stored in main-memory across a shared-nothing cluster of machines.

Potentially many small files. The system must handle billions distinct files.

Scalable read/write throughput. Read and write *throughput* capacity must scale near-linearly and must not depend on replication configuration.

Tolerating slow writes. High update latencies that accommodate WAN round trips for the purposes of consistent replication are acceptable.

Linearizable and snapshot reads. When reading a file, clients must be able to specify one of three modes, each with different latency costs:

- **Full linearizable read.** If a client requires fully linearizable read semantics when reading a file, the read may be required to go through the same log-ordering process as any update operation.
- **Very recent snapshot read.** For many clients, very low-latency reads of extremely recent file system snapshots are preferable to higher-latency linearizable reads. We specifically optimize CalvinFS for this type of read operation, allowing for up to 400ms of staleness. (Note that this only applies to read-only operations. Read-modify-write operations on metadata—such as permissions checks before writing to a file—are always linearizable.)
- **Client-specified snapshot read.** Clients can also specify explicit version/timestamp bounds on snapshot reads. For example, a client may choose to limit staleness to make sure that a recent write is reflected in a new read, even if this requires blocking until all earlier writes are applied at the replica at which the read is occurring. Or a client may choose to perform snapshot read operations at a historical timestamp for the purposes of auditing, restoring a backup, or other historical analyses. Since only current metadata entries for each file/directory are pinned in memory at all times, it is acceptable for historical snapshot reads to incur additional latency when digging up now-defunct versions of metadata entries.

Hash-partitioned metadata. Hash partitioning of file metadata based on full file path is preferable to range- or subtree-partitioning, because it typically provides better load balancing and simplifies data placement tracking. Nonetheless, identifying the contents of a directory should only require reading from a single metadata shard.

Optimize for single-file operations. The system should be optimized for operations that create, delete, modify, or

read one file or directory at a time[2]. Recursive metadata operations such as directory copies, moves, deletes, and owner/permission changes must be fully supported (and should not require data block copying) but the metadata subsystem need not be optimized for such operations.

CalvinFS stores file contents in a non-transactional distributed block store analogous to the collection of chunk servers that make up a GFS deployment. We use our extension of Calvin described above to store all metadata, track directory namespaces, and map logical files to the blocks that store their contents.

Both the block store and the metadata store are replicated across multiple datacenters. In our evaluation, we used three physically separate (and geographically distant) datacenters, so our discussion below assumes this type of deployment and refers to each full system replica as being in its own datacenter. However, the replication mechanisms discussed here can just as easily be used in deployments within a single physical datacenter by dividing it into multiple *logical* datacenters.

As with GFS and HDFS, clients access a CalvinFS deployment not via kernel mounting, but via a provided client library, which provides standard file access APIs and file utils [10, 21]. No technical shortcoming prevents CalvinFS from being fully mountable, but implementation of this functionality remains future work.

4 The CalvinFS Block Store

Although the main focus of our design is metadata management, certain aspects of CalvinFS's block store affect metadata entry format and therefore warrant discussion. Most of these decisions were made to simplify the tasks of implementing, benchmarking, and describing the system; other designs of scalable block stores would also work with CalvinFS's metadata architecture.

4.1 Variable-Size Immutable Blocks

As in many other file systems, the contents of a CalvinFS file are stored in a sequence of zero or more blocks. Unlike most others, however, CalvinFS does *not* set a fixed block size—blocks may be anywhere from 1 byte to 10 megabytes. A 1-GB file may therefore legally consist of anywhere from one hundred to one billion blocks, although steps are taken to avoid the latter case.

Furthermore, blocks are completely immutable once written. When appending data to a file, CalvinFS does not append to the file's final block—rather, a new block containing the appended data (but not the original data) is written to the block store, and the new block's ID and size are added to the metadata entry for the file.

[2]Note that this still involves many distributed transactions. For example, creating or deleting a file also updates its parent directory.

4.2 Block Storage and Placement

Each block is assigned a globally unique ID, and is assigned to a block "bucket" by hashing its ID. Each bucket is then assigned to a certain number of block servers (analogous to GFS Chunkservers [10]) at each datacenter, depending on the desired replication factor for the system. Each block server stores its blocks in files on its local file system.

The mapping of buckets to block servers is maintained in a global Paxos-replicated configuration file and changes only when needed due to hardware failures, load balancing, adding new machines to the cluster, and other global configuration changes. Every CalvinFS node also caches a copy of the bucket map. This allows any machine to quickly locate a particular block by hashing its GUID to find the bucket, then checking the bucket map to find what block servers store that bucket. In the event where a configuration change causes this cached table to return stale data, the node will fail to find the bucket at the specified server, query the configuration manager to update its cached table, then retry.

In the event of a machine failure, each bucket assigned to the failed machine is reassigned to a new machine, which copies its blocks from a non-failed server that also stored the reassigned bucket.

To avoid excessive fragmenting, a background process periodically scans the metadata store and compacts files that consist of many small blocks. Once a compacted file is asynchronously re-written to the block store using larger blocks, the metadata is updated—as long as the file contents haven't changed since this compaction process began. If that part of the file *has* changed, the newly written block is discarded and the compaction process restarts for the file.

5 CalvinFS Metadata Management

The CalvinFS metadata manager logically contains an entry for every version (current and historical) of every file and directory to appear in the CalvinFS deployment. The metadata store is structured as key- value store, where each entry's key is the absolute path of the file or directory that it represents, and its value contains the following:

- **Entry type.** Specifies whether the entry represents a file or a directory[3].
- **Permissions.** CalvinFS uses a mechanism to support POSIX hierarchical access control that avoids full file system tree traversal when checking permissions for

[3]Although we see no major technical barrier to supporting linking in CalvinFS, adding support for soft and hard links remains future work.

an individual file by additionally storing all ancestor directories' permissions (up through the / directory) values in tree- ascending order in each metadata entry.

- **Contents.** For directories, this is a list of files and sub-directories immediately contained by the directory. For files, this is a mapping of byte ranges in the (logical) file to byte ranges within specific (physical) blocks in the block store. For example, if a file's contents are represented by the first 100 bytes of block X followed by the 28 bytes starting at byte offset 100 of block Y, then the contents would be represented as [(X, 0, 100), (Y, 100, 28)][4].

To illustrate this structure, consider a directory `fs` in user `calvin`'s home directory, which contains the source files for, say, an academic paper. The `calvinfs-ls` util (analogous to `ls -lA`) yields the following output:

```
$ calvinfs-ls /home/calvin/fs/
drwxr-xr-x calvin users  ...  figures
-rw-r--r-- calvin users  ...  ref.bib
-rw-r--r-- calvin users  ...  paper.tex
```

The CalvinFS metadata entry for this directory would be:

```
KEY:
  /home/calvin/fs
VALUE:
  type:         directory
  permissions:  rwxr-xr-x calvin users
  ancestor-
   permissions: rwxr-xr-x calvin users
                rwxr-xr-x root root
                rwxr-xr-x root root
  contents:     figures ref.bib paper.tex
```

We see that the entry contains permissions for the directory, plus permissions for three ancestor directories: `/home/calvin`, `/home`, and `/`, respectively. Since the path (in the entry's key) implicitly identifies these directories, they need not explicitly named in the value part of the field.

Since permissions checks need not access ancestor directory entries and the `contents` field names all files and subdirectories contained in the directory, the `calvinfs-ls` invocation above only needed to read that one metadata entry. Note that unlike POSIX-style `ls -lA`, however, the command above did not show the sizes of each file. To output those, additional metadata entries have to be read. For example, the metadata entry for `paper.tex` looks like this:

[4]This particular example might come about by the file being created containing 128 bytes in block Y, then having the first 100 bytes overwritten with the contents of block X.

```
KEY:
  /home/calvin/fs/paper.tex
VALUE:
  type:          file
  permissions:   rw-r--r-- calvin users
  ancestor-
    permissions: rwxr-xr-x calvin users
                 rwxr-xr-x calvin users
                 rwxr-xr-x root root
                 rwxr-xr-x root root
  contents:      0x3A28213A 0 65536
                 0x6339392C 0 65536
                 0x7363682E 0 34061
```

Since `paper.tex` is a file rather than a directory, its `contents` field contains block ids and byte offset ranges in those blocks. We see here that `paper.tex` is about 161 KB in total size, and its contents are a concatenation of byte ranges $[0, 65536)$, $[0, 65536)$, and $[0, 34061)$ in three specified blocks in the block store.

Storing all ancestor directories' permissions in each metadata entry eliminates the need for distributed permissions checks when accessing individual files, but comes with a tradeoff: when modifying permissions for a nonempty directory, the new permission information has to be atomically propagated recursively to all descendents of the modified directory. We discuss our protocol for handling such large recursive operations in Section 5.2.

5.1 Metadata Storage Layer

As mentioned above, the metadata management system in CalvinFS is an instance of Calvin with a custom storage layer implementation that includes compound transactions as well as primitive read/write operations. It implements six transaction types:

- `Read(path)` returns the metadata entry for specified file or directory.
- `Create{File,Dir}(path)` creates a new empty file or directory. This updates the parent directory's entry and inserts a new entry for the created file.
- `Resize(path, size)` a file. If a file *grows* as a result of a resize operation, all bytes past the previous file length are by default set to `0`.
- `Write(path, file_offset, source, source_offset, num_bytes)` writes a specified number of bytes to a file, starting at a specified offset within the file. The source data written must be a subsequence of the contents of a block in the block store.
- `Delete(path)` removes a file (or an empty directory). As with the file creation operation, the parent directory's entry is again modified, and the file's entry is removed.

- `Edit permissions(path, permissions)` of a file or directory, which may include changing the owner and/or group.

Each of these operation types also takes as part of its input the user and group IDs of the caller, and performs the appropriate POSIX-style permissions checking before applying any changes. Any POSIX-style file system interaction can be emulated by composing of multiple of these six built-in operations together in a single Calvin transaction.

Three of these six operations (read, resize, write) access only a single metadata entry. Creating or deleting a file or directory, however, touches two metadata entries: the newly created file/directory and its parent directory. Changing permissions of a directory may involve many entries, since all descendants must be updated, as explained above. Since entries are hash-partitioned across many metadata stores on different machines, the create, delete, and change permissions (of a non-empty directory) operations necessarily constitute distributed transactions.

Other operations, such as appending to, copying, and renaming files are constructed by bundling together multiple built-in operations to be executed atomically.

5.2 Recursive Operations on Directories

Recursive metadata operations (e.g., copying a directory, changing directory permissions) in CalvinFS use Calvin's built-in OLLP mechanism. The metadata store first runs the transaction at no isolation in `Analyze` mode to discover the read set without actually applying any mutations. This determines the entire collection of metadata entries that will be affected by the recursive operation by traversing the directory tree starting at the "root" of the operation—the metadata entry for the directory that was passed as the argument to the procedure.

Once the full read/write set is determined, it is added as an annotation to the transaction request, which is repeated in `Run` mode, during which the directory tree is retraversed from the operation to check that the read/write set of the operation has not grown (e.g., due to a newly inserted file in a subtree). If the read- and write-sets have grown between the `Analyze` and `Run` steps, OLLP (deterministically) aborts the transaction, and restarts it again in `Run` mode with an appropriately updated annotation.

6 The Life of an Update

To illustrate how CalvinFS's various components work together in a scalable, fault-tolerant manner, we present the end-to-end process of executing a simple operation—creating a new file and writing a string to it:

```
echo "import antigravity" >/home/calvin/fly.py
```

The first step is for the client to submit the request to a CalvinFS front-end—a process that runs on every CalvinFS server and orchestrates the actual execution of client requests, then returns the results to the client.

Write File Data

After receiving the client request, the front-end begins by performing the write by inserting a data block into CalvinFS's block store containing the data that will be written to the file. The first step here is to obtain a new, globally unique 64-bit block id β from a block store server. β is hashed to identify the bucket that the block will belong to, and the front-end then looks up in its cached configuration file the set of block servers that store that bucket, and sends a block creation request interface node now sends a block write request ($\beta \rightarrow$ `import antigravity`) to each of those block servers.

Once a quorum of the participating block servers (2 out of 3 in this case) have acknowledged to the front- end that they have created and stored the block, the next step is to update the metadata to reflect the newly created file.

Construct Metadata Operation

Since our system does not provide a single built-in operation that both creates a file and writes to it, this operation is actually a compound request specifying three mutations that should be bundled together:

- create file `/home/calvin/fly.py`
- resize the file to 18 bytes.
- write $\beta : [0, 18)$ to byte range $[0, 18)$ of the file

Once this compound transaction request (let's call it α) is constructed, the front-end is ready to submit it to be applied to the metadata store.

Append Transaction Request to Log

The first step in applying metadata mutation is for the CalvinFS front-end to append α to the log. The CalvinFS front-end sends the request to a Calvin *log* front-end, which appends α to its current batch of log entries, which has some globally unique id γ. When batch γ fills up with requests (or after a specified duration), it is written out another asynchronously replicated block store. Again, the log front-end waits for a majority of block servers to acknowledge its durability, and then does two things: (a) it submits the batch id γ to be appended to the Paxos- replicated metalog, and (b) it goes through the batch in order, forwarding each transaction request to all metadata shards that will participate in its execution.

Apply Update to Metadata Store

Each Calvin metadata shard is constantly receiving transaction requests from various Calvin log front- ends—however it receives them in a completely unspecified order. Therefore, it also reads new metalog entries as they are successfully appended, and uses these to sort the transaction requests coming in from all of the log front-ends, forming the precise subsequence of the log containing exactly those transactions in whose execution the shard will participate. Now, the sequencer at each metadata storage shard can process requests in the correct order.

Our example update α reads and modifies two metadata records: `/home/calvin` and `/home/calvin/fly.py`. Suppose that these are stored on shards P and Q, respectively. Note that each metadata shard is itself replicated multiple times—once in each datacenter in the deployment—but since no further communication is required between replicas to execute α, let us focus on the instantiations of P and Q in a single datacenter (P_0 and Q_0 in datacenter 0, say).

Both P_0 and Q_0 receive request α in its entirety and proceed to perform their parts of it. At P_0, α requests a lock on record `/home/calvin` from the local scheduler; at Q_0, α requests a lock on `/home/calvin/fly.py`. At each machine, α only starts executing once it has received its local locks.

Before we walk through the execution of α at P_0 and Q_0, let us first review the sequence of logical steps that the request needs to complete:

1. **Check parent directory permissions.** Abort transaction if `/home/calvin` does not exist or is not writable.
2. **Update parent directory metadata.** If `fly.py` is not contained in `/home/calvin`'s contents, add it.
3. **Check file permissions.** If the file exists and is not a writable file, abort the transaction.
4. **Create file metadata entry.** If no metadata entry exists for `/home/calvin/fly.py`, create one.
5. **Resize file metadata entry.** Update the metadata entry to indicate a length of 18 bytes. If it was previously longer than 18 bytes, this truncates it. If it was previously shorter (or empty), it is extended to 18 bytes, padded with zeros.
6. **Update file metadata entry's contents.** Write $\beta : [0,18)$ to the byte range $[0,18)$ of `/home/calvin/fly.py`, overwriting any previously existing contents in that range.

Note that steps 1 and 2 involve the parent directory metadata entry at P_0, while steps 3, 4, 5, and 6 involve only the new file's metadata at Q_0. However, steps 4 through 6 depend on the outcome of step 1 (as well as 3), so P_0 and Q_0 do need to coordinate in their handling of this mutation request. The two shards therefore proceed as follows:

P_0	Q_0
Check parent dir permissions (step 1).	Check file permissions; abort if not OK (step 3).
Send ***result*** (OK or ABORT) to Q_0.	Receive parent directory permissions check ***result*** from P_0.
If ***result*** was OK, update parent dir metadata (step 2).	If received ***result*** is OK, perform steps 4 through 6.

Both shards begin with permissions checks (step 1 for P_0 and step 3 for β). Suppose that both checks succeed. Now α sends an OK ***result*** message to β. β receives the ***result*** message, and now both shards execute the remainder of the operation with no further coordination.

Note that we were discussing datacenter 0's P and Q metadata shards. Metadata shards (P_1,Q_1), (P_2,Q_2), etc., in other datacenters independently follow these same steps. Since each shard deterministically processes the same request sequence from the log, metadata state remains strongly consistent: `import antigravity` is written to `/home/calvin/fly.py` identically at every datacenter.

7 Performance Evaluation

CalvinFS is designed to address the challenges of (a) distributing metadata management across multiple machines, and (b) wide area replication for fault tolerance. In exploring the scalability and performance characteristics of CalvinFS, we therefore chose experiments that explicitly stressed the metadata subsystem to its limits.

WAN replication. All results shown here used deployments that replicated all data and metadata three ways—across datacenters in Oregon, Virginia, and Ireland.

Many small data blocks. In order to test the performance of CalvinFS's metadata store (as opposed to the more easily scalable block storage component), we focused mainly on update-heavy workloads in which 99.9% of files were 1KB or smaller. Obviously, most real world file systems typically deal with much larger files; however, by experimenting on smaller files we were able to test the ability of the metadata store to handle billions of files while keeping the cluster size affordably small enough for our experimental budget. Obviously, larger files would require additional horizontal scalability of the block store; however this is not the focus of our work.

We use this setup to examine CalvinFS's memory usage, throughput capacity, latency, and fault tolerance.

7.1 Experimental Setup

All experiments were run on EC2 High-CPU Extra-Large instances[5]. Each deployment was split equally between AWS's US-West (Oregon), US-East (Virginia), and EU (Ireland) regions. Block and metadata replication factors were set to 3, and buckets, metadata shards, and Paxos group members were placed such that each object (metadata entries, log blocks, data blocks, and Paxos log and metalog entries) would be stored once in each datacenter.

Each machine served as (a) a block server (containing 30 buckets), (b) a log front-end, and (c) a metadata shard. In addition, one randomly selected machine from each datacenter participated in the Paxos group for the Calvin metalog. We ran our client load generation program on the same machines (but it did not use any knowledge about data or metadata placement when generating requests, so very few requests could be satisfied locally, especially in large deployments).

We ran each performance measurement on deployments of seven different sizes: 3, 6, 18, 36, 75, 150, and 300 total machines. As mentioned above, we had a limited budget for running experiments, so we could not exceed 300 machines. However, we were able to store billions of files across these 300 machines by limiting the file size. Our results can be translated directly to larger clusters that have more machines and larger files (and therefore the same total number of files to manage). We compare our findings directly to HDFS performance measurements published by Yahoo researchers [20].

7.2 File Counts and Memory Usage

After creating each CalvinFS deployment, we created 10 million files per machine. File sizes ranged from 10 bytes to 1MB, with an average size of 1kB. 90% of files contained only one block, and 99.9% of files had a total size of under 1kB. Most file names (including full directory paths) were between 25 and 50 bytes long.

We found that total memory usage for metadata was approximately 140 bytes per metadata entry—which is closely comparable to the per-file metadata overhead of HDFS [28]. Unlike HDFS, however, the metadata shards did *not* store an in-memory table of block placement data, since Calvin uses a coarser-grained bucket placement mechanism instead. We would therefore expect an HDFS-like file system deployment (with ˜1 block per file) to require approximately twice the amount of total memory to store metadata (assuming the same level of metadata replication). Of course, by partitioning metadata across machines, CalvinFS requires far less memory **per machine**.

Our largest deployment—300 machines—held 3 billion files (and therefore 9 billion total metadata entries) in a total of 1.3 TB of main memory. This large number of files is far beyond what HDFS can handle [27].

[5]Each EC2 High-CPU Extra-Large instance contains 7 GB of memory, 20 EC2 Compute Units (8 virtual cores with 2.5 EC2 Compute Units each with the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor), and 1690 GB of instance storage

7.3 Throughput Capacity

Next, we examined the throughput capacity (Figure 1) and latency distributions (Figure 2) of reading files, writing to files, and creating files in CalvinFS deployments of varying sizes. For each measurement, we created client applications that issued requests to read files, create files, and write to files—but with different frequencies. For experiments on read throughput, 98% of all client requests were reads, with 1% of operations being file creations and 1% being writes to existing files. Similarly, for write benchmarks, clients submitted 98% write requests, and for append benchmarks, clients submitted 98% append requests. For all workloads, clients chose which files to read, write, and create using a Gaussian distribution.

Once key feature of CalvinFS is that throughput is totally unaffected by WAN replication (and the latencies of message passing between datacenters). This is because once a transaction is replicated to all datacenters by the Calvin log component (which happens before request execution begins), no further cross-datacenter communication is required to execute the transaction to completion. Therefore, we only experiment with the three datacenter case of Oregon, Virginia, and Ireland for these set of experiments—changing datacenter locations (or even removing WAN replication entirely) has no effect on throughput results. Latency, however, is affected by the metalog Paxos agreement protocol across datacenters, which we discuss in Section 7.4 below.

Read Throughput

For many analytical applications, extremely high read *throughput* is extremely important, even if it comes at the cost of occasionally poor latencies for reads of specific files. On the other hand, being able to rely on consistent read latencies vastly simplifies the development of distributed applications that face end-users. We therefore performed two separate read throughput experiments: one in which we fully saturated the system with read requests, resulting in "flaky" latency, and one at only partial load that yields reliable (99.9th percentile) latency (Figures 1a and 1b). Because each datacenter stores a full, consistent replica of all data and metadata, each read

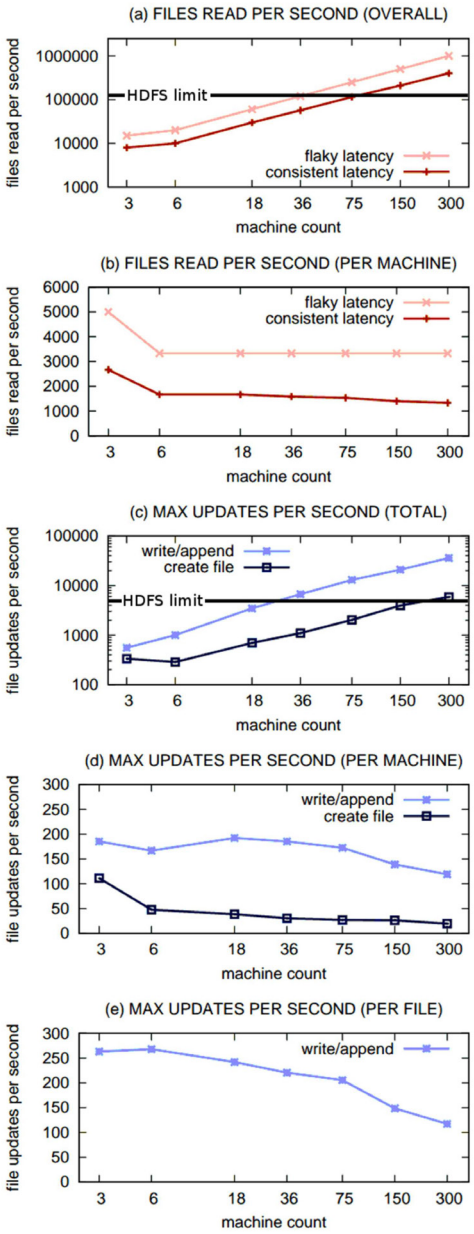

Figure 1: Total and per-machine read (a,b) and update (c,d) throughput, and maximum per-file update throughput (e), for WAN-replicated CalvinFS deployments.

request is routed only to the relevant machine(s) in the same datacenter as the client.

Specifically, we observed that under very heavy load, occasional background tasks such as LSM tree compactions and garbage collection could cause a large number of concurrent read requests to stall, introducing occasional latency spikes and some completely failed reads that then had to be retried. Median and 90th percentile latencies, however, were comparable to those observed for the "partial load" experiments described below.

For our partial load experiments, we reduced the number of clients as far as necessary to completely remove latency spikes. Typically running the system at 50% of the maximum load accomplished this. For our largest deployments, we had to reduce the load to about 45% of maximum throughput to accomplish this [6].

Figures 1a and 1b) show that CalvinFS is able to achieve linear scalability for read throughput, even as millions of files are read per second. At machine count 3, there is only one machine per datacenter, so all reads can be satisfied locally, which yields very high throughput. Starting with machine count 6, however, the probability of at least one non-local access increases rapidly (already at machine count 6 there is a 75% probability that either the file metadata or the file data itself will be non-local).

We include in Figure 1a the upper bound of read request throughput for HDFS, as reported by Yahoo researchers in 2010[20]. Specifically, this corresponds to block location lookups by the NameNode. It was found that the HDFS metadata store can serve no more than 126,119 block location lookups per second. Since read requests involve more metadata operations than just a single block location lookup—such as other metadata entry lookups to check file existence, permissions, and block IDs, not to mention possibly having to look up multiple block locations if the file spans multiple blocks—this is strictly an upper bound. We also assume here that the metadata management layer is the *only* bottleneck for reads, which in HDFS would certainly not be the case for small deployments. It is fair to expect actual HDFS read throughput to be considerably lower than the upper bound plotted in Figure 1a.

Update Throughput

Next, we measured the total number of file creation and append operations that each CalvinFS deployment could perform (Figures 1c and 1d). Append throughput scaled very nearly linearly with the number of machines in the cluster, reaching about 40,000 appends per second with a 300- machine cluster.

However, file creation throughput scaled slightly less smoothly. This is because each file creation operation is implemented as a distributed transaction (since metadata entries had to be modified for both the parent directory and the newly-created file)—requiring coordination between metadata shards to complete. As more machines

[6] The problems of performance isolation between processes and mitigating tail latencies have been studied extensively, and many techniques have been developed that could be applied to CalvinFS to safely increase CPU utilization and improve performance, but these are outside the scope of this paper.

are added, the likelihood increases that at any given time at least one machine will "fall behind," delaying other machines' progress involved in distributed transactions accessing data at that node. This effect is similar to that observed running TPC-C and other OLTP benchmarks using the Calvin framework [24]. Despite slightly sub-linear scaling, file creation capacity in CalvinFS still scales very well overall—far better than alternatives that do not scale metadata management across multiple machines.

We include in Figure 1c the observed HDFS throughput upper bound of 5600 new blocks per second [20]. In our benchmark, each file creation and write involved creating a block, then performing one or more other metadata updates. We therefore expect actual HDFS update throughput of this type to be considerably lower than 5600 operations per second, but this figure serves as a proven upper bound.

Concurrent Writes to Contended Files

Many distributed systems do not perform well when a large number of clients concurrently attempt to write to the same file—such as heavy traffic of simultaneous appends to a shared log file. The systems that provide the best performance for this situation often forgo consistent replication and strong linearizability guarantees to do so.

CalvinFS, however, supports high concurrent write throughput to individual files without sacrificing linearizability. To demonstrate this, we performed an experiment in which we chose one file for every three machines in the full deployment (so 1 file for the 3-machine deployment and 100 files for the 300 machine deployment) and had 100 independent clients per file repeatedly send requests to either append data to that file or perform a random write within the file. Figure 1e shows the resulting per-file throughput. Small-cluster CalvinFS deployments sustained rates of 250 writes or appends per second on each file. Our largest deployments sustained 130 writes or appends per second on each file.

7.4 Latency Measurements

Next, we examined the latency distribution for file read, write, and file creation operations for deployments of 36 and 300 machines (Figures 2a and 2b). Latencies are measured from when a client submits a request until the operation is completed and it receives a final response.

Read Latencies

Our measurement of read latencies was taken under "non-flaky" load, which is about half of maximum read throughput. In all cases, read requests are served by the nearest metadata shard and block server within the same datacenter.

We broke reads down into three categories: (a) reads of files that contain no data in the block store (this includes `ls` operations on directories, since each directory's contents are listed in its metadata entry), reads of files that contain a single block, and reads of multi-block files.

There are several interesting features to note in these plots. First, in the 36-machine deployment, the median latency to read a non-empty file is about 3ms and the 99th percentile latency is about 80ms, while at 300 machines, median latency is about 5 ms, and 99th percentile latency is about 120ms. Although adding more machines to a distributed system invariably introduces performance variabilities, we deemed this a reasonable latency price for nearly an order of magnitude of scaling.

Second, when reading zero-block files in the 36-machine deployment (which has 12 machine per datacenter), about 1 read in 12 is extremely fast—less than 100 microseconds—because 1 in 12 metadata lookups happen to occur on the same machine as the interface node handling the client's request, requiring no network round trips. The same effect is visible for one 100th of metadata-only reads in the 300-machine deployment (which, likewise, has 100 machines per datacenter). LAN round-trip times within a datacenter were 1 ms—about the latency of most non-local metadata-only reads. Similarly, 1-block reads generally incur 2 round trips, while 2+ block reads incur 3 or more. Because of the non-uniform distribution of files read, around 85% of blocks could be served directly from block servers' OS memory cache, without needing to go to disk; only 15% of 1-block reads incur disk I/O costs; among reads of multi-block files, the frequency of I/O latencies appearing is higher.

Although these benchmarks may not be very indicative of real-world usage patterns for distributed file systems (which would likely include many more large files, and in some cases worse cache locality for reads), we chose them to highlight the specific sources of latency that are introduced by components *other* than the block store. Therefore, one can know what to expect if a CalvinFS-style metadata subsystem were coupled with an off-the-shelf block store whose performance and scalability was already well-documented for petabyte-scale data volumes and much larger individual block sizes.

Update Latencies

Latencies for file creation and write/append requests are dominated by WAN round-trip times. Creating a file typically incurs approximately two non-overlapping round trip latencies: one for the log front-end to write its request batch out to a majority of datacenters, and one to append the entry to the Paxos metalog.

Although we saw above that CalvinFS achieves more

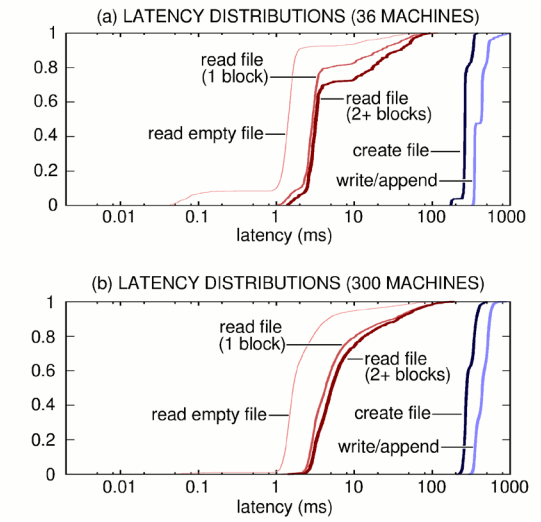

Figure 2: Latency distributions for read, write/append, and create operations for WAN-replicated CalvinFS deployments of (a) 36 machines and (b) 300 machines.

impressive *throughput* for writes and appends to existing files than for file creation, the latency for writes/appends is higher—one additional non- overlapping WAN round trip is necessary to replicate the newly created data blocks before requesting the metadata update.

7.5 Fault Tolerance

Since we designed CalvinFS's WAN replication mechanism with the explicit goal of high availability, we now test our system in the presence of full datacenter failure. In our next experiment, we killed all CalvinFS processes in the Virginia datacenter while a 36-machine CalvinFS deployment the system was running under a mixed read/create/write load. Specifically, we deployed 1000 clients—one third constantly reading files, one third constantly creating new files, and one third constantly appending to files. This saturates the file system's maximum file creation throughput capacity (which is limited by lock contention) and represents approximately 50% read load and 20% append load.

Figure 3 shows throughput (a) and median and 99th-percentile latency (b) for the 30 seconds immediately preceding and following the datacenter "failure". In order to clearly show what effects this had on CalvinFS's core operation capacities, we *immediately* redirected all new client requests that would have been routed to Virginia to either Oregon or Ireland, rather than requiring clients to wait for timeouts before resuming (which would have "unfairly" given the system time to recover from its sudden involuntary reconfiguration).

We see here that total read, create, and write/append throughput capacity is only reduced by a small amount,

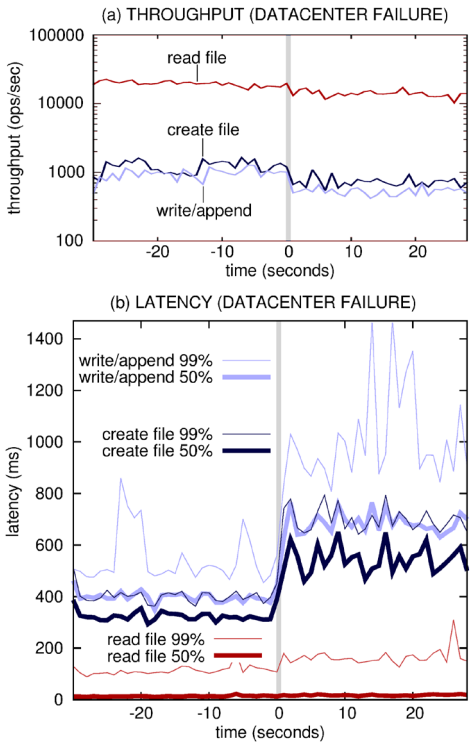

Figure 3: Throughput (a) and latency (b) for the time window preceding and following a datacenter failure.

median read latency remains unchanged, and 99th-percentile read latency only increases by about 30%. File creation and write/append latency, however, roughly double. The reason for this is that the non-overlapping portions of WAN latencies goes from being around 100ms (round trip between either Oregon and Virginia or Virginia and Ireland—each of which pair forms a quorum) to nearly 200ms (round trip between Oregon and Ireland, which now represent the only quorum). No file is at any time unavailable for reading or writing.

In summary, we found that CalvinFS tolerated an unplanned datacenter outage with exceptional grace.

8 Related Work

CalvinFS builds on a long history of research on the scalability and reliability of distributed file systems.

We modeled certain aspects of the CalvinFS design after GFS/HDFS. In particular, our decision to concentrate all metadata in the main memory of a specific metadata component is based on the success of this tactic in GFS/HDFS. CalvinFS' block store is also a simplification of the GFS/HDFS model that uses consistent hashing to simplify block metadata. Our implementation of CalvinFS' novel features—scalable metadata management and consistent WAN replication—was designed

to illuminate a path that a GFS-/HDFS-like file system could take towards eliminating the single metadata master as both a scalability bottleneck and availability hazard [10, 21, 20].

In 2009, Google released a retrospective on the scalability, availability, and consistency challenges that GFS had faced since its creation, attributing many difficulties to its single-master design. The interview also describes a new distributed-master implementation of GFS that stores metadata in Bigtable [10, 14, 6]. Since Bigtable supports neither multi-row transactions nor synchronous replication, it is unclear how (or if) this new GFS implementation supports strongly consistent semantics and linearizable file updates while maintaining high availability—particularly in the case of machine failures in the metadata Bigtable deployment.

The Lustre file system resembles GFS in that it uses a single metadata server (MDS), but it does not store per-block metadata, reducing MDS dependence in the block creation and block-level read paths. The latest release of Lustre allows metadata for specific directory subtrees to be offloaded to special "secondary" MDSs for outward scalability and load balancing. Lustre supports only cluster-level data replication [18].

Tango provides an abstraction of a distributed, transactional data structure backed by a replicated, flash-resident log, and is designed for use in metadata subsystems. Like in the CalvinFS metadata manager, a Tango deployment's state is uniquely determined by a single serialized log of operation requests. Tango transactions use optimistic concurrency control, however: they log a commit entry as the final execution step (readers of the log are instructed to ignore any commit entry that turns out to be preceded by a conflicting one). To avoid high optimistic abort rates under contention, this mechanism requires a log implementation with very low append latency. Since synchronous geo-replication inherently incurs high latencies, Tango is only suited to single-datacenter deployments [5].

IBM's GPFS distributes metadata using a shared-disk abstraction and allows multiple machines to access it concurrently protected by a distributed locking mechanism. When multiple clients access the same object, however, distributed locking mechanisms perform poorly. File systems that store metadata using shared-disk arrays depend on low-latency network fabrics to mitigate these issues [17].

Gluster distributes and replicates both data blocks and file metadata entries using an elastic hashing algorithm. However, adding replicas to a Gluster deployment significantly hinders write throughput. Furthermore, Gluster's implementation of copy and rename operations forces data blocks as well as metadata to be copied between storage shards, which can easily become too expensive [1].

The Ceph file system scales metadata management by dynamically partitioning metadata by directory subtree (and hashing "hotspot" directories across multiple metadata servers). Ceph is optimized for single- datacenter deployments, however; its metadata replication mechanism relies heavily on low latencies between replicas to avoid introducing update-contention bottlenecks [25].

The Panasas File System colocates file metadata with file data on Object-based Storage Devices (OSDs), each of which manages (RAID-based) data replication independently. OSDs optimize caching for high-throughput concurrent reads. Clients cache a global mapping of file system objects to OSDs, updates to which require global synchronization [26].

The Ursa Minor Storage System uses subtree-partitioning to distribute metadata but takes a different approach to outwardly scalable metadata management: any time an atomic operation would span multiple partitions, instead of using a distributed transaction, it repartitions the metadata data, migrating all entries that need to be atomically updated to the same partition [3].

The Farsite file system is designed to unite a collection of "desktop" computers rather than datacenters full of rack servers. Early versions Farsite relied on a single metadata server, but Farsite now supports dynamic subtree-partitioning as well, but no metadata replication [4].

Frangipani and xFS are shared-disk distributed file systems. xFS implemented a "serverless" file system, distributing file data and metadata across a collection of disks, using a globally replicated mapping of file system object locations. All implementation logic is executed by clients, using on-disk state for synchronization. Some currently popular shared-disk-based file systems appear to be loosely based on the xFS design [23, 22].

Panache approaches file system scalability from a different direction—providing scalable caching of both data and metadata for a traditional (and less scalable) file system. Although Panache does not provide a full replacement for a file system's metadata component, it effectively removes some bottlenecks, particularly from the read path, via partitioning and replication [9].

Like CalvinFS, Giga+ uses hash partitioning to distribute metadata for across many servers within a datacenter. However Giga+'s distributed operations are eventually consistent and rely on clever handling of stale client-side state [16].

9 Conclusions

CalvinFS deployments can scale on large clusters of commodity machines to store billions of files and process hundreds of thousands of updates and millions of reads per second—while maintaining consistently low read latencies. Furthermore, CalvinFS deployments can survive entire datacenter outages with only minor performance consequences and no loss of availability at all.

References

[1] Glusterfs. *Gluster Community, http://gluster.org/*.

[2] Riak. *Basho, http://basho.com/riak*.

[3] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. P. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, et al. Ursa minor: Versatile cluster-based storage. In *FAST*, volume 5, 2005.

[4] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. *ACM SIGOPS Operating Systems Review*, 36(SI):1–14, 2002.

[5] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: Distributed data structures over a shared log. In *SOSP*, 2013.

[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *TOCS*, 2008.

[7] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *Proc. of OSDI*, pages 251–264, 2012.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

[9] M. Eshel, R. L. Haskin, D. Hildebrand, M. Naik, F. B. Schmuck, and R. Tewari. Panache: A parallel file system cache for global file access. In *FAST*, 2010.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proc. of SOSP*, 2003.

[11] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, and E. Cesario. The xtreemfs architecture: a case for object-based file systems in grids. *Concurrency and computation*, 20(17), 2008.

[12] A. Lakshman and P. Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 2010.

[13] P. I. LLC. National survey on data center outages. 2010.

[14] M. K. McKusick and S. Quinlan. Gfs: Evolution on fast-forward.

[15] R. McMillan. Guns, squirrels, and steel: The many ways to kill a data center. *Wired*, 2012.

[16] S. Patil and G. A. Gibson. Scale and concurrency of giga+: File system directories with millions of files. In *FAST*, 2011.

[17] F. B. Schmuck and R. L. Haskin. Gpfs: A shared-disk file system for large computing clusters. In *FAST*, volume 2, 2002.

[18] P. Schwan. Lustre: Building a file system for 1000-node clusters. In *Proceedings of the 2003 Linux Symposium*, 2003.

[19] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. O. K. Littlefield, D. Menestrina, S. E. J. Cieslewicz, I. Rae, et al. F1: A distributed sql database that scales. *Proceedings of the VLDB Endowment*, 6(11), 2013.

[20] K. Shvachko. Hdfs scalability: the limits to growth. 2009.

[21] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *MSST*, 2010.

[22] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the xfs file system. In *USENIX Annual Technical Conference*, volume 15, 1996.

[23] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: a scalable distributed file system. *SIGOPS Oper. Syst. Rev.*, 1997.

[24] A. Thomson, T. Diamond, S. chun Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD*, 2012.

[25] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn. Ceph: a scalable, high-performance distributed file system. In *OSDI*, 2006.

[26] B. Welch, M. Unangst, Z. Abbasi, G. A. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable performance of the panasas parallel file system. In *FAST*, volume 8, 2008.

[27] T. White. The small files problem. *Cloudera Blog, blog.cloudera.com/blog/2009/02/the-small-files-problem/*.

[28] T. White. *Hadoop: The Definitive Guide*. 2nd edition, 2010.