The Design and Implementation of the Warp Transactional Filesystem

Robert Escriva and Emin Gün Sirer, *Cornell University*

https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/escriva

This paper is included in the Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16).

March 16–18, 2016 • Santa Clara, CA, USA

ISBN 978-1-931971-29-4

The Design and Implementation of the Warp Transactional Filesystem

Robert Escriva, Emin Gün Sirer
Computer Science Department, Cornell University

Abstract

This paper introduces the Warp Transactional Filesystem (WTF), a novel, transactional, POSIX-compatible filesystem based on a new *file slicing* API that enables efficient zero-copy file transformations. WTF provides transactional access spanning multiple files in a distributed filesystem. Further, the file slicing API enables applications to construct files from the contents of other files without having to rewrite or relocate data. Combined, these enable a new class of high-performance applications. Experiments show that WTF can qualitatively outperform the industry-standard HDFS distributed filesystem, up to a factor of four in a sorting benchmark, by reducing I/O costs. Microbenchmarks indicate that the new features of WTF impose only a modest overhead on top of the POSIX-compatible API.

1 Introduction

Distributed filesystems are a cornerstone of modern data processing applications. Key-value stores such as Google's BigTable [11] and Spanner [14], and Apache's HBase [7] use distributed filesystems for their underlying storage. MapReduce [15] uses a distributed filesystem to store the inputs, outputs, and intermediary processing steps for offline processing applications. Infrastructure such as Amazon's EBS [2] and Microsoft's Blizzard [28] use distributed filesystems to provide storage for virtual machines and cloud-oblivious applications.

Yet, current distributed filesystems exhibit a tension between retaining the familiar semantics of local filesystems and achieving high performance in the distributed setting. Often, designs will compromise consistency for performance, require special hardware, or artificially restrict the filesystem interface. For example, in GFS, operations can be inconsistent or, "consistent, but undefined," even in the absence of failures [19]. GFS-backed applications must account for these anomalies, leading to additional work for application programmers. HDFS [4] side-steps this complexity by prohibiting concurrent or non-sequential modifications to files. This obviates the need to worry about nuances in filesystem behavior, but fails to support use cases requiring concurrency or random-access writes. Flat Datacenter Storage [29] is eventually consistent and requires a network with full-bisection bandwidth, which can be cost prohibitive and is not possible in all environments.

This paper introduces the Warp Transactional Filesystem (WTF), a new distributed filesystem that exposes transactional support with a new API that provides *file slicing* operations. A WTF transaction may span multiple files and is fully general; applications can include calls such as read, write, and seek within their transaction. This file slicing API enables applications to efficiently read, write, and rearrange files without rewriting the underlying data. For example, applications may concatenate multiple files without reading them; garbage collect and compress a database without writing the data; and even sort the contents of record-oriented files without rewriting the files' contents.

The key design decision that enables WTF's advanced feature set is an architecture that represents filesystem data and metadata to ensure that filesystem-level transactions may be performed using, solely, transactional operations on metadata. Custom storage servers hold filesystem data and handle the bulk of I/O requests. These servers retain no information about the structure of the filesystem; instead, they treat all data as opaque, immutable, variable-length arrays of bytes, called *slices*. WTF stores references to these slices in HyperDex [17] alongside metadata that describes how to combine the slices to reconstruct files' contents. This structure enables bookkeeping to be done entirely at the metadata level, within the scope of HyperDex transactions.

Supporting this architecture is a custom concurrency control layer that decouples WTF transactions from the underlying HyperDex transactions. This layer ensures that transactions only abort when concurrently-executing transactions change the filesystem and generate an application-visible conflict. This seemingly minor functionality enables WTF to support concurrent operations with minimal abort-induced overheads.

Overall, this paper makes three contributions. First, it describes a new API for filesystems called file slicing that enables efficient file transformations. Second, it describes an implementation of a transactional filesystem with minimal overhead. Finally, it evaluates WTF and the file slicing interfaces, and compares them to the non-transactional HDFS filesystem.

2 Design

WTF's distributed architecture consists of four components: the metadata storage, the storage servers, the replicated coordinator, and the client library. Figure 1 summarizes this architecture. The metadata storage builds on top of HyperDex and its expansive API. The storage servers hold filesystem data, and are provisioned for high I/O workloads. A replicated coordinator service serves as a rendezvous point for all components of the system, and maintains the list of storage servers. The client library contains the majority of the functionality of the system, and is where WTF combines the metadata and data into a coherent filesystem.

In this section, we first explore the file slicing abstraction to understand how the different components contribute to the overall design. We will then look at the design of the storage servers to understand how the system stores the majority of the filesystem information. Finally, we discuss performance optimizations and additional functionality that make WTF practical, but are not essential to the core design, such as replication, fault tolerance, and garbage collection.

2.1 The File Slicing Abstraction

WTF represents a file as a sequence of byte arrays that, when overlaid, comprise the file's contents. The central abstraction is a *slice*, an immutable, byte-addressable, arbitrarily sized sequence of bytes. A file in WTF, then is a sequence of slices and their associated offsets. This representation has some inherent advantages over block-based designs. Specifically, the abstraction provides a separation between metadata and data that enables filesystem-level transactions to be implemented using, solely, transactions over the metadata. Data is stored in the slices, while the metadata is a sequence of slices. WTF can transactionally change these sequences to change the files they represent, without rewriting data.

Concretely, file metadata consists of a list of *slice pointers* that indicate the exact location on the storage servers of each slice. A slice pointer is a tuple consisting of the unique identifier for the storage server holding the slice, the local filename containing the slice on that storage server, the offset of the slice within the file, and

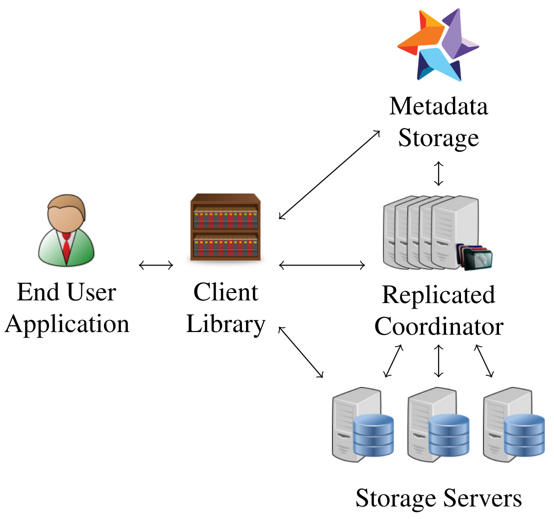


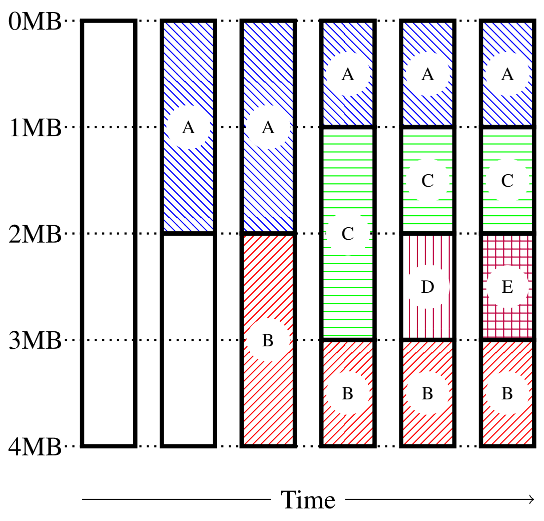
Figure 1: WTF employs a distributed architecture consisting of metadata storage, data storage, a replicated coordinator, and the client library. The client library unifies the metadata storage and storage servers to provide a filesystem interface.

the length of the slice. Associated with each slice pointer is an integer offset that indicates where the slice should be overlaid when reconstructing the file. Crucially, this representation is self-contained: everything necessary to retrieve the slice from the storage server is present in the slice pointer, with no need for extra bookkeeping elsewhere in the system. As we will discuss later, the metadata also contains standard info found in an inode, such as modification time, and file length.

This slice pointer representation enables WTF to easily generate new slice pointers that refer to subsequences of existing slices. Because the representation directly reflects the global location of a slice on disk, WTF may use simple arithmetic to create new slice pointers.

This representation also enables applications to modify a file with only localized modifications to the metadata. Figure 2 shows an example file consisting of five different slices. Each slice is overlaid on top of previous slices. Where slices overlap, the latest additions to the metadata take precedence. For example, slice C takes precedence over slices A and B; similarly, slice E completely obscures slice D and part of C. The file, then, consists of the corresponding slices of A, C, E, and B. The figure also shows the *compacted* metadata for the same file. This compacted form contains the minimal slice pointers necessary to reconstruct the file without reading data that is hidden by another slice. Crucially, all file modifications can be performed by appending to the list of slice pointers.

The procedures for reading and writing follow directly from the abstraction. A writer creates one or more slices on the storage servers, and overlays them at the appropriate positions within the file by appending their slice



Final Metadata:
A@[0,2], B@[2,4], C@[1,3], D@[2,3], E@[2,3]
Compacted Final Metadata:
A@[0,1], C@[1,2], E@[2,3], B@[3,4]

Figure 2: Writers append to the metadata list to change the file. Every prefix of the shown metadata list represents a valid state of the file at some point in time. The compacted metadata occupies less space by rearranging the metadata list to remove overwritten data.

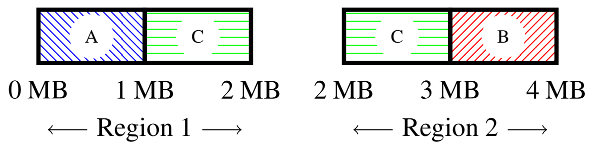
pointers to the metadata list. Readers retrieve the metadata list, compact it, and determine which slices must be retrieved from the storage servers to fulfill the read.

The correctness of this design relies upon the metadata storage providing primitives to atomically read and append to the list. HyperDex natively supports both of these operations. Because each writer writes slices before appending to the metadata list, it is guaranteed that any transaction that can see these immutable slices is serialized *after* the writing transaction commits. It can then retrieve the slices directly. The transactional guarantees of WTF extend directly from this design as well: a WTF transaction will execute a single HyperDex transaction consisting of multiple append and retrieve operations.

2.2 Storage Server Interface

The file slicing abstraction greatly simplifies the design of the storage servers. Storage servers deal exclusively with slices, and are oblivious to files, offsets, or concurrent writes. The minimal API required by file slicing consists of just two calls to create and retrieve slices.

A storage server processes a request to create a slice by writing the data to disk and returning a slice pointer to the caller. The structure of this request intentionally grants the storage server complete flexibility to store the slice anywhere it chooses because the slice pointer containing the slice's location is returned to the client only after the slice is written to disk. A storage server can



Region 1 Metadata:
A@[0,2], C@[1,2]

Region 2 Metadata:
B@[2,4], C@[2,3]

Figure 3: Files are partitioned into multiple regions to decouple the size of metadata lists from the size of the file. This figure shows the fourth state of the file from Figure 2 partitioned into 2 MB regions. Writes that are entirely within a single region are appended solely to that region's metadata. Writes that cross regions are transactionally appended to multiple lists.

retrieve slices by following the information in the slice pointer to open the named file, read the requisite number of bytes, and return them to the caller.

The direct nature of the slice pointer minimizes the bookkeeping required of the storage server implementation and permits a wide variety of implementation strategies. In the simplest strategy, which is the strategy used in the WTF implementation, each WTF storage server maintains a directory of slice-containing backing files and information about their own identities in the system. Each backing file is written sequentially as the storage server creates new slices.

As an optimization, each storage server maintains multiple backing files to which slices are appended. This serves three purposes: First, it allows servers to avoid contention when writing to the same file; second, it allows the storage server to spread data across multiple filesystems if configured to do so; and, finally, it allows the storage server to use hints provided by writers to improve locality on disk, as described in Section 2.7.

2.3 File Partitioning

Practically, it is desirable to keep the list of slice pointers small so that they can be stored, retrieved, and transmitted with low overhead; however, it would be impractical to achieve this by limiting the number of writes to a file. In order to achieve support for both arbitrarily large files and efficient operations on the list of slice pointers, WTF partitions a file into fixed size regions, each with its own list. Each region is stored as its own object in HyperDex under a deterministically derived key.

Operations on these partitioned metadata lists directly follow from the behavior of the system with a single metadata list. When an operation spans multiple regions, it is decomposed into one operation per region, and the decomposed operations execute within the context of a single HyperDex transaction. This guarantees that multi-region operations execute as one atomic action. Figure 3 shows a sample partitioning of a file, and how operations can span multiple metadata lists.

API	Description
`yank(fd,sz):slice,[data]`	Copy `sz` bytes from `fd`; return slice pointers and optionally the data
`paste(fd, slice)`	Write `slice` to `fd` and increment the offset
`punch(fd, amount)`	Zero-out `amount` bytes at the fd offset, freeing the underlying storage
`append(fd, slice)`	Append `slice` to the end of file `fd`
`concat(sources, dest)`	Concatenate the listed files to create dest
`copy(source, dest)`	Copy source to dest using only the metadata

Table 1: WTF's new file slicing API. Note that these supplement the POSIX API, which includes calls for moving a file descriptor's offset via `seek`. `concat` and `copy` are provided for convenience and may be implemented with `yank` and `paste`.

2.4 Filesystem Hierarchy

The WTF filesystem hierarchy is modeled after the traditional Unix filesystem, with directories and files. Each directory contains entries that are named links to other directories or files, and WTF enables files to be hard linked to multiple places in the filesystem hierarchy.

WTF implements a few changes to the traditional filesystem behavior to reduce the scope of a transaction when opening a file. Path traversal, as it is traditionally implemented, puts every directory along the path within the scope of a transaction, and requires multiple round trips to both HyperDex and the storage servers.

WTF avoids traversing the filesystem on open by maintaining a pathname to inode mapping. This enables a client to map a pathname to the corresponding inode with just one HyperDex lookup, no matter how deeply nested the pathname. To enable applications to enumerate the contents of a single directory, WTF maintains traditional-style directories, implemented as special files, alongside the one-lookup mapping. The two data structures are atomically updated using HyperDex transactions. This optimization simplifies the process of opening files, without significant loss of functionality.

Inodes are also stored in HyperDex, and contain standard information, such as link count and modification time. The inode also maintains ownership, group, and permissions information, though WTF differs from POSIX in that permissions are not checked on the full pathname from the root. Each inode also stores a reference to the highest-offset region for the file, enabling applications to find the end of the file. The inode refers to a region instead of a particular offset so that the inode is only written when the file grows beyond the bounds of a region, instead of every time the file changes in size.

Because HyperDex permits transactions to span multiple keys across independent schemas, updates to the filesystem hierarchy remain consistent. For example, to create a hardlink for a file, WTF atomically creates a new pathname to inode mapping for the file, increments the inode's link count, and inserts the pathname and inode pair into the destination directory, which requires a write to the file holding the directory entries.

2.5 File Slicing Interface

The file slicing interface enables new applications to make more efficient use of the filesystem. Instead of operating on bytes and offsets as traditional POSIX systems do, this new API allows applications to manipulate subsequences of files at the structural level, without copying or reading the data itself.

Table 1 summarizes the new APIs that WTF provides to applications. The `yank`, `paste`, and `append` calls are analogous to read, write, and append, but operate on slices instead of sequences of bytes. The `yank` call retrieves slice pointers for a range of the file. An application may provide these slice pointers to a subsequent call to `paste` or `append` to write the data back to the filesystem, reusing the existing slices. These write operations bypass the storage servers and only incur costs at the metadata storage component.

The `append` call is internally optimized to improve throughput. A naive `append` call could be implemented as a transaction that seeks to the end of the file, and performs a `paste`. While not incorrect, such an implementation would prohibit concurrency because only one append could commit for each value for the end of file. Instead, WTF stores alongside the metadata list an offset representing the end of the region. An `append` call translates to a conditional list append call within HyperDex that only succeeds when the current offset plus the length of the slice to be appended does not exceed the bounds of the region. When an append is too large to fit within a single region, WTF will fall back on reading the offset of the end of file, and performing a write at that offset. This enables multiple `append` operations to proceed in parallel in the common case.

The remaining calls in the file slicing API are provided for convenience, as they may be implemented in terms of `yank` and `paste`. `concat` concatenates multiple files to create one unified output file. `copy` copies a file by copying the file's compacted metadata.

2.6 Transaction Retry

To guarantee that WTF transactions never spuriously abort, WTF implements its own concurrency control that retries aborted metadata transactions. WTF operations in the client library often read metadata during the course of an operation that is not exposed to the calling application. A change to this data after it is read may force the metadata transaction to abort, but to abort the corresponding WTF transaction would be spurious from the perspective of the application.

For example, consider a file opened in "append" mode. Each write to the file must be written at the end-of-file offset, but the application does not learn this offset from the write. Internally, the client library computes the end of file, and then writes data at that offset. If the file changes in size between these two operations, the metadata transaction will abort. WTF masks this abort from the application by re-reading the end of file, and re-issuing the write at the new offset.

The mechanism that retries transactions is a thin layer between the WTF client library and the user's application. Each API call the application makes is logged in this layer by recording the arguments provided to the call and the value returned from the call. Should a metadata transaction abort during the WTF transaction commit, the WTF client library replays each operation from the log using the originally supplied arguments. If any replayed operation returns a value different from the logged call, the WTF transaction signals an abort to the application. Otherwise, WTF will commit the metadata changes from the replayed log to HyperDex. This process repeats as necessary until the metadata transaction, and, thus, the WTF transaction, commit, or a replayed operation triggers an WTF abort. This guarantees that WTF transactions are lockfree with zero spurious aborts.

To reduce the size of the replay log, the replay log refers to bytes of data that pass through the interface using slice pointers instead of copying the data. For example, a write of 100 MB will not be copied into the log; instead, the WTF client library writes the 100 MB to the requisite number of servers, and records the slice pointers in the log. Similarly, reads record slice pointers retrieved from the metadata, and not the slices themselves.

2.7 Locality-Aware Slice Placement

As an optimization, the WTF client library carefully places writes to the same region near each other on the storage servers to simultaneously improve locality for readers and to improve the efficiency of metadata compaction. When an application writes to a file sequentially, the locality-aware placement algorithm ensures that, with high probability, writes that appear consecutively in the metadata list will be consecutive on the storage servers' disks. During metadata compaction, the slice pointers for these consecutive writes are replaced by a single slice pointer that directly refers to the entire contiguous sequence of bytes on each storage server.

Two levels of consistent hashing [23] make it unlikely that two writes will map to the same backing files on the same storage server unless they are for the same metadata region. The WTF client library chooses the servers for each write by using consistent hashing across the list of storage servers. The client then provides the slice and identity of the metadata region to these servers, which use a different consistent hashing algorithm to map the write to disk. When collisions in the hash space do inevitably occur, it is unlikely that the colliding writes are issued so close in time as to be totally interleaved on disk in a way that eliminates opportunities for optimization.

2.8 Metadata Compaction and Defragmentation

The client library automatically compacts metadata during read and write operations to improve efficiency of future read and write operations. During write operations, the client library tracks the number of bytes written to both the metadata and the data for each region. When the ratio of metadata to data in a region exceeds a pre-defined threshold, the library retrieves the metadata list for the region, compacts it as shown in Figure 2, and writes the newly compacted list. When reading, the client compacts the metadata list via the same process.

When metadata compaction alone cannot reduce the metadata to data ratio below the pre-defined threshold, the client library defragments the list by rewriting the data. The library rewrites fragmented data within a region into one single slice and replaces the metadata list with a single pointer to this slice. For efficiency's sake, defragmentation happens only on read, not on writes,because the client library necessarily reads the fragmented slices to fulfill the read; it can rewrite the slices without the overall system paying the cost of reading the fragmented slices twice. This mechanism is unused in the common case because locality-aware slice placement avoids fragmentation.

2.9 Garbage Collection

WTF employs a garbage collection mechanism to prevent the number of unreferenced slices from growing without bound. Metadata compaction and defragmentation ensures that metadata will not grow without bound, but in the process creates garbage slices that are not referenced from anywhere in the filesystem.

Because WTF performs all bookkeeping within the metadata storage, storage servers cannot directly know which portions of its local data are garbage. One possible way to inform the storage servers would be to maintain a reference count for each slice. This method, however, would require that the reference count on the storage server be maintained within the scope of the metadata transactions. Doing so, while not infeasible, would significantly complicate WTF's design and require custom transaction handling on the storage servers.

Instead of reference counting, WTF periodically scans the entire filesystem metadata and constructs a list of in-use slice pointers for each storage server. For simplicity of implementation, these lists are stored in a reserved directory within the WTF filesystem so that they need not be maintained in memory or communicated out of band to the storage servers. Storage servers link the WTF client library and read the list of in-use slices to discover unused regions in their local storage space. The garbage collection mechanism runs periodically at a configurable interval that exceeds the longest-possible runtime of a transaction. Storage servers do not collect an unused slice until it appears in two or more consecutive scans.

Storage servers implement garbage collection by creating sparse files on the local disk. To compress a file containing garbage slices, a storage server rewrites the file, seeking past each unused slice. This creates a sparse file that occupies disk space proportional to the in-use slices it contains. Files with the most garbage are the most efficient to collect, because the garbage collection thread seeks past large regions of garbage and only writes the small number of remaining slices. Backing files with little garbage incur much more I/O, because there are more in-use slices to rewrite. WTF chooses the file with the most garbage to compact first, because it will simultaneously delete the most garbage and incur the least I/O. Some filesystems enable applications to selectively punch holes in the file without rewriting the data; although our implementation does not use these capabilities, an improved implementation could do so.

2.10 Fault Tolerance

WTF uses replication to add fault tolerance to the system. Changing WTF to be fault tolerant requires modifying the metadata lists' structure so that each entry references multiple replicas of the same data, each with a different slice pointer. On the write path, writers create multiple replica slices on distinct servers and append their pointers atomically as one list entry. Readers may read from any replica, as they hold identical data.

The metadata storage derives its fault tolerance from the guarantees offered by HyperDex. Specifically, that it can tolerate f concurrent failures for a user-configurable value of f. HyperDex uses value-dependent chaining to coordinate between the replicas and manage recovery from failures [16].

3 Implementation

Our implementation of WTF implements the file slicing abstraction. The implementation is approximately 30 k lines of code written. It relies upon HyperDex with transactions, which is approximately 85 k lines of code, with an additional 37 k lines of code of supporting libraries written for both projects. The replicated coordinator for both HyperDex and WTF is an additional 19 k lines of code. Altogether, WTF constitutes 171 k lines of code that were written for WTF or HyperDex.

WTF's fault tolerant coordinator maintains the list of storage servers and a pointer to the HyperDex cluster. It is implemented as a replicated object on top of Replicant, a Paxos-based replicated state machine service. The coordinator consists of just 960 lines of code that are compiled into a dynamically linked library that is passed to Replicant. Replicant deploys multiple copies of the library, and sequences function calls into the library.

4 Evaluation

To evaluate WTF, we will look at a series of both end-to-end and micro benchmarks that demonstrate WTF under a variety of conditions. The first part of this section looks at how the features of WTF may be used to implement a variety of end-to-end applications. We will then look at a series of microbenchmarks that characterize the performance of WTF's conventional filesystem interface.

All benchmarks execute on a cluster of fifteen dedicated servers. Each server is equipped with two Intel Xeon 2.5 GHz L5420 processors, 16 GB of DDR2 memory with ECC, and between 500 GB and 1 TB SATA spinning-disks. The servers are connected with gigabit ethernet via a single top of rack switch. Installed on each server is 64-bit Ubuntu 14.04, HDFS from Apache Hadoop 2.7, and WTF with HyperDex.

For all benchmarks, HDFS and WTF are configured similarly. Both systems are deployed with three nodes reserved for the meta-data—a single HDFS name node, or a HyperDex cluster—and the remaining twelve servers are allocated as storage nodes for the data. Clients are spread across the twelve storage nodes. Except for changes necessary to achieve feature parity, both systems were deployed in their default configuration. To bring the semantics of HDFS up to par with WTF, each `write` is followed by an `hflush` call to ensure that the write is flushed from the client-side buffer to HDFS. The `hflush` ensures that writes are visible to readers, and does *not* flush to disk. This is analogous to changing from the C library's `fwrite` to a UNIX `write` in a traditional application. The resulting guarantees are equivalent to those provided by WTF.

Additionally, in order to work around a bug with append operations [5], the HDFS block size was set to 64 MB. Without this change to the configuration, HDFS can report an out-of-disk-space condition when only 3% of the disk space is in use. Instead of gracefully handling the condition and falling back to other replicas as is done in WTF, the failure cascades and causes multiple writes to fail, making it impossible to complete some benchmarks. The change is unlikely to impact the performance of data nodes because the increase from 64 MB to

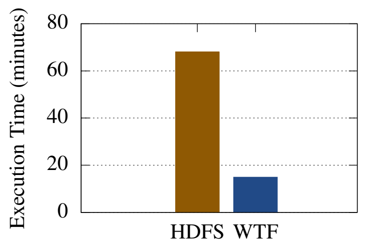


Figure 4: Total execution time for sorting 100 GB with map-reduce (512 kB rec.).

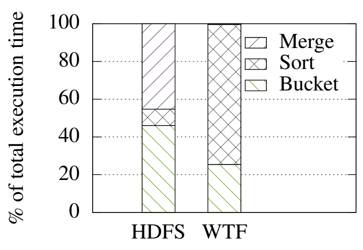


Figure 5: Execution time of sort broken down by stage of the map-reduce.

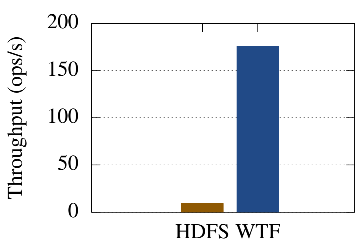


Figure 6: A concurrent work queue implementation.

128 MB was not motivated by performance [6]. WTF is also configured to use 64 MB regions.

Except where otherwise noted, both systems replicate all files such that two copies of the file exist. This allows the filesystem to tolerate the failure of any one storage server throughout the experiment without loss of data or availability. It is possible to tolerate more failures so long as all the replicas for a file do not fail simultaneously.

4.1 Applications

This section examines multiple applications that each demonstrate a different aspect of WTF's feature set.

Map Reduce: Sorting MapReduce [15] applications often build on top of filesystems like HDFS and GFS. In MapReduce, sorting a file is a three-step process that breaks the sort into two map jobs followed by a reduce job. The first map task partitions the input file into buckets, each of which holds a disjoint, contiguous section of the keyspace. These buckets are sorted in parallel by the second map task. Finally, the reduce phase concatenates the sorted buckets to produce the sorted output.

Each intermediate step of this application is written to the filesystem and the entire data set will be read or written several times over. Here, WTF's file slicing API can improve the efficiency of the application by reducing this excessive I/O. Instead of reading and writing whole records, WTF-based sort uses `yank` and `paste` to rearrange records. File slicing eliminates almost all I/O of the reduce phase using a `concat` operation.

Empirically, file slicing operations improve the running time of WTF-based sort. Figure 4 shows the total running time of both systems to sort a 100 GB file consisting of 500 kB records indexed by 10 B keys that were generated uniformly at random. In this benchmark, the intermediate files are written without replication because they may easily be recomputed from the input. We can see that WTF sorts the entire file in one fourth the time taken to perform the same task on HDFS.

The speedup is largely attributable file-slicing. From Figure 5, we can see that the WTF-based sorting application spends less time in the partitioning and merging steps than the HDFS-based sort. HDFS spends the majority of its execution time performing I/O tasks; just 8.5% of execution time is spent in the CPU-intensive sort task. In contrast, WTF spends 74.1% of its time in the CPU intensive task and seconds in the merge task.

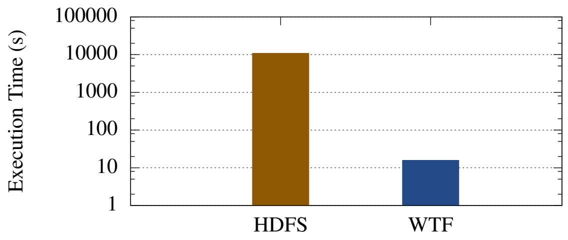


Figure 7: Time taken to generate a readily-playable video file from individual scenes.

Work Queue Work queues are a common component of large scale applications. Large work units may be durably written to the queue and handled by the application at a later point in time in FIFO order.

One simple implementation of a work queue is to use an append-only file as the queue itself. The application appends each work unit to the file, and can dequeue from the work queue by reading through the file sequentially—the file itself encodes the FIFO nature of the queue. This benchmark consists of an application with multiple writers that concurrently write to a single file on the filesystem. Each work unit is 1 MB in size and written atomically. The application runs on each client server, for a total of twelve application instances.

Figure 6 shows the aggregate throughput for the work queue built on top of both HDFS and WTF. We can see that WTF's throughput is 19× that of HDFS for this workload. Each work unit is saved to WTF in 55 ms, while the application built on HDFS waits 1.3 s on average to enqueue each work unit.

Image Host Image hosting sites, such as flickr or imgur have become the de-facto way of sharing images on the Internet. While imgur's implementation serves images from Amazon S3, Facebook's image serving solution, called Haystack [9], stores multiple photos in a single file to reduce the costs of reading and maintaining metadata. In Haystack, servers read into memory a map of the photos' locations on disk so that reading a single photo from disk does not entail any additional disk reads.

This example application models an imgur-like website built using the multi-photo file technique used within

Haystack. Photos are written to multi-gigabyte files, each of which has a footer mapping photos to their offsets in the files. The application loads this map into memory so that it may serve requests by locating the file's offset within the map, and then reading the file directly from the offset in the filesystem.

To better simulate a real photo-sharing website, photos are randomly generated to match the size of photos served by imgur. The distribution of photo sizes was collected from the front page of imgur.com over a 24-hour period. Because imgur serves both static images and gifs, the size of photos varies widely. Median image size is 332 kB, while the average image size is 8.5 MB. Because the precise request distribution of requests is not available from imgur, the workload re-uses the Zipf request distribution specified for YCSB workloads [13]. For this workload, we measured that WTF achieves 88.8% the throughput of the same application on top of HDFS. The performance difference is largely attributable to the reads of smaller files. As we will explore in the microbenchmarks section, WTF needs further optimization for small read and write operations.

Video Editing WTF's file slicing API can be used to reorganize large files with orders of magnitude less I/O. One particular domain where this can be useful is video editing of high-definition raw video. Such videos tend to be large in size, and will be rearranged frequently during the editing process. While specialized applications can edit and then play back videos, WTF enables another point in the design space.

This application uses WTF's file slicing to move scenes around in a video file without physically rewriting the video. The chief benefit of this design, over editors on existing filesystems, is that an off-the-shelf video player can play the edited video file because it is in a standard container format. To benchmark this application, we used our video editor to randomly rearrange the scenes in a 2 h movie, such that the movie out of chronological order. The source material was 1080p raw video dumped from a Bluray disk. Overall the raw video/audio occupies approximately 377 GB or 52 MB/s. Figure 7 shows the time taken to rewrite the file using HDFS's conventional API compared to WTF's file-slicing API. WTF takes three orders of magnitude less time to make a file readable—on the order of seconds—while conventional techniques require nearly three hours.

Sandboxing The transactional API of WTF makes it easy to use the filesystem as a sandbox where tasks may be committed or aborted depending on their outcome. The WTF implementation includes a FUSE module that enables users to mount the filesystem as if it were a local filesystem. This enables shell navigation of the filesystem hierarchy and allows regular applications to read and write WTF filesystems without modification. In addition to implementing the full filesystem interface, the FUSE bridge exposes special `ioctls` to permit users to control transactions. Users may begin, abort, or commit transactions via command-line tools that wrap these `ioctls`.

```
# wtf fuse ./mnt
# cd ./mnt
# wtf fuse-begin-transaction
# ls
/data.0000  /data.0001
/data.0002  /data.0003
....
# rm *
# ls
# wtf fuse-abort-transaction
# ls
/data.0000  /data.0001
/data.0002  /data.0003
....
```

Figure 8: WTF's transactional functionality enables users to manipulate the filesystem in isolation.

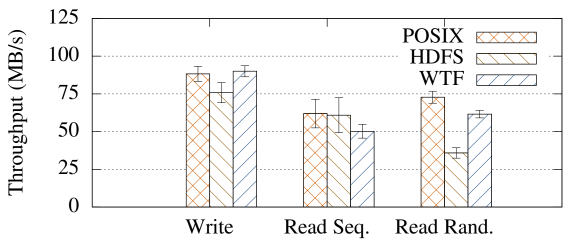


Figure 9: Performance of a one-server deployment of HDFS and WTF compared with the ext4 filesystem. Error bars indicate the standard error of the mean across seven trials.

The transactional features of the FUSE bridge enables users to perform risky actions within the context of a transaction; the transactional isolation provides a degree of safety users would otherwise not be afforded. The actions taken by the user are not visible until the user commits, and should the user abort, the actions will never be persisted to the filesystem. Figure 8 shows a sample interaction with an WTF filesystem containing data for a sample research project. We can see that the user begins a transaction and inadvertently removes all of the research data. Because the errant `rm` command happened in a transaction, the data remains untouched.

4.2 Micro Benchmarks

In this section we examine a series of microbenchmarks that quantify the performance of the POSIX API for both HDFS and WTF. Here HDFS serves as a gold-standard. With ten years of active development, and deployment across hundreds of nodes, including large deployments at both Facebook and LinkedIn [12], HDFS provides a reasonable estimate of distributed filesystem performance. Although we cannot expect WTF to grossly outperform HDFS—both systems are limited by the speed of the hard disks in the cluster—we can use the degree to which WTF and HDFS differ in performance to estimate the overheads present in WTF's design.

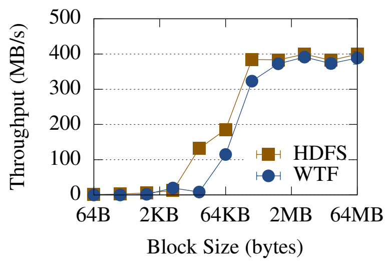


Figure 10: Throughput of a sequential write workload. Error bars report the standard error of the mean across seven trials[1].

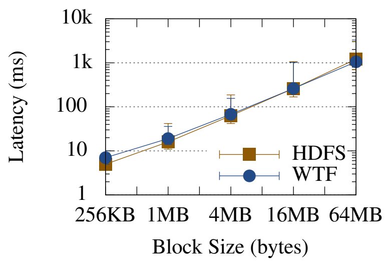


Figure 11: Median latency of write operations. Error bars report the 5th and 95th percentile latencies.

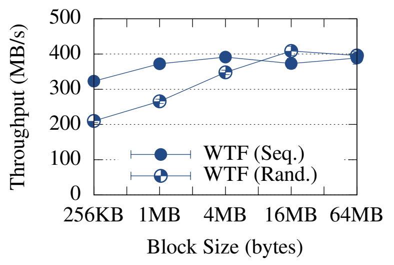


Figure 12: Throughput of a random write workload. Error bars report the standard error of the mean across seven trials.

Setup The workload for these benchmarks is generated by twelve distinct clients, one per storage server in the cluster, that all work in parallel. This configuration was chosen after experimentation because additional clients do not significantly increase the throughput, but do increase the latency significantly. All benchmarks operate on 100 GB of data, or over 16 GB per machine once replication is accounted for. This is large enough that our workload blocks on disk on Linux [25].

Single server performance This first benchmark executes on a single server to establish the baseline performance of a one node cluster. Here, we'll compare the two systems to each other and the same workload implemented on a local ext4 filesystem. The comparison to a local filesystem provides an upper bound on performance. To reduce the impact of round trip time in each distributed system the client and storage server are collocated. Figure 9 shows the throughput of write and read operations in the one-server cluster. From this we can see that the maximum throughput of a single node is 87 MB/s, which means the total throughput of the cluster peaks at approximately 1 GB/s.

Sequential Writes WTF guarantees that all readers in the filesystem see a write upon its completion. This benchmark examines the impact that write size has on the aggregate throughput achievable for filesystem-based applications. Figure 10 shows the results for block sizes between 64 B and 64 MB. For writes greater than 1 MB, WTF achieves 97% the throughput of HDFS. For 256 kB writes, WTF achieves 84% of the throughput of HDFS.

The latency for the two systems is similar, and directly correlated with the block size. Figure 11 shows the latency of writes across a variety of block sizes. We can see that WTF's median latency is very close to HDFS's median latency for larger writes, and that the 95th percentile latency for WTF is often lower than for HDFS.

Random Writes WTF enables applications to write at random offsets in a file without restriction. Because HDFS does not support random writes, we cannot use it as a baseline; instead, we will compare against the sequential write performance of WTF.

[1]Blocks $<$256 kB wrote smaller files to limit execution time.

Figure 12 shows the aggregate throughput achieved by clients writing to random offsets within WTF files. We see that the random write throughput is always within a factor of two of the sequential throughput, and that throughput converges as the size of the writes approaches 8 MB.

Because the common case for a sequential write and a random write in WTF differ only at the stage where metadata is written to HyperDex, we expect that such a difference in throughput is directly attributable to the metadata stage. HyperDex provides lower latency variance to applications with a small working set than applications with a large working set with no locality of access. We can see the difference this makes in the tail latency of WTF writes in Figure 13, which shows the median and 99th percentile latencies for both the sequential and random workloads. The median latency for both workloads is the same for all block sizes. For block sizes 4 MB and larger, the 99th percentile latencies are approximately the same as well. Writes less than 4 MB in size exhibit a significant difference in 99th percentile latency between the sequential and random workloads. These smaller writes spend more time updating HyperDex than writing to storage servers. We expect that further optimization of HyperDex would close the gap between sequential and random write performance.

Sequential Reads Batch processing applications often read large input files sequentially during both the map and reduce phases. Although a properly-written application will double-buffer to avoid small reads, the filesystem should not rely on such behavior to enable high throughput. This experiment shows the extent to which WTF can be used by batch applications by reading through a file sequentially using a fixed-size buffer.

Figure 14 shows the aggregate throughput of concurrent readers reading through a 100 GB of data. We can see that for all read sizes, WTF's throughput is at least 80% the throughput of HDFS. The throughput reported here is double the throughput reported in the write benchmarks because only one of the two active replicas is consulted on each read. For smaller reads, WTF's throughput matches that of HDFS. The difference at larger sizes

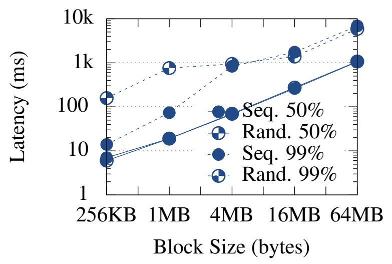


Figure 13: 50th/99th percentile latencies for sequential and random WTF writes.

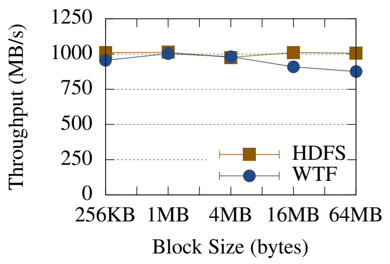


Figure 14: Throughput of a sequential read workload. Error bars report the standard error of the mean across seven trials.

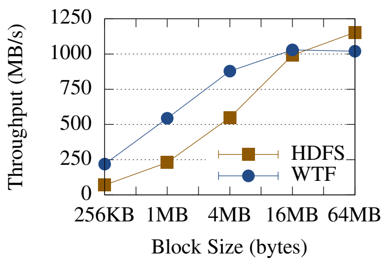


Figure 15: Throughput of a random read workload. Error bars indicate the standard error of the mean across seven trials.

is largely an artifact of the implementations. HDFS uses readahead on both the clients and storage servers in order to improve throughput for streaming workloads. By default, the HDFS readahead is configured to be 4 MB, which is the point at which the systems start to exhibit different characteristics. Our preliminary WTF implementation does not have any readahead mechanism, and exhibits lower throughput.

Random Reads Applications built on a distributed filesystem, such as key-value stores or record-oriented applications often require random access to the files. Figure 15 shows the aggregate throughput of twelve concurrent random readers reading from randomly chosen offsets within 100 GB of data. We can see that for reads of less than 16 MB, WTF achieves significantly higher throughput—at its peak, WTF's throughput is 2.4× the throughput of HDFS. Here, the readahead and client-side caching that helps HDFS with larger sequential read workloads adds overhead to HDFS that WTF does not incur. The 95th percentile latency of a WTF read is less than the median latency of a HDFS read for block sizes less than 4 MB.

Scaling Workload This experiment varies the number of clients writing to the filesystem to explore how concurrency affects both latency and throughput. This benchmark employs the workload from the sequential-write benchmark with a 4 MB write size and a variable number of workload-generating clients.

Figures 16 and 17 shows the resulting throughput and latency for between one and twelve clients. We can see that the single client performance is approximately 60 MB/s, while twelve clients sustain an aggregate throughput of approximately 380 MB/s. WTF's throughput is approximately the same as the throughput of HDFS for each data point. Running the same workload with forty-eight clients did not increase the throughput of either system beyond the throughput achieved with twelve clients, but did result in higher latency.

Fault Tolerance WTF's fault tolerance mechanism enables it to rapidly recover from failures. To demonstrate this mechanism, this benchmark performs sequential writes at a target throughput of 200 MB/s. Figure 18 shows the throughput of the benchmark over time. Thirty seconds into the benchmark, one storage server is taken offline; ten seconds later, the coordinator reconfigures the system to remove the failed storage server. In the time between the failure and reconfiguration, clients may try to use the failed server, fail to write to it, and fall back to another server. This increased effort is reflected in the lower throughput between failure and reconfiguration. After reconfiguration, throughput returns to to its rate before the failure. During the entire experiment, no writes failed, and the cluster as a whole remained available.

Garbage Collection This benchmark calculates the overhead of garbage collection on a storage server. As mentioned in Section 2.9, it is more efficient to collect files with more garbage than files with less garbage, and WTF preferentially garbage collects these larger files. Figure 19 shows the rate at which the cluster can collect garbage, for varying amounts of randomly located garbage, when all resources are dedicated to the task. We can see that when the cluster consists of 90% garbage, the cluster can reclaim this garbage at a rate of over 9 GB of garbage per second, because it need only write 1 GB/s to reclaim the garbage.

It is, however, impractical to dedicate all resources to garbage collection; instead, WTF dedicates only a fraction of I/O to the task. Storage servers initiate garbage collection when disk usage exceeds a configurable threshold, and ceases when the amount of garbage drops below 20%. Figure 19 shows that the maximum overhead required to maintain the system below this threshold is 4%.

Small Writes WTF's design is optimized for larger writes. The performance of smaller writes will largely be determined by the cost of updating the metadata. Writing a slice to the storage servers requires just one round trip because replicas are written to in parallel. Writing to the metadata store requires one round trip between client and the cluster, and multiple round trips within the cluster to propagate and commit the data. Further each write to the metdata requires writing approximately 50 B to HyperDex, so as writes to WTF shrink in size, the dominating

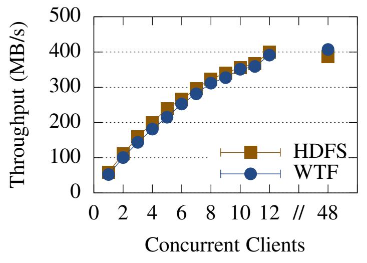


Figure 16: Throughput for varying numbers of writers. Error bars show the standard error of the mean across seven trials.

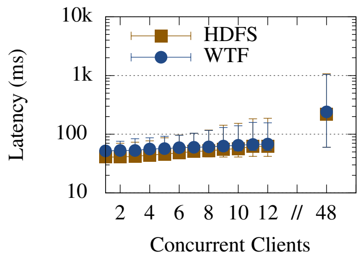


Figure 17: Median write latency for varying numbers of writers. Error bars show the 5th and 95th percentile latencies.

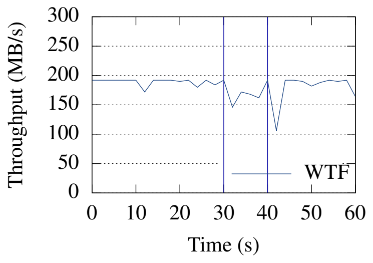


Figure 18: WTF tolerates failures—the failure occurs at the 30s mark—without a loss of availability.

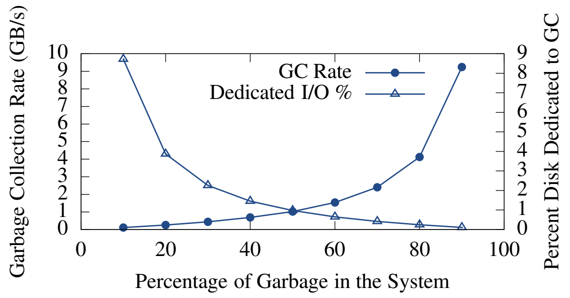


Figure 19: The maximum rate of garbage collection is positively correlated with the amount of garbage to be collected. Consequently, WTF dedicates a small fraction of its overall I/O to garbage collection.

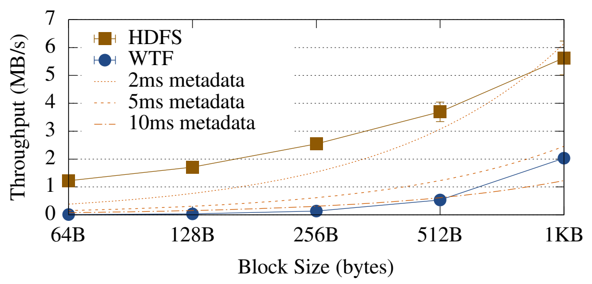


Figure 20: The time spent in metadata operations establishes an upper bound on the total throughput achievable by the system. This figure plots a portion of Figure 10 and theoretical maximum throughput for multiple metadata latencies.

cost becomes related to metadata.

Figure 20 focuses on a portion of the experiment shown in Figure 10, specifically writes less than 1 kB in size. HDFS achieves 140× higher throughput for 64 B writes, while the difference is only a factor of 2.8× for 1 kB writes. The figure also shows the calculated theoretical maximum throughput when the latency involved in writing to the metadata server is 2 ms, 5 ms, and 10 ms. This shows that the throughput of small operations is largely dependent upon the latency of metadata operations. Most workloads can avoid small operations with client side buffering, and further optimization of the metadata component could improve the throughput for small WTF writes.

5 Related Work

Filesystems have been an active research topic since the earliest days of systems research. Existing approaches related to WTF can be broadly classified into two categories based upon their design.

Distributed filesystems Distributed filesystems expose one or more units of storage over a network to clients. AFS [22] exports a uniform namespace to workstations, and stores all data on centralized servers. Other systems [21, 31, 33], most notably xFS [3] and Swift [10] stripe data across multiple servers for higher performance than can be achieved with a single disk. Petal [24] provides a virtual disk abstraction that clients may use as a traditional block device. Frangipani [38] builds a filesystem abstraction on top of Petal. NASD [20] and Panasas [42] employ customized storage devices that attach to the network to store the bulk of the metadata. In contrast to these systems, WTF provides transactional guarantees that can span hundreds or thousands of disks because its metadata storage scales independently of the number of storage servers.

Farsite [1] separates data from metadata to implement a byzantine fault tolerant filesystem where only the metadata replicas employ BFT algorithms. WTF uses a similar insight to leverage the transactional guarantees provided by the metadata storage to enable transactional guarantees to extend across the whole filesystem.

Recent work focuses on building large-scale datacenter-centric filesystems. GFS [19] and HDFS [4] employ a centralized master server that maintains the metadata, mediates client access, and coordinates the storage servers. Salus [41] improves HDFS to support storage and computation failures without loss of data, but retains the central metadata server. This centralized master approach, however, suffers from scalability bottlenecks inherent to the limits of a single server [27]. WTF overcomes the metadata scalability bottleneck using the scalable HyperDex key-value store [17].

CalvinFS [39] focuses on fast metadata management using distributed transactions in the Calvin [40] transaction processing system. Transactions in CalvinFS are limited, and cannot do read-modify-write operations on the filesystem without additional mechanism. Further, CalvinFS addresses file fragmentation using a heavy-weight garbage collection mechanism that entirely rewrites fragmented files; in the worst case, a sequential writer could incur I/O that scales quadratically in the size of the file. In contrast, WTF provides fully general transactions and carefully arranges data to improve sequential write performance.

Another approach to scalability is demonstrated by Flat Datacenter Storage [29], which enables applications to access any disk in a cluster via a CLOS network with full bisection bandwidth. To eliminate the scalability bottlenecks inherent to a single master design, FDS stores metadata on its tract servers and uses a centralized master solely to maintain the list of servers in the system. Blizzard [28] builds block storage, visible to applications as a standard block device, on top of FDS, using nested striping and eventual durability to service the smaller writes typical of POSIX applications. These systems are complementary to WTF, and could implement the storage server abstraction.

"Blob" storage systems behave similarly to file systems, but with a restricted interface that permits creating, retrieving, and deleting blobs, without efficient support for arbitrarily changing or resizing blobs. Facebook's f4 [37] ensures infrequently accessed files are readily available. Pelican [8] enables power-efficient cold storage by over provisioning storage, and selectively turning on subsets of disks to service requests. The design goals of these systems are different from the applications that WTF enables; WTF could be used in front of these systems to generate, maintain, and modify data before placing it into blob storage.

Transactional filesystems Transactional filesystems enable applications to offload much of the hard work relating to update consistency and durability to the filesystem. The QuickSilver operating system shows that transactions across the filesystem simplify application development [32]. Further work showed that transactions could be easily added to LFS, exploiting properties of the already-log-structured data to simplify the design [35]. Valor [36] builds transaction support into the Linux kernel by interposing a lock manager between the kernel's VFS calls and existing VFS implementations. In contrast to the transactions provided by WTF, and the underlying HyperDex transactions, these systems adopt traditional pessimistic locking techniques that hinder concurrency.

Optimistic concurrency control schemes often enable more concurrency for lightly-contended workloads. PerDiS FS adopts an optimistic concurrency control scheme that relies upon external components to reconcile concurrent changes to a file [18]. This allows users and applications to concurrently work on the same file. Liskov and Rodrigues show that much of the overhead of a serializable filesystem can be avoided by running read-only transactions in the recent past, and employing an optimistic protocol for read-write transactions [26]. WTF builds on top of HyperDex's optimistic concurrency and provides operations such as `append` that avoid creating conflicts between concurrent transactions.

WTF is not the first system to choose to employ a transactional datastore as part of its design. Inversion [30] builds on PostgreSQL to maintain a complete filesystem. KBDBFS [36] and Amino [43] both build on top of BerkeleyDB; the former is an in-kernel implementation of BerkeleyDB, while the latter eschews the complexity and takes a performance hit with a userspace implementation. WTF differs from these designs in that it stores solely the metadata in the transactional data store; data is stored elsewhere and not managed within the transactional component.

Stasis [34] makes the argument that no one design support all use cases, and that transactional components should be building blocks for applications. WTF's approach is similar: HyperDex's transactions are used as a base primitive for managing WTF's state, and WTF supports a transactional API. Applications built on WTF can use this API to achieve their own transactional behavior.

6 Conclusion

This paper described the Warp Transactional Filesystem (WTF), a new distributed filesystem that enables applications to operate on multiple files transactionally without requiring complex application logic. A new filesystem abstraction called *file slicing* further boosts performance by completely changing the filesystem interface to focus on metadata manipulation instead of data manipulation. Together, these features are a potent combination that enables a new class of high performance applications.

A broad evaluation shows that WTF achieves throughput and latency similar to industry-standard HDFS, while simultaneously offering stronger guarantees and a richer API. Sample applications show that WTF is usable in practice, and applications will often those built on a traditional filesystem—sometimes by orders of magnitude.

Acknowledgments

We'd like to thank our shepherd, Jon Howell, our anonymous reviewers, and Robbert van Renesse for their helpful feedback. We would like to thank Sean Ogden for his assistance in implementing WTF. This work is supported by the National Science Foundation under Grant No. CNS-1518779 and the Facebook Graduate Fellowship program. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

[1] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger Wattenhofer. FARSITE: Federated, Available, And Reliable Storage For An Incompletely Trusted Environment. In Proceedings of the *Symposium on Operating System Design and Implementation,* Boston, Massachusetts, December 2002.

[2] Amazon Web Services. Elastic Block Store. `http://aws.amazon.com/ebs/`.

[3] Thomas E. Anderson, Michael Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless Network File Systems. In Proceedings of the *Symposium on Operating Systems Principles,* pages 109-126, Copper Mountain, Colorado, December 1995.

[4] Apache Hadoop. `http://hadoop.apache.org/`.

[5] Apache Hadoop Jira. DFS Used Space Is Not Correct Computed On Frequent Append Operations. `https://issues.apache.org/jira/browse/HDFS-6489`.

[6] Apache Hadoop Jira. Increase The Default Block Size. `https://issues.apache.org/jira/browse/HDFS-4053`.

[7] Apache HBase. `http://hbase.apache.org/`.

[8] Shobana Balakrishnan, Richard Black, Austin Donnelly, Paul England, Adam Glass, David Harper, Sergey Legtchenko, Aaron Ogus, Eric Peterson, and Antony I. T. Rowstron. Pelican: A Building Block For Exascale Cold Data Storage. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 351-365, Broomfield, Colorado, October 2014.

[9] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding A Needle In Haystack: Facebook's Photo Storage. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 47-60, Vancouver, Canada, October 2010.

[10] Luis-Felipe Cabrera and Darrell D. E. Long. Swift: Using Distributed Disk Striping To Provide High I/O Data Rates. In *Computing Systems,* 4(4):405-436, 1991.

[11] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, and Robert Gruber. Bigtable: A Distributed Storage System For Structured Data. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 205-218, Seattle, Washington, November 2006.

[12] Asaf Cidon, Stephen M. Rumble, Ryan Stutsman, Sachin Katti, John K. Ousterhout, and Mendel Rosenblum. Copysets: Reducing The Frequency Of Data Loss In Cloud Storage. In Proceedings of the *USENIX Annual Technical Conference,* San Jose, California, June 2013.

[13] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems With YCSB. In Proceedings of the *Symposium on Cloud Computing,* pages 143-154, Indianapolis, Indiana, June 2010.

[14] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 261-264, Hollywood, California, October 2012.

[15] Jeffrey Dean and Sanjay Ghemawat. MapReduce: A Flexible Data Processing Tool. In *Communications of the ACM,* 53(1):72-77, 2010.

[16] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A Distributed, Searchable Key-Value Store. In Proceedings of the *SIGCOMM Conference,* pages 25-36, Helsinki, Finland, August 2012.

[17] Robert Escriva, Bernard Wong, and Emin Gün Sirer. Warp: Lightweight Multi-Key Transactions For Key-Value Stores. Cornell University, Ithaca, Technical Report, 2013.

[18] João Garcia, Paulo Ferreira, and Paulo Guedes. The PerDiS FS: A Transactional File System For A Distributed Persistent Store. In Proceedings of the *European SIGOPS Workshop,* pages 189-194, Sintra, Portugal, September 1998.

[19] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In Proceedings of the *Symposium on Operating Systems Principles,* pages 29-43, Bolton Landing, New York, October 2003.

[20] Garth A. Gibson, David Nagle, Khalil Amiri, Jeff Butler, Fay W. Chang, Howard Gobioff, Charles Hardin, Erik Riedel, David Rochberg, and Jim Zelenka. A Cost-Effective, High-Bandwidth Storage Architecture. In Proceedings of the *Architectural Support for Programming Languages and Operating Systems,* pages 92-103, San Jose, California, October 1998.

[21] John H. Hartman and John K. Ousterhout. The Zebra Striped Network File System. In *ACM Transactions on Computer Systems,* 13(3):274-310, 1995.

[22] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, Mahadev Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale And Performance In A Distributed File System. In *ACM Transactions on Computer Systems,* 6(1):51-81, 1988.

[23] David R. Karger, Eric Lehman, Frank Thomson Leighton, Rina Panigrahy, Matthew S. Levine, and Daniel Lewin. Consistent Hashing And Random Trees: Distributed Caching Protocols For Relieving Hot Spots On The World Wide Web. In Proceedings of the *ACM Symposium on Theory of Computing,* pages 654-663, El Paso, Texas, May 1997.

[24] Edward K. Lee and Chandramohan A. Thekkath. Petal: Distributed Virtual Disks. In Proceedings of the *Architectural Support for Programming Languages and Operating Systems,* pages 84-92, Cambridge, Massachusetts, October 1996.

[25] Linux Kernel Developers. Documentation For /proc/sys/vm/*. `https://www.kernel.org/doc/Documentation/sysctl/vm.txt`.

[26] Barbara Liskov and Rodrigo Rodrigues. Transactional File Systems Can Be Fast. In Proceedings of the *European SIGOPS Workshop,* page 5, Leuven, Belgium, September 2004.

[27] Kirk McKusick and Sean Quinlan. GFS: Evolution On Fast-Forward. In *Communications of the ACM,* 53(3):42-49, 2010.

[28] James W. Mickens, Edmund B. Nightingale, Jeremy Elson, Darren Gehring, Bin Fan, Asim Kadav, Vijay Chidambaram, Osama Khan, and Krishna Nareddy. Blizzard: Fast, Cloud-Scale Block Storage For Cloud-Oblivious Applications. In Proceedings of the *Symposium on Networked System Design and Implementation,* pages 257-273, Seattle, Washington, April 2014.

[29] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen S. Hofmann, Jon Howell, and Yutaka Suzue. Flat Datacenter Storage. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 1-15, Hollywood, California, October 2012.

[30] Michael A. Olson. The Design And Implementation Of The Inversion File System. In Proceedings of the *USENIX Winter Technical Conference,* pages 205-218, San Diego, California, January 1993.

[31] Frank B. Schmuck and Roger L. Haskin. GPFS: A Shared-Disk File System For Large Computing Clusters. In Proceedings of the *Conference on File and Storage Technologies,* pages 231-244, Monterey, California, January 2002.

[32] Frank B. Schmuck and James C. Wyllie. Experience With Transactions In QuickSilver. In Proceedings of the *Symposium on Operating Systems Principles,* pages 239-253, Pacific Grove, California, October 1991.

[33] Seagate Technology LLC. Lustre Filesystem. `http://lustre.org/`.

[34] Russell Sears and Eric A. Brewer. Stasis: Flexible Transactional Storage. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 29-44, Seattle, Washington, November 2006.

[35] Margo I. Seltzer. Transaction Support In A Log-Structured File System. In Proceedings of the *IEEE International Conference on Data Engineering,* pages 503-510, Vienna, Austria, April 1993.

[36] Richard P. Spillane, Sachin Gaikwad, Manjunath Chinni, Erez Zadok, and Charles P. Wright. Enabling Transactional File Access Via Lightweight Kernel Extensions. In Proceedings of the *Conference on File and Storage Technologies,* pages 29-42, San Francisco, California, February 2009.

[37] Muralidhar Subramanian, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Sivakumar Viswanathan, Linpeng Tang, and Sanjeev Kumar. F4: Facebook's Warm BLOB Storage System. In Proceedings of the *Symposium on Operating System Design and Implementation,* pages 383-398, Broomfield, Colorado, October 2014.

[38] Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A Scalable Distributed File System. In Proceedings of the *Symposium on Operating Systems Principles,* pages 224-237, Saint Malo, France, October 1997.

[39] Alexander Thomson and Daniel J. Abadi. CalvinFS: Consistent WAN Replication And Scalable Metadata Management For Distributed File Systems. In Proceedings of the *Conference on File and Storage Technologies,* pages 1-14, Santa Clara, California, February 2015.

[40] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. Calvin: Fast Distributed Transactions For Partitioned Database Systems. In Proceedings of the *SIGMOD International Conference on Management of Data,* pages 1-12, Scottsdale, Arizona, May 2012.

[41] Yang Wang, Manos Kapritsos, Zuocheng Ren, Prince Mahajan, Jeevitha Kirubanandam, Lorenzo Alvisi, and Mike Dahlin. Robustness In The Salus Scalable Block Store. In Proceedings of the *Symposium on Networked System Design and Implementation,* pages 357-370, Lombard, Illinois, April 2013.

[42] Brent Welch, Marc Unangst, Zainul Abbasi, Garth A. Gibson, Brian Mueller, Jason Small, Jim Zelenka, and Bin Zhou. Scalable Performance Of The Panasas Parallel File System. In Proceedings of the *Conference on File and Storage Technologies,* pages 17-33, San Jose, California, February 2008.

[43] Charles P. Wright, Richard P. Spillane, Gopalan Sivathanu, and Erez Zadok. Extending ACID Semantics To The File System. In *ACM Transactions on Storage,* 3(2), 2007.