WiscFS, a Client-Server File System

Natan Lidukhover, James Sorenson, Mia Weaver

February 2023

1 Introduction

`WiscFS` is a simple, crash-tolerant distributed file system that is 1. built on top of `unreliableFS`, a FUSE-based *fault-injection* file system implemented in user space and 2. padded with additional functionality mimicking the semantics and functionality established in `AFSv1` [How+88], [Bro21]. `AFS` is a distributed file system designed specifically for scale that also provides some guarantees on cache consistency.

The idea behind provisioning a simple and unreliable file system with `AFS`-like functionality is to enable support for concurrent file accesses across many clients, each of whom has a consistent local file cache, and to gracefully handle crash cases within the file server.

2 Implementation

WiscFS is made up of `AFS`-like modifications implemented on top of `unreliableFS`. These modifications are intended to grant WiscFS scalability and performance while maintaining cache consistency as it is defined in [How+88]. Specifically, these modifications are as follows.

1. **Multi-Client Support** : FS supports concurrent accesses of files by a variety of clients.
2. **Local File Caching** : performance enhancements from local caching of server files.
3. **Cache Consistency** : upon client `open()`, check if file needs to be re-fetched from server
4. **Last Closer Wins** : concurrent modifications resolved by flushing the *most recently modified* version of the concurrently opened file to the disk.
5. **Crash Recovery** : FS tolerates crashes and ensures crash consistency, meaning file operations are atomic. If a client crashes during a write to the server, the write is either fully committed or completely disregarded.

We use the Google's RPC mechanism, `gRPC`, to communicate between clients and servers, and `protobufs` to specify the serialization of communicated data. To implement these changes on top of `unreliableFS`, we add functionality to the available FUSE code base. For cache consistency, we use a scheme in which files are named according to and accessed by their `SHA1 hash`. This way, we minimize the possibility of file collisions and remove the need for a directory structure.

One of the key challenges to providing cache consistency is ensuring that 1. writes to concurrently opened files are handled and 2. a file opened from the cache is up-to-date with the server version. We implement a `last closer wins` policy in which the last write to a concurrently opened file committed to the server is updated as the server file copy. Cache consistency is ensured by calling back local files from the cache when opened if their last modification date or file size is

3 Performance Evaluation

We examine the performance of `WiscFS` across a variety of publicly available workloads as well as a series of custom made workloads, timing the file operations under each.

The simple custom workloads are as follows:

1. **Increasing File Size Writes** : repeated writes of file string of n = 2, 4, 8, ..., 524288 bytes to a file. Times of each individual file write are plotted against the bytes stored in the file.

2. **Increasing File Size Reads** : repeated reads of files containing file strings of n = 2, 4, 8, ..., 524288 bytes. Times of each individual file read are plotted against the bytes stored in the file.

3. **Repeated File Reads** : repeated reads of a fixed size file containing file strings of n = 524288 bytes. Times of each individual file read are plotted against the sequential access order.

And our results for these workloads were the following:

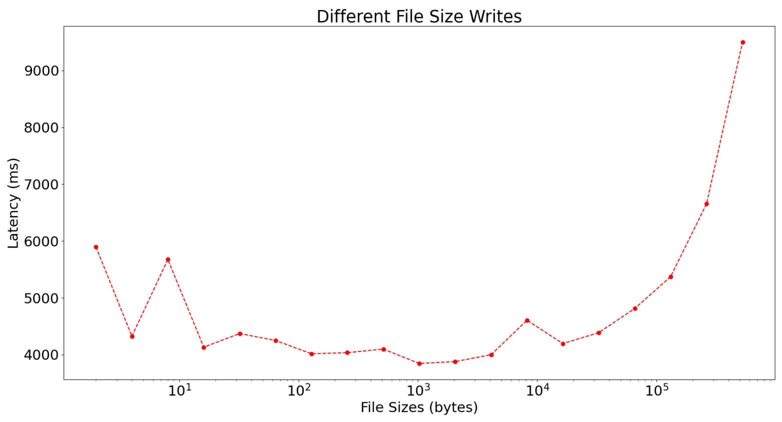

Figure 1: Increasing file writes

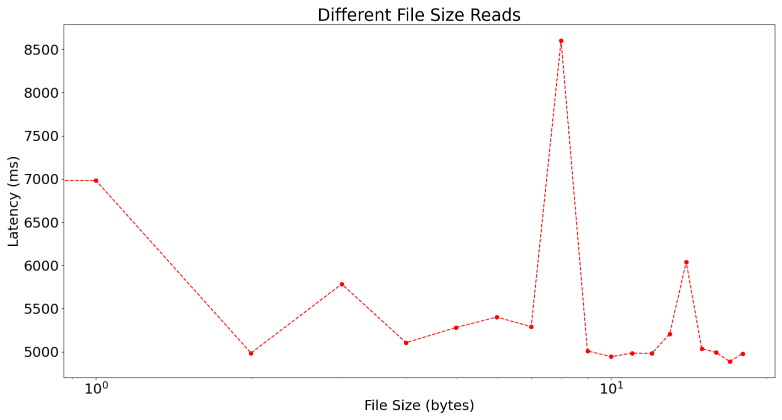

Figure 2: Increasing file reads

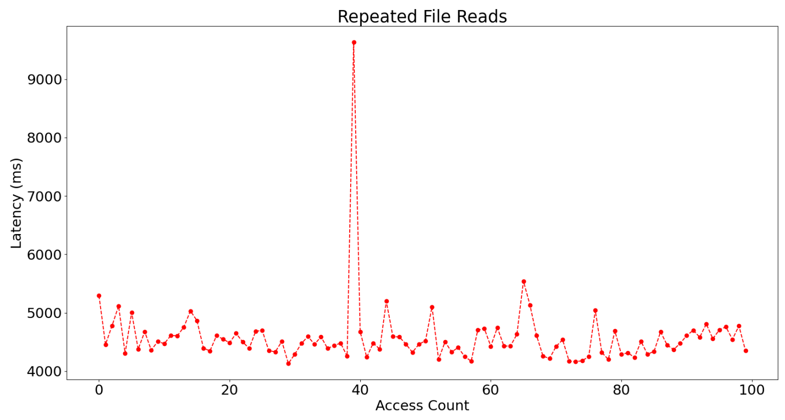

Figure 3: Repeated file reads

4 Functionality

4.1 Filebench

We successfully ran Filebench on our file system across all the given workloads.
Here is a table of the results:

Workload	**Ops**	**Ops/s**	**Throughput (MB/s)**	**Time/Op (ms/op)**
filemicro_create.f	1933	193.287	193.2	5.168
filemicro_createfiles.f	8397	839.641	0.3	1.188
filemicro_createrand.f	2178	311.121	139.7	3.207
filemicro_delete.f	5016	1003.128	0	15.373
filemicro_rread.f	1128	563.953	0.6	1.761
filemicro_rwritedsync.f	8282	1656.281	3.2	1.061
filemicro_seqread.f	9449	944.835	943.8	1.057
filemicro_seqwrite.f	1930	192.986	192.9	5.168
filemicro_statfile.f	57859	5785.491	0	3.435
filemicro_writefsync.f	117982	11797.393	92.1	0.084
fileserver.f	18462	1846.055	42.9	26.654
mongo.f	5957	595.66	2	1.675
varmail.f	126145	2102.575	7.1	7.599
webserver.f	21106	2110.427	11	23.378

And here are figures for graphical representations of this data:

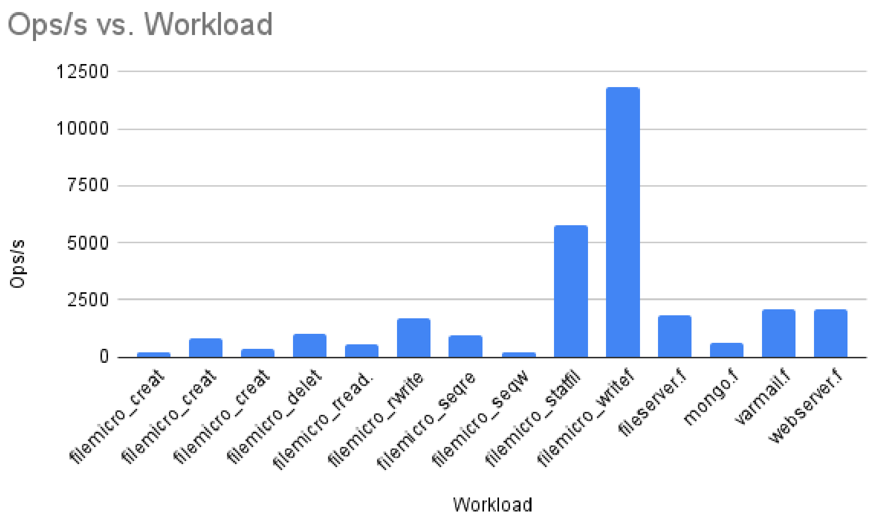

Figure 4: Filebench operations per second

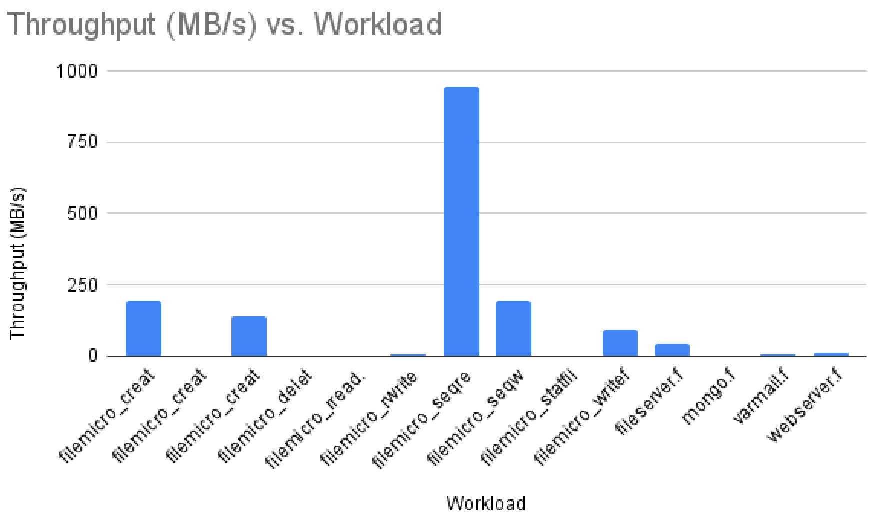

Figure 5: Filebench throughput in megabytes per second

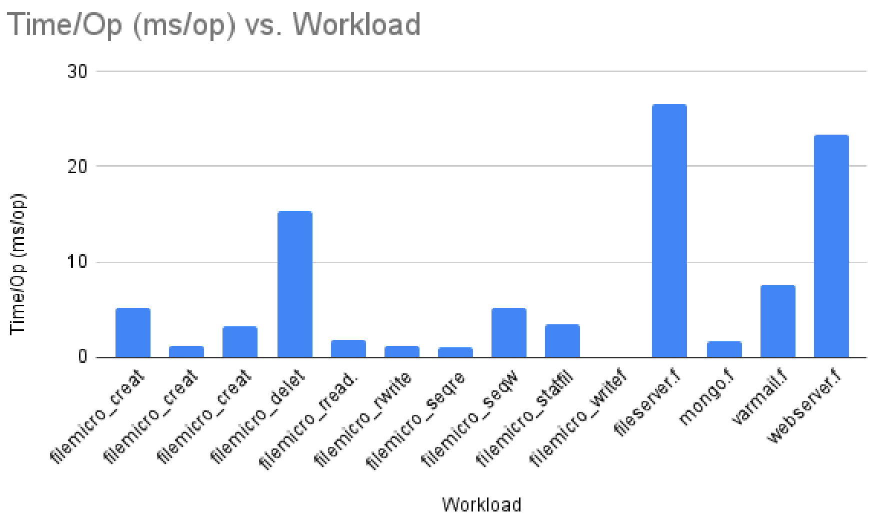

Figure 6: Filebench time to execute in seconds

In general, most workloads were not too demanding in terms of operations per second, but the writefile and statfile workloads clearly stressed the system most in this area. The seqread workload was by far the biggest standout in throughput, while time per operation was the most uniform distribution, where webserver and fileserver took the most time.

4.2 Applications

We timed making xv6 and leveldb with multiple different thread settings in our filesystem. These were the results:

Workload	Real (s)	User (s)	Sys (s)
xv6-public	5.434	2.239	0.429
leveldb-1 thread	76.227	27.141	4.576
leveldb-2 threads	43.774	35.200	5.649
leveldb-3 threads	39.304	47.082	7.221
leveldb-4 threads	35.200	53.034	7.687

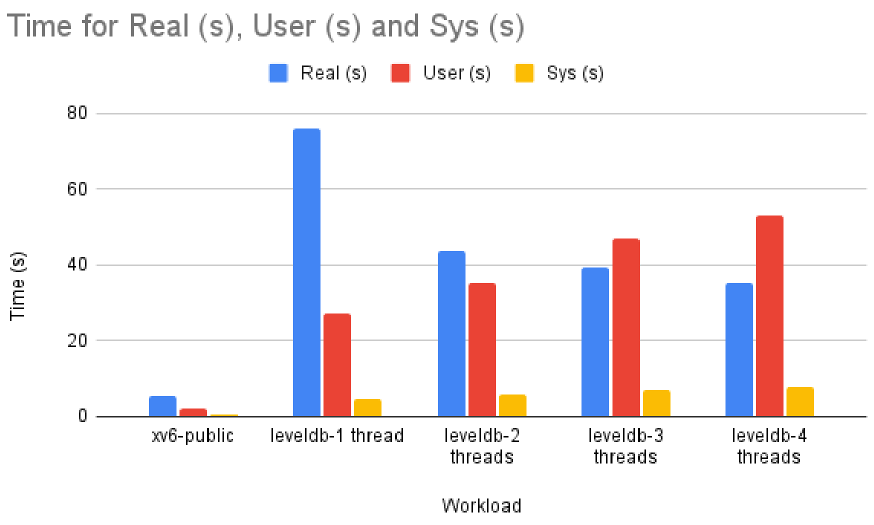

Figure 7: Make time for various build tasks

With no baseline to compare to, the timing result for xv6 is not that meaningful. Regarding leveldb, however, it is interesting to see that real-time improved for the build as the number of cores increased, while user- and sys-time increased.

5 Consistency

5.1 Basic Cache Consistency

We modified the tests and the changes are located in the consistency_tests folder. This includes tests that involve killing a client and a server in the middle of execution; however, for simplicity, rather than programmatically killing these processes, we included a long sleep time for the user to manually call pkill in Linux to kill them.

5.2 Who is the Winner?

We chose to do option A here. Initially, one host was winning all the time. However, we discovered this was because the server we were running was an AWS EC2 instance in the us-west-2 region (PDX availability zone). One of the clients was also an EC2 instance in the same availability zone. Because these hosts were part of the same network fabric, the latency between them with requests was quite small. However, the other client was from a CloudLab Ubuntu instance in Madison, so its latency was far higher.

To make the test more event, we switched to making both of the clients call the server from Madison's CloudLab instance. We noticed that as we increased the write size, the variance increased. More small requests allowed us to get a better determination of which host was the winner. However, as the slowdown increased, over the course of hundreds of runs of the test, we were able to determine a good, deterministic probability for the winner. This is because the release time was delayed, which accounted for the small system variations of the server.

6 Durability

We elected to test durability by following the specifications and including two error injections: errinj_alice_reorder, where the file operations will be reordered and then batched to the local host file system; and errinj_alice_delay, where the file operation will be delayed before the request is sent to the local host file system. This was implemented by creating a small caching layer sitting above the local file system calls. A request for an operation would come into the queue in this caching layer.

Upon receiving an injection for the reorder error, this cache is reordered by reversing it. If a delay was unspecified, the behavior would be as normal since the cache would execute immediately. However, if the delay error injection was used, the head of the cache would have its local operation executed at a time after the call was invoked precisely equal to the number of nanoseconds specified by the delay. The workload we used to exploit this was our test_script.py–namely, the consecutive writes portion of it. Writes would come in sequentially, but as the writes were occurring out of order with the last modified time we use to key off of not matching, the data was corrupted, resulting in an IOError and crash.

7 Discussion

The results of our bench-marking and evaluation demonstrate the potential performance benefits of an `AFS`-like file system implemented in user space, including multi-client scalability, consistent local file caching, support for concurrent file accesses, and reasonable latency of file operations, with noticeable improvements to performance after hot files have been locally cached.

Future implementations could benefit from an enhanced cache consistency callback method. Currently, we compare local file modification date and file size to that of the server file in order to determine if the cached file is stale. Hashing file contents and comparing between server and client would ensure more precise callbacks.

References

[Bro21] Sergey Bronnikov. "UnreliableFS". In: (2021).

[How+88] John H Howard et al. “Scale and performance in a distributed file system”. In: *ACM Transactions on Computer Systems (TOCS)* 6.1 (1988), pp. 51–81.