A Hybrid Concurrency Control Protocol for TSFS

Libo Chen

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-196
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-196.html

December 1, 2020

A Hybrid Concurrency Control Protocol for TSFS

by Libo Chen

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Professor Joseph M. Hellerstein
Research Advisor

5/28/2020

(Date)

* * * * * * *

Professor David E. Culler
Second Reader

5/28/2020

(Date)

A Hybrid Concurrency Control Protocol for TSFS

Abstract

A Hybrid Concurrency Control Protocol for TSFS

by

Libo Chen

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Serverless computing with stateless cloud functions has been gaining popularity. This has raised interest in studying how to provide state management in the serverless setting. Current cloud providers offer a limited number of choices for state management, none of them like a POSIX-compliant file system that most programmers are familiar with. TSFS, a Transactional Shared File System for serverless computing, is an attempt to add a missing piece to state management in serverless computing. To support consistency guarantees implied by POSIX specification and the performance requirements necessary for large number of concurrent cloud function invocations, we implement a hybrid distributed concurrency control protocol that resolves read-write or write-read conflicts optimistically, but write-write conflicts pessimistically while maintaining cache consistency. Both concurrency control and caching mechanisms are unified through the concept of logical leases. We conduct an experiment to see if this hybrid concurrency control protocol is a good fit for TSFS, and whether it might offer a potential alternative for optimistic concurrency control protocol that TSFS presently implements.

Contents

List of Figures

Chapter 1

Introduction

Serverless computing has became the new buzzword in the academia and industry [4, 10, 15], and its popularity has grown rapidly over the the recent years [18]. Serverless computing, often offered as Functions-as-a-Service or "FaaS" [22], provides an attractive paradigm for the deployment of cloud applications. Developers can decompose an application into multiple components where each components is encapsulated as a function in a way that is similar to application decomposition for microservices, however a function in FaaS often is typically more lightweight and does less than a service, even a "micro" one. Developers upload their functions and specify how and when the function should be triggered. These functions are often written in a high-level programming language such as JavaScript (with Node.js), Python, or Java, and it is also possible to provide a custom runtime for other languages. The cloud provider handles everything after that, all of the complex infrastructure management associated with the deployment, including thing such as resource provisioning, security, and any other administrative responsibilities. All of the complexity involved in allowing an application to scale is hidden from developers. The lightweight nature of cloud functions is is enforced by various constraints imposed by the cloud providers. One important constraint is the statelessness of cloud functions. Once a cloud function finishes its execution or is forced to exit due to time limit, any data accumulated during execution in the memory or local storage will be purged. This allows each instance of a function to be set up or torn down very quickly. The compute units and storage units can be scaled up and down separately and independently. This also leads to pay-as-you-go pricing model which often is one of the key arguments for adopting FaaS.

FaaS fits well with stateless applications such as web services that use a RESTful design patterns. To run stateful applications such as ML applications, developers need state management. Figure 1.1 shows a typical solution for state management in serverless comupting. In the case of Amazon AWS, storage services such as S3 and DynamoDB are popular options for state management. S3 [3] is an object storage service with extremely high durability. Its API provides operations such as `PUT` and `GET` at object level. It offers read-after-write consistency for new `PUTs`, and eventual consistency for other `PUTs`, and for `DELETEs`. DynamoDB [2], a NoSQL database service aiming for 99.999% of availability, offers eventual

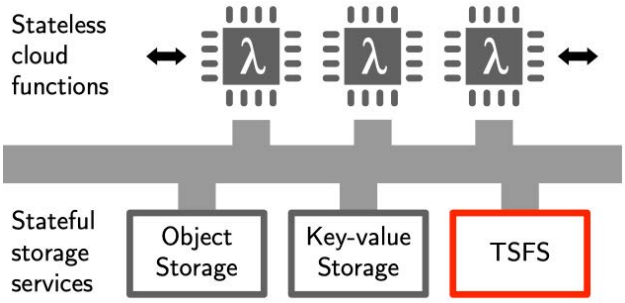

Figure 1.1: Serverless applications use stateless cloud functions and scalable storage services such as object storage, key-value storage etc. TSFS can offers a storage service with familiar a familiar POSIX file system API.

consistency with one exception: reads can be optionally requested with strong consistency, at the cost of a lower guarantee of availability. Each alternative has its own performance and cost characteristics and tradeoffs, but none of them have APIs as popular and ubiquitous as POSIX-compliant file-systems. The mismatch between POSIX APIs and current cloud storage APIs makes it unattractive to port existing code to cloud functions. For example, without the support for the POSIX file system API, it is almost impossible to run build systems such as `make` in serverless environment without heavy engineering efforts. As another example, the popular open source ray-tracing program POV-Ray [12] relies heavily on the underlying file systems for storage of intermediate data passed between processing steps (this includes Partial Output Options, Interrupting Options and Resuming Options that allow flexible handling of image files). POV-Ray includes significant amount of file system-specific code for file handling. To port applications such as POV-Ray to cloud functions, developers need to either modify the source code or add additional shim layers to convert between POSIX APIs and storage APIs provided by the cloud providers. The former is extremely inconvenient if not infeasible for larger and more complex applications and is prone to errors. The latter is sub-optimal as it adds additional performance overhead and introduces extra complexity. These disadvantages diminish the benefits of serverless computing for use with existing applications.

We designed TSFS, a Transactional Shared File System for Serverless Computing, in an attempt to explore the possibility of using a POSIX file system for state management with cloud functions. This paper focuses on the transaction and cache consistency parts of TSFS. In the initial prototype, we choose optimistic concurrency control implemented it with traditional timestamp-order protocols. This works well when read-write conflicts are relatively moderate [1]. In this extension, we propose a hybrid concurrency control algorithm inspired by Sundial [25]. The algorithm resolves write-write conflicts pessimistically through two phase locking, and read-write conflicts with the aid of optimistically extended logical leases [24]. That is, when resolving read-write conflicts it first tries to extend the lease

associated with each data. If it is unable to do so, as in the standard optimistic concurrency control protocol, the transaction will be aborted. The use of logical leases integrates nicely with both the concurrency control mechanism with the caching mechanism. Our goal is to match the performance of the original optimistic concurrency control implementation under low and moderate contention, and beat it under high contention.

Chapter 2

Preliminaries

2.1 Architecture of TSFS

Each cloud provider offers its own unique cloud function service. To simplify our design, we focus on AWS Lambda. Figure 2.1 shows the internal architecture of TSFS. Application code access file system through GNU C Library, `glibc` which provides the interface specified in the POSIX API. Since the Lambda environment has no root privileges, we cannot load a custom kernel module or mount FUSE, the Filesystem in Userspace. Instead we seek to intercept system calls and route them to an user-space file server. Syscalls get intercepted

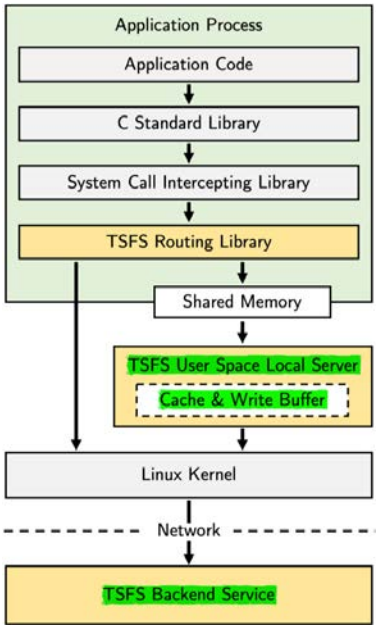

Figure 2.1: Architecture of TSFS. This paper focus on the highlighted components.

through the System Call Intercepting Library [21] by hotpatching machine code of `glibc` making it possible to divert any system call to our application's code paths, rather than sending it directly to the operating system.

With the TSFS Routing Library, POSIX system calls can be routed to TSFS based on the file path, e.g., one can route all call with the prefix `/mnt/tsfs/` to TSFS, and it will always route calls on file descriptors created in TSFS back co it. All others will be sent through to the kernel directly. The TSFS User Space Local Server takes in the routed POSIX syscalls and communicates with the TSFS Backend Service, utilizing the serialization and RPC mechanism from Golang. The Cache and Write Buffer are maintained between the cloud function and backend service to improve read latency. The User Space Local Server, like a typical local file system, implements Linux VFS layer, providing operations such as Open, Stat, Read, Write, etc.. The Local Server also acts as a client in the distributed setting, while the backend service acts as a server in a way similar to NFS [14]. The Backend Service holds the ground truth: all the committed data, including file directories and regular files, are stored here. Multiple local servers from different cloud functions concurrently modifying the file data can introduce consistency problems. Our concurrency control mechanism and cache mechanisms are implemented in the three components highlighted in green in Figure 2.1.

2.2 Consistency Guarantee and Transaction

Transaction

The average duration of an invocation of Lambda function is roughly 1 second, with a long tail [18]. About 88% of them take no more than 10 seconds. The short runtime and discrete begin and end boundaries of cloud functions are a natural fit to transactional APIs and mechanisms. The start and end of an invocation of a cloud function correspond to the start and end of a transaction, so we can transparently wrap a cloud function invocation in a transaction.

Linearizability

From the POSIX specification, we observe that POSIX implies linearizability for `read()` and `write()`, as it states "Writes can be serialized with respect to other reads and writes. If a read() of file data can be proven (by any means) to occur after a write() of the data, it must reflect that write(), even if the calls are made by different processes" [20]. More specifically,"After a write() to a regular file has successfully returned: Any successful read() from each byte position in the file that was modified by that write shall return the data specified by the write() for that position until such byte positions are again modified" The use of words "after ... successful returned" and the fact that each process uses wall clock time dictate a real-time linear order of operations. We can see that the specification does require a real-time guarantee on the behavior of reads and writes on a single object: file.

Strict Serializability

Transactions offers atomicty for groups of one or more operations over one or more objects. Strict serializability, a strong transactional consistency model, guarantees the execution of a set of transactions is equivalent to a serial order corresponding to real time. If a transaction is a single operation on a single object, then we can see that linearizability as special case of strict serializability. To approach the linearizability guarantee required by POSIX, but in a transactional context, we strive to to ensure strict serializability.

Chapter 3

Design

In the initial prototype, we implemented an optimistic concurrency control protocol that defers consistency checks. At the beginning of each transaction, a client obtains a read timestamp, *readTS*, from the backend, which represents the most-recent-committed timestamp of the file system. All reads in the transaction need to be at or after *readTS*. The client maintains read set and write set that are sent to the backend at the commit time, where they get validated and can either become committed or aborted. In this paper, we propose a hybrid concurrency control implementation that uses the concept of a logical lease to reorder transactions and enforce cache consistency.

3.1 Logical Lease

Logic leases [24] defined by a pair of timestamps associated with each data item. We represent each timestamp by a signed 64-bit integer. The data to which the lease applies can be wither a file, which is a variable size vector of bytes, or a fixed-sized block. The first timestamp of the lease, denoted as wts, is always equal to the commit timestamp of the most recent version of the associated data item. It represents the start of the lease. A transaction can read a data item if the commit timestamp of the transaction $txnCommitTs$ is no less than wts, i.e., $wts \leqslant txnCommitTs$. The second timestamp, denoted as rts, is the end of lease. rts is never less than wts, $wts \leqslant rts$. A transaction can read a data item only if $wts \leqslant txnCommitTs \leqslant rts$. We define $T.R$ as the set of data items read but not written by a transaction T. In order to successfully validate all reads in T, its commit timestamp $T.txnCommitTs$ must satisfy,

$$\forall i \in T.R, T.txnCommitTS \in \bigcap_{i \in T.R} [wts_i, rts_i]$$

To avoid abort, the commit timestamp of a successful transaction, $T.txnCommitTS_{success}$, needs to be as large as the largest wts and as small as the smallest rts among those data

items in $T.R$,

$$\max_{i \in T.R} wts_i \leqslant T.txnCommitTS_{success} \leqslant \min_{i \in T.R} rts_i \tag{3.1}$$

Logical leases (wts, rts) provide windows for a transaction to read the data. On the other hand, a transaction can write to a data item only after the lease associated with the data item expires. I.e., rts_i of a data item i written by the transaction needs to be less than the commit timestamp, $txnCommitTs \geqslant rts+1$. We define $T.W$ as the set of data items written by transaction T. $T.txnCommitTS_{success}$ needs to be larger than the largest rts among data items in $T.W$. I.e., in order to avoid abort on writes,

$$T.txnCommitTS_{success} \geqslant \max_{j \in T.W} rts_j + 1 \tag{3.2}$$

By combining equation 3.1 and 3.2, we have

$$\max(\max_{j \in T.W} rts_j + 1, \max_{i \in T.R} wts_i) \leqslant T.txnCommitTS_{success} \leqslant \min_{i \in T.R} rts_i \tag{3.3}$$

Logical leases provide a partial order among concurrent accesses to shared objects. The partial order provides valuable flexibility because it decouples the commit timestamp order from the order in which commits are computed. With logical leases it becomes possible to commit "in the past," by assigning a commit timestamp that is lower than some previously assigned timestamp, provided it is consistent with Equation 3.3.

3.2 Cache

In cloud functions, it is infeasible to deploy invalidation-based approaches where the backend issues invalidation requests to relevant cloud functions. The primary reason for this is that cloud functions can go into a frozen state, where they are unresponsive to timers or events on the network. Instead, we check the freshness of a cached data item by consulting its logical lease whenever a transaction tries to read from it. So long as its commit timestamp falls within the lease of cached data item, $wrt_{cached} \leqslant txnCommitTS \leqslant rts_{cached}$, reading from it can still guarantee strict serializability. If the lease associated with the cached copy is outdated, meaning the cached copy is stale, it's possible that a transaction that reads from the stale cached copy will be aborted because the start of the lease associated with the cached copy of data item i, $wts_{i_{cached}}$, will not equal to the corresponding timestamp, $wts_{i_{backend}}$. This occurs if another transaction modifies data item i. Such aborts can be a problem when the backend experiences many concurrent writes to the same data items. Currently we don't optimize for such case in favor of simplicity. In the future, we hope to explore more sophisticated policies to reduce the abort rate under contention.

3.3 The Protocol

We base our design of the concurrency control protocol on Sundial [25], a distributed hybrid concurrency control protocol for in-memory distributed databases. Sundial uses a hybrid scheme to resolve different types of conflicts while efficiently maintaining cache consistency. For write-write conflicts, it resolves contention them with two-phase locking and avoids deadlock with a Wait-Die scheme [13] based on wall clocks. Concurrent writes to the same data item may result in early transaction aborts, i.e., the transaction manager does not have to wait until commit time to abort transactions. For any read operations, conflicts are handled optimistically as described in Section . Sundial uses logical leases to partially order concurrent operations (i.e., reads and writes) and dynamically calculates the commit timestamp that in an effort to reduce read-write or write-read conflicts. Logical leases resemble the housing rent leases: they have a fixed start time, and an initial end time, however the end time can be extended, possibly multiple times, when both parties agree to do so. Here, lease extension provides a mechanism for negotiating a commit timestamp so that a transaction can avoid aborting. Every item in the database has an associated logical lease. The granularity of the data item also determines the granularity of write lock in 2PL. Our hybrid concurrency control protocol consists of four phases, each of which coincides with a phase of traditional OCC presently implemented in TSFS:

1. Begin phase. We initialize the commit timestamp, $txnCommitTS$, by requesting the largest committed timestamp from the remote backend. This is equivalent to $readTS$ in our OCC implementation.

2. Modify phase.

   - Read operations first check if the requested data item *data* is already in the read set. If it is, it returns the previously read data. If not, it requests the most recent committed version of data from the remote backend, caches the result, and adds it to the read set. It then updates the commit timestamp: if $wts > txnCommitTS$, it sets $txnCommitTS$ to wts. This guarantees that the commit timestamp is no less than $\max_{i \in T.R} wts_i$, as required by Equation 3.1. It then applies any previous writes intersecting with the item *data* and returns the result to the application.
   - Write operations try to acquire a lock on the data item if the data is not already in the write set. This succeeds if the data in the remote server has not been modified since the last time the transaction accessed, if not, it aborts the transaction. In the case of success, it updates the write set and updates $txnCommitTS$ to be larger than the wts retrieved from the backend. This guarantees that the commit timestamp is no less than $\max_{j \in T.W} rts_j$ as required by Equation 3.2.

3. Validate phase. The modify phase already guarantees a lower bound on the commit timestamp of a successful transaction, given by Equation 3.3. To provide an upper bound and to help ensure a successful commit, the validate phase extends rts when

necessary so that it is no less than $txnCommitTS$. There are two cases where such lease extension can fail and lead to abort:

a) For data item i in $T.R$, having lease (wts_i, rts_i) in the client and its lease (wts_i^*, rts_i^*) in the backend, if $wts_i \neq wts_i^*$, then we know data item i was modified by other transactions before transaction T can safely commit. Thus, the read on data item i is not the most recent committed version.

b) Data item i is locked by another transaction.

Before aborting the transaction, data item i is removed from the cache.

4. Commit/Abort Phase. Both phases release the locks held by the transaction and guarantee that their effects are atomic.

Logical leases are also used in the cache consistency protocol to reduce the number of freshness checks needed for read request. During a read operation we can use a locally cached copy, without making a freshness check request to the backend, so long as the commit time of the transaction falls within the lease of the data item in the cache. This can help reduce overheads associated with the cache consistency protocol while maintaining strict serializability.

This protocol provides three major benefits for TSFS:

- It is very similar to our current optimistic concurrency control (OCC) implementation of backend, local server and caching. It also has the same Begin, Modify, Validate and Commit phases that a typical OCC. The only two major distinctions from OCC are 2PL and logical leases, but they are quite manageable in terms of complexity. So it requires less engineering efforts on implementation and testing than a dramatically different protocol such as the complex distributed locking mechanism used in Lustre file system [9].

- The cache consistency mechanism of this hybrid protocol does not requires invalidation from the backend when data has been updated by other transactions. In the case of AWS Lambda where inbound network connections are prohibited, approaches like leases [8] that requires invalidation requests sent from the server to all the clients after a installed file is update will not work.

- This hybrid approach maintains most of the benefits of OCC when writes are infrequent, and it also improves on its shortcomings with high-contention workloads. OCC protocols often perform poorly when resources are saturated [1], and the hybrid approach extends improves their suitability in such situations. The strong consistency guarantee that we provide is not known for high performance and high scalability, so it is important that we strive to lower the overheads of our concurrency control protocol to achieve acceptable performance. This is particularly important there may be many cloud functions running concurrently in a serverless application.

Chapter 4

Implementation

4.1 Data distribution

In a distributed DB setting such as the one in Sundial, data is often partitioned, or sharded, across servers. This load distribution can improve availability, speed up recovery, and increase aggregate write throughput. Each partition of the database is mapped to one or more sites. Often sharding a database introduces additional operational overheads, which might have too great a performance cost for a a file system. As mentioned earlier, there is no "peer-to-peer" network among cloud functions, and local storage in a cloud function is not persistent beyond the lifetime of the function. Thus partitioning data or even synchronizing cached data among cloud functions is not an option. In TSFS, a local client (cloud function) never stores any persistent state locally; instead it maintains a limited size cache of files and directories in memory. In order to read any uncached data, local clients need to request the data from the remote server. In contrast, a site in a sharded DDBMS requests a tuple from a remote site only if the tuple is not found among that mapped into its local database. As a result, TSFS can experience more round-trip delays and more network traffic than a traditional DDBMS because almost all data needs to be requested from the remote server. It is important that we reduce the number of round trips in order to alleviate this performance impact. One simple solution is to increase the cache size. Another optimization is to renew leases all at once, batching the renewal instead of renewing them one-by-one.

4.2 Granularity of Logical Leases

In a database system, an obvious choice for the granularity of logical lease is the tuple, the unit of read or write operation. When it comes to file systems, the choice becomes less obvious. One choice is a file, the other is a block. Files can grow or shrink dynamically, whereas block size is fixed and is only set before the file system is initialized. At the granularity of blocks, locking imposes a larger overhead, but could potentially provide higher concurrency. Since we use 2PL with wait-die to resolve write-write conflicts, it is relatively

more convenient to lock a file than to lock blocks, even though certain portion of the file data may not be modified. For a write, it either grabs one lock or none. Guaranteeing atomicity can be more difficult at the block level because this may require locking many blocks, but if acquiring just one of them fails then all of the other must be unlocked. For example, with 1 KB blocksize and a `pwrite` to byte range [0:4096], in order to lock the whole range we need to acquire a lock on four different blocks. If we fail to acquire the lock on the fourth block, to ensure atomicity, the locks on previous three blocks need to be released back. The need to acquire multiple locks can significantly increase the latency of a write operation. Maintaining leases and locks at the block level introduces extra memory overhead as well as extra complexity to the whole file system. Therefore we choose to associate logical leases to files.

4.3 Read set and write set

Each transaction will maintain a write set, WS, and a read set, RS, both indexed by inode number. If a file (even only a small portion of it) has been read previously, then the read set will contain an entry for that file. This entry reflects the file's wts, rts and the content of the file. The read set can keep track of all of the bytes of the file, or only that portion requested by the transaction. Both was have their own advantages. We chose to keep just the requested byte range because keeping a potentially ever-growing file can incur significant memory overheads.

If a file has been written before, then the write set will contain an entry for that file; the entry indicates that the file has been locked. We can also apply uncommitted changes made by the current transaction to the version of the file kept in the write set. This essentially creates a snapshot of accessed file such that future reads or writes only need to consult this local snapshot without having to communicate with remote backend. This approach should benefit longer transactions, especially those with many read-after-write or write-after-write operations. However, it suffers high memory overhead when a transaction is short or when local snapshots are rarely utilized. We pursued file snapshots but eventually abandoned them as early experiments showed a significant performance degradation for most of the workloads we ran.

4.4 Write

For `pwrite(inode, offs, buf)` in algorithm 1, if *inode* is not already in the write set, i.e. this file has not be written in the same transaction, it first tries to acquire a lock on the file. On success, it updates the commit timestamp to $rts+1$, if this does not put it after the lease expires. On failure, or if the returned wts doesn't match the wts in the read set (meaning the file stored at the backend has been updated by another transaction), we immediately abort the whole transaction; it is doomed to fail the read validation phase.

Algorithm 1: Transaction T writes *Buf* at *offs* to a file with *inode*

```
Result: The number of bytes actually written
if inode ∉ T.WS then
    success, wts, rts = LockRemoteFile(inode) ;
    if not success or inode ∈ T.RS and T.RS[inode].wts ≠ wts then
        T.Abort();
    else
        T.txnCommitTS = MAX(rts+1, T.txnCommitTS);
Save Buf and offs to T.WS[inode] ;
return len(buf);
```

4.5 Read

Algorithm 2: Transaction T reads from file with *inode* at *offs* to *buf*

```
Result: The number of bytes actually read
if inode ∈ T.RS and byte range [offs, offs+len(buf)] found in T.RS[inode].Data
 then
    n = copy(buf, T.RS[inode].Data[offs, offs+len(buf)]);
else
    n, wts, rts = T.preadCommitted(inode, offs, buf);
    T.txnCommitTS = MAX(T.RS[inode].wts, T.txnCommitTS);
Apply all changes in T.WS[inode] to buf;
Save Buf, offs, wts and rts to T.RS[inode] ;
return n;
```

`pread(inode, offs, buf)` in algorithm 2 takes in an *inode*, an offset and a read buffer. If the inode number is found in the transaction's read set, we know it is already been read earlier. We can directly fetch the data from the read set if the byte range falls within the range in the read set, otherwise we retrieve the requested range of the file from the backend or cache through `preadCommitted`. `preadCommitted` first looks for the file inode in the cache and retrieves fileinfo containing the logical lease associated with the file, either from cache or the backend. File data is cached in blocks, and if the requested blocks are not in the cache, it we go ahead and fetch them from the remote backend. Newly fetched blocks replace the least recently used ones in the cache once the cache becomes full. Once we have the underlying file data requested, as well as its logical lease, we then apply previous writes from the transaction on top of it. The resulting content in the read buffer reflects the most recent version of the file from this transaction's perspective. We also update the commit timestamp if *wts*, the returned end of lease of the file, is more recent than the current commit timestamp.

4.6 Validate Reads

Algorithm 3: Validate read set and renew lease

```
   Result: Success or Failure
1  for inode ∈ T.RS.keys() \ T.WS.keys() do
2  |  if T.txnCommitTS ¿ T.RS[inode].rts then
3  |  |  success = RenewLease(inode, T.RS[inode].wts, T.txnCommitTS);
4  |  |  if not success then
5  |  |  |  remove cache entries for inode;
6  |  |  |  return Failure;
7  return Success;
```

The Validate phase in algorithm 3 checks each read-only file to see if its *rts* is less than the commit timestamp. In that case, we need to send a request to renew the logical lease on the file. The Lease is renewed `RenewLease(inode, wts, commitTS)` in following steps:

- As mentioned in the earlier section, if *wts* doesn't match *wts* in the backend or *commitTS* is larger than *rts* in the backend and the file is locked, then lease renewal fails.
- Otherwise, *rts* in the backend is set to *commitTS* if *commitTs* is larger.

If lease renewal fails for one file, then the read validation algorithm first clears that file from the cache, then immediately returns a code indicating that the read validation has failed.

4.7 Commit and Abort

Commit and abort are fairly straightforward. They are no different than in the usual optimistic concurrency control protocol, with one exception: both need to unlock previously locked files present in the write set. Commit also needs to update cached data and logical leases so that $[wts, rts] \leftarrow [txnCommitTS, txnCommitTS]$.

Chapter 5

Evaluation

We conducted experiments on synthetic workloads to evaluate the performance of this hybrid protocol compared to our current OCC protocol. Our main goals were to understand, under various workloads, how and why this hybrid protocol may be better or worse than straightforward OCC.

5.1 Setup

Currently all experiments are run on a single Amazon EC2 instance (m5.xlarge) that has 4 virtual CPUs and 16GB memory. Because the user space local server and backend server reside on the same machine, there is no network latency. When we move to AWS Lambda, we expect that network latency will dominate the performance of some operations. Each experiment is run 15 times and compares the hybrid protocol described in the paper with an existing OCC implementation. We report the results on the last 10 runs so that cache is warmed in a consistent way across experiments.

For a given benchmark, it can be run in 5 different VFS implementations:

- *mem*: an simple non-transactional in-memory file system. *This will be the most the performant variant in all scenarios.*
- *txn-ac*: a transactional filesystem with auto commit. Each operation will be committed before the next operation starts. *This models linearizability required by POSIX*
- *txn-nc*: an transactional filesystem with no auto commit.
- *txn-nc-cached*: txn-nc with cache enabled.
- *txn-ac-cached*: txn-ac with cache enabled.

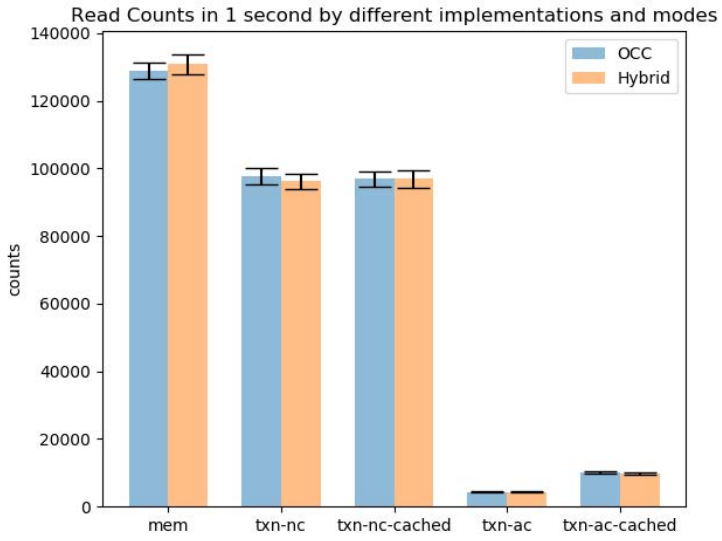

Figure 5.1: Number of reads in 1 second.

5.2 Metrics

We mainly look at three metrics:

- *Latency* in ms/op. The latency per each operation, averaged for each experiment.
- *IOPS* in ops/s. The number of IO operations per second in each experiment.
- *Throughput* in MB/s. The number of bytes accessed per second in each experiment.

5.3 Results

Reads and Writes

We first run a simple benchmark that either keeps randomly reading 1KB data from a 1 MB file or writing 1 KB to a 1 MB file, without growing the file, for 1 second. It counts the number of read or write operations during this period. Since we fix the block size to 1 KB, this benchmarks the IOPs of read or write in different implementation ("OCC" vs "Hybrid") with different modes. *mem* is used as a reference since it's the same for both "OCC" and "Hybrid" and is considered as the fastest implementation possible. From Figure 5.1, we observe that the read performance is pretty much identical in this benchmark, "Hybrid" performs on average 3% worse than "OCC". This does match well with our expectations, as the overhead of logical leases due to extra metadata and lease renewal (extra RPC calls) are fairly negligible. On the other hand, in Figure 5.2, largely due to two phase lock in write

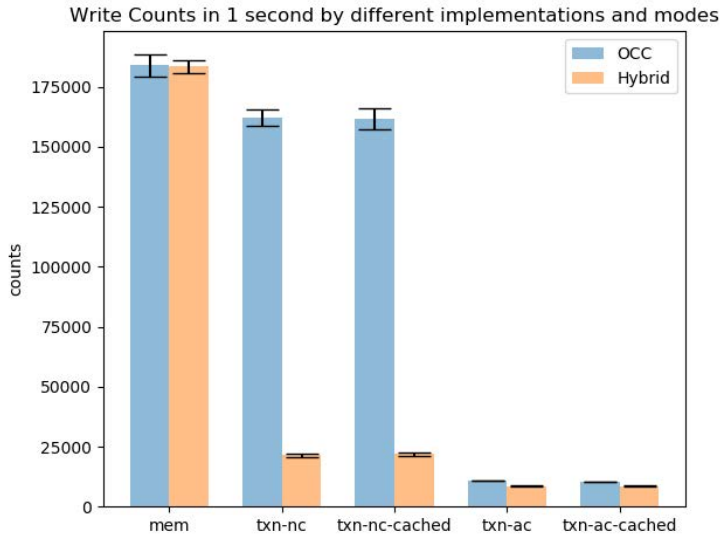

Figure 5.2: Number of writes in 1 second.

operation, "Hybrid"'s write performance drops significantly for two modes. *txn-nc* and *txn-nc-cached* do not commit each write automatically, and without the extra complexity we introduced to the validate and commit phases for our "Hyrbid" implementation, the only difference between "Hybrid" write and "OCC" write is the locking mechanism. The locking mechanism essentially brings down the write performance in *txn-nc* and *txn-nc-cached* to the level of *txn-ac*, a staggering 86% decrease in number of writes per second. Improving our locking implementation could help alleviate this gap. Another interesting observation is that for *txn-ac* and *txn-ac-cached*, the costs of the validate phase seems to dominate the total cost. This is because the only differences between *ac* and *nc* are the extra begin, validate and commit/abort phases in auto-commit mode. We see more than 90% decrease in number of writes per second from *txn-nc* to *txn-ac* for "OCC". We also see only an 18% to 20% drop in write counts from "OCC" to "Hybrid" in auto-commit mode, which is considerably less than what we see in no-autocommit modes. "Hybrid" adds extra overheads to the validate and abort phases because read validation needs to renew leases and transaction abort needs to release file locks through RPC calls to the backend. Meanwhile, "Hybrid" adds almost nothing to commit and begin phases. Caching doesn't play much of a role here, since the benchmark is conducted in the local environment, and random I/O makes cache hits less likely.

Filebench

Filebench[5, 19], a configurable file system and storage benchmark, provides various workloads simulating complex real-world applications. We pick 5 workloads: *fileserver*, *varmail*,

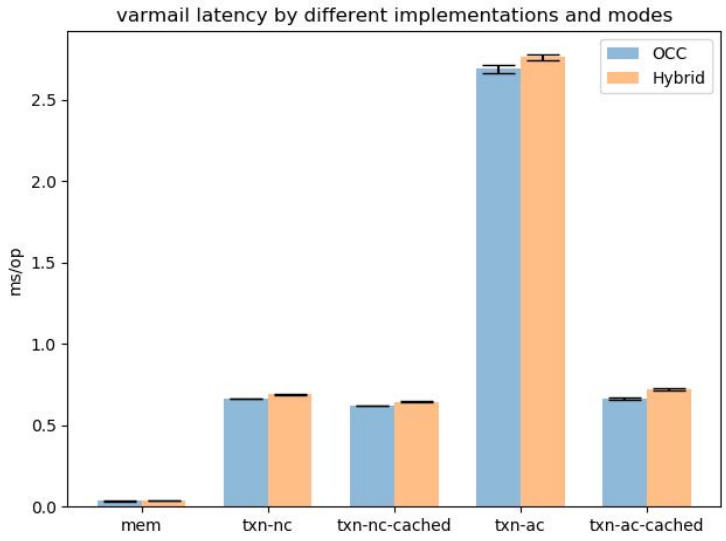

Figure 5.3: Latency Comparison in Varmail Workload.

networkfs, *videoserver*, *webproxy* and *webserver* to see how "Hybrid" perform under various conditions in compare with "OCC" implementation. We present two of them here that are most representative.

In workload *varmail* which emulates I/O activity of a simple mail server, it consists of a multi-threaded set of create-append-sync, read-append-sync, read and delete operations in a single large directory with 1:1 read/write ratio. The number of threads is set to be 10, the average file size is set to be 16KB. We set the average entries of a directory to 1000000 and the number of files to be 2000, so with very high probability all the 16KB small files will be directly under the same directory. IO block size and mean append size is set to be 1MB and 1KB respectively. The total runtime is set to 10 seconds and each read operation reads the whole file. This workload tends to have many small reads and writes with high possibility of conflicts.

Somewhat interesting is even though we have seen that "Hybrid" seems to perform badly on write operations in no auto-commit modes , we don't see such impact on this end-to-end benchmark. From Figure 5.5 and Figure 5.4, we notice that "OCC" and "Hybrid" have almost identical throughput. "Hybrid" doesn't do better or worse than "OCC" in both metrics: IOPS/s and MB/s. From Figure 5.8, "OCC" havs a small advantage on latency per each operation, although the difference is below 0.1 ms per second. We conclude that "OCC" and "Hybrid" are essentially identical in this benchmark. We do notice that caching helps significantly improve all three metrics for auto-commit mode, whether using the "OCC" or "Hybrid" protocol, and brings its performance to the same level as no-auto-commit mode with cache enabled.

textitfileserver is another workload that emulates a simple file-server. It performs a

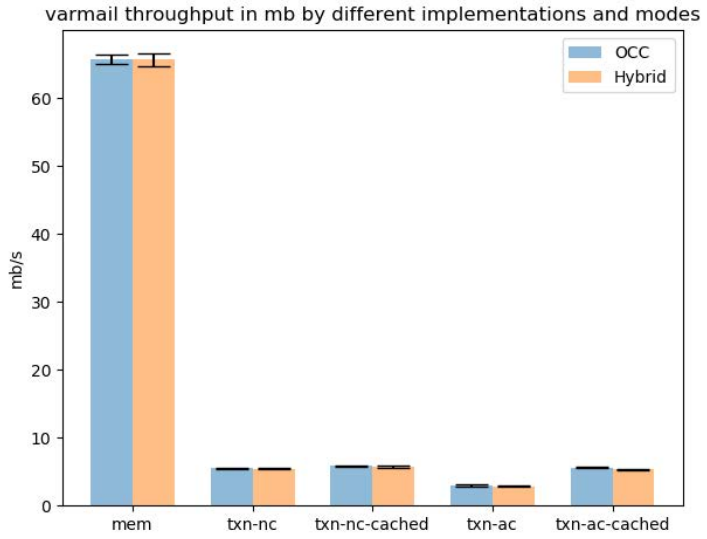

Figure 5.4: Throughput Comparison in Varmail Workload.

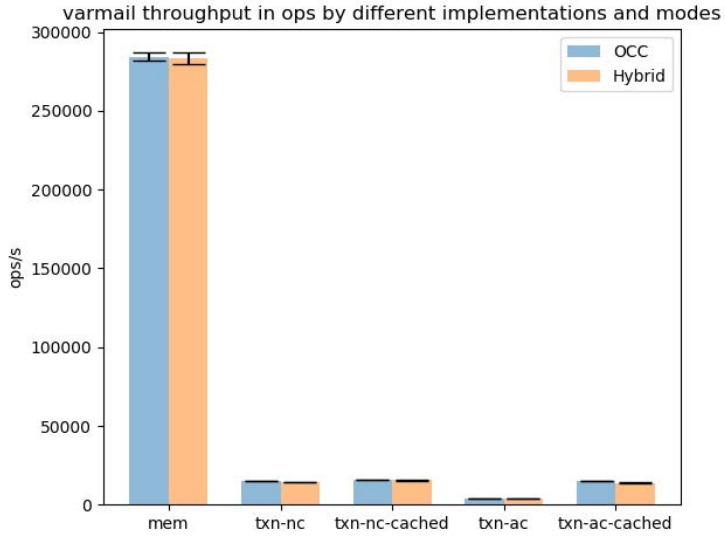

Figure 5.5: IOPS Comparison in Varmail Workload.

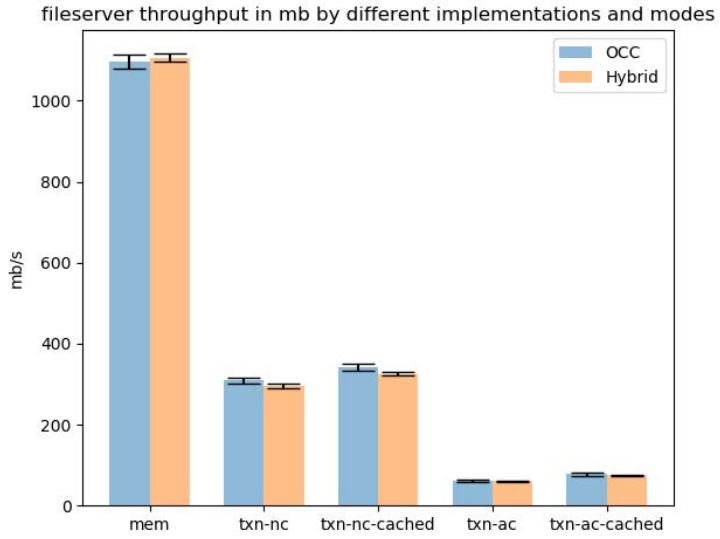

Figure 5.6: Throughput Comparison in Fileserver Workload.

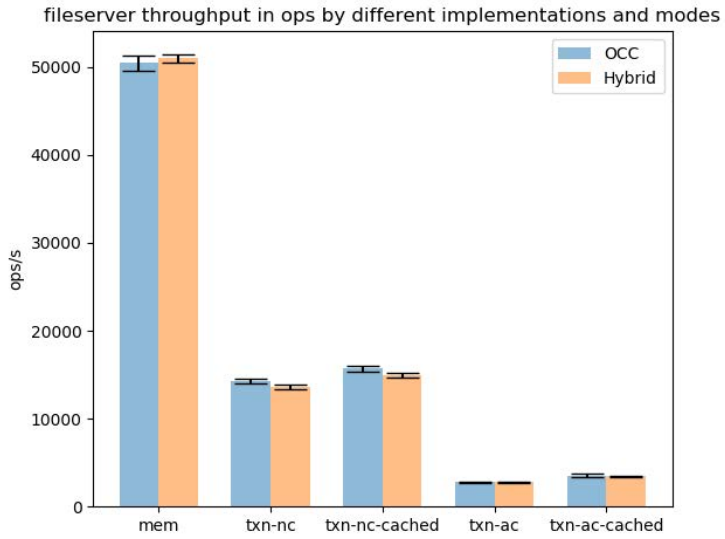

Figure 5.7: IOPS Comparison in Fileserver Workload.

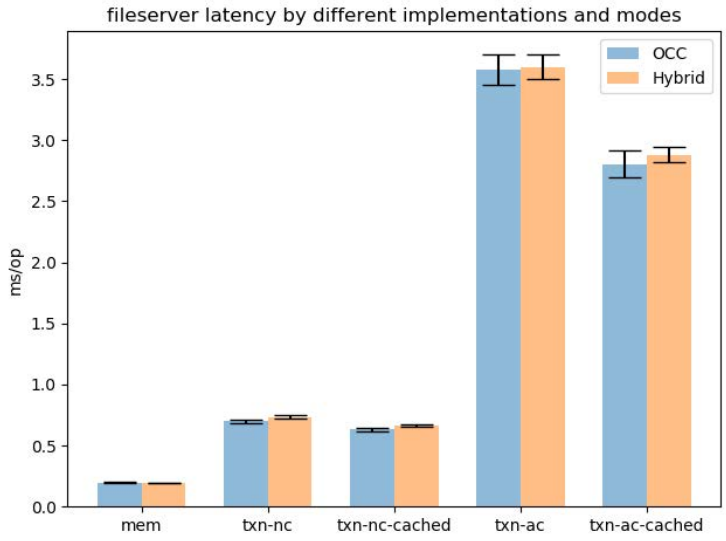

Figure 5.8: Latency Comparison in Fileserver Workload.

sequence of creates, deletes, appends, reads, writes, and attribute operation on a directory tree with 1:2 read/write ratio. We set the number of threads to 10, the IO block size to 1 MB and the mean append size to 1KB. We also set the mean file size to 128 KB and number of total files to 1000, so files tend to be spread among different directories. The total runtime is set to 10 seconds and each read operation reads the whole file as in *varmail*. *fileserver* tends to have fewer conflicts than *varmail*, as files are scattered around multiple directories. As in *varmail*, "Hybrid" experiences a little higher latency per operation, as seen in Figure 5.8, while still maintaining almost identical throughput and IOPs performance to "OCC" as seen in Figure 5.6, 5.7. Although in *fileserver* caching is unable to provide the same performance boost as in *varmail*. Cache hits become less likely due to a smaller chance of reuse of the same file, and due to larger reads. To our surprise, both benchmarks show that the "Hybrid" implementation only incurs a very small latency overhead compared to the "OCC" implementation. By looking into the detailed breakdown of each operation, we notice that since each read operations read the whole file, which is on average 16 to 128 times larger than each write, the majority of throughput, IOPs and latency comes from reads. That's why we can see large impacts from slow writes in the "Hybrid" implementation.

We also notice that the extra latency caused by locking on average is well below 10 ms. In the FaaS environment, the overhead associated with cloud function invocation is on the order of tens of milliseconds. This suggests that the extra overhead caused by locking may be acceptable for cloud functions. We plan on conducting same benchmarks in AWS Lambda to understand how caching policy makes a difference in that setting, and to confirm that locking overhead becomes insignificant.

Chapter 6

Related Work

To our best knowledge, there has not been other work on providing a shared file system as an option for state management for cloud functions. Although there ha been much research on NoSQL databases [17, 23] and object storage [3] for cloud functions. Our design is heavily based on Sundial [25], which seamlessly integrates the concurrency control mechanism and the caching mechanism in an in-memory database system. A good survey on transactional client cache consistency can be found here [6]. There is also plenty of research on distributed file systems such as NFS [14], the Coda file system [11] and the Lustre file system[9]. Most of them are shared non-transactional file systems. Many modern distributed file systems like GFS [7], and HDFS [16] do not prioritize compliance with POSIX, and so are not comparable for our work.

Chapter 7

Future Work

7.1 Future Experiments

We plan to run the Filebench benchmark suite, and to use AWS Lambda in the future. One of the major distinctions between running on Lambda and running on EC2 instances is the network latency. The strong consistency guarantee, strict serializability, that we offer has a reputation for poor performance. The benchmark results from the evaluation of Section 5 has shown a $\leqslant 5ms$ increase in latency per operation. But the network latency in the Lambda environment can dominate the latency per operations and make this overhead less of an issue. Therefore, we hope to draw a comparison with various distributed storage systems such as EFS, or even key-value stores, and to determine whether TSFS with the hybrid concurrency control protocol can perform similarly with those state management solutions presently popular for use with FaaS.

In the future, we also want to benchmark and compare the abort rates between "Hybrid" and "OCC" implementations. The Filebench test suite does not offer a good way to log the abort/success rates of a transactional file system. We currently are developing a tool to run customized tests on Lambda at scale. One of the key potential benefits from this hybrid protocol is lower abort rates under high contention, as compared to the original OCC protocol. This hybrid protocol uses logical leases to reduces unnecessary aborts, and resolves write-write conflicts with 2PL so that the file system can release resources held by the transaction doomed to be aborted at the earliest possible time. We expect to see a flatter slope of abort rates for the hybrid protocol when the number of Lambda function invocation increases.

7.2 Future Optimizations

We also want to experiment with multiple cache policies in the future. As mentioned before, current cache policy will result in more aborts when there are frequent concurrent writes to the same files because a cache block is always used by a read request, even if it may be stale.

This is fine and desirable when reads are frequent and writes are rare, but is undesirable in a write-intensive workload. One way to accommodate both types of workloads is switch policies based upon a fixed use / request ratio, with a threshold determined empirically to work well for most workloads. The other way is to dynamically adjust between use and requesting depending on whether recent requested cached data leads to successful commits or not.

Last but no least, we want to improve our two phase locking implementation to make it more performant and more scalable. We want to experiment with finer granularity of locking in the future to see if it can offer better scalability. Reducing round-trip time between the backend and the client is also of great importance. The current lease renewal algorithm requires one round trip for each file in the read set. We can improve it with bulk renewals, by send one request that renews all the leases in the write at once. We expect this can improve performance of "Hybrid" in no-auto-commit mode as there is only one commit at the end of run, and the transaction can potentially have multiple leases that need to be renewed.

Chapter 8

Conclusion

We design and implement a new concurrency control protocol based on Sundial for TSFS. It contrasts OCC with its use of 2PL for write-write conflicts, and in its use of logical leases associated with each data item. These leases may be extended during the validation phase to avoid unnecessary aborts. We implement this concurrency control protocol in the context of FaaS, where its cache mechanism and its ability to provide strong consistency guarantee are well-suited to meet the needs of TSFS. Our purely local evaluation shows that even though it does increase latency for write operations, as we expected, it doesn't negatively impact overall throughput for real end-to-end applications; in our experiments these applications issue many more reads than writes. We seen a good potential from this new protocol and we think it could be an good alternative choice for TSFS.

Bibliography

[1] Rakesh Agrawal, Michael J Carey, and Miron Livny. "Concurrency control performance modeling: Alternatives and implications". In: *ACM Transactions on Database Systems (TODS)* 12.4 (1987), pp. 609–654.

[2] *Amazon DynamoDB*. `https://aws.amazon.com/dynamodb/`. Accessed: 2020-05-06.

[3] *Amazon S3*. `https://aws.amazon.com/s3/`. Accessed: 2020-05-06.

[4] Paul Castro et al. "The rise of serverless computing". In: *Communications of the ACM* 62.12 (2019), pp. 44–54.

[5] *Filebench*. `https://github.com/filebench/filebench/wiki`. Accessed: 2020-05-06.

[6] Michael J Franklin, Michael J Carey, and Miron Livny. "Transactional client-server cache consistency: Alternatives and performance". In: *ACM Transactions on Database Systems (TODS)* 22.3 (1997), pp. 315–363.

[7] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google file system". In: *Proceedings of the nineteenth ACM symposium on Operating systems principles*. 2003, pp. 29–43.

[8] Cary Gray and David Cheriton. "Leases: An efficient fault-tolerant mechanism for distributed file cache consistency". In: *ACM SIGOPS Operating Systems Review* 23.5 (1989), pp. 202–210.

[9] *Introduction to Lustre* Architecture*. `http://wiki.lustre.org/images/6/64/LustreArchitecture-v4.pdf`. Accessed: 2020-05-06.

[10] Eric Jonas et al. "Cloud programming simplified: A berkeley view on serverless computing". In: *arXiv preprint arXiv:1902.03383* (2019).

[11] James J Kistler and Mahadev Satyanarayanan. "Disconnected operation in the Coda file system". In: *ACM Transactions on Computer Systems (TOCS)* 10.1 (1992), pp. 3–25.

[12] *POV-Ray - The Persistence of Vision Raytracer*. `https://github.com/POV-Ray/povray`. Accessed: 2020-05-06.

[13] Daniel J Rosenkrantz, Richard E Stearns, and Philip M Lewis. "System level concurrency control for distributed database systems". In: *ACM Transactions on Database Systems (TODS)* 3.2 (1978), pp. 178–198.

[14] Russel Sandberg et al. "Design and implementation of the Sun network filesystem". In: *Proceedings of the Summer USENIX conference*. 1985, pp. 119–130.

[15] *Serverlessconf*. `https://serverlessconf.io`. Accessed: 2020-05-06.

[16] Konstantin Shvachko et al. "The hadoop distributed file system". In: *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee. 2010, pp. 1–10.

[17] Swaminathan Sivasubramanian. "Amazon dynamoDB: a seamlessly scalable non-relational database service". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 2012, pp. 729–730.

[18] *State of Serverless*. `https://www.datadoghq.com/state-of-serverless/`. Accessed: 2020-05-06.

[19] Vasily Tarasov, Erez Zadok, and Spencer Shepler. "Filebench: A flexible framework for file system benchmarking". In: *USENIX; login* 41.1 (2016), pp. 6–12.

[20] *The Open Group Base Specifications Issue 7, 2018 edition, IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)*. `https://pubs.opengroup.org/onlinepubs/9699919799/functions/write.html`. Accessed: 2020-05-06.

[21] *The system call intercepting library*. `https://github.com/pmem/syscall_intercept`. Accessed: 2020-05-06.

[22] *What Is Function as a Service (FaaS)?* `https://www.cloudflare.com/learning/serverless/glossary/function-as-a-service-faas/`. Accessed: 2020-05-06.

[23] Chenggang Wu, Vikram Sreekanti, and Joseph M Hellerstein. "Autoscaling tiered cloud storage in Anna". In: *Proceedings of the VLDB Endowment* 12.6 (2019), pp. 624–638.

[24] Xiangyao Yu. "Logical leases: scalable hardware and software systems through time traveling". PhD thesis. Massachusetts Institute of Technology, 2018.

[25] Xiangyao Yu et al. "Sundial: Harmonizing concurrency control and caching in a distributed OLTP database management system". In: *Proceedings of the VLDB Endowment* 11.10 (2018), pp. 1289–1302.