

CS 7680 Serverless Computing
(Special Topics in Systems)

Final Project Serverless Filesystem

April 28, 2022

Ryan Yang

Motivation

- Serverless computing provides “infinite” computing power, and is able to scale down to zero.
- Can we do similar job with **storage services**?
 - Great for serverless application to develop with states
 - Create **no bottleneck** of throughput
- Building a filesystem on serverless
 - Also a good fit for general applications
 - Create an uniform interface for functions to communicate
 - socket-like file for direct communication
 - Custom function can add into the system to create custom chain of transformation
 - Write file with encryption, compression, stream video with specified quality etc.

Goal

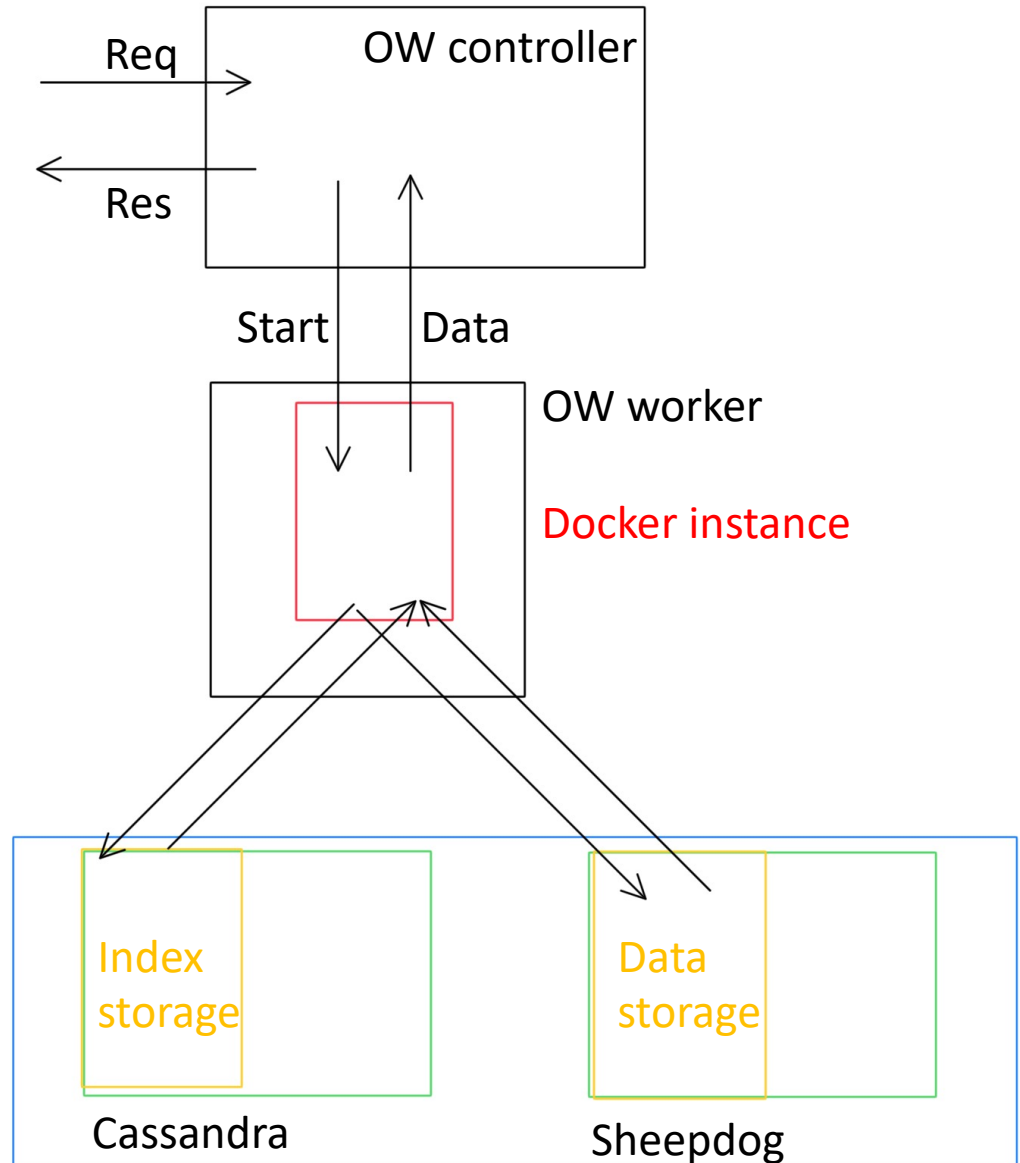
- Build a working serverless filesystem
 - This will serve as a base of the future development.
 - Identify which part is the bottleneck and try to improve
- Future goals:
 - Able to serve serverless functions without creating the bottleneck
 - Able to scale-down to zero
 - Data consistency must handle well

Assumptions

- Data-center network
- Serverless framework:
 - Run at least once for each request
 - Support Docker with
 - /init to inject binary
 - /run to run one function
 - (Openwhisk model)
 - Stateless
- Multiple storage services
 - Each have different consistency model/performance
 - Present storage with continuous index space with optimal atomic read/write size

Implementation

- Overview:
 - Main component
 - Index
 - Data
 - Storage Layer
 - Serverless Framework



Implementation

- Implement in serverless function to insure the scalability
- Single function image implement in c++ (remove cold-start for subsequent invocation)
 - Interface:
 - touch /dir/ectory/multi/file.txt
 - mkdir /dir/ectory/multi
 - write /dir/ectory/multi/file.txt, data="123"
 - read /dir/ectory/multi/file.txt
 - Shares function between all users in this data center (user create their own root).

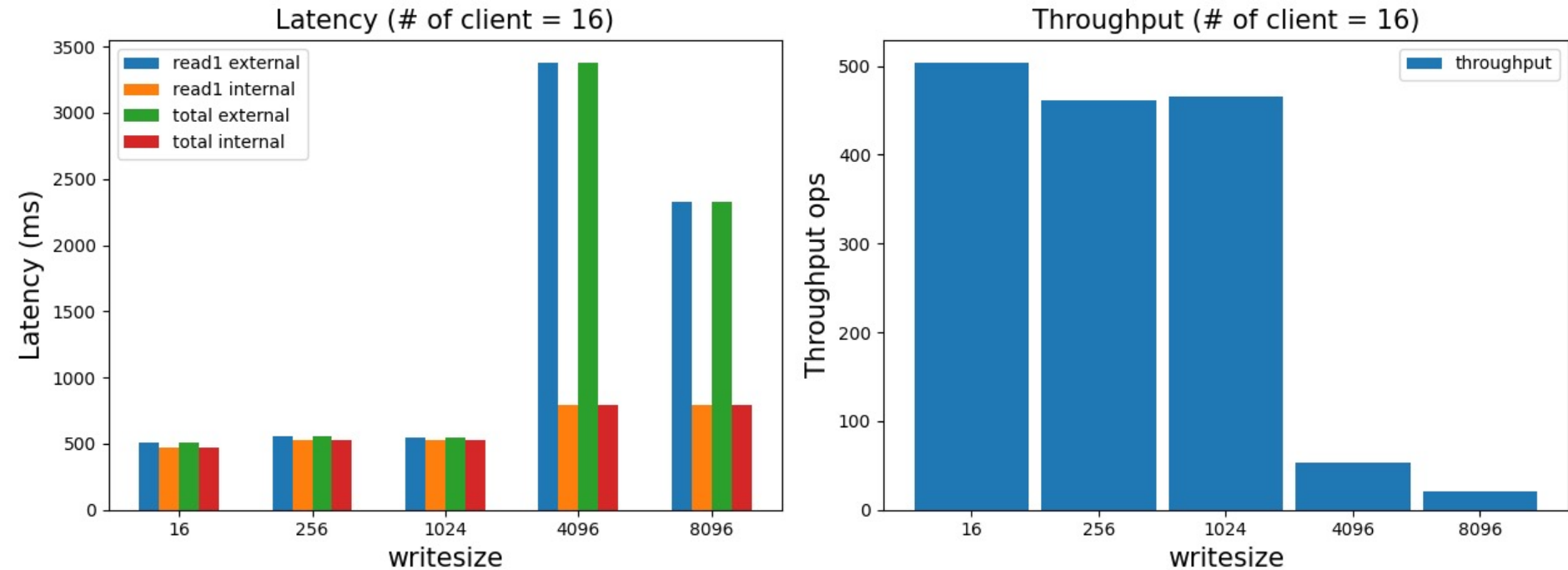
Implementation

- Design detail
 - Index storage
 - This layer manages the namespace
 - inode <attr, blocks>,
 - 16 blocks <vol_id, offset, size> or dirblock <filename, inode_ptr>
 - function <vol_id, access function>
 - Total struct size = 266 bytes (compare to Linux inode 4096 bytes)
 - Store filename in this layer so the RTT to data storage can reduced.
 - Data storage
 - Simple management and simple interface
 - Can be anything as long as it can provide as a pair<offset, size>
 - Local File | KV store | iSCSI

Evaluation

- Evaluation on 5 different tests
 - Single write
 - Single read
 - Write and Read
 - Mkdir /one
 - Mkdir 6 directory
- Evaluation metrics:
 - latency of each request
 - External: the latency from the client side
 - Internal: the latency reported from OpenWhisk (start /run to finish)
 - Throughput (success operations per second)

Evaluation (Read: d/d writesize)

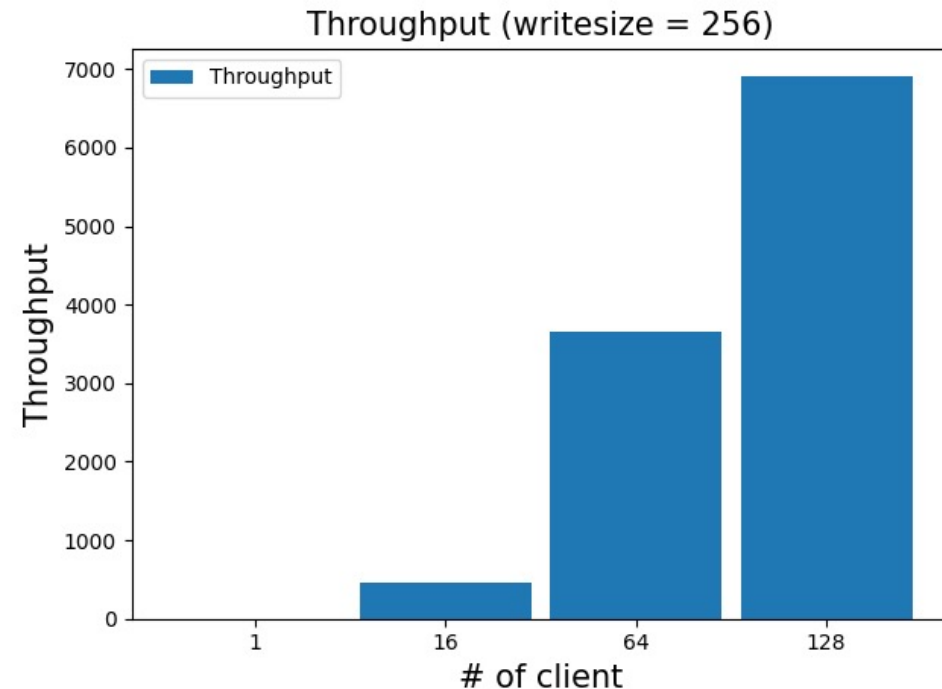
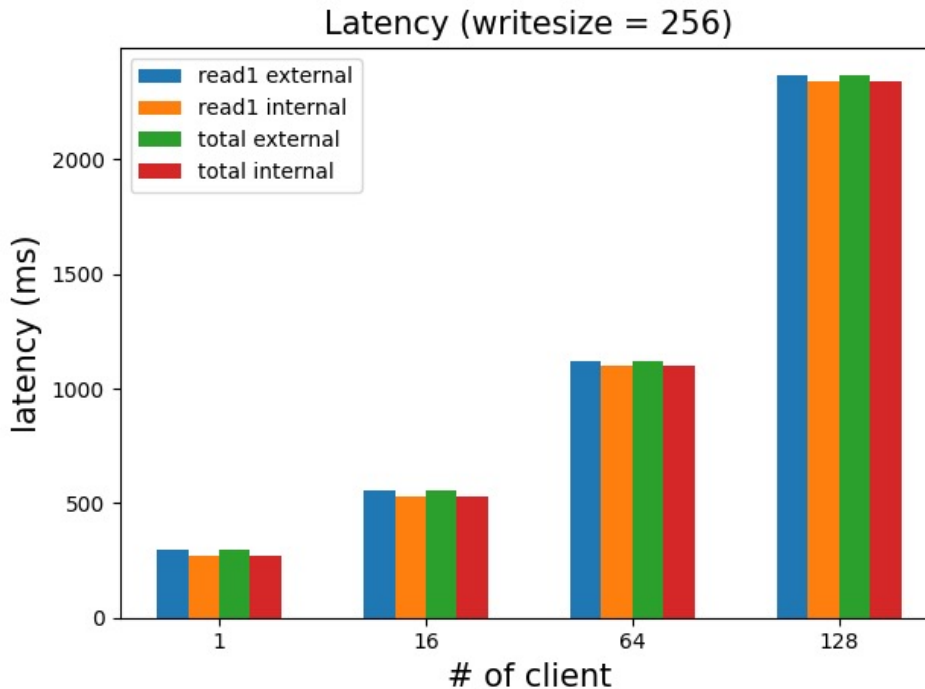


(1) OW Overhead increase greatly at writesize = 4096 (may due to extra packets)

(2) Read operation = Establish connection +
read free list + read root inode +
Establish connection + read data offset * (size / 1024)

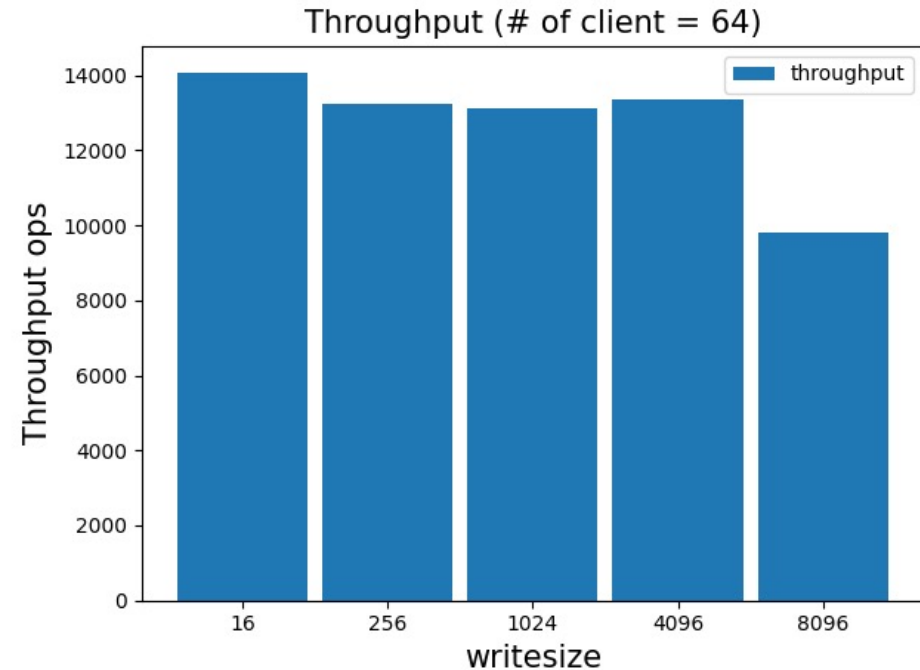
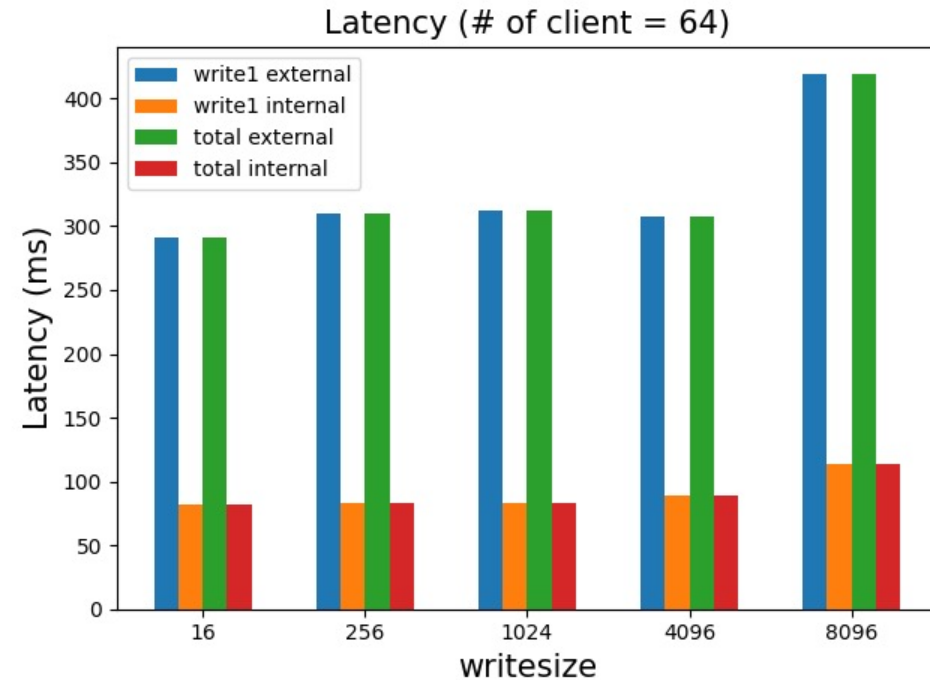
500 ms \approx 30 ms + 130 ms + 130 ms + 30 ms + 130 ms

Evaluation (Read: d/d client)



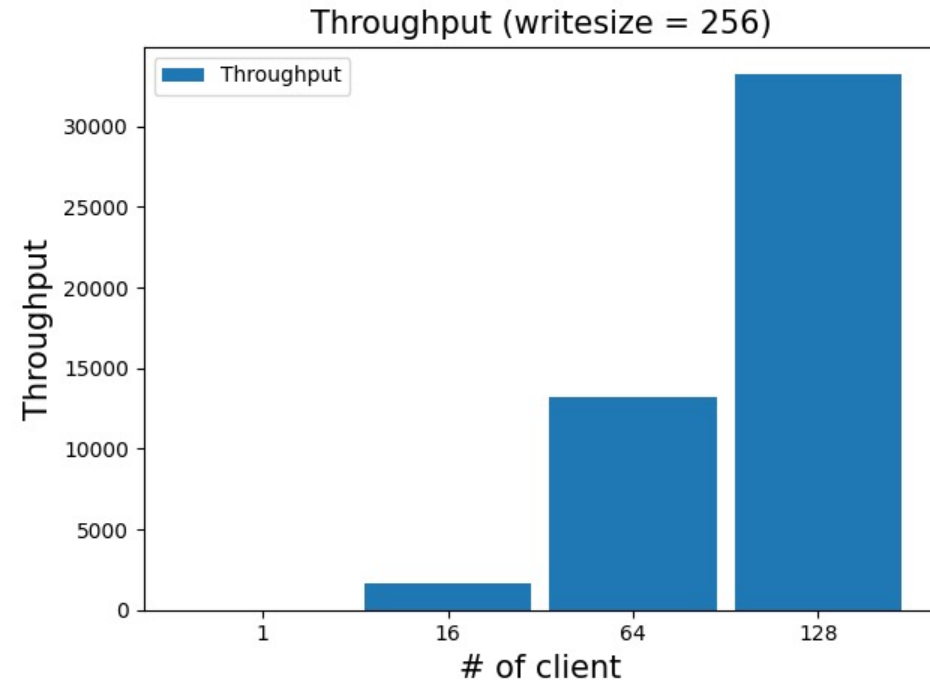
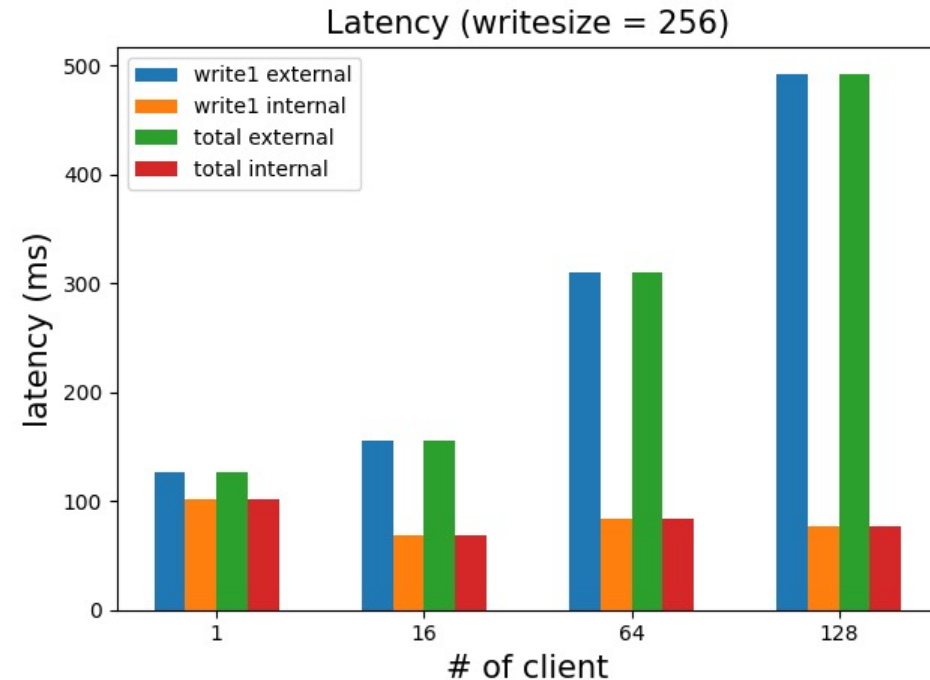
- (1) Increase the # of client would increase the latency and the throughput
- (2) The throughput figure shows that this filesystem can scale

Evaluation (Write: d/d writesize)



- (1) OW have large overhead (~200ms, may due to large network traffic)
- (2) Write operation = Establish connection +
read free list + read root inode + write root inode + write inode +
Establish connection + write data offset * (size / 1024)

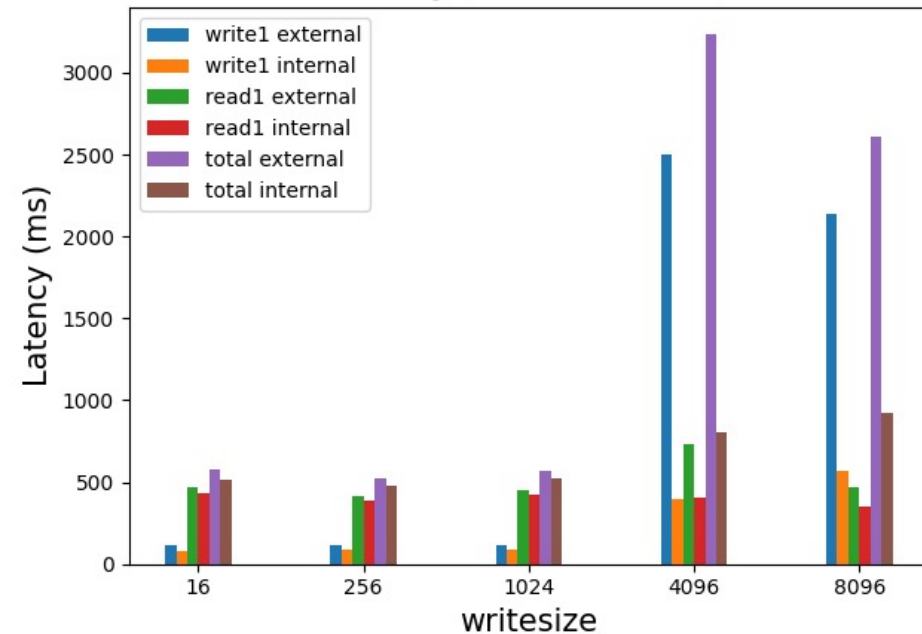
Evaluation (Write: d/d client)



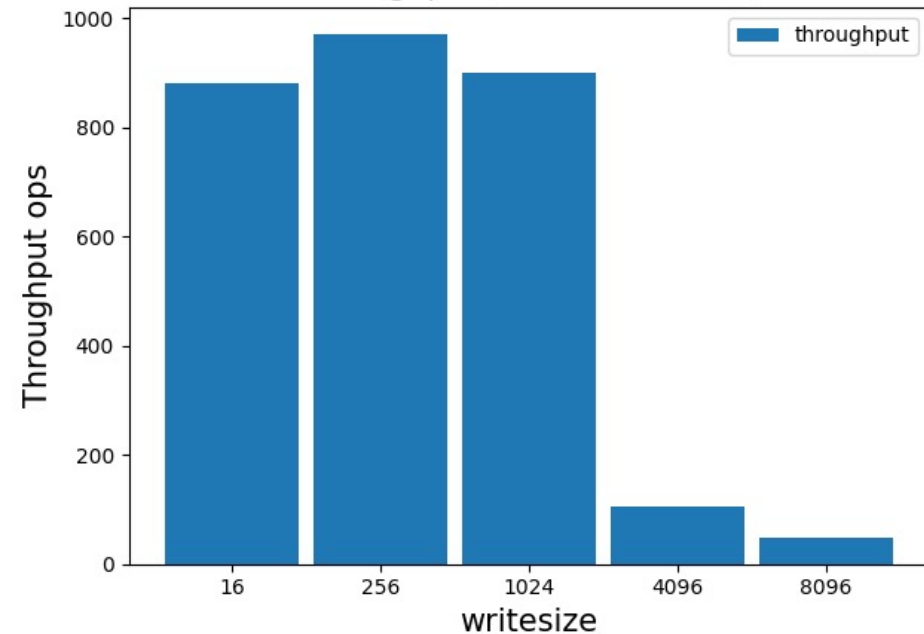
- (1) OW increase overhead when increase # of client
- (2) The throughput figure shows that this filesystem can scale

Evaluation (1 Write + 1 Read: d/d writesize)

Latency (# of client = 16)

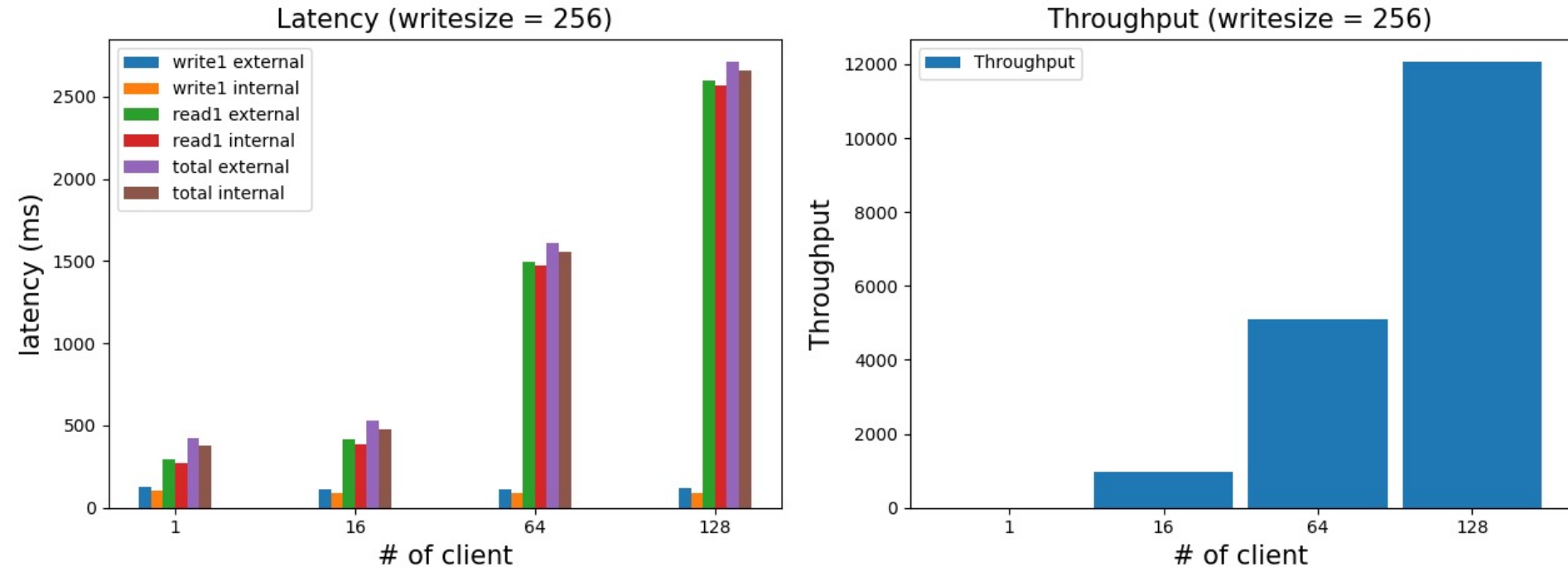


Throughput (# of client = 16)



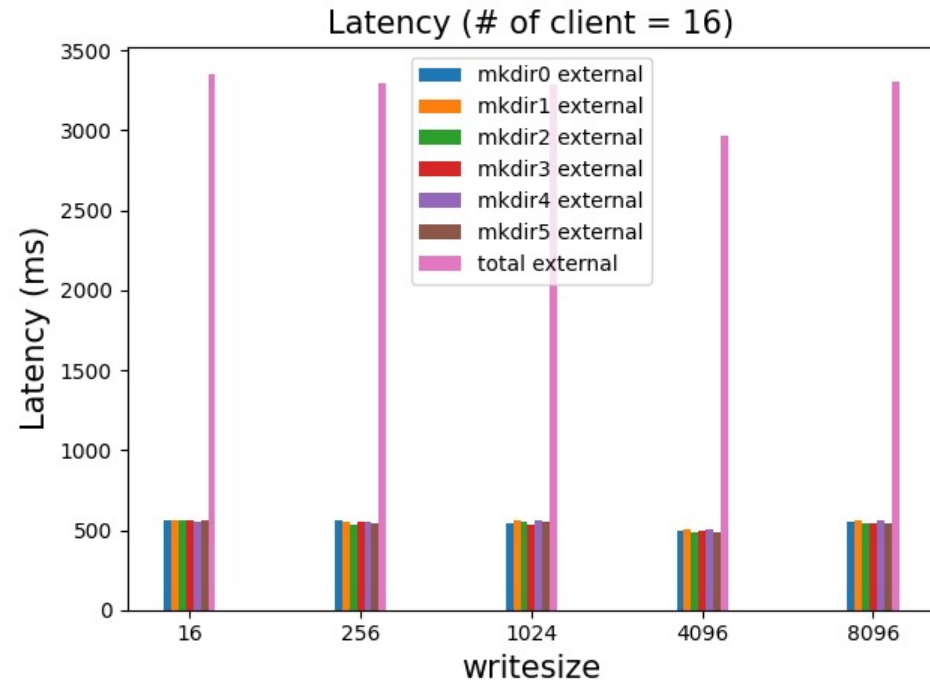
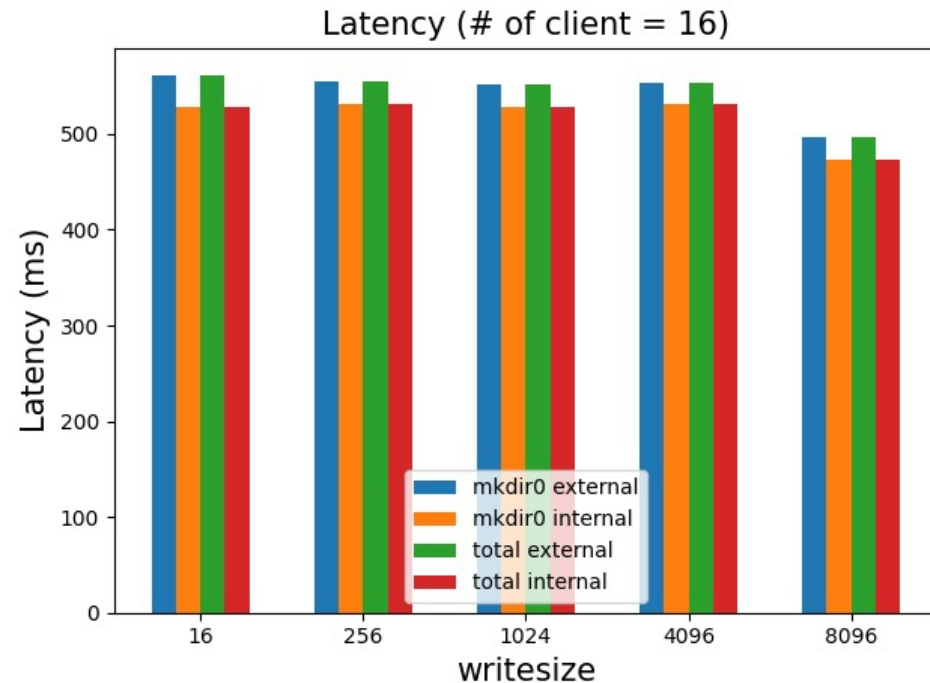
- (1) OW have large overhead when writesize is over 4096
- (2) The operation latency is dominated by read operation (may due to underlying Cassandra)

Evaluation (1 Write + 1 Read: d/d client)



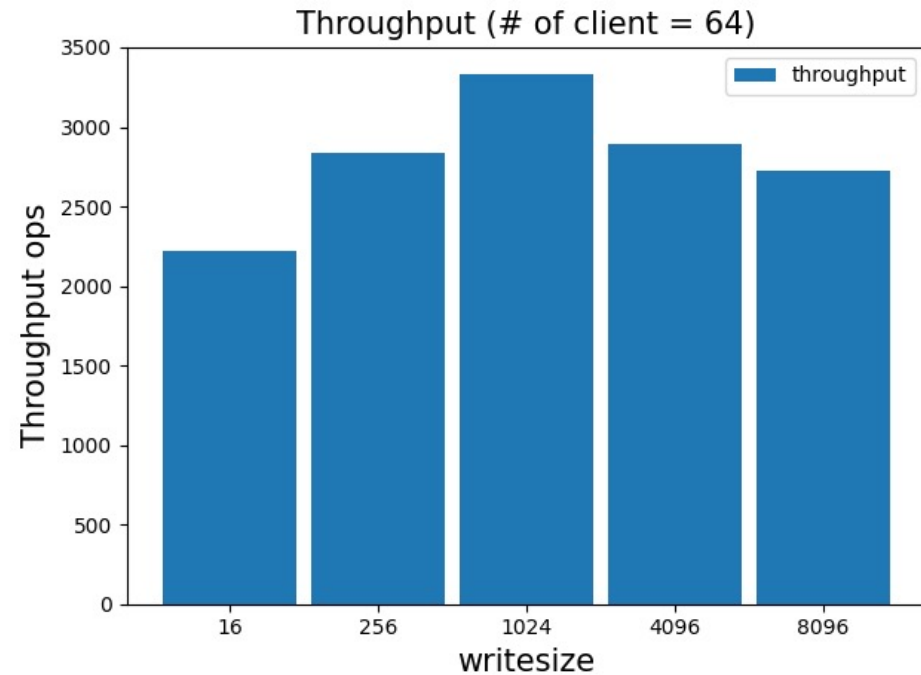
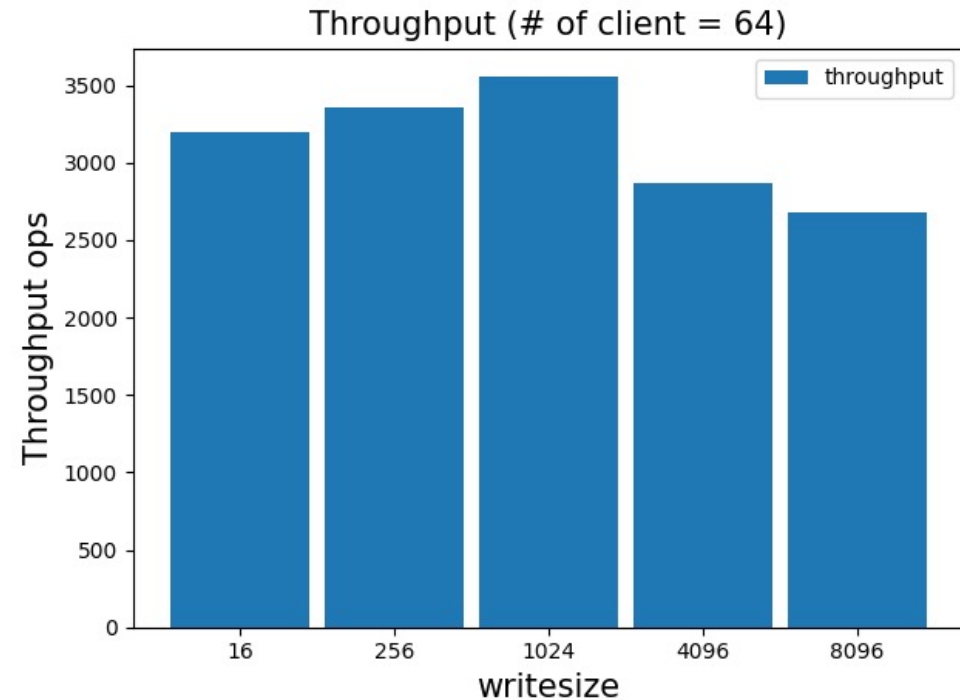
- (1) Write operation latency almost stays as constant
- (2) Read operation still the dominate factor in this setup

Evaluation (Mkdir)



- (1) Pure index operation
- (2) Do not observe performance degrade when writesize ≥ 4096 (expected because this test never use this variable).
- (3) Adding a new file inode takes 500 ms

Evaluation (Mkdir)



- (1) The throughput follows same pattern when pathlen = 1 and pathlen = 6
- (2) The total operation have a maximum number and spread to all instance

Plan to achieve but not done

- Handling failures
- Caching (inode/writes)

Next Step

- Add handling failures
- Add caching (inode/writes)
- Add different type of disks
- Cost and billing

What you have learned.

- OpenWhisk impose large latency
- Share connections and caches would improve the system greatly
- Cassandra does not have fast reads operations thus not suitable for serving the index storage.
- Batch reads and cache from index storage can also improve the performance
- Should redesign inode structure to reduce the read path latency

CS 7680 Serverless Computing (Special Topics in Systems)

DEMO