FalconFS: Distributed File System for Large-Scale Deep Learning Pipeline

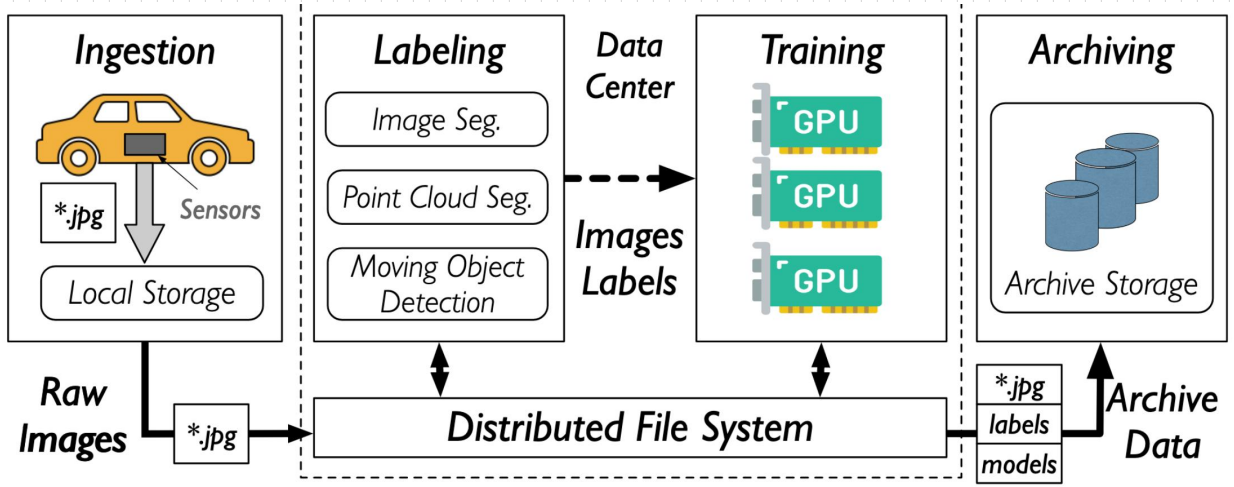
Jingwei Xu, Junbin Kang, Mingkai Dong, Mingyu Liu, Lu Zhang, Shaohong Guo, Ziyan Qiu, Mingzhen You, Ziyi Tian, Anqi Yu, Tianhong Ding, Xinwei Hu, and Haibo Chen

Presenter: Chi Zhang, Jiahao Li

Observations & Challenges

Workload Patterns in DL pipelines

1. Numerous small files in large directories



DL pipeline for autonomous driving in Huawei

content format	images. point clouds, etc.
dataset scale	300 billion files, hundreds of PiB data
file size	< 256KiB
directoried by	timestamps, vehicle ID, camera ID, etc.

Observations & Challenges

Workload Patterns in DL pipelines

2. Burst file access

During the labeling stage, inference tasks **access all the files in one directory** and then another.

⚠ Challenge 1: Burst file access within one directory causes MDS load imbalance

Observations & Challenges

Workload Patterns in DL pipelines

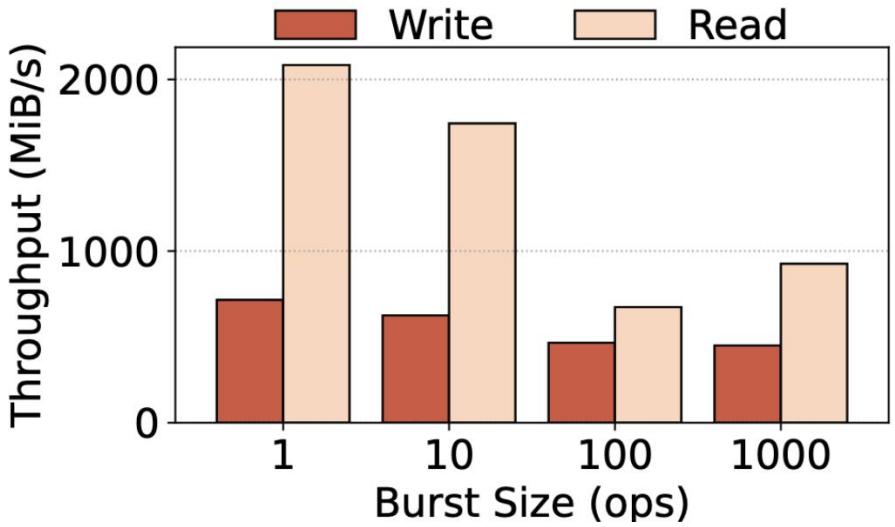
2. Burst file access

During the labeling stage, inference tasks **access all the files in one directory** and then another.

⚠ Challenge 1: Burst file access within one directory causes MDS load imbalance

- System: CephFS, 4 MDS
- Workload: accessing 64KiB files in the same directory with different burst sizes

DFS performance degrades as burst size increases



Observations & Challenges

Workload Patterns in DL pipelines

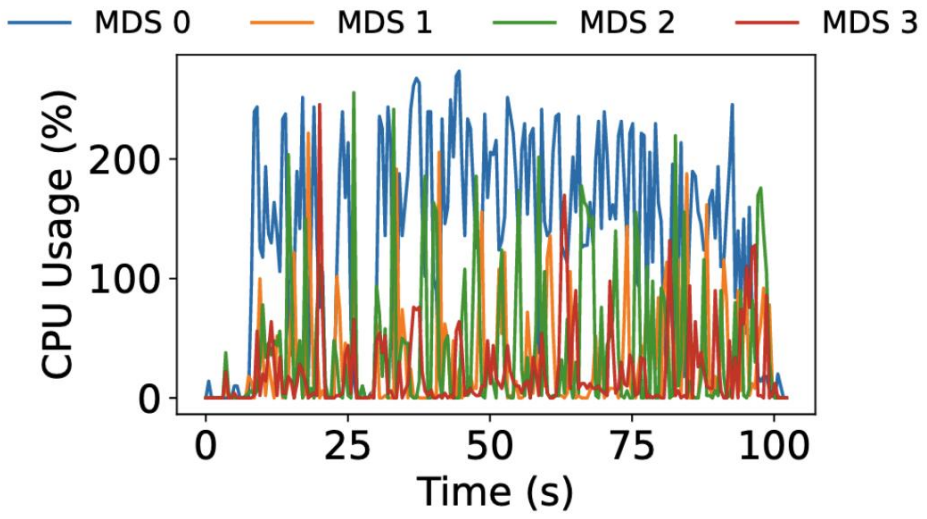
2. Burst file access

During the labeling stage, inference tasks **access all the files in one directory** and then another.

⚠ Challenge 1: Burst file access within one directory causes MDS load imbalance

- System: CephFS, 4 MDS
- Workload: accessing 64KiB files in the same directory with different burst sizes

Severe load imbalance observed during read operations



Load variance
(read operations with 100 burst size)

Observations & Challenges

Workload Patterns in DL pipelines

3. Random file traversal

Each file in dataset is **randomly** accessed **exactly once** in one epoch.

Unfriendly access patterns necessitate a larger dentry cache for efficiency.

4. Tight resource budget on client node

Online data augmentation consumes excessive CPU and memory, leaving insufficient resources for the DFS.

Challenge 2: Client-side metadata cache is inefficient

Observations & Challenges

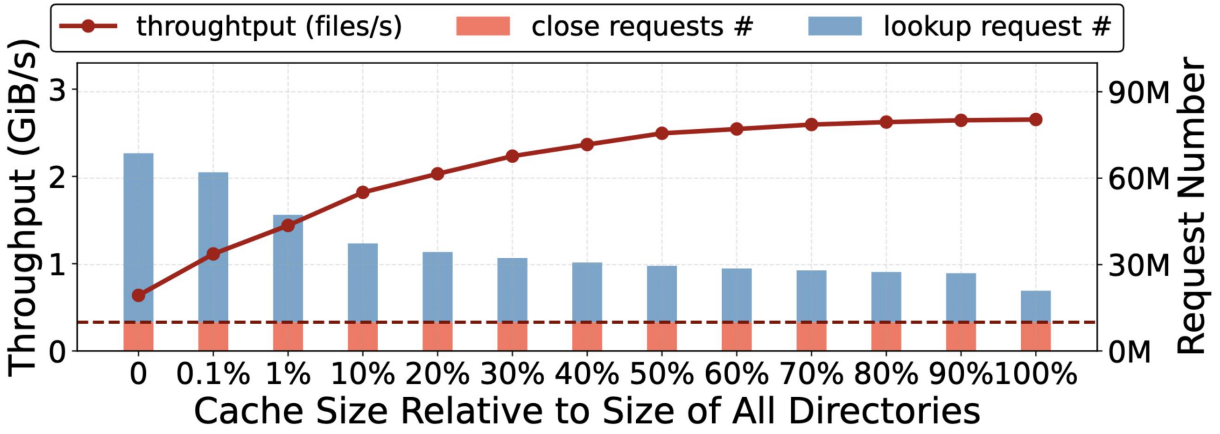
Workload patterns in DL pipelines

3. Random file traversal
4. Tight resource budget on client node

⚠ Challenge 2: Client-side metadata cache is inefficient

- System: CephFS cluster 4 MDS
- Workload: 512 threads randomly read 10 million small files stored across 1 million directories with a depth of 7 levels.

1. Smaller cache size results in increased request number and worse read throughput.
2. 10% in training set can be as large as 80GB, which is prohibitively expensive for clients.



Observations & Challenges: Summary

Workload Patterns in DL pipelines

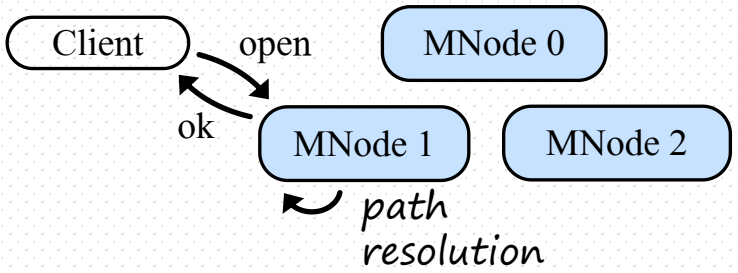
1. Numerous small objects in large directories
2. Burst file access
3. Random file traversal
4. Tight resource budget on client node

Challenges for DFS targeting DL workloads

1. Burst file access within one directory causes MDS load imbalance
2. Client-side metadata cache is inefficient

Proposal: DFS with Stateless Client

- **Abandon metadata cache on the client side**
- **Move path resolution to server side**



still provide one-hop access for most operations

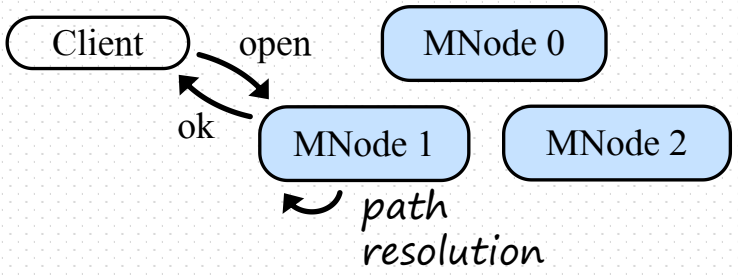
Two key problems

How could a client find the right MDS when performing file operations?

How could a MDS resolve path locally?

Proposal: DFS with Stateless Client

- **Abandon metadata cache on the client side**
- **Move path resolution to server side**



still provide one-hop access for most operations

	Key	Value	Partition by
dentry	pid, name	id, perm.	replicated
inode	pid, name	id, attr	Hybrid indexing

Design: Hybrid Metadata Indexing

How to find the right MDS?

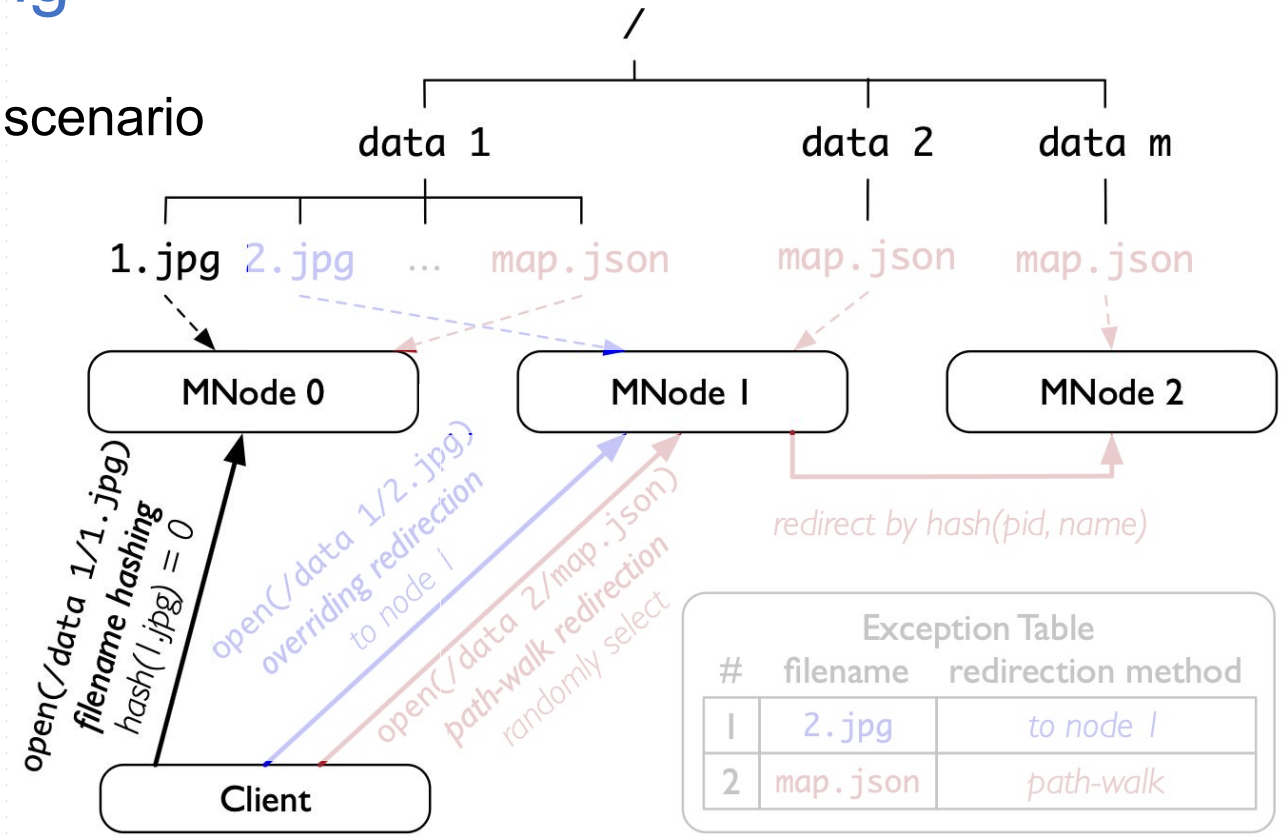
Common case: filename hashing

- Fast
- Keep load balance under large directory scenario

Example

Executing open("/data 1/1.jpg")

1. client calculates hash("1.jpg") = 0
2. client sends request to MNode 0
3. MNode 0 performs path resolution



Design: Hybrid Metadata Indexing

How to find the right MDS?

Filename hashing may still cause load imbalance

Scenario 1: Hot Filename

Some filenames may occur many times

`Makefile, .git, .DS_Store, etc.`

Hot filenames may cause load imbalance

Method: path-walk redirection

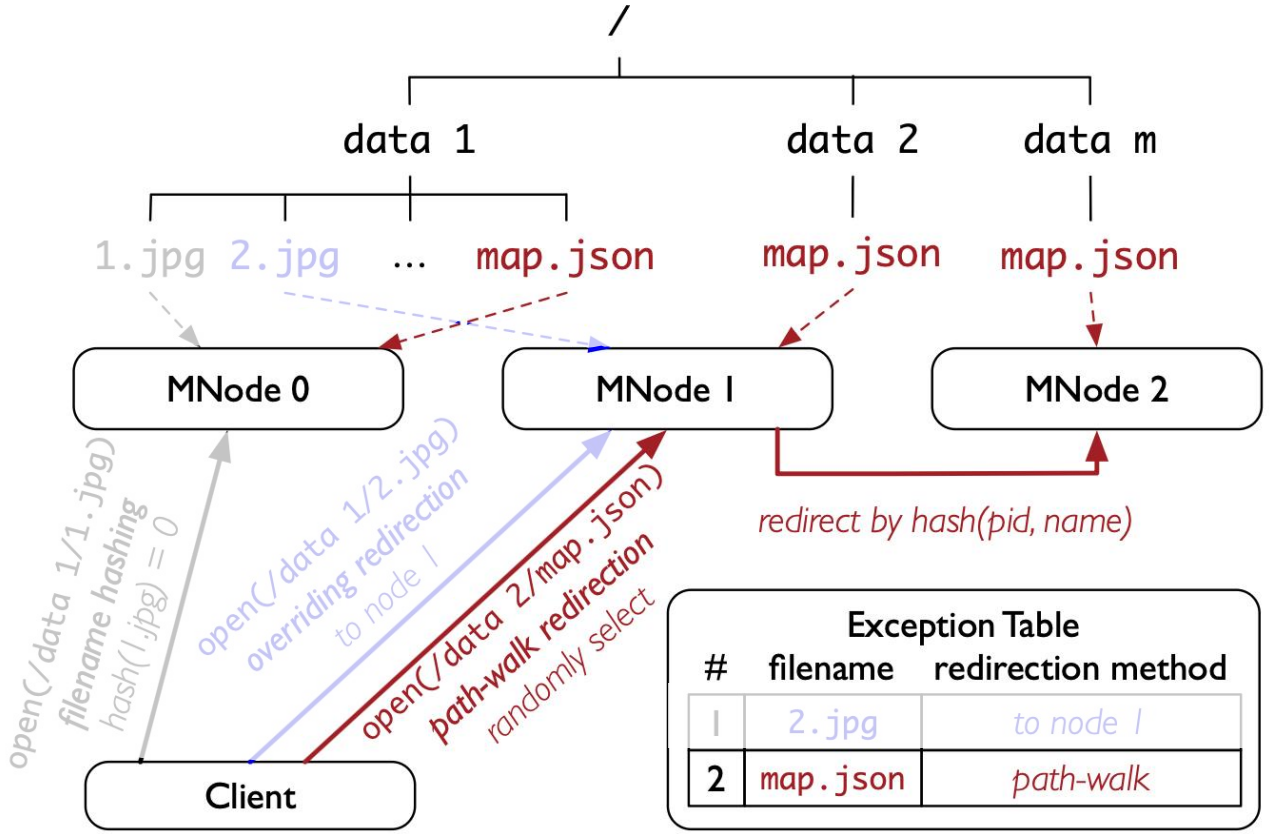
Use `hash(pid+filename)`

instead of `hash(filename)` to locate inode

Example

Executing ("/data 2/map.json"):

1. client finds map.json in exception table
2. client randomly sends request to a MNode
3. MNode 1 performs path resolution and calculate hash(pid+filename) = 2
4. MNode 1 forward the request to MNode 2



Design: Hybrid Metadata Indexing

How to find the right MDS?

Filename hashing may still cause load imbalance

Scenario 2: Hash Variance

The number of unique filenames in filesystem is not far more than that of MNodes, resulting in a highly uneven filename distribution across MNodes.

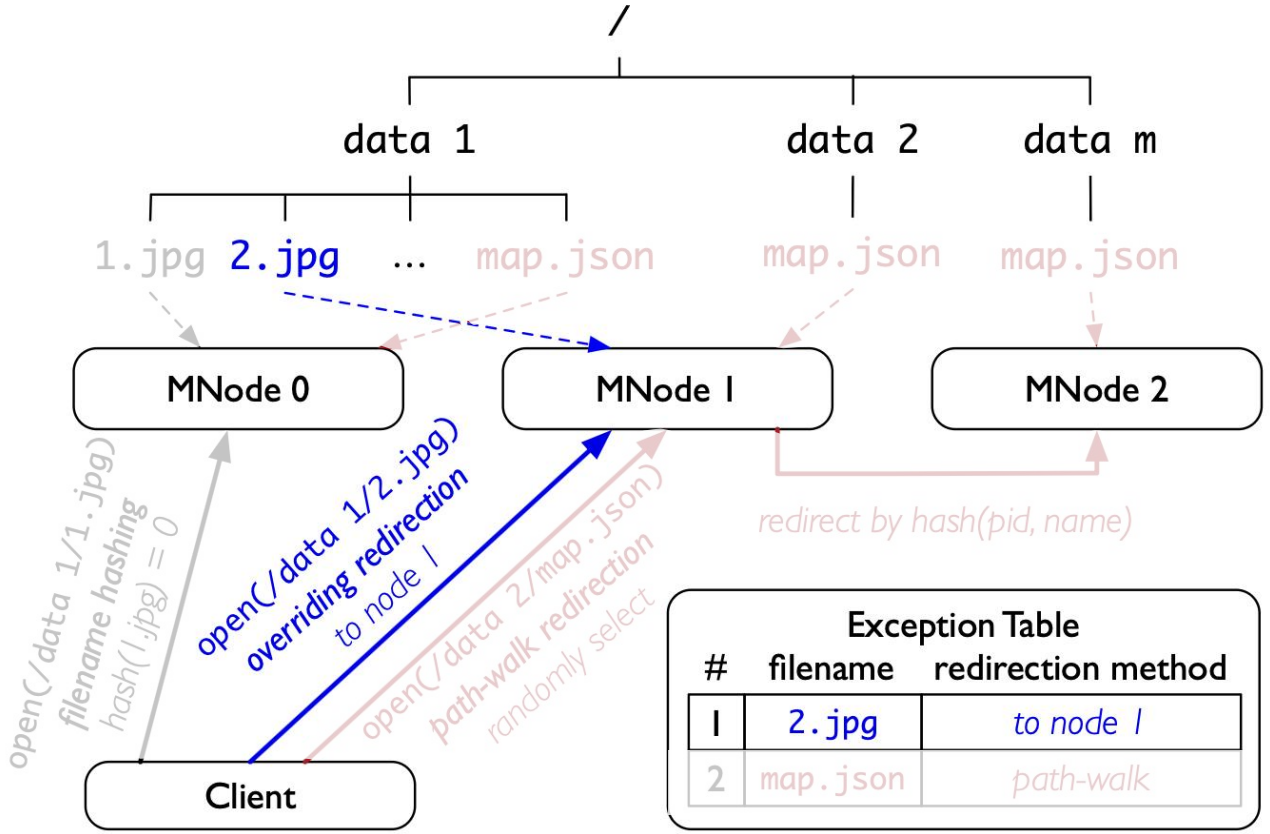
Method: Overriding redirection

Directly specify target MNode in exception table

Example

Executing open("/data 1/2.jpg")

1. client finds 2.jpg in exception table
2. client directly sends request to MNode 1



Design: Hybrid Metadata Indexing

How to find the right MDS?

Use exception table

Location

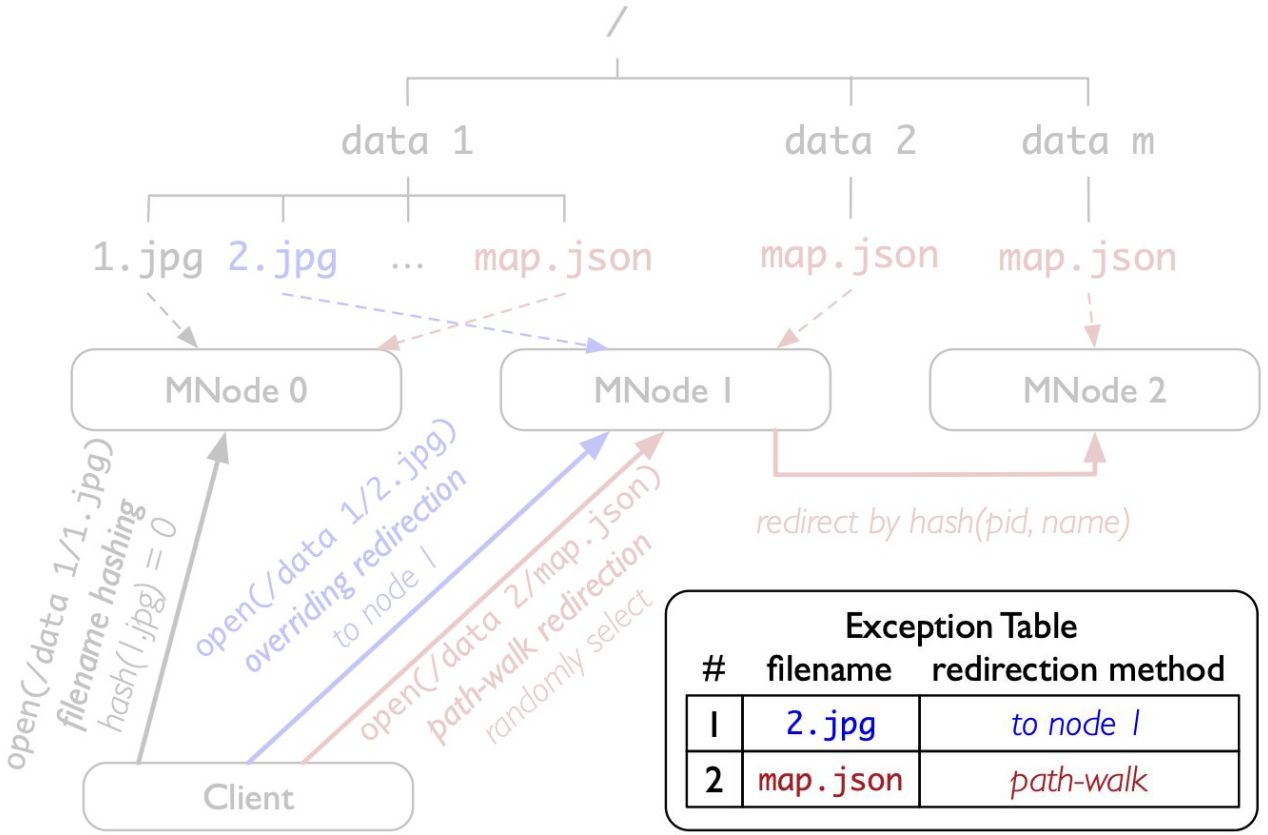
MNodes, clients, and the coordinator

Synchronization

Updated by coordinator, eagerly pushed to MNodes, lazily fetched by clients when receiving response from MNodes.

Request Validation

MNodes validate every request using its local exception table and forward misdirected requests to the proper destinations.



Design: Hybrid Metadata Indexing

Load is imbalanced when inode ratio of a node exceeds $\frac{1}{n} + \epsilon$.

Load balancing algorithm (assume n nodes):

1. Identify the most and least loaded nodes, $N_{\max}$ and $N_{\min}$ (inode counts are denoted by $\langle N_{\max} \rangle$ and $\langle N_{\max} \rangle$).
2. Let F be the *most frequent file name* in $N_{\max}$.
3. Estimate the load after two redirection approaches are applied:
   I. Path-walk redirection: $\langle N_i \rangle' = \begin{cases} \langle N_{max} \rangle - \frac{n-1}{n}|F|, \text{ if } N_i = N_{\max} \\ \langle N_i \rangle + \frac{1}{n}|F|, \text{ if } N_i \neq N_{\max} \end{cases}$
   II. Override redirection: $\langle N_i \rangle' = \begin{cases} \langle N_{max} \rangle - |F|, \text{if } N_i = N_{\max} \\ \langle N_{min} \rangle + |F|, \text{ if } N_i = N_{\min} \\ \langle N_i \rangle, \text{ if } N_i \notin \{N_{\min}, N_{max}\} \end{cases}$
4. Choose the method that minimizes the max inode count.

Hybrid Metadata Indexing: Summary

❑ Common case: filename hashing

- Fast
- Keep load balance under large directory scenario

❑ Two load imbalance cases: selective redirection

Hot Filenames Path-walk Redirection

Use hash(pid-filename) instead of hash(filename) to locate inode

Hash Variance Overriding redirection

Directly set a new MNode

❑ How to record redirected filenames?

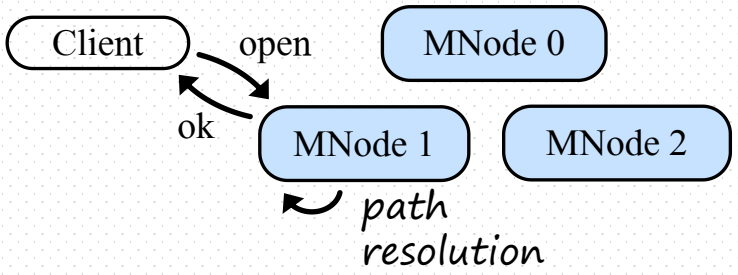
Shared Exception Table: Updated by coordinator, eagerly pushed to MNodes, lazily fetched by clients.

❑ How to decide which filename to be redirected?

Coordinator performs a statistical load balancing algorithm.

Proposal: DFS with Stateless Client

- **Abandon metadata cache on the client side**
- **Move path resolution to server side**



still provide one-hop access for most operations

Two key problems

How could a client find the right MDS when performing file operations?

❑ Hybrid Metadata Indexing

How could a MDS resolve path locally?

Design: Lazy Namespace Replication

How could a MDS resolve path locally?

Maintain a consistent namespace replica on each MNode and a coordinator.

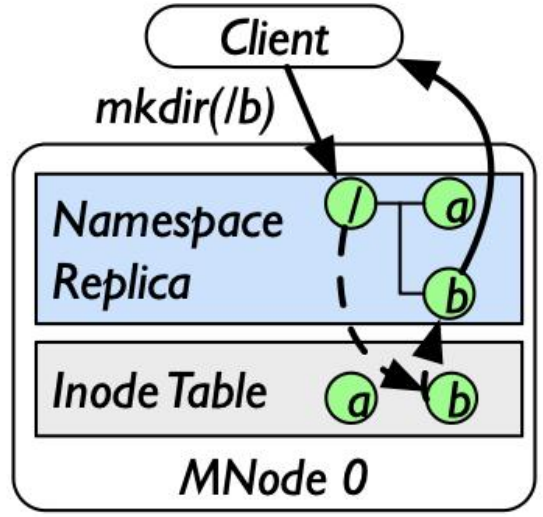
Key principles to reduce consistency overhead:

- Delaying synchronization until access
- Using invalidation as lightweight locking

Design: Lazy Namespace Replication

Procedure of mkdir(/b):

1. Calculate /b's inode location using hybrid metadata indexing
2. Send request to the corresponding MNode (i.e., MNode 0)
3. MNode 0 perform path resolution and create dentry and inode locally.



Directory creation

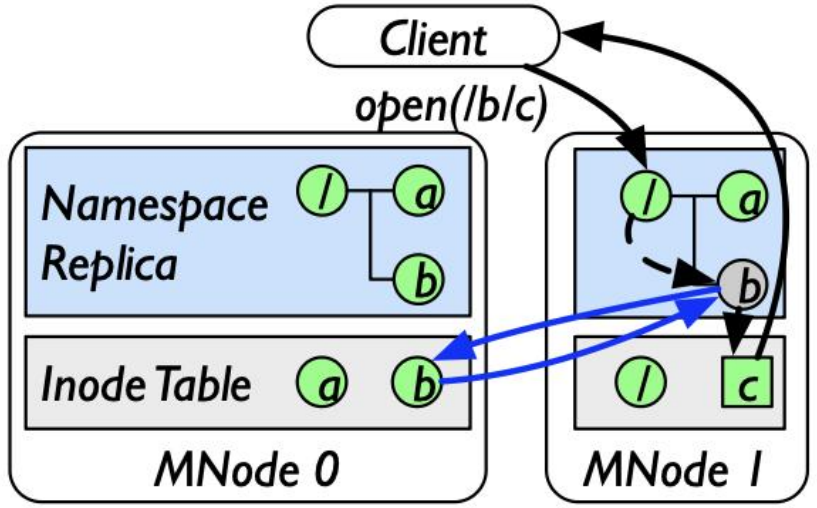
FalconFS does not proactively replicate the newly created directory entry.

Design: Lazy Namespace Replication

Procedure of open(/b/c):

1. Calculate /b/c's inode location using hybrid metadata indexing.
2. Send request to the corresponding MNode (i.e., MNode 1).
3. MNode 1 sends a lookup to MNode 0 to fetch the missing entry to complete path resolution and continues processing.

Directory entries are replicated lazily



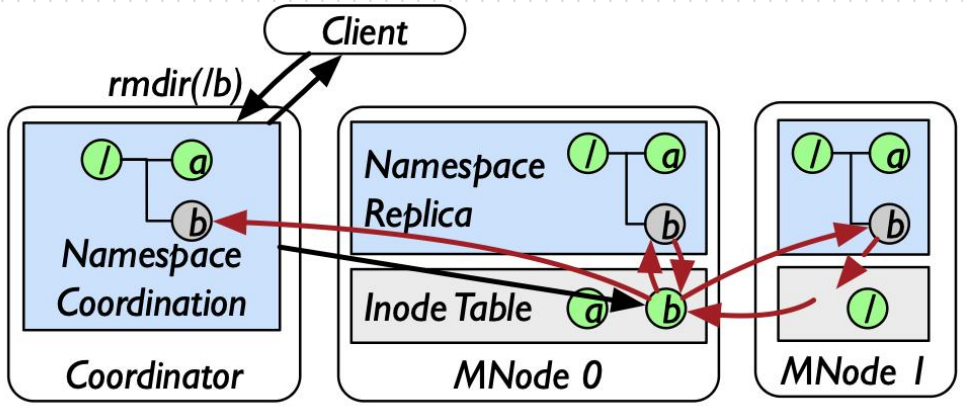
Directory lookup and miss handling

Design: Lazy Namespace Replication

Procedure of rmdir(/b):

1. Send request to the coordinator.
2. The coordinator forwards the request to /b's inode owner (i.e., MNode 0).
3. MNode 0 broadcasts invalidation info.
4. All MNodes invalidate /b in their local replicas and reply whether /b is empty.
5. MNode 0 collect all replies, if no children exist, remove /b's inode and finish request.

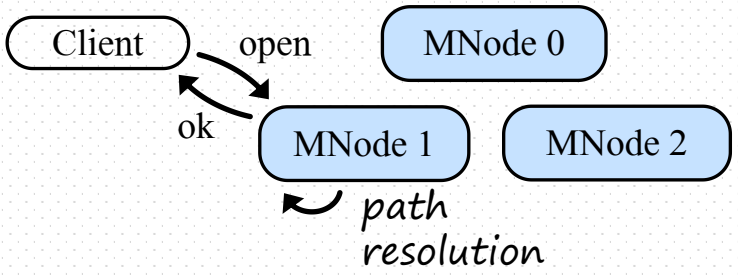
setattr and rename are similar with rmdir.



Directory change permission, removal and rename

Proposal: DFS with Stateless Client

- **Abandon metadata cache on the client side**
- **Move path resolution to server side**



still provide one-hop access for most operations

Two key problems

How could a client find the right MDS when performing file operations?

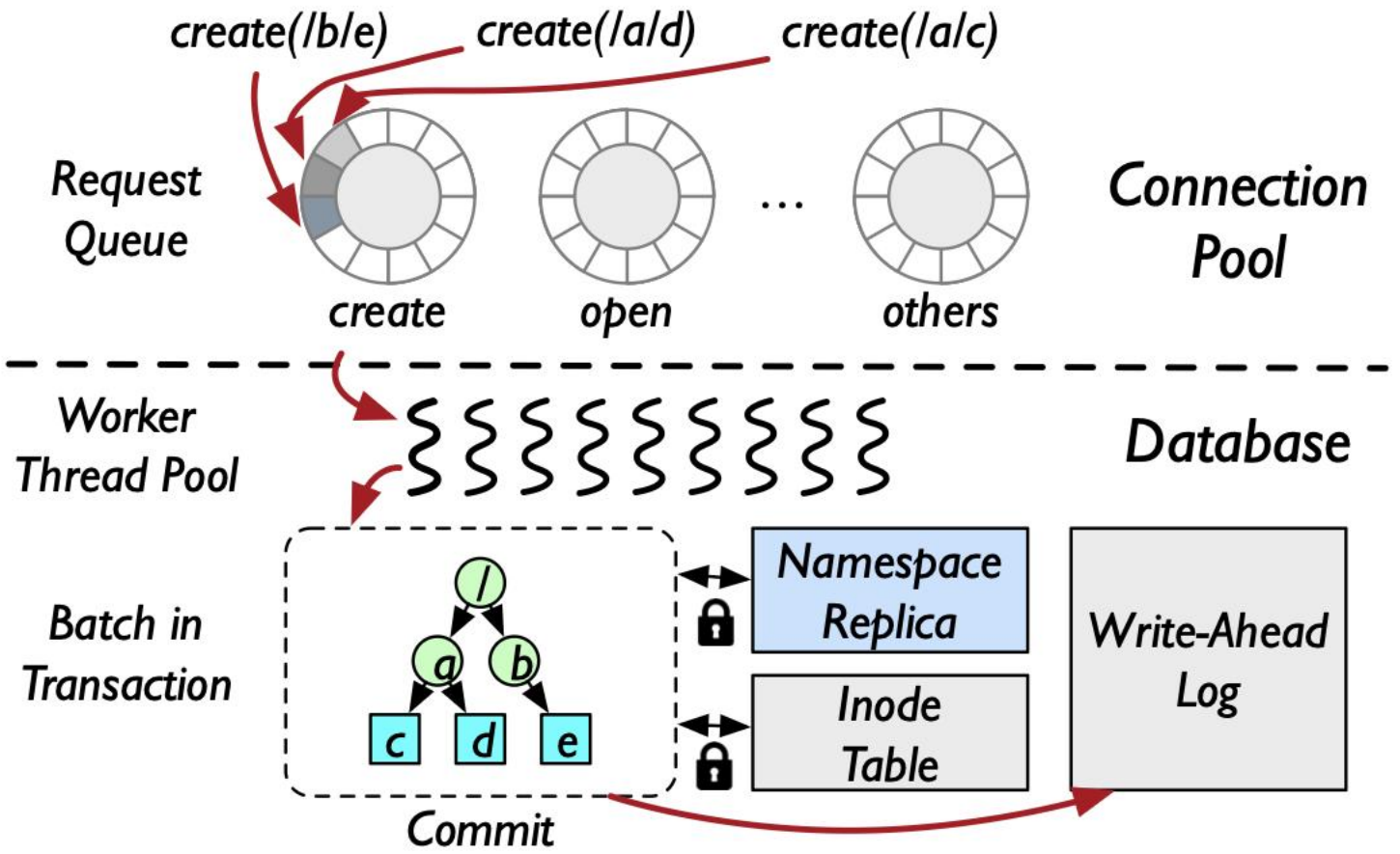
❑ Hybrid Metadata Indexing

How could a MDS resolve path locally?

❑ Lazy Namespace Replication

Design: Concurrent Request Merging

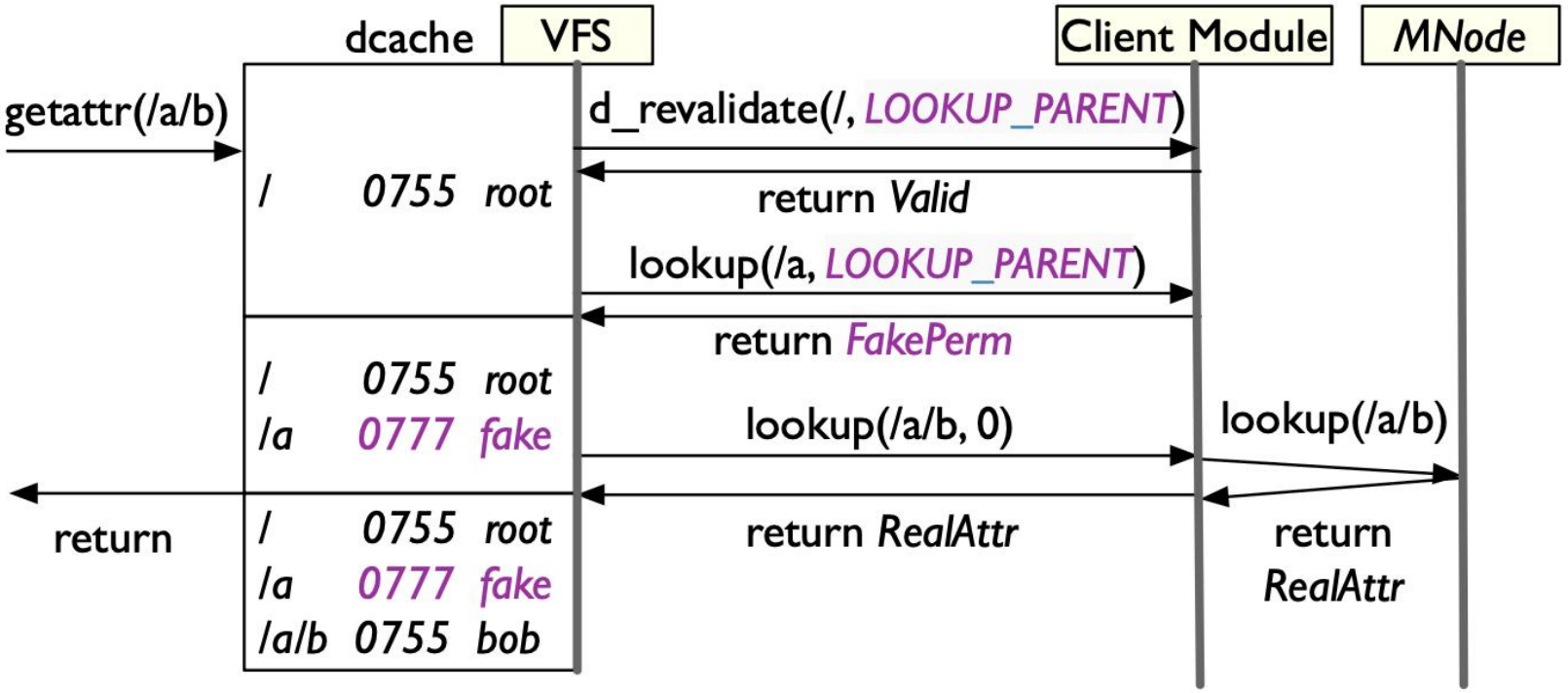
all the requests in the same queue are executed in a single database transaction



Design: VFS shortcut

Bypass VFS dentry cache

- Use 0777 fake permission to pass VFS permission check
- Utilize d_revalidate call to avoid fake metadata from being exposed to users



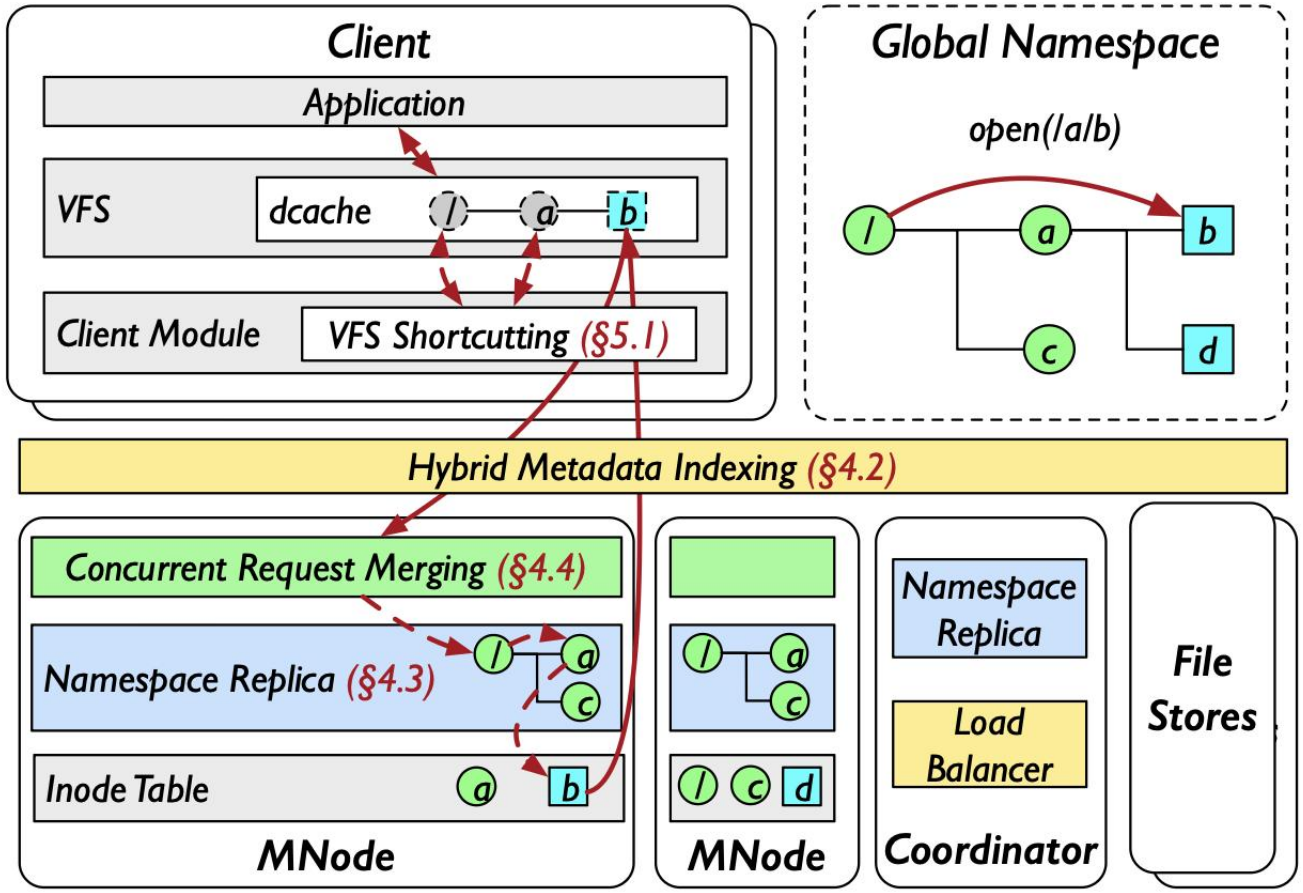
intermediate and final lookups.

Design Summary

- **Hybrid metadata indexing**
  Perform inode partition and load balancing
- **Lazy namespace replication**
  Make MNode resolve path locally
- **Concurrent Request Merging**
  Improve single MDS throughput
- **VFS Shortcutting**
  Bypass VFS cache

Metadata Scheme

	Key	Value	Partition by
dentry	pid, name	id, perm.	replicated
inode	pid, name	id, attr	hybrid indexing



barrier

Experimental Setup

Testbed

- 13 dual socket machines, abstract each machine into two nodes.
- Each node is bound to one socket, one SSD and one NIC.
- Restrict server resource to 4 cores per node for saturation.

Baseline Systems

- CephFS v12.2.13
  - libcephfs client
- JuiceFS v1.2.1
  - metadata engine: TiKV
  - data engine: TiKV
  - VFS client
- Lustre v2.15.6
  - VFS client

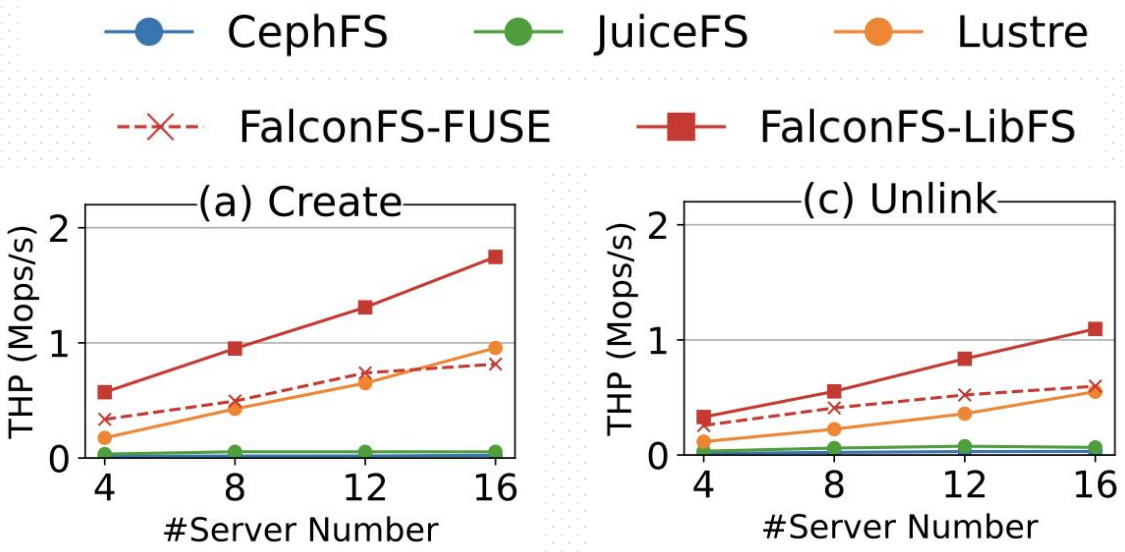
CPU	2 × Intel Xeon 3.00 GHz, 12 cores
Memory	16 × DDR4 2933 MHz 16 GB
Storage	2 × NVMe SSD 960 GB
Network	2 × 100 GbE

Configuration of a single machine

Replication is disabled for both metadata and data.

Evaluation – Metadata Performance

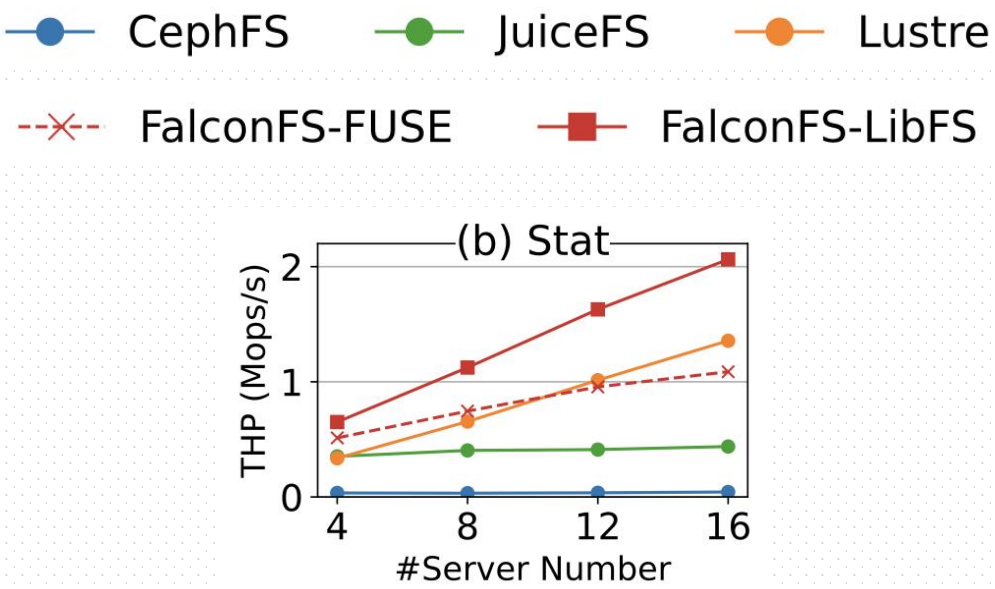
- ❑Both FalconFS's FUSE client and LibFS client is evaluated.
- ❑Assume private directory access and all cache hit.
- ❑Different server number, record peak throughput.



- Achieve speedups of 0.82-2.26x on Lustre, and higher gains on CephFS and JuiceFS.
- FalconFS does not maintain mtime and ctime of directory, lowering create cost.
- Concurrent request merging effectively reduces WAL cost.

Evaluation – Metadata Performance

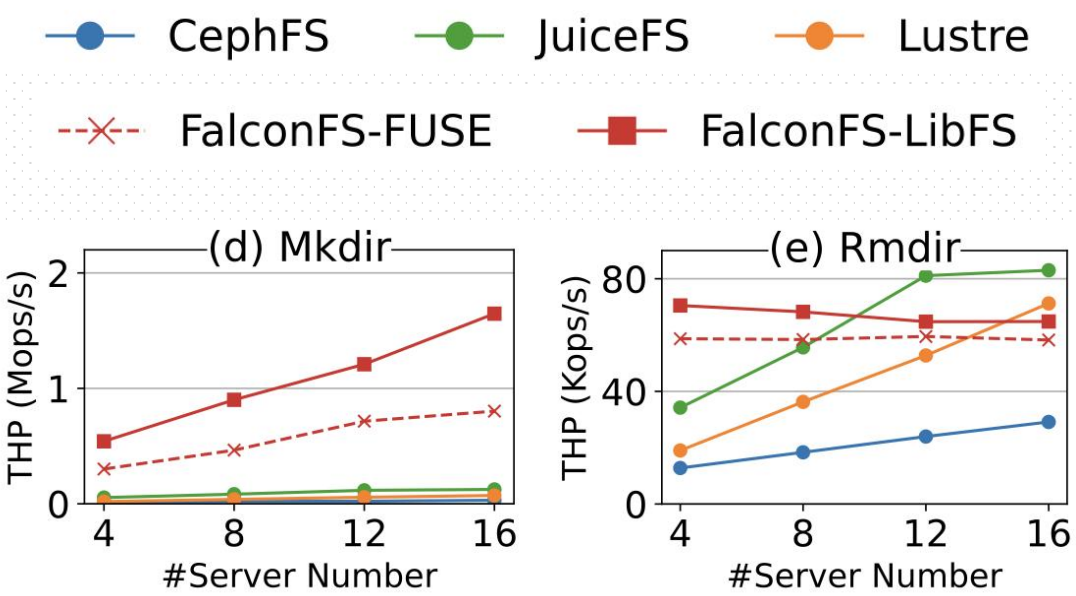
- ❑Both FalconFS's FUSE client and LibFS client is evaluated.
- ❑Assume private directory access and all cache hit.
- ❑Different server number, record peak throughput.



- Achieve speedups of 0.52-0.93x on Lustre.
- Concurrent request merging boosts server concurrency.
- No cache coherence lock overhead.

Evaluation – Metadata Performance

- ❑Both FalconFS's FUSE client and LibFS client is evaluated.
- ❑Assume private directory access and all cache hit.
- ❑Different server number, record peak throughput.

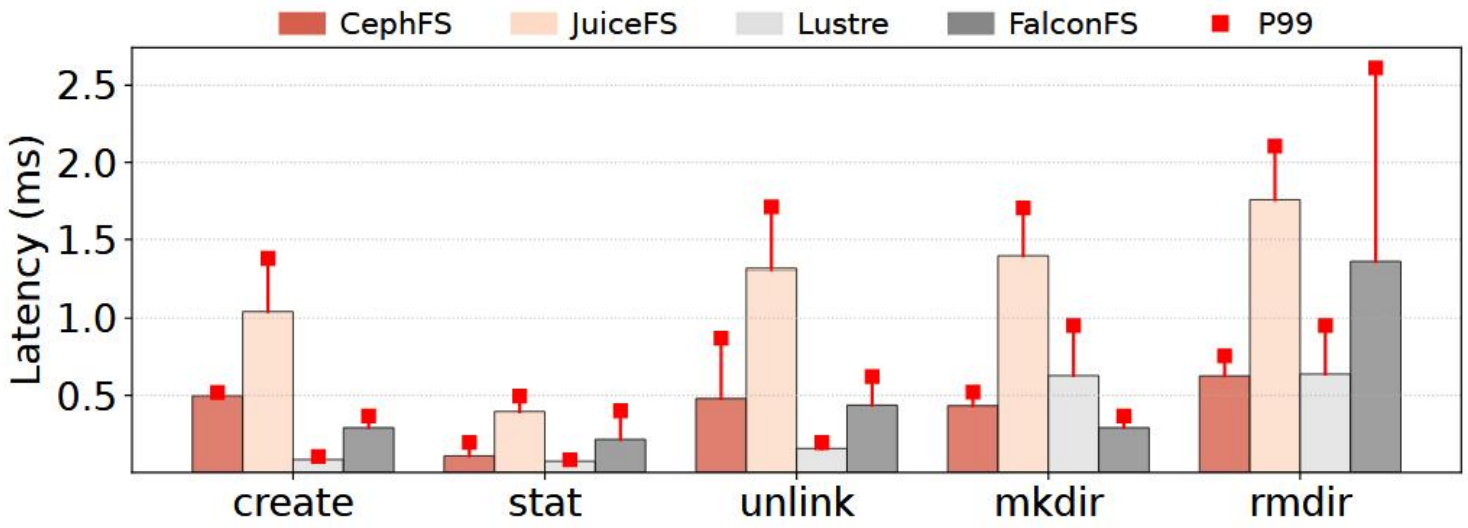


- Scalable performance for mkdir
- rmdir throughput declines as the number of servers grows.

Invalidation cost grows proportionally to cluster size

Evaluation – Latency

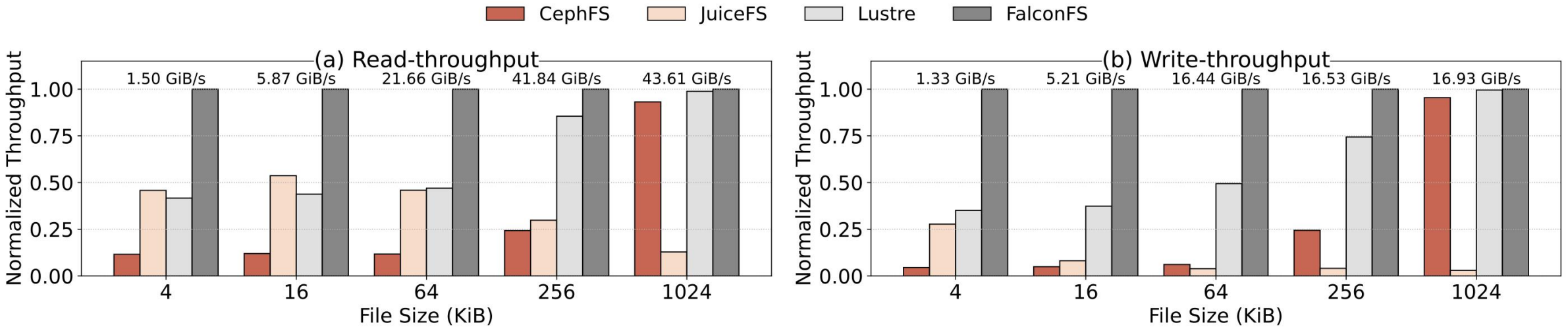
❑Four metadata servers, single client.



- FalconFS has higher latency than Lustre because it uses batch optimizations
- FalconFS shows higher p99 latency in rmdir due to the broadcasting of invalidation information.

Evaluation – Data Performance

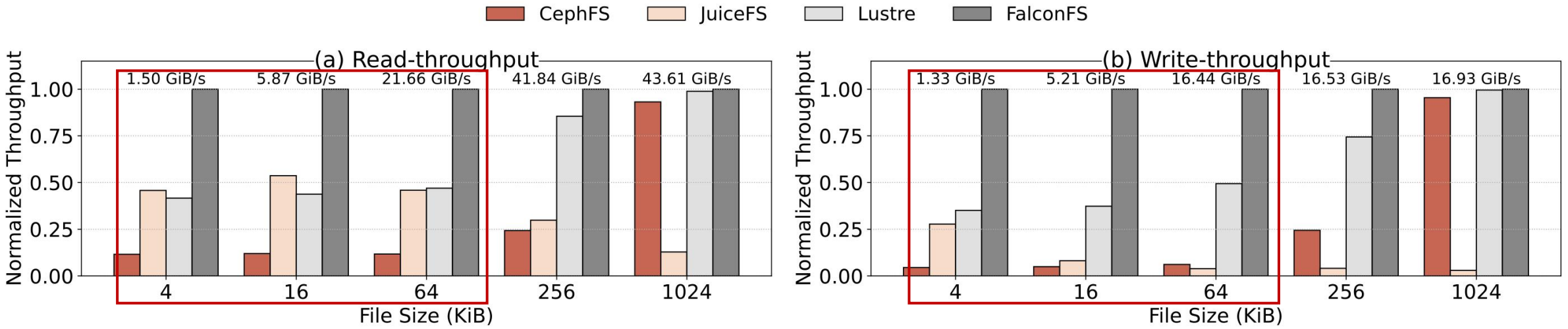
- ❑Vary file size from 4KB to 1024KB.
- ❑4 metadata servers, saturate DFSs using 2560 client threads.
- ❑Each thread accesses 1024 pre-created files within private directories.



- FalconFS outperforms baseline systems in every setting.

Evaluation – Data Performance

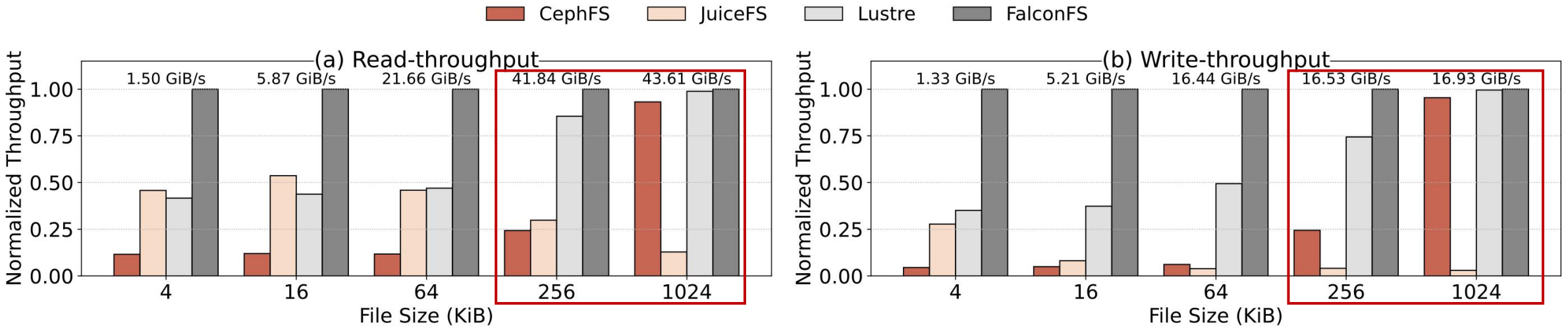
- ❑Vary file size from 4KB to 1024KB.
- ❑4 metadata servers, saturate DFSs using 2560 client threads.
- ❑Each thread accesses 1024 pre-created files within private directories.



- When file size < 256KB, throughput grows proportionally to file size, indicating **IO bound**.

Evaluation – Data Performance

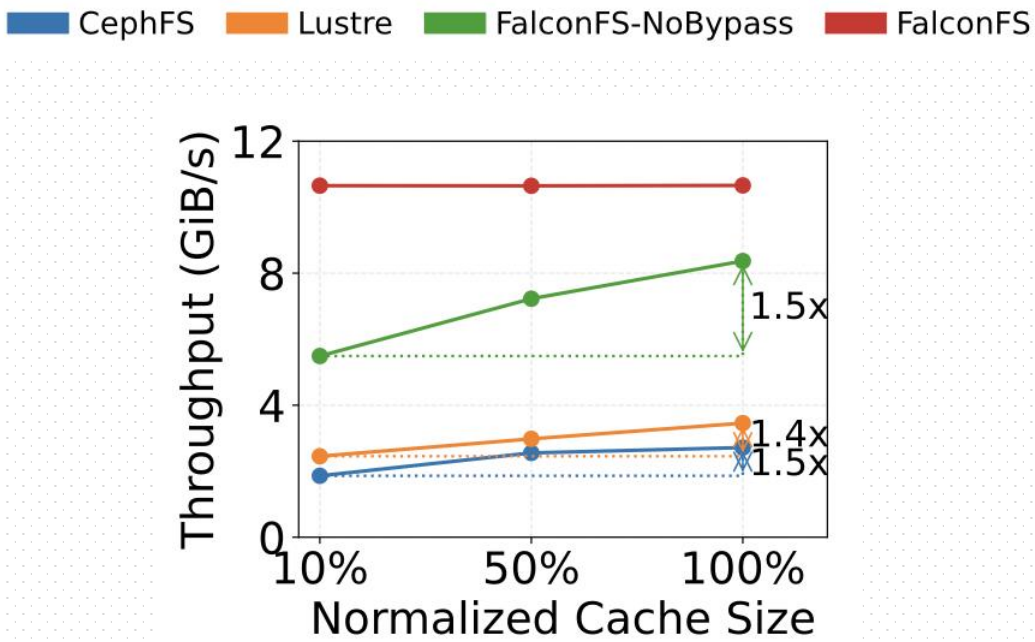
- Vary file size from 4KB to 1024KB.
- 4 metadata servers, saturate DFSs using 2560 client threads.
- Each thread accesses 1024 pre-created files within private directories.



- When file size < 256KB, throughput grows proportionally to file size, indicating **IO bound**.
- When file size > 256KB, SSD capacity reaches with 43GB/s and 16GB/s for read and write, respectively

Evaluation – Impact of Client Memory

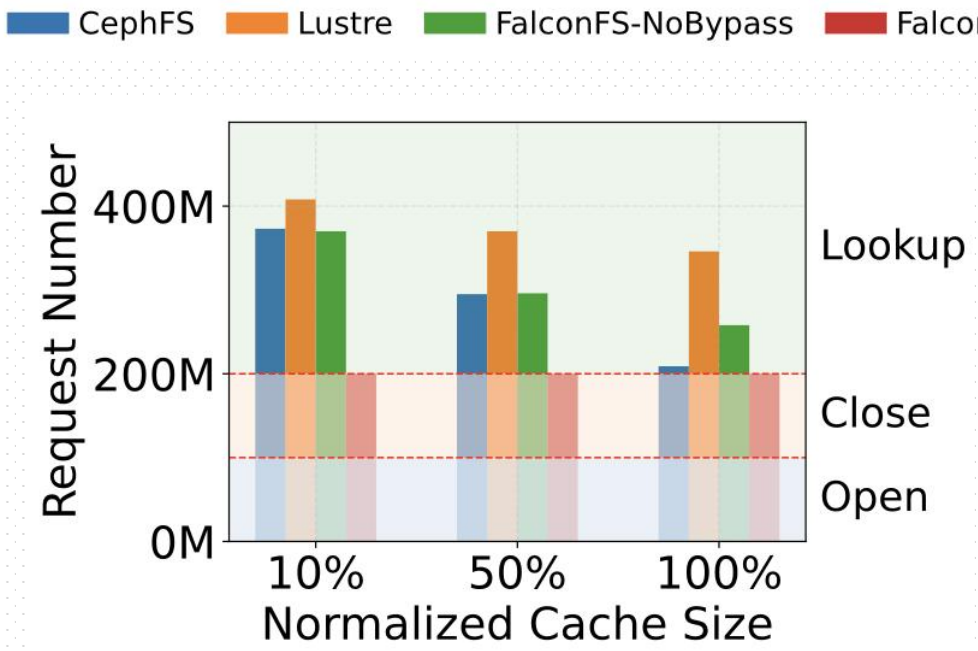
- Directory structure: 8-level, fanout: 10, filesize: 64KB
- FalconFS-NoBypass utilizes cache by VFS
- 4 metadata servers
- Workload: 256 clients, random file traversal in the directory tree



- FalconFS shows stable high throughput under different memory budgets.
- 1.4-1.5x performance gap between the 10% and 100% memory budget for cache schemes.

Evaluation – Impact of Client Memory

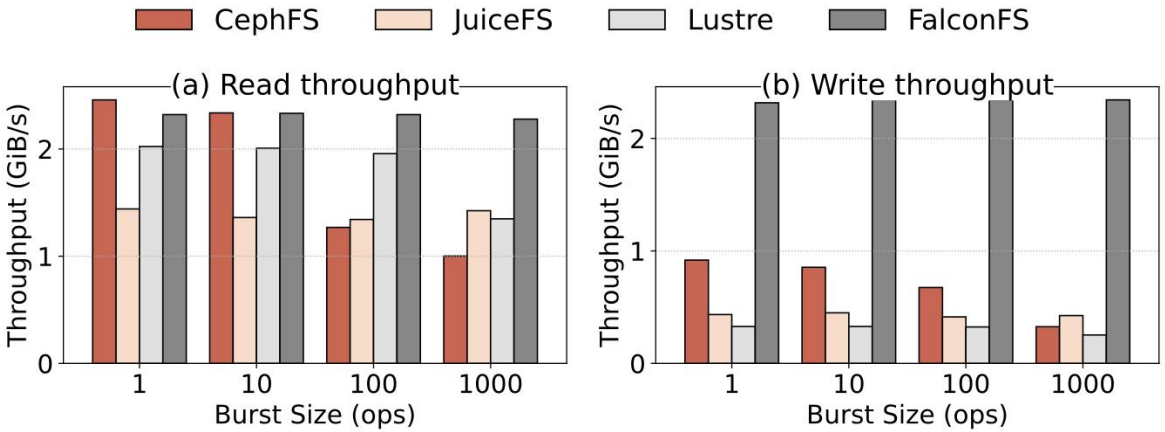
- Directory structure: 8-level, fanout: 10, filesize: 64KB
- FalconFS-NoBypass utilizes cache by VFS
- 4 metadata servers
- Workload: 256 clients, random file traversal in the directory tree



- FalconFS doesn't need lookup
- FalconFS-NoBypass still requires many lookup requests even under 100% memory budget due to extra space from file inode

Evaluation – Burst IO

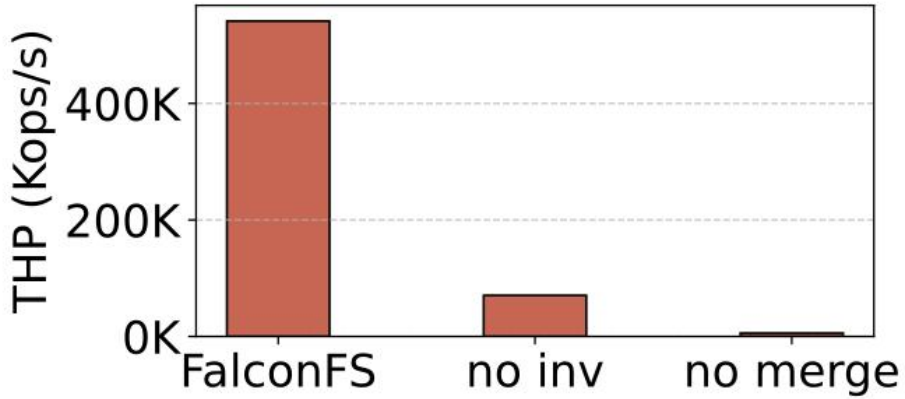
- 4 metadata servers
- Workload: 256 clients, sequentially access files within the same directory
- Burst size means the number of files that are accessed in the same directory



- CephFS and Lustre observes a degradation in read and write throughput as burst size grows.
- FalconFS does not suffer from burst IO.
- JuiceFS also does not degrade, due to constant load imbalance among metadata engine nodes.

Evaluation – Contribution breakdown

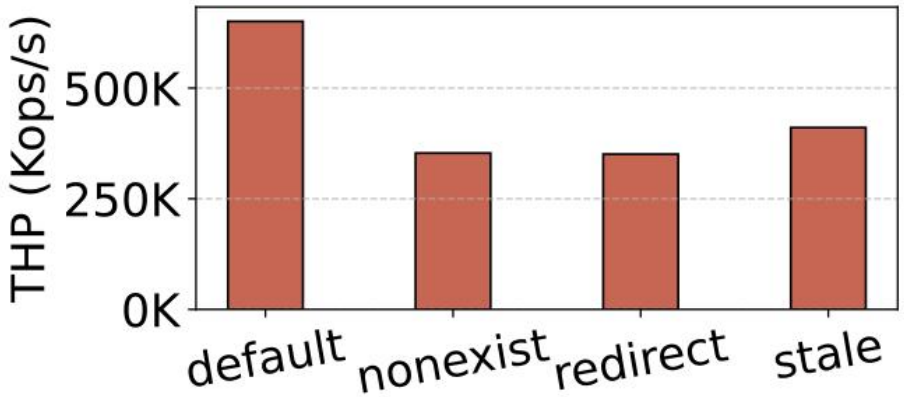
- 4 metadata server
- Check how each optimization contributes to FalconFS's mkdir operations.
- Under *no inv*, distributed transaction is used for each mkdir.
- Workload: 256 clients, mkdir on private directories



- *no inv* decreases throughput by 86.9%
- *no merge* further reduces throughput by 91.8%

Evaluation – Corner Case Performance

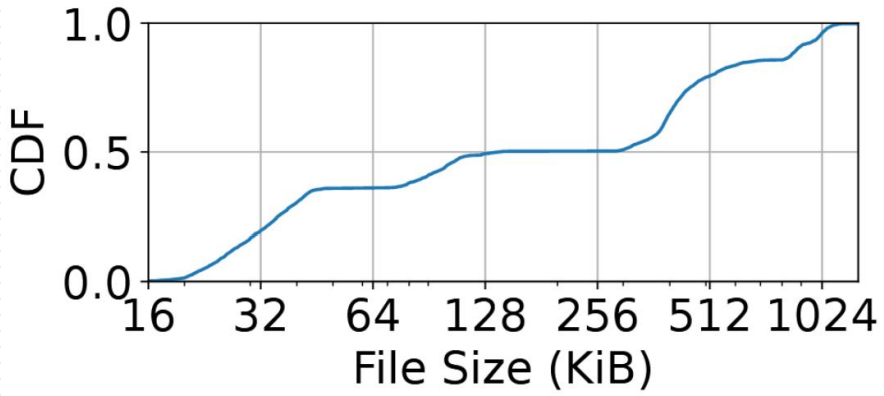
- 4 metadata server
- *nonexist*: lookup on a non-existing path
- *redirect*: lookup on a path redirected by path redirection.
- *stale*: client has a stale exception table
- Workload: 256 clients, getattr



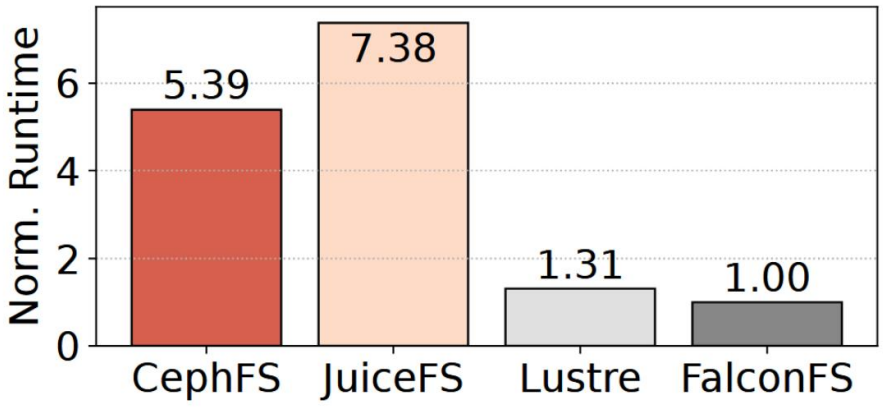
- Throughput reduces by 36.8%-49.6% under those corner cases due to additional hops.

Evaluation – Application: Labeling Task

- 4 metadata server, 12 data servers
- Workload: replay trace of splitting large images into small ones



(a) Distribution of file size.

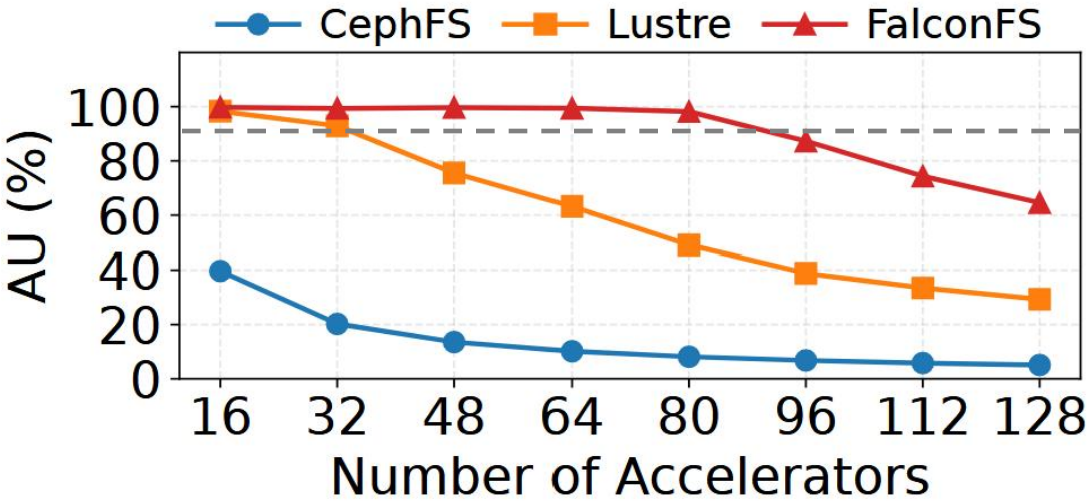


(b) Normalized trace runtime.

FalconFS reduces runtime by **23.8%-86.4%**

Evaluation – Application: Training Task

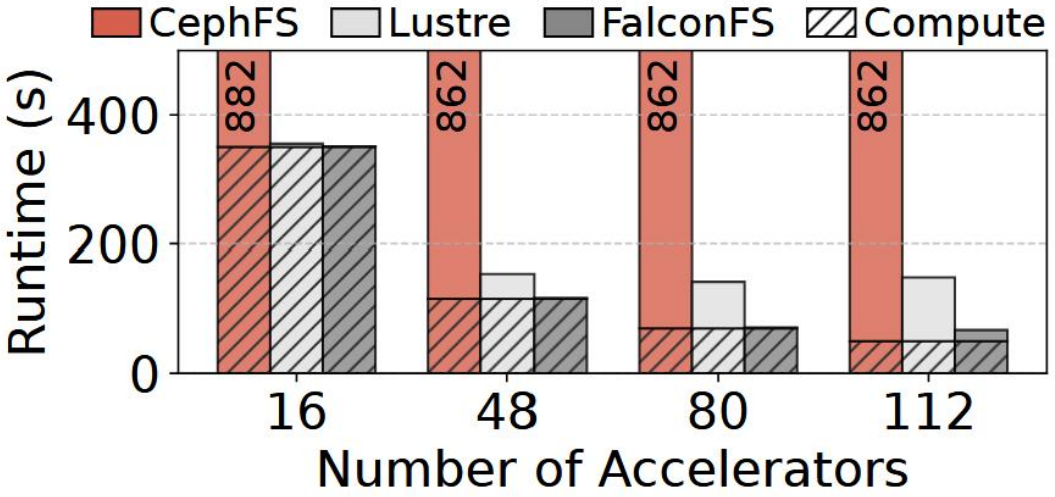
- 4 metadata server, 12 data servers
- Dataset: 10 million files, 1 million directories, filesize=112KB, 1064GB
- Workload: MLPerf Storage Benchmark, ResNet-50
- $AU = T_{\text{compute}}/T_{\text{total}}$, indicating the ratio of time accelerator is computing.



- With 90% AU threshold:
  - FalconFS: up to 80 GPUs
  - Lustre: up to 32 GPUs
  - CephFS: not satisfied.
- With 80 to 128 GPUs, FalconFS achieves training throughput speedups of **11.09-11.81×** over CephFS and **0.99–1.23×** over Lustre.

Evaluation – Application: Training Task

- 4 metadata server, 12 data servers
- Dataset: 10 million files, 1 million directories, filesize=112KB, 1064GB
- Workload: MLPerf Storage Benchmark, ResNet-50



- FalconFS's I/O overlaps with computation, thereby reducing the overall training runtime.

Discussion

- In practice, path-walk redirection is avoided due to an additional hop.
- After the training, data will be archived in external storage. Therefore, the scale of FalconFS is limited.
- Readdir needs to merge scan results from all metadata servers.

Conclusion

- Strengths
  - Provide a detailed analysis of DL training workload patterns and identifies inefficiencies in client-side caching.
  - Efficient architecture for lookup and load balancing

- Weakness
  - The architecture may be less efficient for general-purpose workloads.
  - The performance comparison is not apples-to-apples, making it difficult to determine whether the improvements stem from architectural design or implementation details.