USENIX
ATC '23

SingularFS: A Billion-Scale Distributed File System Using a Single Metadata Server

Hao Guo, Youyou Lu, Wenhao Lv, Xiaojian Liao, Shaoxun Zeng, Jiwu Shu

Tsinghua University

Outline

- ❖ Background & Motivation
- ❖ Design
- ❖ Evaluation
- ❖ Conclusion

Billion-Scale Distributed File Systems

- **Billion-scale distributed file systems dominate modern datacenters**
  - Cloud service vendors, small-scale clusters (within billion-scale)
  - Hyperscale clusters: Alibaba (billion-scale on average)

Billion-Scale Distributed File Systems

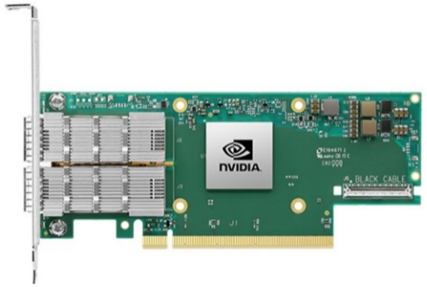
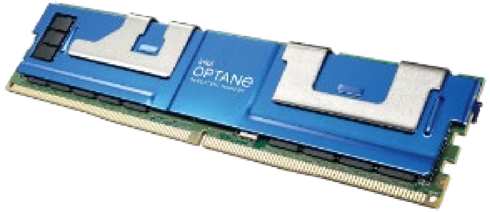
- **Billion-scale distributed file systems dominate modern datacenters**
  - Cloud service vendors, small-scale clusters (within billion-scale)
  - Hyperscale clusters: Alibaba (billion-scale on average)

- **Using a single metadata server is desirable and possible**
  - Easy implementation
  - TCO reduction
  - Capacity: 1TB / 256B (typical inode size) = 4.29 billions

- **But what about performance?**

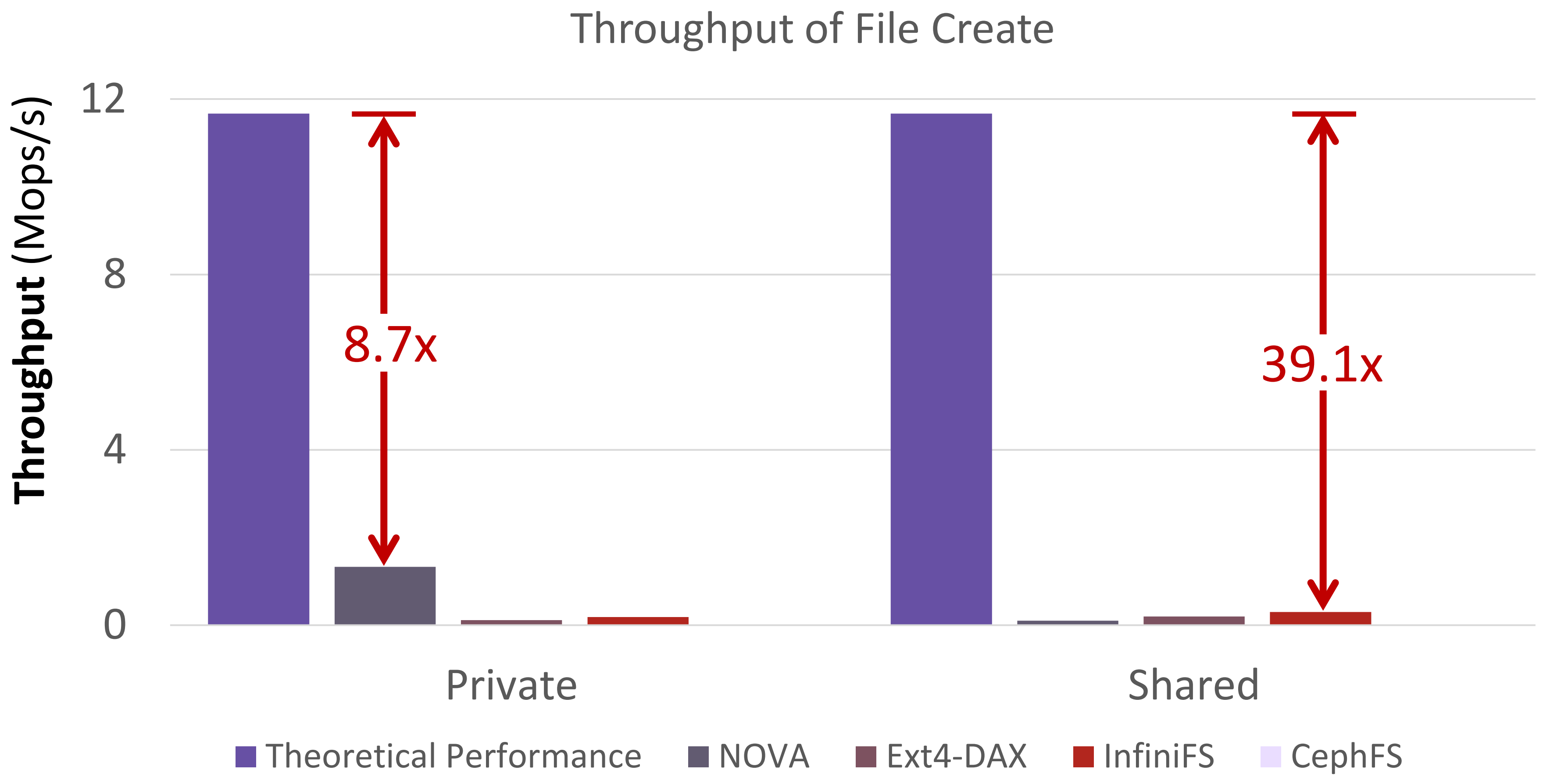
Performance Opportunities

- **New hardware provides performance opportunities for metadata**
  - Metadata is typically small (e.g., 256B for inode, 263B for directory entry)
  - New hardware shows high small-granularity IOPS

	New Hardware		Compared Hardware	
Network		RDMA NIC 112Mops/s (64B)	Ethernet NIC 1.48Mops/s (64B)	
Storage		Persistent Memory 29.1Mops/s (read) 8.75Mops/s (write)	NVMe TLC SSD 1.10Mops/s (read) 0.20Mops/s (write)	

Analysis of Existing Solutions

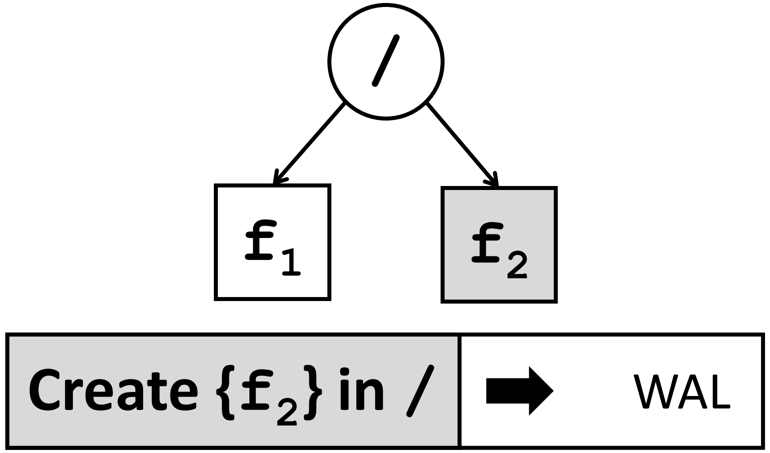
❖ **Huge gap between existing solutions and theoretical performance**



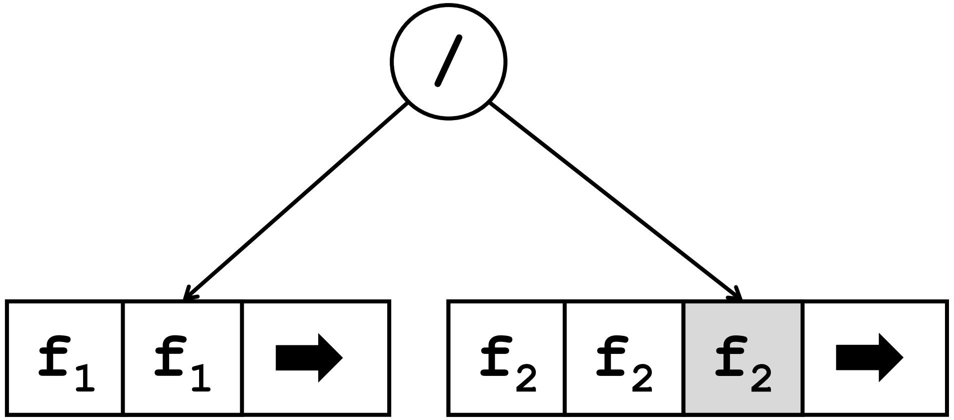
Theoretical performance: 3 PM writes (2 inodes, 1 dirent) + 1 network RPC.
Setup: 4 PM DIMMs, 1 RDMA NIC

Challenges

1. Crash consistency overhead



Write-ahead logging



Log-structured

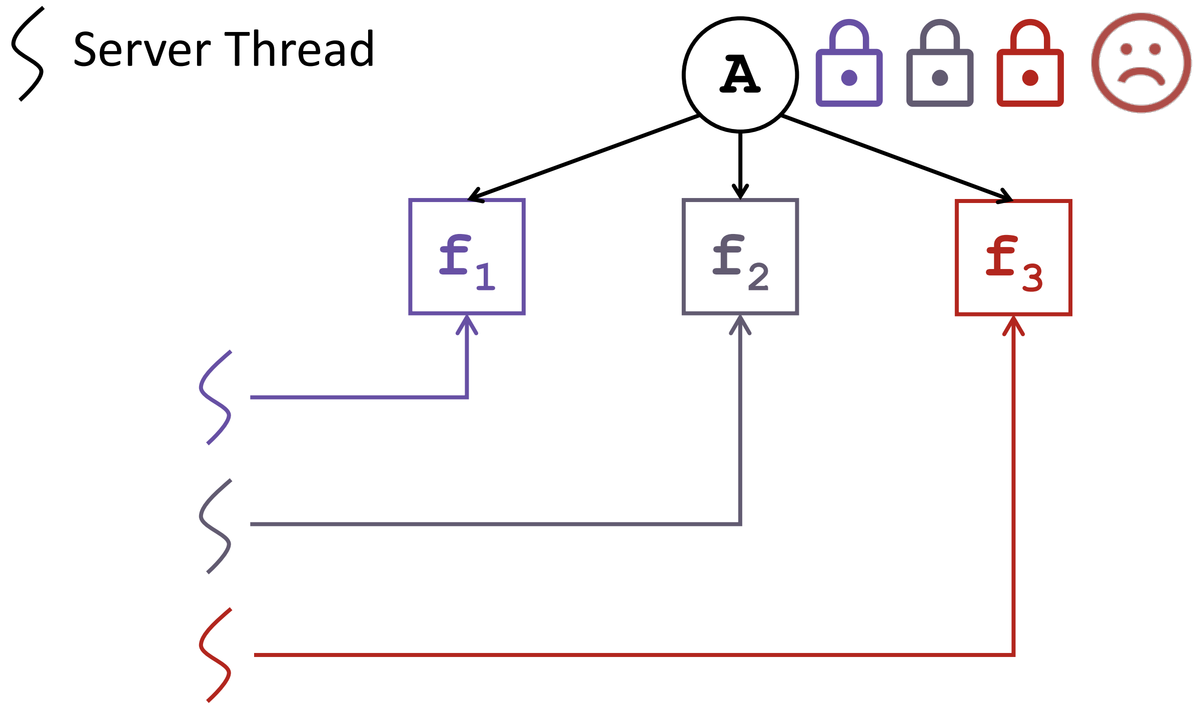
Double write
In-order checkpoint

Garbage collection
(GC) overhead

Challenges

2. Concurrency control in a shared directory

- ❖ High lock contention caused by concurrent update of shared parent's metadata

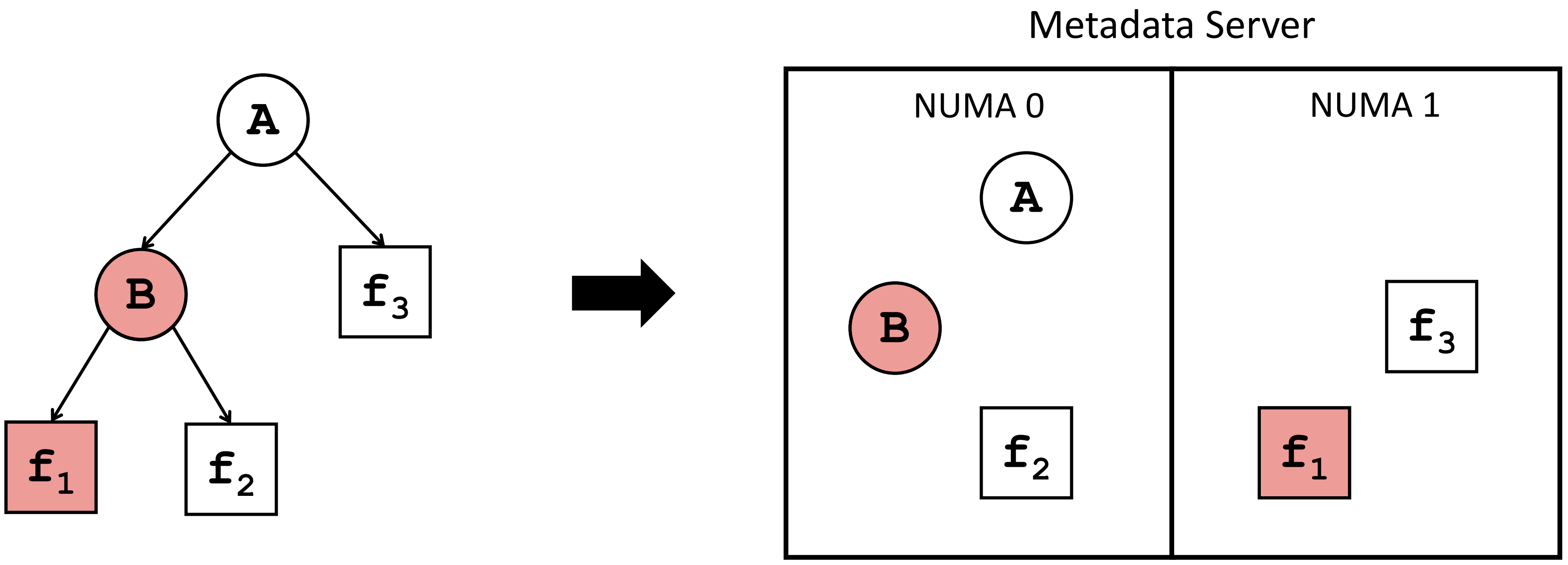


Concurrent file create in a shared directory

Challenges

3. NUMA scalability

- ❖ Existing solutions randomly scatter metadata to different NUMA nodes



NUMA locality can't be ensured for file create / delete

Outline

- ❖ Background & Motivation
- ❖ Design
- ❖ Evaluation
- ❖ Conclusion

SingularFS Architecture

A billion-scale distributed file system using a single metadata server

Optimizations

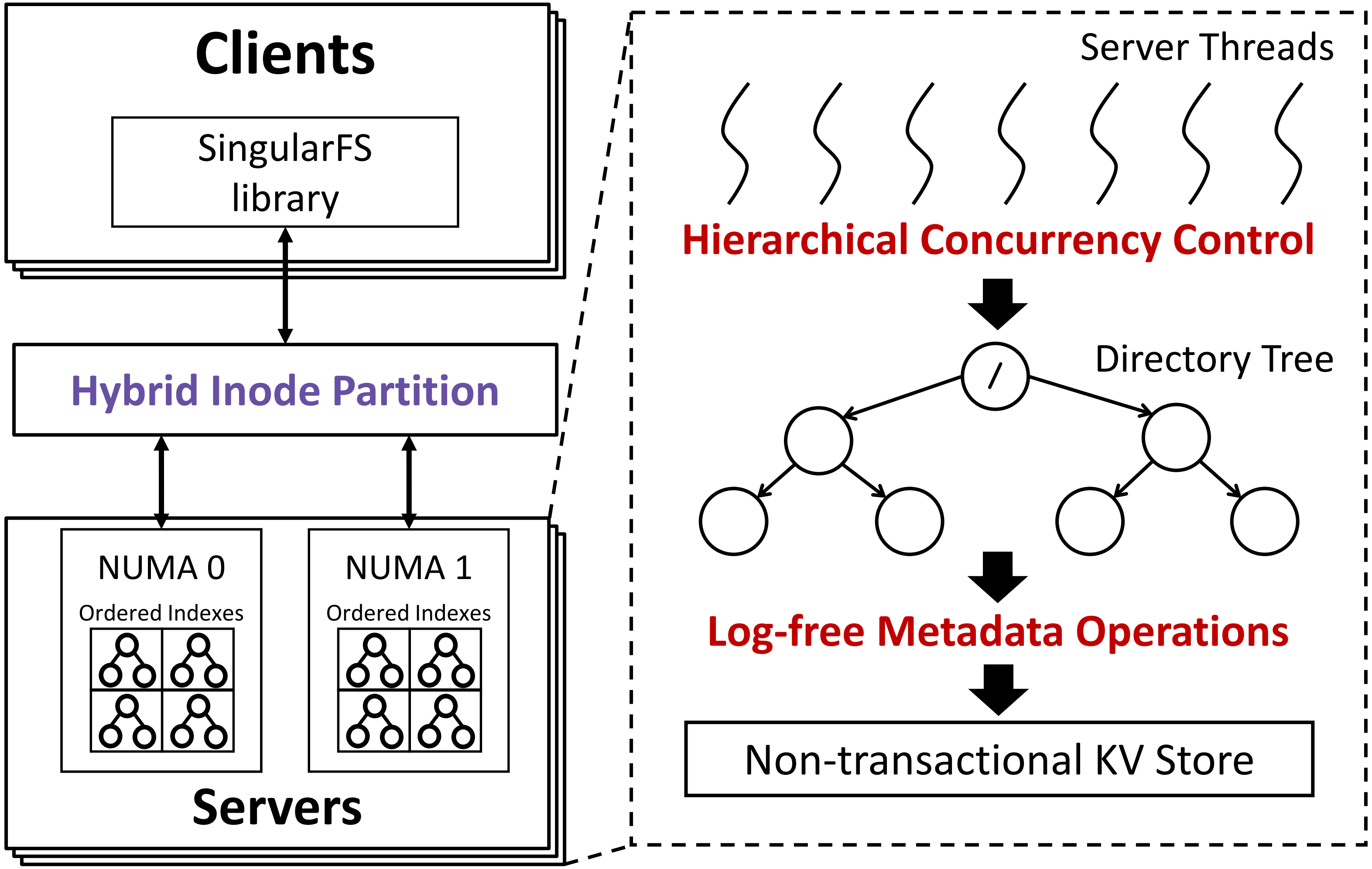
- ❖ Metadata Storage
- ❖ Metadata Operations

Metadata Storage

- ❖ Hybrid Inode Partition

Metadata Operations

- ❖ Hierarchical Concurrency Control
- ❖ Log-free Metadata Operations



Key Designs

- Crash consistency overhead

  1. Log-free Metadata Operations

- Concurrency control in a shared directory

  2. Hierarchical Concurrency Control

- NUMA scalability

  3. Hybrid Inode Partition

1. Log-free Metadata Operations

Crash consistency guarantee for different metadata write operations

Type	Operations	Modified Inodes		
		Target	Parent	Others
Single-Node	`open/close` `read/write/…`	●		
Double-Node	`mkdir/rmdir` `create/delete`	●	●	
Rename	`rename`	●	●	●

1. Log-free Metadata Operations

Crash consistency guarantee for different metadata write operations

Type	Operations	Modified Inodes		
		Target	Parent	Others
Single-Node	open/close read/write/...	●		
Double-Node	mkdir/rmdir create/delete	●	●	
Rename	rename			●

Non-transactional key-value (KV) operations without additional crash consistency cost

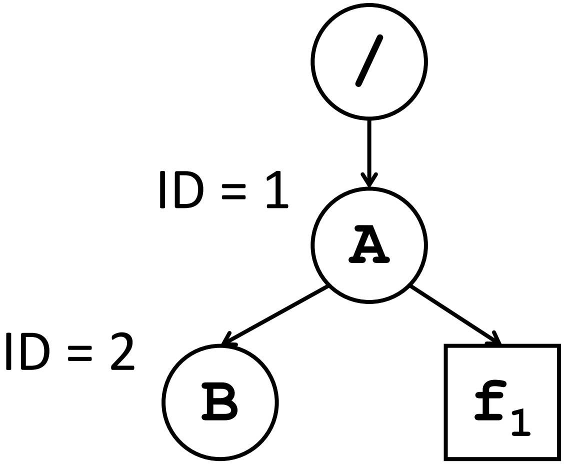
Rarely happens, use journaling

1. Log-free Metadata Operations

Step 1. Use KV Store to co-locate directory entries (dirents) and inodes

- KV pair: <parent_ID+name> → <inode>
- `ls` operation:
  - Prefix matching with key <parent_ID>
  - Extract the keys for name, values for ID and type

Directory tree



KV Store (partial)

Key	Value
1/B	B's access meta
$1/f_1$	f_1's inode
1	A's timestamps
2	B's timestamps

Directory entry in `ls`

- inode name
- inode ID
- inode type

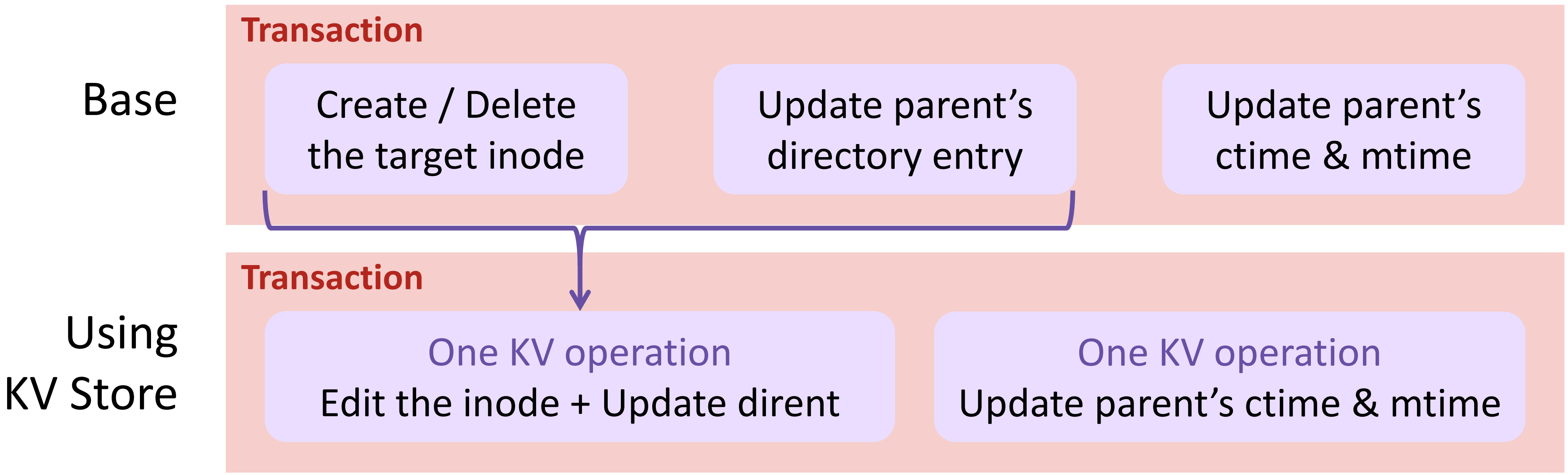
Note: access meta and timestamps will be discussed later in Hybrid Inode Partition.

1. Log-free Metadata Operations

What happens after Step 1?

- ❖ Single-Node operations: Crash consistency is guaranteed with KV Store
- ❖ Double-Node operations: Directory entries are embedded in KV pairs

Double-Node operations



Note: In POSIX semantics, ctime is the metadata change time, not the create time.

1. Log-free Metadata Operations

What happens after Step 1?

- Single-Node operations: Crash consistency is guaranteed with KV Store
- Double-Node operations: Directory entries are embedded in KV pairs

Double-Node operations

Transaction

Base

Create / Delete the target inode

Update parent's directory entry

Update parent's ctime & mtime

How to eliminate the transaction of the two KV operations?

Transaction

Using KV Store

One KV operation
Edit the inode + Update dirent

One KV operation
Update parent's ctime & mtime

Note: In POSIX semantics, ctime is the metadata change time, not the create time.

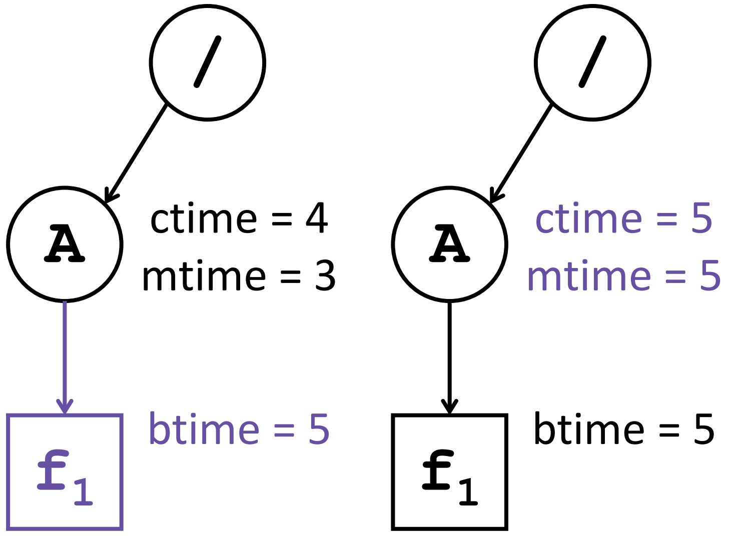
1. Log-free Metadata Operations

Transaction is needed for inserting inode and updating timestamps

Parent's ctime is not smaller than the born / death time of child inodes

Step 2. Ordered metadata update

- Insert the target inode with its born time (btime)
- Update the parent's ctime & mtime to the target inode's btime



Operation: create /A/f_1 at t = 5

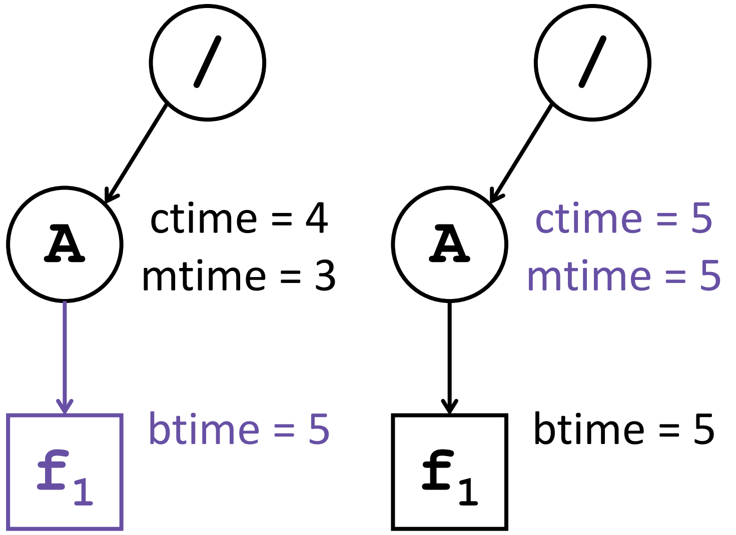
1. Log-free Metadata Operations

Transaction is needed for inserting inode and updating timestamps

Parent's ctime is not smaller than the born / death time of child inodes

Step 2. Ordered metadata update

- Insert the target inode with its born time (btime)
- System crashes
- Update parent's ctime & mtime with max(child inodes' btime)



Operation: create /A/f_1 at t = 5

1. Log-free Metadata Operations

What happens after Step 2?

- ❖ Single-Node operations: Crash consistency is guaranteed with KV Store
- ❖ Double-Node operations: Transactions are eliminated

Double-Node operations

Using KV Store

Transaction:
- One KV operation: Edit the inode + Update dirent
- One KV operation: Update parent's ctime & mtime

Log-free

One KV operation: Edit the inode + Update dirent → One KV operation: Update parent's ctime & mtime

1. Log-free Metadata Operations

What happens after Step 2?

- Single-Node operations: Crash consistency is guaranteed with KV Store
- Double-Node operations: Transactions are eliminated

Double-Node operations

Using KV Store

Transaction

Most metadata operations are transformed to non-transactional KV operations

Log-free

One KV operation
Edit the inode + Update dirent

→

One KV operation
Update parent’s ctime & mtime

2. Hierarchical Concurrency Control

Minimize the critical area of operations in a shared directory

Double-Node operations in a shared directory

Base

- **Concurrent**
  - Create / Delete the target inode
- **Serialized**
  - Update parent's directory entry
  - Update parent's ctime & mtime

Using KV Store

- **Conflicting transactions**
  - One KV operation: Edit the inode + Update dirent
  - One KV operation: Update parent's ctime & mtime

Log-free

- **Concurrent**
  - One KV operation: Edit the inode + Update dirent
- → **Serialized**
  - One KV operation: Update parent's ctime & mtime

2. Hierarchical Concurrency Control

Minimize the critical area of operations in a shared directory

Double-Node operations in a shared directory

Base

- **Concurrent**
  - Create / Delete the target inode
- **Serialized**
  - Update parent's directory entry
  - Update parent's ctime & mtime

Using KV Store

- **Conflicting transactions**
  - One KV operation: Edit the inode + Update dirent
  - One KV operation: Update parent's ctime & mtime

SingularFS

- **Concurrent**
  - One KV operation: Edit the inode + Update dirent
- → **Minimized critical area**
  - One 16B atomic operation: Update parent's ctime & mtime

Critical area of operations in a shared directory is minimized to one 16B atomic operation

2. Hierarchical Concurrency Control

Double-Node operations need the parent directory's write lock

Treat these ops specially as they only update the parent's timestamps

Operations related to an inode

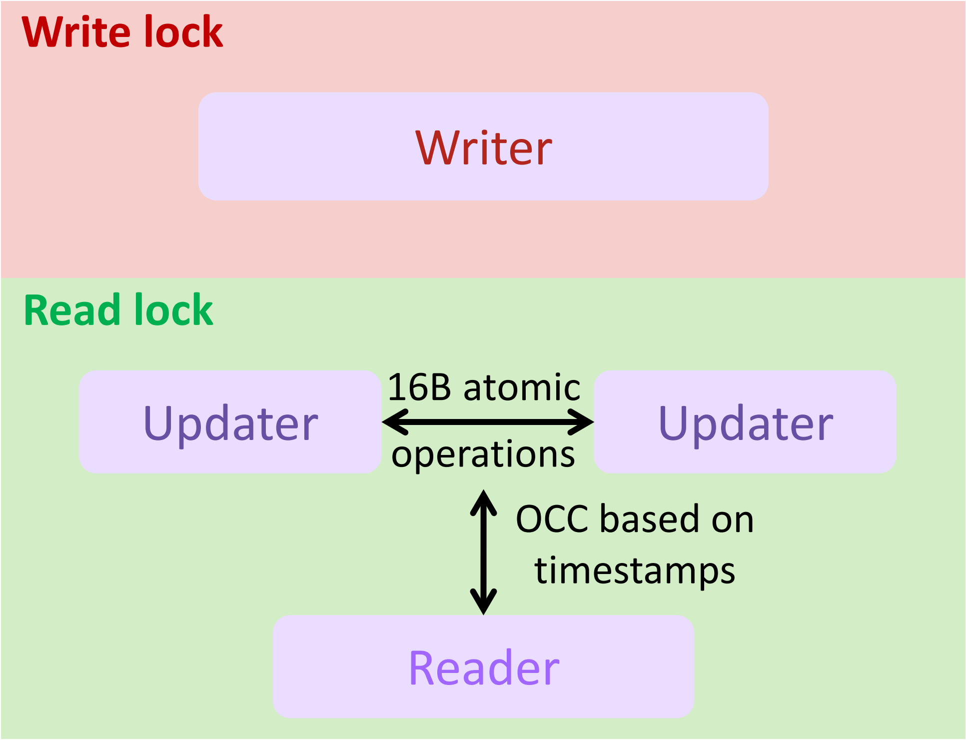
- Updater: timestamp update operations
- Writer: other update operations
- Reader: metadata read operations

1st layer: Writer with other ops

- Based on the target inode's rwlock

2nd layer: Updater with Reader

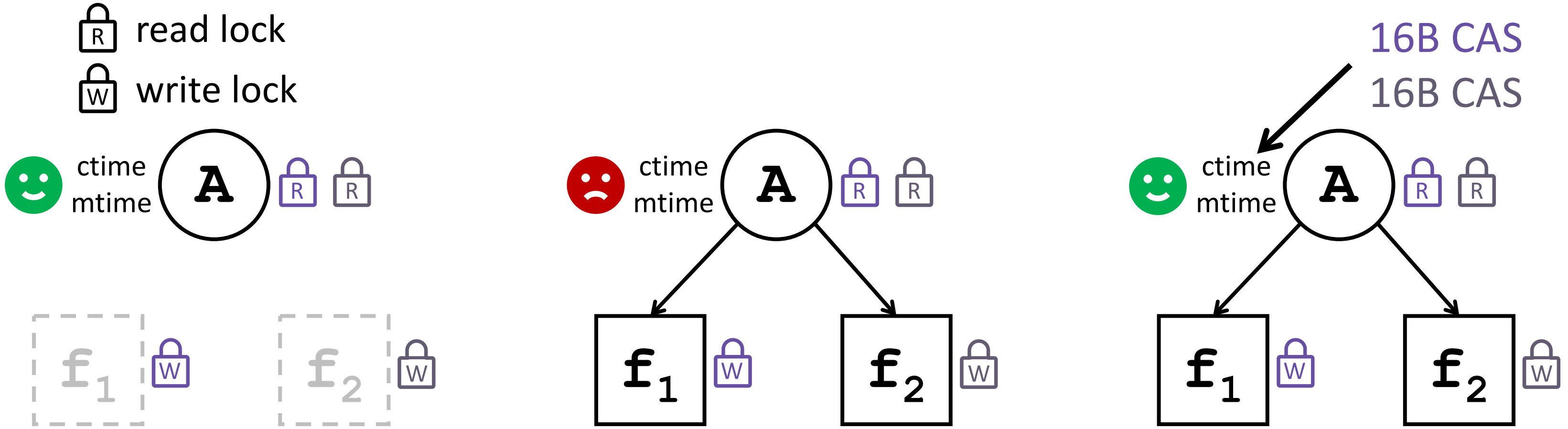
- Updater-Updater: 16B atomic operations
- Updater-Reader: OCC based on timestamps



2. Hierarchical Concurrency Control

Example 1. Concurrent file create in a shared directory

- ❖ Acquire the target inode's write lock (Writer of the target inode)
- ❖ Acquire the parent directory's read lock (Updater of the parent directory)
- ❖ Insert the metadata KV pairs concurrently
- ❖ Update the timestamps using 16B atomic CAS



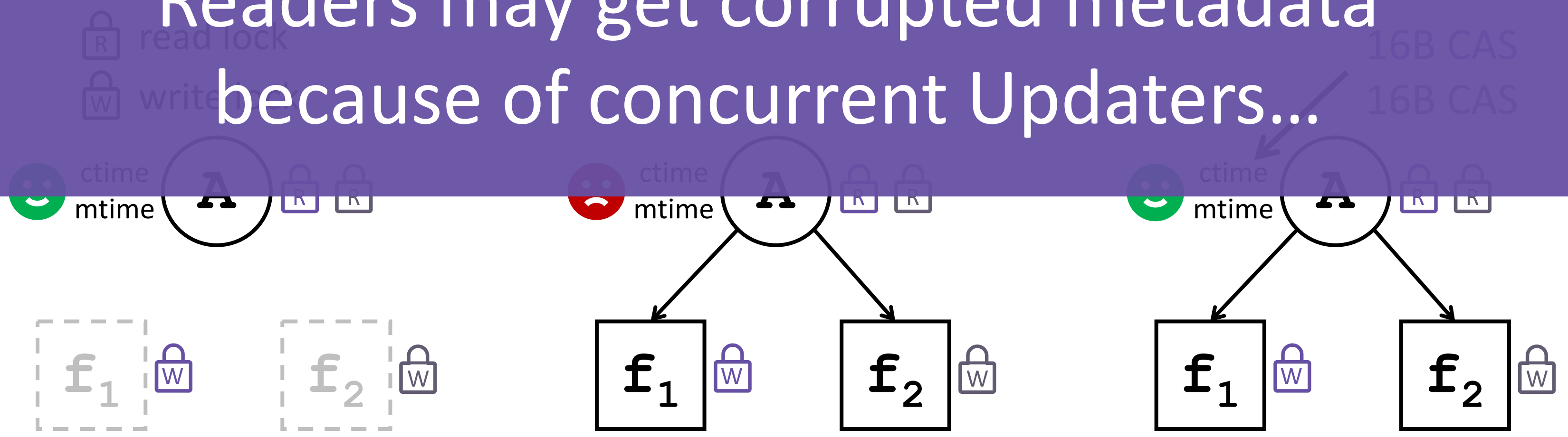
Operation: thread 1 create /A/f_1, thread 2 create /A/f_2 concurrently

2. Hierarchical Concurrency Control

Example 1. Concurrent file create in a shared directory

- ❖ Acquire the target inode's write lock (Writer of the target inode)
- ❖ Acquire the parent directory's read lock (Updater of the parent directory)
- ❖ Insert the metadata KV pairs concurrently
- ❖ Update the timestamps using 16B atomic CAS

Readers may get corrupted metadata because of concurrent Updaters…



Operation: thread 1 create /A/f_1, thread 2 create /A/f_2 concurrently

2. Hierarchical Concurrency Control

OCC needs a version number for ensuring data consistency

Inode's ctime has the same semantic as a version number

Example 2. Concurrent directory stat with other operations

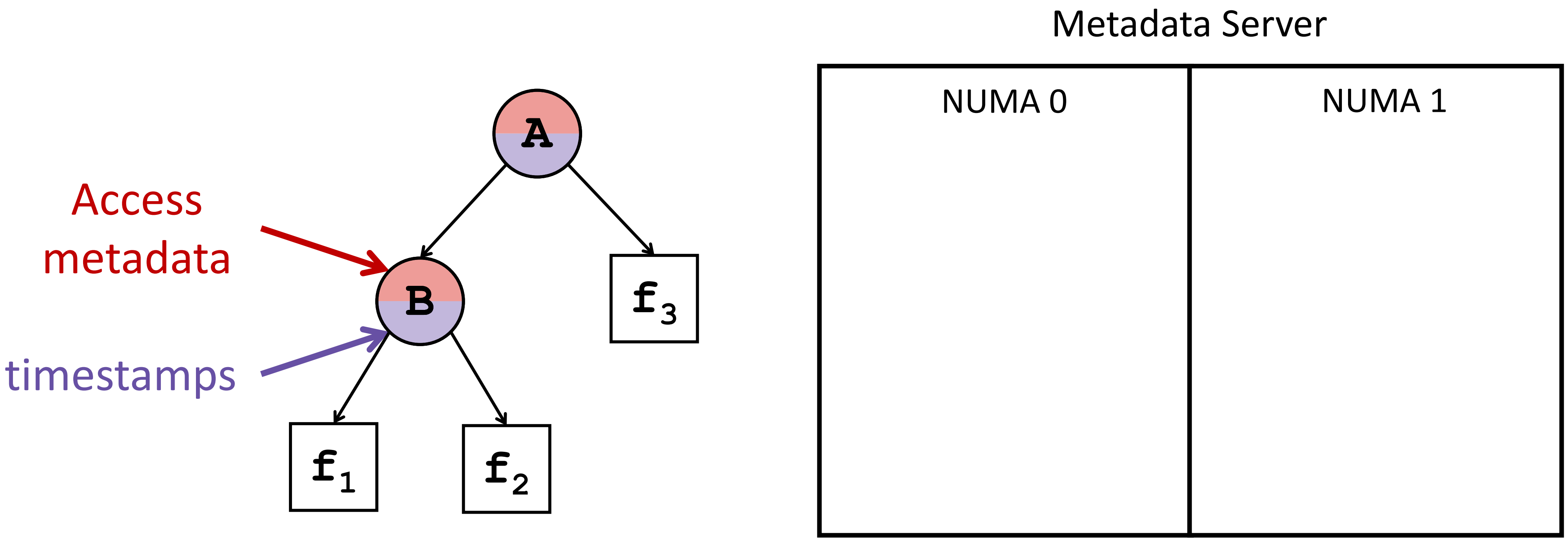
- ❖ Acquire the target inode's read lock (Reader of the target directory)
- ❖ OCC using the target inode's ctime as the version number

Operation: thread 1 stat directory A

3. Hybrid Inode Partition

NUMA-locality of Double-Node file operations can't be ensured

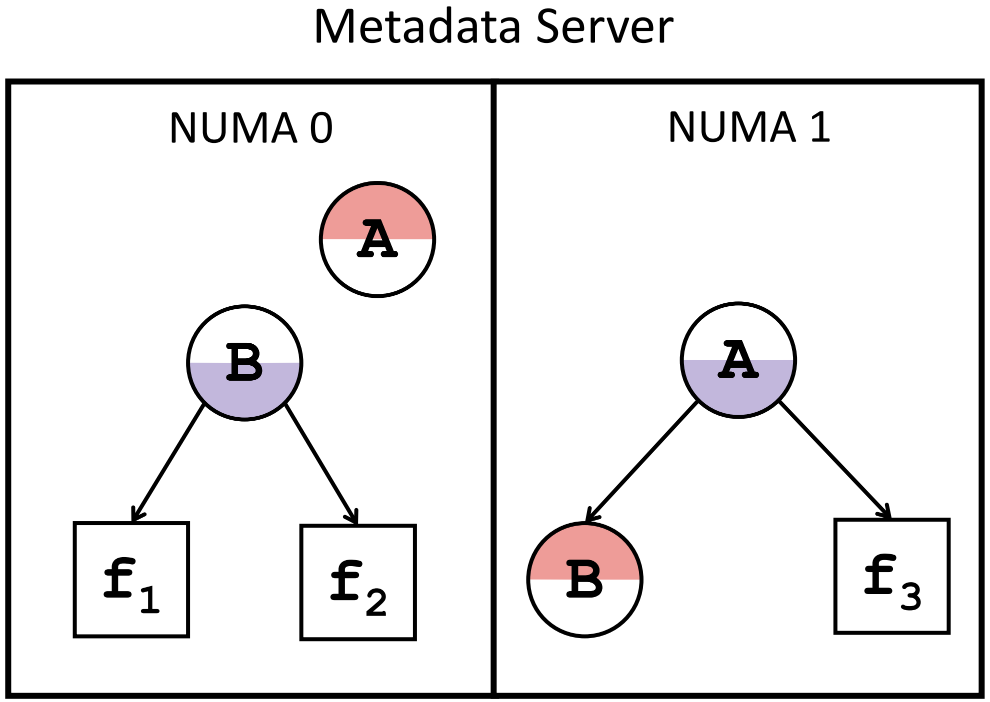
Group the involved metadata into the same NUMA node



3. Hybrid Inode Partition

NUMA-locality of Double-Node file operations can't be ensured

Group the involved metadata into the same NUMA node

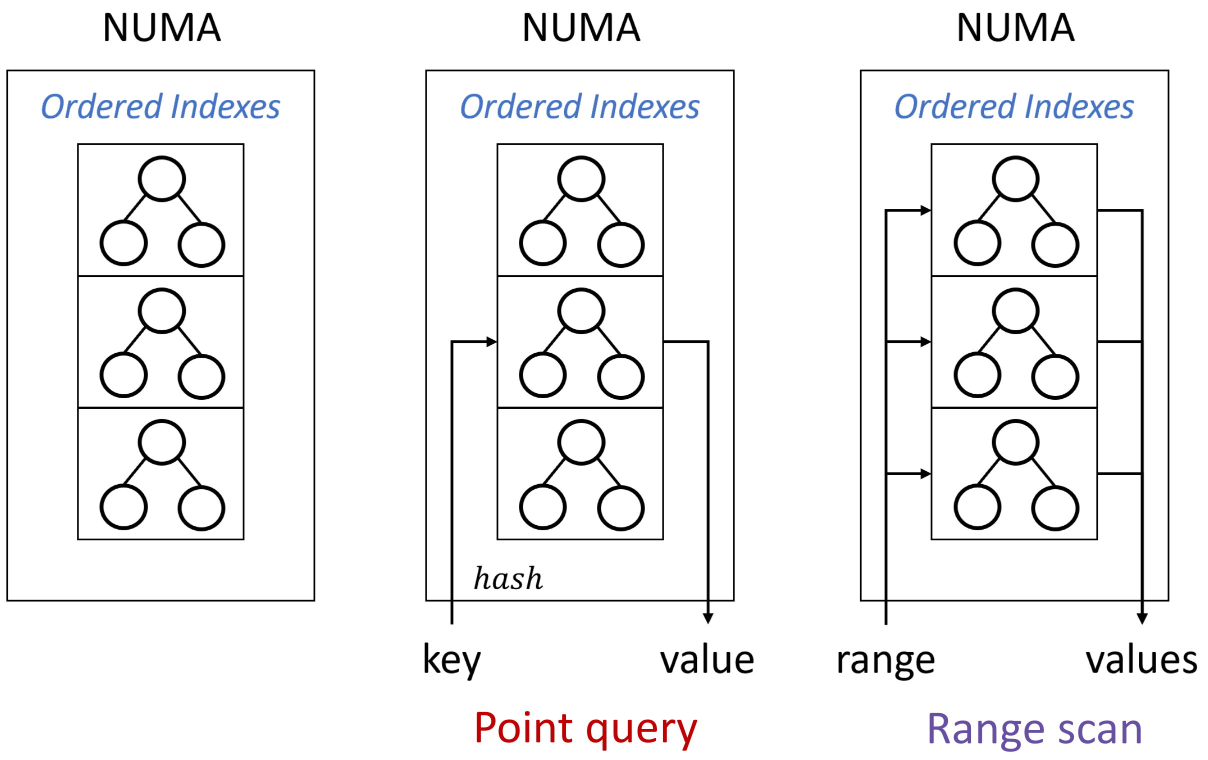


3. Hybrid Inode Partition

Lock contention inside the tree index limits metadata performance

Partition the intra-NUMA tree index into multiple sub-indexes

- Point query (common):
  - Hash the key to a sub-index
  - Directly get the result
- Range scan (in `ls`):
  - Scan all the indexes
  - Combine all the results



Outline

- ❖ Background & Motivation
- ❖ Design
- ❖ Evaluation
- ❖ Conclusion

Experimental Setup

Hardware Platform

- 1 server + 2 clients, 2 NUMA nodes per machine

CPU	Intel Xeon, 56 cores (server), 72 cores (client)
Memory	Samsung DDR4 3200MHz 32GB * 16
Storage	Intel Optane DCPMM Gen2 128GB * 8
Network	Mellanox ConnectX-6 200Gbps * 2

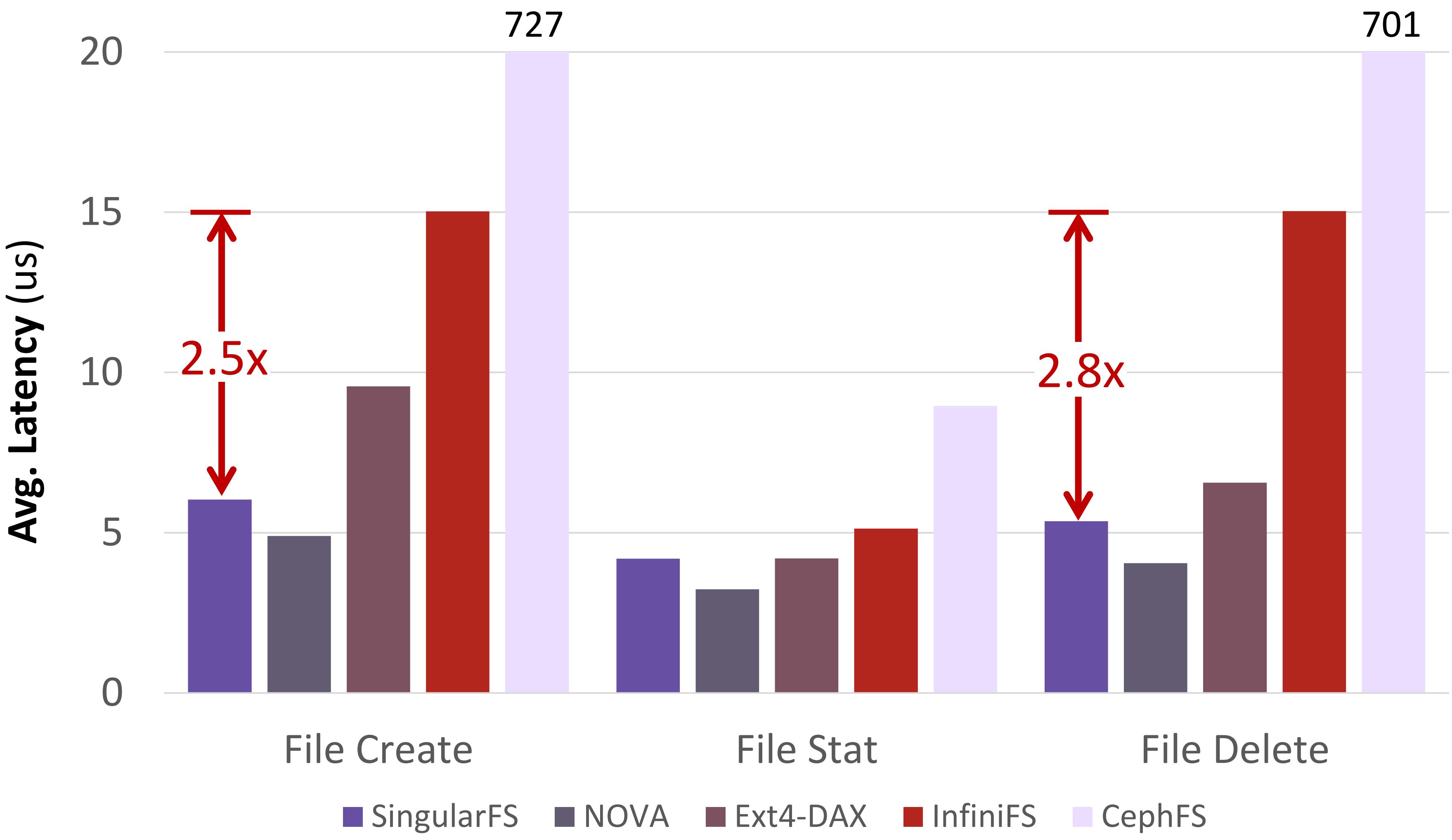
Compared Systems

- Local PM file systems: Ext4-DAX, NOVA [FAST '16]
- Distributed file systems: InfiniFS [FAST '22], CephFS [OSDI '06]

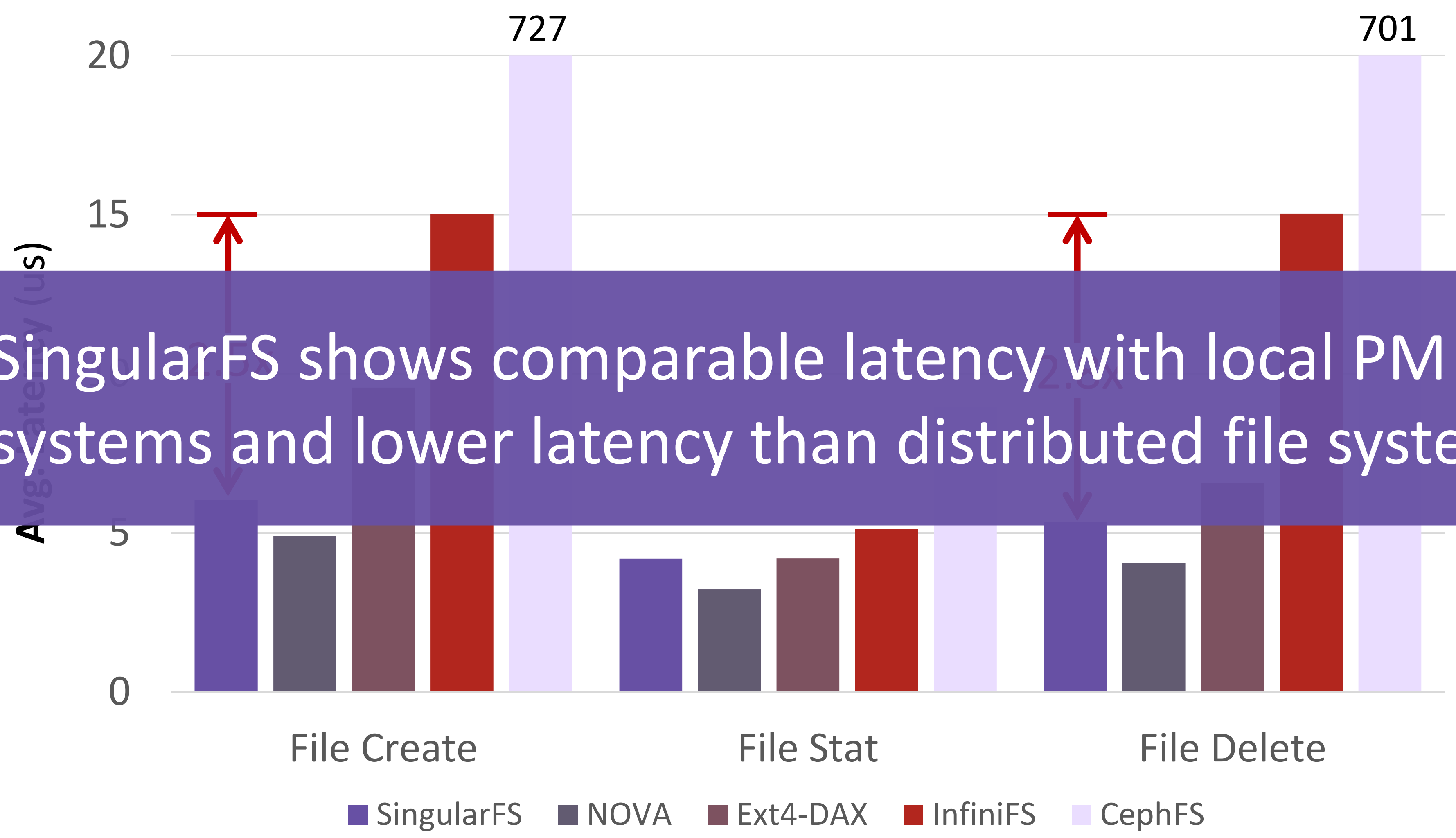
Benchmark

- Metadata performance: mdtest benchmark
- End-to-end performance: Filebench Fileserver & Varmail

Metadata Latency

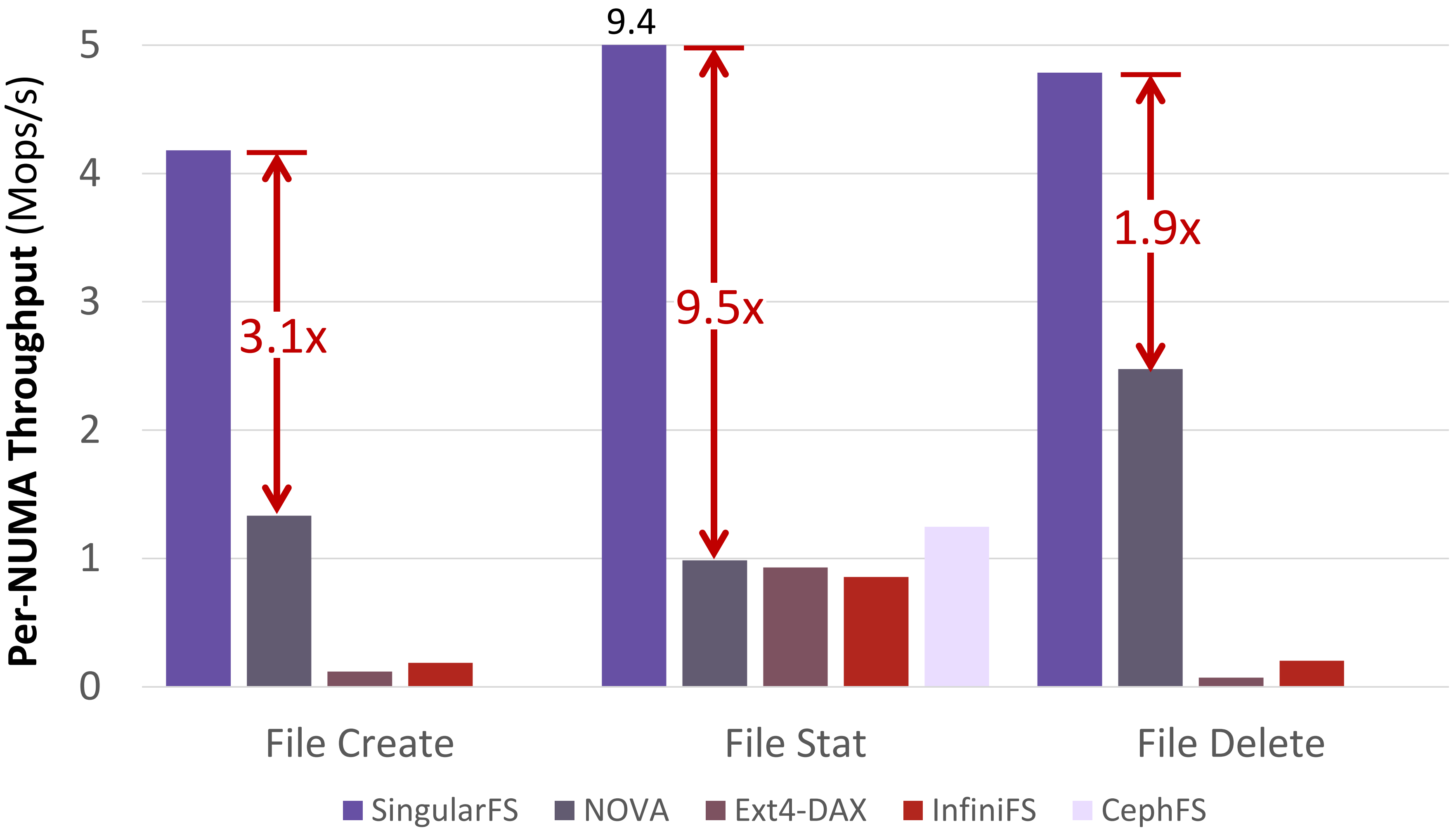


Metadata Latency



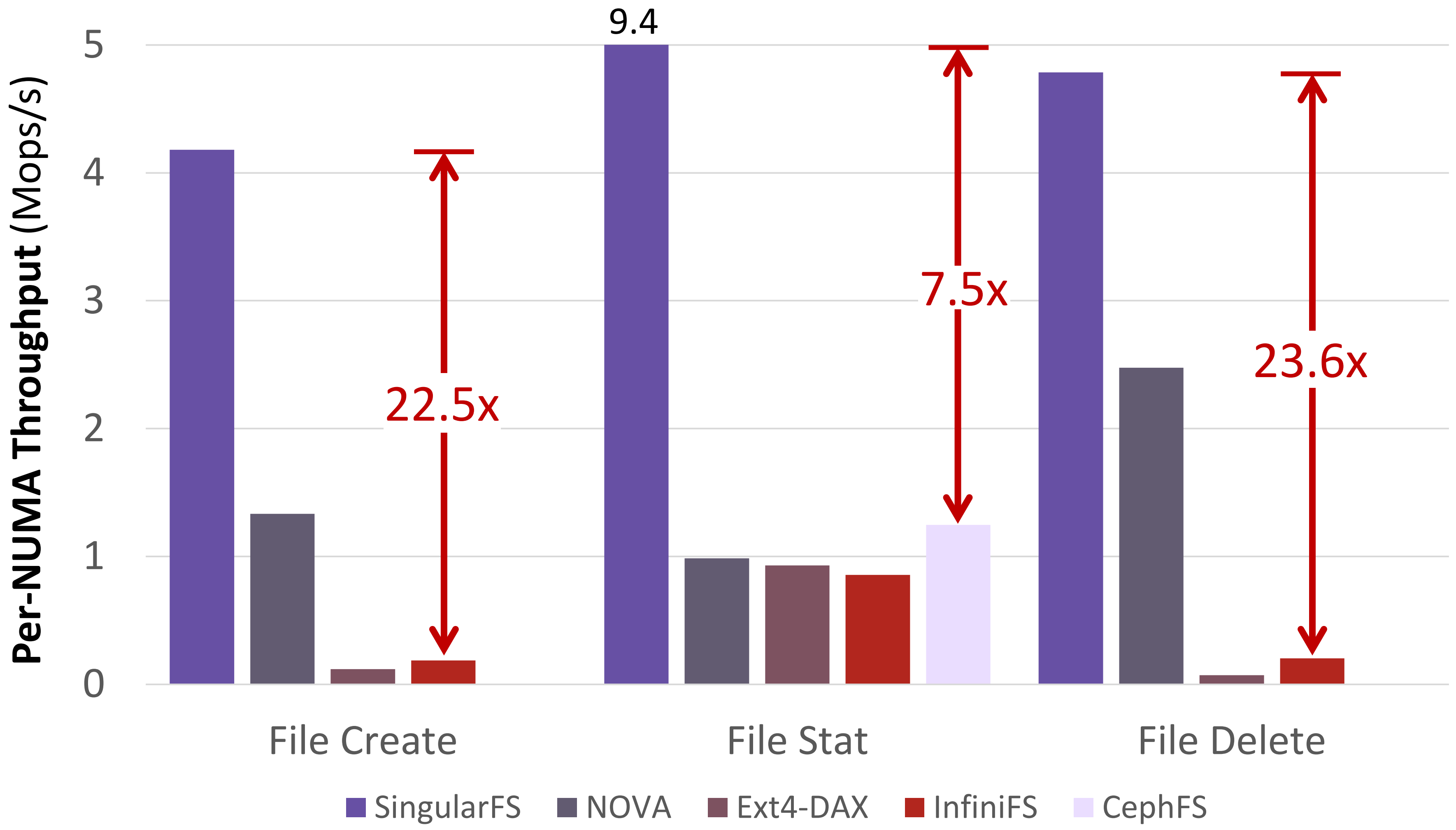
SingularFS shows comparable latency with local PM file systems and lower latency than distributed file systems

Metadata Throughput



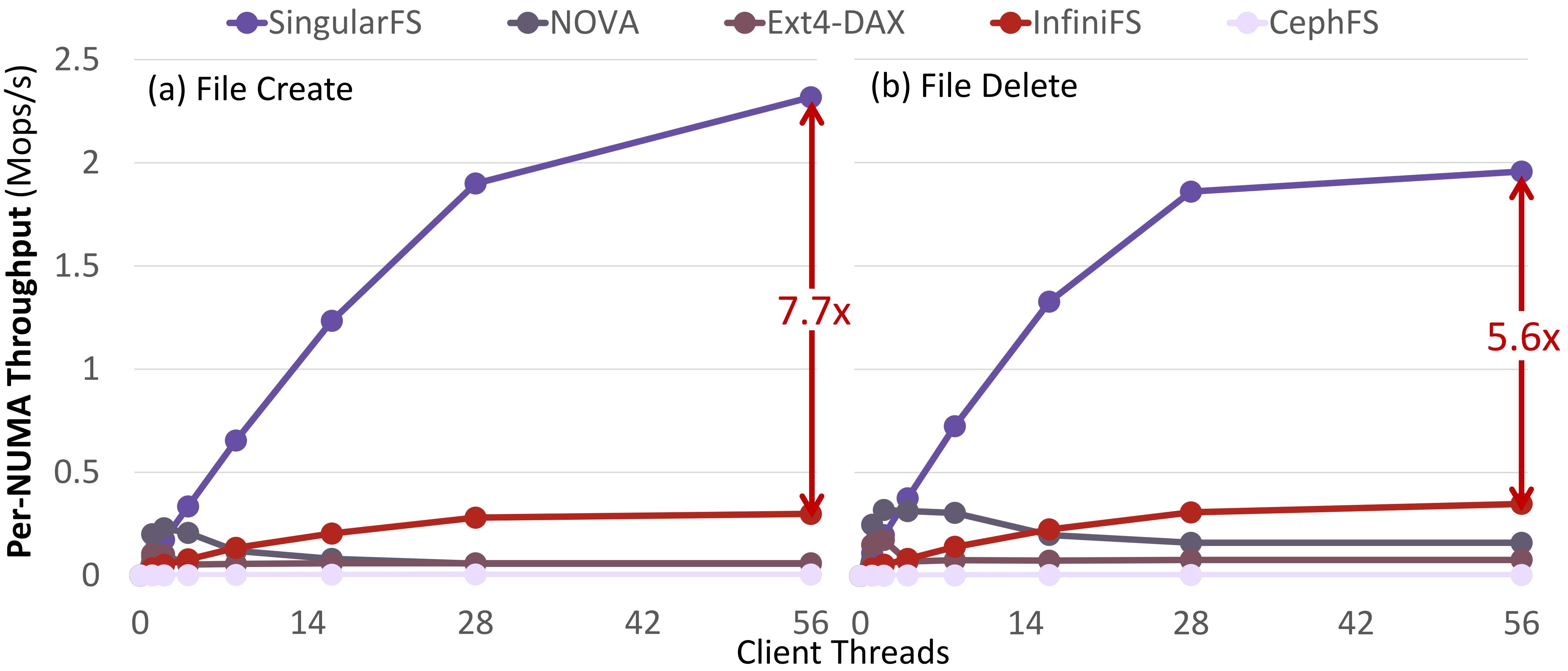
SingularFS has higher throughput than local PM file systems

Metadata Throughput



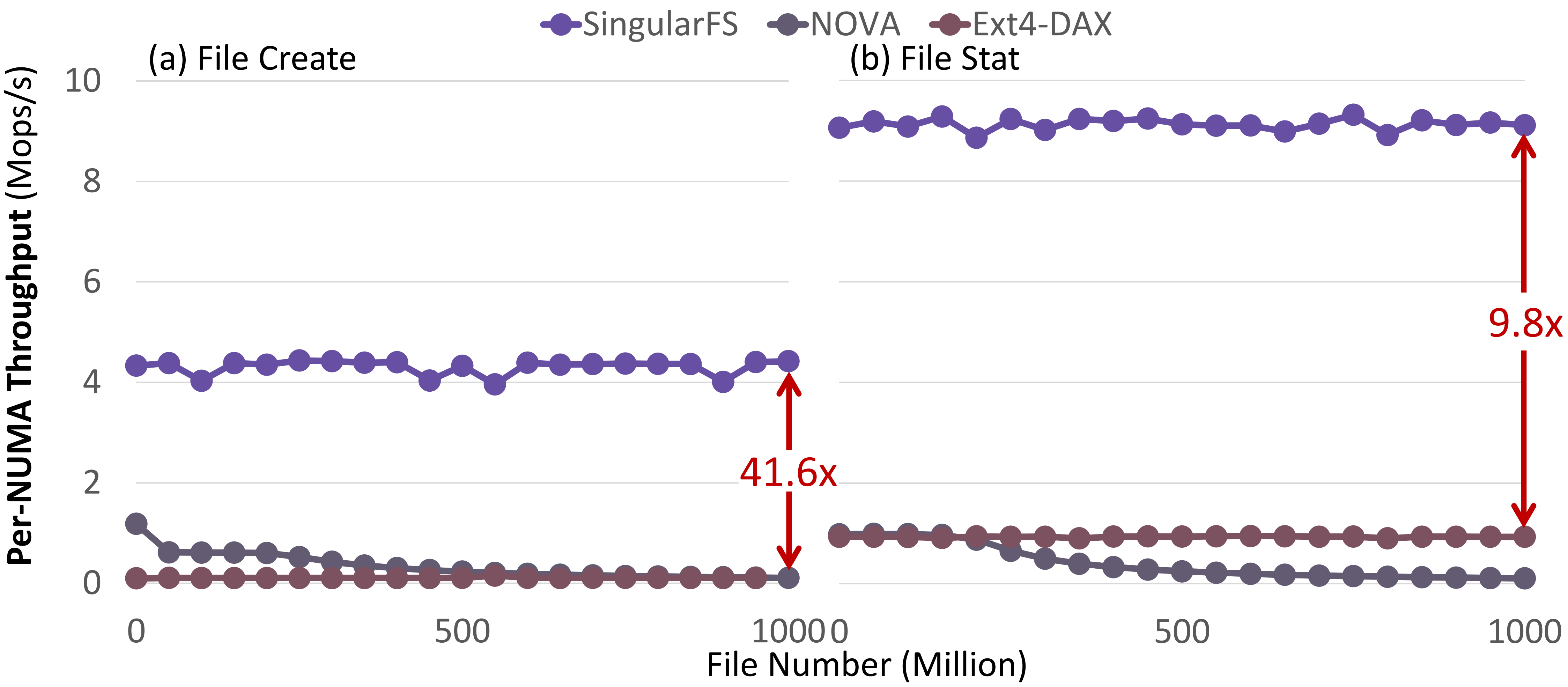
SingularFS has about an order of magnitude higher throughput than DFS

Operations in a Shared Directory



SingularFS shows high throughput in a shared directory

Billion-Scale Directory Tree



SingularFS efficiently supports billion-scale directory tree

Outline

- ❖ Background & Motivation
- ❖ Design
- ❖ Evaluation
- ❖ Conclusion

Conclusion

- Goal
  - Exploit the performance of a single metadata server to support billions of files
- Key Techniques of SingularFS
  - Log-free metadata operations
  - Hierarchical concurrency control
  - Hybrid inode partition
- Results
  - SingularFS shows comparable latency with local PM file systems
  - SingularFS has high throughput in both private and shared directories

Other Details

Design & Implementation

- Lazy recovery to reduce recovery overheads
- Log-free directory operations after introducing inode partition

Evaluation

- End-to-end benchmark
- Rename, crash recovery, billion-scale directory tree, …

Please check our paper for more details!

USENIX ATC '23

Thanks & Q/A

SingularFS: A Billion-Scale Distributed File System Using a Single Metadata Server

Hao Guo, Youyou Lu, Wenhao Lv, Xiaojian Liao, Shaoxun Zeng, Jiwu Shu

Contact: gh23@mails.tsinghua.edu.cn