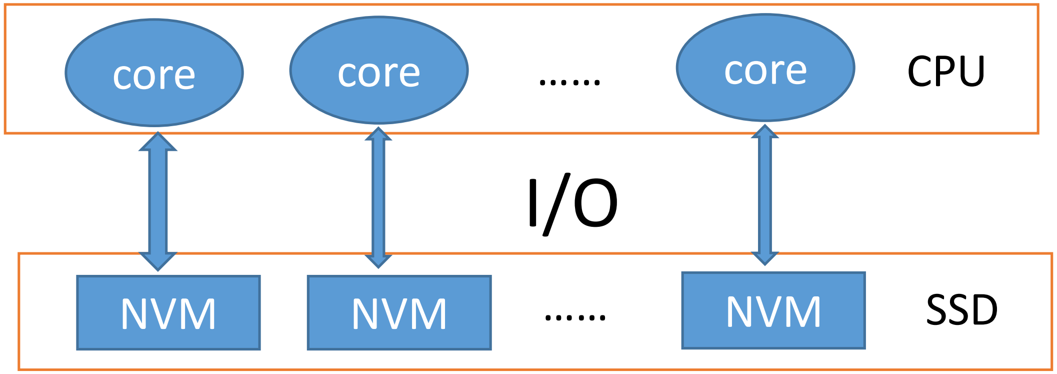
SpanFS:
A Scalable File System on Fast Storage Devices

Junbin Kang, Benlong Zhang, Tianyu Wo, Weiren Yu,
Lian Du, Shuai Ma and Jinpeng Huai

Beihang University

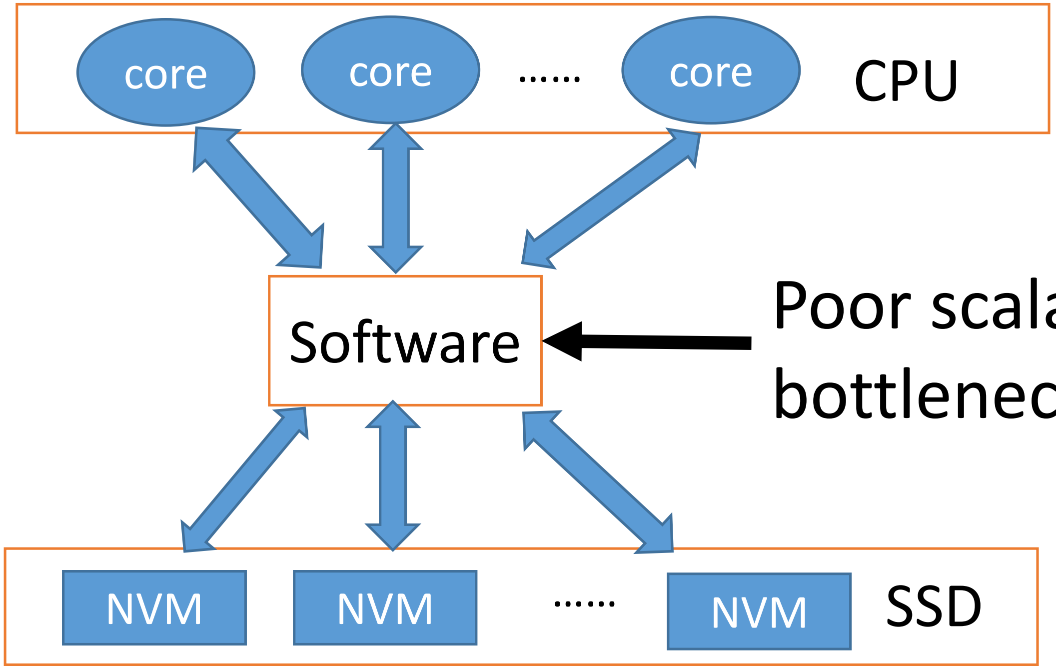
Advances of emerging hardware

- Multi-/many-core processors
  - High parallelism
- Flash-based or next-generation NVM-based SSDs
  - High parallelism
  - Low latency



The advanced hardware is expected to deliver high application-level I/O parallelism

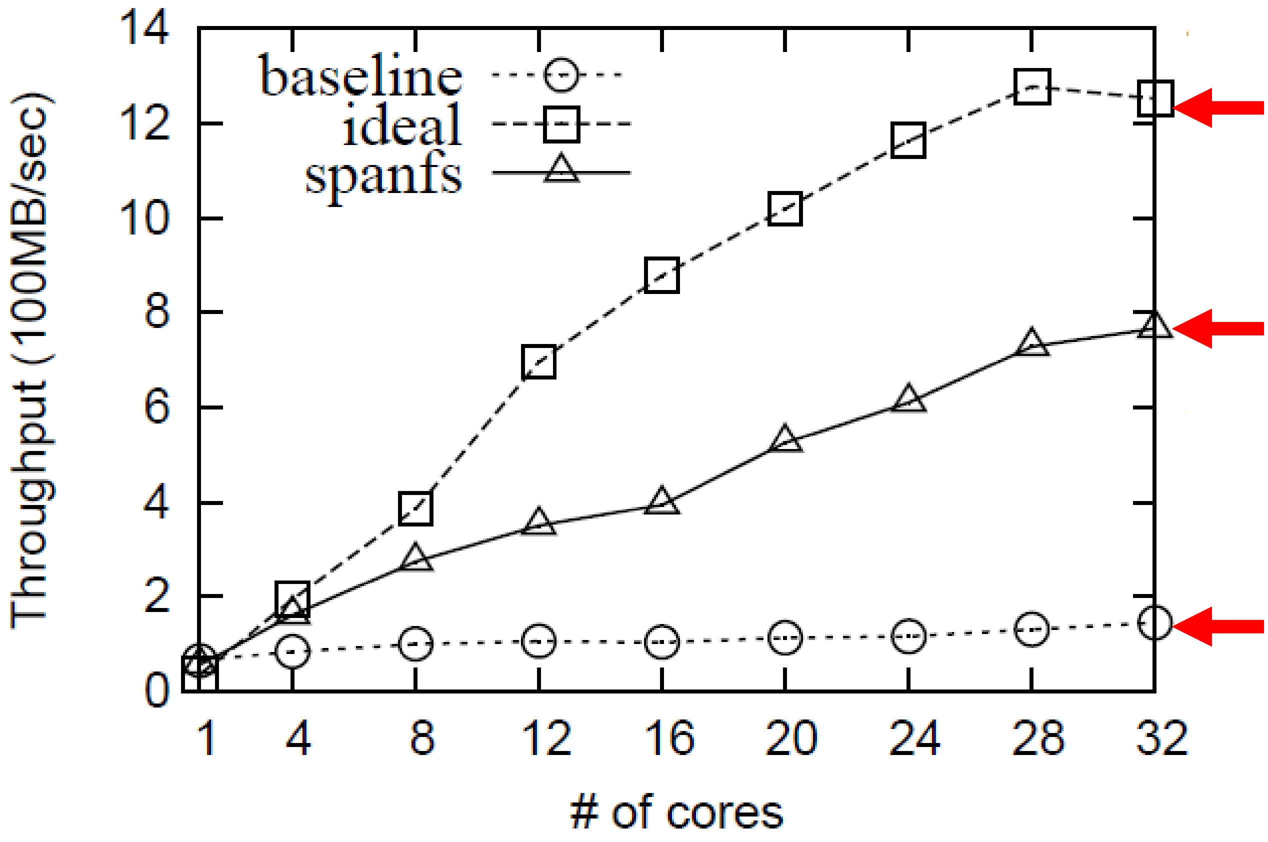
Software deficiency can be a bottleneck



Poor scalability can be a bottleneck

Scalability Evaluation

SysBench: 4KB synchronous writes to 128 files



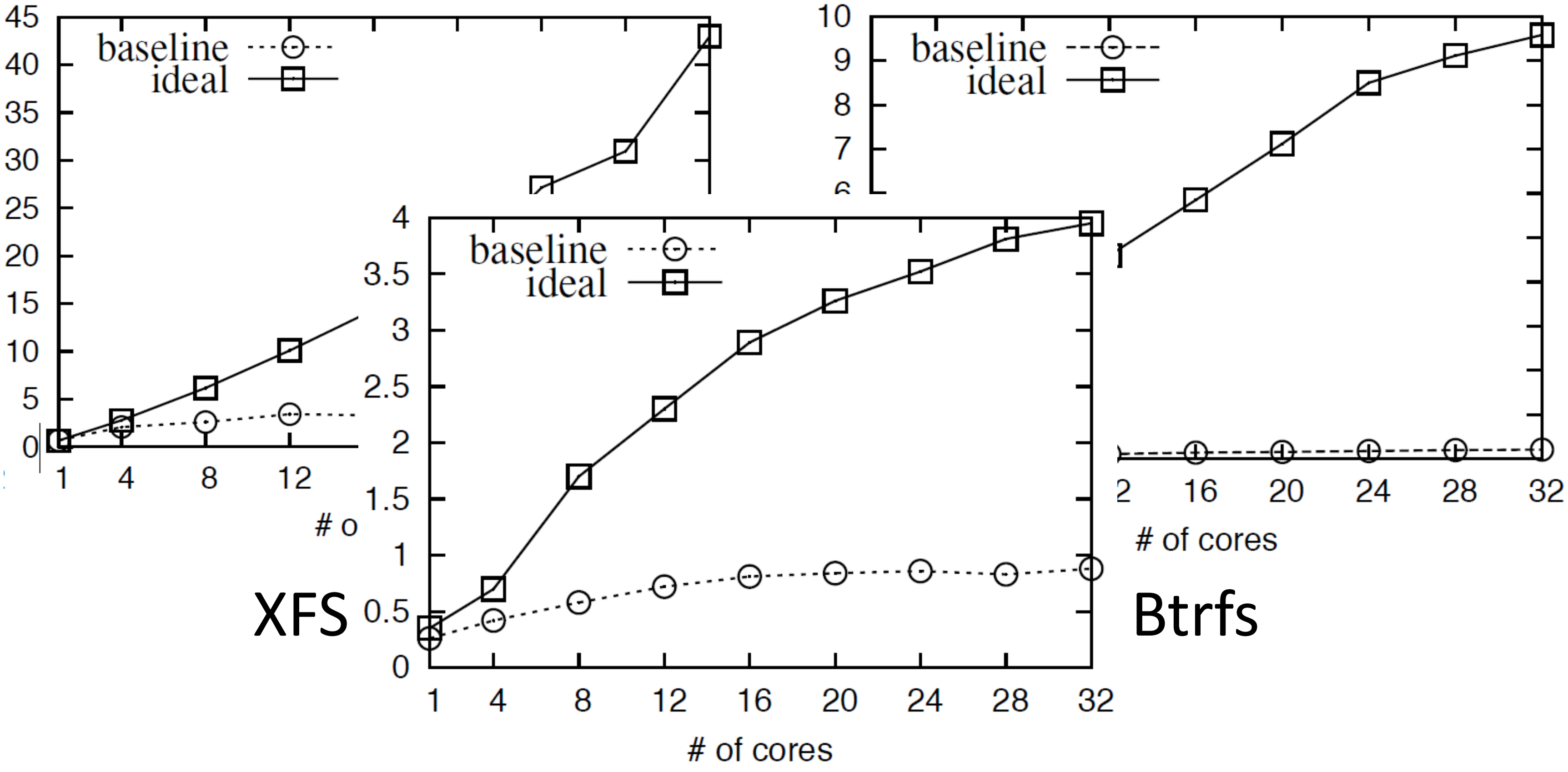
Ideal performance

SpanFS performance

Baseline performance

Ext4

Scalability Evaluation



XFS

Btrfs

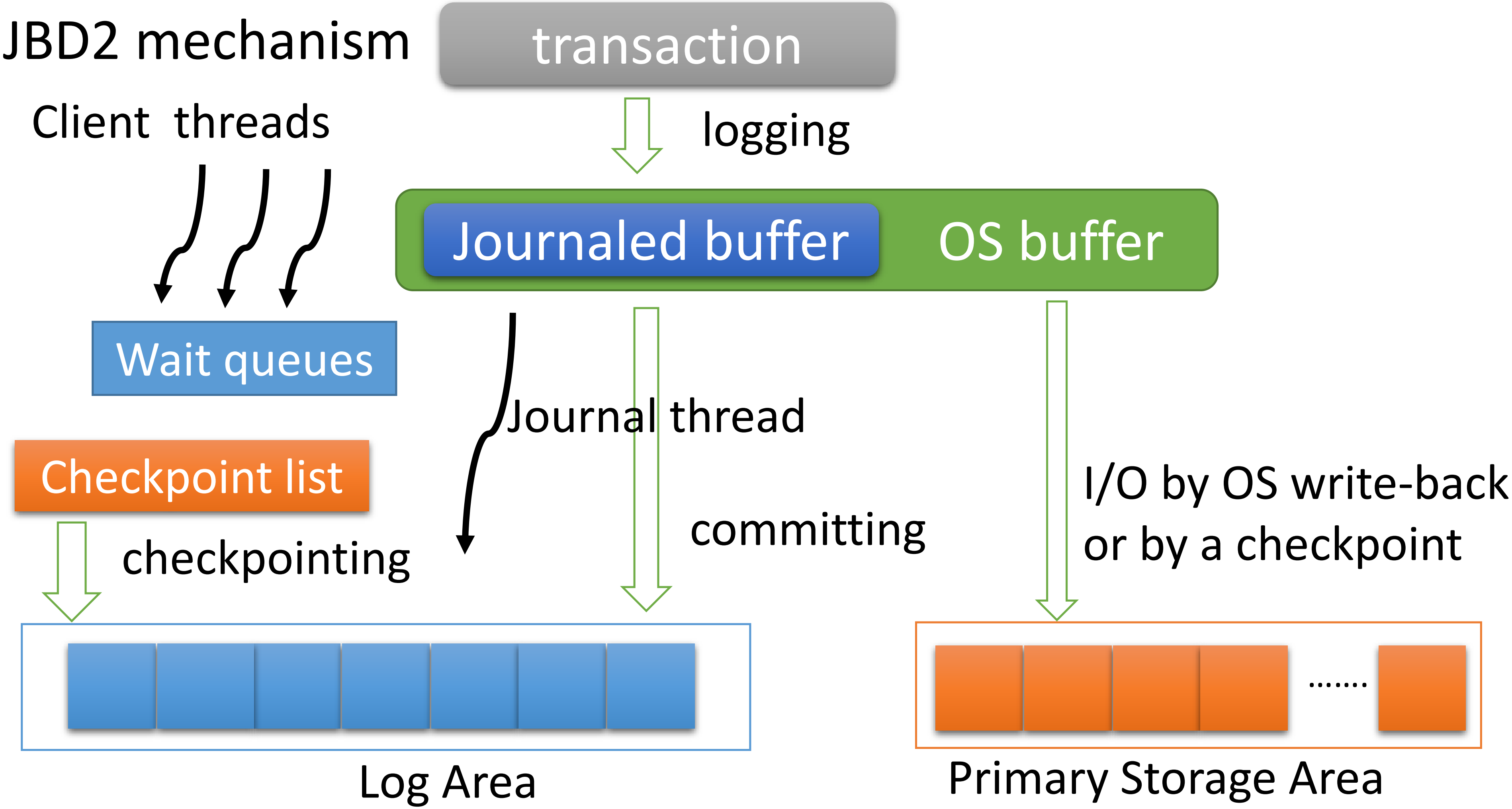
ZFS

Why file systems scale poorly ?

- We focus on the scalability issues of journaling file systems

- We take Ext4/JBD2 as an example for analysis

Why file systems scale poorly ?



JBD2 mechanism

transaction

logging

Client threads

Journaled buffer

OS buffer

Wait queues

Journal thread

Checkpoint list

checkpointing

committing

I/O by OS write-back
or by a checkpoint

Log Area

Primary Storage Area

Why file systems scale poorly ?

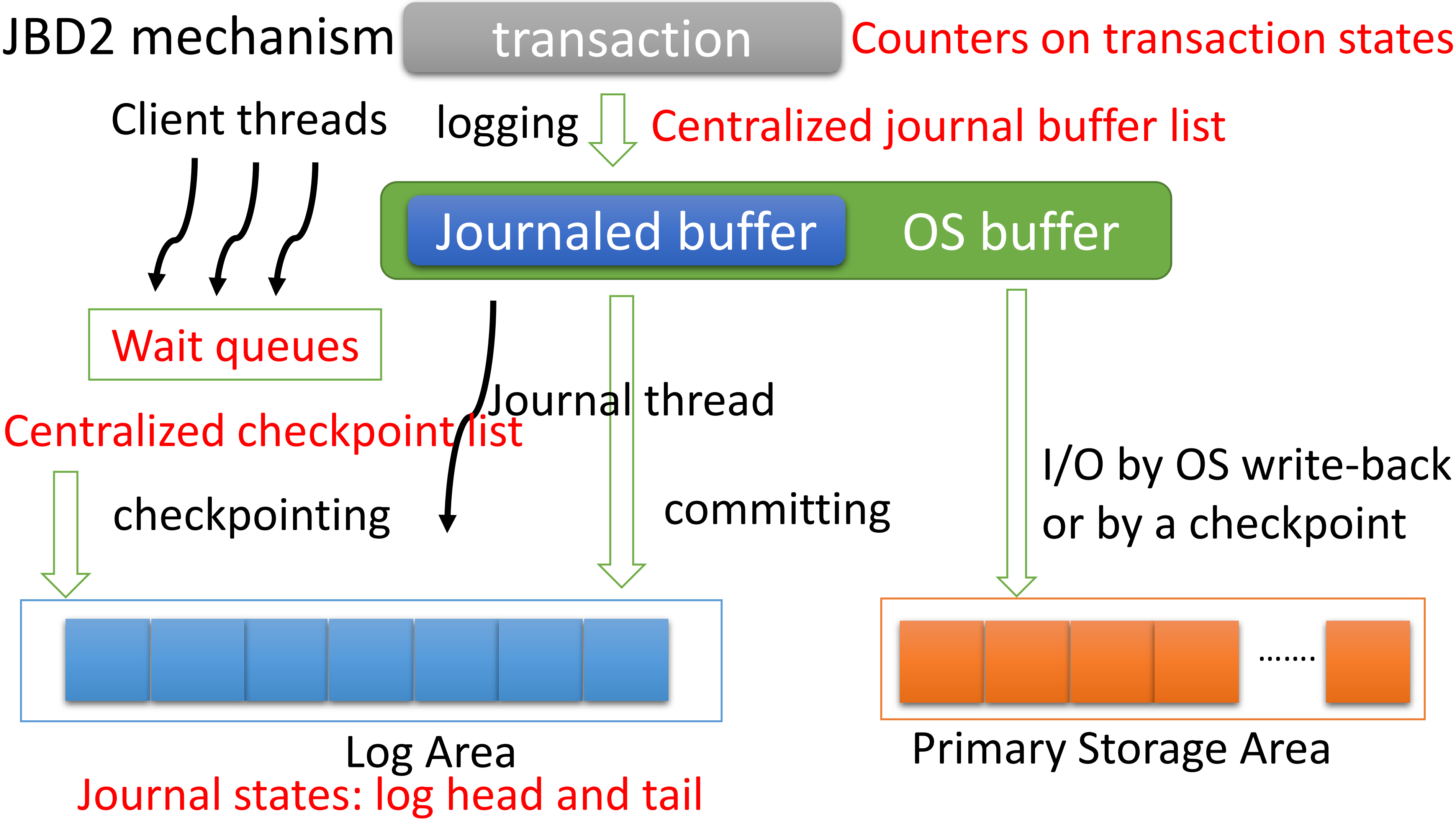
- Issue #1: serialization of journaling activities on devices
  - Sequential transaction commits
  - Sequential transaction checkpoints

- Journaling needs to ensure transaction order for correctness
  - Dependencies between transactions

Why file systems scale poorly ?

- Issue #2: unavoidable use of shared data structures

Why file systems scale poorly ?



JBD2 mechanism

Counters on transaction states

Centralized journal buffer list

Wait queues

Centralized checkpoint list

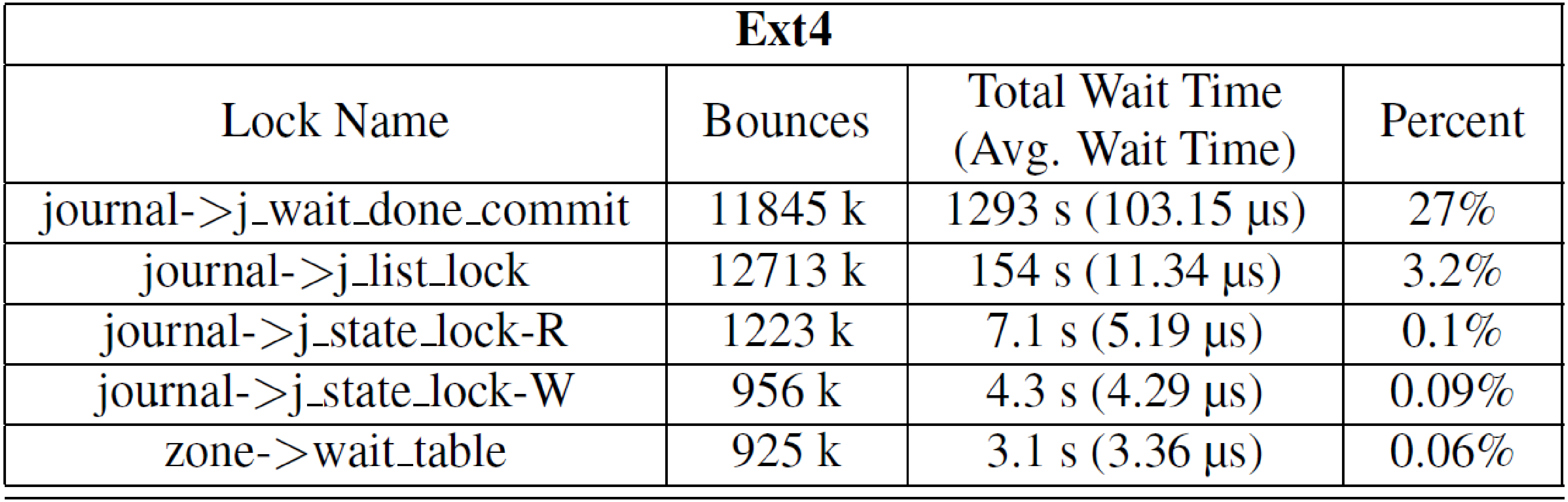
Journal states: log head and tail

Why file systems scale poorly ?

- Issue #2: unavoidable use of shared data structures

Shared data structures	Synchronization
Journaling states	j_state_lock (read-write lock)
Shared counters	Atomic operation
On-disk structures	bh state lock (bit-based spin lock)
Journaling buffer list	J_list_lock (spin lock)
Checkpoint transaction list	J_list_lock (spin lock)
Wait queues	J_wait_done_commit (spin lock)

Data profiling



Ext4			
Lock Name	Bounces	Total Wait Time (Avg. Wait Time)	Percent
journal->j_wait_done_commit	11845 k	1293 s (103.15 µs)	27%
journal->j_list_lock	12713 k	154 s (11.34 µs)	3.2%
journal->j_state_lock-R	1223 k	7.1 s (5.19 µs)	0.1%
journal->j_state_lock-W	956 k	4.3 s (4.29 µs)	0.09%
zone->wait_table	925 k	3.1 s (3.36 µs)	0.06%

Lock contention limits the file system scalability

Can they all be fixed using parallel programming techniques?

- Scalable read-write locks
  - E.g., RCU locks [McKenney ‘01] and Prwlocks [Liu ‘14]
  - They are scalable for read-mostly workloads
  - JBD2 has many writes to the shared states
- Per-core counters
  - E.g., sloppy counters [Boyd-Wickizer ‘10] and Refcache [Clements ‘13]
  - It is very expensive when reading the true values of these counters [Clements ‘13]

Can they all be fixed by using parallel programming techniques?

- Per-core data structures
  - Using Per-core lists may be effective for the journaling buffer lists
  - It is not suitable for the checkpoint transaction list
    - JBD2 needs to checkpoint the transactions in sequence for correctness
- Per-core wait queues [Liu '14]
  - It can be effective to solve the JBD2 wait queue bottleneck

Summary

- Using parallel programming techniques cannot fix all the bottlenecks

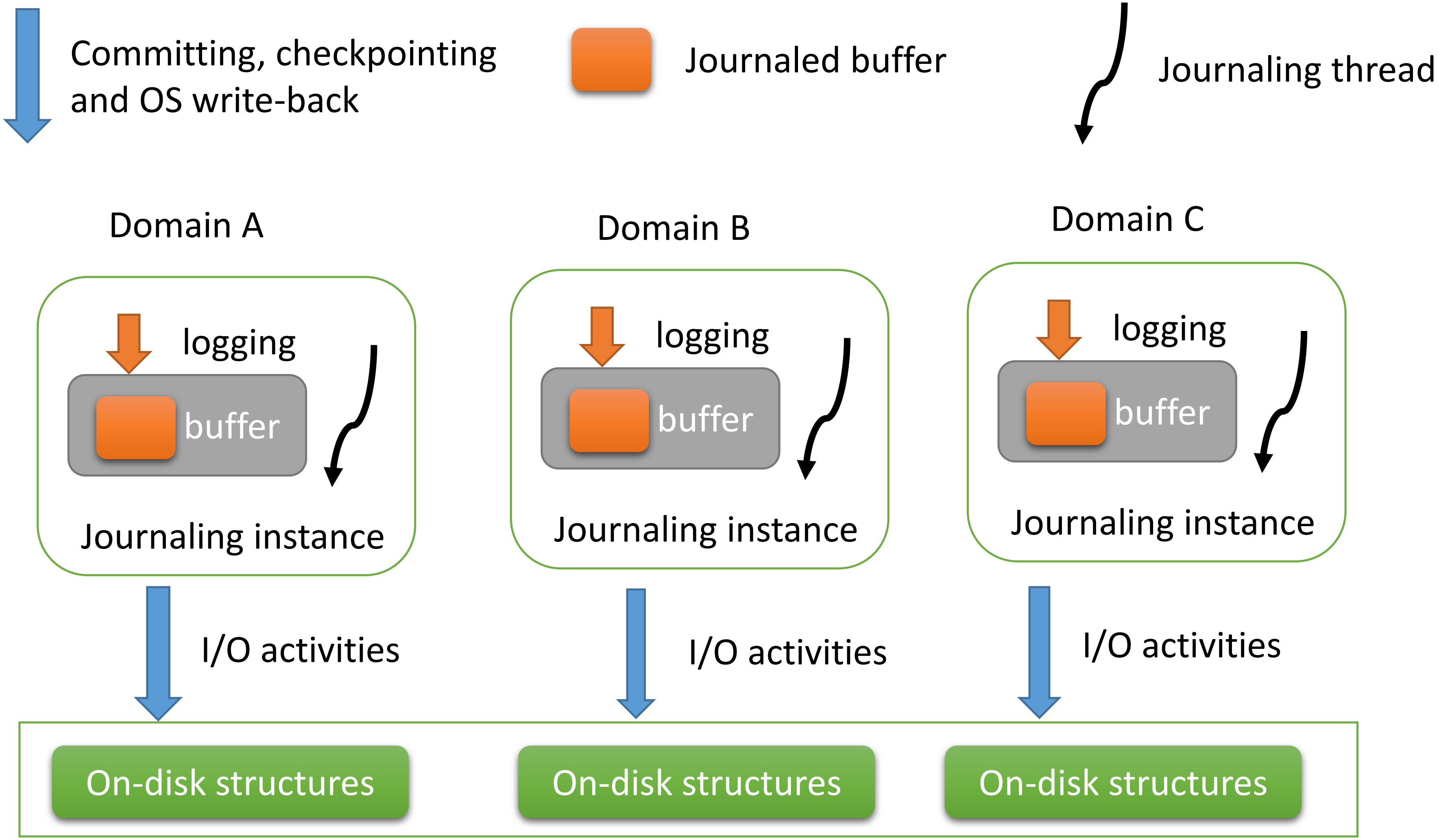
- The centralized journaling design
  - Issue #1: Serialization of I/O activities
  - Issue #2: The use of shared data structures

- **We need a new file system structure**

Our solution: SpanFS

- Replace the centralized file system service with multiple micro file system services called *domains*
    - Provide parallel file system services

Parallel file system services



Committing, checkpointing and OS write-back

Journaled buffer

Journaling thread

Domain A

logging

buffer

Journaling instance

I/O activities

Domain B

logging

buffer

Journaling instance

I/O activities

Domain C

logging

buffer

Journaling instance

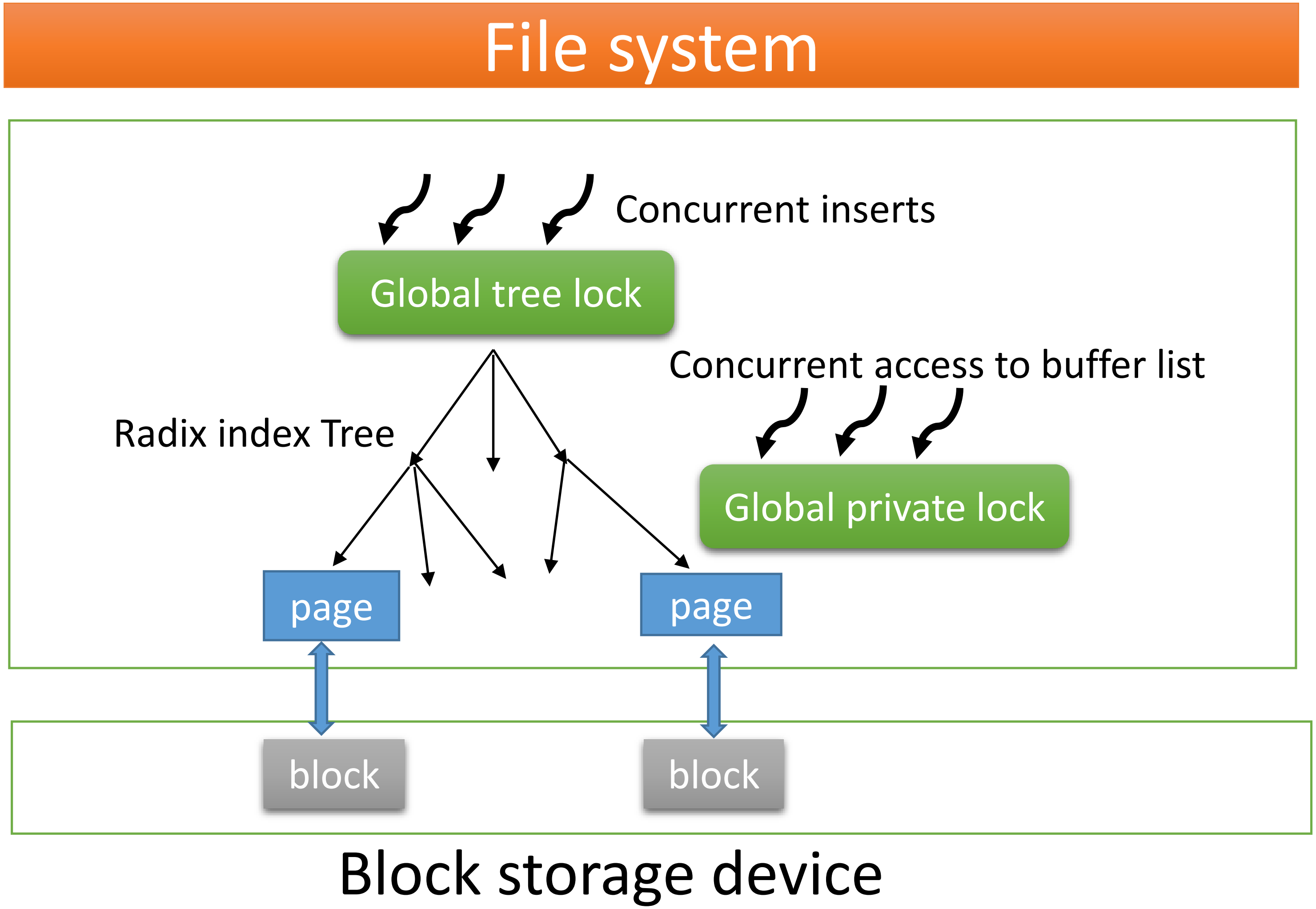
I/O activities

On-disk structures

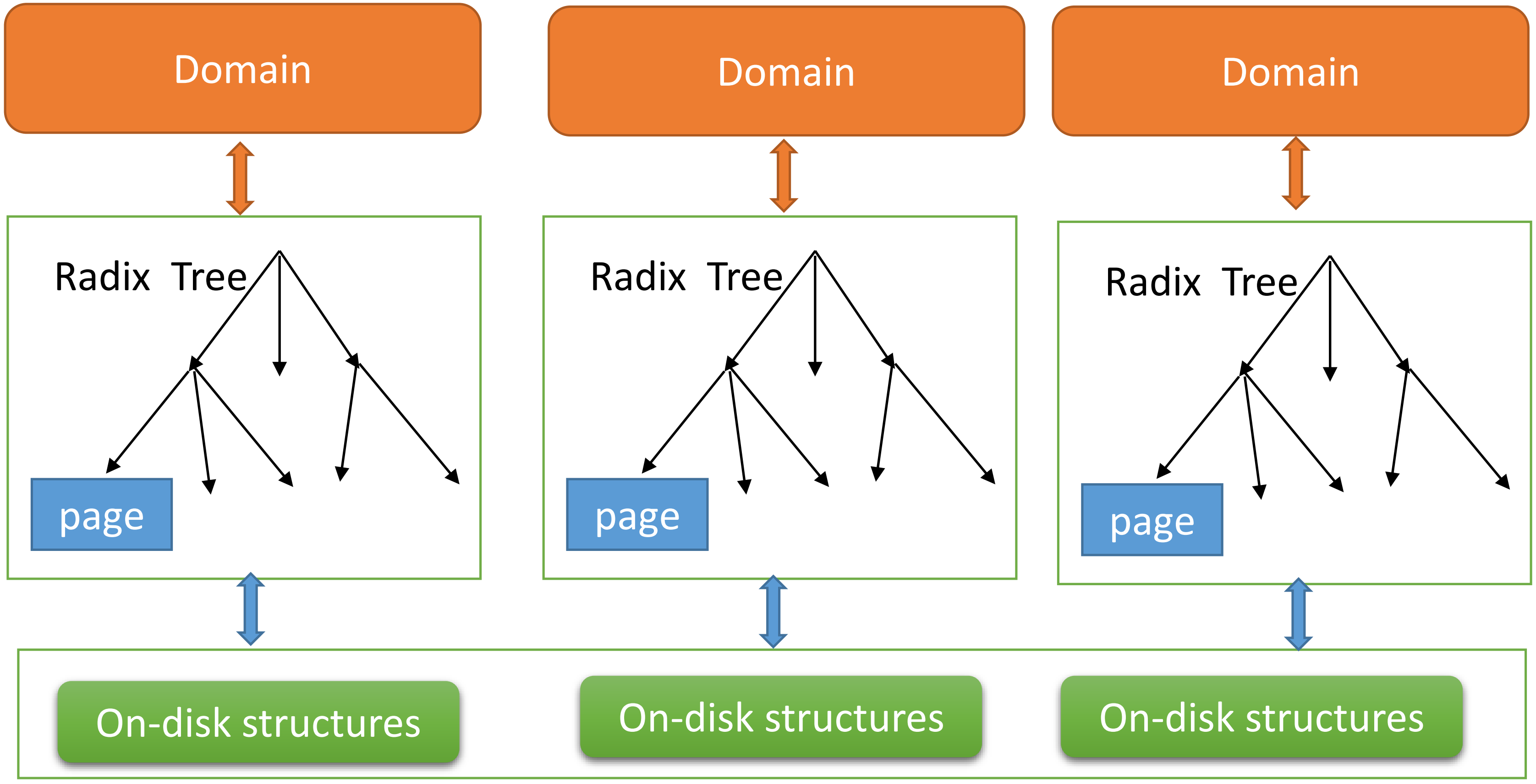
On-disk structures

On-disk structures

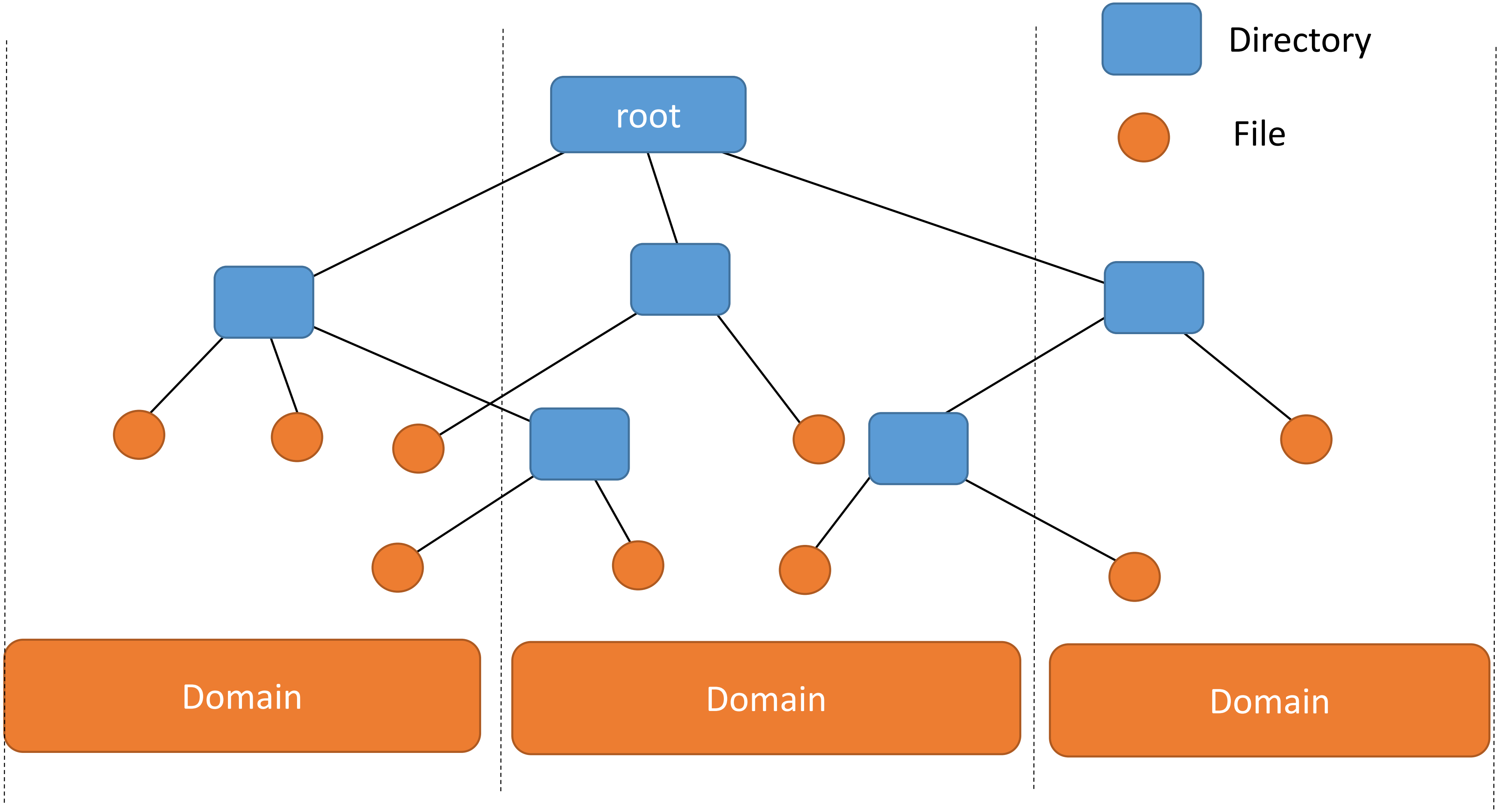
Beneath the file system: global device buffer cache address space



Dedicated buffer cache address space

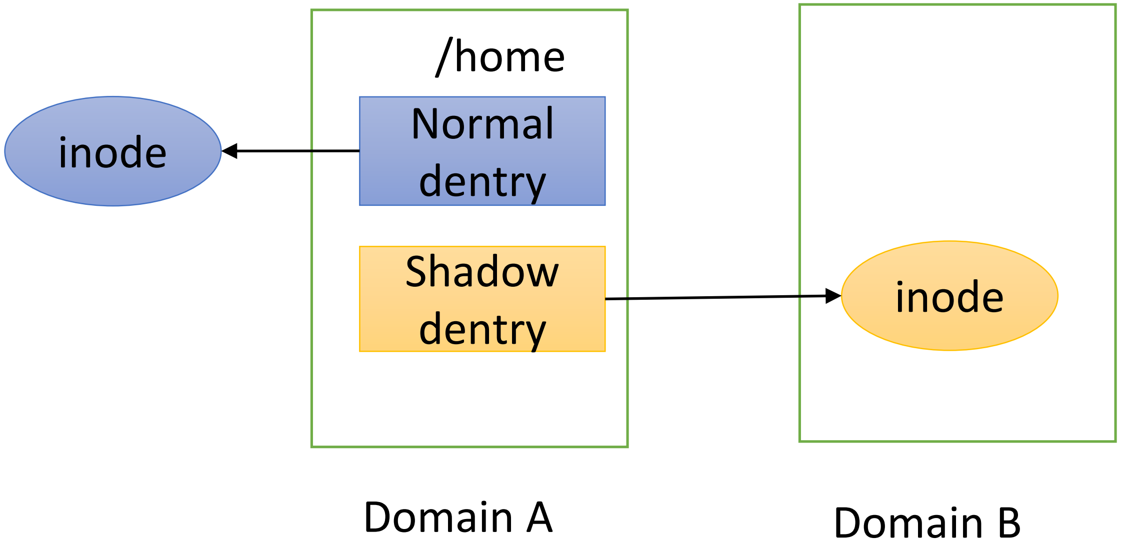


Distributed namespace

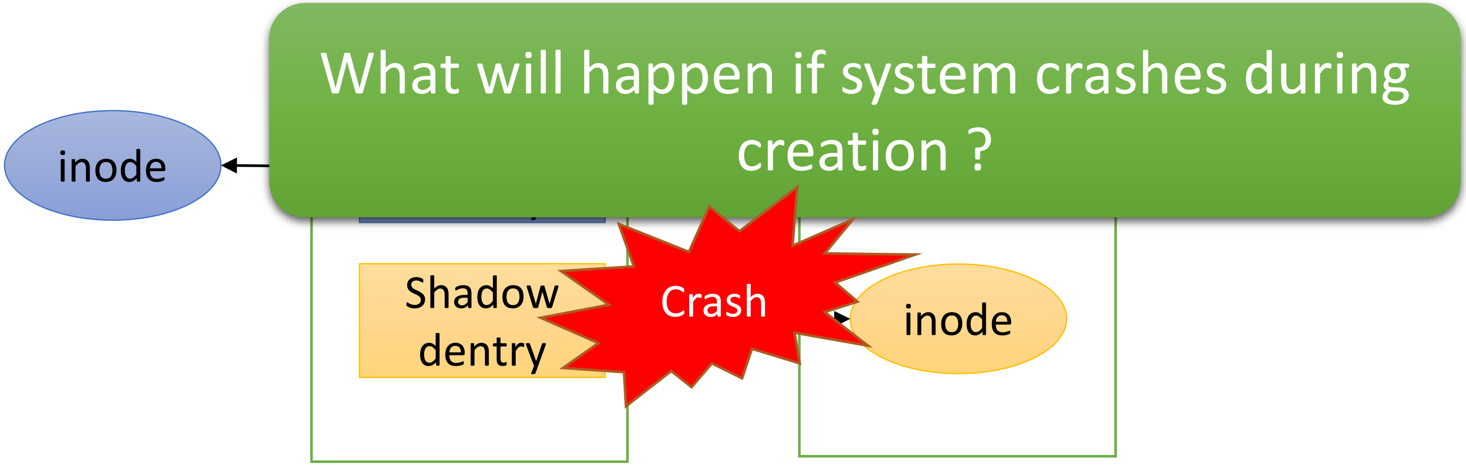


Distributed namespace

- Distributed object
  - Store shadow dentry under the parent directory
  - Distribute its inode to a remote domain

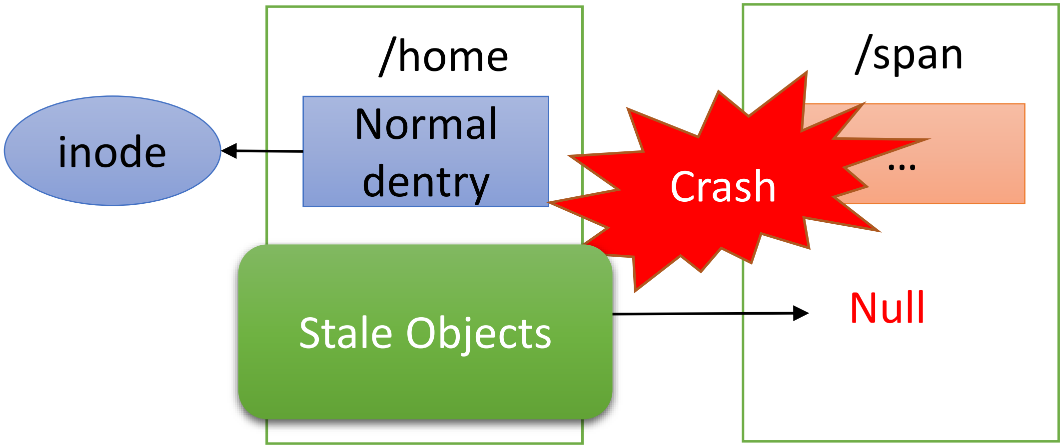


Crash consistency issues



What will happen if system crashes during creation ?

Possible inconsistency states



/home

inode

Normal dentry

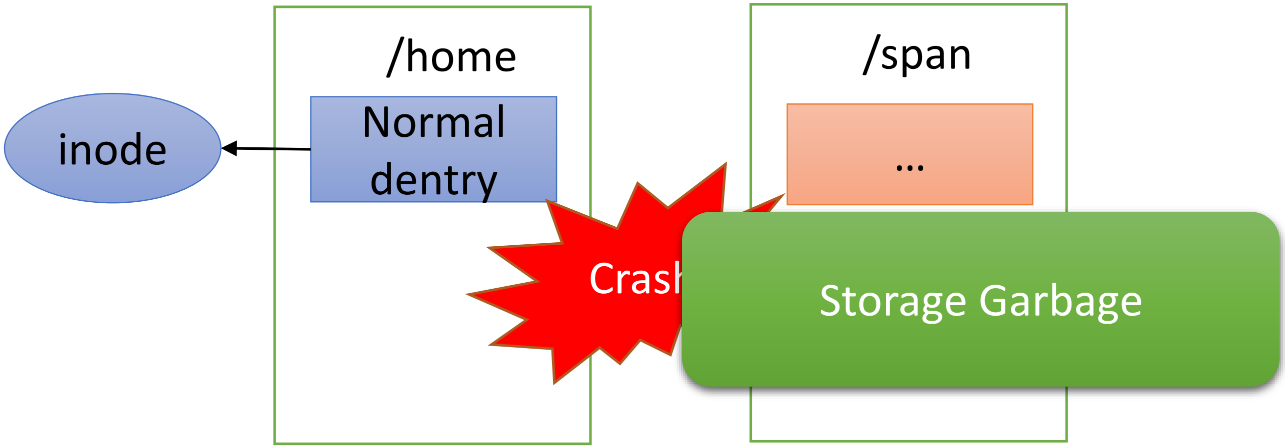
Crash

/span

…

Stale Objects

Null



/home

inode

Normal dentry

Crash

/span

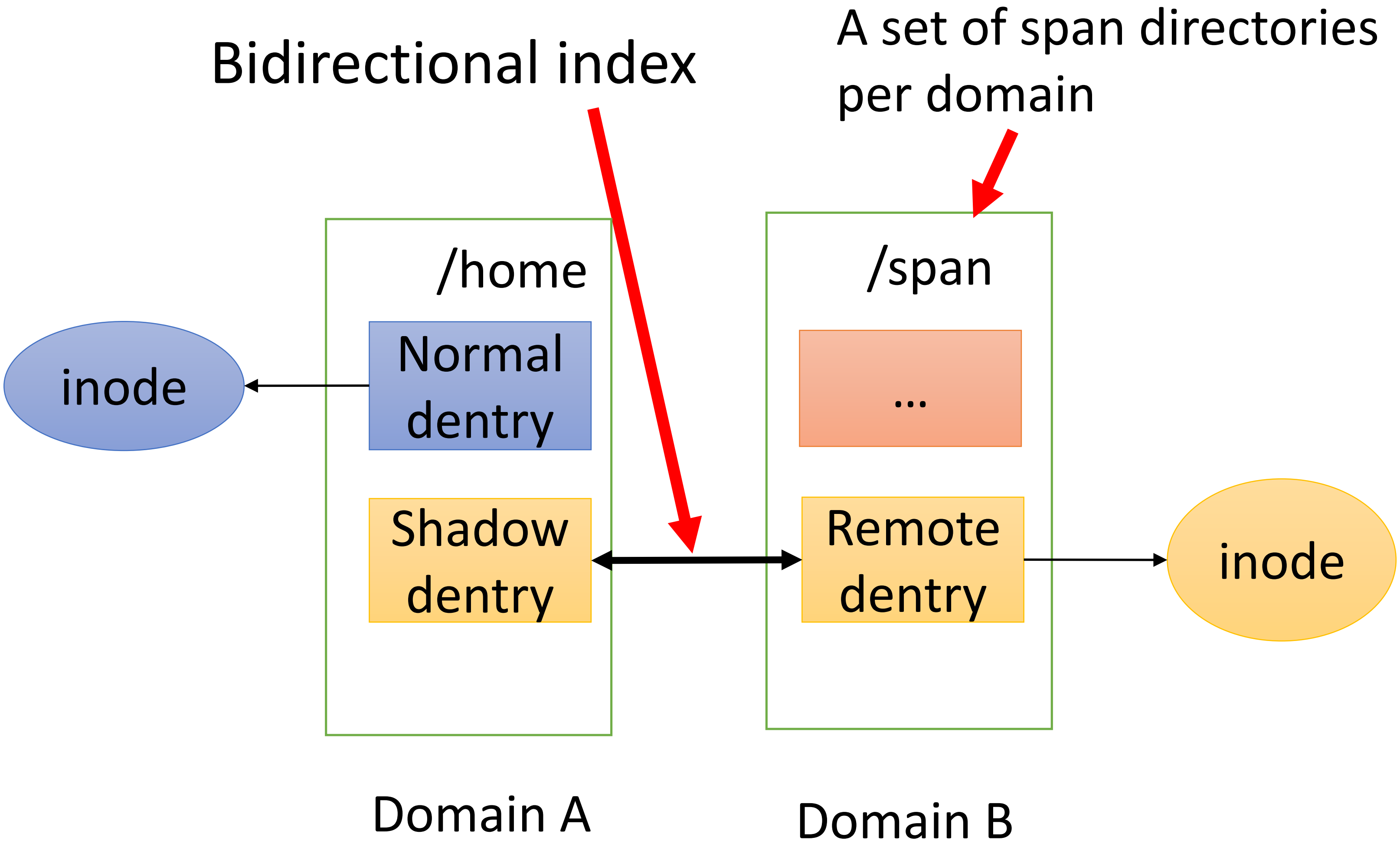
…

Storage Garbage

Crash consistency model

- We propose to build logical connection between domains beyond journaling

Logical connection beyond journaling

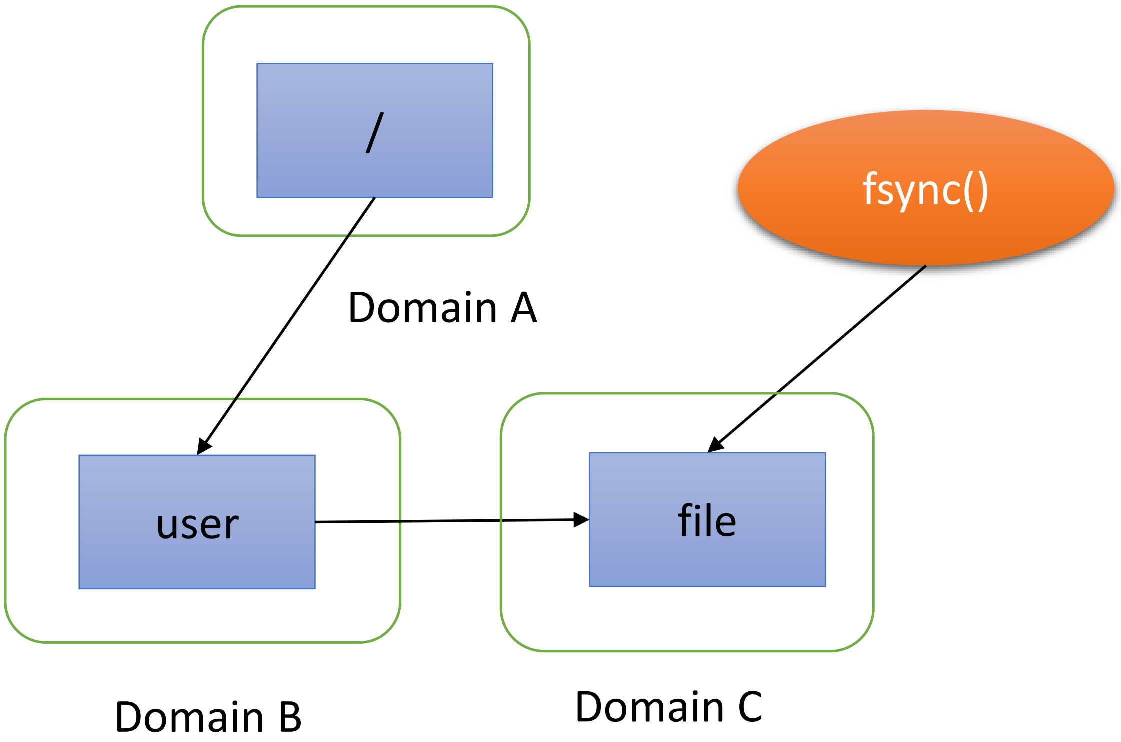


Crash consistency model

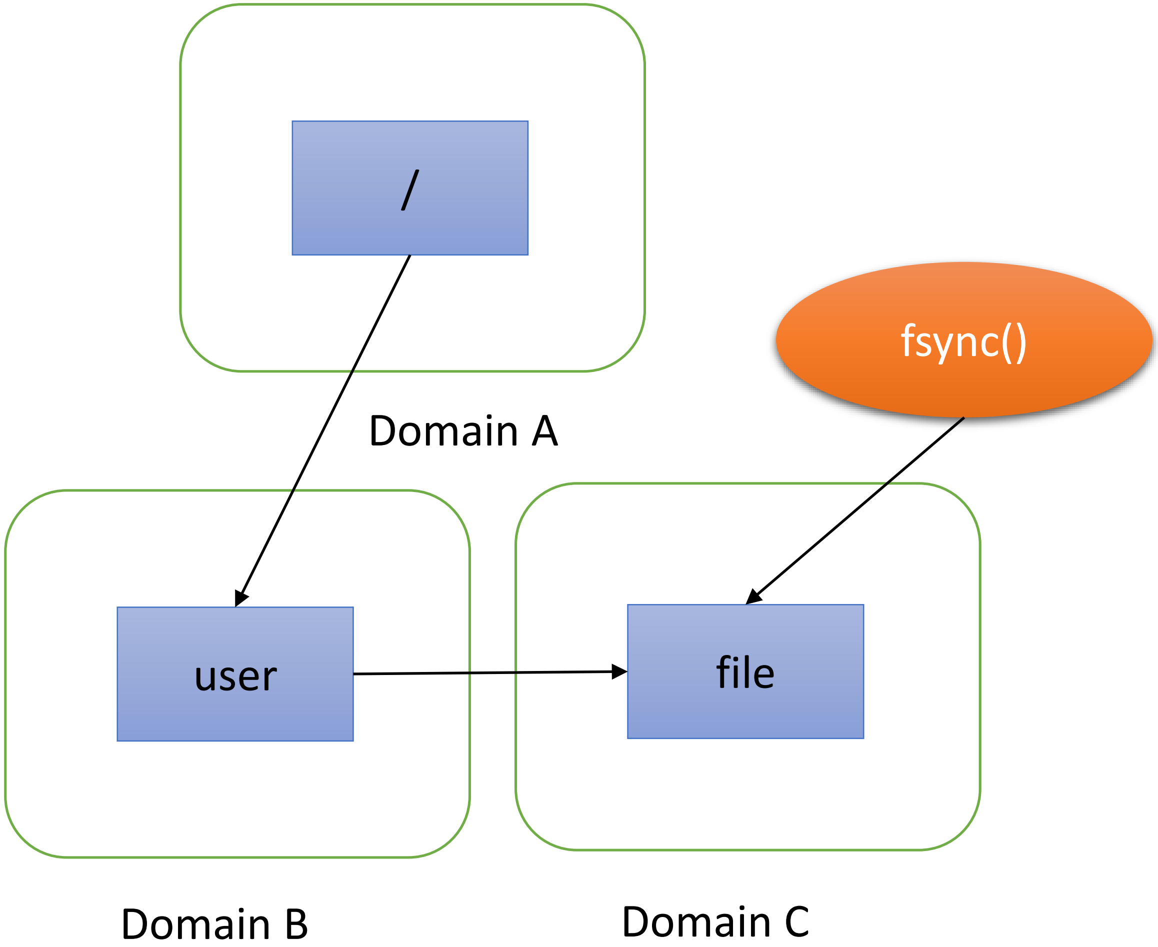
- Stale object deletion
  - Integrity validation during lookup and readdir
  - Remove the shadow dentries without remote objects
- Garbage Collection (GC)
  - Background GC thread runs in case of a system crash
  - GC deletes the remote objects without shadow dentries

Distributed synchronization

- Applications usually issue fsync() to explicitly persist their data



Distributed synchronization



/

fsync()

Domain A

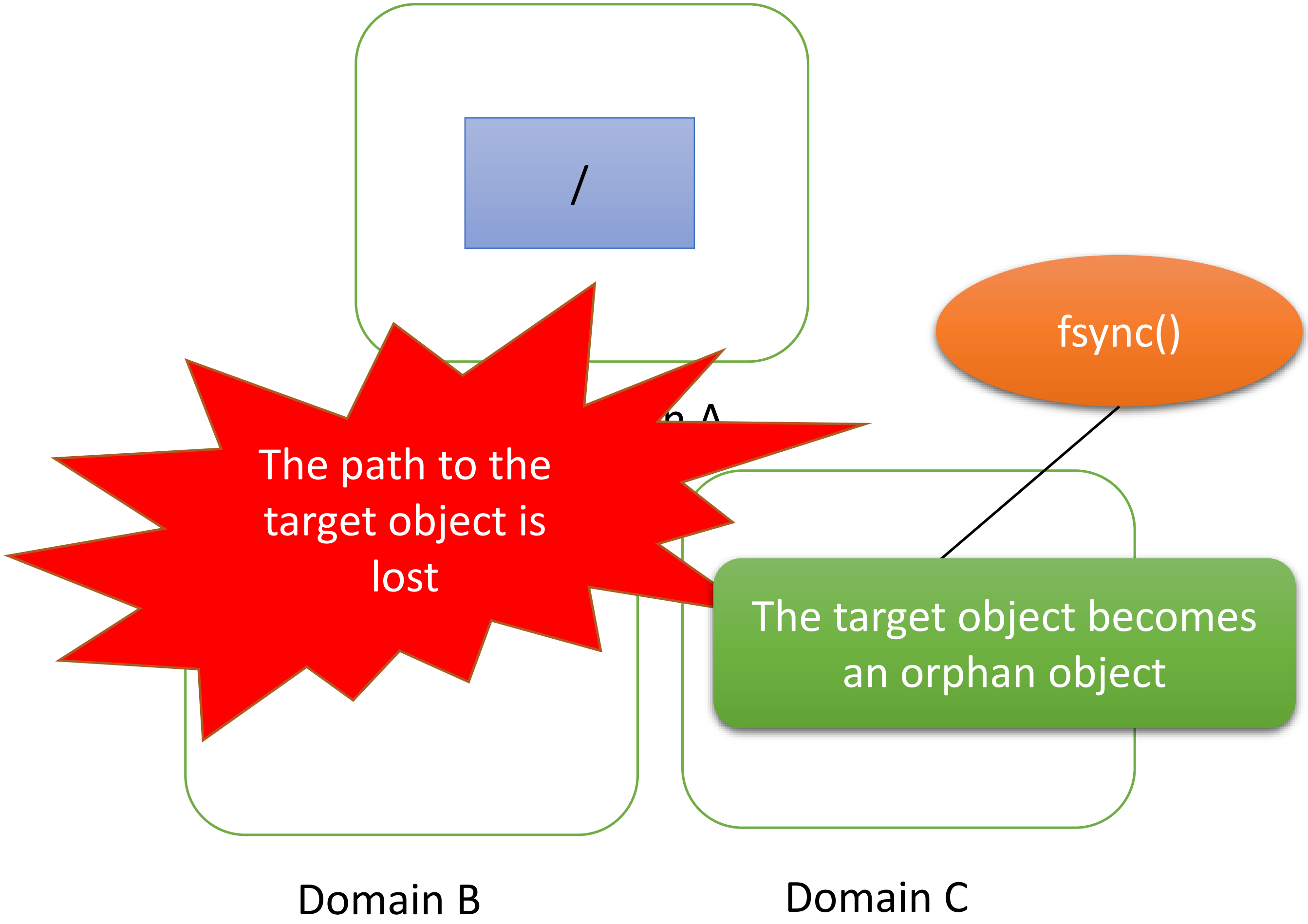
user

file

Domain B

Domain C

Possible inconsistency states



The path to the target object is lost

The target object becomes an orphan object

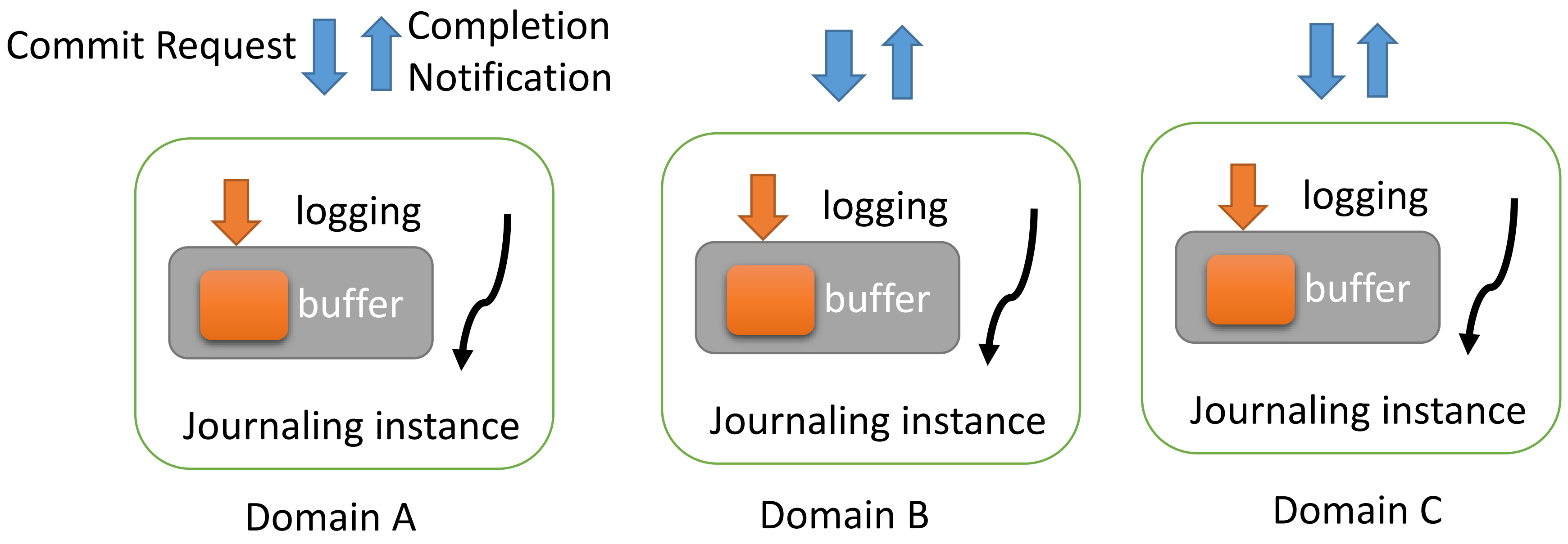
Intuitive solution

- Iteratively synchronize all the objects along the file path until reaching the root directory
    - Similar to what Ext4 with no journaling does

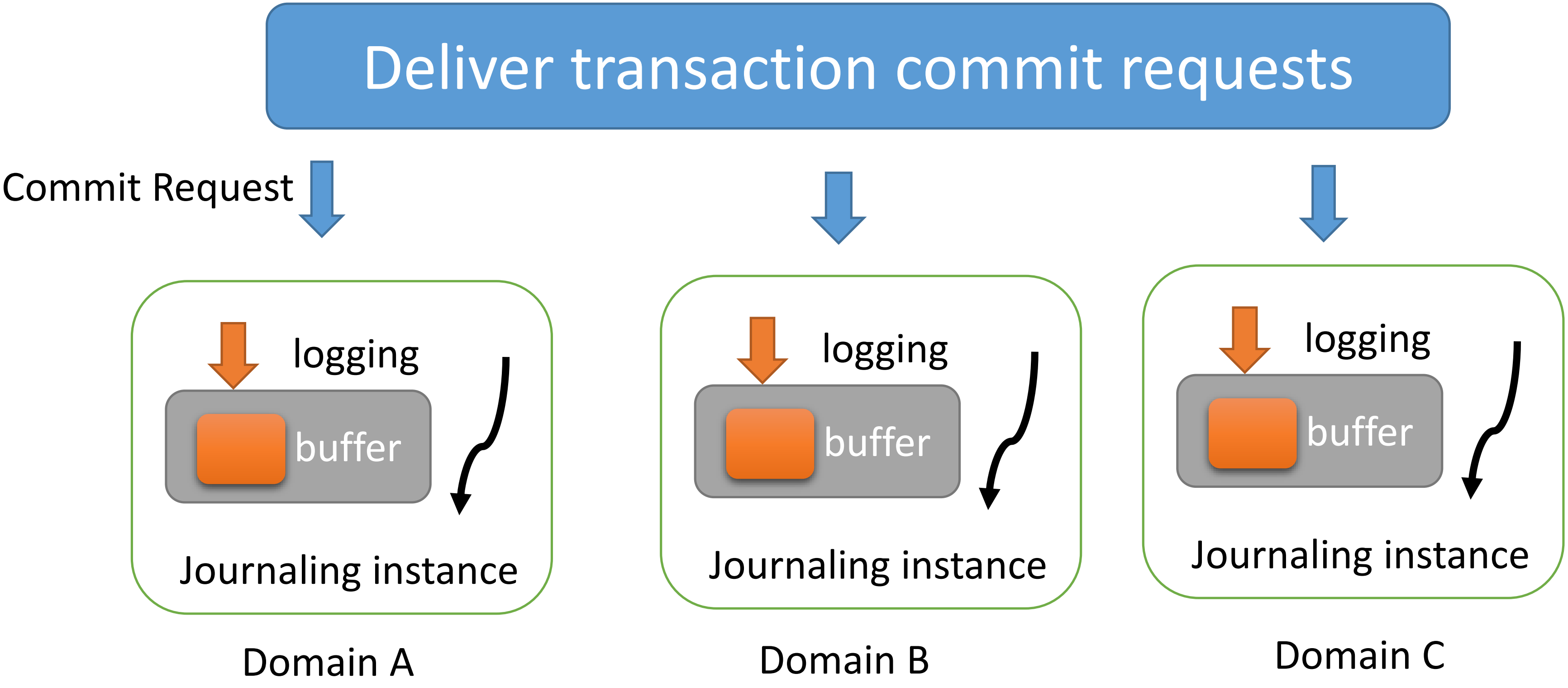
- The distributed synchronization latency is long:
    - Latency = latency(O1) + latency(O2) + …. + latency(On)

Parallel two-phase synchronization

- Leverage the client-server architecture of JBD2 to commit the transactions in parallel
- Check and wait for their completion in the end

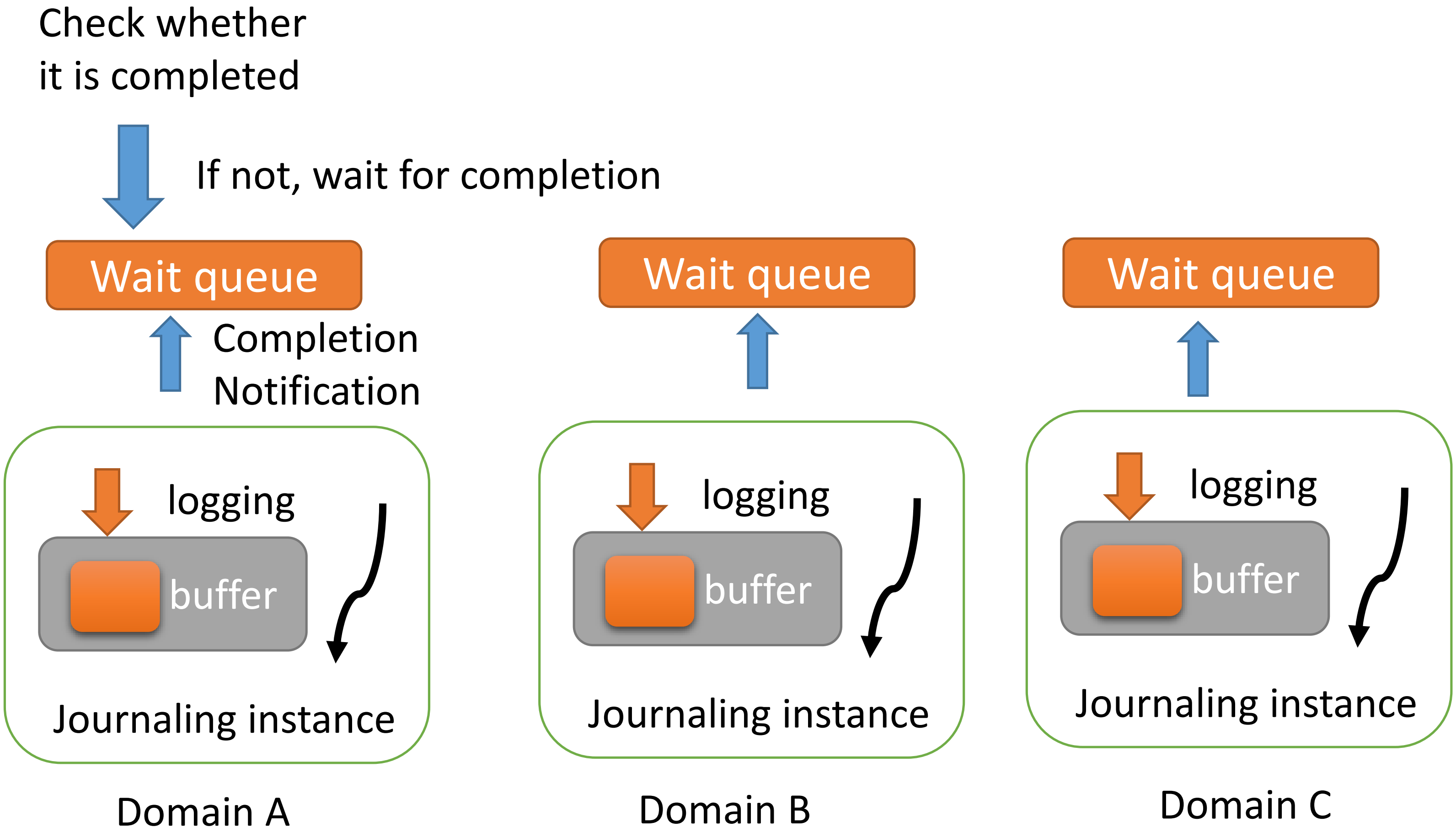


Committing phase



Deliver transaction commit requests

Validating phase



Check whether
it is completed

If not, wait for completion

Wait queue

Completion
Notification

Wait queue

Wait queue

Rename

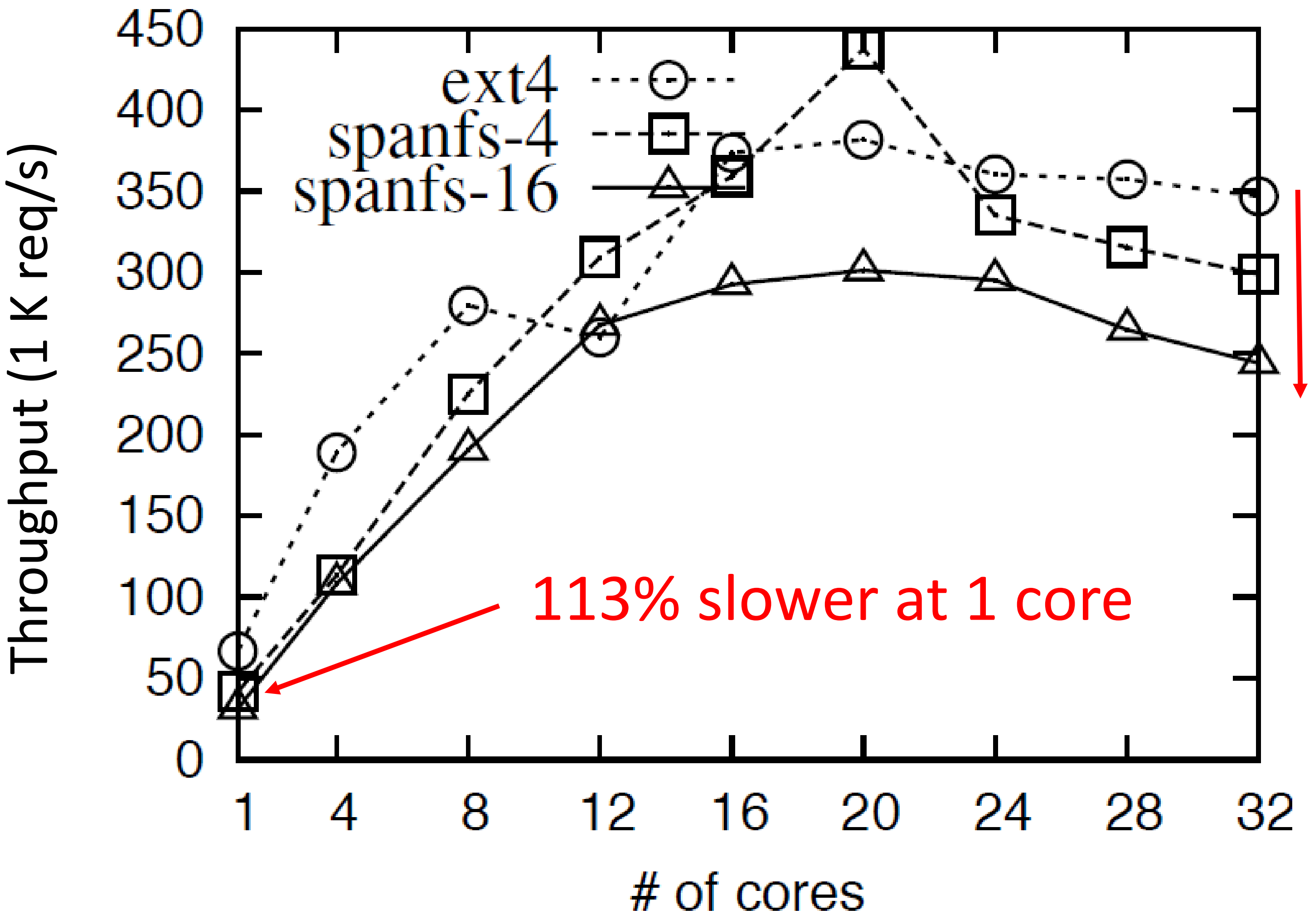
- The rename operation may involve multiple JBD2 handles across multiple domains

- We proposed the ordered transaction commit mechanism to achieve rename atomicity
  - Control the commit sequence of the JBD2 handles
  - System crashes lead to a small number of inconsistencies
  - These inconsistency states can be verified online

Experiments

- We implemented SpanFS based on Ext4 in Linux 3.18.0
- We evaluated SpanFS against Ext4 on an intel 32 core machine with a FusionIO SSD

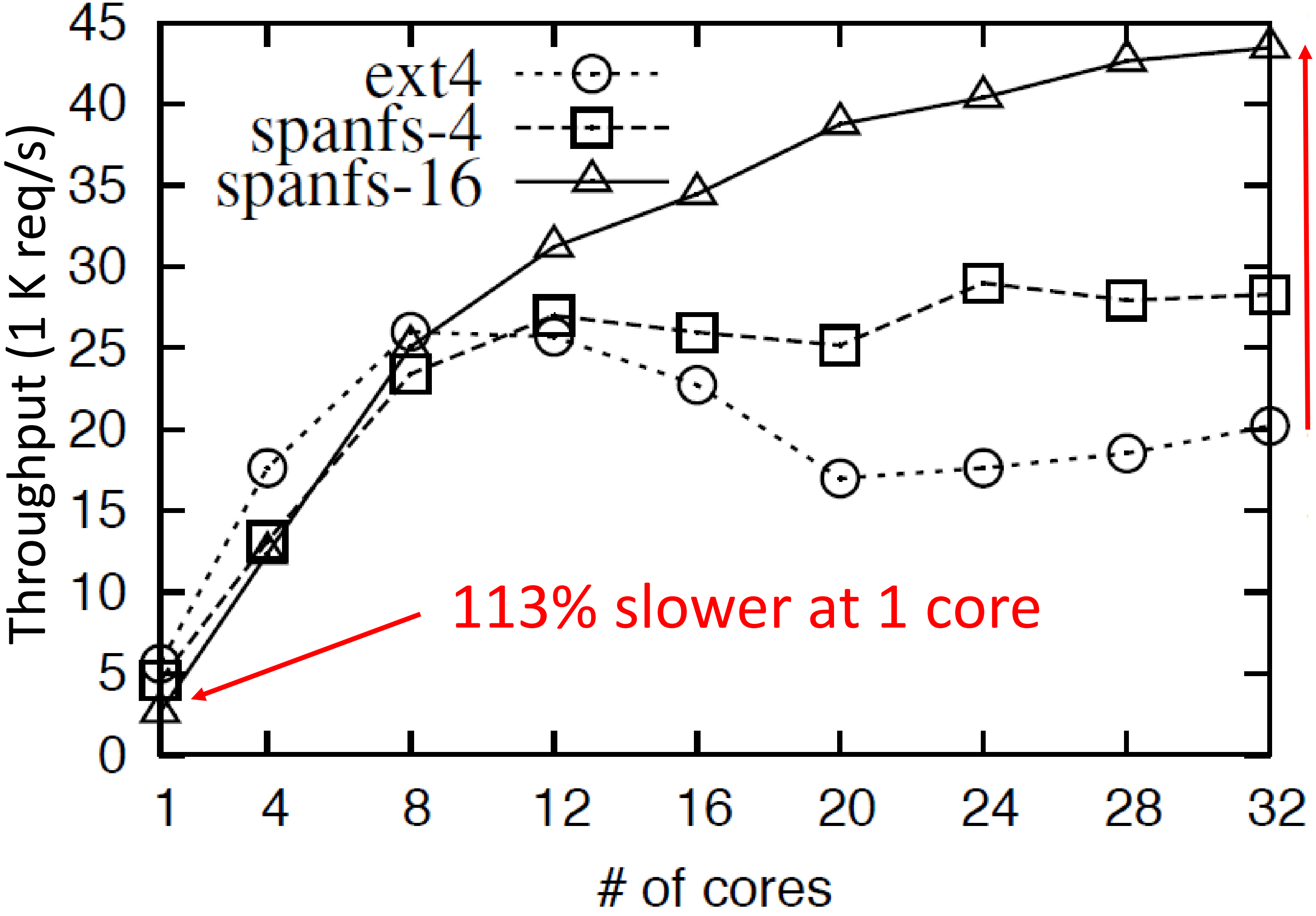
Create



42% slower

113% slower at 1 core

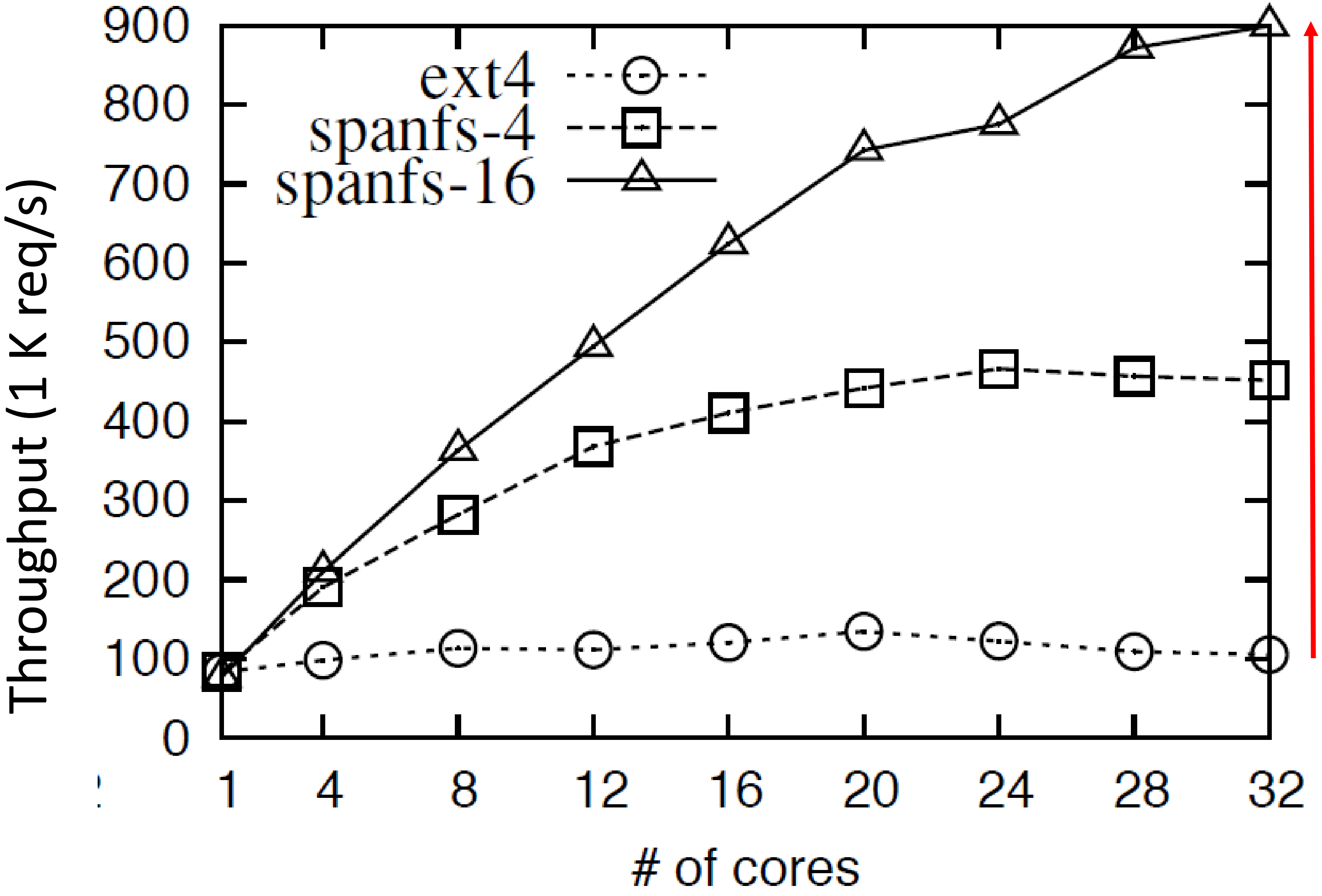
Append



1.15X faster

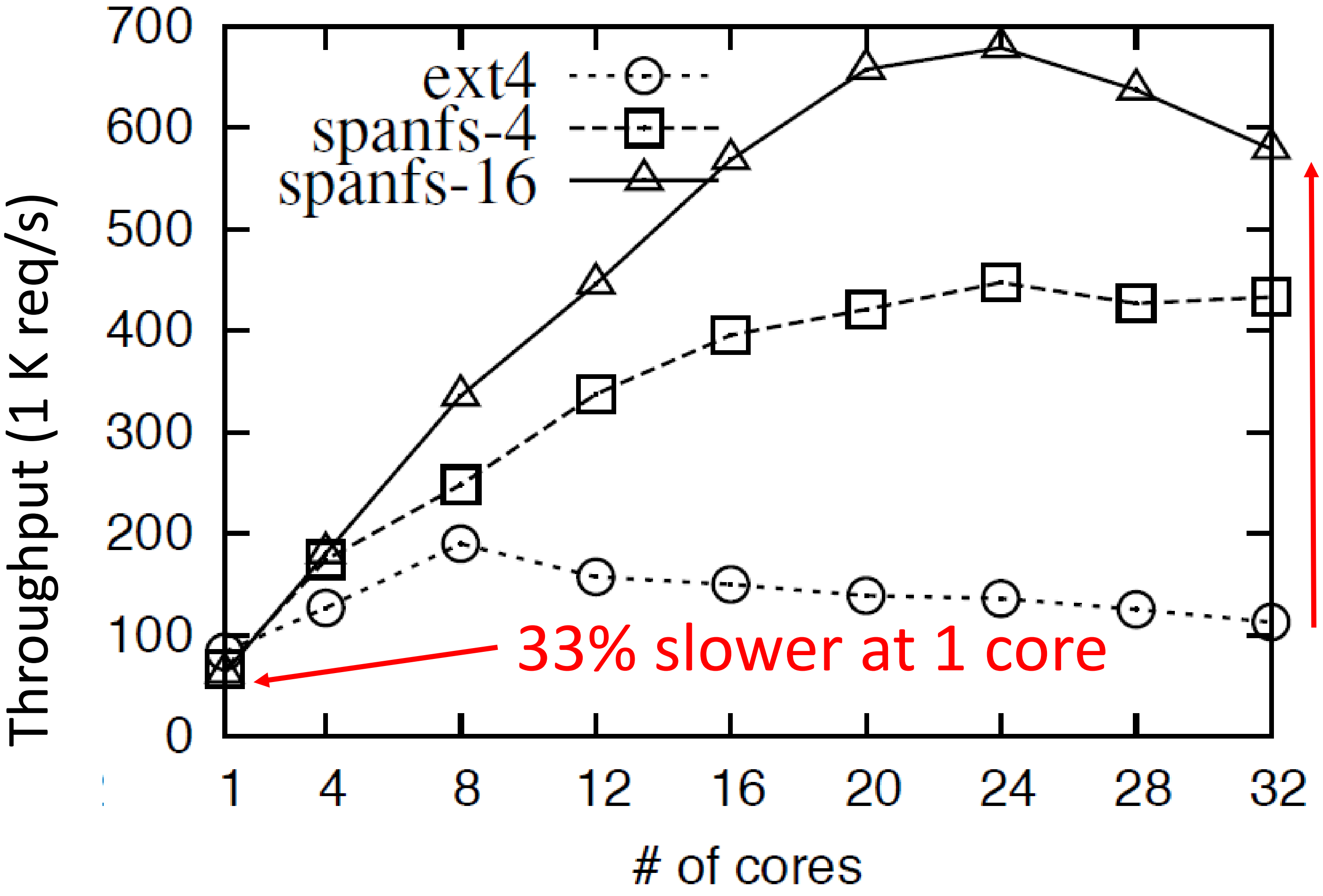
113% slower at 1 core

Truncate



7.53X faster

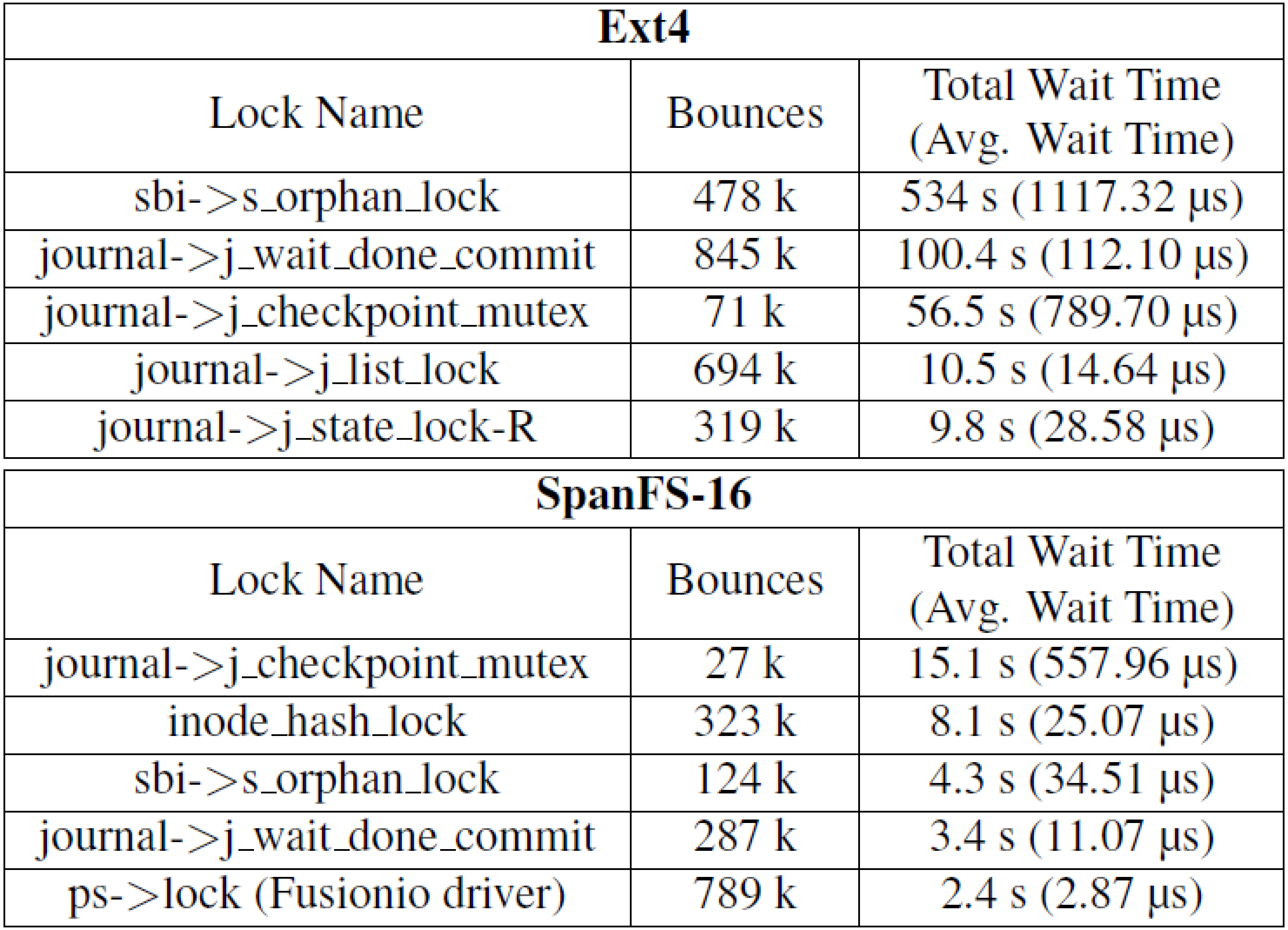
Delete



4.13X faster

33% slower at 1 core

Data profiling



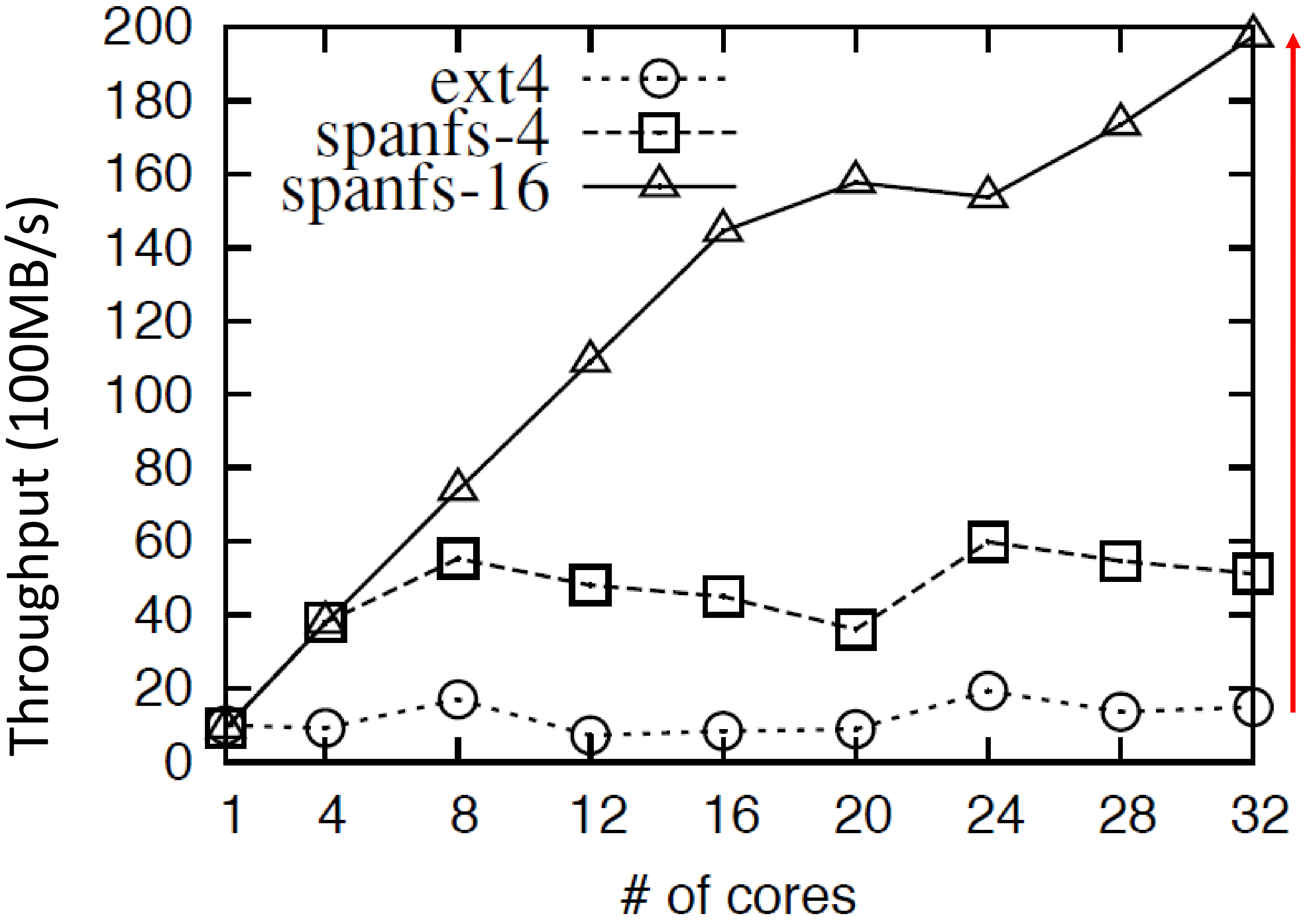
Ext4		
Lock Name	Bounces	Total Wait Time (Avg. Wait Time)
sbi->s_orphan_lock	478 k	534 s (1117.32 µs)
journal->j_wait_done_commit	845 k	100.4 s (112.10 µs)
journal->j_checkpoint_mutex	71 k	56.5 s (789.70 µs)
journal->j_list_lock	694 k	10.5 s (14.64 µs)
journal->j_state_lock-R	319 k	9.8 s (28.58 µs)

681.2 s

SpanFS-16		
Lock Name	Bounces	Total Wait Time (Avg. Wait Time)
journal->j_checkpoint_mutex	27 k	15.1 s (557.96 µs)
inode_hash_lock	323 k	8.1 s (25.07 µs)
sbi->s_orphan_lock	124 k	4.3 s (34.51 µs)
journal->j_wait_done_commit	287 k	3.4 s (11.07 µs)
ps->lock (Fusionio driver)	789 k	2.4 s (2.87 µs)

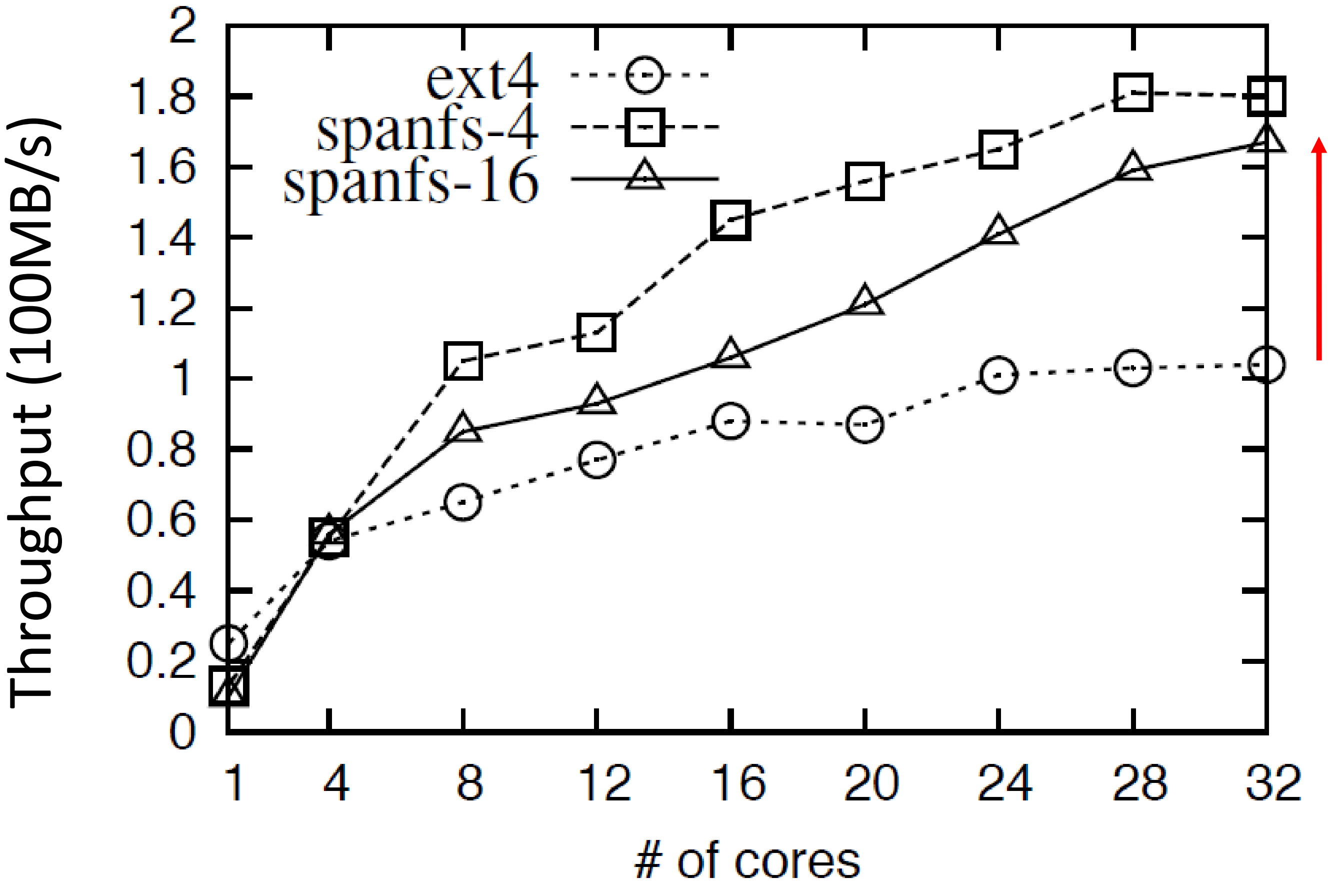
33.3 s

Sequential buffered writes



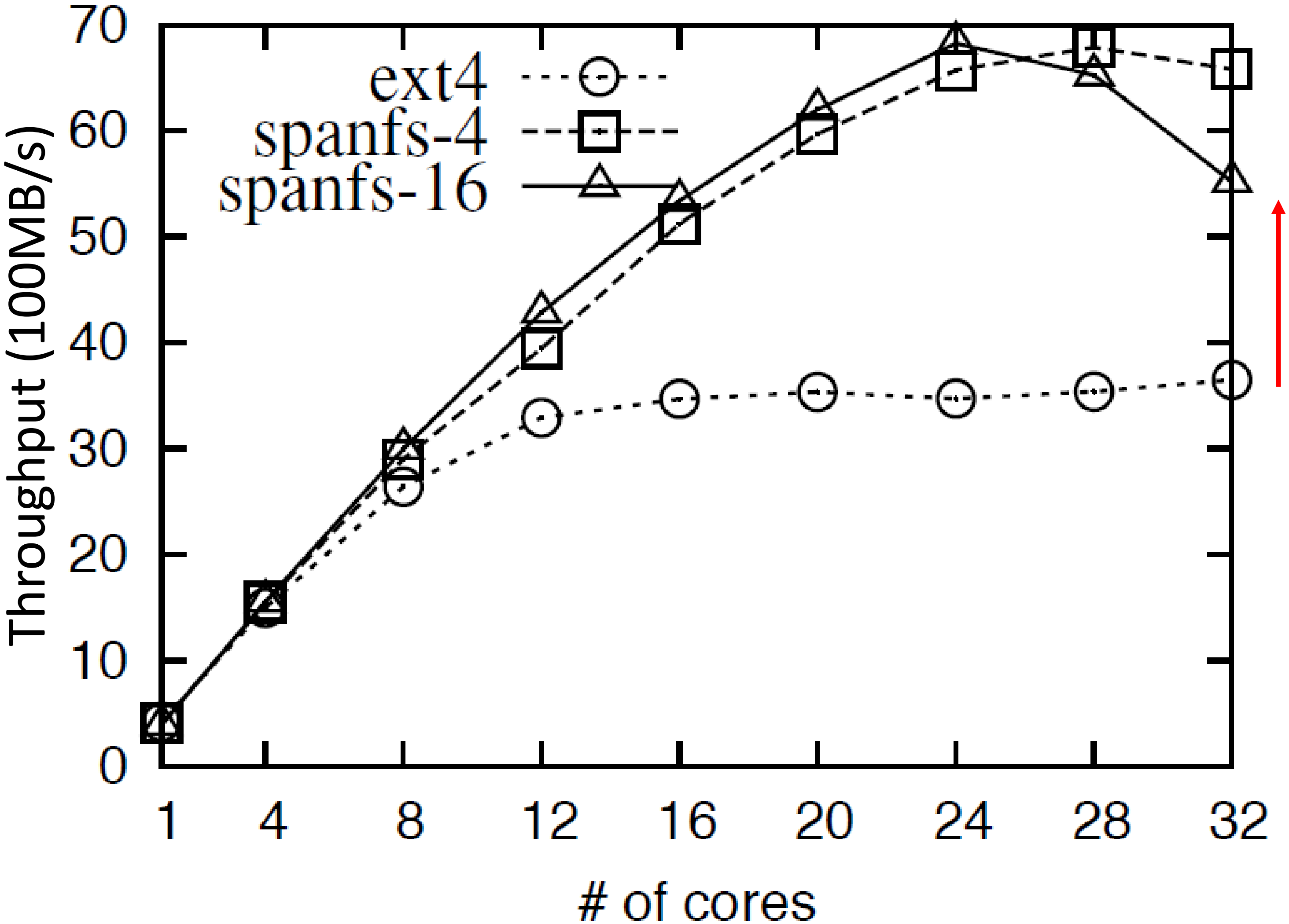
1226% faster

Sequential synchronous writes



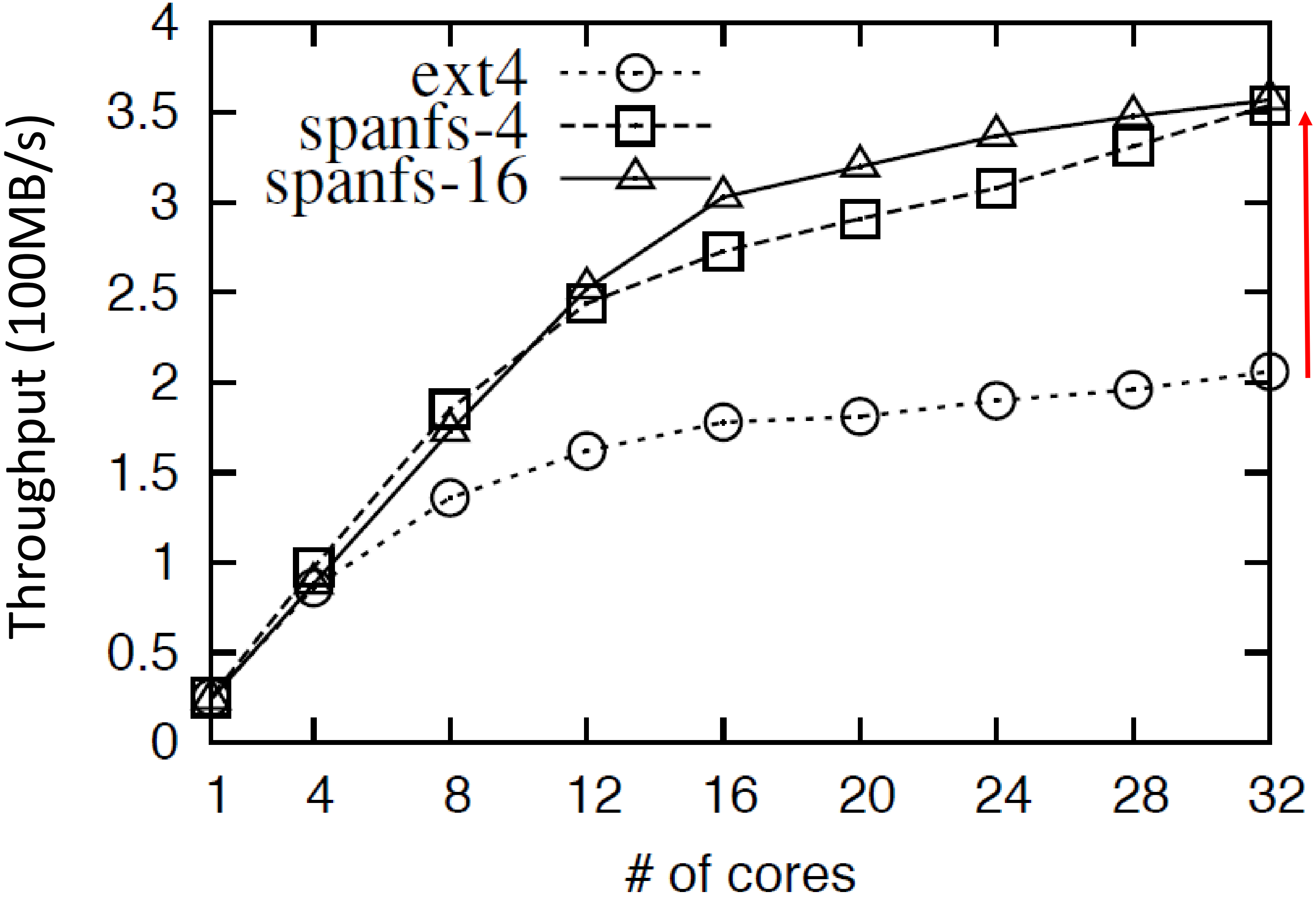
60% faster

Fileserver



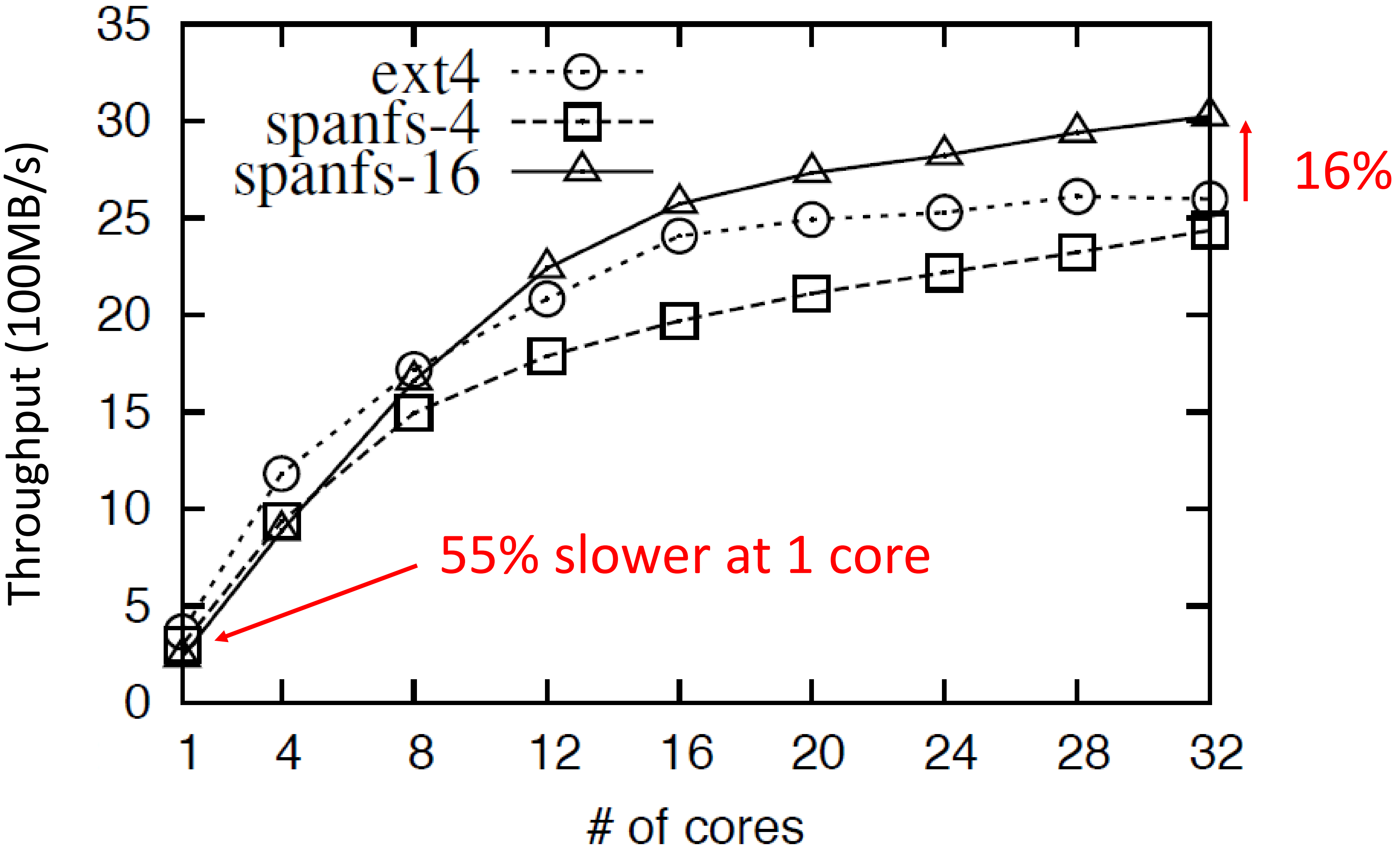
51% faster

Varmail



73% faster

Dbench

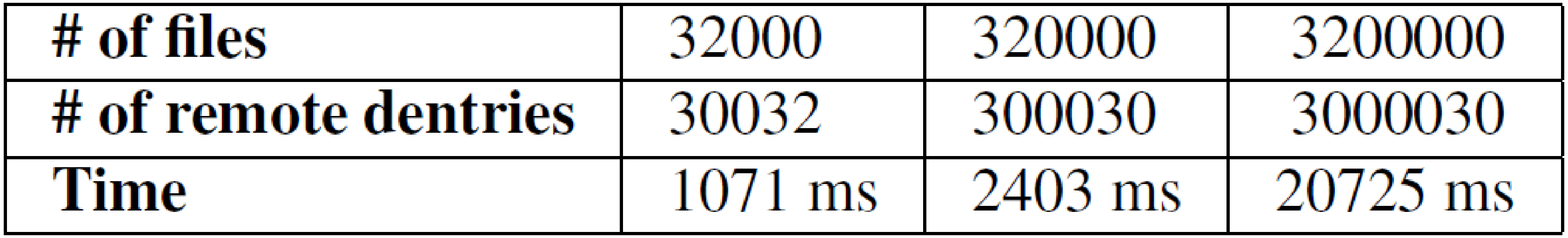


16%

55% slower at 1 core

Garbage collection performance

The time taken to scan different numbers of files by GC



# of files	32000	320000	3200000
# of remote dentries	30032	300030	3000030
Time	1071 ms	2403 ms	20725 ms

Garbage collection performance

- Measure the GC performance impact on the foreground I/O workloads
  - Prepare 3.2 millions of files
  - Run the GC thread after remount
  - Run 32 Varmail instances for 60 s
- The GC thread takes 21.9 s to complete the scan
- The total throughput of the Varmail workload has been degraded by 12%

Conclusion

- Present an exhaustive analysis of the scalability bottlenecks of existing file systems.
- Attribute the scalability issues to their centralized design
    - Contention on shared data structures in memory
    - Serialization of I/O actions on devices
- Propose a novel journaling file system SpanFS to achieve scalability on many-core
- Demonstrate that SpanFS scales much better than the baseline

Thanks