Designing a True Direct-Access File System with DevFS

Sudarsun Kannan, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau

University of Wisconsin-Madison

Yuangang Wang, Jun Xu, Gopinath Palani

Huawei Technologies

Modern Fast Storage Hardware

- Faster nonvolatile memory technologies such as NVMe, 3D Xpoint

Hard Drives

PCIe-Flash

3D Xpoint

	Hard Drives	PCIe-Flash	3D Xpoint
BW:	2.6MB/s	250MB/s	**1.3GB/s**
H/W Lat:	7.1ms	68us	**12us**
S/W cost:	8us	8us	**6us**
OS cost:	5us	5us	**4us**

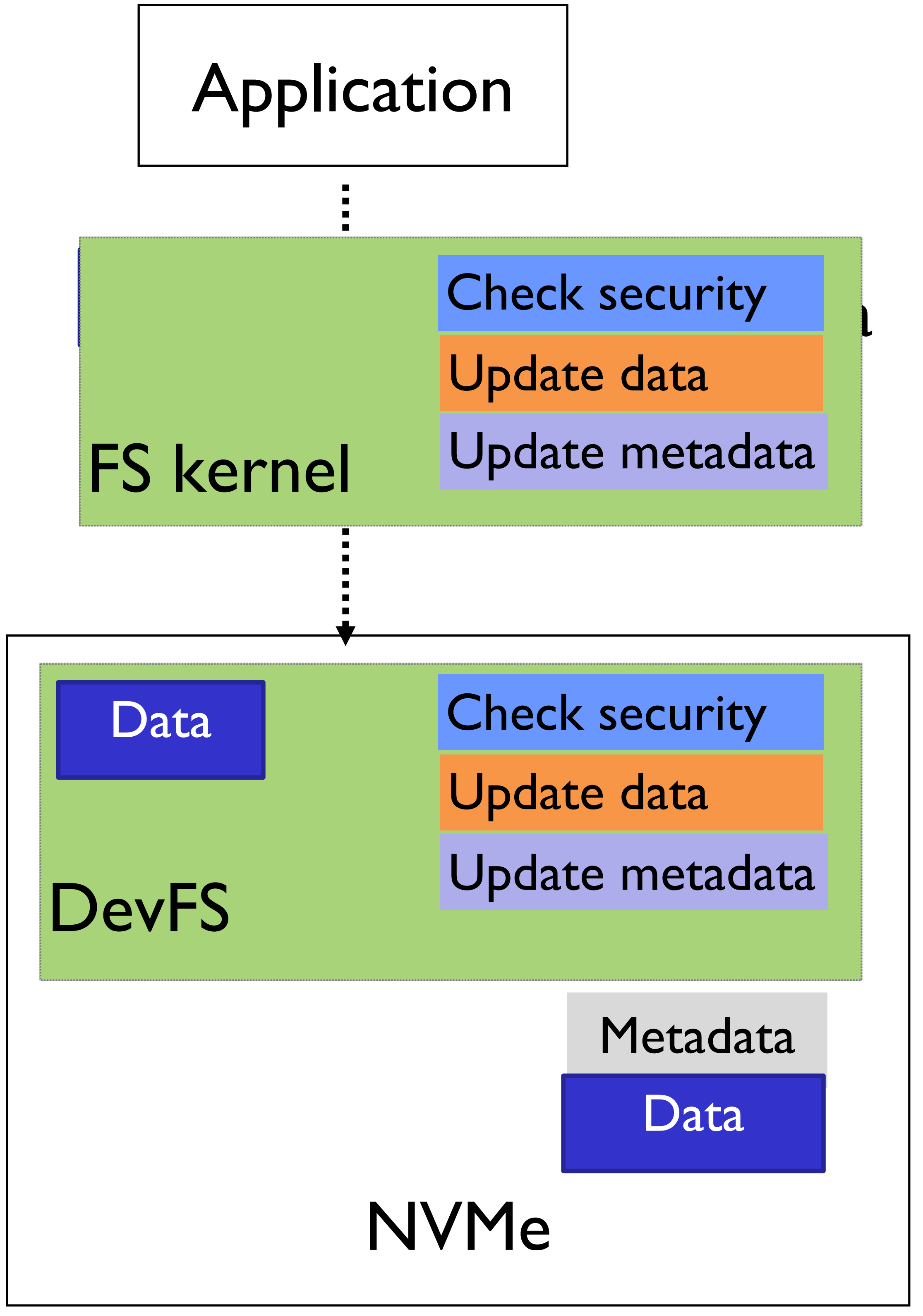
- Bottlenecks shift from hardware to software (file system)

Why Use OS File System?

- Millions of applications use OS-level file system (FS)
  - Guarantees integrity, concurrency, crash-consistency, and security
- Object stores have been designed to reduce OS cost [HDFS, CEPH]
  - Developers unwilling to modify POSIX-interface
  - Need faster file systems and not new interface
- User-level POSIX-based FS fail to satisfy fundamental properties

Device-level File System (DevFS)

- Move file system into the device hardware
- Use device-level CPU and memory for DevFS
- Apps. bypass OS for **control and data plane**
- DevFS handles integrity, concurreny, crash-consistency, and security
- Achieves **true direct-access**



Challenges of Hardware File System

- Limited memory inside the device
  - Reverse-cache inactive file system structures to host memory
- DevFS lack visibility to OS state (e.g., process permission)
  - Make OS share required (process) information with “down-call”

Performance

- Emulate DevFS at the device-driver level
- Compare DevFS with state-of-the-art NOVA file system
- Benchmarks - more than 2X write and 1.8X read throughput
- Snappy compression application - up to 22% higher throughput
- Memory-optimized design reduces file system memory by 5X

Outline

Introduction

Background

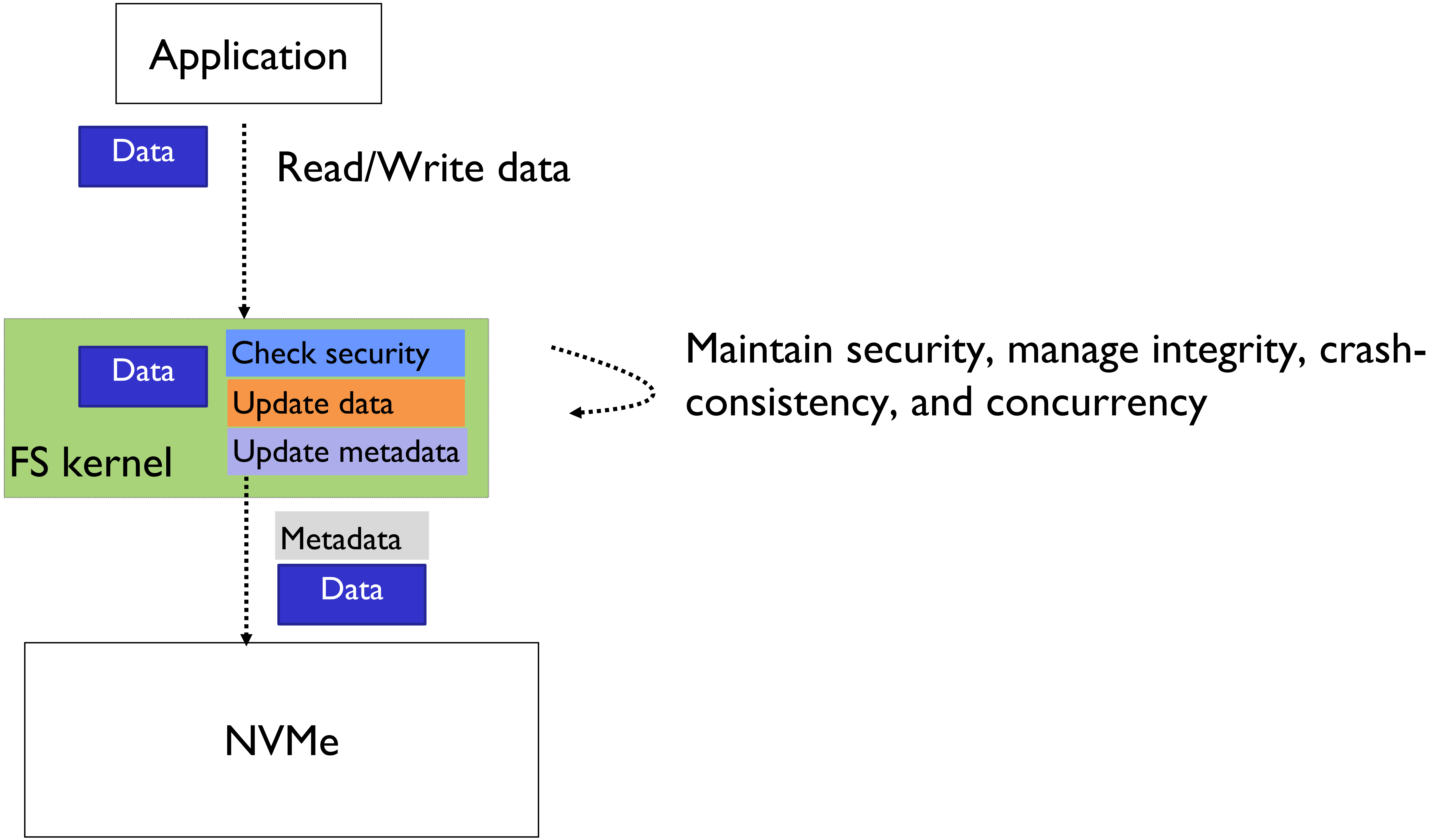
Motivation

DevFS Design

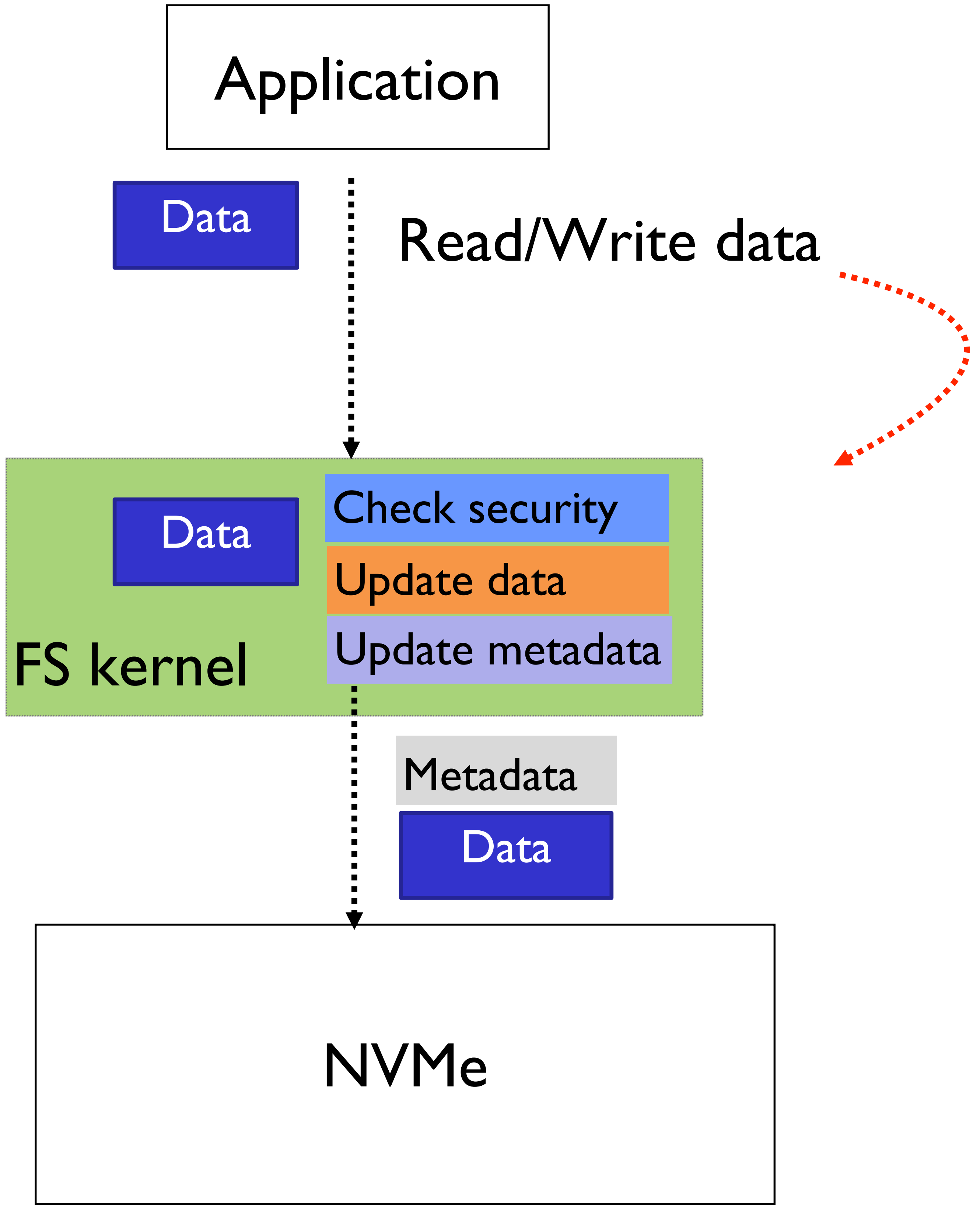
Evaluation

Conclusion

Traditional S/W Storage Stack



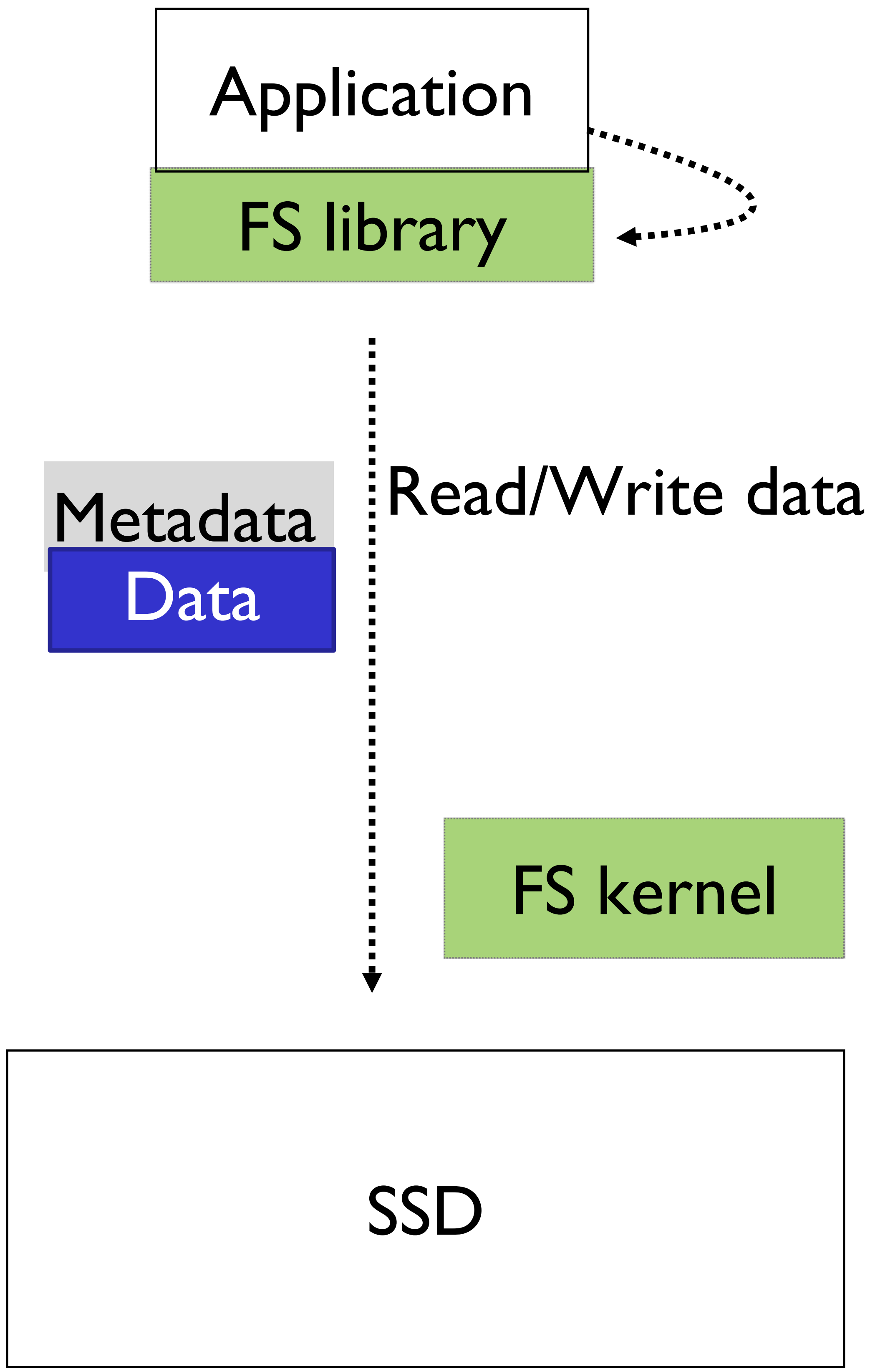
Traditional S/W Storage Stack



User-to-kernel switch for every data plane operation

High software-indirection latency before storage access

Holy grail of Storage Research



Challenge 1: How to bypass OS and provide direct-storage access?

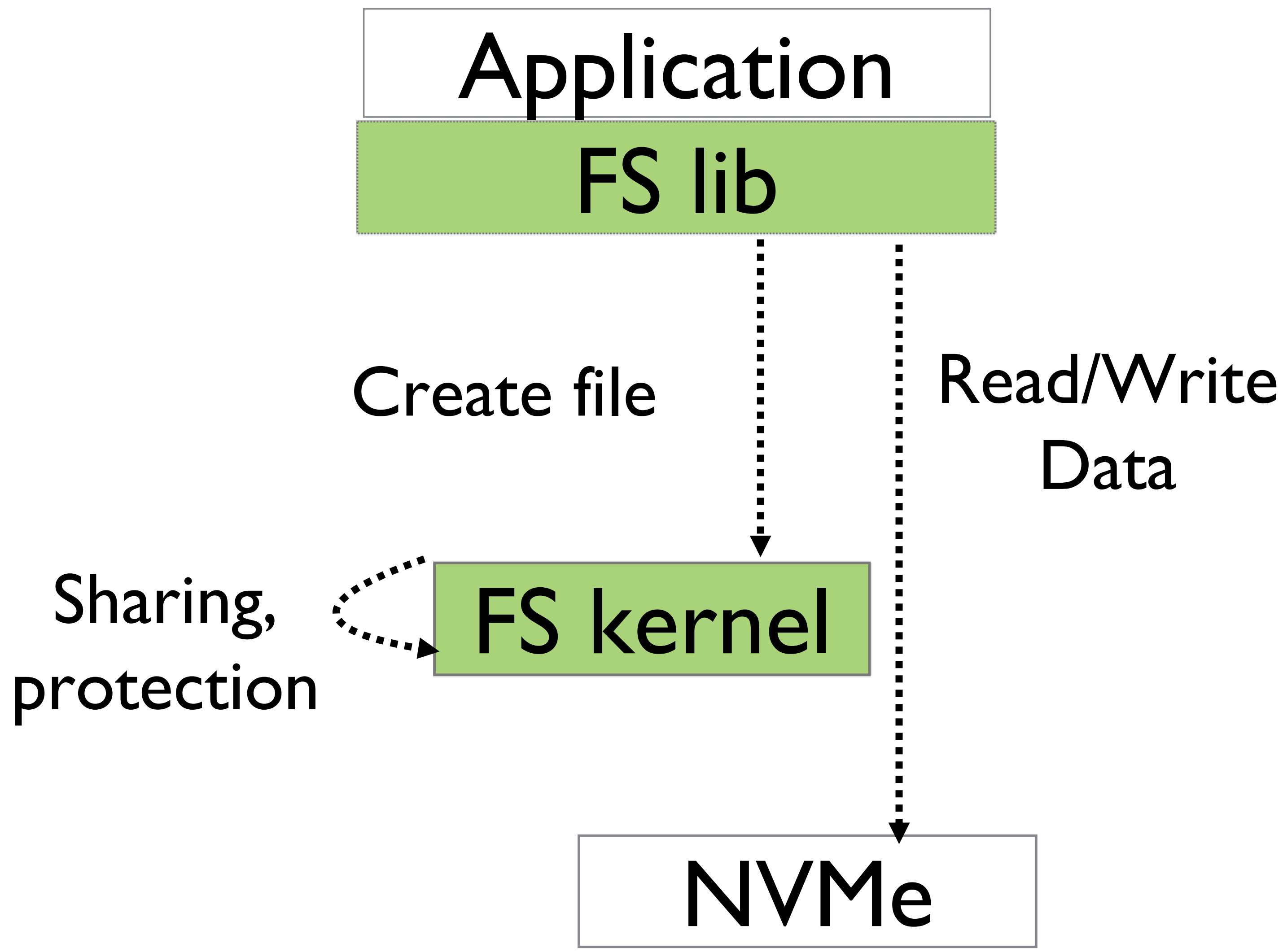
Challenge 2: How to provide direct-access without compromising integrity, concurrency, crash-consistency, and security?

Classes of Direct-Access File Systems

- Prior approaches have attempted to provide user-level direct access
- We categorize them into four classes:
  - Hybrid user-level
  - Hybrid user-level with trusted server (Microkernel approach)
  - Hybrid device
- Full device-level file system (proposed)

Hybrid User-level File System

- Split file system into user library and kernel file components
- Kernel FS handles control plane (e.g., file creation)
- Library handles data plane (e.g., read, write) and manages metadata

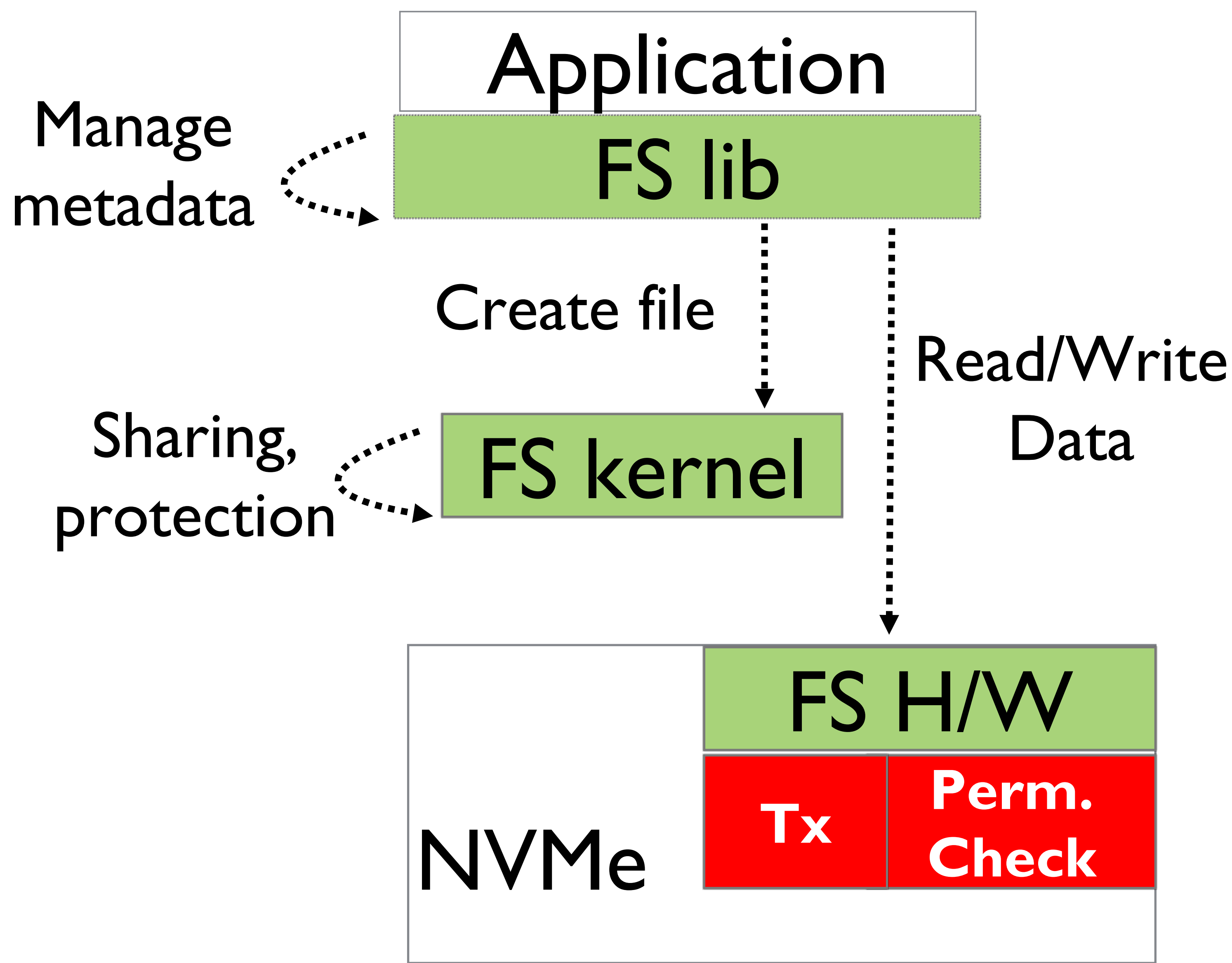


Well known hybrid approaches

- Arrakis (OSDI '14)
- Strata (SOSP '17)

Hybrid Device File System

- File system split across user-level library, kernel, and hardware
- Control and data-plane operations same as hybrid user-level FS
- However, some functionalities moved inside the hardware



Well known hybrid approaches

- Moneta-D (ASPLOS '12)
- TxDev (OSDI '08)

Outline

Introduction

Background

Motivation

DevFS Design

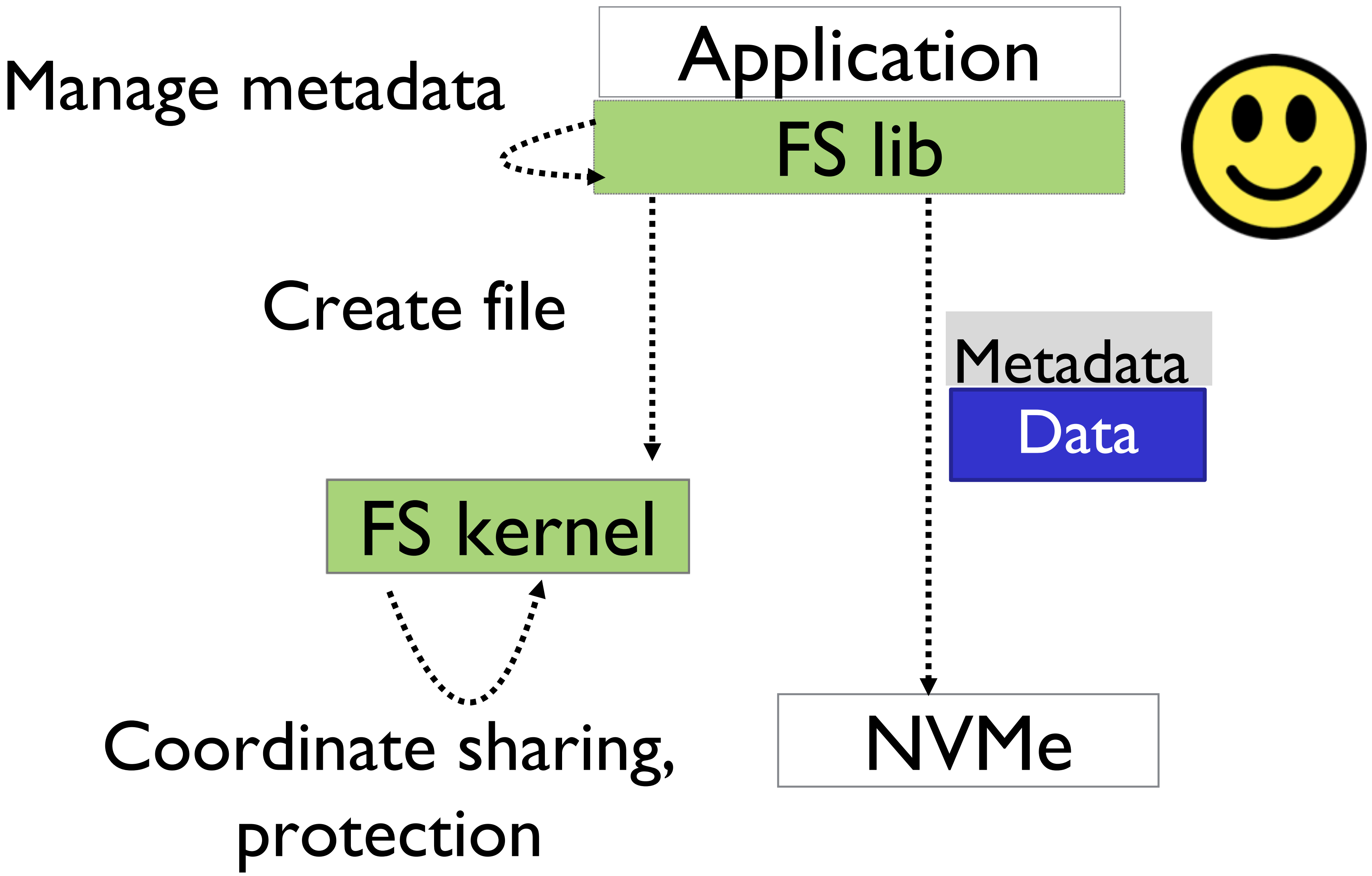
Evaluation

Conclusion

File System Properties

- Integrity
  - Correctness of FS metadata for single & concurrent access
- Crash-consistency
  - FS metadata consistent after a failure
- Security
  - No permission violation for both **control and data-plane**
  - OS-level file system checks permission for control and data plane

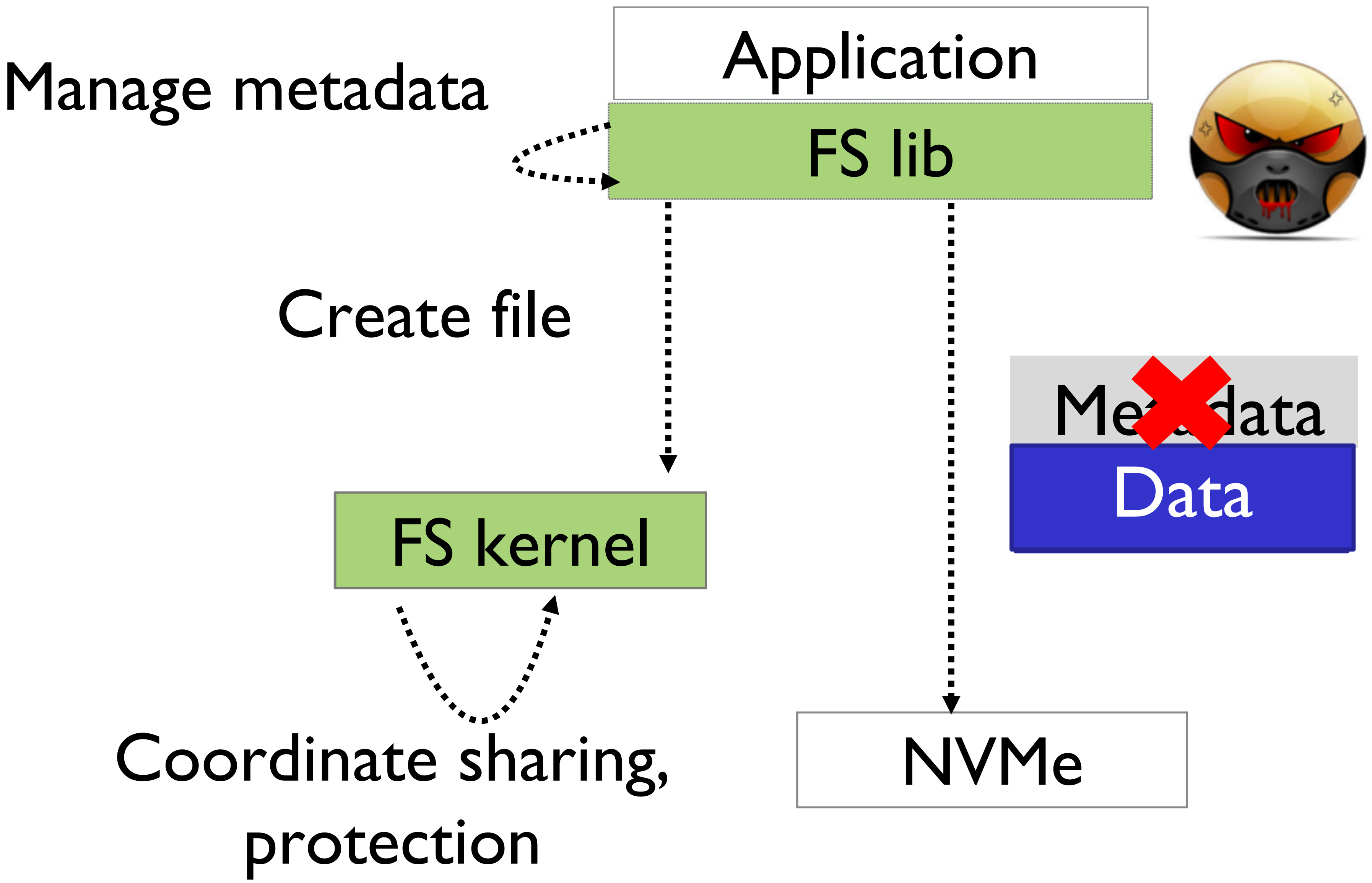
Hybrid User-level FS Integrity Problem



Direct-access for the data-plane

Arrakis (OSDI '14), Strata (SOSP '17)

Hybrid User-level FS Integrity Problem



Untrusted (buggy or malicious)

Can compromise metadata integrity and impact crash consistency

Data plane security compromised

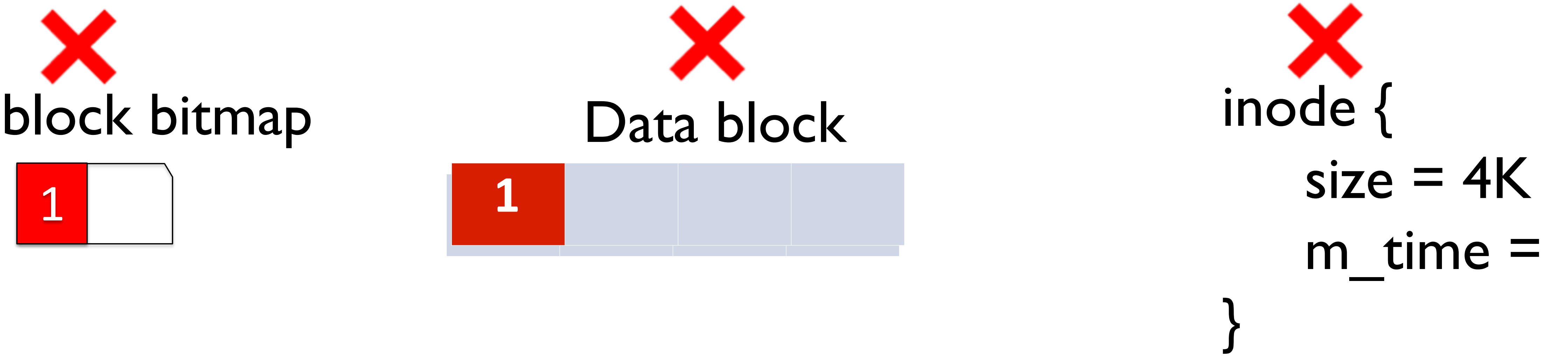
Concurrent Access?

App. 1
FS lib
Append(F1, buff, 4k)

Set bitmap
Append
Update inode

App. 2
FS lib
Append(F1, buff, 4k)

Skip locking
Set bitmap
Append
Update inode



Arrakis and Strata trap into OS for data-plane and control plane – No direct access

Approaches Summary

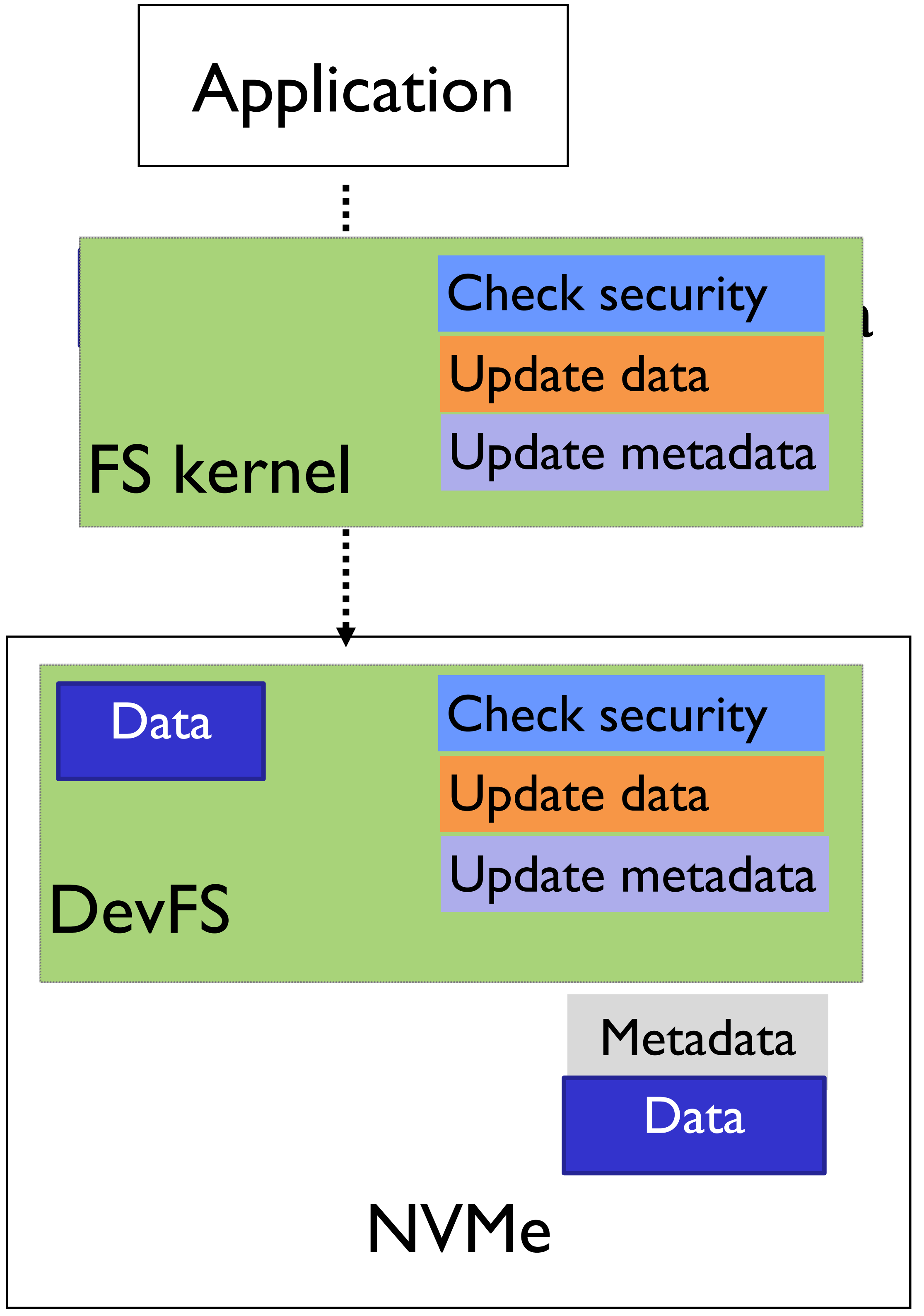
Class	File System	Integrity	Crash Consistency	Security	Concurrency	POSIX support	Direct-access
Kernel-level FS	NOVA	✓	✓	✓	✓	✓	✗
Hybrid user-level FS	Arrakis	✗	✓	✗	✗	✓	✗
	Strata	✓	✓	✗	✓	✓	✗
Microkernel	Aerie	✓	✓	✓	✓	✓	✗
Hybrid-device FS	Moneta-D	✗	✓	✓	✓	✓	✗
	TxDev	✗	✓	✓	✓	✓	✗
FUSE	Ext4-FUSE	✓	✓	✓	✓	✓	✗
Device FS	DevFS	✓	✓	✓	✓	✓	✓

Outline

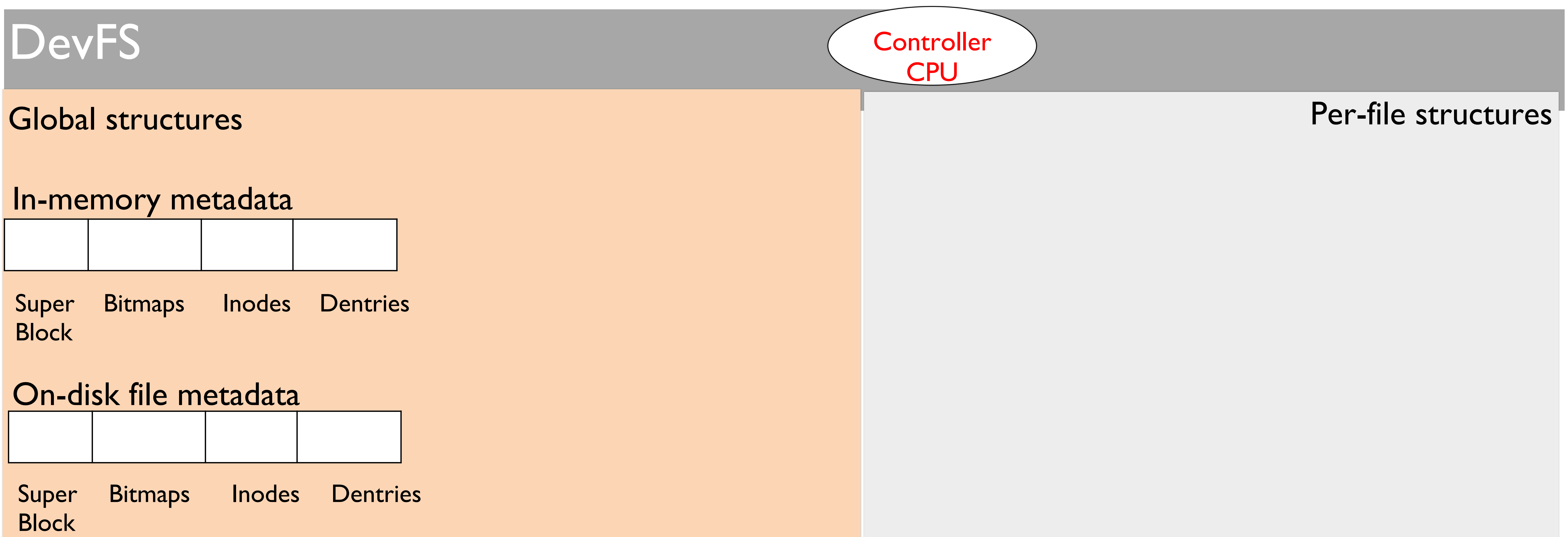
- Introduction
- Background
- Motivation
- DevFS Design
- Evaluation
- Conclusion

Device-level File System (DevFS)

- Move file system into the device hardware
- Use device-level CPU and memory for DevFS
- Apps. bypass OS for **control and data plane**
- DevFS handles integrity, concurreny, crash-consistency, and security
- Achieves **true direct-access**

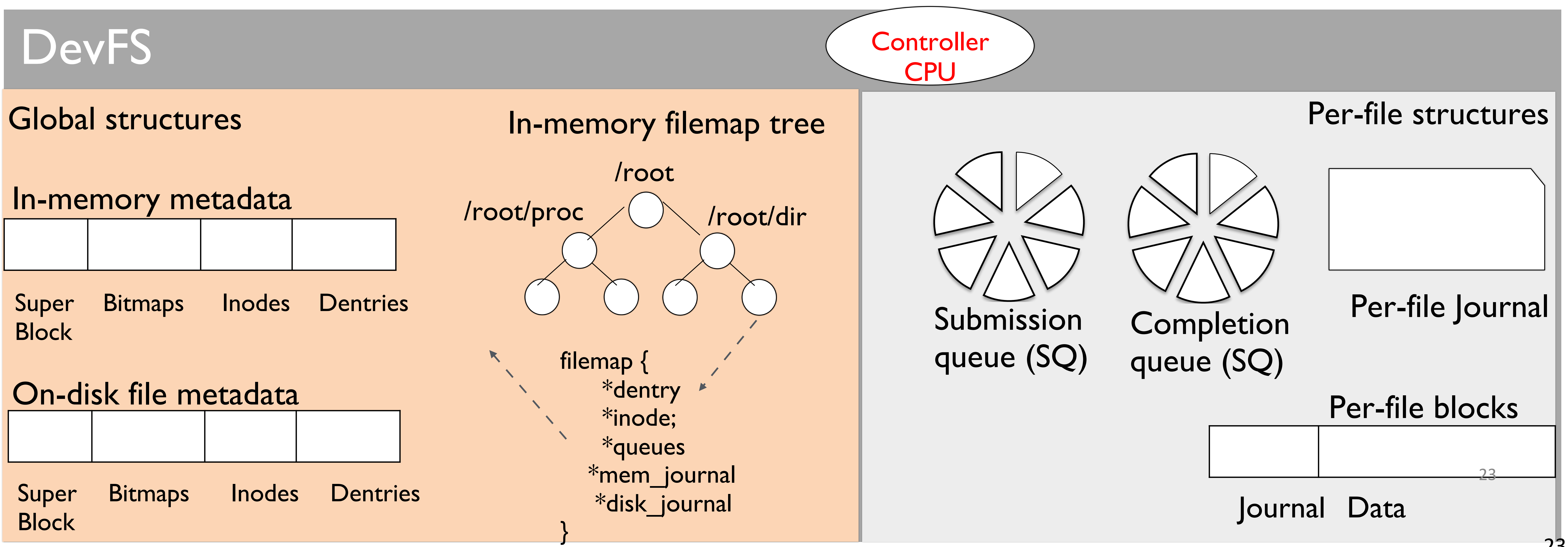


DevFS Internals

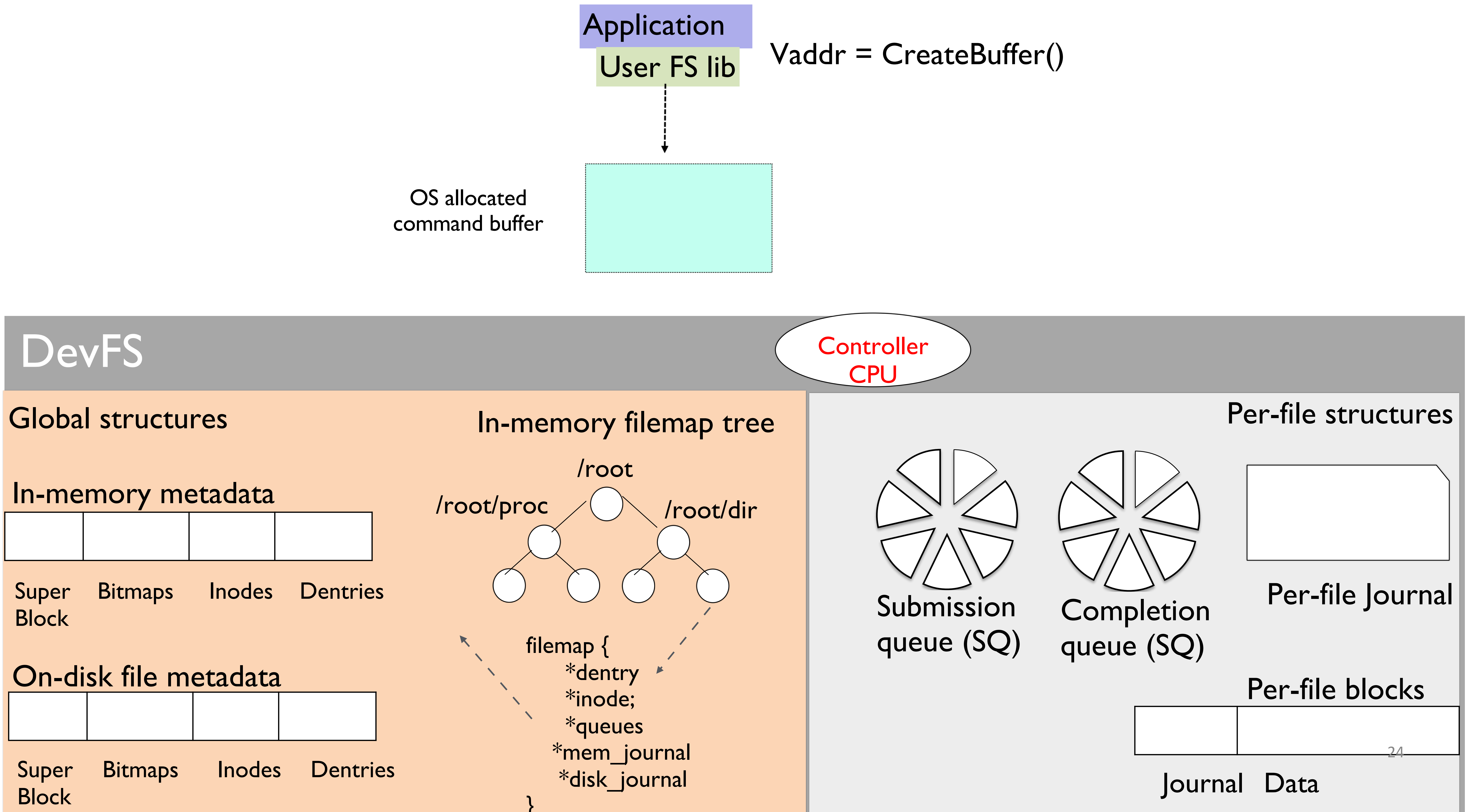


DevFS Internals

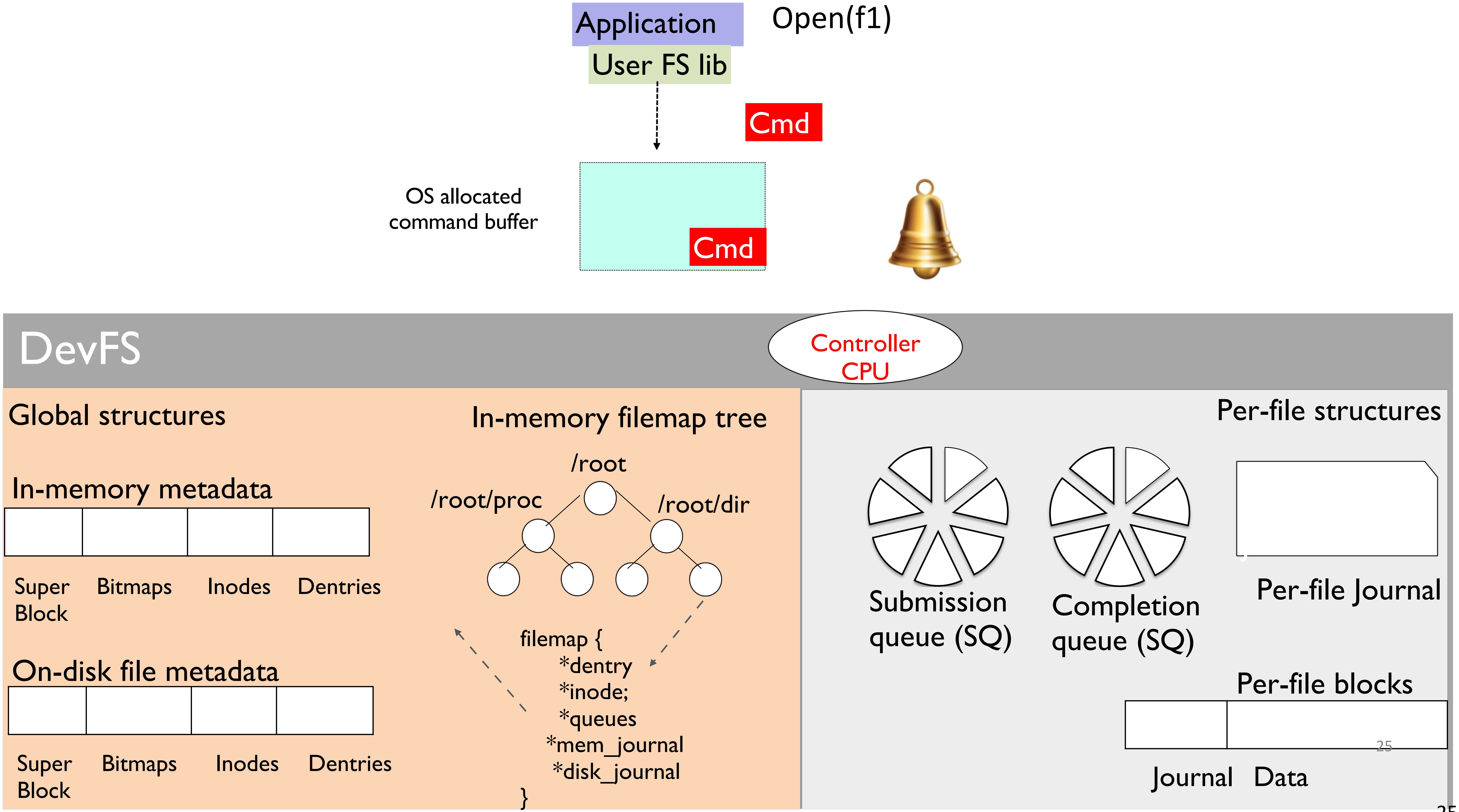
- Modern storage device contain multiple CPUs
- Support up to 64K I/O queues
- To exploit concurrency, each file has own I/O queue and journal



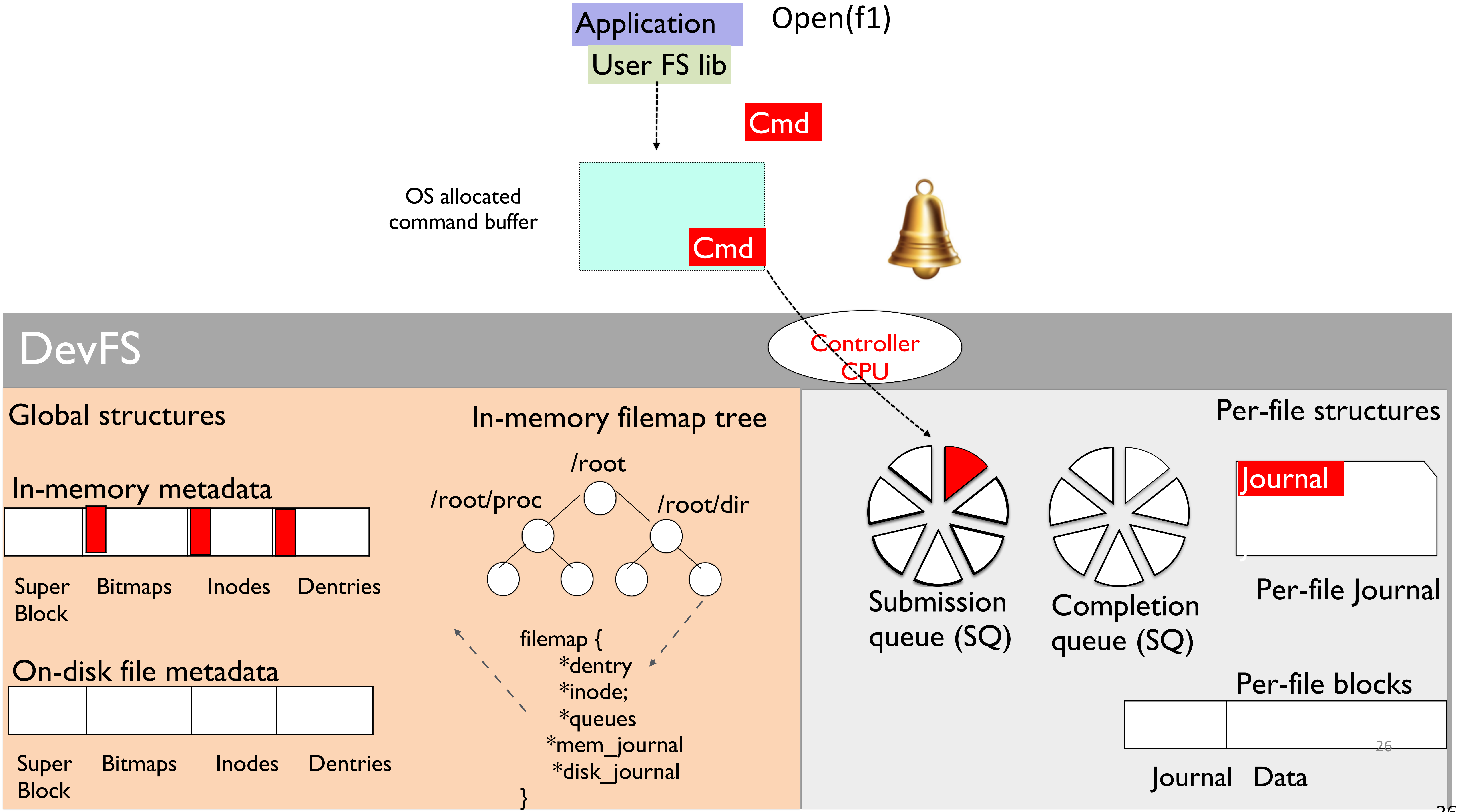
DevFS Internals



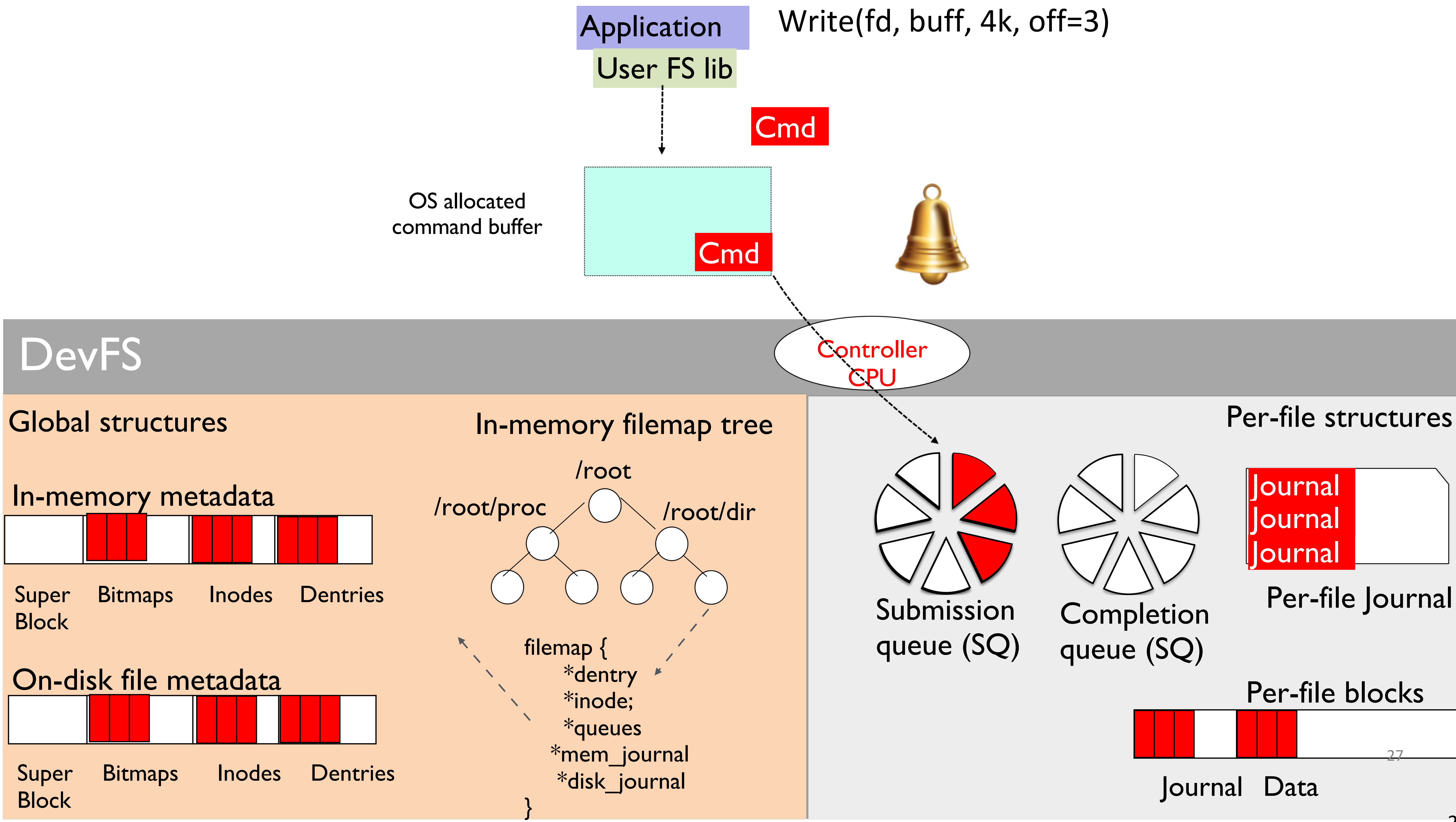
DevFS I/O Operation



DevFS I/O Operation



DevFS I/O Operation



Capacitance Benefits Inside H/W

- Writing journals to storage has high overheads
- Modern storage devices have device-level capacitors
- Capacitors safely flush memory state to storage after power failure
- DevFS uses device memory for file system state
  - Can avoid writing in-memory state to disk journal
  - Overcomes the "double writes" problem
- Capacitance support improves performance

Challenges of Hardware File System

- Limited memory inside the storage device ⟵ today's focus
  - Reverse-cache inactive file system structures to host memory
- DevFS lack visibility to OS state (e.g., process permission)
  - Make OS share required information with "down-call"
  - Please see the paper for more details

Device Memory Limitation

- Device RAM size constrained by cost ($) and power consumption
- RAM used mainly by file translation layer (FTL)
  - RAM size proportional to FTL's logical-to-physical block mapping
  - Example: 512 GB SSD uses 2 GB RAM to support translations

Unlike kernel FS, device FS footprint must be kept small

Memory Consuming File Structures

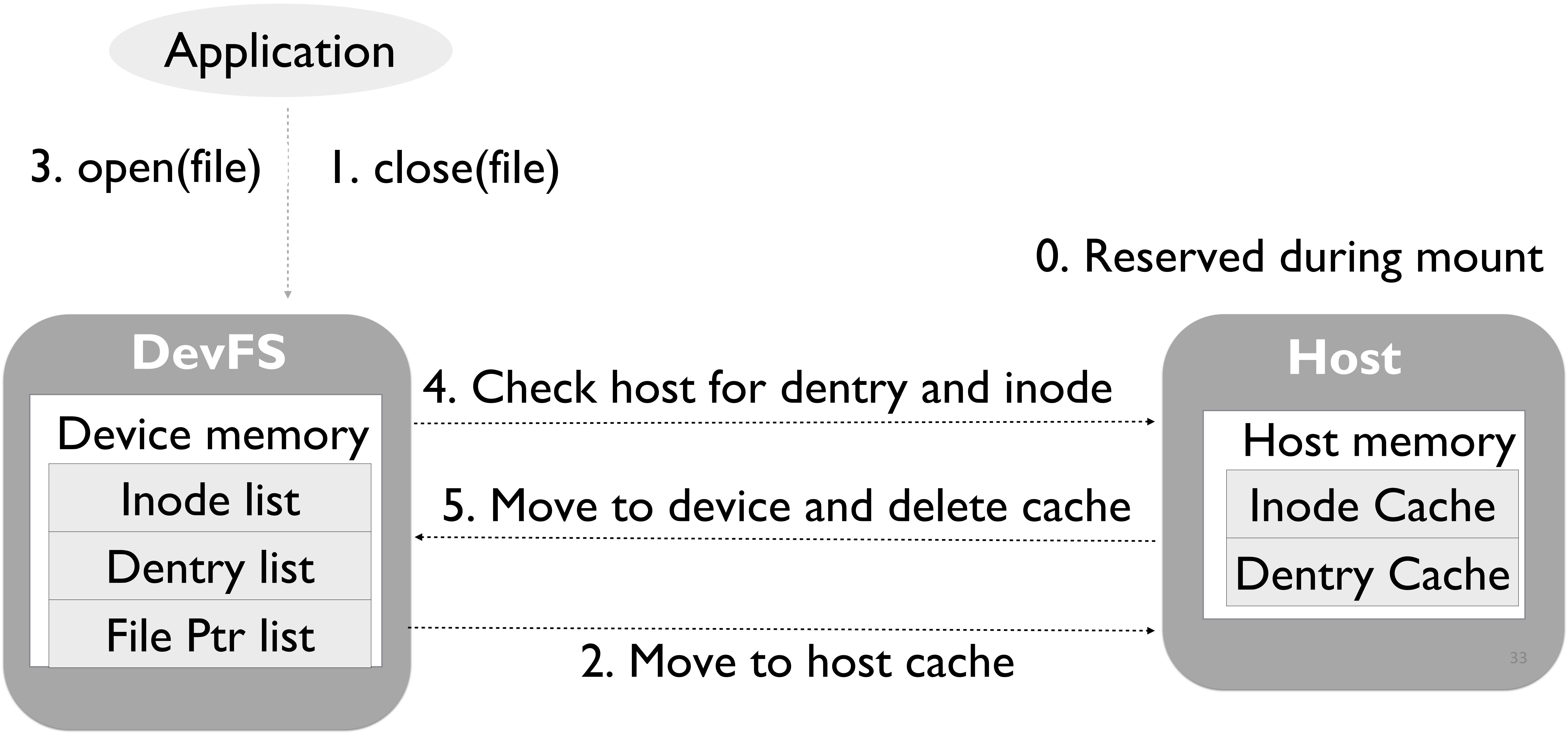
- Our analysis shows four in-memory structures using 90% of memory
  - Inode (840 bytes) - created for file open, not freed until deletion
  - Dentry (192 bytes) - created for file open, kept in a cache
  - File pointer (256 bytes) - released when file is closed
  - Others (156 bytes) - e.g., DevFS file map structure
- Simple workload - open and close 1 million files
  - DevFS memory consumption ~1.2 GB (60% of device memory)

Reducing Memory Usage

- On-demand allocation of structures
    - Structures such as filemap not used after file is closed
    - Allocated after first write and released when a file is closed
- Reverse Caching
    - Move inactive structures to host memory

Reverse-Caching to Reduce Memory

- Move inactive inode and dentry structures to host memory



Decompose FS Structures

- Reverse caching for a complicated for inode
- Inode's fields accessed even file closing (e.g., directory traversal)
- Frequently moving between host cache and device can be expensive!
- Our solution – split file system structures (e.g., inode) into a host and device structure

Decompose FS Structures

Devfs inode structure

struct devfs_inode_info {

inode_list

page_tree

journals

…….

struct inode vfs_inode

} **840 bytes**

Decomposed DevFS structure

struct devfs_inode_info {

/*always kept in device*/

struct *inode_device

/*moved to host after close*/

struct *inode_host **593 bytes**

}

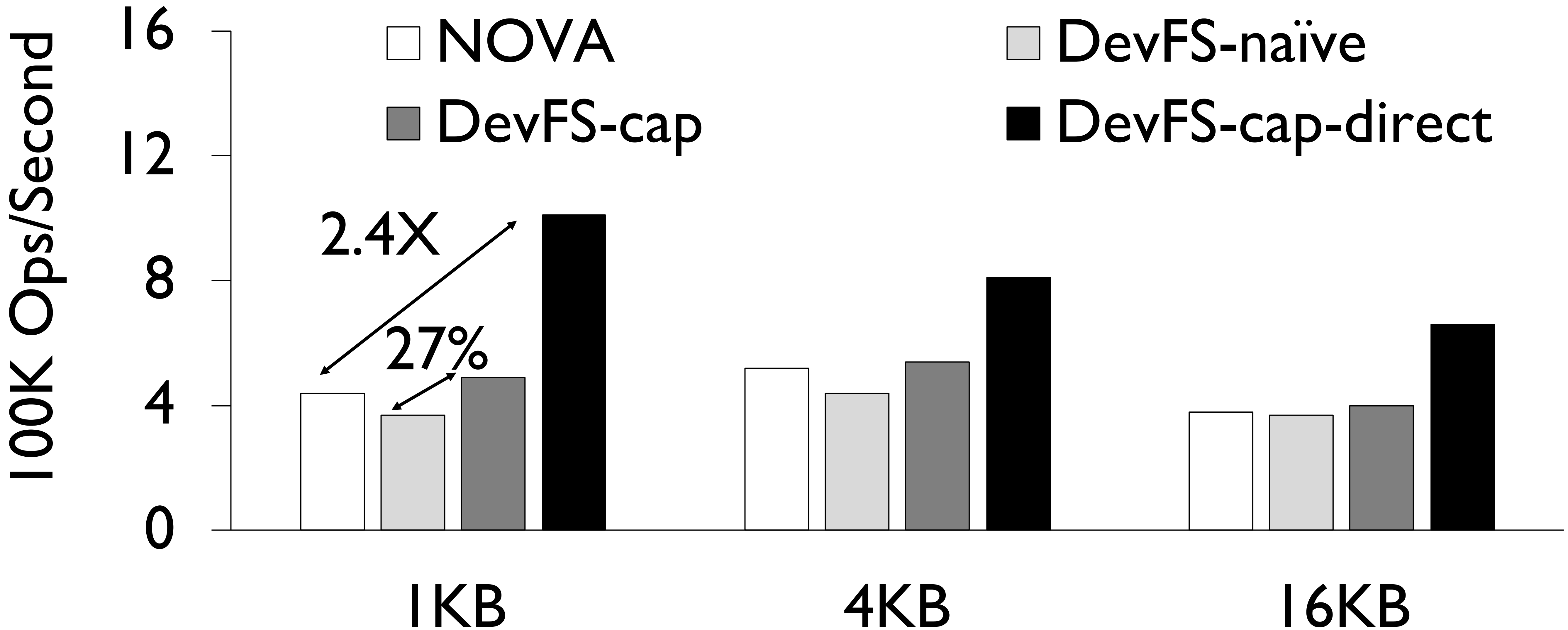
Outline

- Introduction
- Background
- Motivation
- DevFS Design
- Evaluation
- Conclusion

Evaluation

- Benchmarks and Applications
  - Filebench
  - Snappy – widely used multi-threaded file compression
- Evaluation comparison
  - NOVA – state-of-the-art in-kernel NVM file system
  - DevFS-naïve – DevFS without direct access
  - DevFS-cap – without direct access but with capacitor support
  - DevFS-cap-direct – capacitor support + direct access
- For direct-access, benchmark and applications run as driver

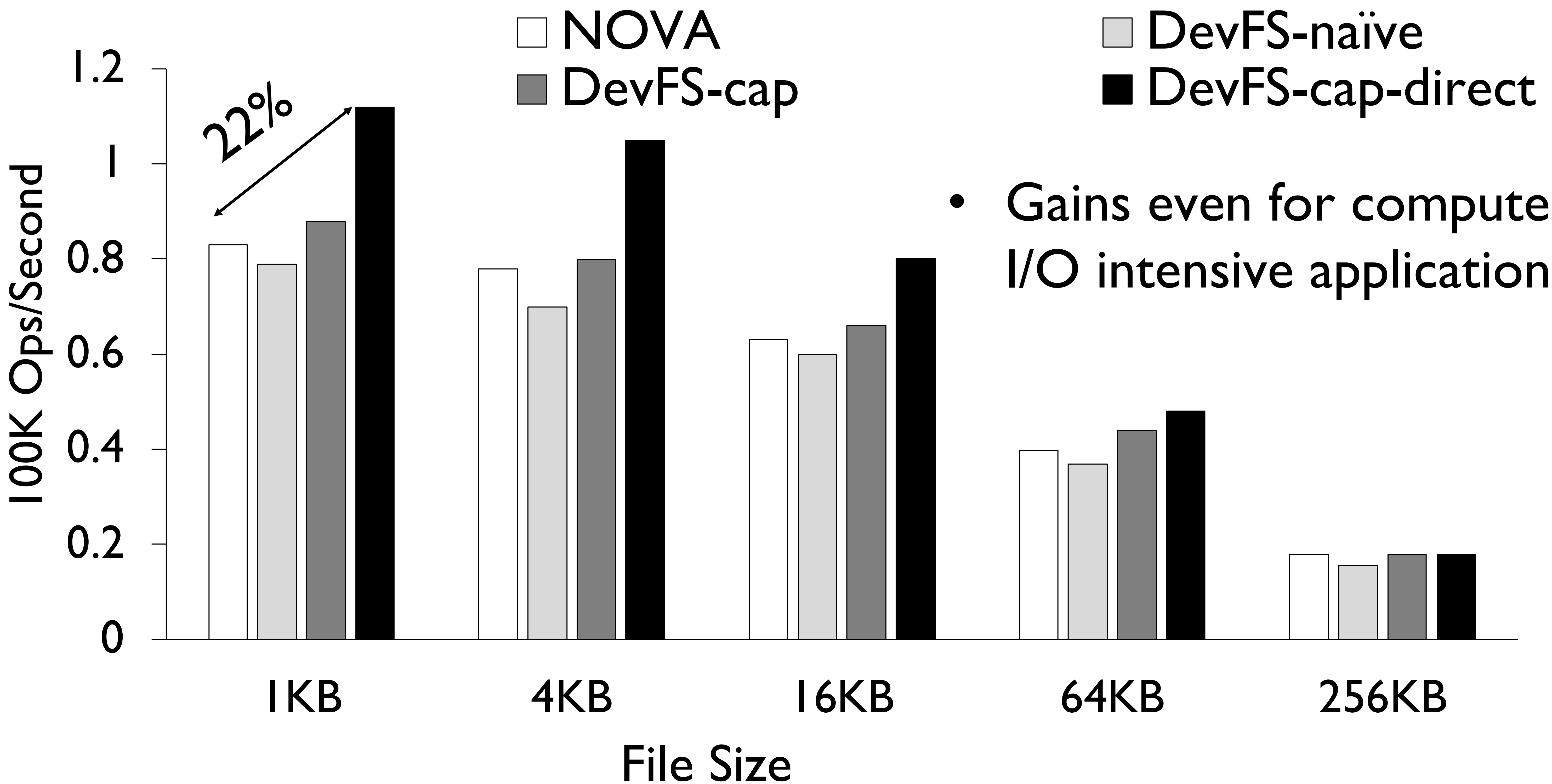
Filebench - Random Write



- DevFS-naïve suffers from high journaling overhead
- DevFS-cap uses capacitors to avoid on-disk journaling
- DevFS-cap-direct achieves true direct-access bypassing OS

Snappy Compression Performance

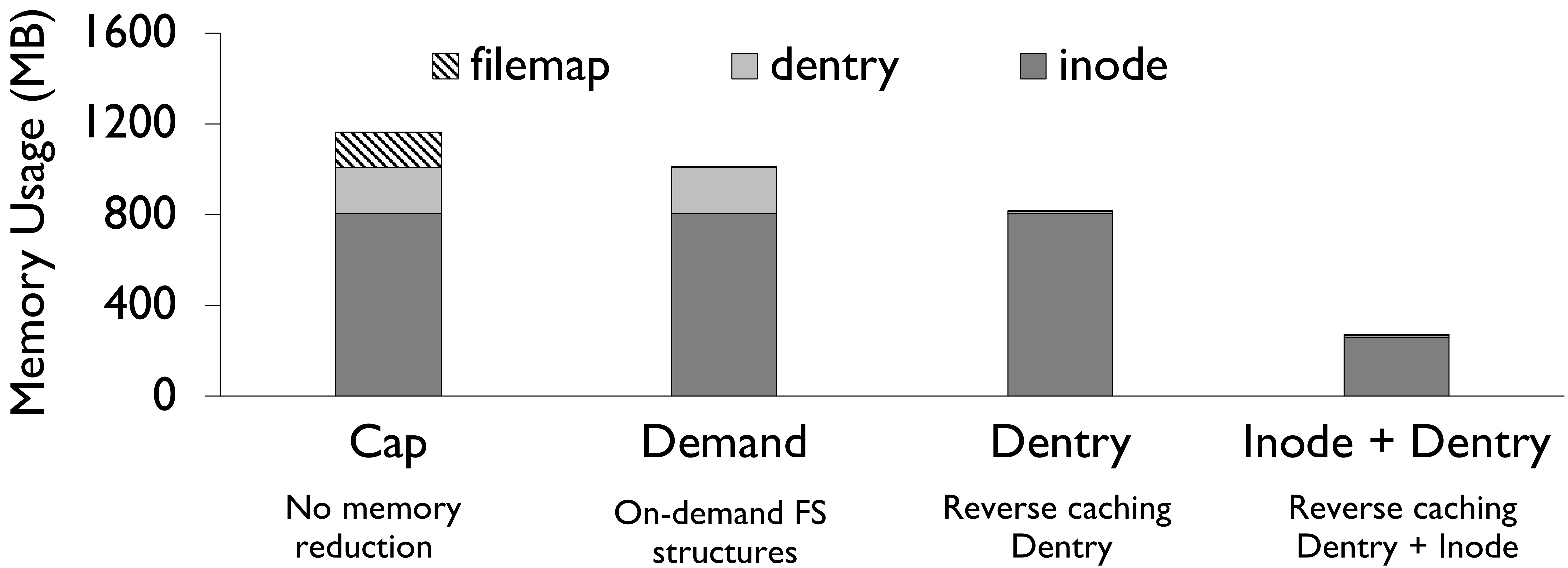
Read a file ⟹ Compress ⟹ Write output ⟹ Sync file



- Gains even for compute + I/O intensive application

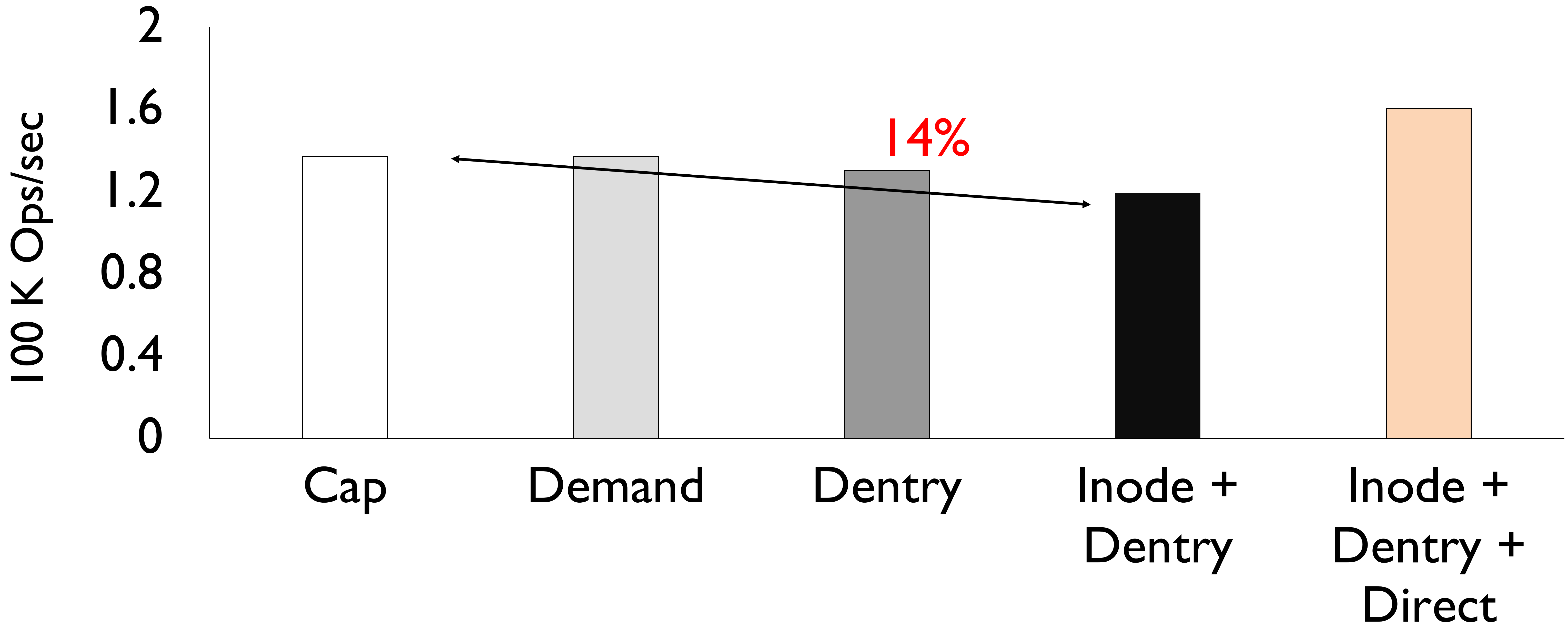
Memory Reduction Benefits

- Filebench – File Create workload (Create 1M files and close files)



- Demand allocation reduces memory consumption by 156MB (14%)
- Inode and Dentry reverse caching reduces memory by 5X

Memory Reduction Performance Impact



- Dentry and Inode reverse caching overhead less than 14%
- Overhead mainly due to structure movement cost

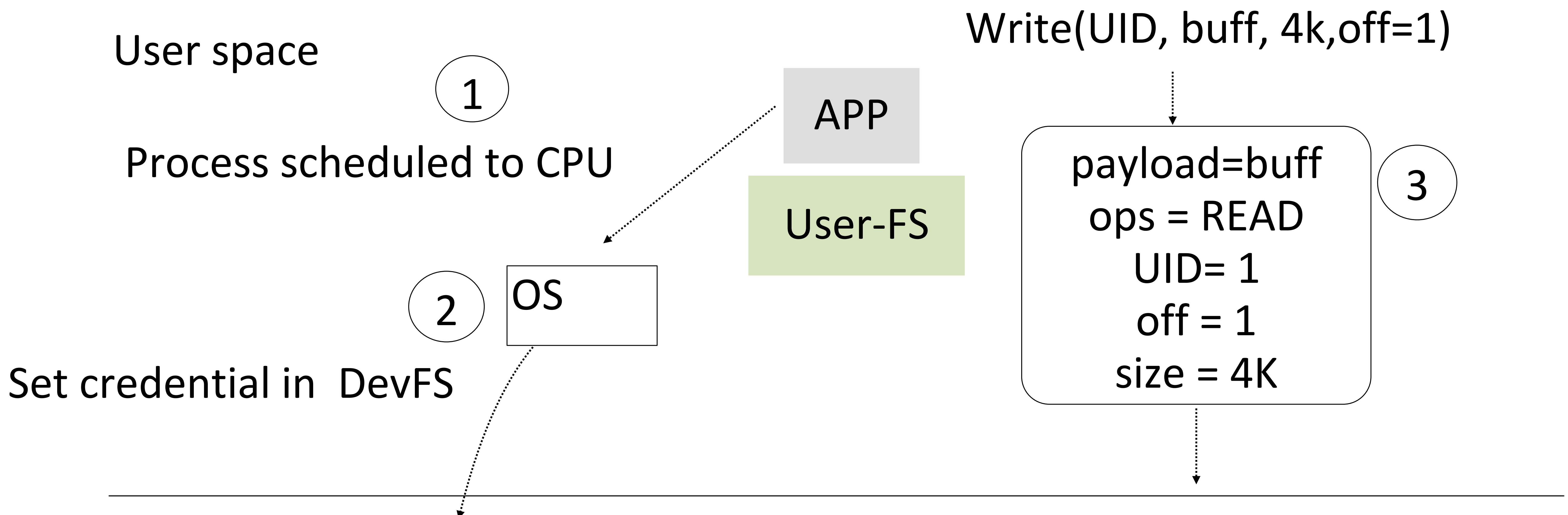
Summary

- Motivation
  - Eliminating OS overhead and providing direct access is critical
  - Hybrid user-level file systems compromise fundamental properties
- Solution
  - We design DevFS that moves FS into the storage H/W
  - Provides direct-access without compromising FS properties
  - To reduce memory footprint of DevFS designs reverse-caching
- Evaluation
  - Emulated DevFS shows up to 2X I/O performance gains
  - Reduces memory usage by 5X with 14% performance impact

Conclusion

- We are moving towards a storage era with microsecond latency
- Eliminating software (OS) overhead is critical
  - But without compromising fundamental storage properties
- Near-hardware access latency requires embedding S/W into H/W
- We take first step towards moving file system in H/W
- Several challenges such as H/W integration, support for RAID, snapshots, and deduplication yet to be addressed

Permission Checking



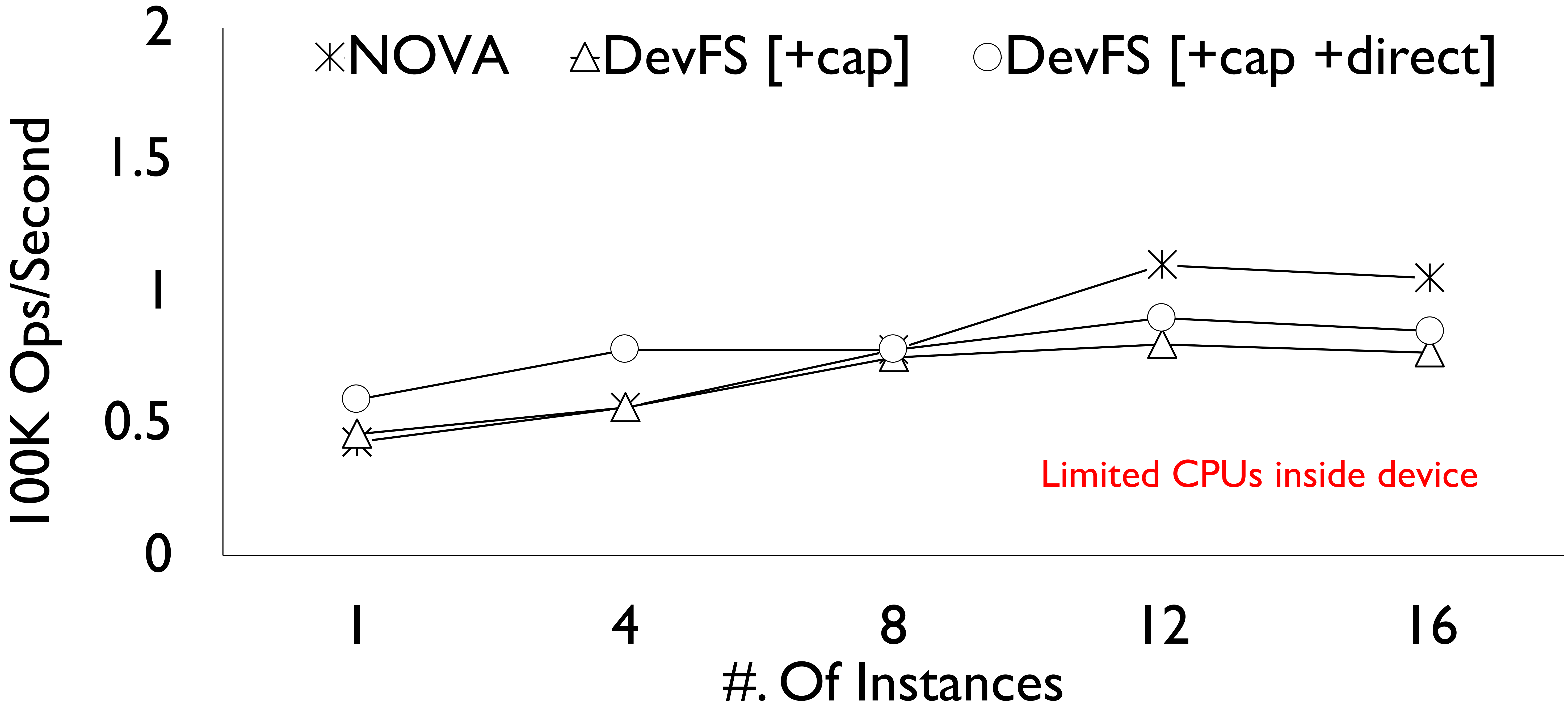
DevFS credential region

Host CPU	Credentials
0	Task1.cred
1	Task1.cred
…	…
24	Task2.cred

Permission manager (4)

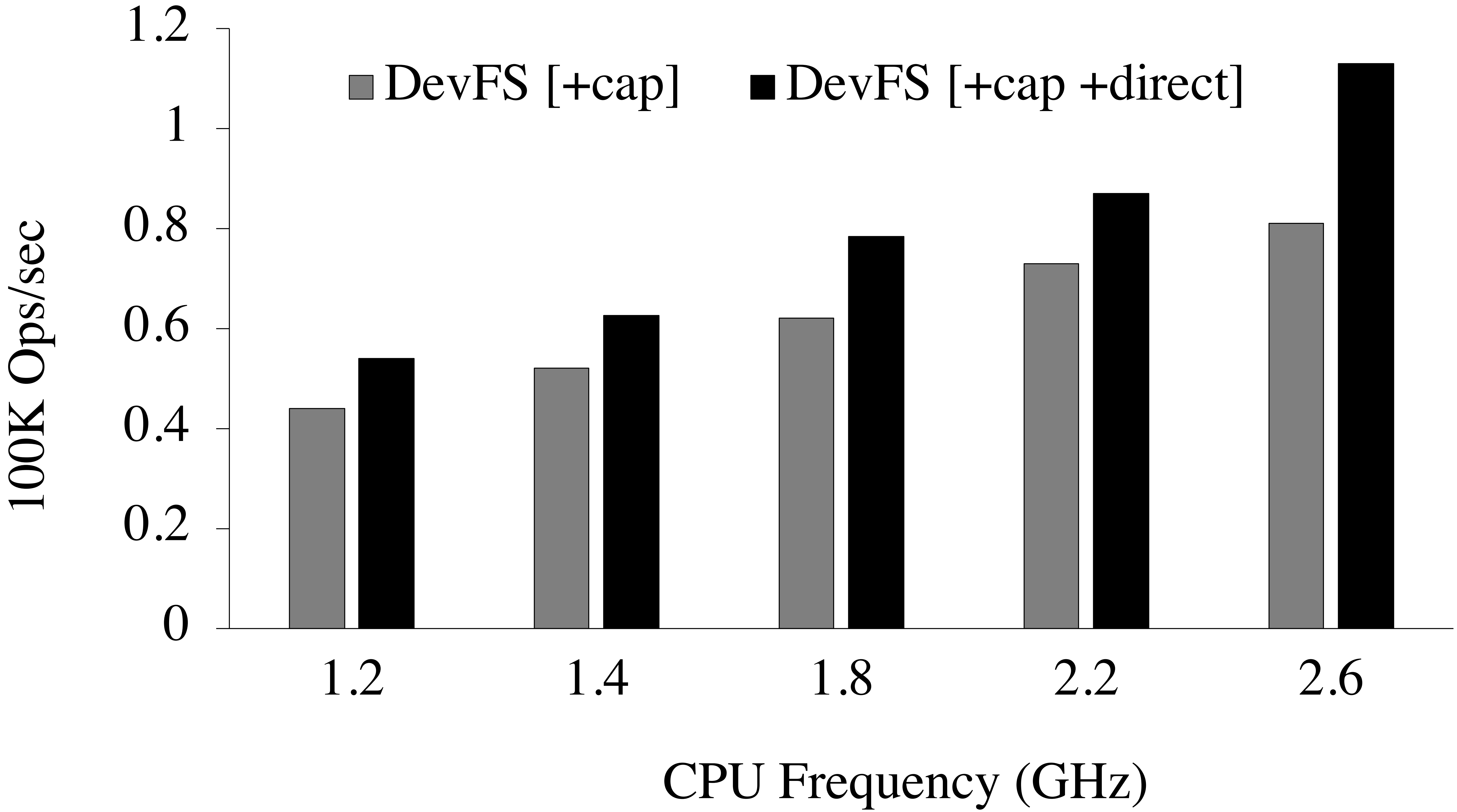
t_cred = get_task_cred(CPUID)
inode_cred= get_inode_cred(fd)
compare_cred(t_cred, inode_cred)

Concurrent Access



- DevFS uses only 4 device CPU
- Limited device CPUs restricts DevFS scaling

Slow CPU Impact – Snappy 4KB



Thanks!

Questions?