PERSISTENT MEMORY + SUMMIT 2022 COMPUTATIONAL STORAGE
VIRTUAL EVENT • MAY 24-25, 2022

Organic Redesign of Abstractions for Computational Storage Devices using CISCOps

Jian Zhang, Yujie Ren, **Sudarsun Kannan**
Rutgers University

Storage Hardware and Software Trends

- Hardware trend: fast microseconds latency devices with increasing in-storage compute capabilities
- Software trend: fast user-level file systems to bypass the OS for reducing software overheads ("boundary crossing")
- Unfortunately, dominating I/O overheads like data copy, system calls, PCI communication costs remain
- Lack of organic support for leveraging in-storage compute for I/O and data processing operations and reducing I/O overheads

Evolving Storage with Fast Compute

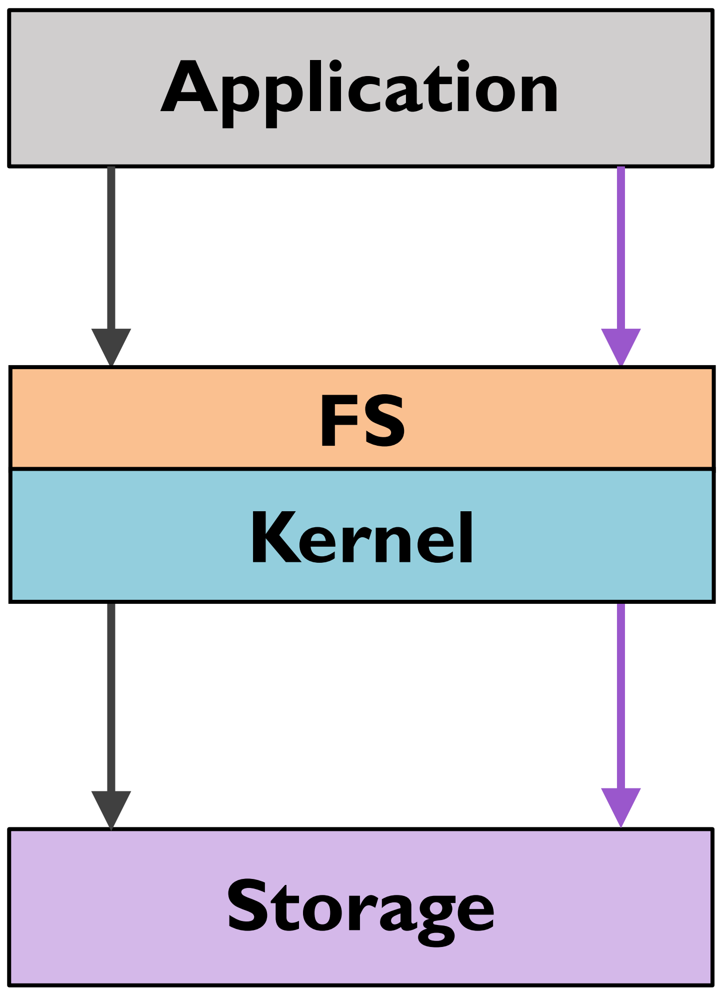
	Intel X25M	Samsung 840	Samsung 970	Samsung PM1743
Year:	2008	2013	2018	2022
Interface:	SATA 3.0	SATA 3.0	PCIe 4.0	PCIe 5.0
CPU:	2-core	3-core	5-core	> 8 cores *
RAM:	128MB DDR2	512MB LPDDR2	1GB LPDDR4	> 2GB LPDDR4 *
B/W:	250 MB/s	500 MB/s	3300 MB/s	6600 MB/s
Latency:	~70μs	~60μs	~40μs	~20μs

* Speculated specs

In-storage compute is becoming powerful!

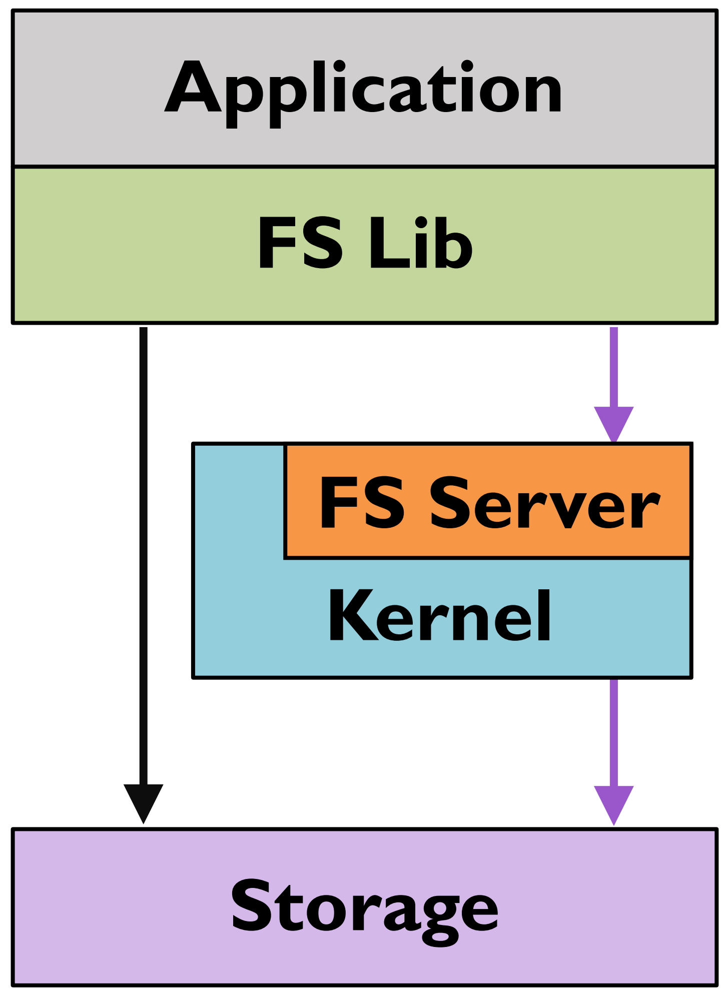
State-of-the-art Designs

KernelFS



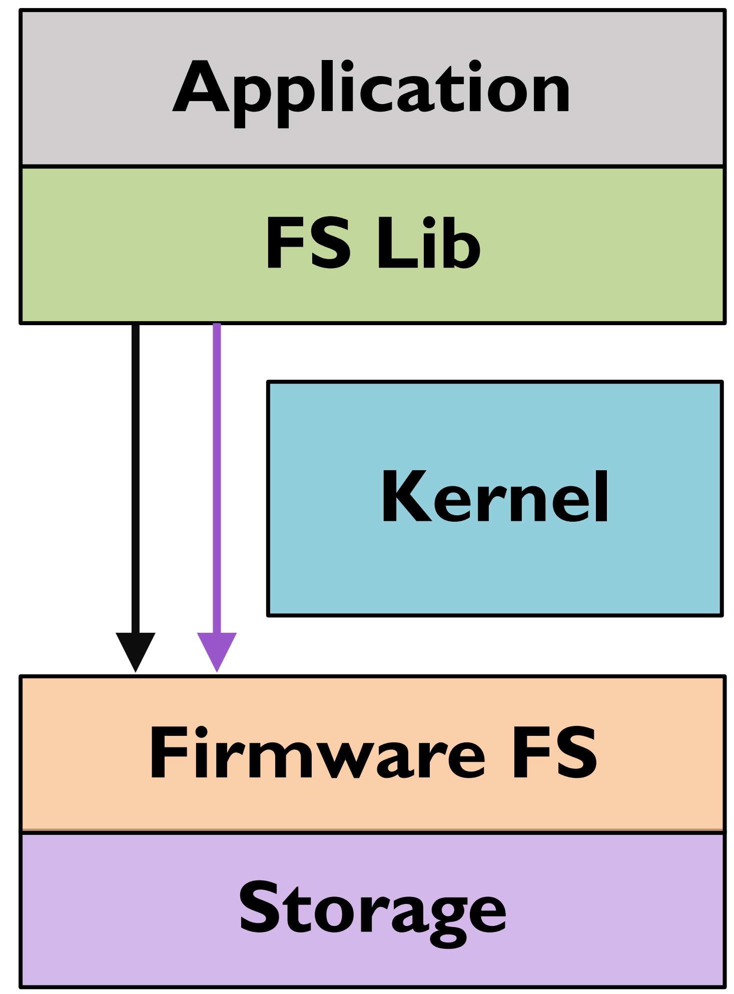
ext4-DAX
F2FS (FAST '15)
NOVA (FAST '16)

UserFS



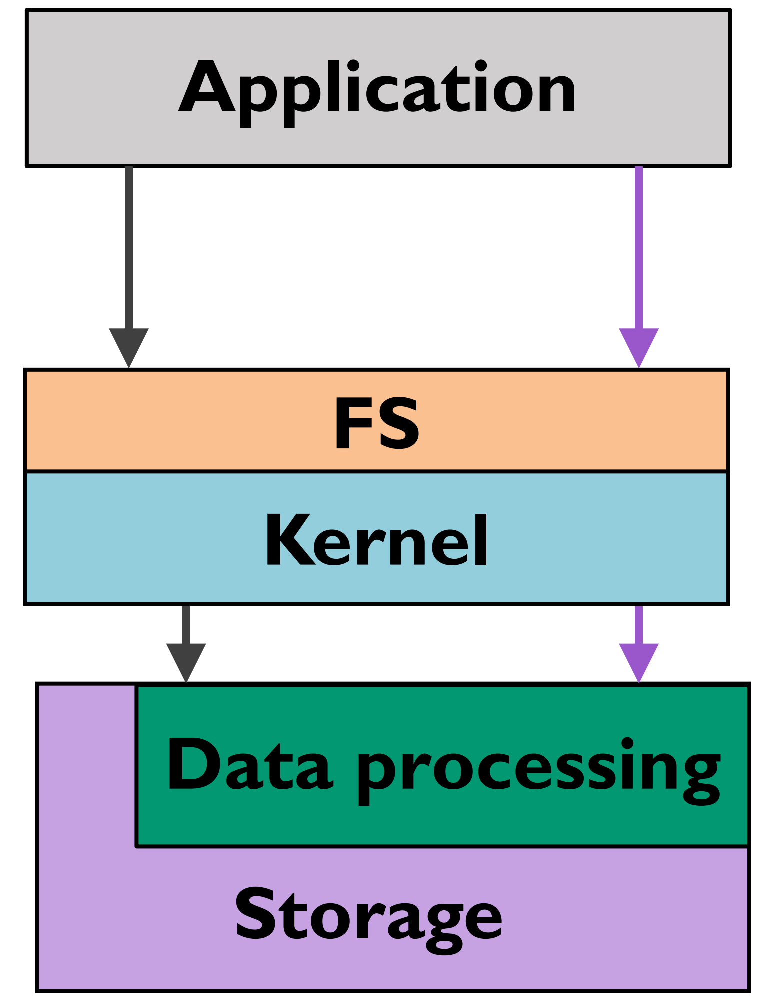
Strata (SOSP '17)
SplitFS (SOSP '19)
FSP (SOSP '21)

DeviceFS



DevFS (FAST' 18)
Insider (ATC '19)
CrossFS (OSDI '20)

Compute Offloading



PolarDB (FAST '20)
Newport CSD
ScaleFlux CSD

→ : data-plane ops   → : control-plane ops

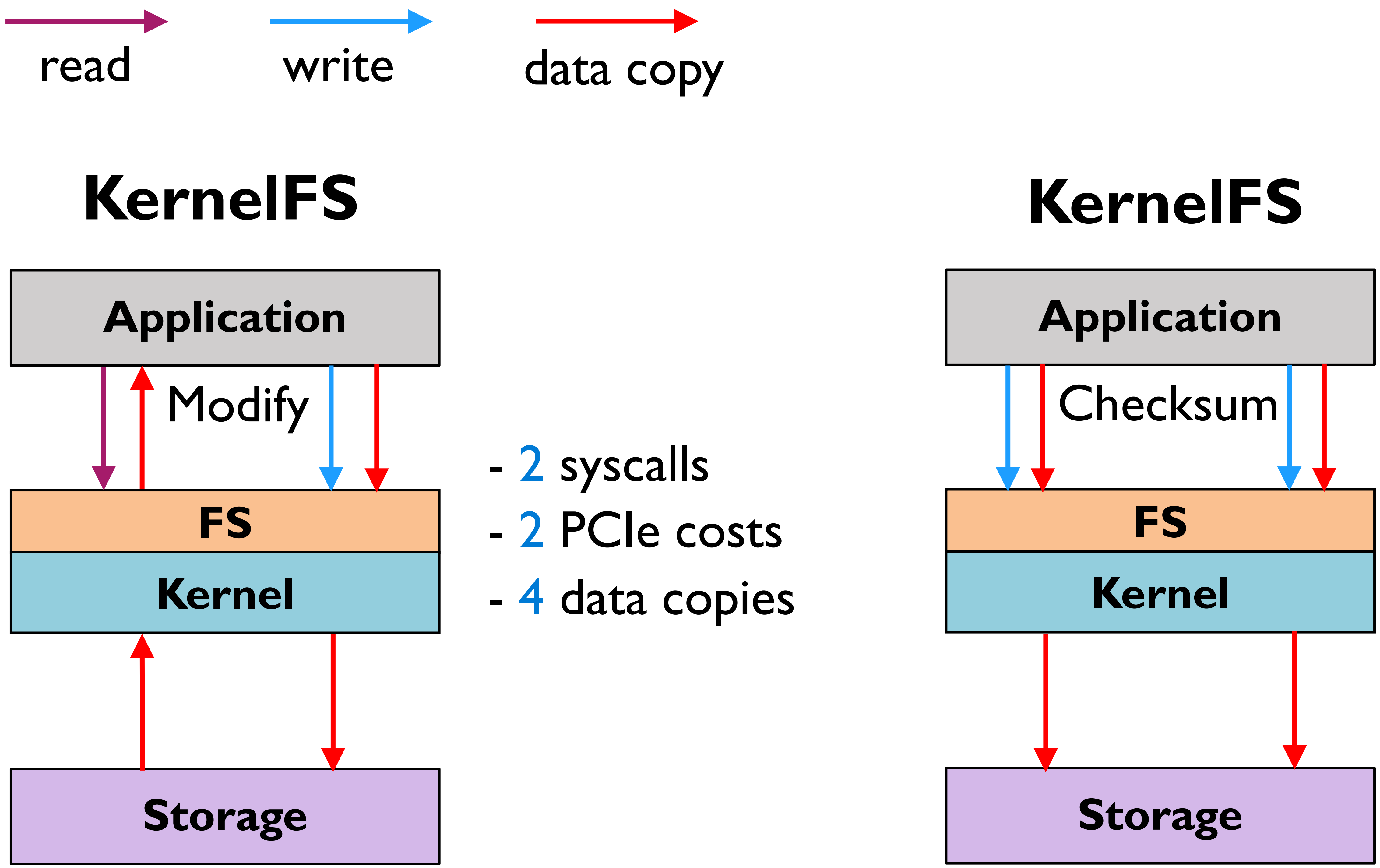
Common I/O Sequences in Applications

- Simple I/O operations to store or read state (e.g., *write*, *read*)
- Sequence of I/O operations (e.g., *open-read-write-close* in file servers)
- Operations coupled with data processing (e.g., *append-checksum-write* in key-value stores)
- Reducing I/O overheads, such as data copy, PCIe costs, and syscalls, across all I/O sequences is critical.

Outline

- Background
- Motivation
- Design
- Evaluation
- Conclusion

Dominant I/O Overheads



- 2 syscalls
- 2 PCIe costs
- 4 data copies

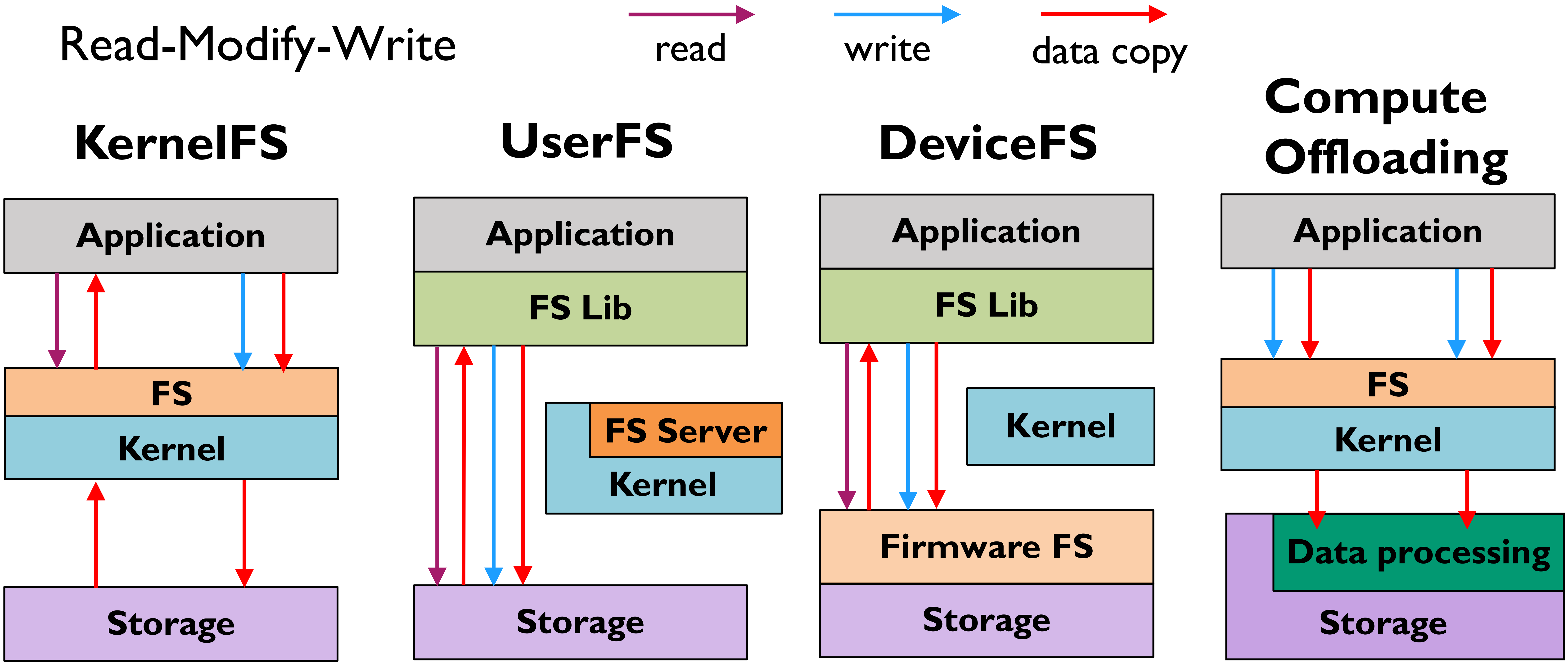
Read-Modify-Write

- 2 syscalls
- 2 PCIe costs
- 4 data copies
- Processing in Host

Append-Checksum-Write

Dominant I/O Overheads

Read-Modify-Write



KernelFS:
2 syscalls
2 PCIe costs
4 data copies

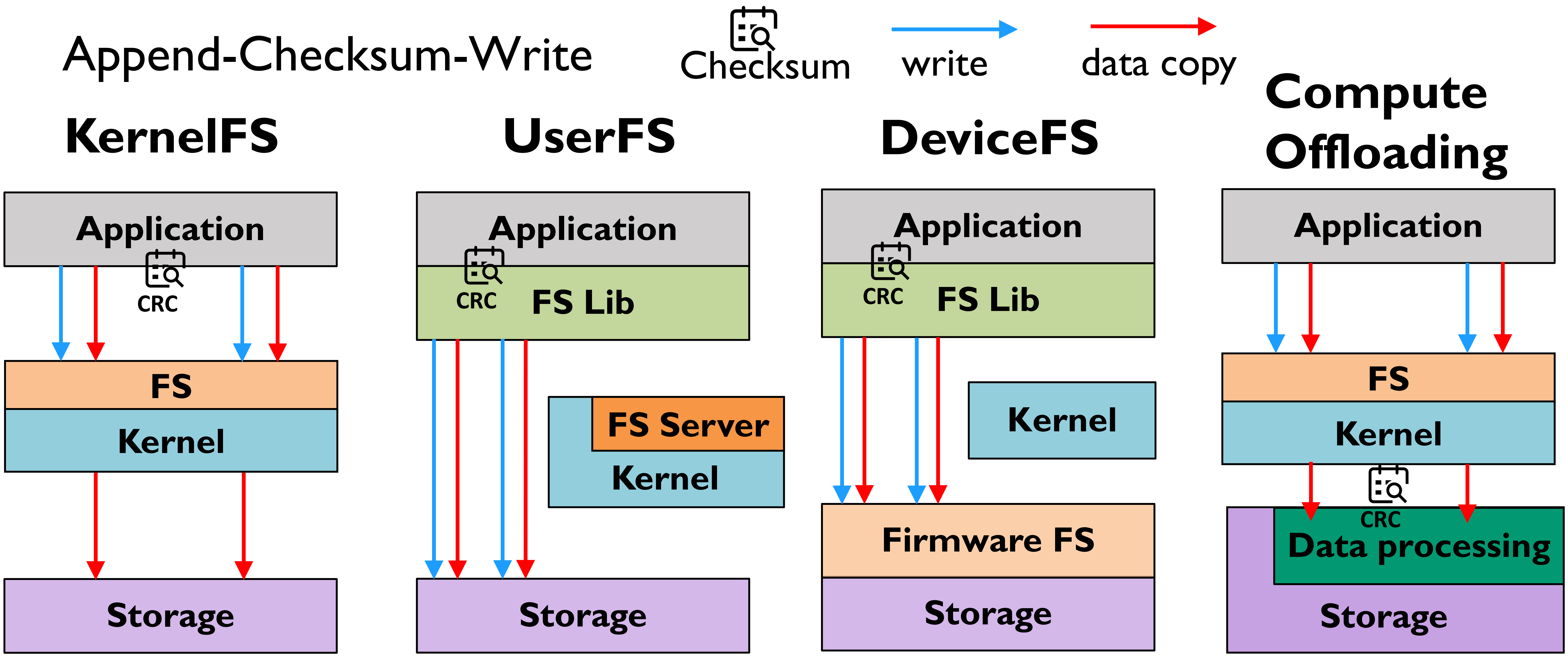
UserFS:
2 PCIe cost
2 data copies

DeviceFS:
2 PCIe cost
2 data copies

Compute Offloading:
2 syscalls
2 PCIe cost
2 data copies

Dominant I/O Overheads

Append-Checksum-Write



KernelFS
2 syscalls
2 PCIe costs
4 data copies
Processing in Host

UserFS
2 PCIe costs
2 data copies
Processing in Host

DeviceFS
2 PCIe costs
2 data copies
Processing in Host

Compute Offloading
2 syscalls, 2 PCIe costs
2 data copies
Processing in Storage

Storage Approaches Summary

Properties	KernelFS	UserFS	DeviceFS	Compute offload	**FusionFS**
Direct-access	✗	●	✓	✗	✓
Reduce data copy	✗	●	●	●	✓
Reduce PCIe cost	✗	✗	✗	●	✓
In-storage management	✗	✗	✓	✗	✓
In-storage processing	✗	✗	✗	✓	✓
Durability	Data	Data	Data	Data	Data & Compute
Resource management	✓	✗	✗	✗	✓
Security	✓	●	✓	●	✓

✓ Satisfy ● Partially satisfy ✗ Not satisfy

Outline

- Background
- Motivation
- Design
- Evaluation
- Conclusion

Our Solution: FusionFS

- FusionFS aggregates I/O and data processing sequences into $CISC_{Ops}$ **(Inspiration: CISC ISAs)**

- To reduce I/O overheads, FusionFS offloads **$CISC_{Ops}$** to storage

- Manages and provides fairness of in-storage resources through CFS

- Exploits storage compute for fine-grained crash consistency and faster recovery

RISC vs CISC

- Two widely used ISAs: **RISC and CISC**

- Reduced instruction set computer (RISC)
  - More instructions
  - Each instruction takes one cycle time
  - More complex compiler

- Complex instruction set computer (CISC)
  - Fewer and **richer** instructions **composed of simple instructions**
  - Each instruction takes a longer amount of cycle time
  - More complex hardware logic

Everlasting Debate

Power Struggles: Revisiting the RISC vs. CISC Debate on Contemporary ARM and x86 Architectures

Emily Blem, Jaikrishnan Menon, and Karthikeyan Sankaralingam
University of Wisconsin - Madison
{blem,menon,karu}@cs.wisc.edu

Abstrac

RISC vs. CISC wars raged in the processor design complexity were tl desktops and servers exclusively dom scape. Today, energy and power ai straints and the computing landscap growth in tablets and smartphones r is surpassing that of desktops and la ISA). Further, the traditionally low- ing the high-performance server mai high-performance x86 ISA is enterin; vice market. Thus, the question of wh

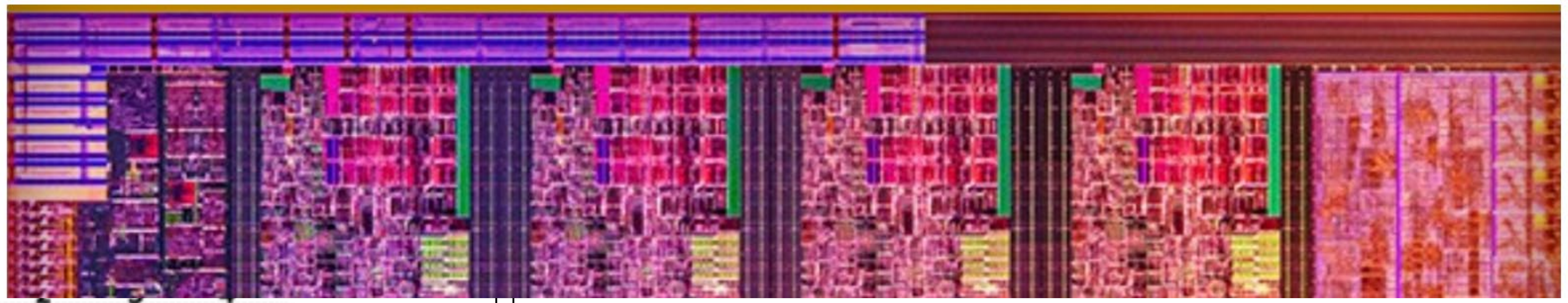
RISC ARCHITECTURE

The simplest way to examine the advantages and by contrasting it with it's predecessor: CISC (Com architecture.

Multiplying Two Numbers in Memory
On the right is a diagram representing the storage scheme for a generic computer. The

RISC vs. CISC Is the Wrong Lens for Comparing Modern x86, ARM CPUs

By Joel Hruska on December 29, 2021 at 3:15 pm | Comments



FEATURES —

RISC vs. CISC: the Post-RISC Era: A historical approach to the debate

Ars takes a look at the RISC vs. CISC debate in the post-RISC era.

JON STOKES - 10/1/1999, 2:00 PM

0

Framing the Debate

The majority of today's processors can't rightfully be called completely RISC or completely CISC. The two textbook architectures have evolved towards each other to such an extent that there's no longer a clear distinction between their respective approaches to increasing performance and efficiency. To be specific

Our Goal

Explore RISC and richer CISC-styled I/O and data processing operations to reduce dominant overheads

FusionFS: RISC vs CISC operations

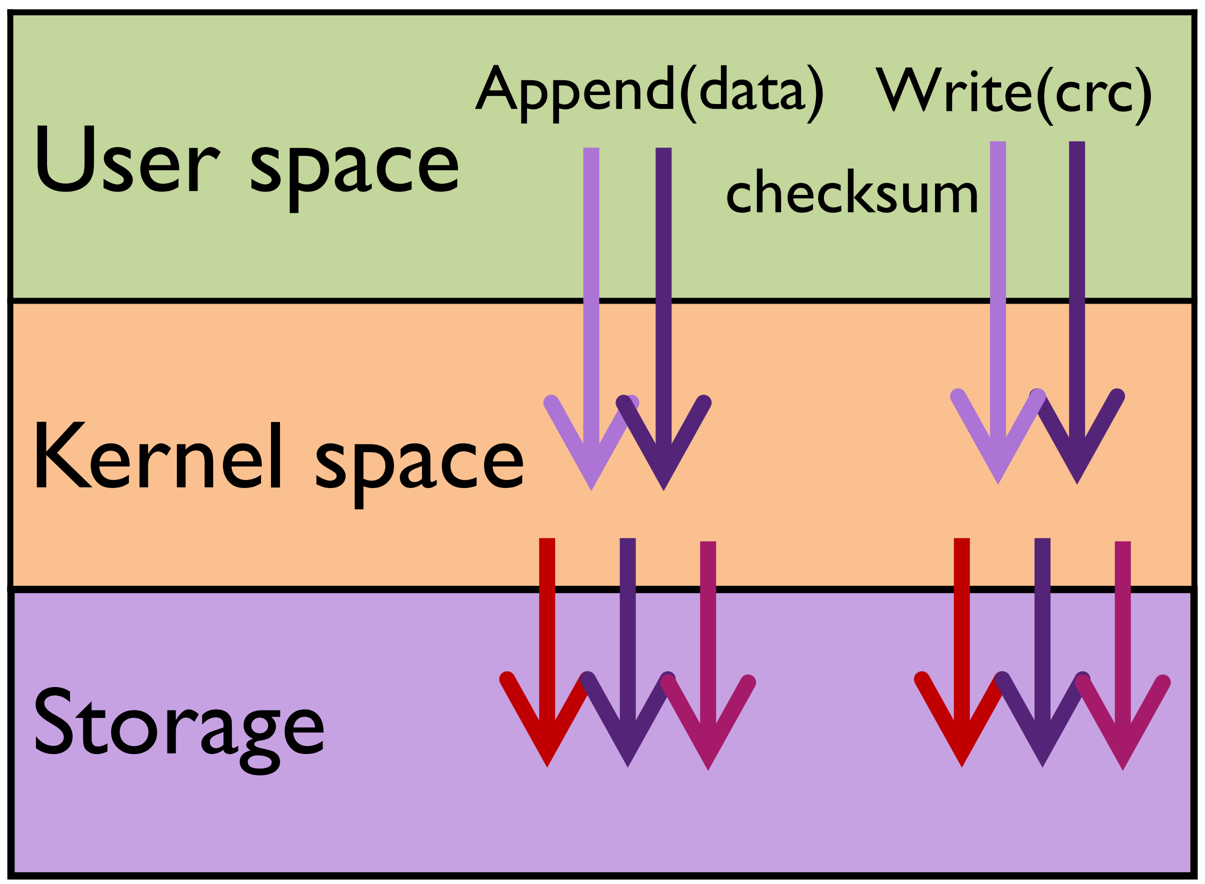
- RISC operations are simple POSIX I/O (e.g., read, write, close)
- CISC operations (**CISC$_{Ops}$**) are aggregated I/O and data processing operations (e.g., *append-checksum-write, open-read-write-close*)
- Unlike POSIX I/O vectors, CISC$_{Ops}$ combines identical and non-identical I/O and processing operations
- We offload RISC and CISC operations to an in-storage file system (we also study **CISC$_{Ops}$** for traditional kernel file systems)
- CISC$_{Ops}$ can significantly reduce dominant I/O overheads

FusionFS: CISC Operations

Append-Checksum-Write

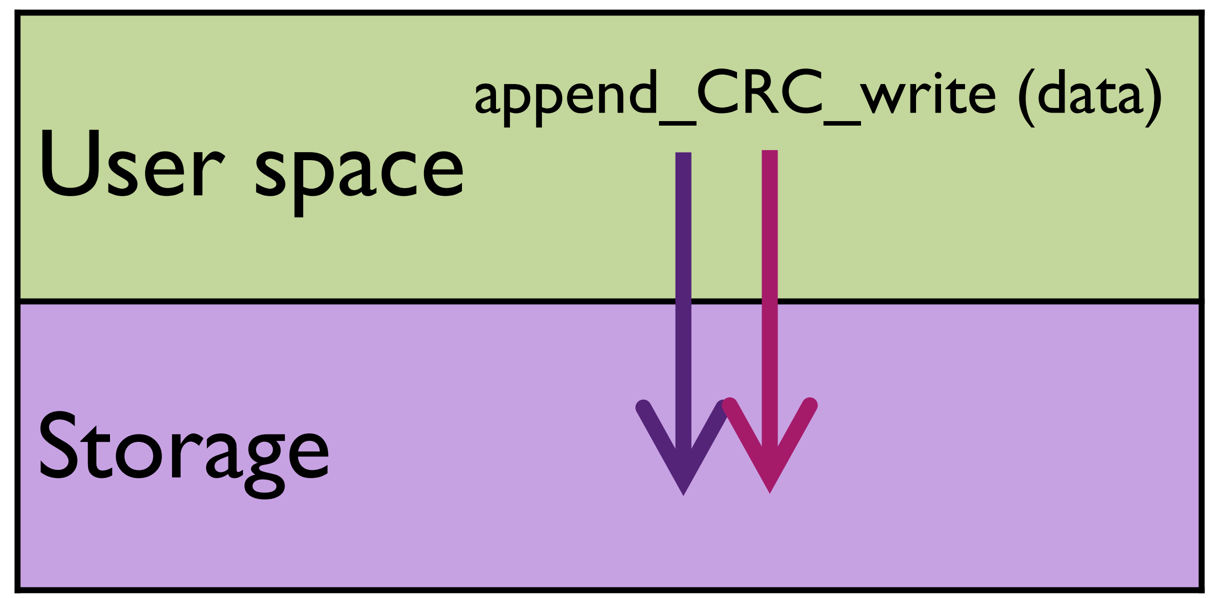
: Kernel Trap
: Metadata Copy
: Data Copy
: PCIe Cost

KernelFS Path:



2 syscalls + 4 data copies
2 metadata copies + 2 PCIe costs

CISCops Path:



Only 1 data copy and 1 PCIe access with direct access and offload computing

CISCops reduces data copy, syscalls, and PCIe overheads!

FusionFS: CISCops Command

- Append-CRC-Write sequence in vanilla LevelDB code and proposed CISCops

```
WriteRawBlock(data) {
        status = file->Append(data)
        crc = crc32c::Value(data, size);
        crc = crc32c::Extend(crc, trailer, 1);
        EncodeFixed32(…crc32c::Mask(crc))
        status = file->Append(Slice(trailer, size)
}
```

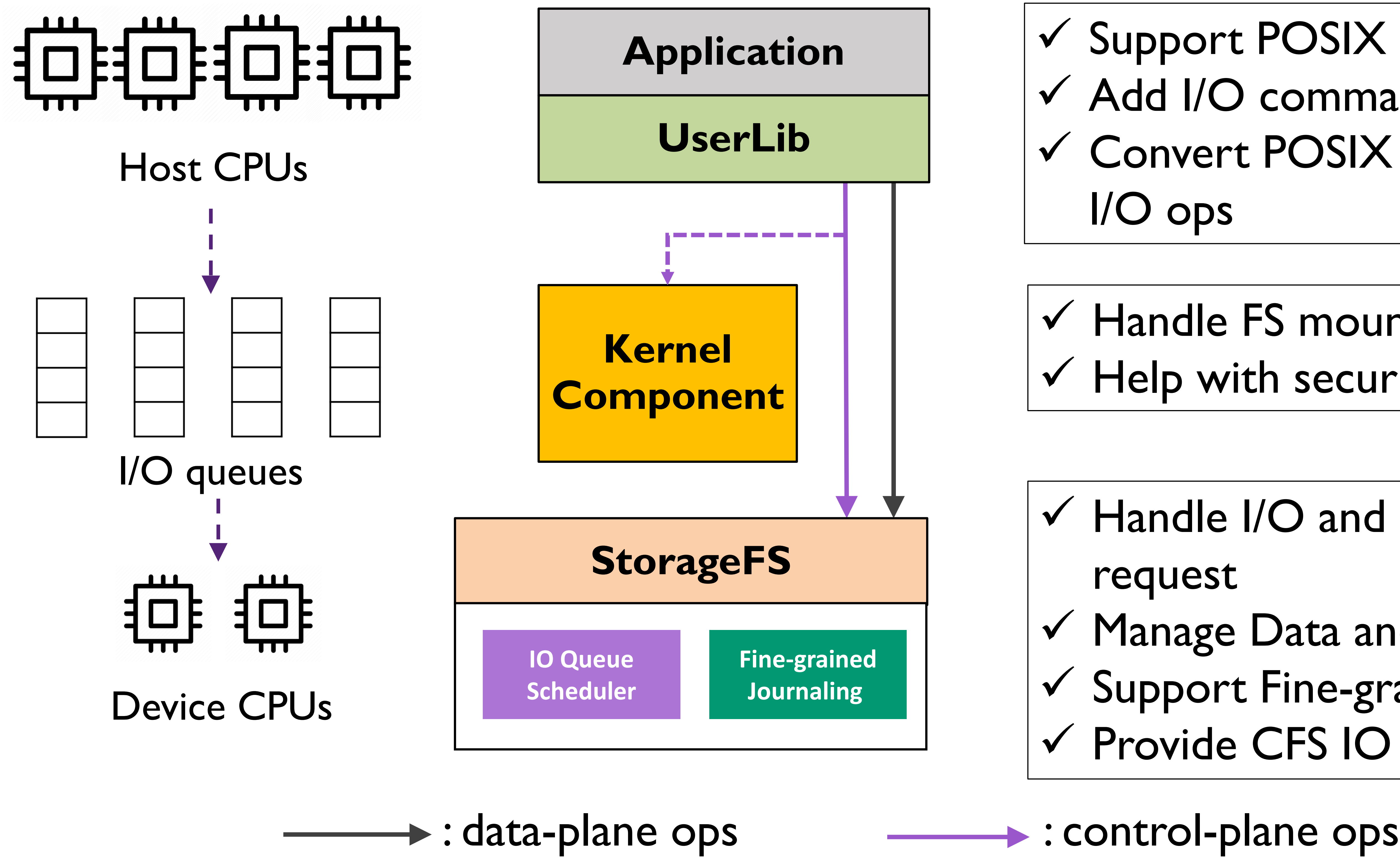
LevelDB CRC with OS FS

↓

```
WriteRawBlock(data) {
        status = file->Append-CRC-Write(data)
}
```

With CISCops

FusionFS Components

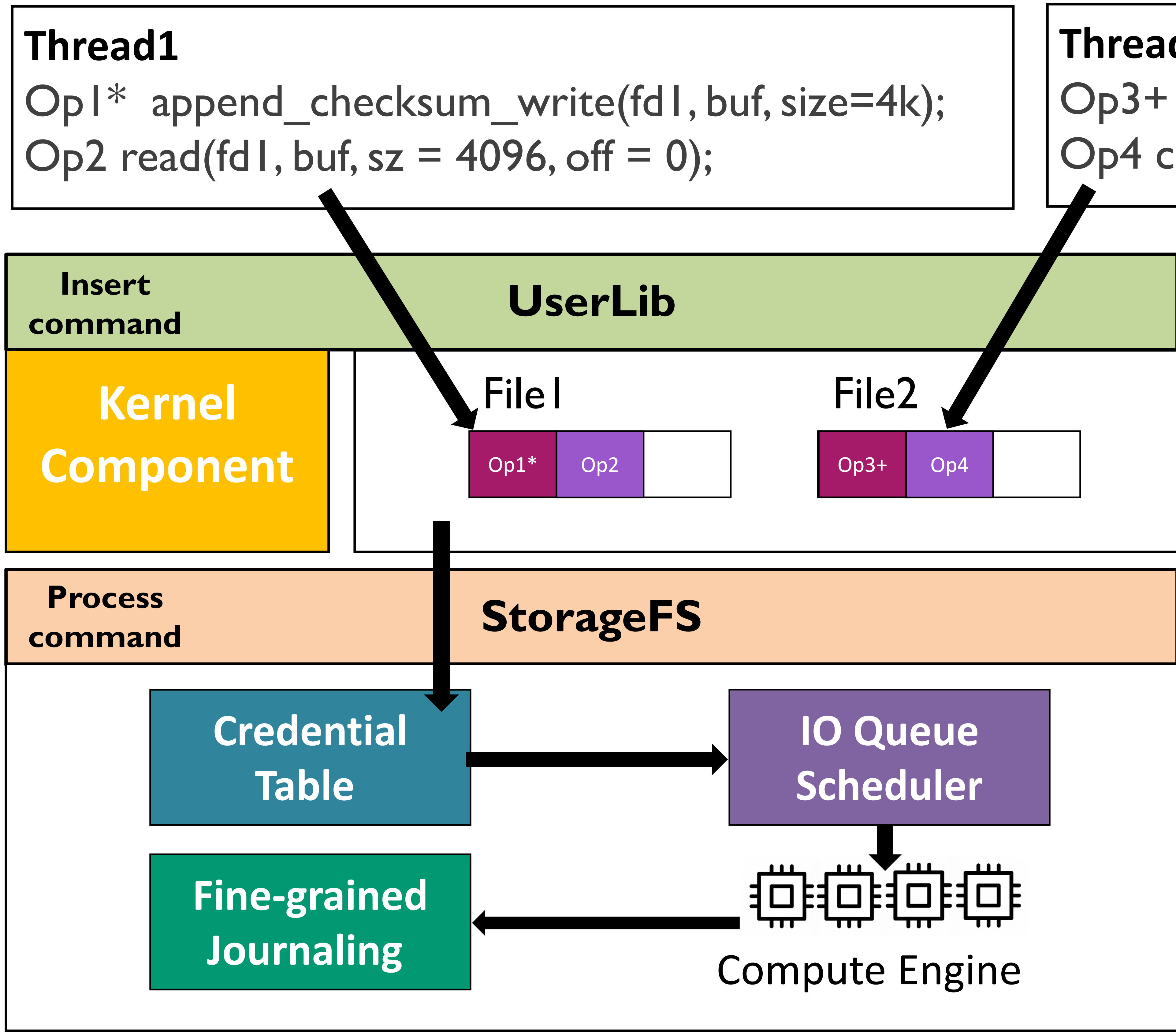


- ✓ Support POSIX semantics
- ✓ Add I/O commands to I/O queue
- ✓ Convert POSIX I/O ops to CISC I/O ops

- ✓ Handle FS mount and setup
- ✓ Help with security

- ✓ Handle I/O and Data processing request
- ✓ Manage Data and metadata
- ✓ Support Fine-grained Journaling
- ✓ Provide CFS IO Scheduling

FusionFS I/O Processing Example



Convert POSIX I/O ops to CISC IO ops

Insert I/O commands to **inode-queue**

StorageFS fetches CICS IO commands from inode-queues

IO scheduler provides fairness across multiple tenants

Journaling mechanism supports fine-grained crash consistency

FusionFS I/O Permissions

- The StorageFS maintains a credential table that maps a unique process ID to its credentials
- OS generates random (128-bit) unique ID for each process and updates the firmware credential table
- StorageFS checks if a request's unique ID matches credential table

Challenges Introduced by CISCops

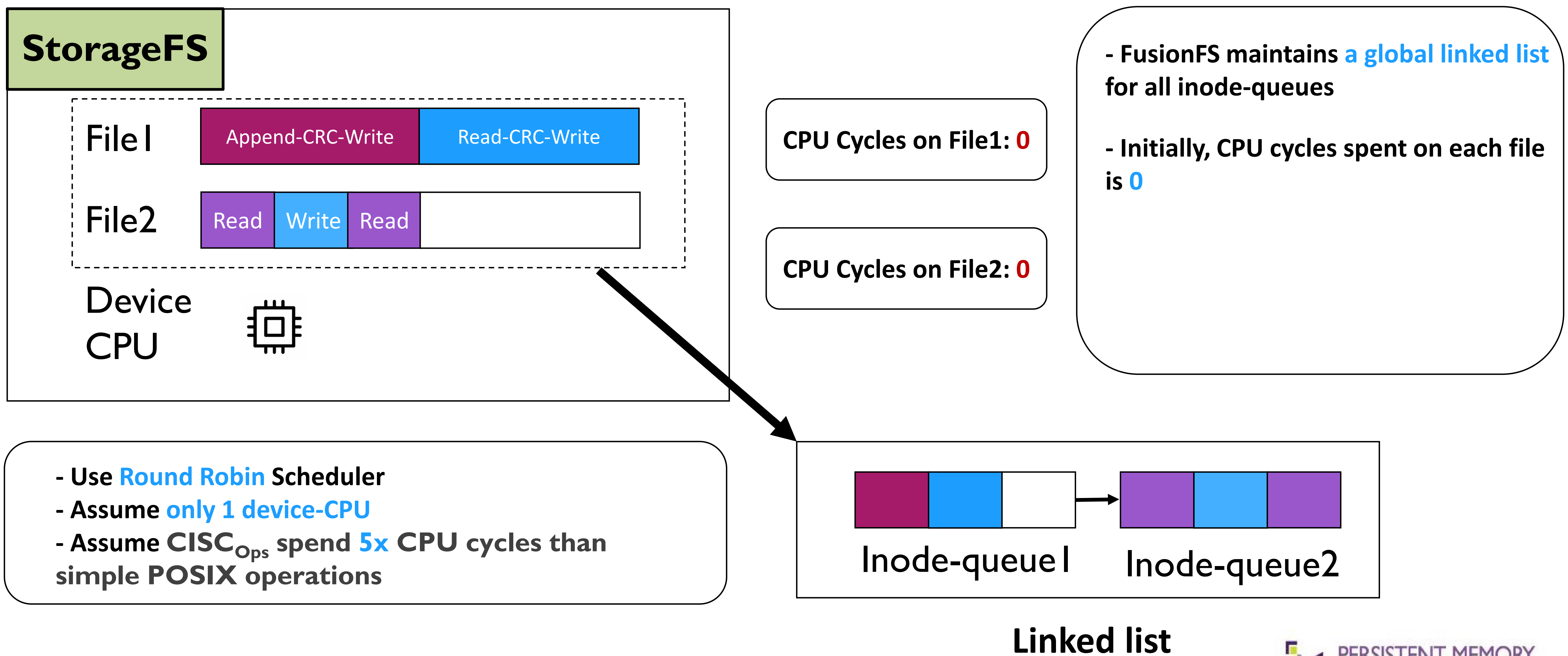
- How to transparently generating, and offloading CISCOps?
  - Solution: Partial Support for Automatic Offloading (AutoMerge)

- How to provide fairness and efficient across tenants?
  - Tenants using $CISC_{Ops}$ can consume high device compute resources
    - Device memory resources could also be high!
    - Impacts tenants doing simple I/O
  - Solution: CFS I/O Scheduler

- How to provide crash consistency for $CISC_{Ops}$?
  - Recovery the internal computational state after crash
  - Solution: MicroTx with Auto Recovery

In-storage Resource Scheduling

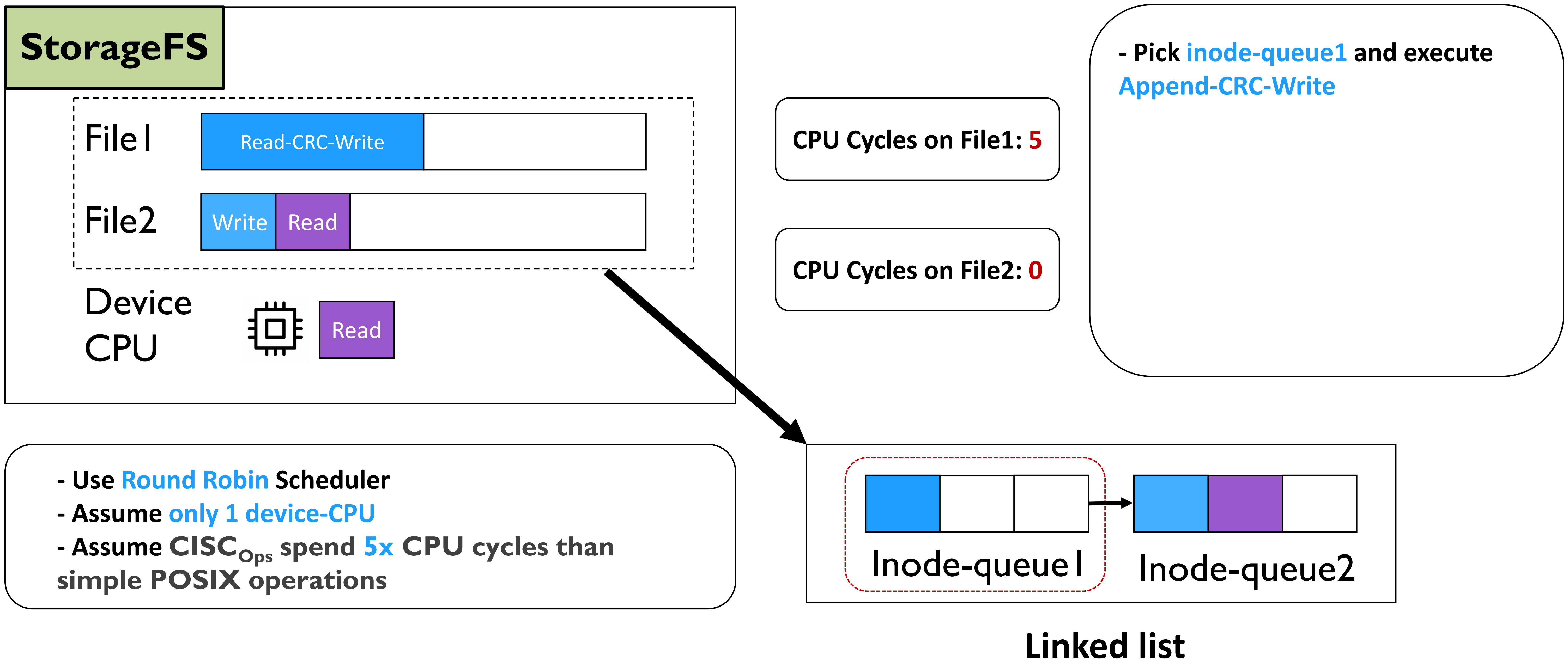
- Round Robin uses global Linked list to store inode-queues



 |

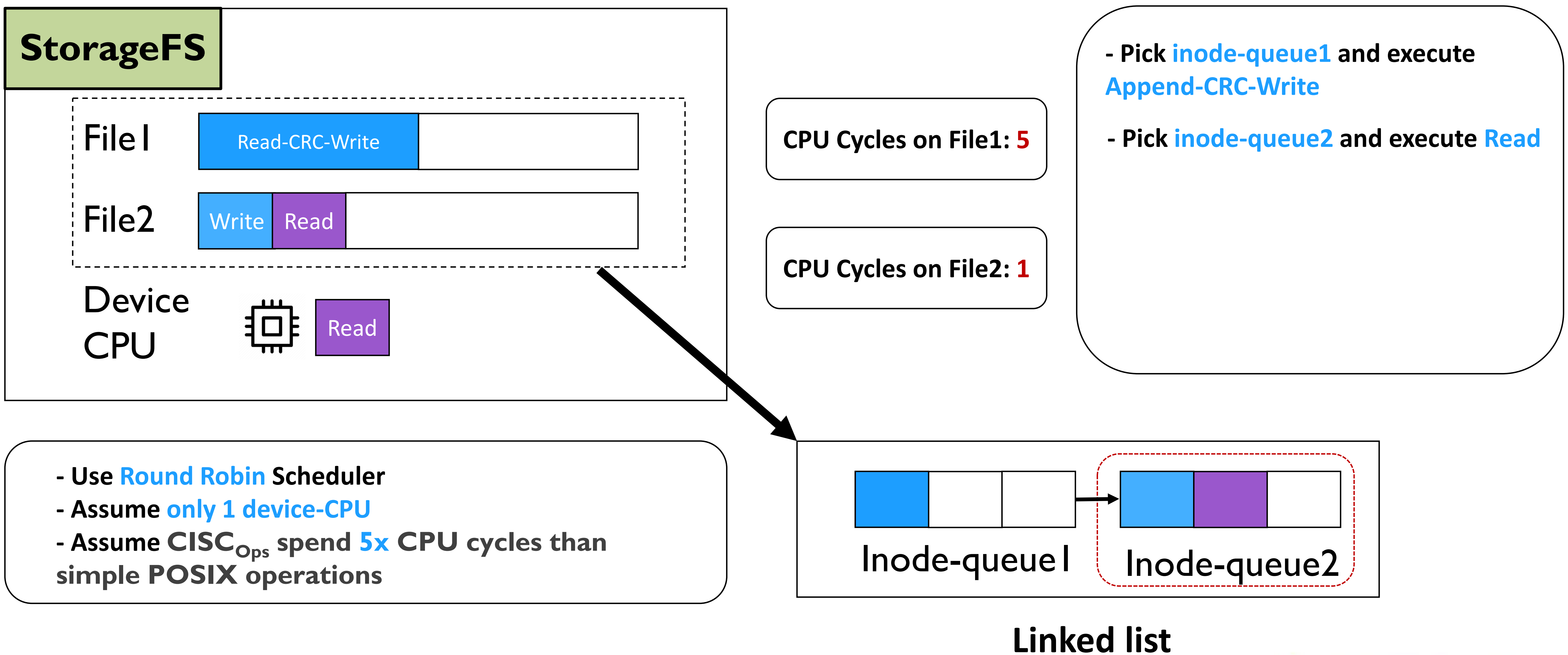
In-storage Resource Scheduling

- Round Robin uses global Linked list to store inode-queues



In-storage Resource Scheduling

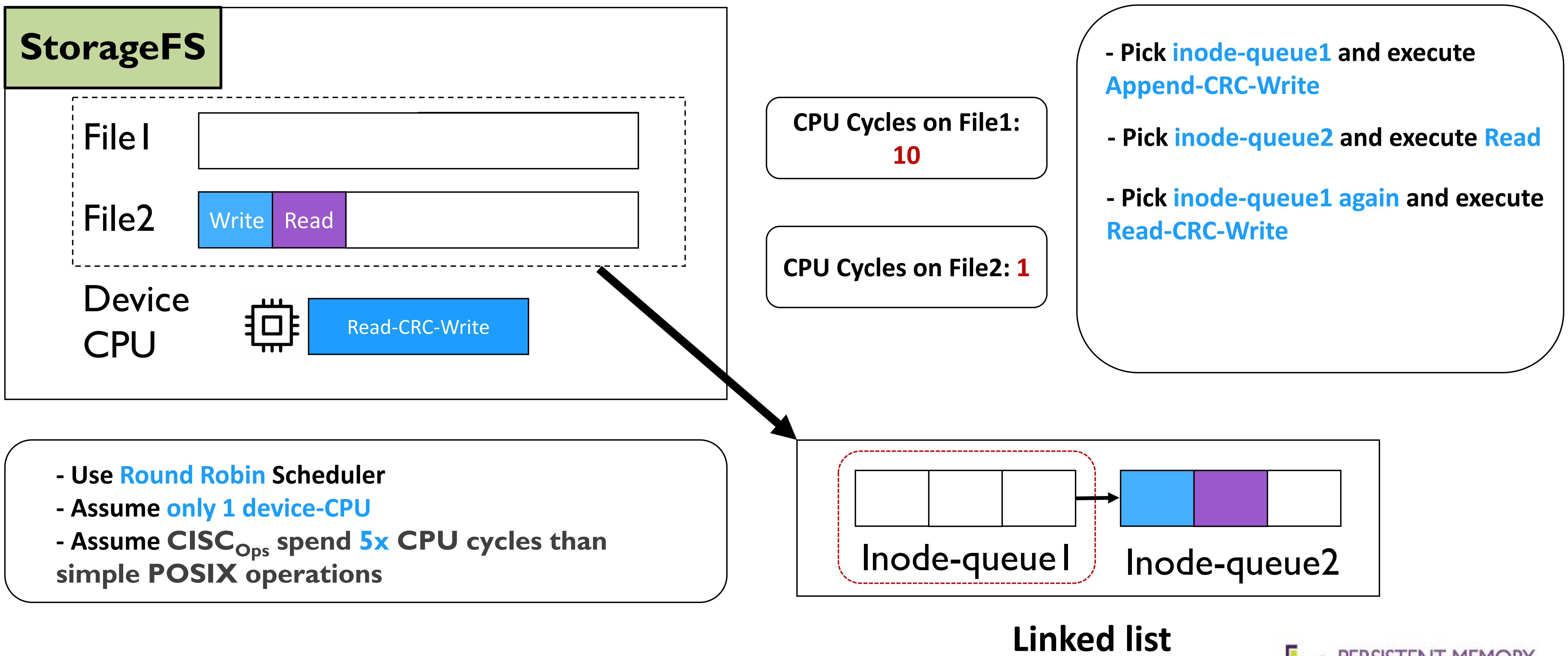
- Round Robin uses global Linked list to store inode-queues



SNIA PERSISTENT MEMORY + COMPUTATIONAL STORAGE SUMMIT 2022

In-storage Resource Scheduling

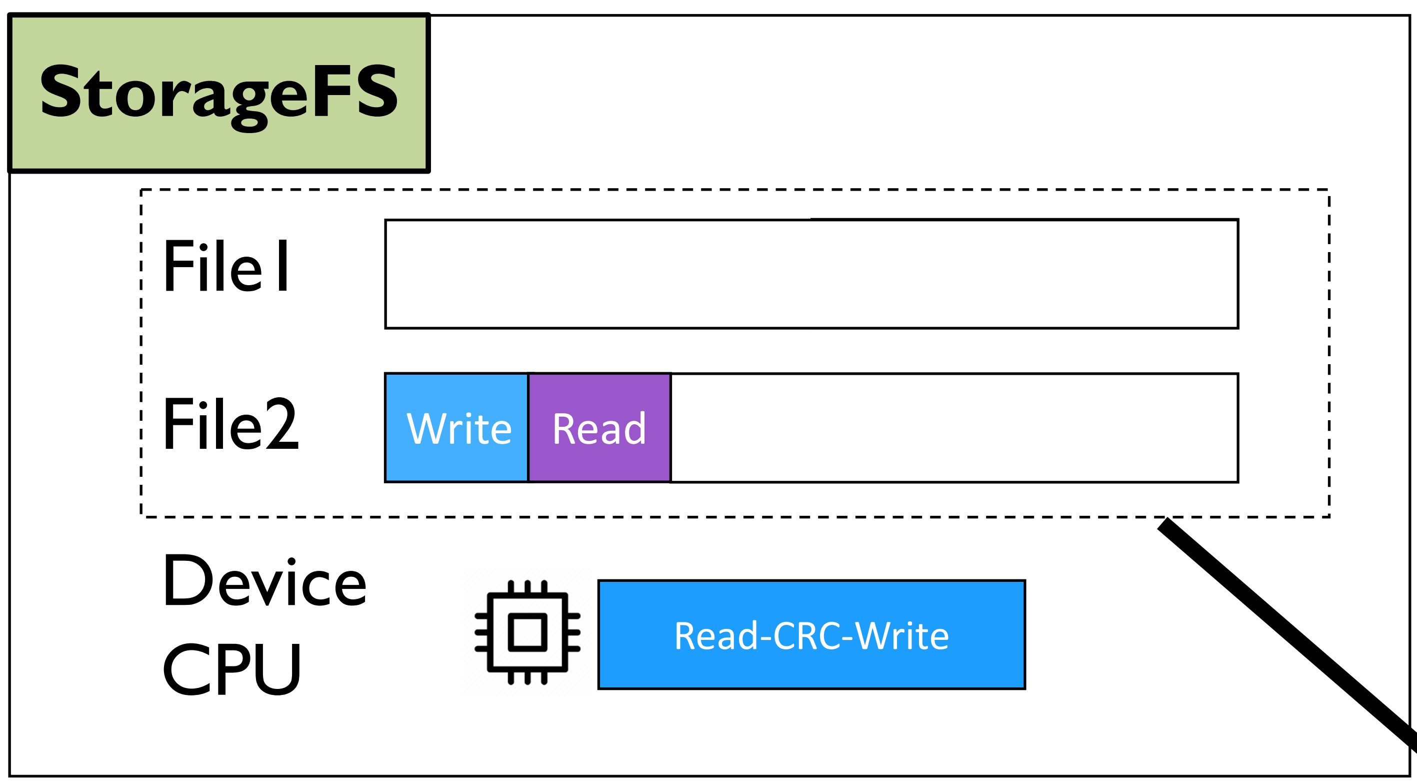
- Round Robin uses global Linked list to store inode-queues



In-storage Resource Scheduling

- Round Robin uses global Linked list to store inode-queues

How to provide fairness across tenants?

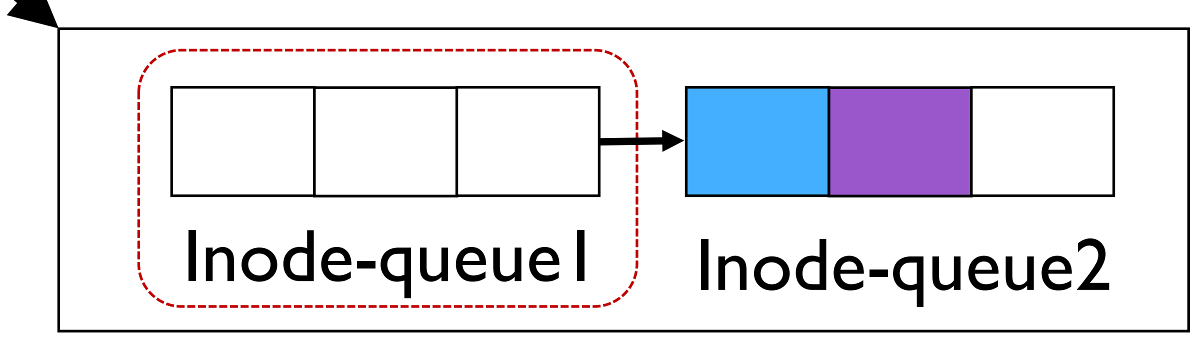


CPU Cycles on File1: 10

CPU Cycles on File2: 1

- Pick inode-queue1 and execute Append-CRC-Write
- Pick inode-queue2 and execute Read
- Pick inode-queue1 and execute Read-CRC-Write
- Write op. for File2 must wait 5 CPU cycles!!!

- Use Round Robin Scheduler
- Assume only 1 device-CPU
- Assume CISC_{Ops} spend 5x CPU cycles than simple POSIX operations

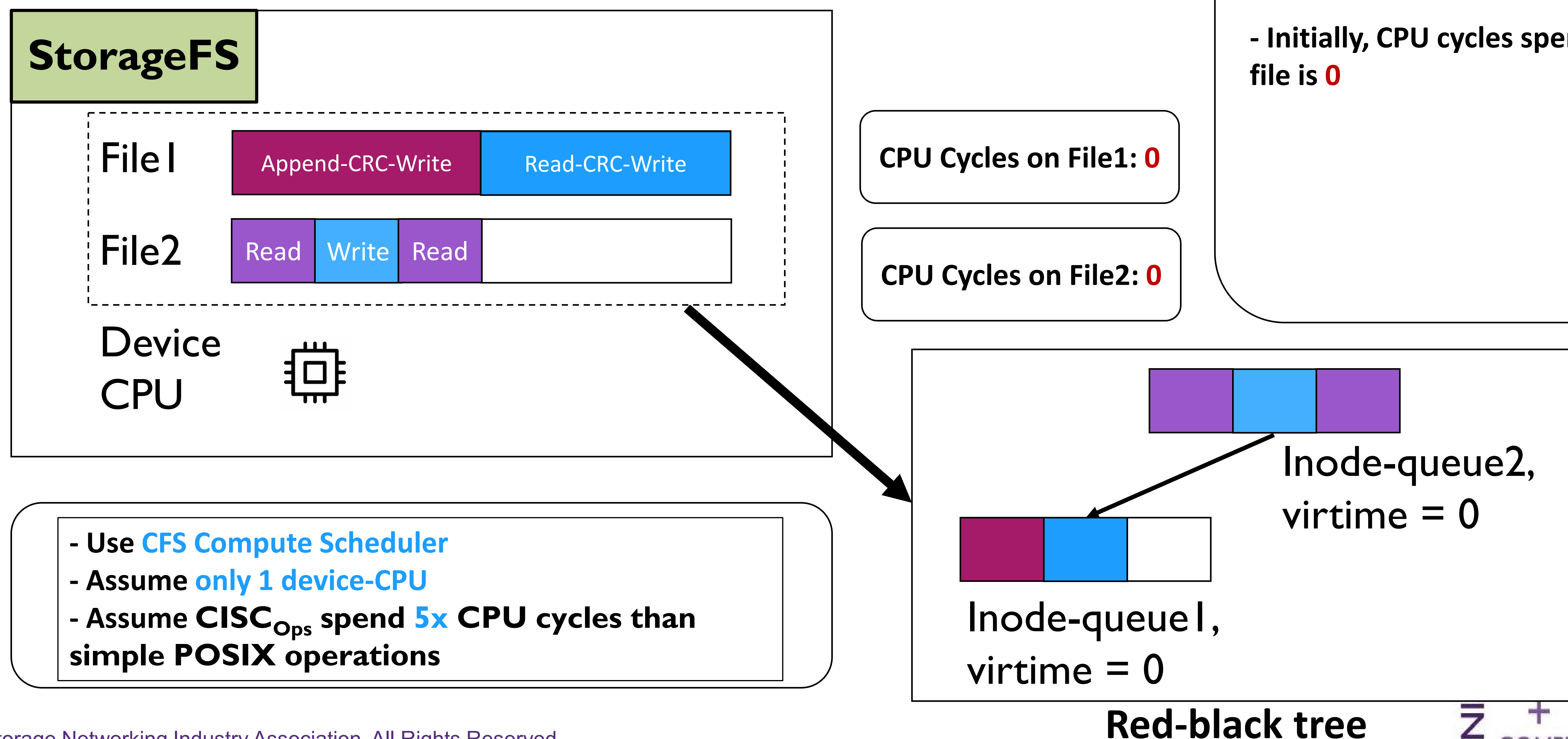


Linked list

In-storage CFS Resource Scheduling

- Prioritize inode-queues with the least CPU usage (i.e., virtual CPU runtime)
  - StorageFS uses global RB-tree to store sorted virtime of inode-queues

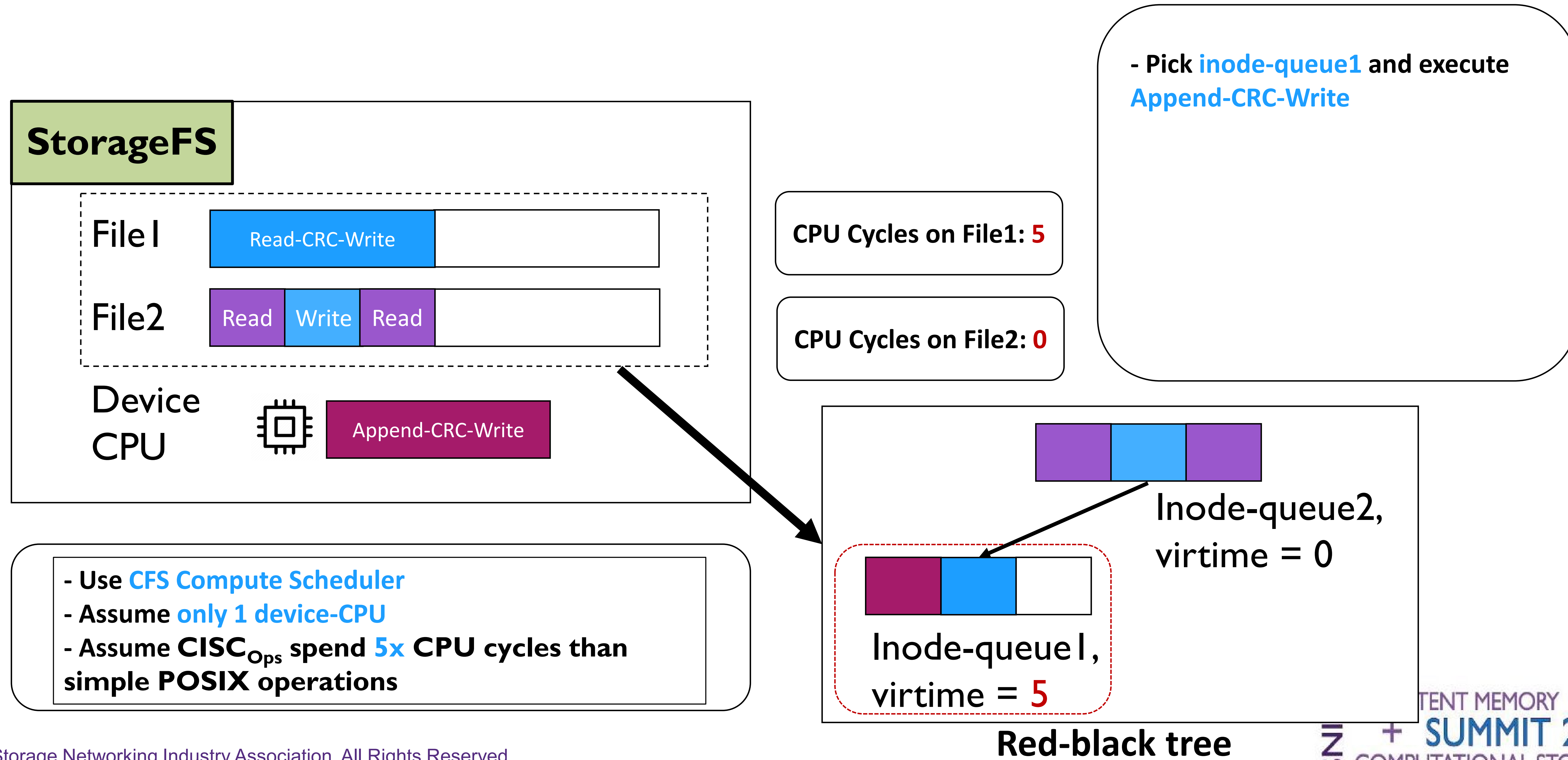
- Red-black tree for all inode-queues
- Initially, CPU cycles spent on each file is 0



- Use CFS Compute Scheduler
- Assume only 1 device-CPU
- Assume $CISC_{Ops}$ spend 5x CPU cycles than simple POSIX operations

In-storage CFS Resource Scheduling

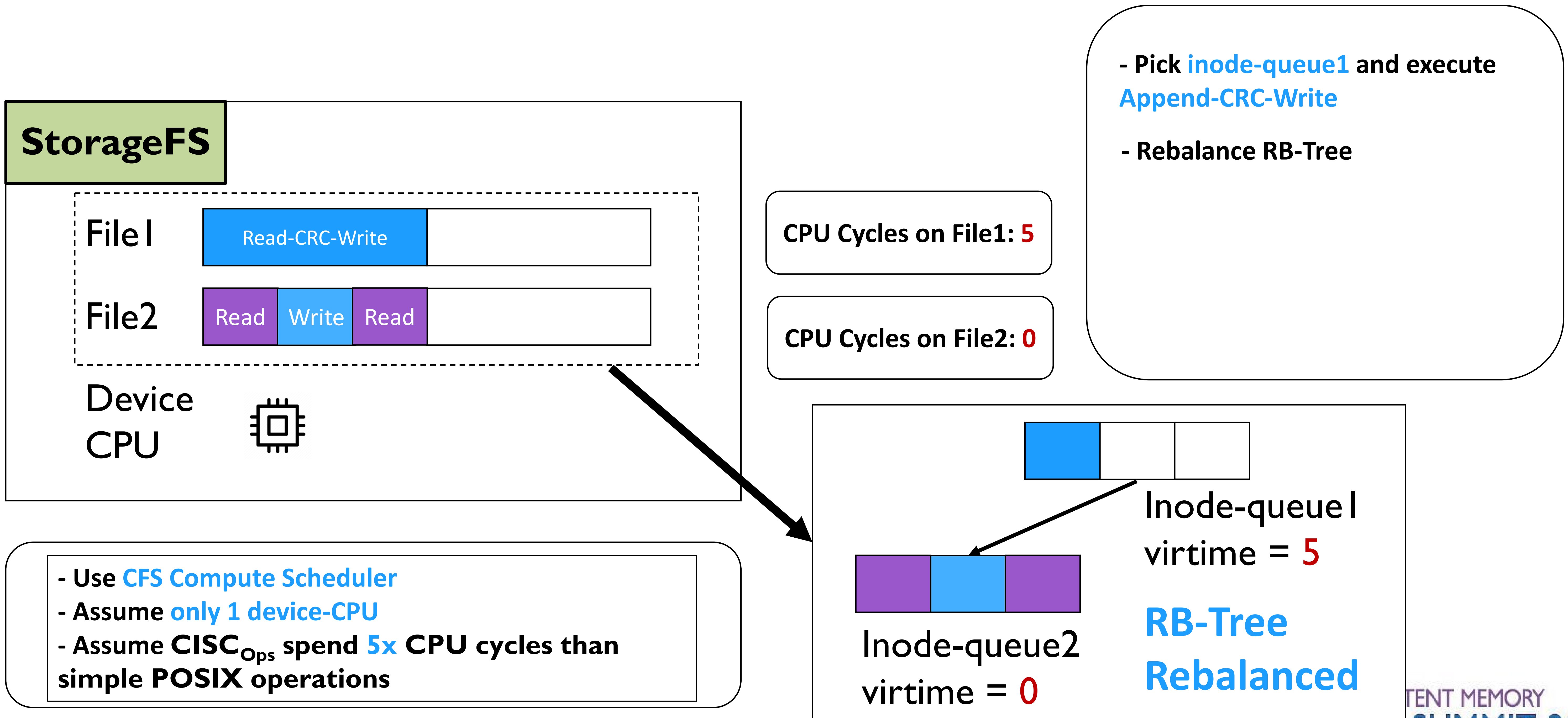
- Prioritize inode-queues with the least CPU usage (i.e., virtual CPU runtime)
  - StorageFS uses global RB-tree to store sorted virtime of inode-queues



SNIA PERSISTENT MEMORY + COMPUTATIONAL STORAGE SUMMIT 2022

In-storage CFS Resource Scheduling

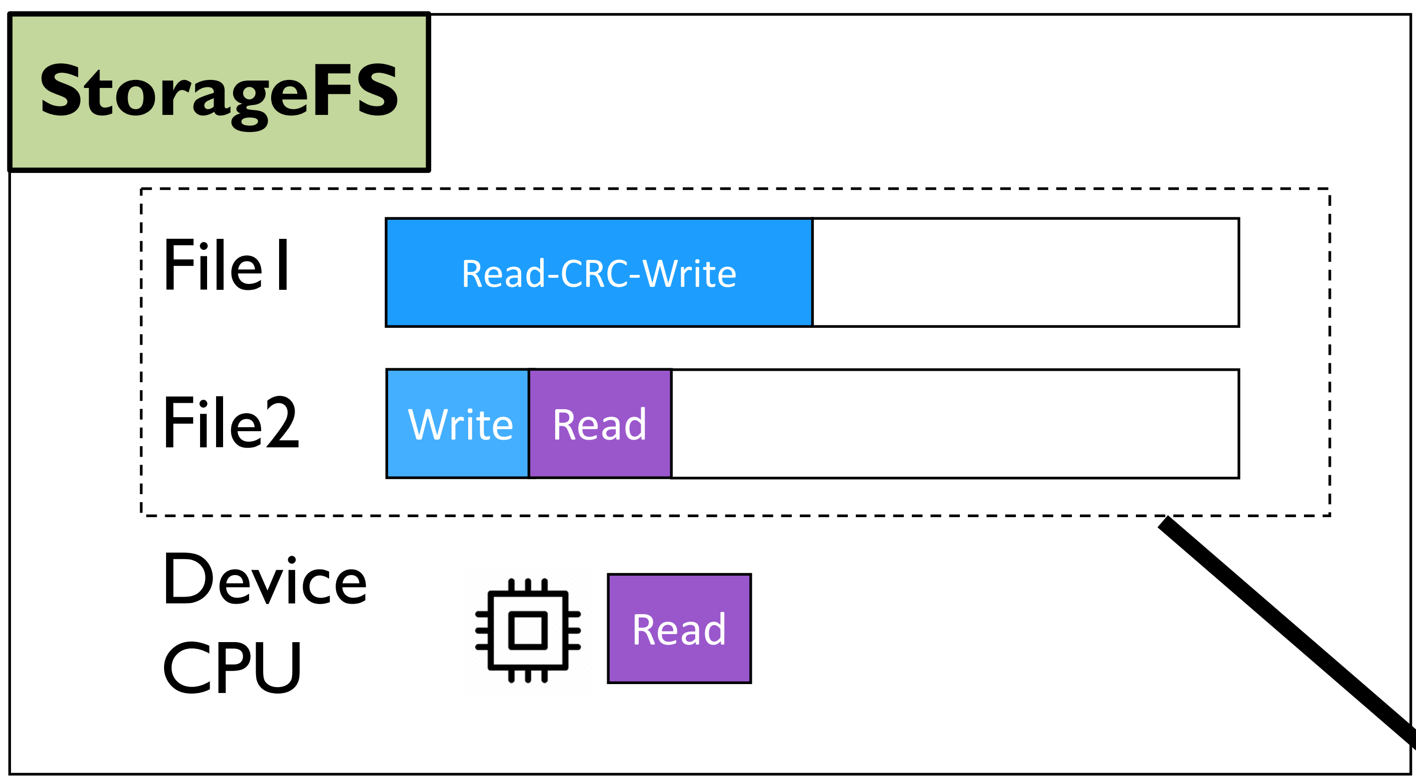
- Prioritize inode-queues with the least CPU usage (i.e., virtual CPU runtime)
  - StorageFS uses global RB-tree to store sorted virtime of inode-queues



Red-black tree

In-storage CFS Resource Scheduling

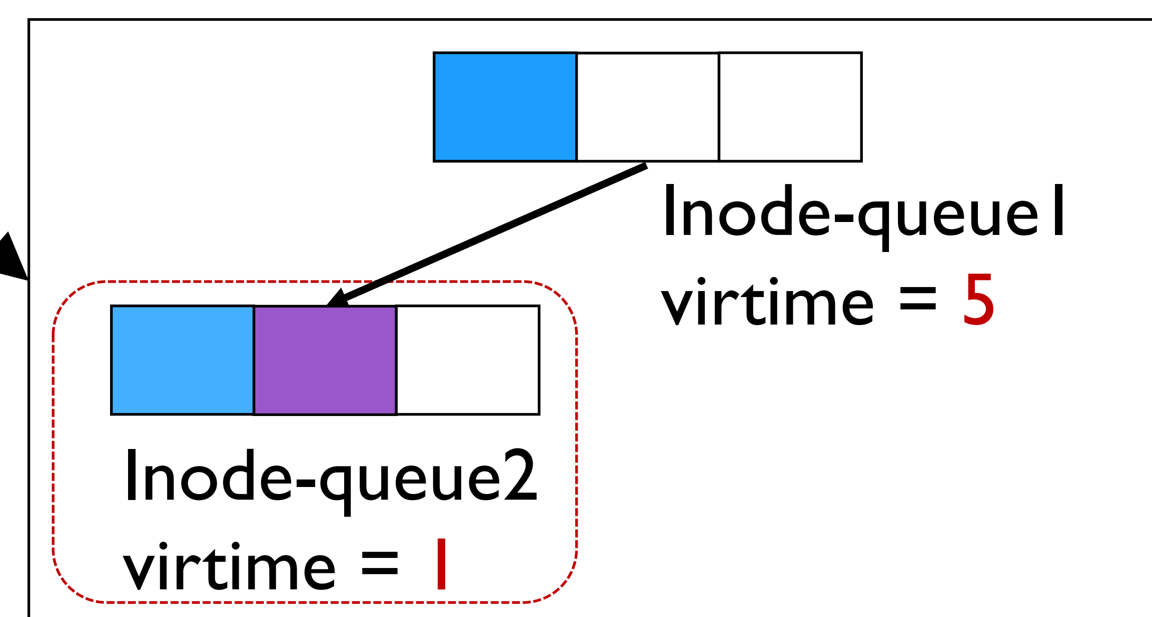
- Prioritize inode-queues with the least CPU usage (i.e., virtual CPU runtime)
  - StorageFS uses global RB-tree to store sorted virtime of inode-queues



CPU Cycles on File1: 5

CPU Cycles on File2: 1

- Pick inode-queue1 and execute Append-CRC-Write
- Rebalance RB-Tree
- Pick inode-queue2 and execute Read



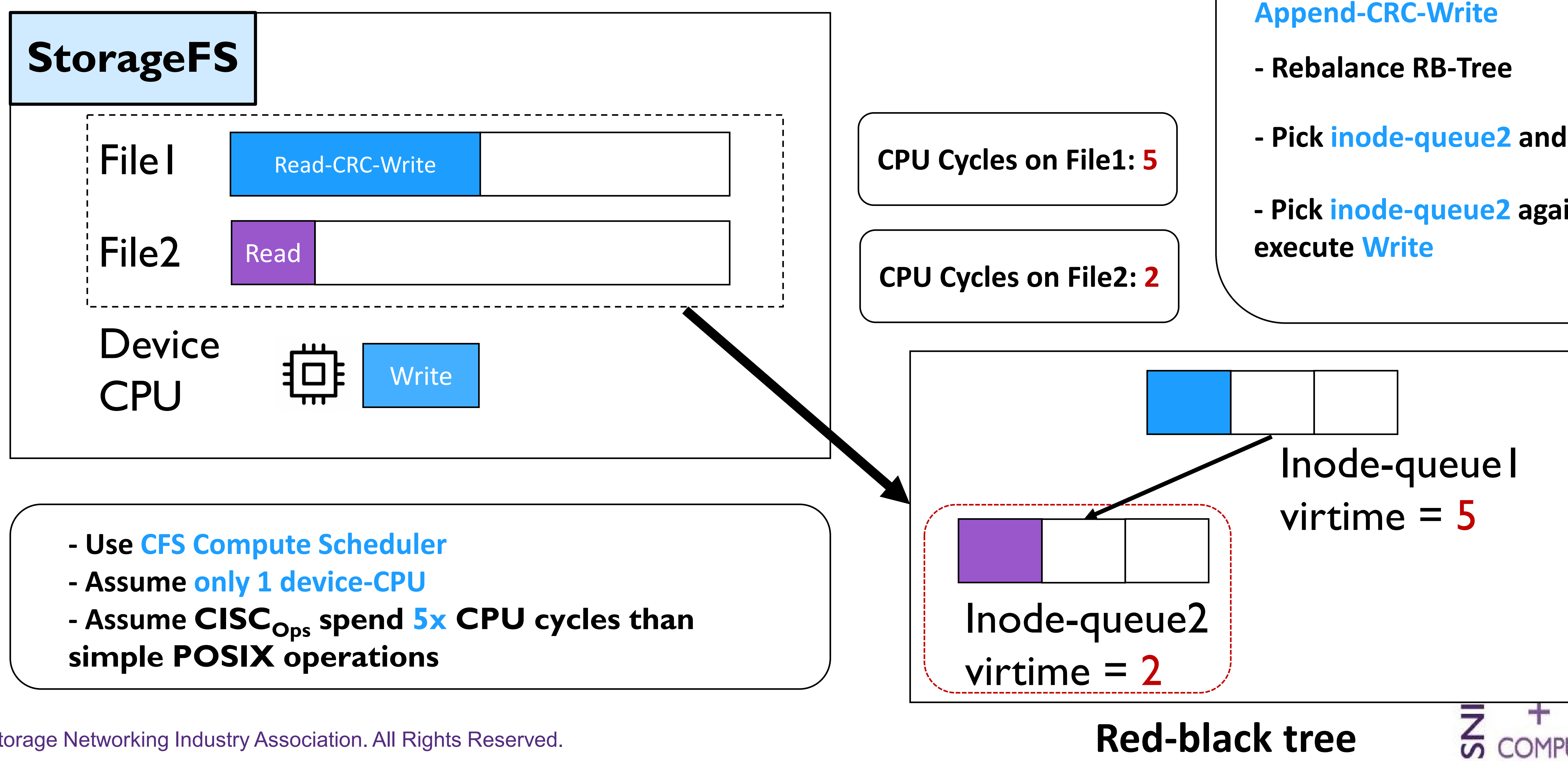
Red-black tree

- Use CFS Compute Scheduler
- Assume only 1 device-CPU
- Assume $CISC_{Ops}$ spend 5x CPU cycles than simple POSIX operations

In-storage CFS Resource Scheduling

- Prioritize inode-queues with the least CPU usage (i.e., virtual CPU runtime)
  - StorageFS uses global RB-tree to store sorted virtime of inode-queues

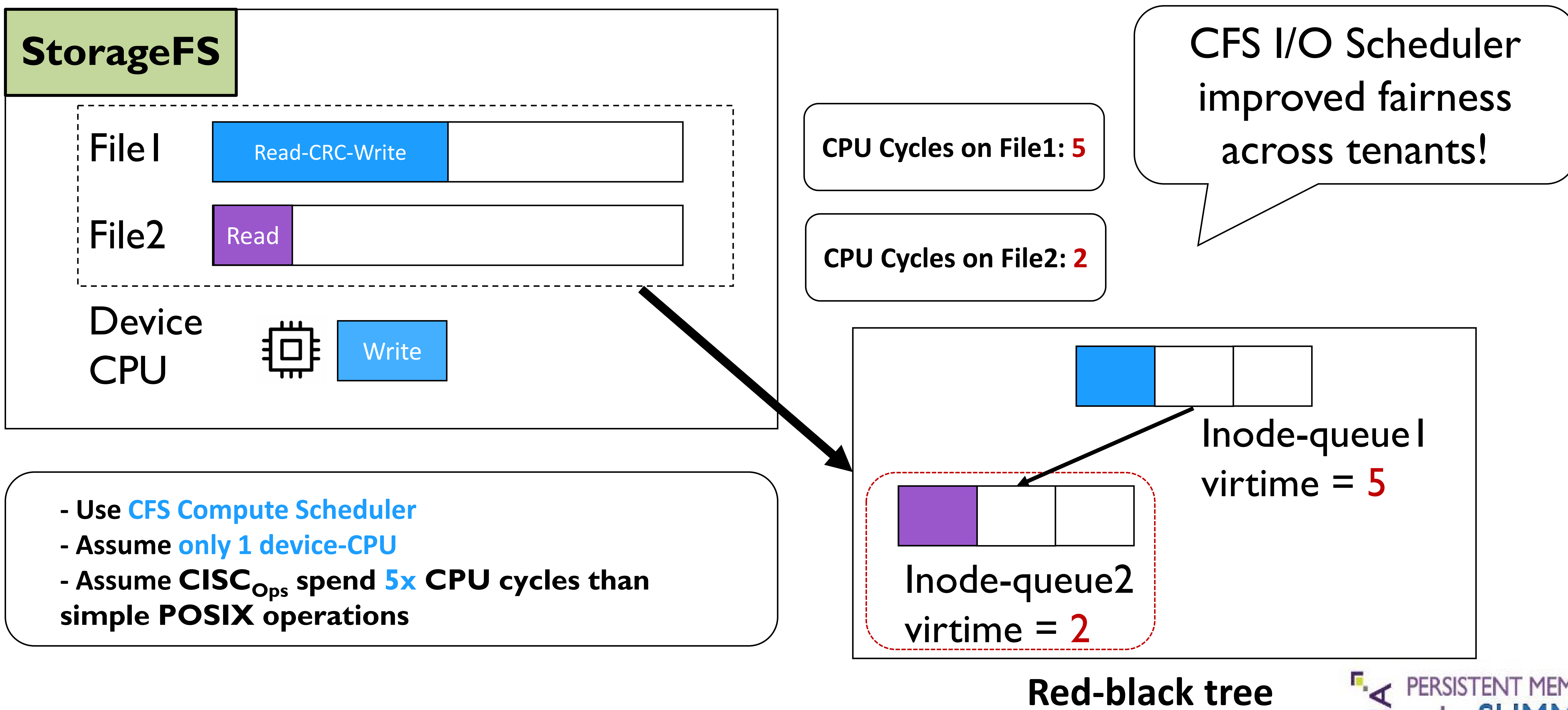
- Pick inode-queue1 and execute Append-CRC-Write
- Rebalance RB-Tree
- Pick inode-queue2 and execute Read
- Pick inode-queue2 again and execute Write



- Use CFS Compute Scheduler
- Assume only 1 device-CPU
- Assume $CISC_{Ops}$ spend 5x CPU cycles than simple POSIX operations

In-storage CFS Resource Scheduling

- Prioritize inode-queues with the least CPU usage (i.e., virtual CPU runtime)
  - StorageFS uses global RB-tree to store sorted virtime of inode-queues



In-storage CFS Resource Scheduling

- Efficient management of device-RAM is critical for fairness
  - Modern CSDs are equipped with 4-16GB of memory

- A combination of in-storage data processing and POSIX could cause in-storage memory contention and starvation

- Enhance the CFS scheduler with memory usage (memuse) accounting for each inode-queue

Crash Consistency for CISC_{Ops}

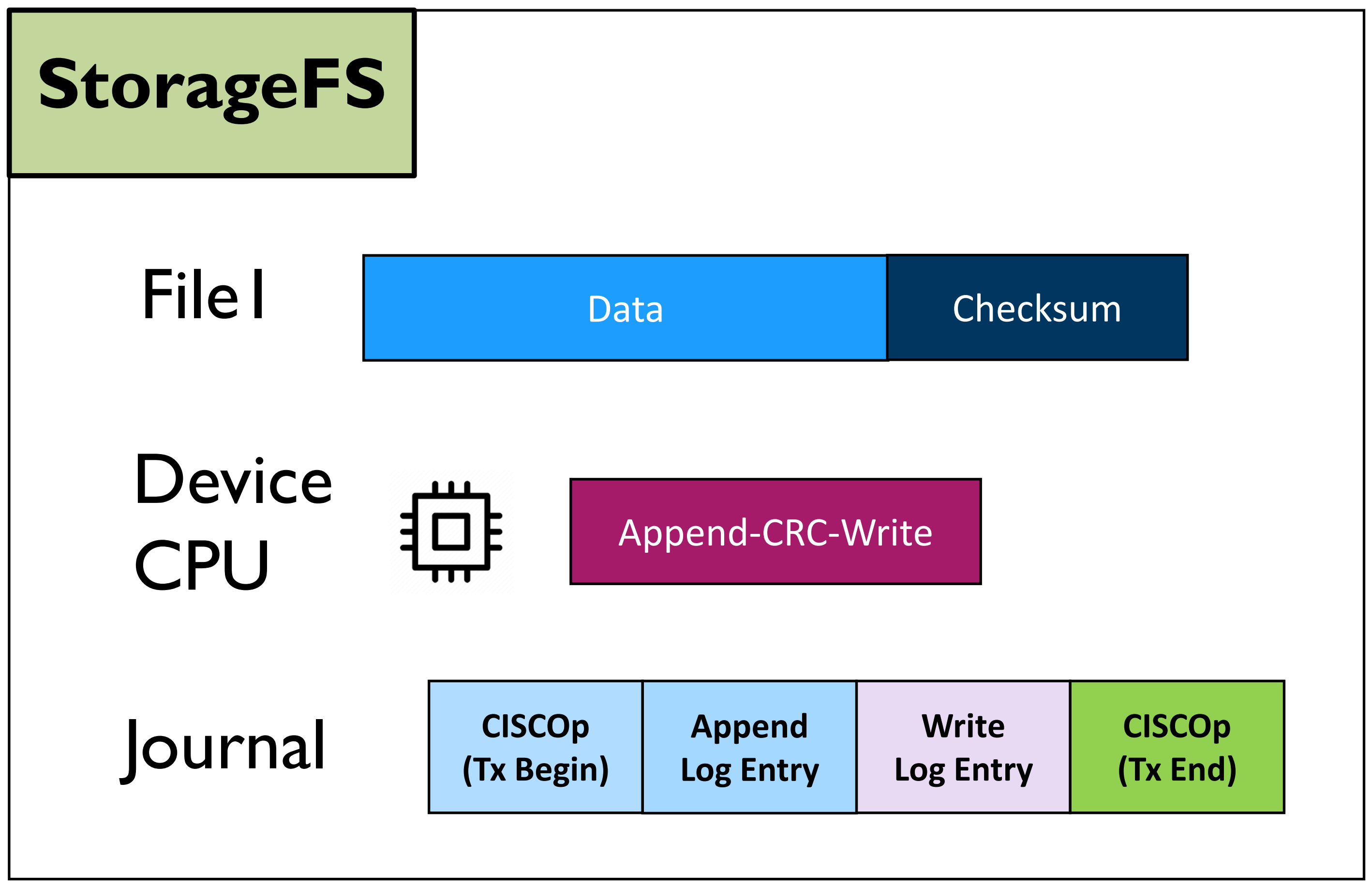
- How to provide crash consistency for CISC_{Ops}?

- Macro-transactions (MacroTx): all-or-nothing approach

- Micro-transactions (MicroTx): recover **partially committed** CISC_{Ops}

MacroTx: All-or-nothing Approach

- Commits and recovers an **entire** CISC_{Op} including data processing state or **nothing**



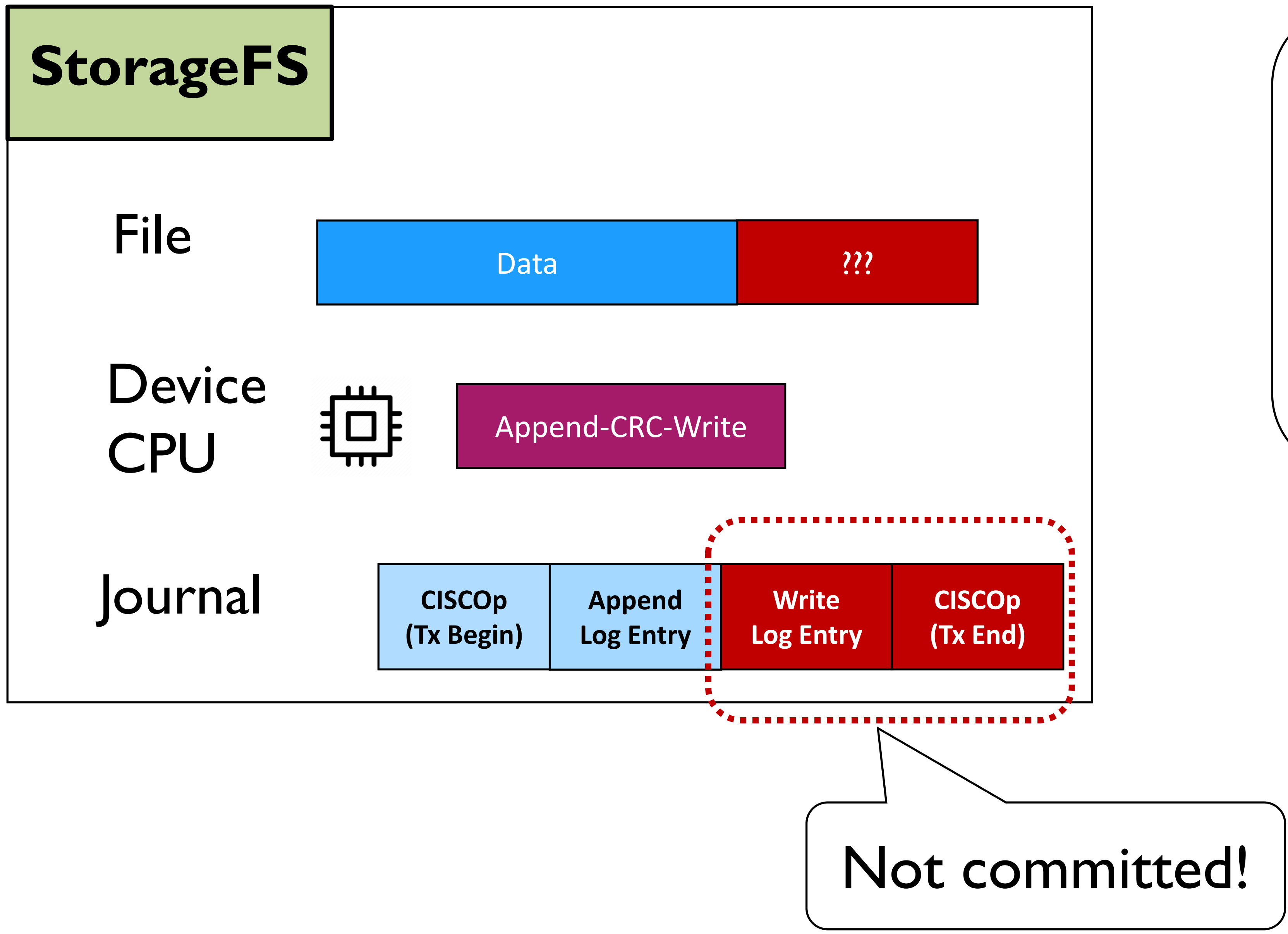
- Add transaction TxB
- Add log entry for Append
- Execute Checksum on Device CPU
- Add log entry for Write
- Commit entire transaction (TxE)

Redo the journal log to recover the state.

PERSISTENT MEMORY + SUMMIT 2022 COMPUTATIONAL STORAGE

MacroTx: All-or-nothing Approach

- Commits and recovers an **entire** $CISC_{Op}$ including data processing state or **nothing**

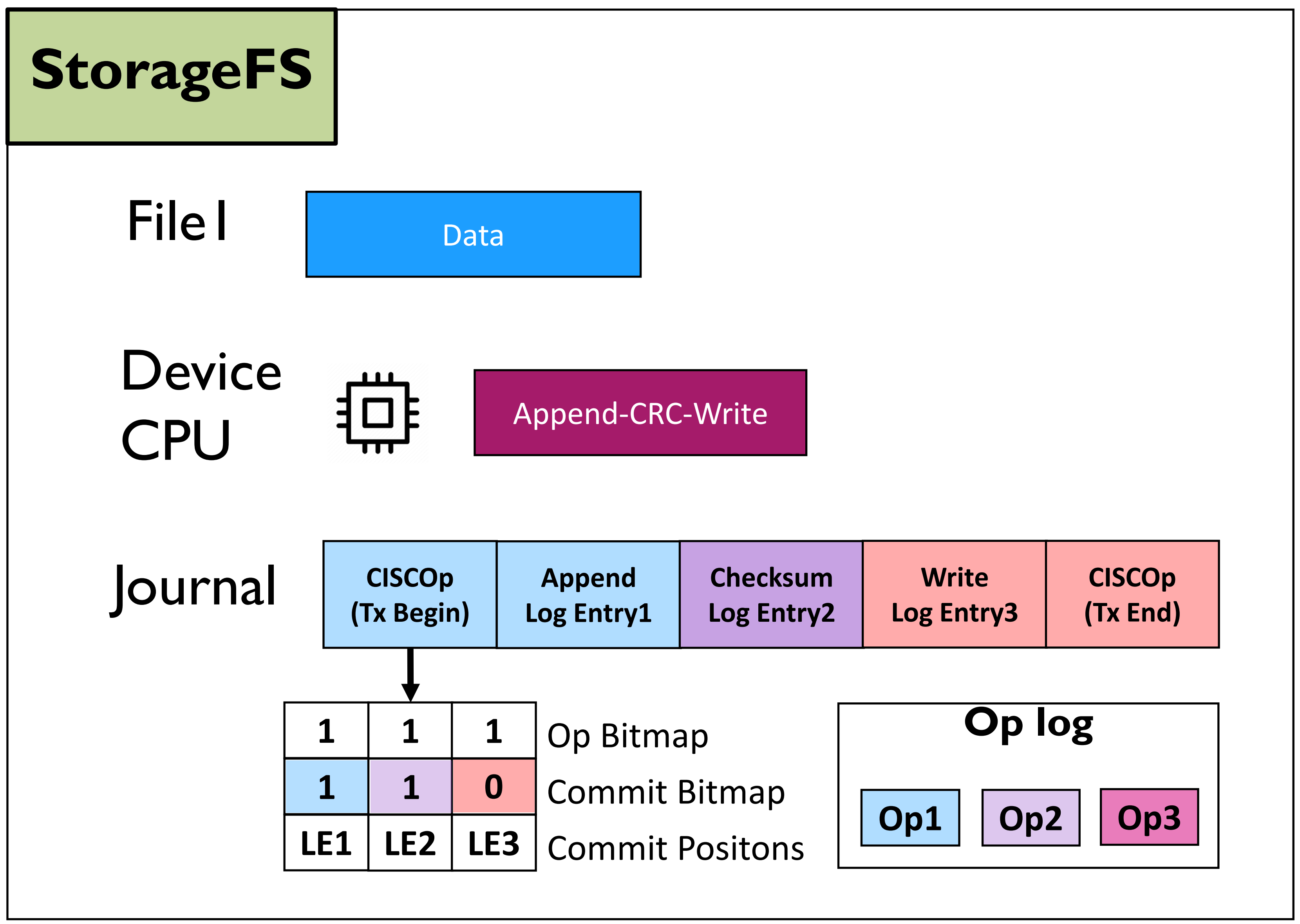


- Add transaction TxB
- Add log entry for Append
- Execute Checksum on Device CPU
- Crash!
- Add log entry for Write
- Commit entire transaction (TxE)

How to recover the computational state after crash?

MicroTx with Auto Recovery

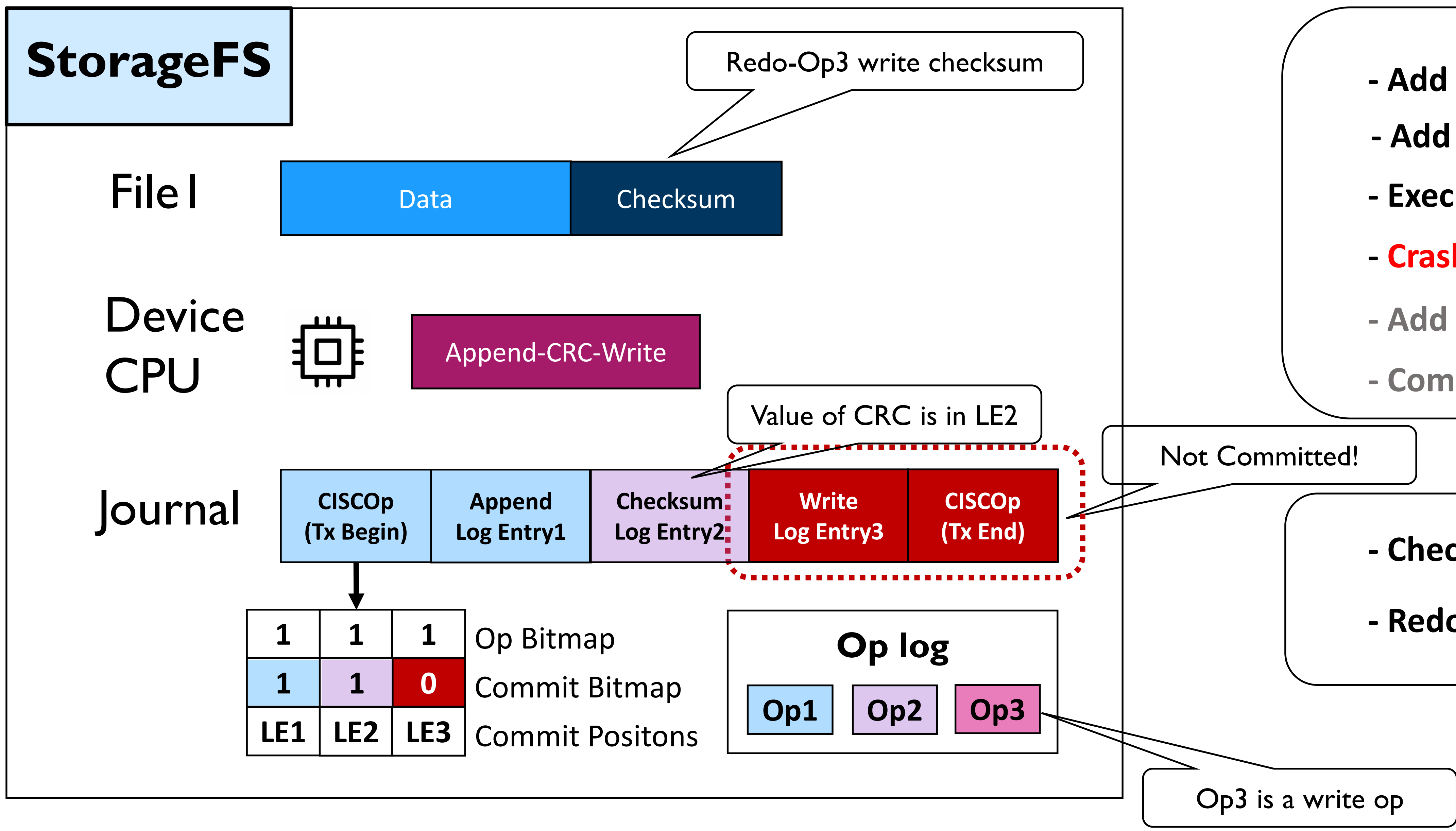
- Supports crash consistency of **partially committed** $CISC_{Ops}$
- Each operation (micro-op) of a $CISC_{Op}$ can be independently committed



- **Add transaction TxB and op log**
- **Add log entry for Append**
- **Add log entry for Checksum**
- **Crash!**

MicroTx with Auto Recovery

- Auto recovery: replay journal by checking op log and uncommitted bitmap



- Add transaction TxB and op log
- Add log entry for Append
- Execute Checksum on Device CPU
- Crash!
- Add log entry for Write
- Commit entire transaction (TxE)

- Check uncommitted micro-op
- Redo log entry with Op log

Outline

- Background
- Motivation
- Design
- Evaluation
- Conclusion

Experimental Setup

- Hardware platform
  - Dual-socket 64-core Xeon Scalable CPU @ 2.6GHz
  - 512GB Intel Optane DC NVM

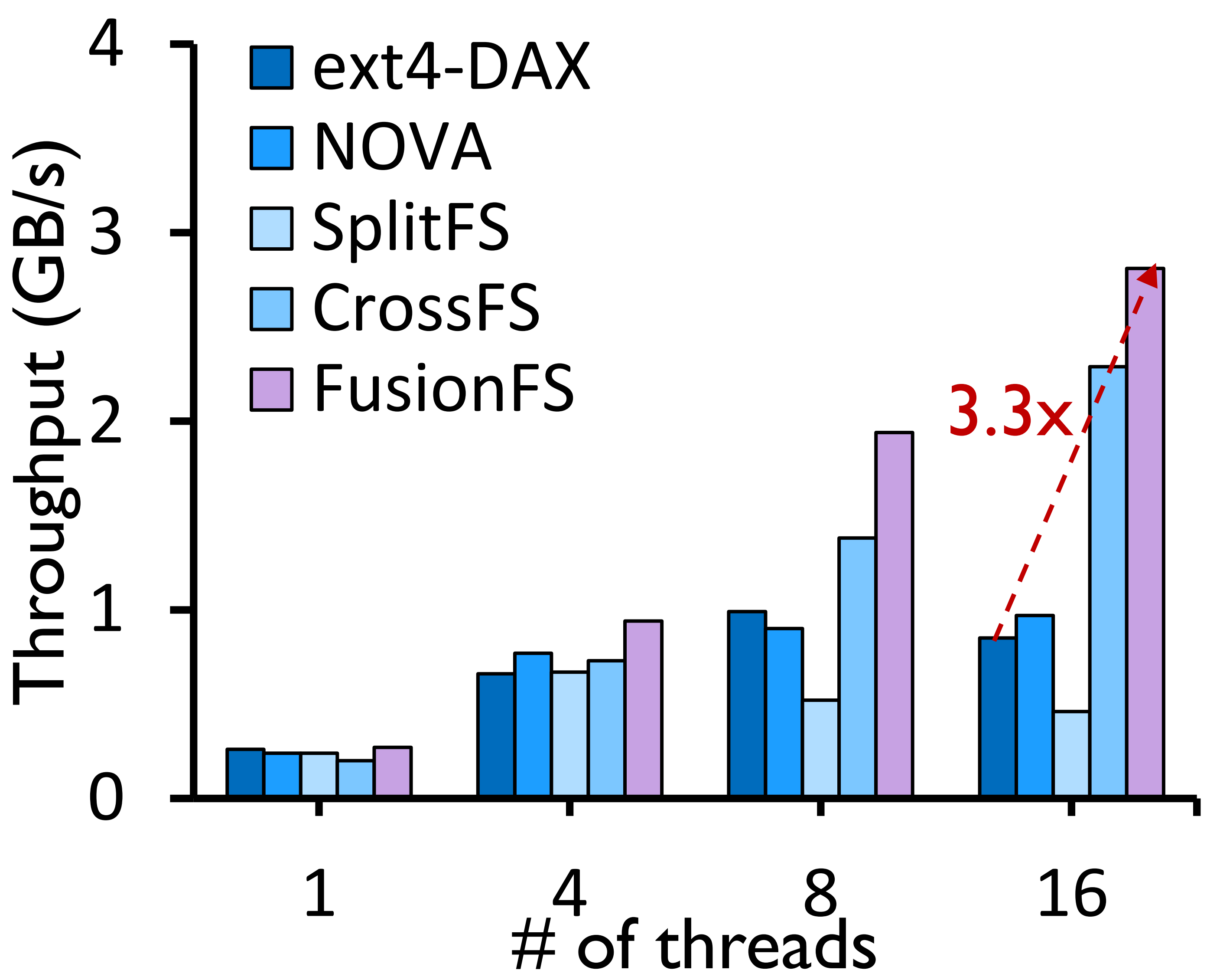
- Emulated in-storage FS (no programmable storage H/W)
  - Dedicate device threads for handling I/O requests
  - Add PCIe latency for all I/O operations
  - Reduce CPU frequency for device CPUs (and memory bandwidth)

- State-of-the-art file systems
  - **ext4-DAX**, **NOVA** [FAST' 16] (Kernel-level file system)
  - **SplitFS** [SOSP' 19] (User-level file system)
  - **CrossFS** [OSDI' 20] (Firmware-level file system)

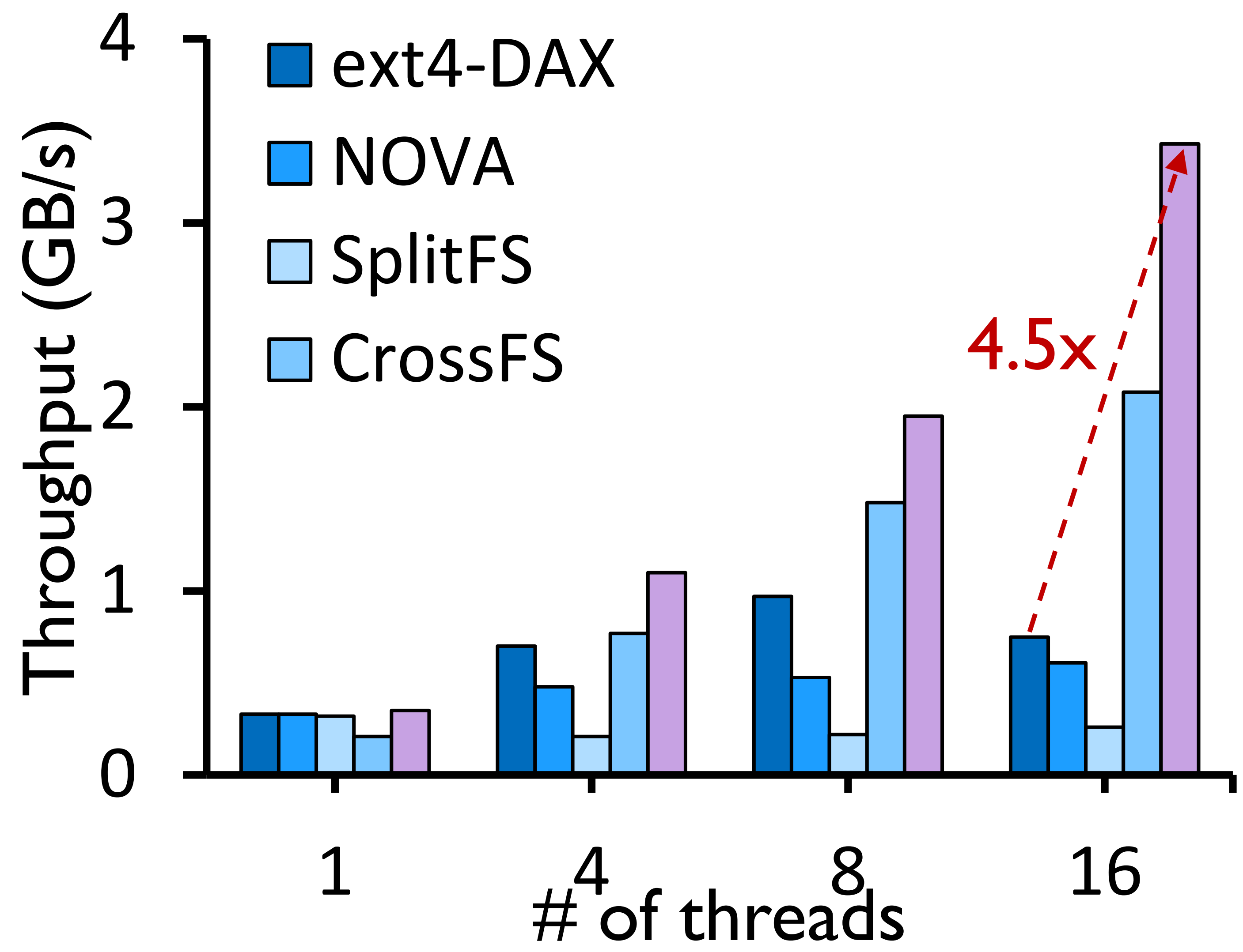
Evaluation Goals

- **Understand effectiveness of FusionFS and $CISC_{Ops}$ to reduce I/O overheads**
- Study MicroTx's durability and auto-recovery benefits
- Evaluate effectiveness of CFS scheduler for resource fairness across tenants?
- Discuss overall Real-world application impact

Microbenchmark



Append-CRC-Write

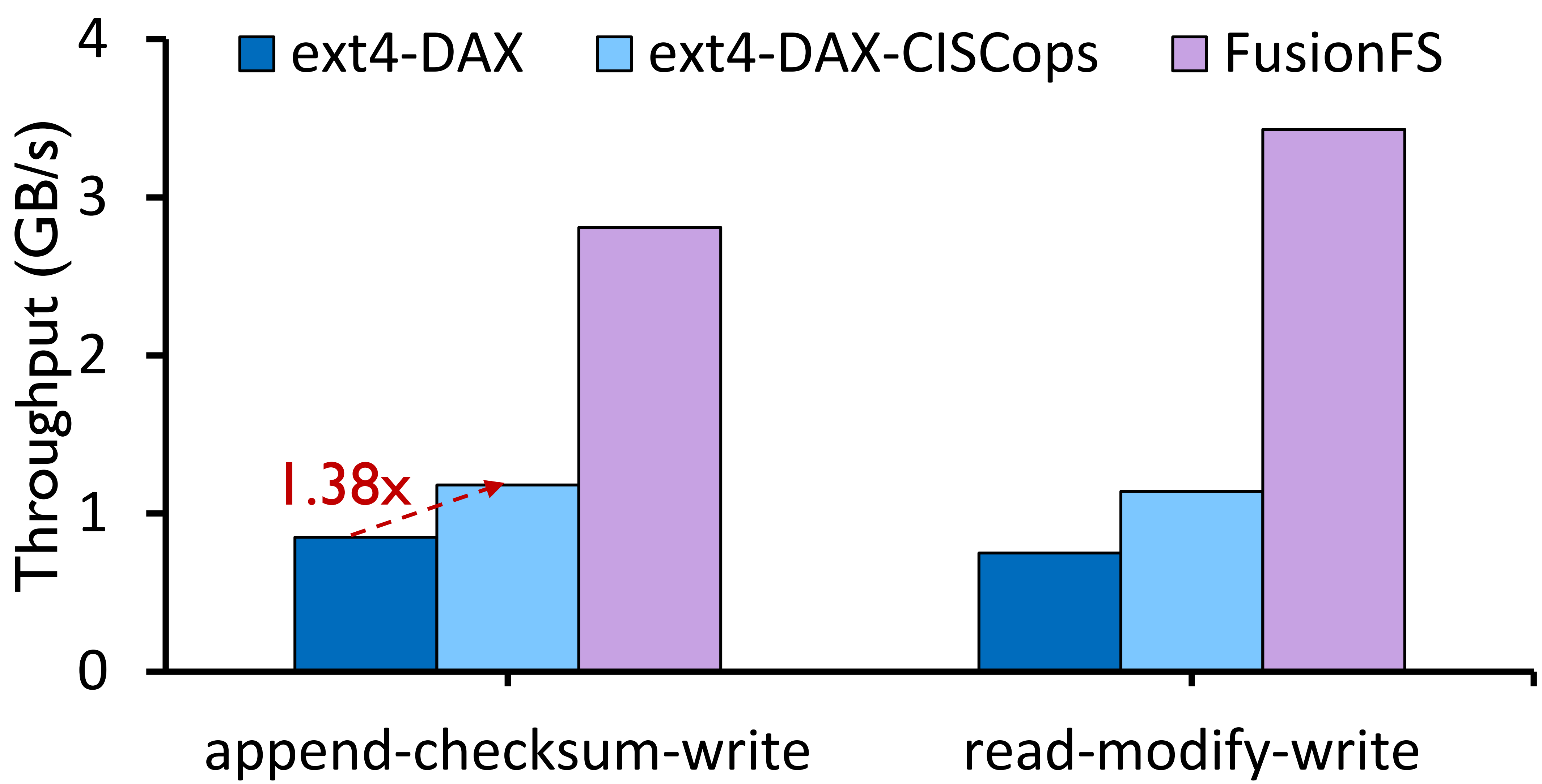


Read-Modify-Write

FusionFS achieves higher throughput by reducing data movement and system call overhead with CISCops

Microbenchmark : CISCops for ext4-DAX

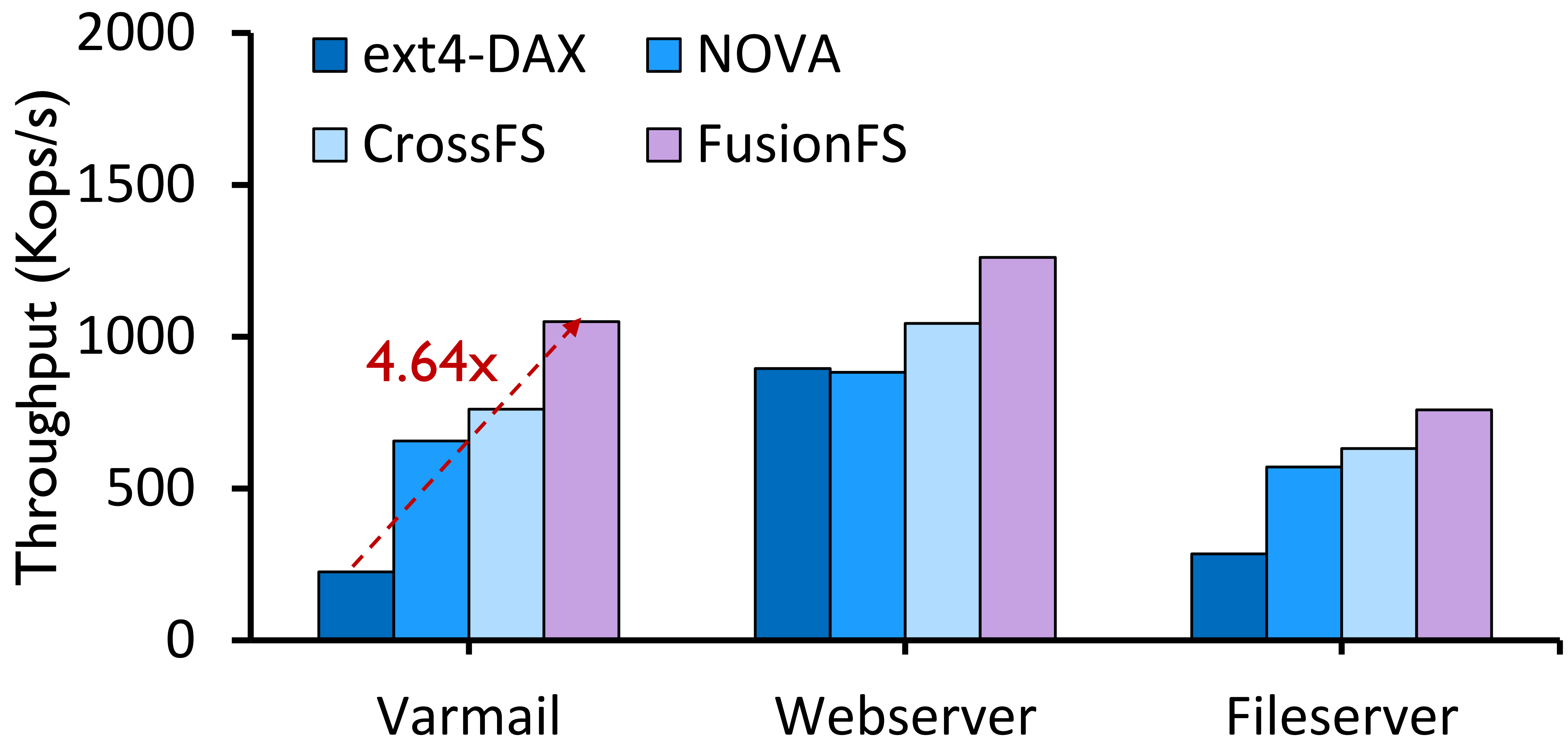
We applied CISCops on KernelFS: ext4-DAX



CISCops shows better performance on KernelFS

Macro-benchmark: Filebench

For each workload, FusionFS will aggregated some common IO sequence to CISCops. (e.g., open-write-close)



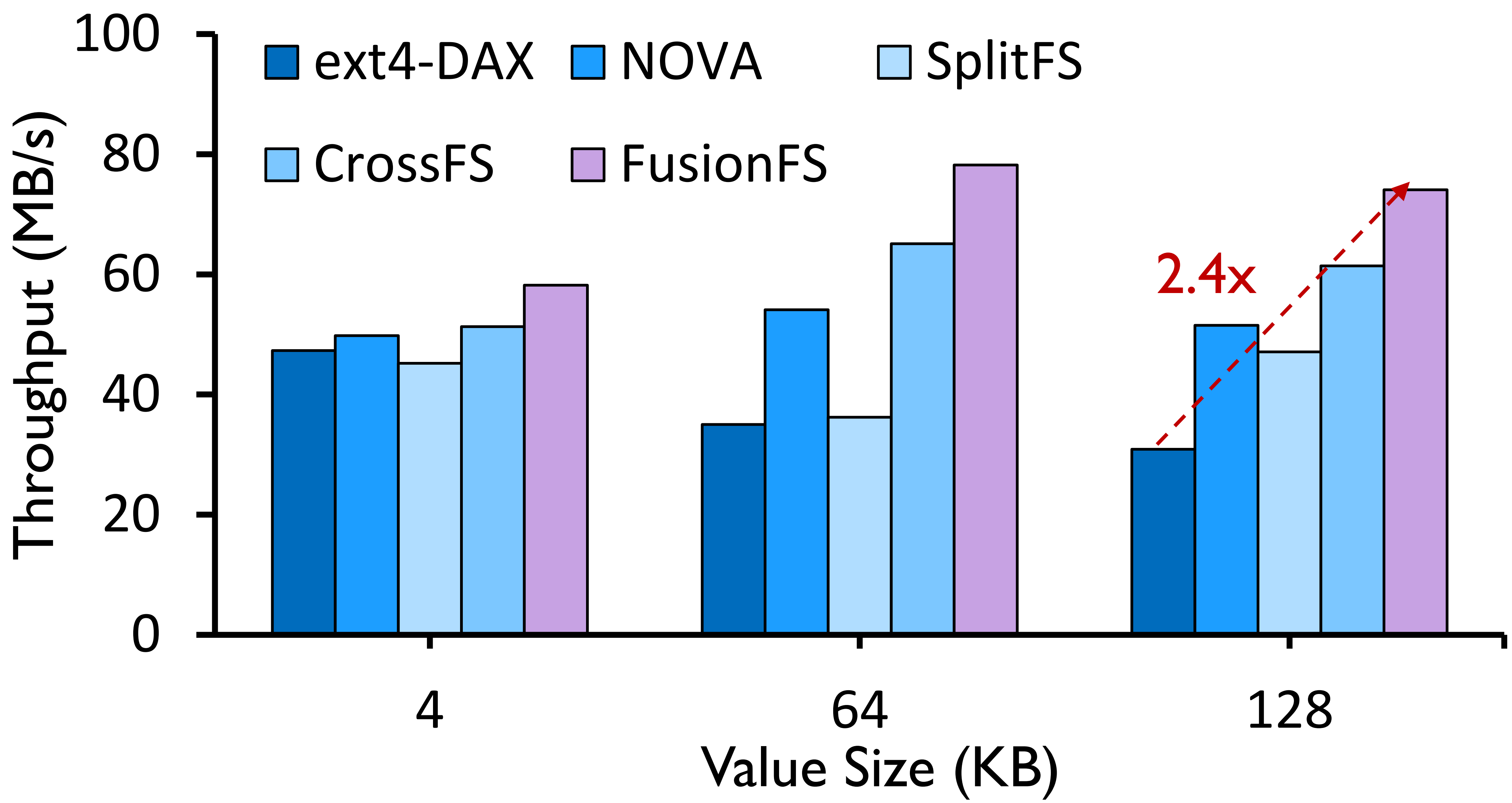
FusionFS shows promising speedup with all the workloads

Evaluation Goals

- Understand effectiveness of FusionFS and $CISC_{Ops}$ to reduce I/O overheads
- Study MicroTx's durability and auto-recovery benefits
- Evaluate effectiveness of CFS scheduler for resource fairness across tenants?
- **Discuss overall real-world application impact**

Application - LevelDB

DBbench's random write workload by replacing checksum logic with *append-CRC-write CSICops*



FusionFS also shows high performance in LevelDB

Summary

- Motivation
  - Reducing I/O overheads such as data copy, system calls, and PCI costs critical
  - Leverage in-storage compute for I/O and data processing is critical!

- Solution – **FusionFS**
  - Fuse I/O and data processing operations into one ($CISC_{Ops}$) and offload
  - CFS I/O scheduler for fairness across multiple tenants
  - MicroTx supports crash consistency and fast recovery

- Evaluation
  - FusionFS shows up to 4x micro-benchmark performance gains
  - Shows up to 2x application performance gains

Conclusion

- We believe it is critical to utilize in-storage resources to reduce I/O latency
- It is time for richer I/O abstractions that organically supports data processing
- Using **CISCops**, we take the first steps towards richer I/O abstractions
- We observe, efficient utilization of in-storage resources are critical for addressing durability and resource management challenges

Source code available at

https://github.com/RutgersCSSystems/FusionFS

Thanks! Questions?

sudarsun.kannan@rutgers.edu

PERSISTENT MEMORY + SUMMIT 2022 COMPUTATIONAL STORAGE

VIRTUAL EVENT • MAY 24-25, 2022

Please take a moment to rate this session.

- Your feedback is important to us.

