



Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs

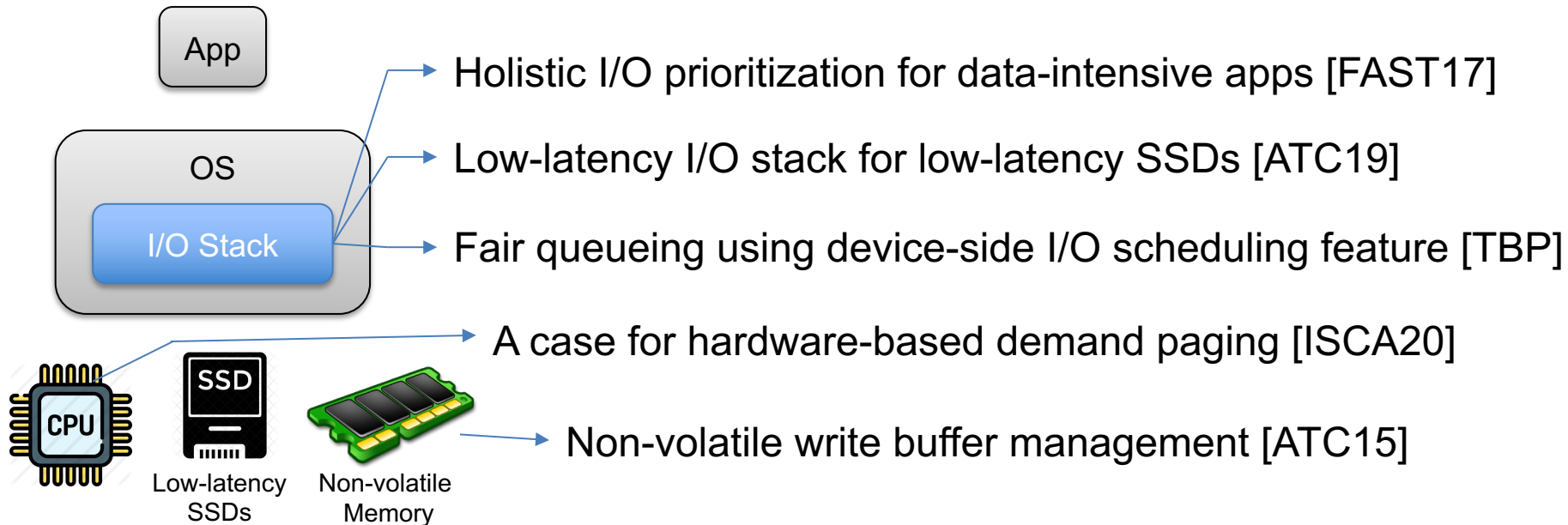
Jinkyu Jeong

jinkyu@skku.edu

Sungkyunkwan University (SKKU)

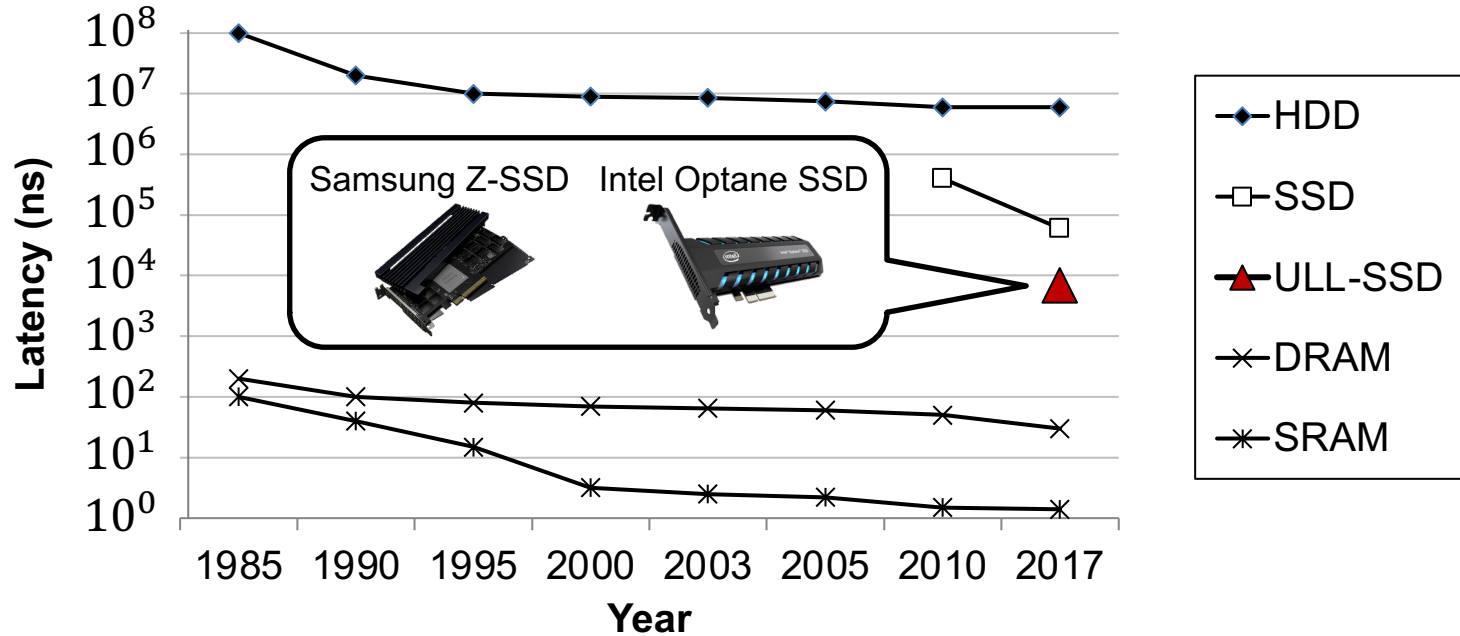
My Recent Research

- Machine virtualization for cloud computing
- Memory management for mobile devices
- I/O stack for emerging storage technology



Storage Performance Trends

- Emerging ultra-low latency SSDs deliver I/Os in a few μ s

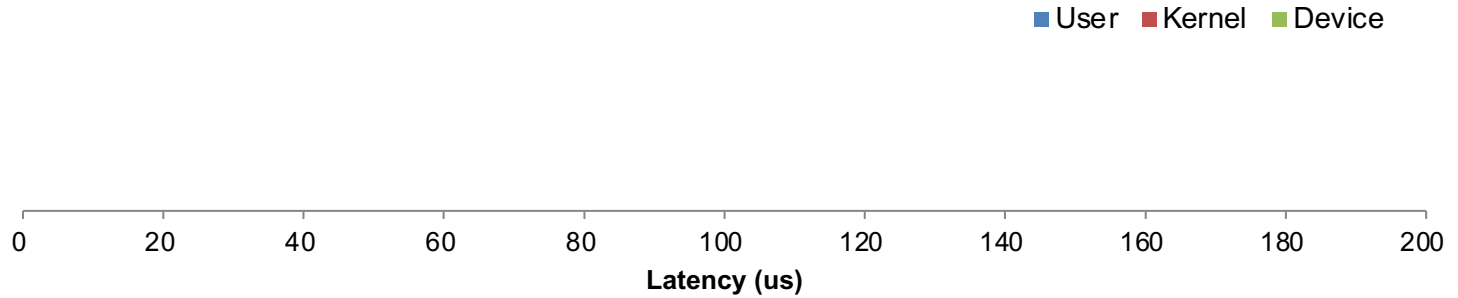


Source: R. E. Bryant and D. R. O'Hallaron, Computer Systems: A Programmer's Perspective, Second Edition, Pearson Education, Inc., 2015

Overhead of Kernel I/O Stack

- Low-latency SSDs expose the overhead of kernel I/O stack

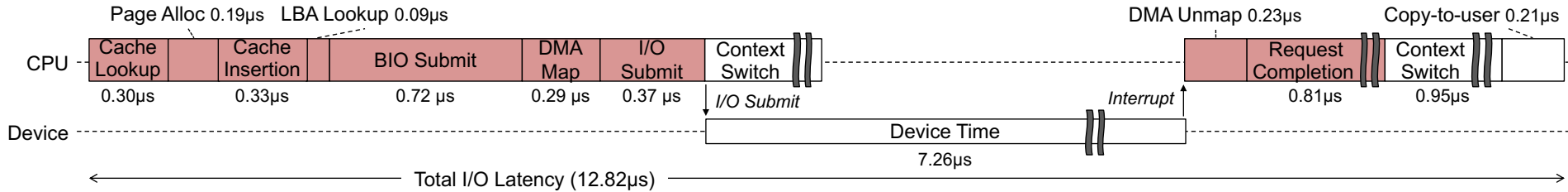
4KB read()



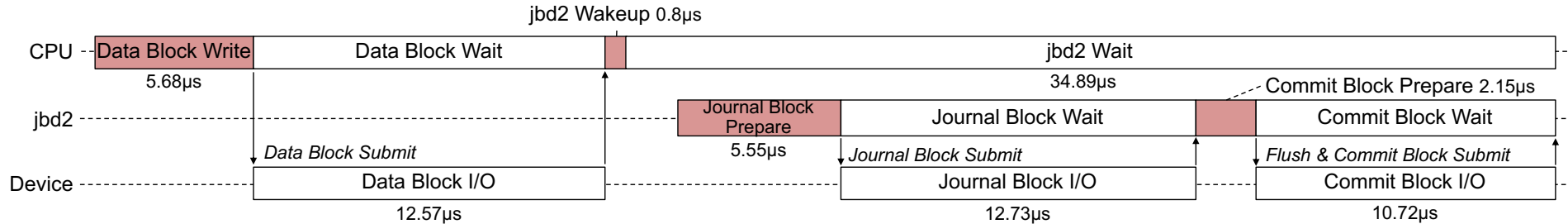
Problem: kernel I/O stack accounts for a large fraction in total I/O latency

Breakdown of I/O Stack Operations

• Read()

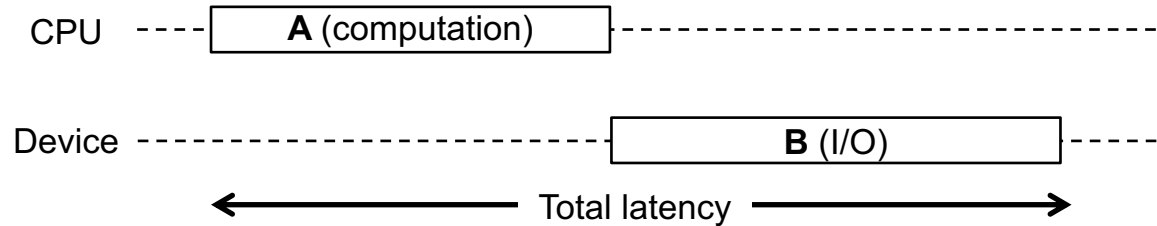


• Write() + fsync()

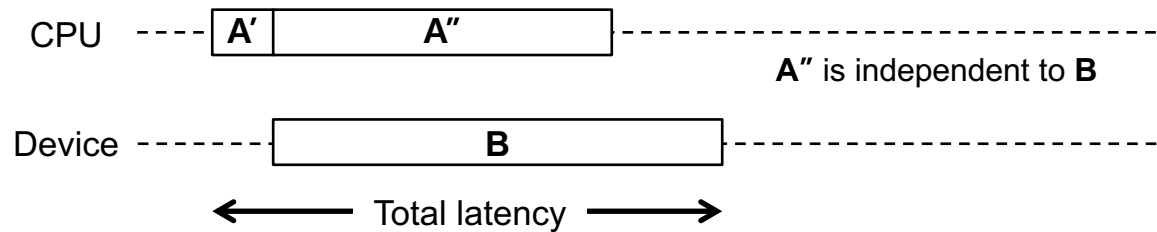


Synchronous I/O vs. Asynchronous I/O

Synchronous I/O



Asynchronous I/O

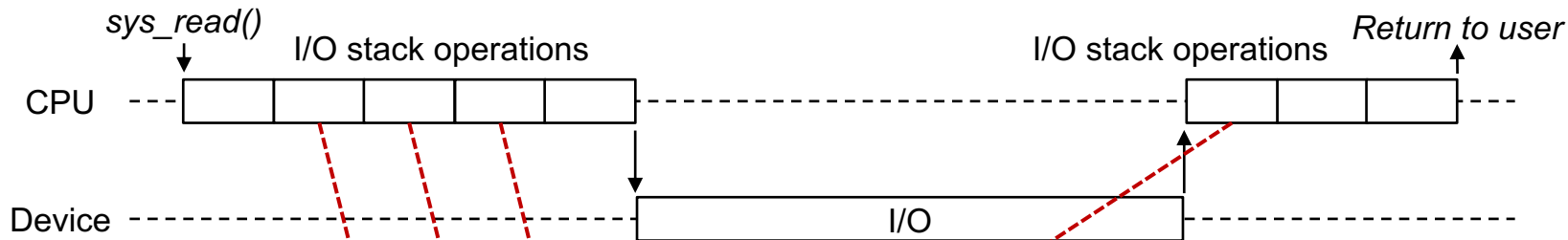


Throughput  Total latency 

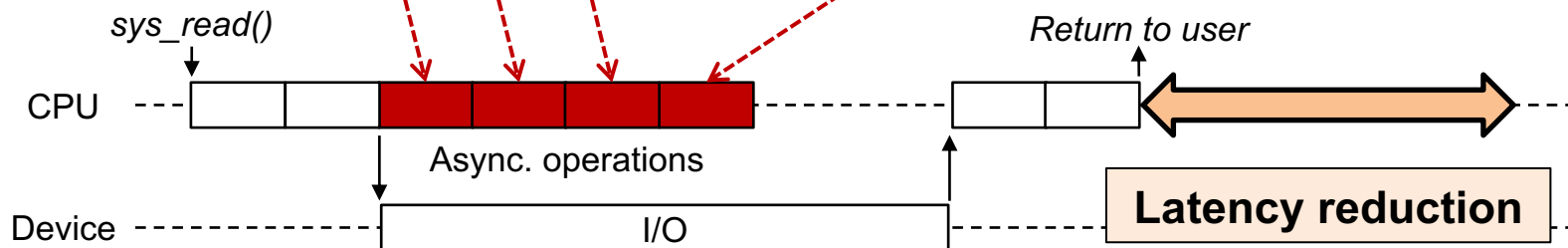
Our Idea: apply asynchronous I/O concept to the I/O stack itself

Read Path Overview

Vanilla Read Path



Proposed Read Path



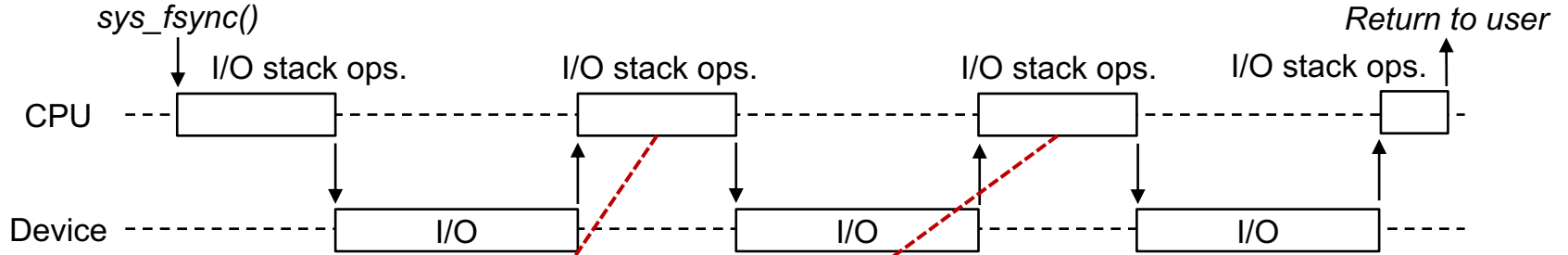
Write Path Overview

Vanilla Write Path

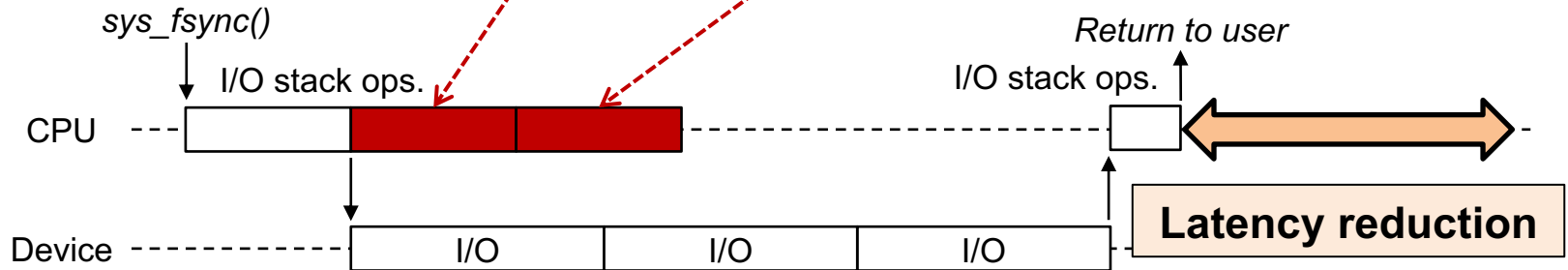


Write Path Overview

Vanilla Fsync Path



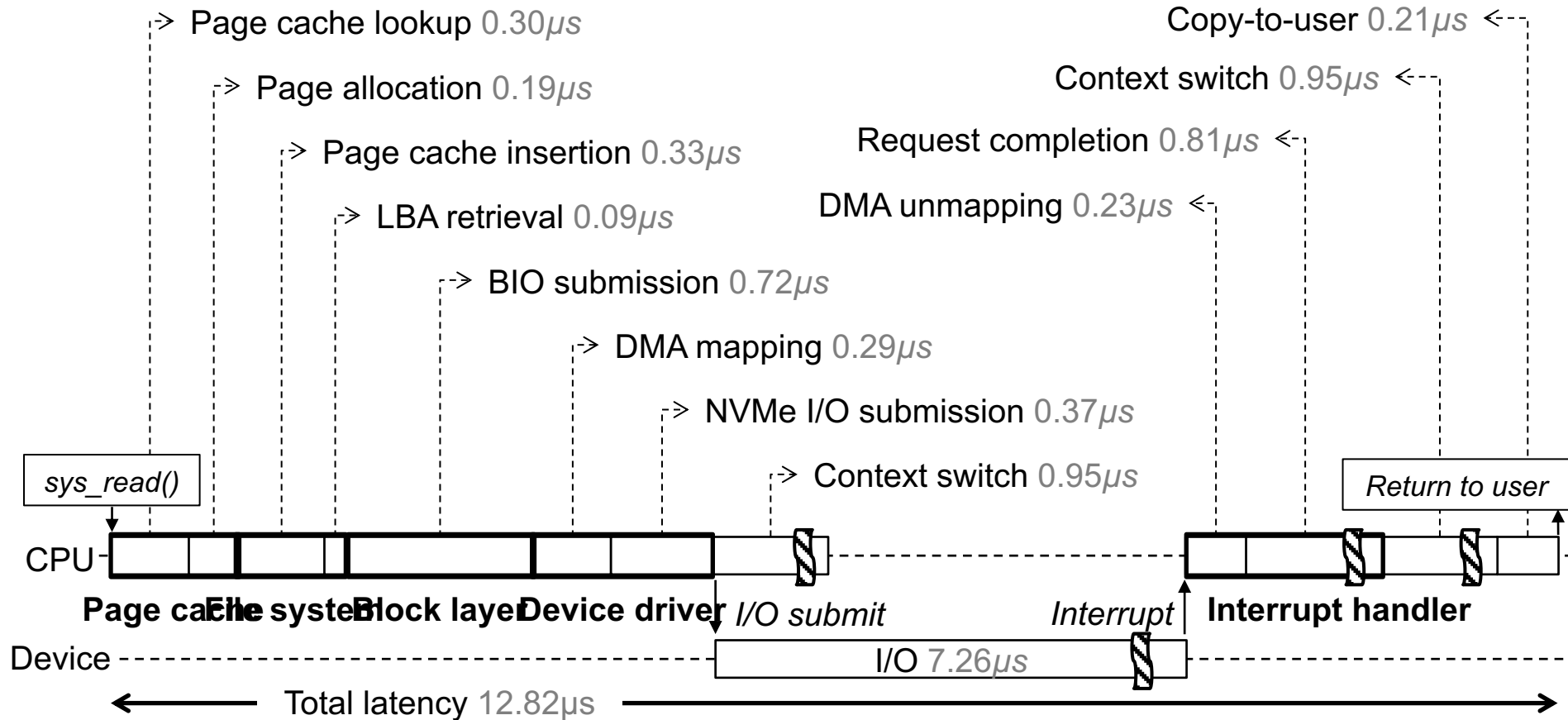
Proposed Fsync Path



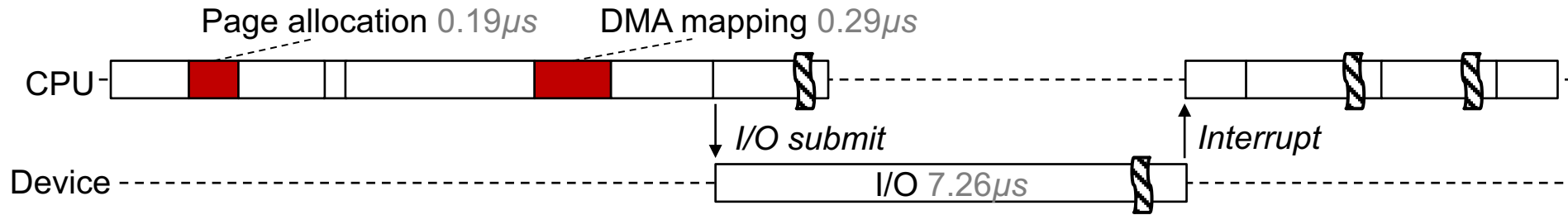
Agenda

- **Read path**
 - Analysis of vanilla read path
 - Proposed read path
- **Light-weight block I/O layer**
- **Write path**
 - Analysis of vanilla write path
 - Proposed write path
- **Evaluation**
- **Conclusion**

Analysis of Vanilla Read Path

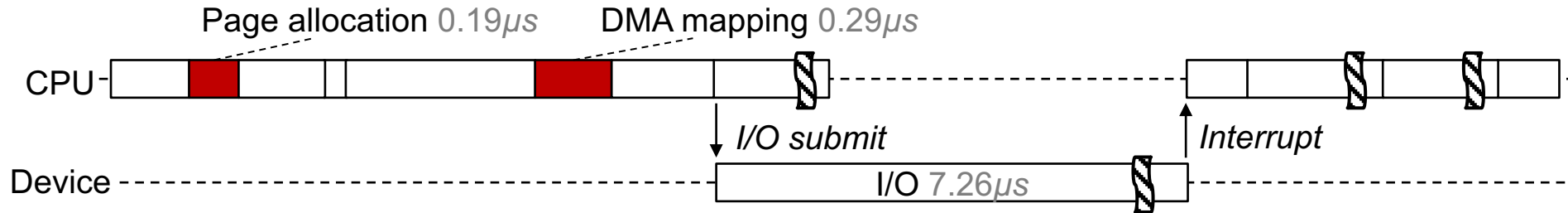
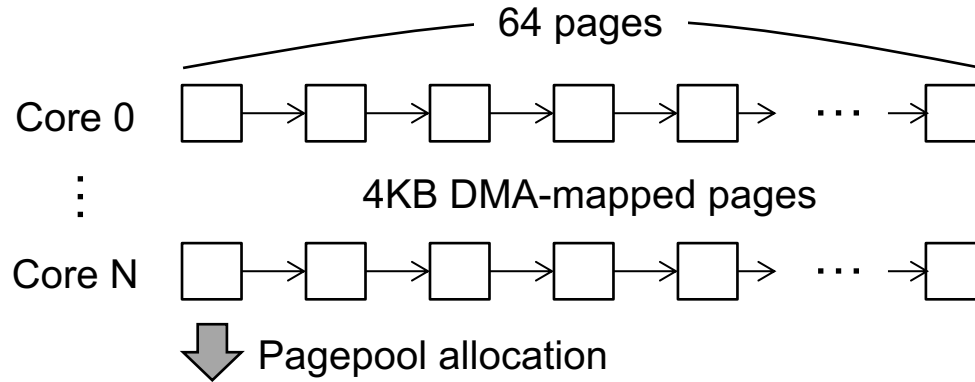


Page Allocation / DMA Mapping



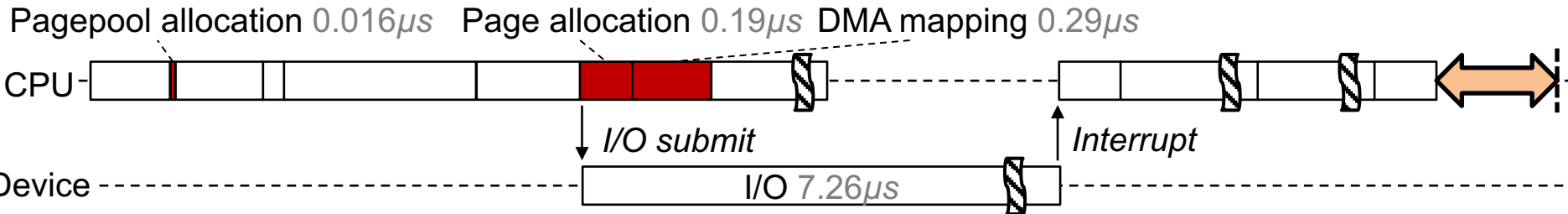
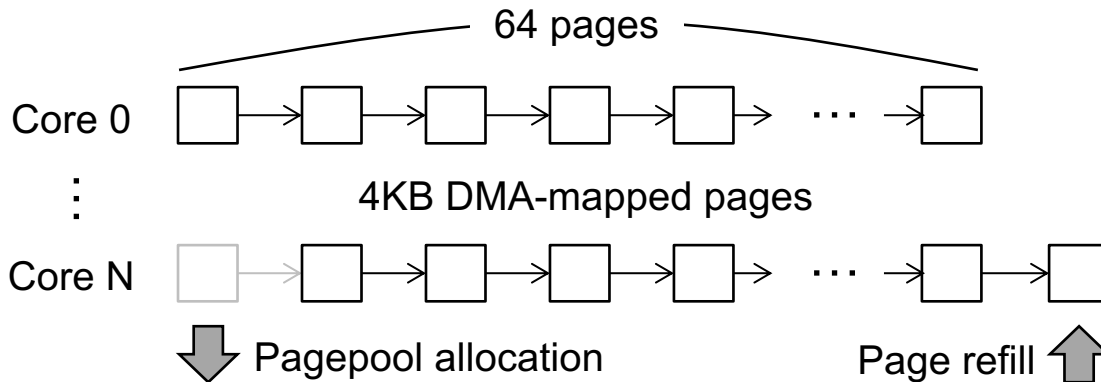
Asynchronous Page Allocation / DMA Mapping

- **DMA-mapped page pool**



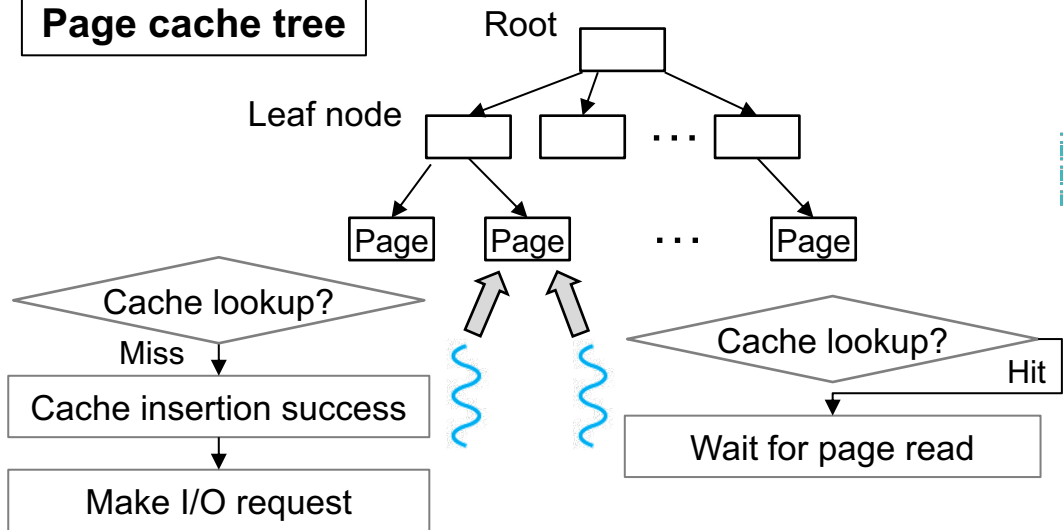
Asynchronous Page Allocation / DMA Mapping

- **DMA-mapped page pool**



Page Cache Insertion

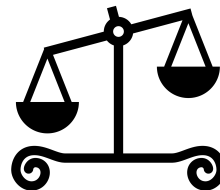
Page cache tree



Page cache lookup overhead

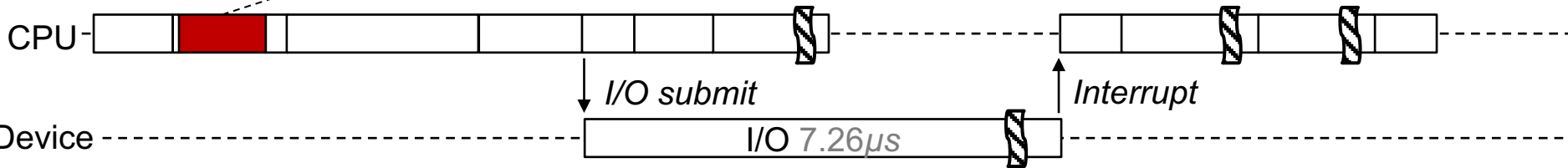


Page cache tree extension overhead



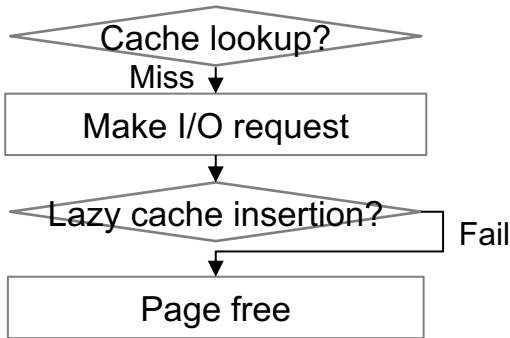
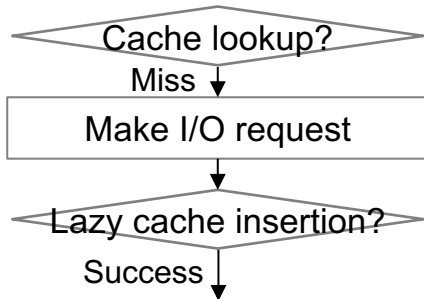
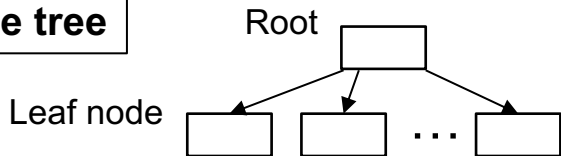
Prevention from duplicated I/O requests for the same file index

Page cache insertion $0.33\mu s$



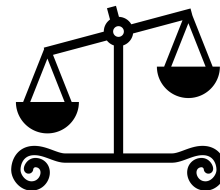
Lazy Page Cache Insertion

Page cache tree



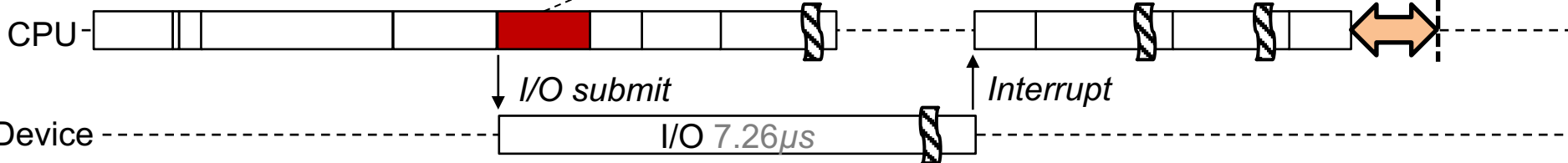
 ~~Page cache lookup overhead~~

 ~~Page cache tree extension overhead~~

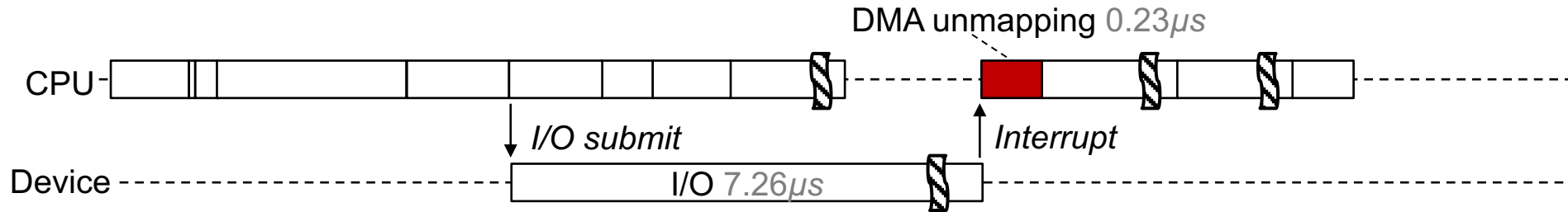


**Duplicated I/O requests
(extremely low frequency)**

Page cache insertion $0.35\mu s$



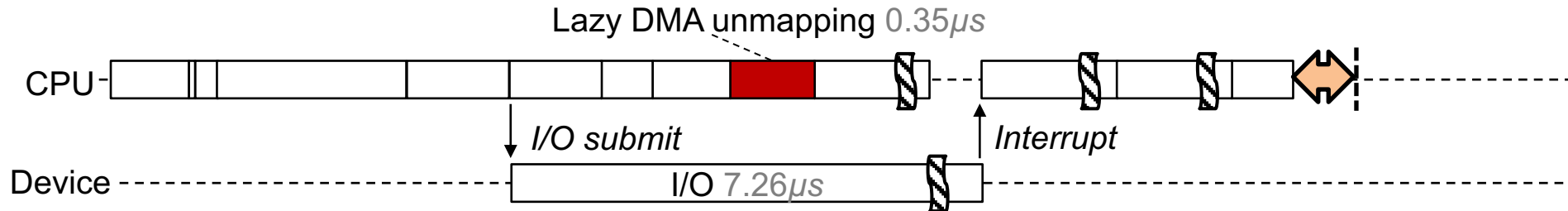
DMA Unmapping



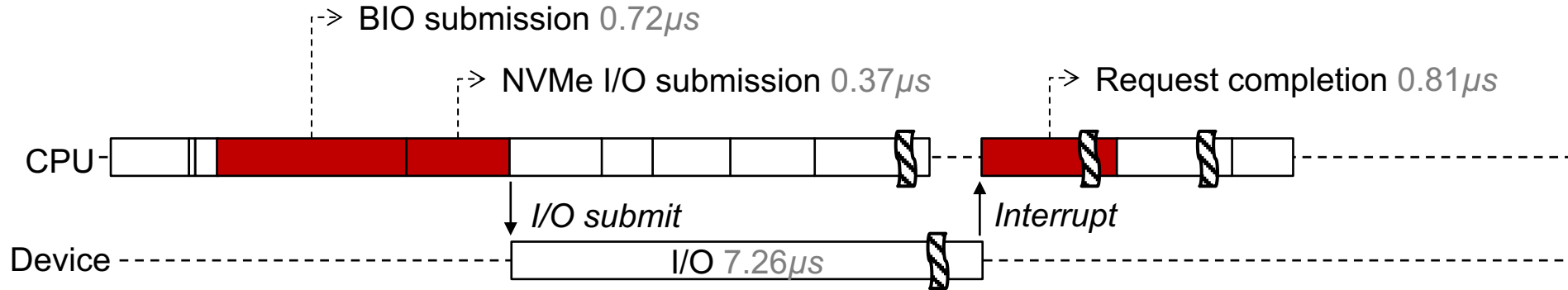
Lazy DMA Unmapping

- **Implementation**

- Delays DMA unmapping to when a system is idle or waiting for another I/O requests
- Extended version of the deferred protection scheme in Linux [ASPLOS '16]
- Optionally disabled for safety



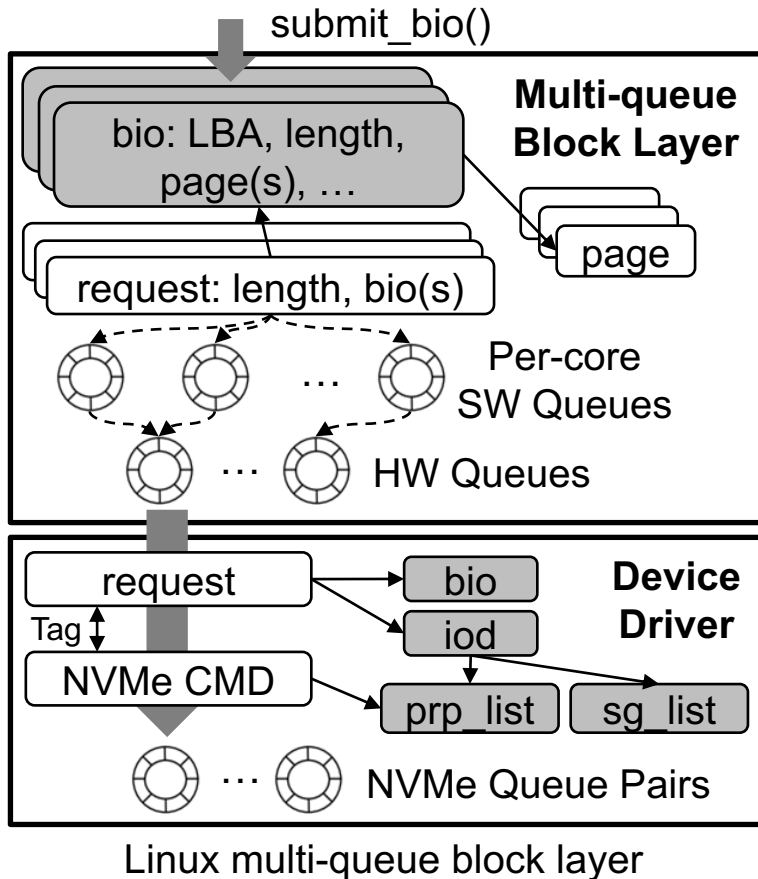
Remaining Overheads in the Proposed Read Path



Agenda

- **Read path**
 - Analysis of vanilla read path
 - Proposed read path
- **Light-weight block I/O layer**
- **Write path**
 - Analysis of vanilla write path
 - Proposed write path
- **Evaluation**
- **Conclusion**

Linux Multi-queue Block I/O Layer



- **Structure conversion**

- Merge bio with pending request via I/O merging
- Assign new tag & request and convert from bio

- **Multi-queue structure**

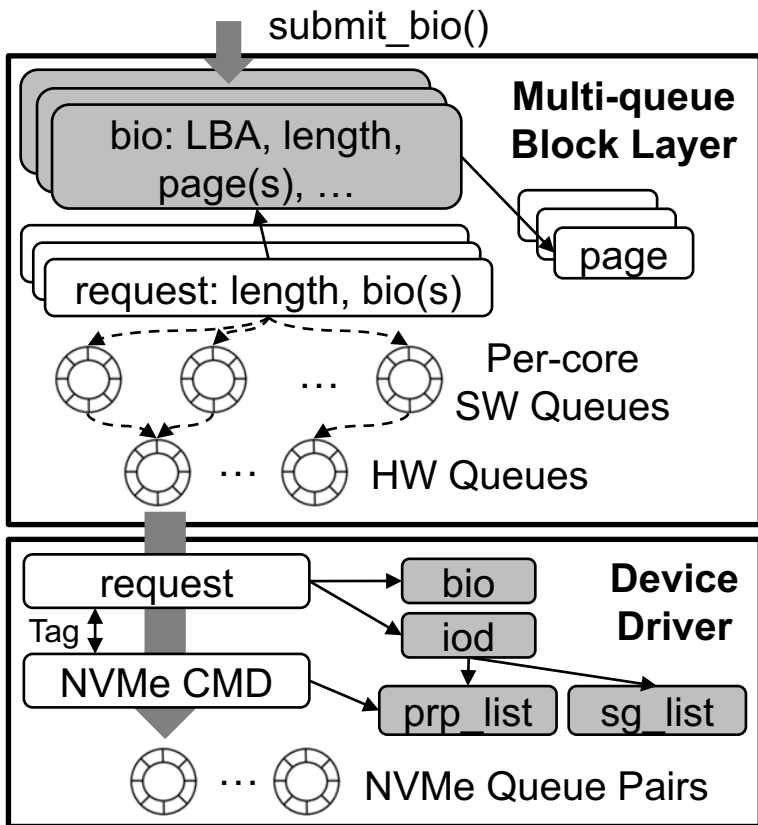
- Software staging queue (SW queue)
 - ✓ Support I/O scheduling and reordering
- Hardware dispatch queue (HW queue)
 - ✓ Deliver the I/O request to the device driver

- **Multiple dynamic memory allocations**

- Bio (block layer)
- NVMe iod, scatter/gather list, NVMe PRP* list (device driver)

*PRP: physical region page

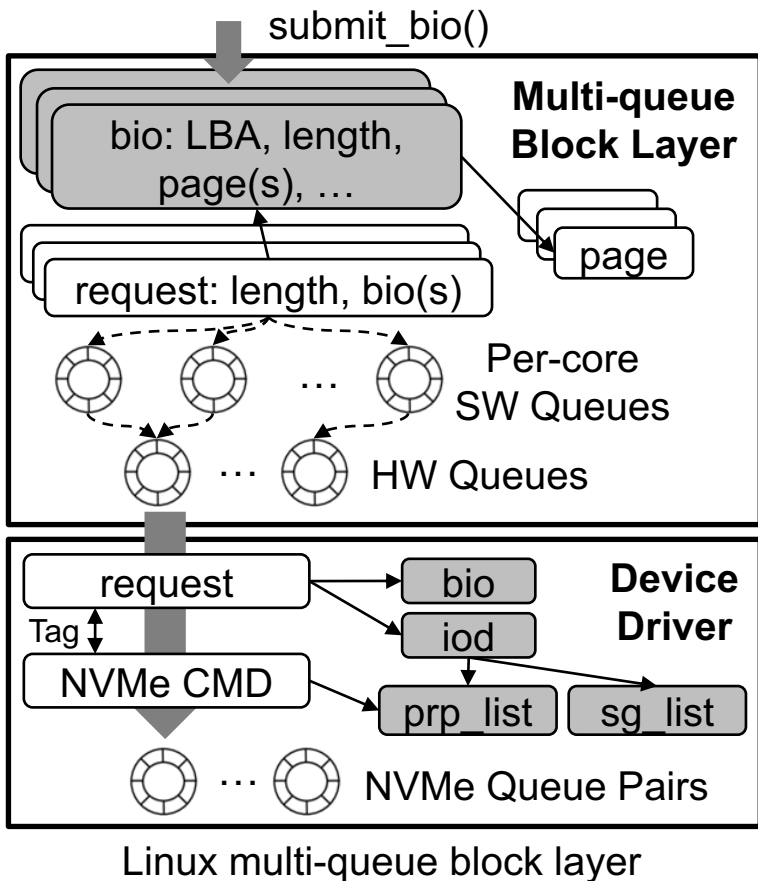
Linux Multi-queue Block I/O Layer



Linux multi-queue block layer

- **Structure conversion**
 - Inefficiency of I/O merging [Zhang, OSDI'18]
 - ✓ Useful feature for low-performance storage device
- **Multi-queue structure**
- **Multiple dynamic memory allocations**

Linux Multi-queue Block I/O Layer



- **Structure conversion**

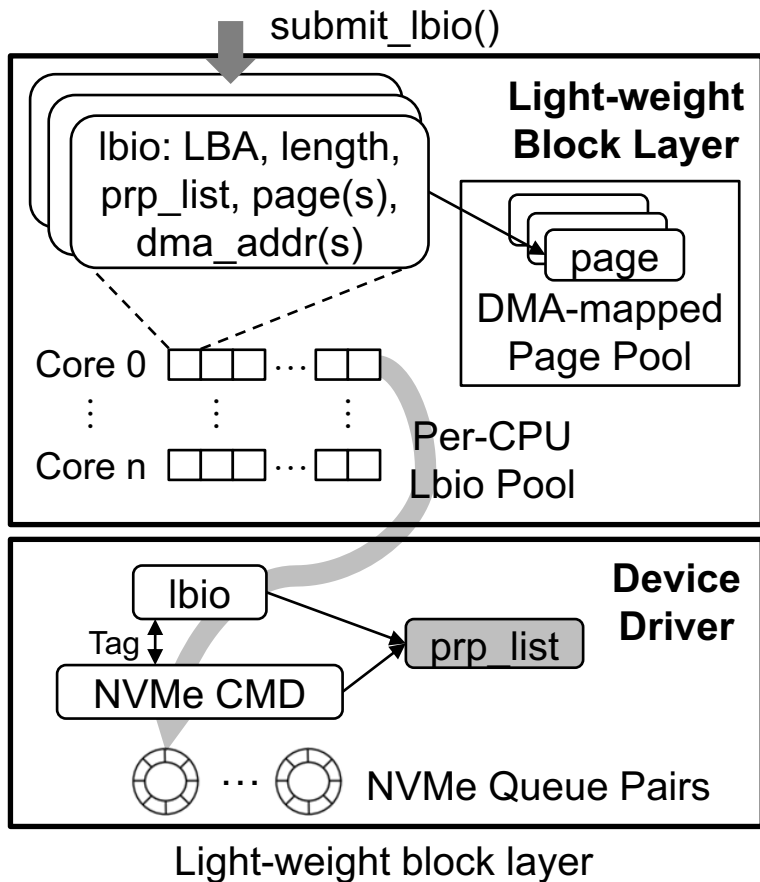
- Inefficiency of I/O merging [Zhang, *OSDI'18*]
 - ✓ Useful feature for low-performance storage device

- **Multi-queue structure**

- Inefficiency of I/O scheduling for low-latency SSDs [Saxena, *ATC'10*] [Xu, *SYSTOR'15*]
 - ✓ Default configuration is noop scheduler
- Bypass multi-queue structure [Zhang, *OSDI'18*]
- Device-side I/O scheduling [Peter, *OSDI'14*] [Joshi, *HotStorage'17*]

- **Multiple dynamic memory allocations**

Light-weight Block I/O Layer



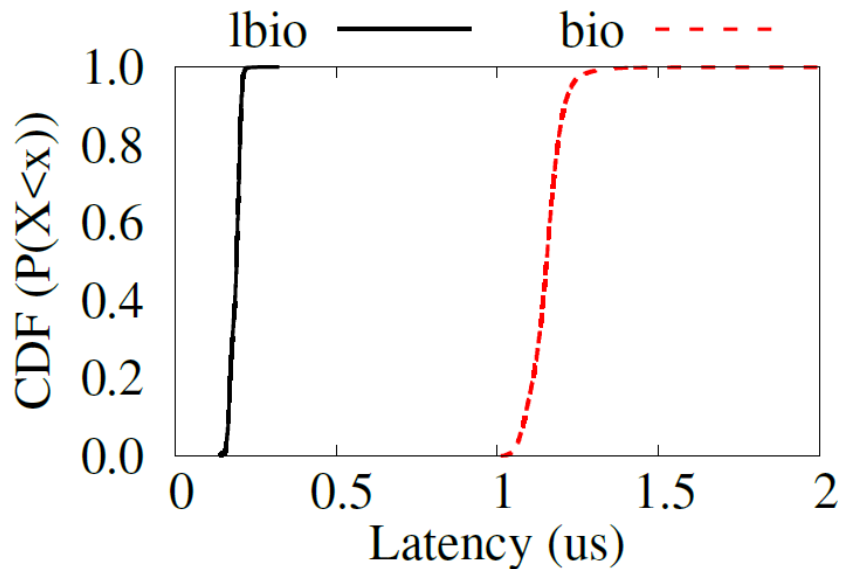
- **Light-weight bio (lbio) structure**
 - Contains only essential arguments for to make NVMe I/O request
 - Eliminates unnecessary structure conversions and allocations
- **Per-CPU lbio pool**
 - Supports lockless lbio object allocation
 - Supports tagging function
- **Single dynamic memory allocation**
 - NVMe PRP* list (device driver)

*PRP: physical region page

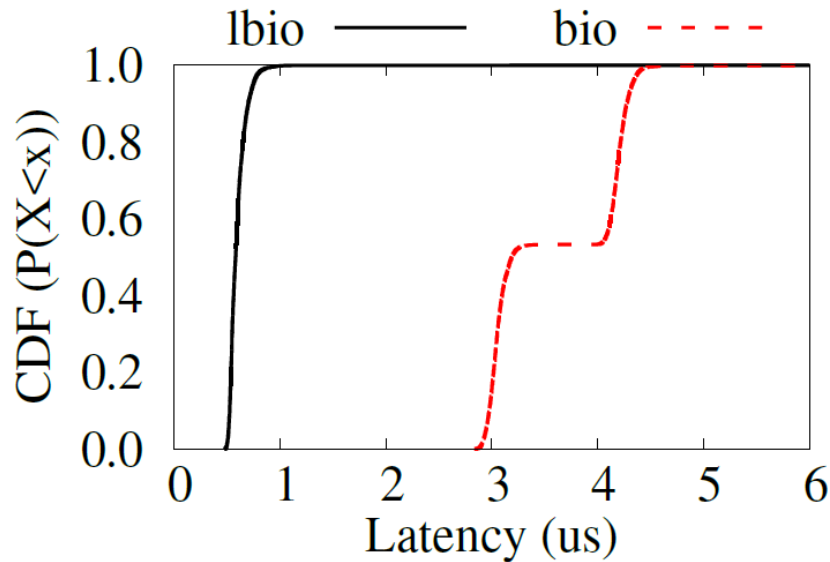
CDF of the Block I/O Submission Latency

- **Block I/O submission latency**

- Time from allocating a object (bio or lbio) to dispatching an I/O command to a device



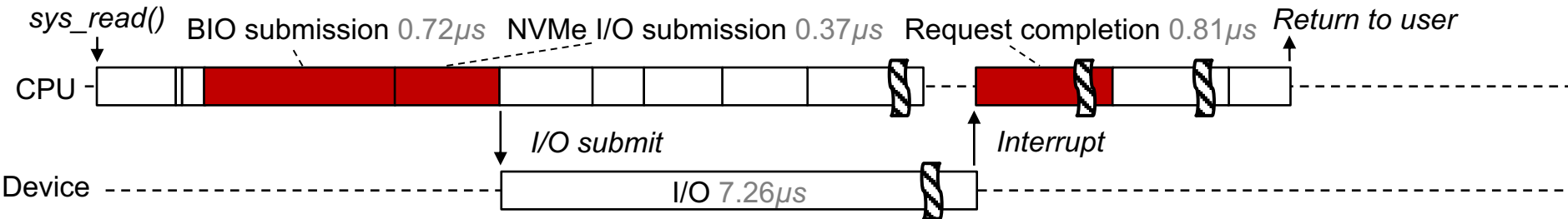
4KB random read



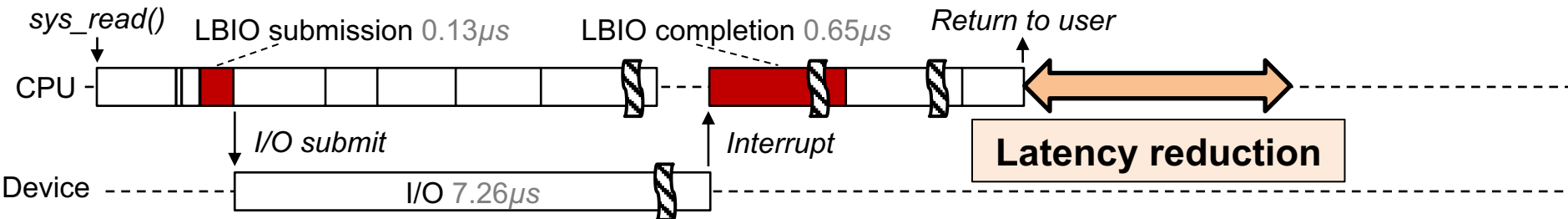
32KB random read

Read Path Comparison

Proposed Read Path (before applying light-weight block I/O layer)

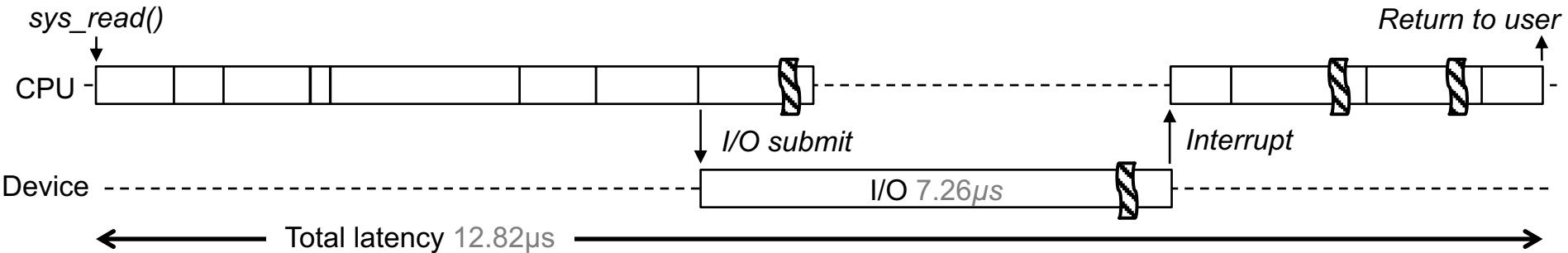


Proposed Read Path

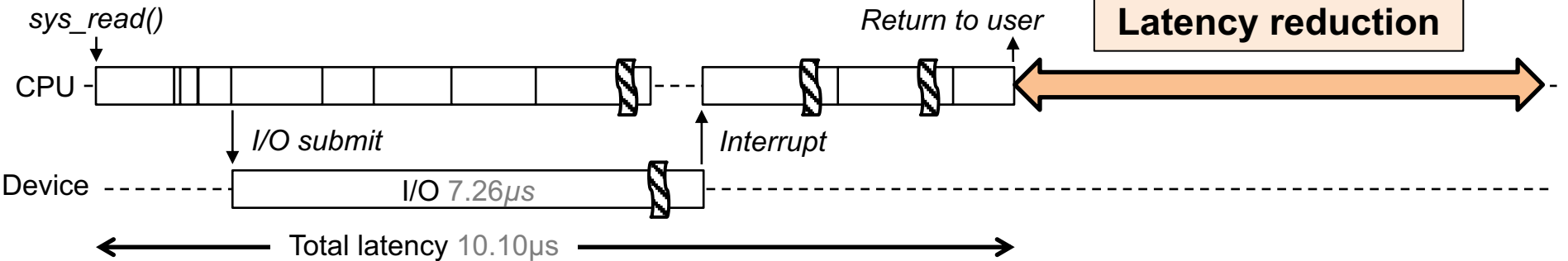


Read Path Comparison

Vanilla Read Path



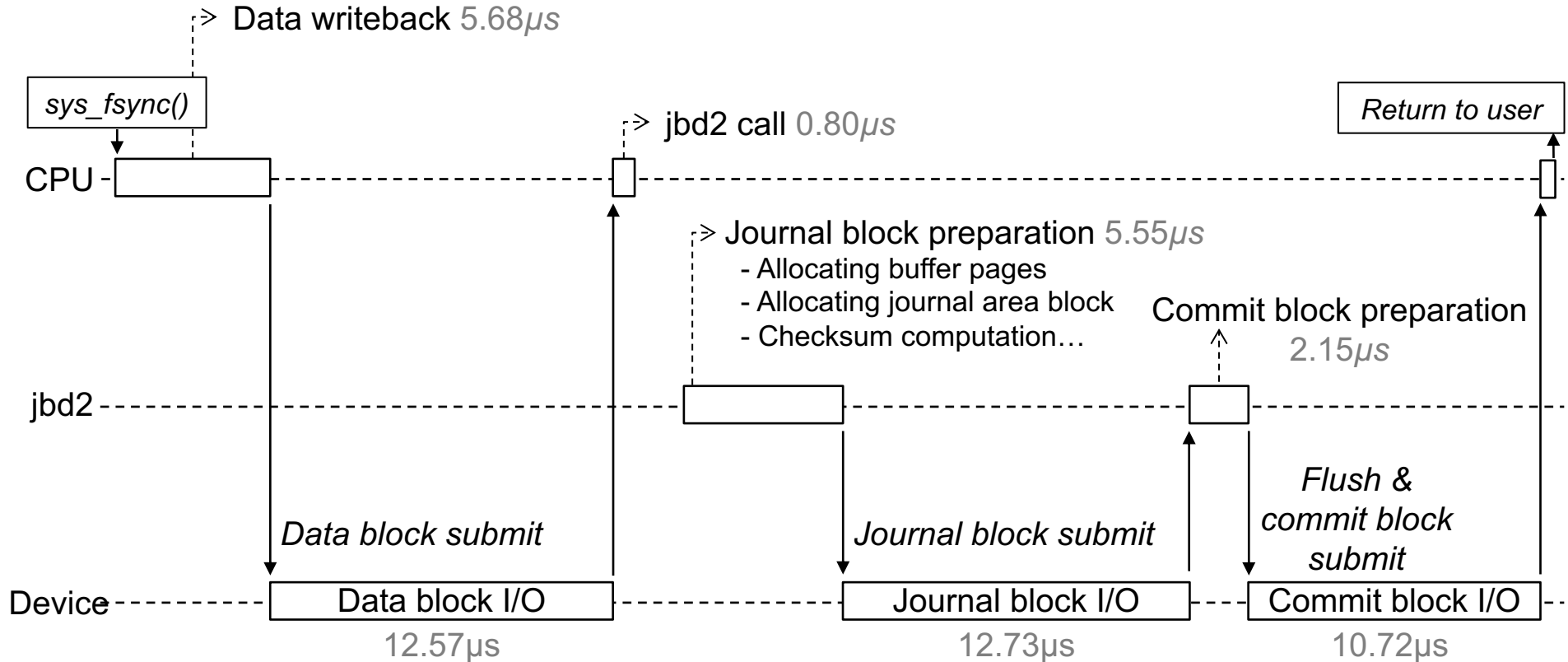
Proposed Read Path



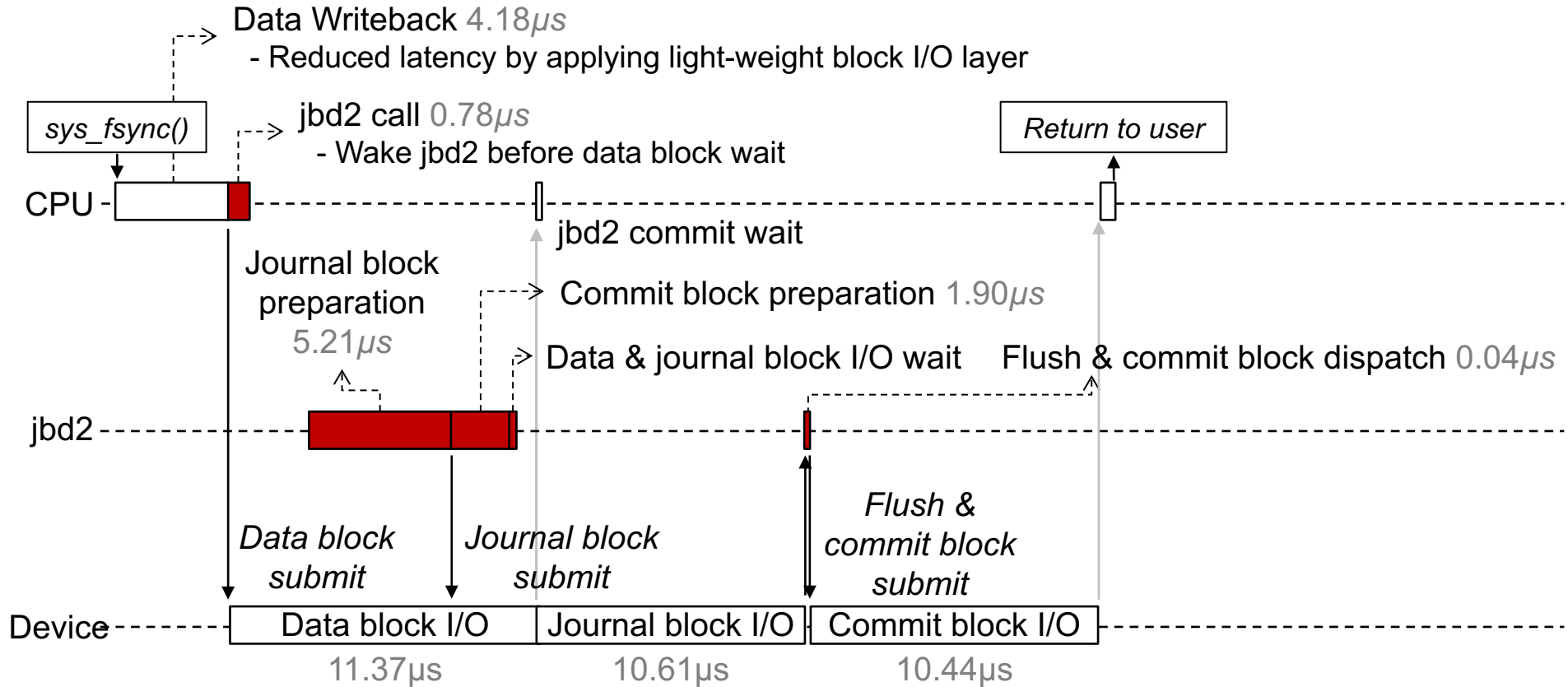
Agenda

- **Read path**
 - Analysis of vanilla read path
 - Proposed read path
- **Light-weight block I/O layer**
- **Write path**
 - Analysis of vanilla write path
 - Proposed write path
- **Evaluation**
- **Conclusion**

Analysis of Vanilla Fsync Path (Ext4 Ordered Mode)

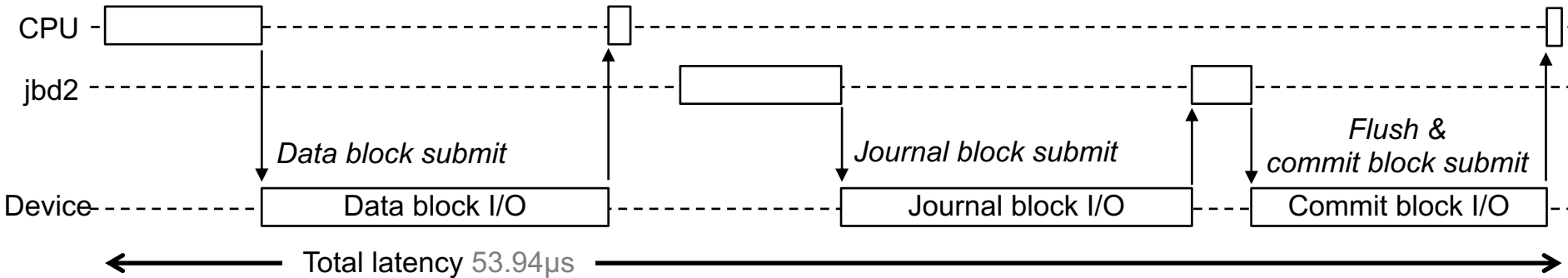


Proposed Fsync Path (Ext4 Ordered Mode)

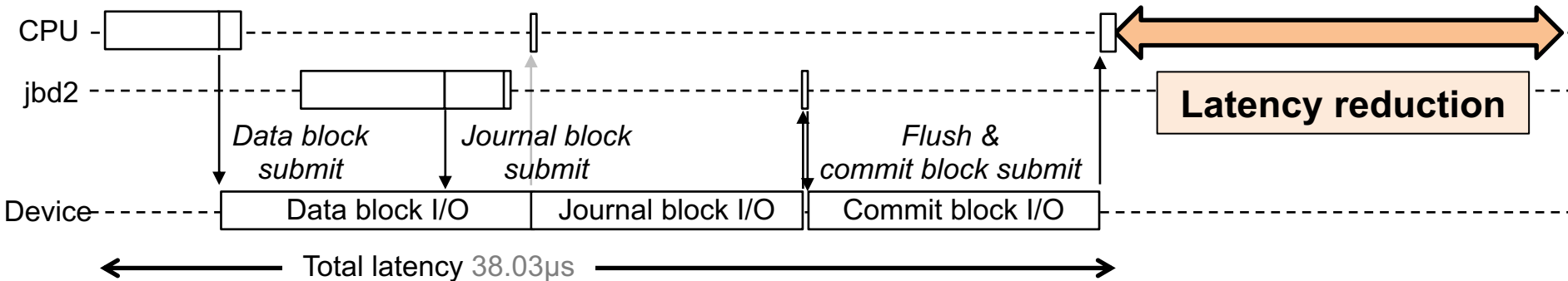


Fsync Path Comparison (Ext4 Ordered Mode)

Vanilla Fsync Path



Proposed Fsync Path



Agenda

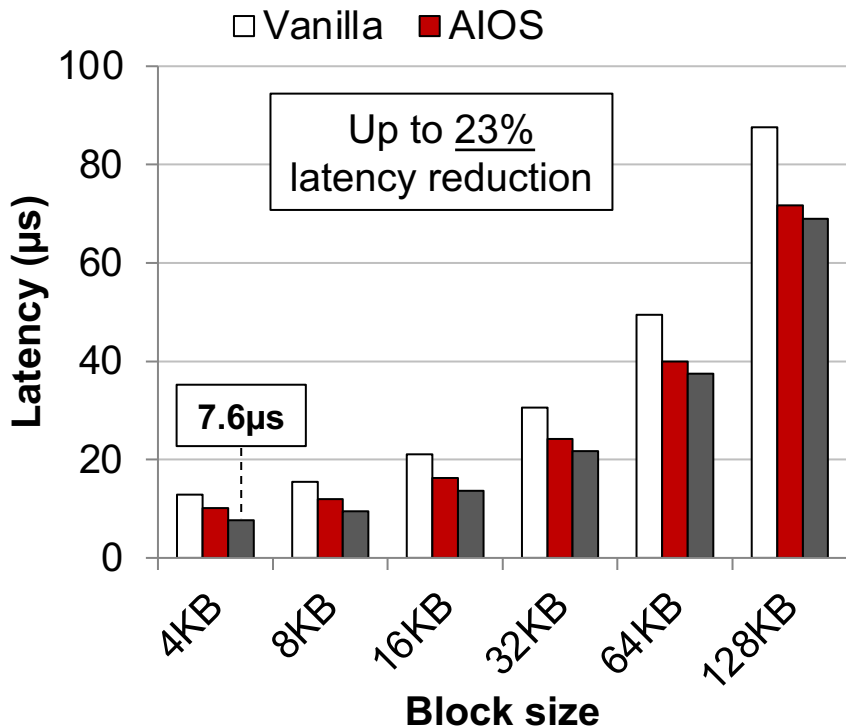
- **Read path**
 - Analysis of vanilla read path
 - Proposed read path
- **Light-weight block I/O layer**
- **Write path**
 - Analysis of vanilla write path
 - Proposed write path
- **Evaluation**
- **Conclusion**

Experimental Setup

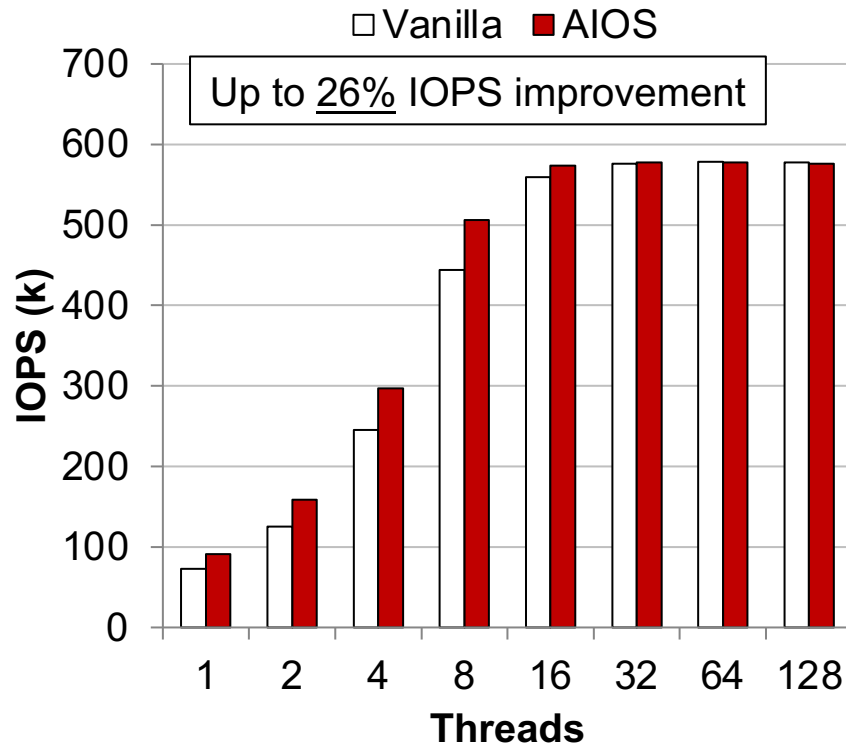
Server	Dell R730
OS	Ubuntu 16.04.4
Base kernel	Linux 5.0.5
CPU	Intel Xeon E5-2640v3 2.6GHz 8-cores
Memory	DDR4 32GB
Storage devices	Z-SSD: Samsung SZ985 800GB <u>Optane SSD: Intel Optane 905P 960GB</u>
Workloads	Synthetic micro-benchmark: FIO Real-world workload: RocksDB DBbench

FIO Performance (Random Read)

- **Single thread**

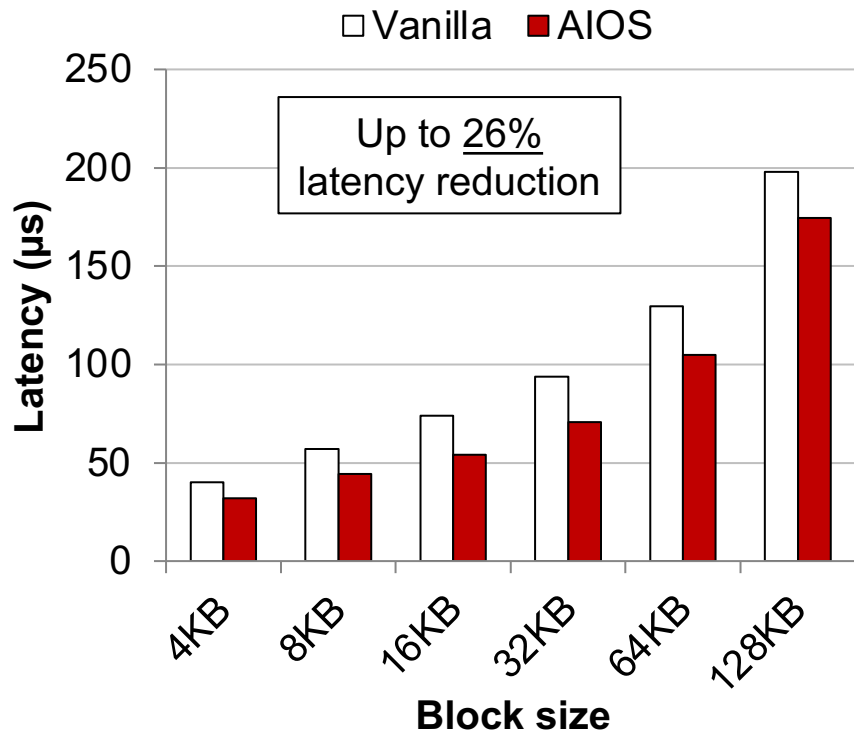


- **4KB block size**

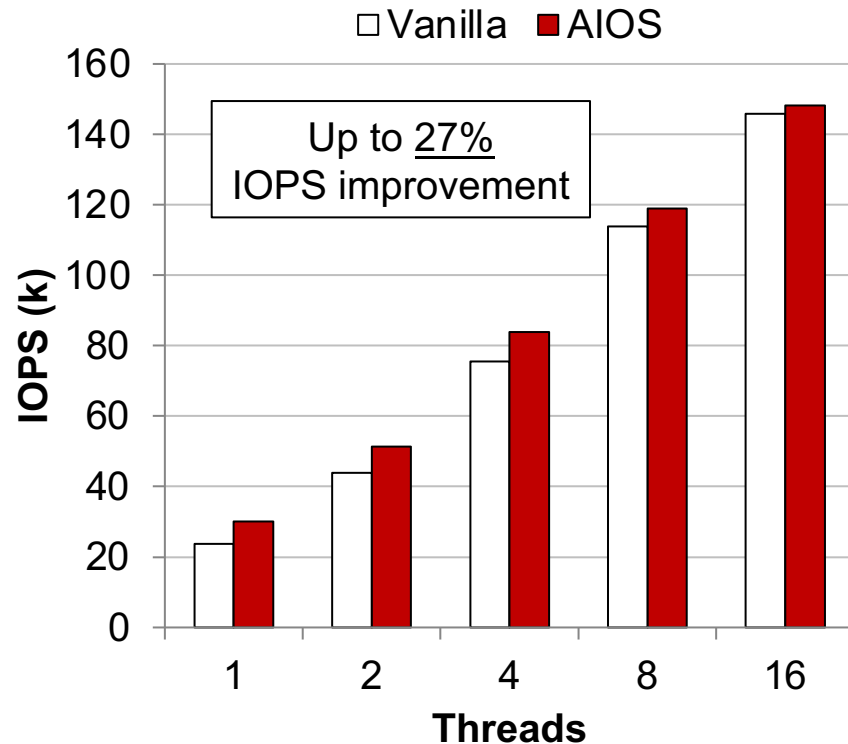


FIO Performance (Random Write+Fsync, Ext4 Ordered)

- **Single thread**



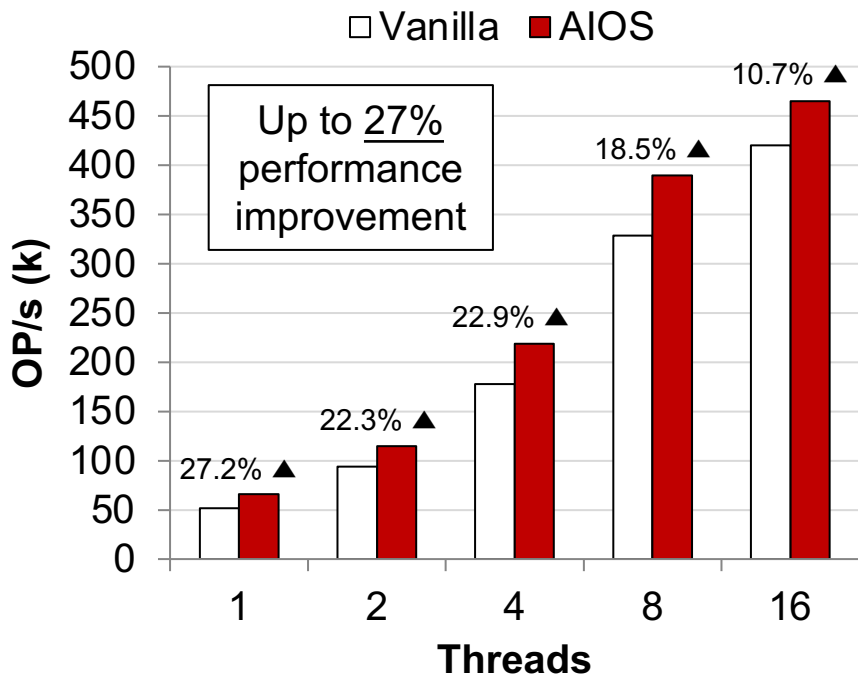
- **4KB block size**



RocksDB Performance

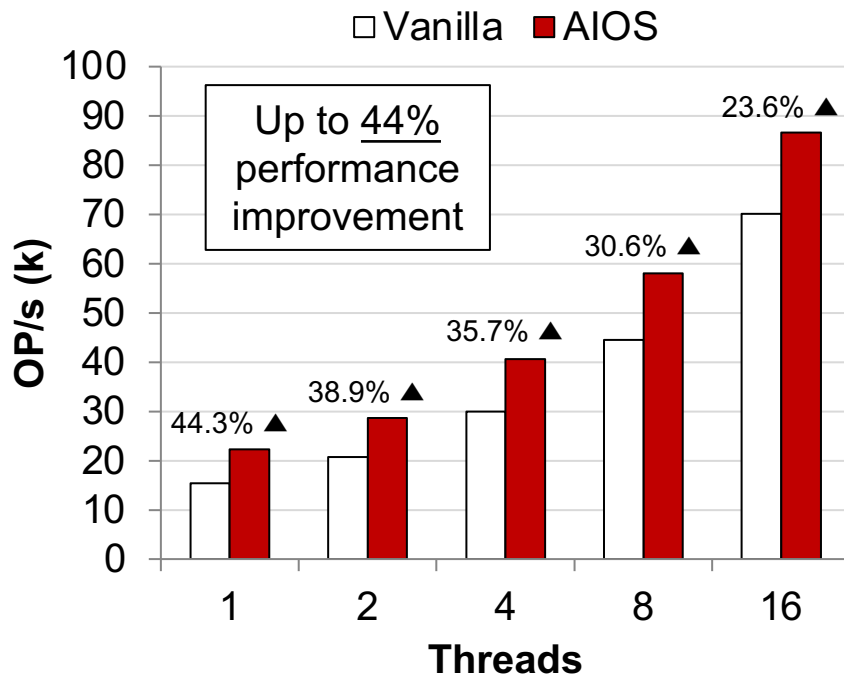
- **DBbench readrandom**

– 64GB dataset

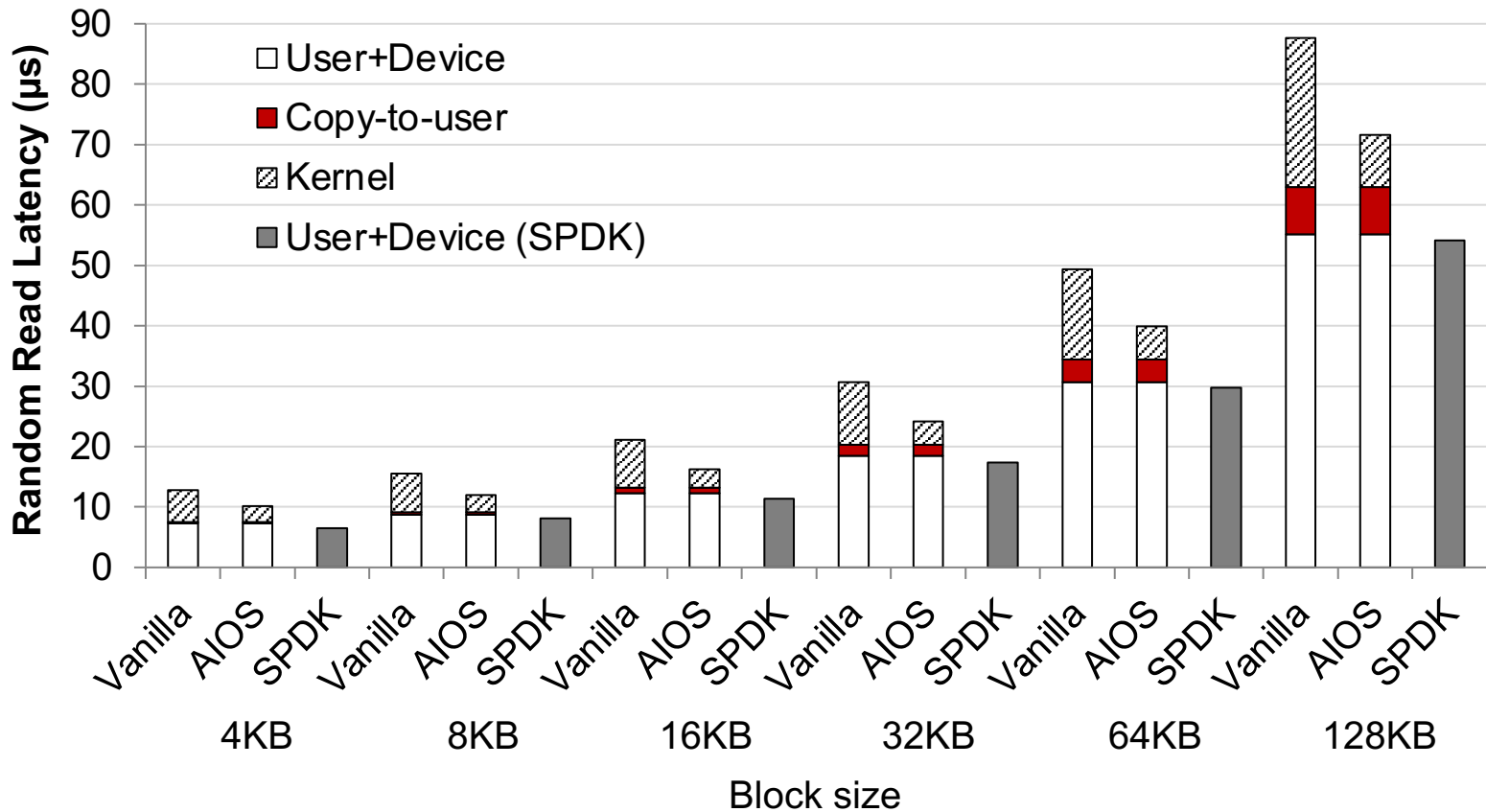


- **DBbench fillsync**

– 16GB dataset



Comparison with User-level Direct Access Approach



Conclusion

- **Asynchronous I/O stack**

- Applies asynchronous I/O concept to the kernel I/O stack itself
- Overlaps computation with I/O to reduce total I/O latency

- **Light-weight block I/O layer**

- Provides low-latency block I/O services for low-latency NVMe SSDs

- **Performance evaluation**

- Achieves a single-digit microsecond I/O latency on Optane SSD
- Achieves significant latency reduction and performance improvement on real-world workloads

Source code: <https://github.com/skkucsl/aio>

Q&A

- **Thank you**