

# Storage Systems (StoSys)

## XM\_0092

### Lecture 11: CXL and io\_uring

Animesh Trivedi

<https://stonet-research.github.io/>

Autumn 2023, Period 1

# Syllabus outline

- ~~1. Welcome and introduction to NVM~~
- ~~2. Host interfacing and software implications~~
- ~~3. Flash Translation Layer (FTL) and Garbage Collection (GC)~~
- ~~4. NVM Block Storage File systems~~
- ~~5. NVM Block Storage Key-Value Stores~~
- ~~6. Emerging Byte-addressable Storage~~
- ~~7. Networked NVM Storage~~
- ~~8. Trends: Specialization and Programmability~~
- ~~9. Distributed Storage / Systems - I~~
- ~~10. Distributed Storage / Systems - II~~
11. Emerging Topics 

# Today is the last course lecture

We survived, it has been quite fun to teach this course

Hope you also had fun and learn a lot of advancements happening in the area of storage research

In coming days and weeks

- **Next Tuesday:** Milestone 5 interview - **sign up!**
- **Next Wednesday:** Guest Lecture from Nikolas
- **Afterwards:** Prepare for the exam - Good luck !
- **In the End:** We will ask for some feedback on the course
  - Me as a teacher
  - Broadly about the course - *you can be frank!*
  - ***Want to be the TA next year?***



# If you are interested in such research ...

Individual research projects (XM\_405088)

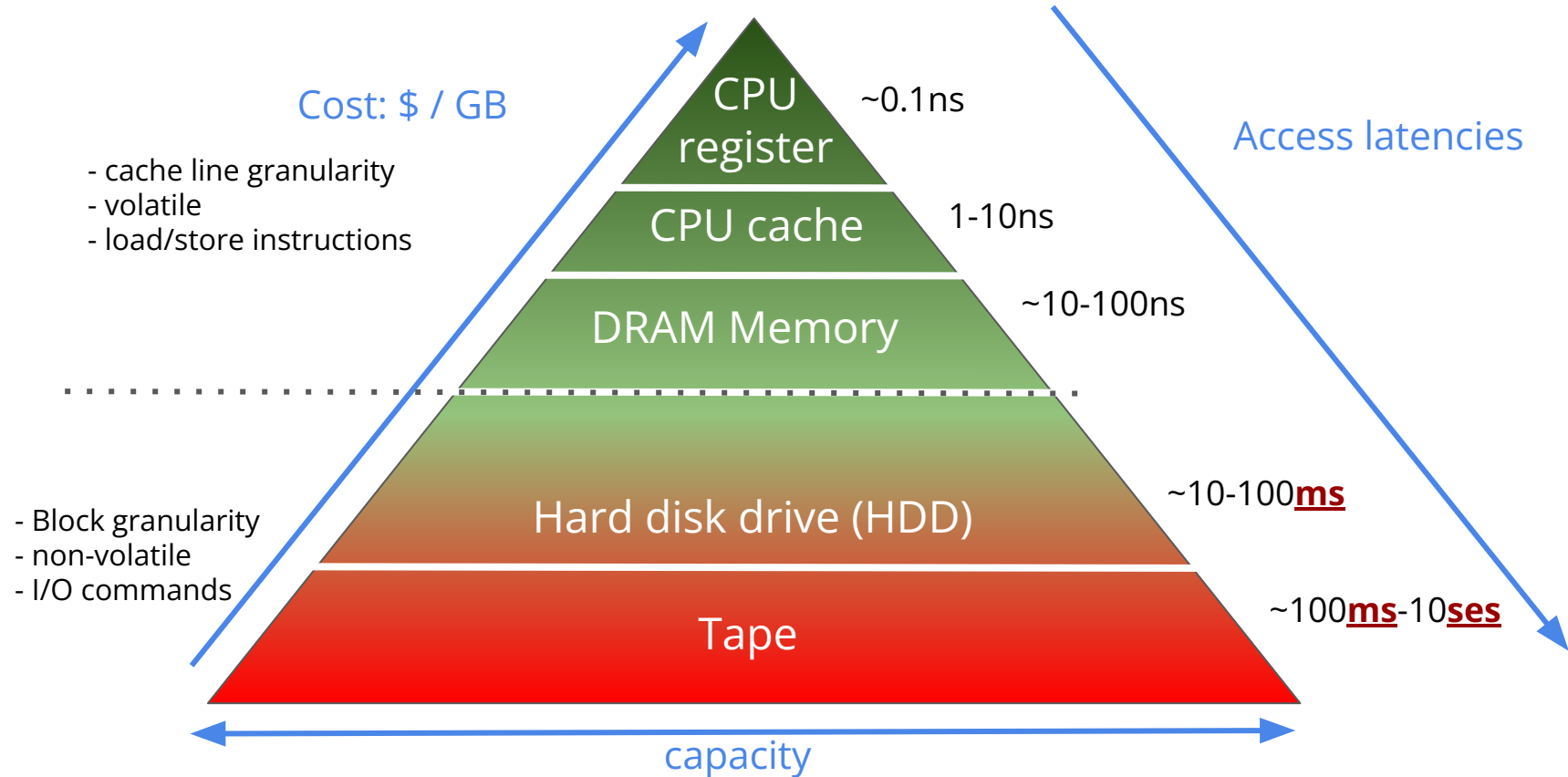
- 6 or 12 ECTS credits

Master projects / literature study

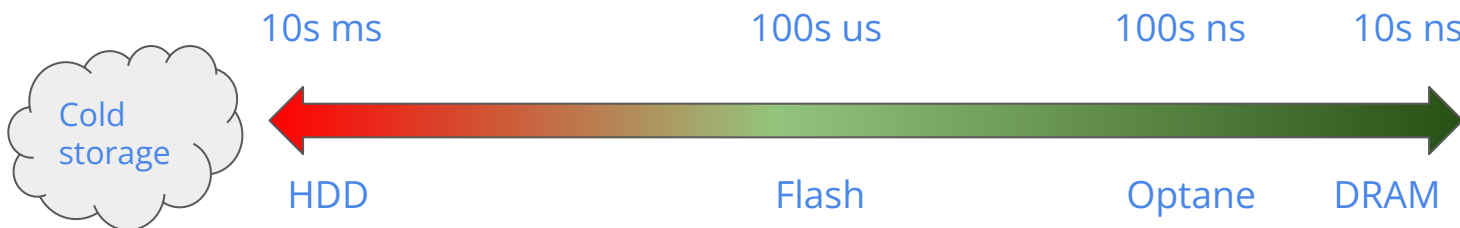
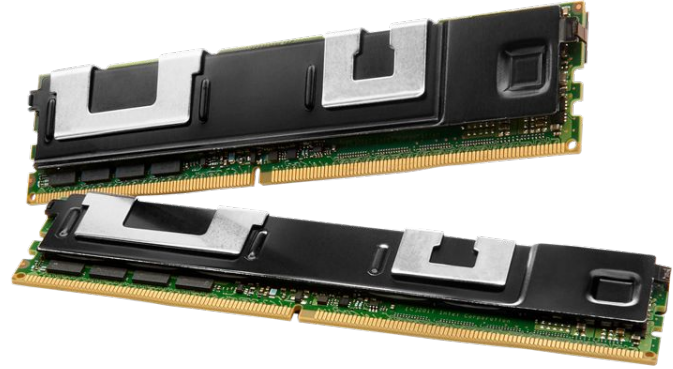
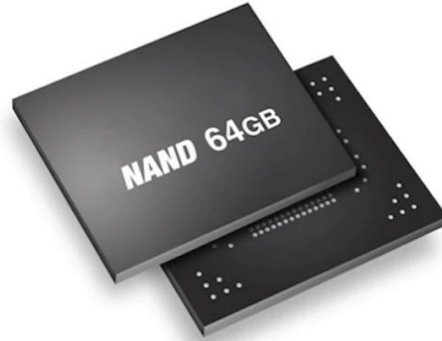
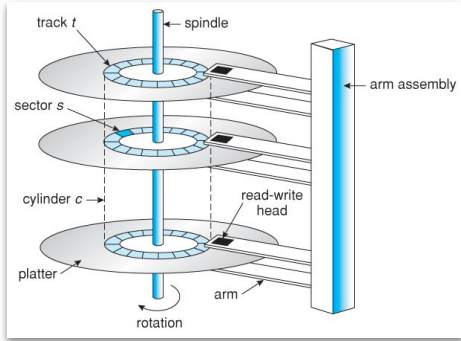
- Benchmarking the storage benchmarks
- io\_uring/CXL research (*today's lecture*)
- Integrating NVM(e)/NVMoF storage in ML runtime to train large models  
(Swapping Tensors)
- Building computation storage device prototype in QEMU
- Virtualizing ZNS/NVMe devices
- Scheduling I/O operations for workload-specific optimizations
- *Your favorite idea ... I am broadly open to ideas from your side, pick a paper and lets discuss*



# The triangle of storage hierarchy

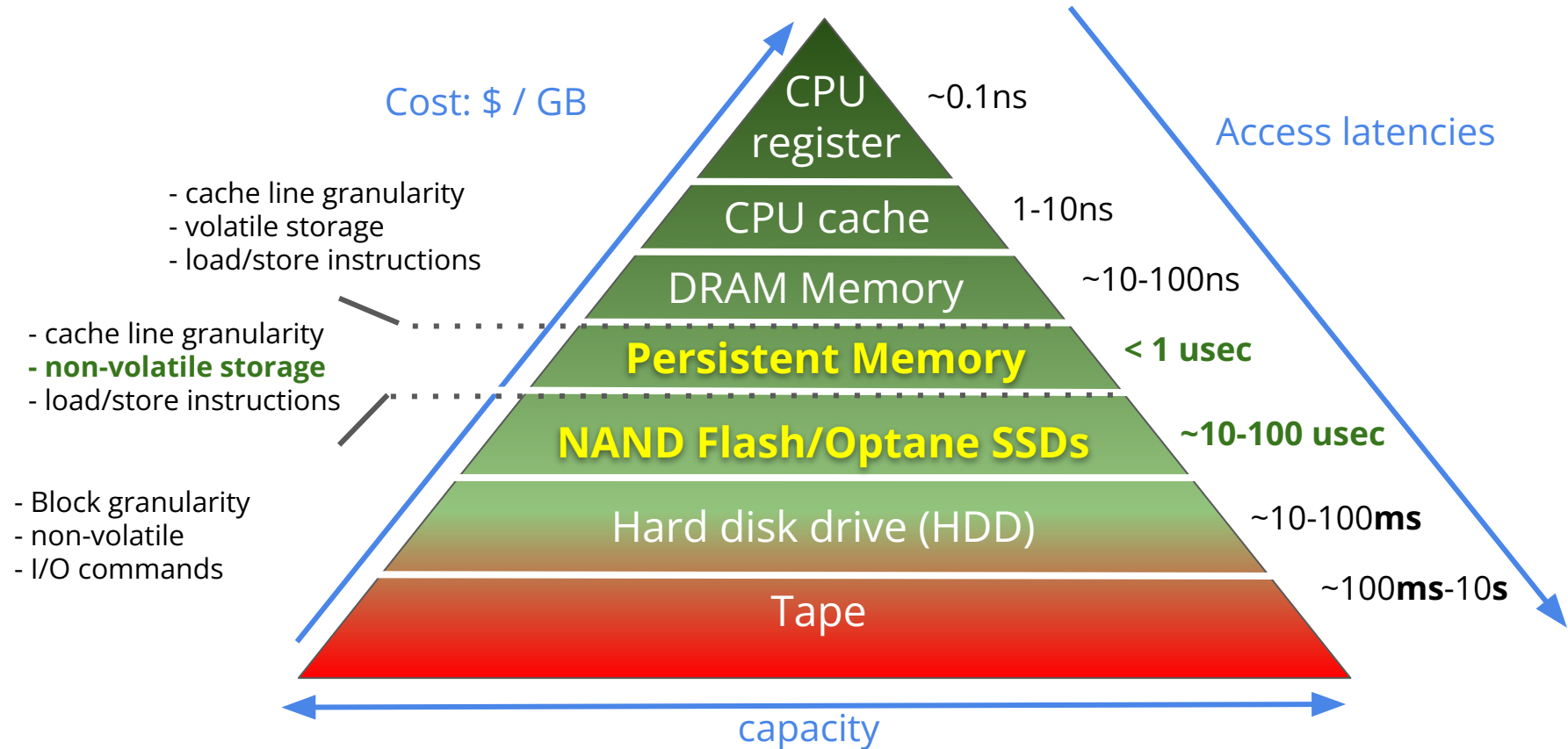


# Recap: From HDDs to Persistent Memories (PMem)



<http://pages.cs.wisc.edu/~remzi/OSTEP/file-disks.pdf>  
<https://www.partitionwizard.com/help/what-is-chs.html>

# The (new) triangle of storage hierarchy



# Multiple Emerging Topics (non-exhaustive)

Domain-specific/specialized storage solutions

Storage virtualization, Disaggregation (end-to-end software-defined-\*)

Quality-of-service in Storage Ecosystems (scheduling, multi-tenancy)

Energy Considerations

**CPU-free Computing** (re-thinking the computing architecture)

- [CPU-free Computing: A Vision with a Blueprint | Proceedings of the 19th Workshop on Hot Topics in Operating Systems](#)

**Hardware changes: Computer Express Link (CXL)**

- *Brief motivation and capabilities (without getting into too much hw/PCIe details)*

**New software APIs: io\_uring (Linux, also being ported to other OSes)**

- *How is it different than other APIs and what options does it provide, performance implications*



# The Key Problems 1 / 2

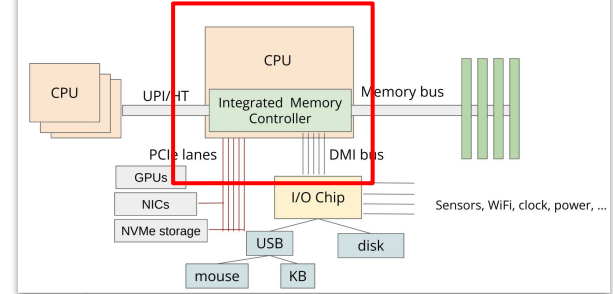
## The CPU is the center of computing

- direct memory access
- center of coherency
- controller of the devices

and the final coordinator and arbiter

## The CPU performance was fast!

A more modern (simplified) setup



Central processing unit (CPU)

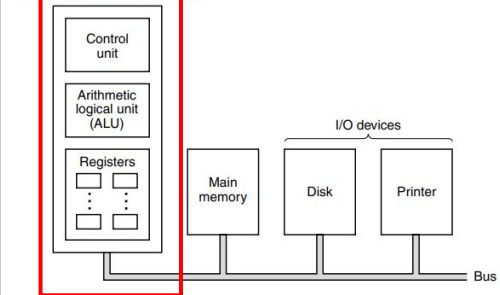
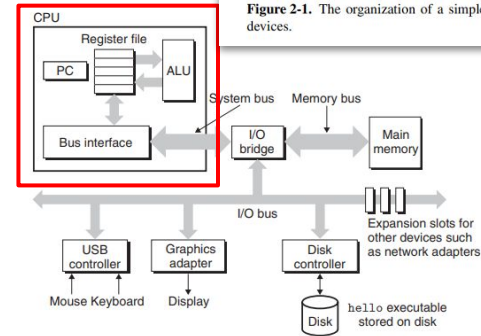


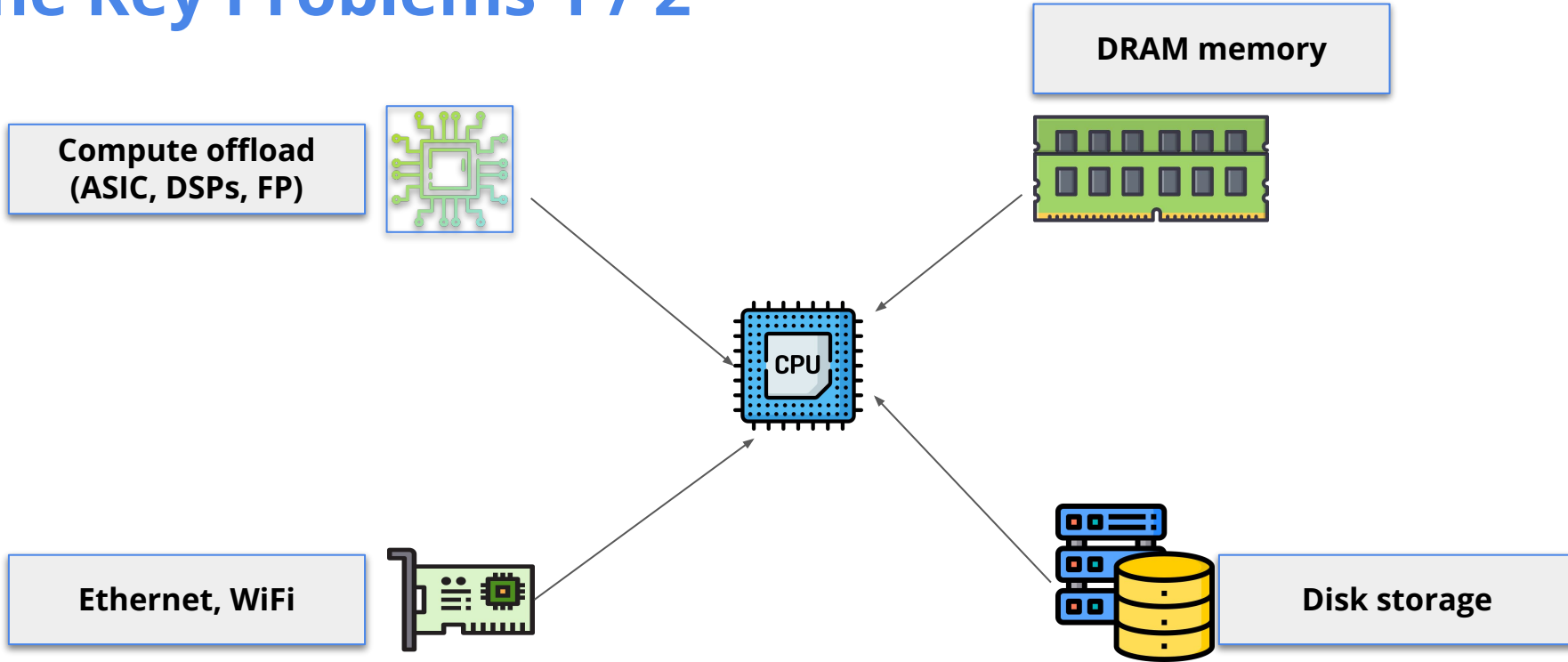
Figure 2-1. The organization of a simple computer with one CPU and two I/O devices.

Figure 1.4

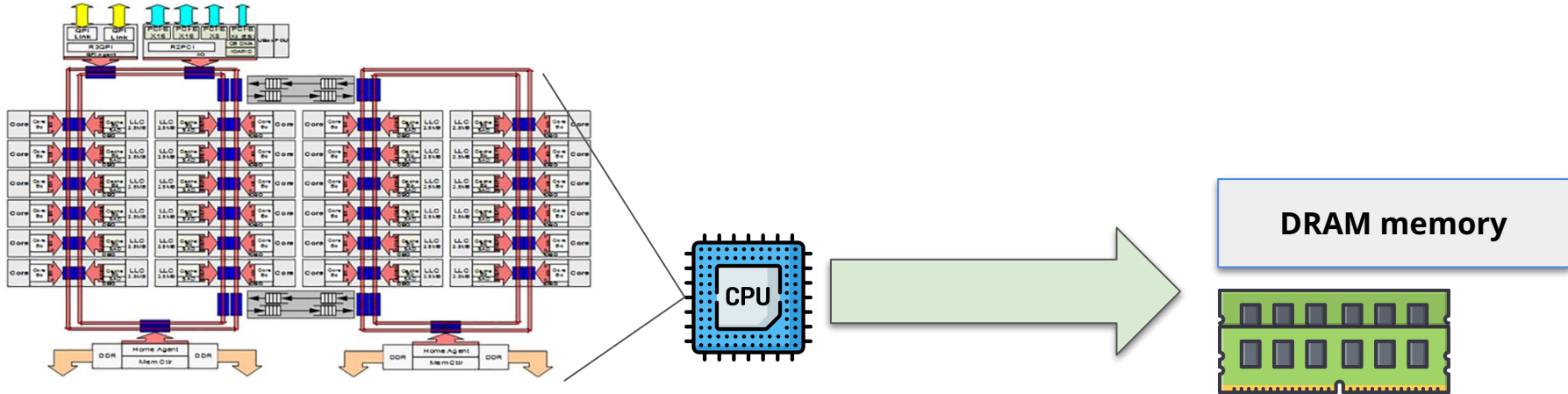
Hardware organization of a typical system. CPU: Central Processing Unit, ALU: Arithmetic/Logic Unit, PC: Program counter, USB: Universal Serial Bus.



# The Key Problems 1 / 2



# The Key Problems 1 / 2

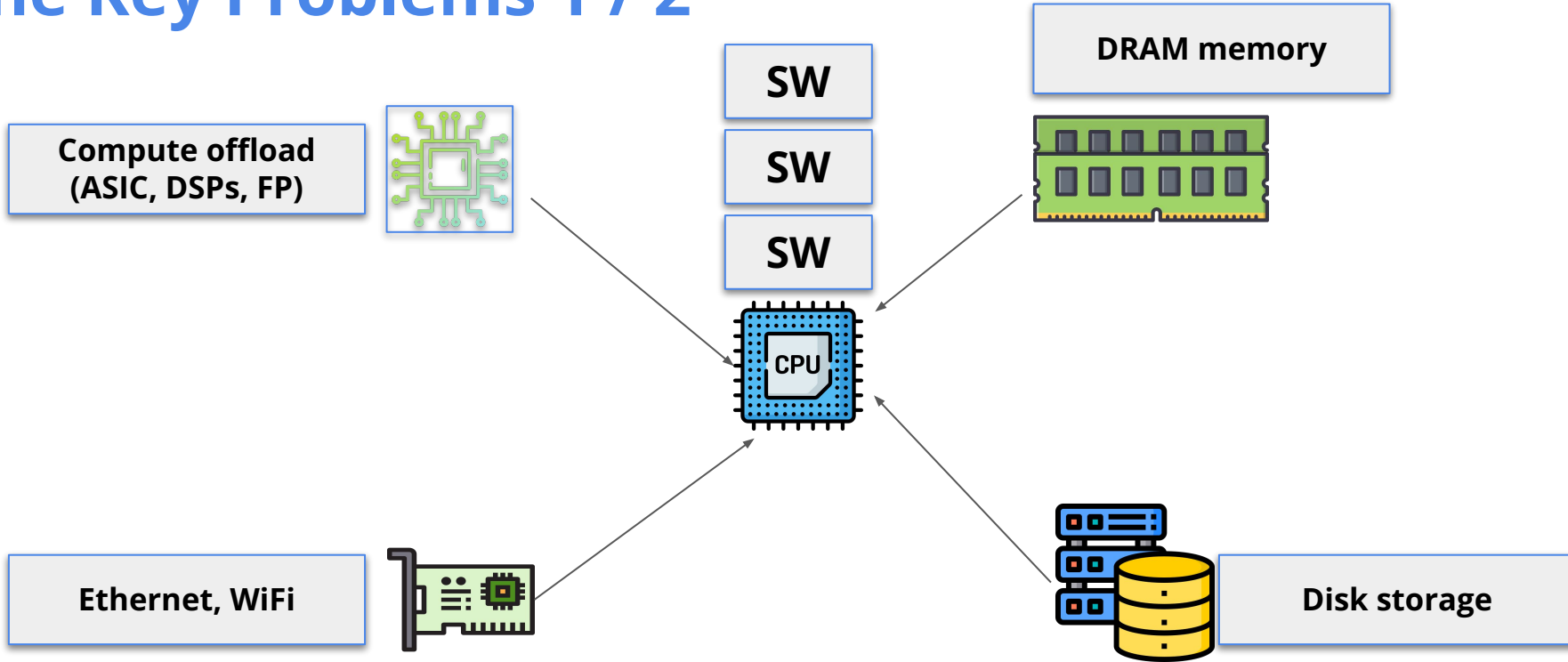


**CPU cache management is non-trivial and complex  
(even with same/similar homogeneous CPU architectures)**

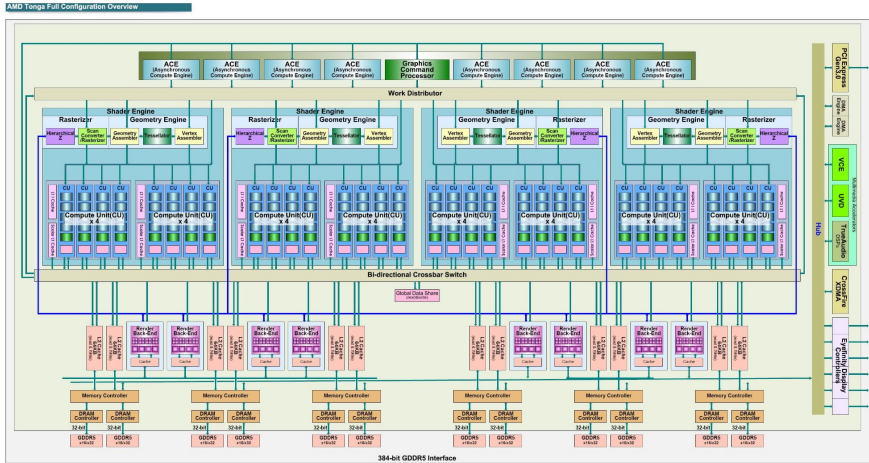
(2022) Intel® Xeon® Processor Scalable Family Technical Overview,

<https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html>

# The Key Problems 1 / 2

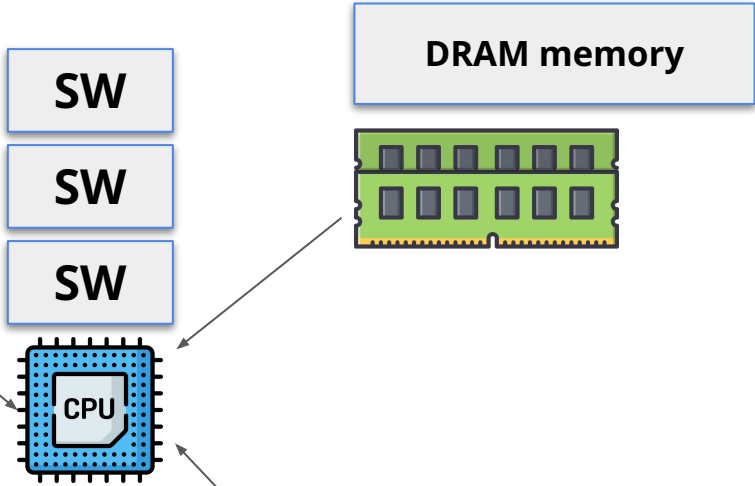
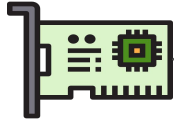


# The Key Problems 1 / 2



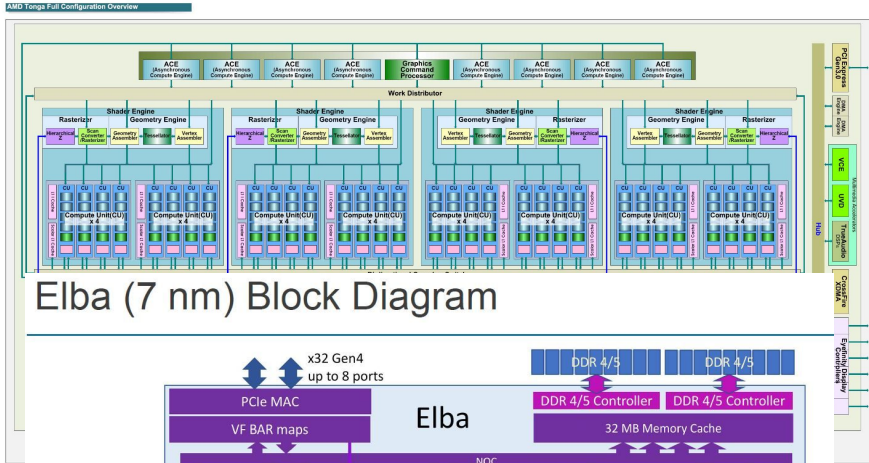
Copyright © 2014 Message Data. All rights reserved.

Ethernet, WiFi

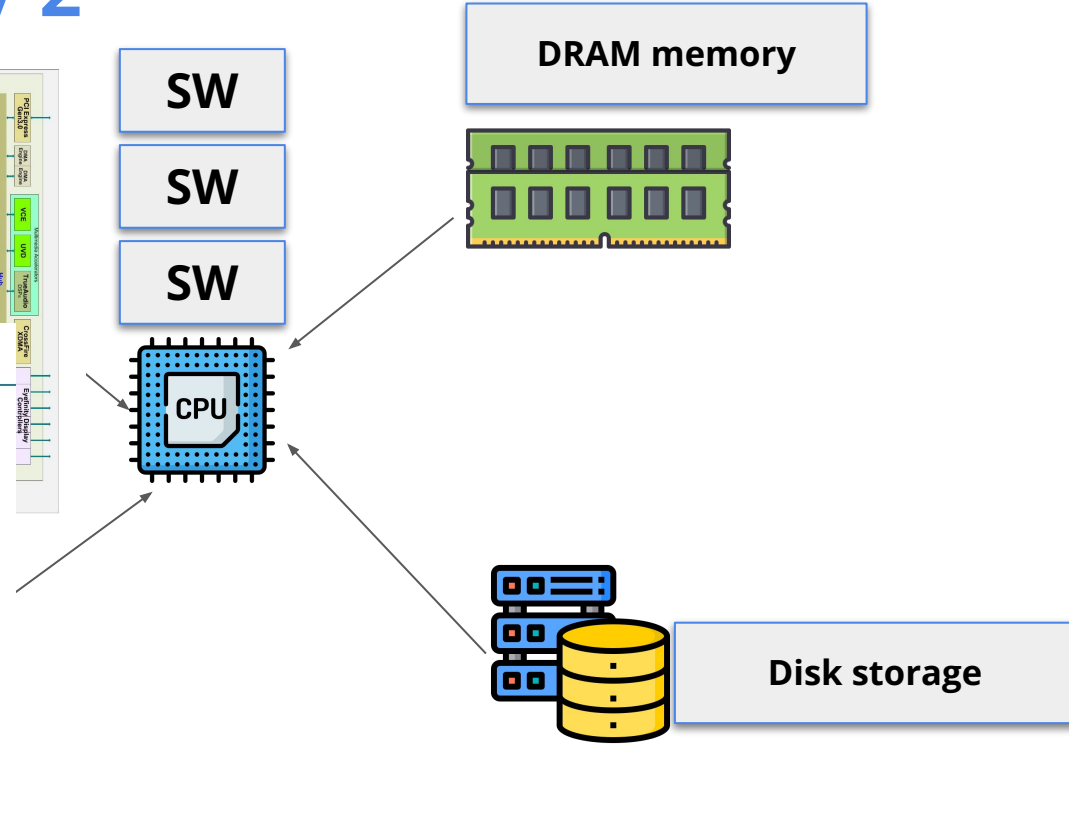


Disk storage

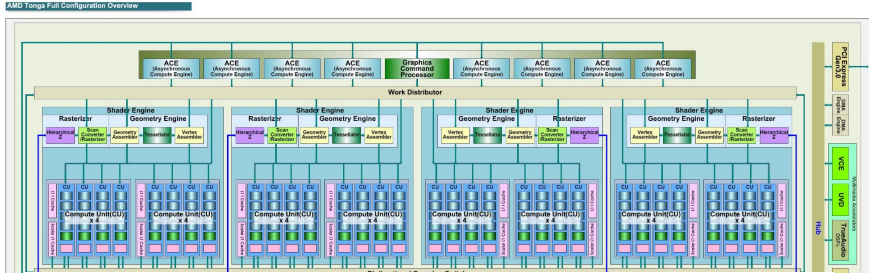
# The Key Problems 1 / 2



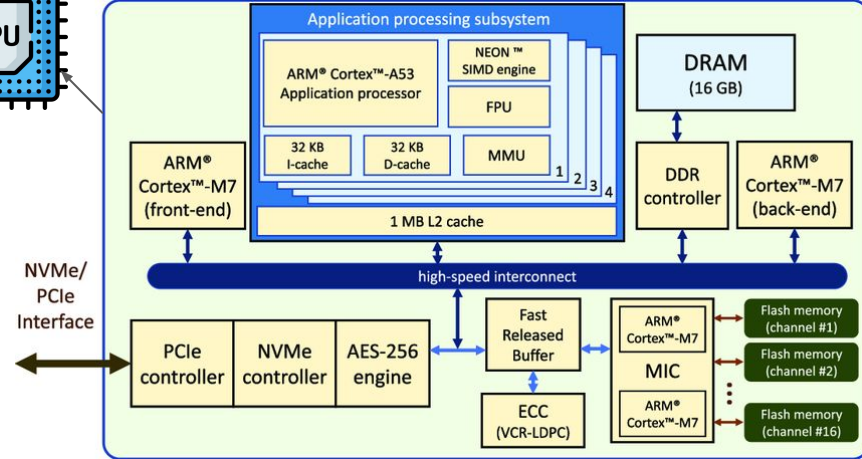
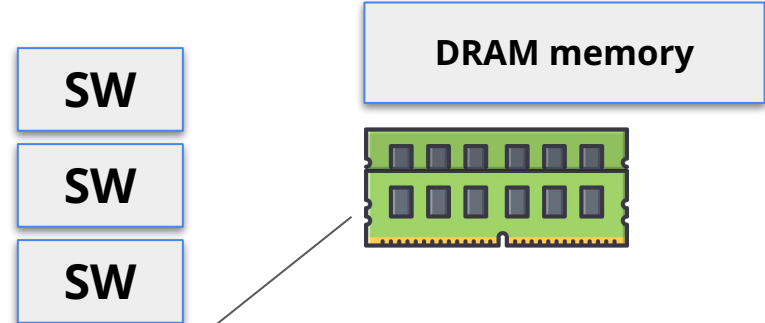
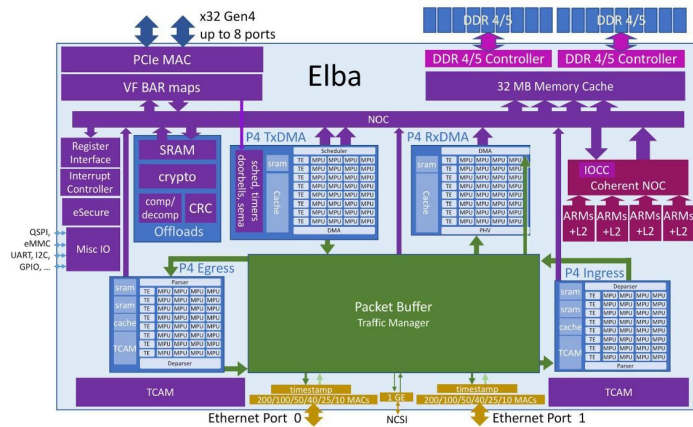
Elba (7 nm) Block Diagram



# The Key Problems 1 / 2

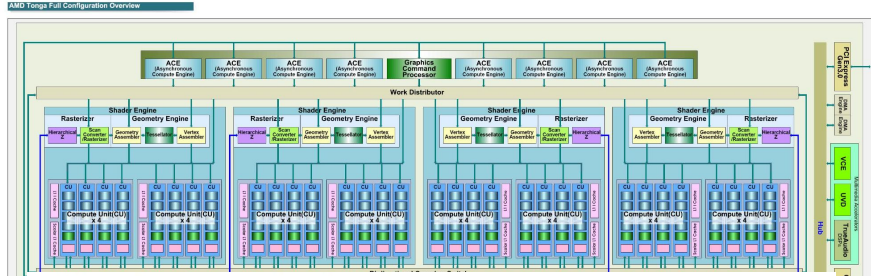


Elba (7 nm) Block Diagram

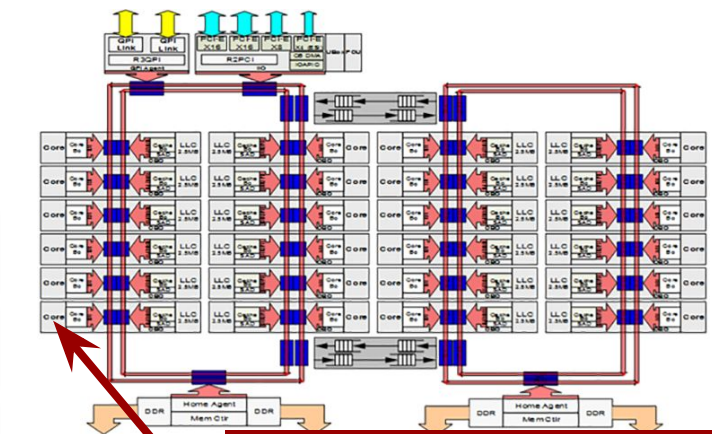
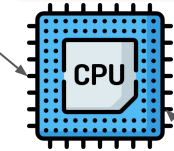
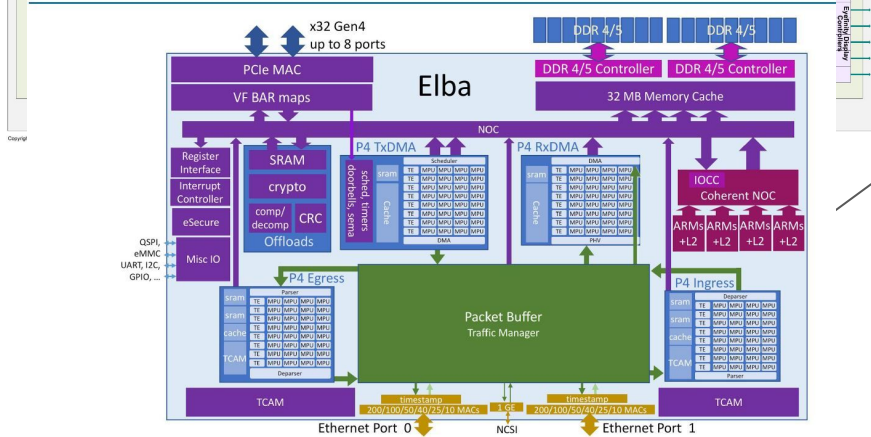




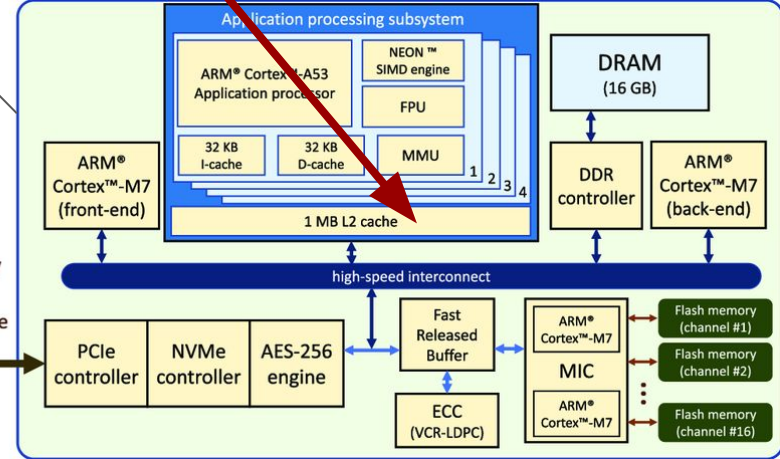
# The Key Problems 1 / 2



Elba (7 nm) Block Diagram



**How are these two caches synchronized?**





# The Key Problems 1 / 2

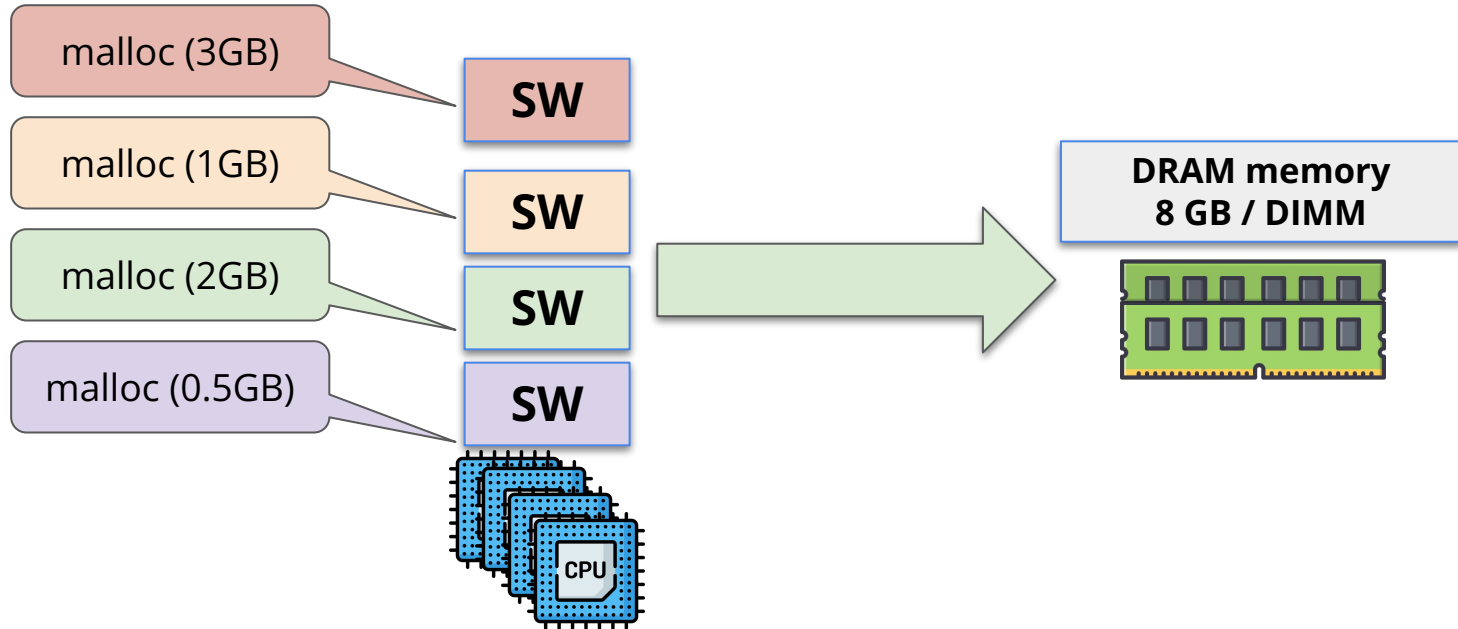
## These accelerators can have :

- Compute elements (specialized - FPGA, or general - ARM)
- Memory elements
- Storage chips
- Multi-level caches
- Outside connectivity

*Who manages “coherency”, “data flow”, “configuration”, “management” of memories/caches/devices here? Software, hardware? Performance?*

*Cost of development of new APIs, protocols?*

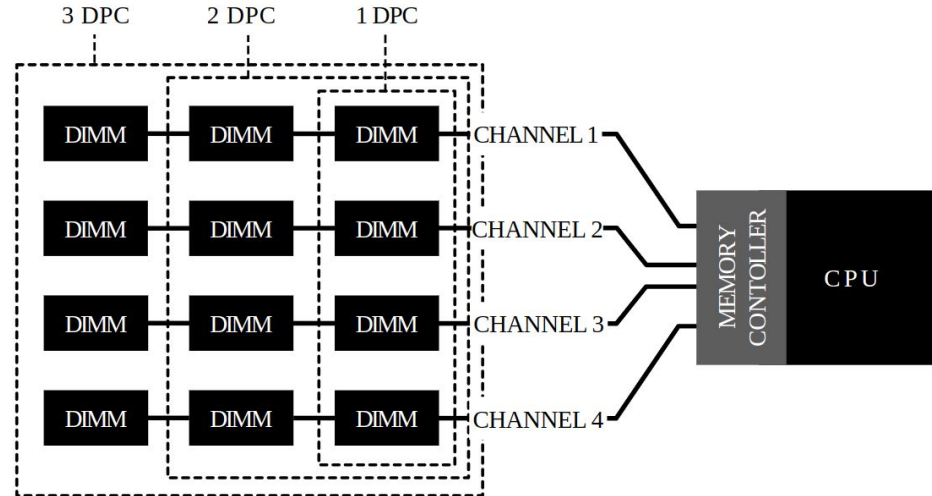
# The Key Problems 2 / 2 : CPU - DRAM Coupling



- **What happens to the remaining 1.5 GB DRAM?**
- **Do applications use all the DRAM what they ask for?**

# The Key Problems 2 / 2 : CPU - DRAM Coupling

1. Can not mix and match different DRAM technologies and generations
2. More performance means more capacity (need to buy more DIMMs)
3. Limit to how much DRAM can be packed in a single machine



**Very close coupling of CPU-DRAM (1) DRAM technology; (2) Density, capacity; and (3) Performance**

# The Key Problems 2 / 2 : CPU - DRAM Coupling

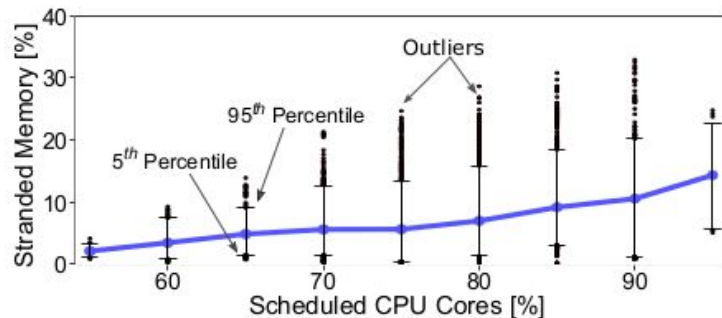
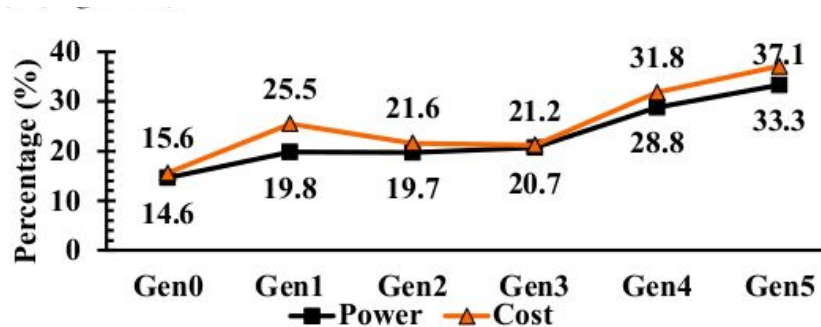


Figure 2: Memory stranding (§3.1). Stranding increases significantly as more CPU cores are scheduled. Error bars indicate the 5<sup>th</sup> and 95<sup>th</sup> percentiles (outliers in dots).

DRAM is a big power and cost factor in data center (up to ~40%)

A big part can remain underutilized

Azure with VMs : on average ~10% (but as high as ~30%)

# The Key Problems 2 / 2 : CPU - DRAM Coupling

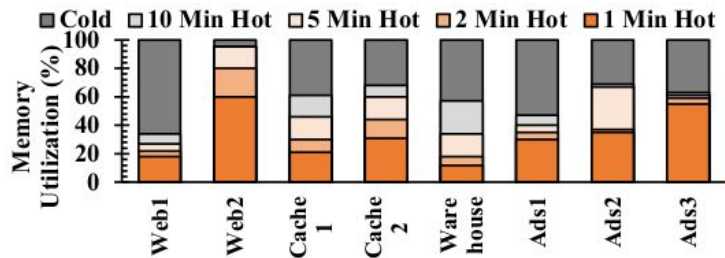


Figure 7: Application memory usage over last N mins.

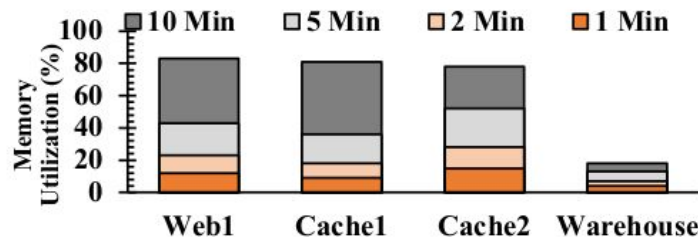


Figure 11: Fraction of pages re-accessed at different intervals.

Not all pages allocation are used **uniformly**:

- (1) Only a small fraction of memory is accessed in 1-2 minutes window
- (2) For Web, almost 80% of the pages are re-accessed within a ten-minute interval but for warehouse it is 20%.

*(do they all have to be in DRAM?)*

# Summary Problem

There has to be a better way to

- Manage non-CPU memories and caches (accelerators)
  - Manage CPU-attached memories (allocation, disaggregate from the CPU)
  - Expand beyond the CPU-attached memories
- + **Think of non-volatile memories ...**
- Persistent memories
  - Fast storage

Solution : **Compute Express Link (CXL)** *(the last protocol we will ever need)*

# Computer Express Link (CXL)

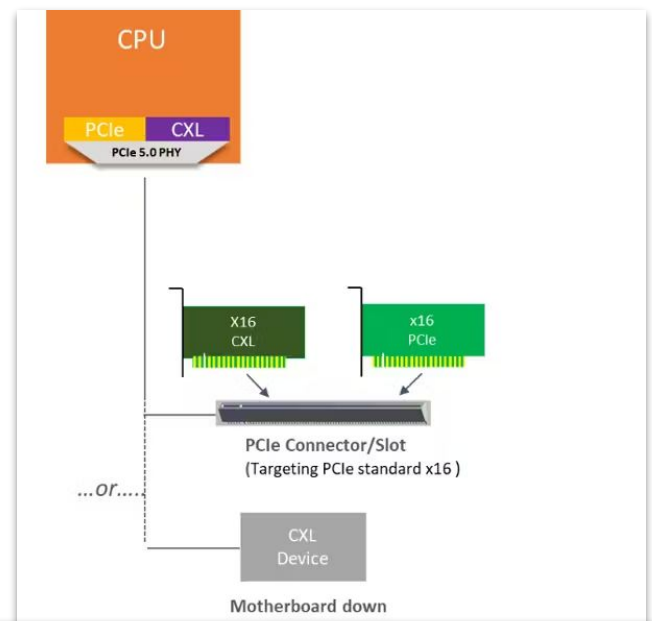
A **cache coherent Interconnect** between

- The CPU
- Accelerators
- Memory expansion cards

## Asymmetric protocol

A set of standardized protocols defined on the top of PCIe 5.0 (PHY)

- Runs in the standard PCIe slots
- 32 GT/s, or 4 GB/lane  $\Rightarrow$  x32 card = **128 GB/sec**
- *Latencies approaching the NUMA CPU (with v6.0)*



| PCIe Specification | Data Rate per Lane (GT/s) | Encoding  | x16 Unidirectional Bandwidth (GB/s) | Specification Ratification Year |
|--------------------|---------------------------|-----------|-------------------------------------|---------------------------------|
| 1.x                | 2.5                       | 8b/10b    | 4                                   | 2003                            |
| 2.x                | 5                         | 8b/10b    | 8                                   | 2007                            |
| 3.x                | 8                         | 128b/130b | 15.75                               | 2010                            |
| 4.0                | 16                        | 128b/130b | 31.5                                | 2017                            |
| 5.0                | 32                        | 128b/130b | 63                                  | 2019                            |
| 6.0                | 64                        | PAM4/FLIT | 128                                 | 2022                            |

# Three CXL Protocols

## CXL.io

- Mandatory for all hosts, and CXL supported devices
- Discovery, enumerations, capabilities (DMA, interrupts, IOV), and host physical address configuration
- Same in spirit to what any basic PCIe device would support

## CXL.mem

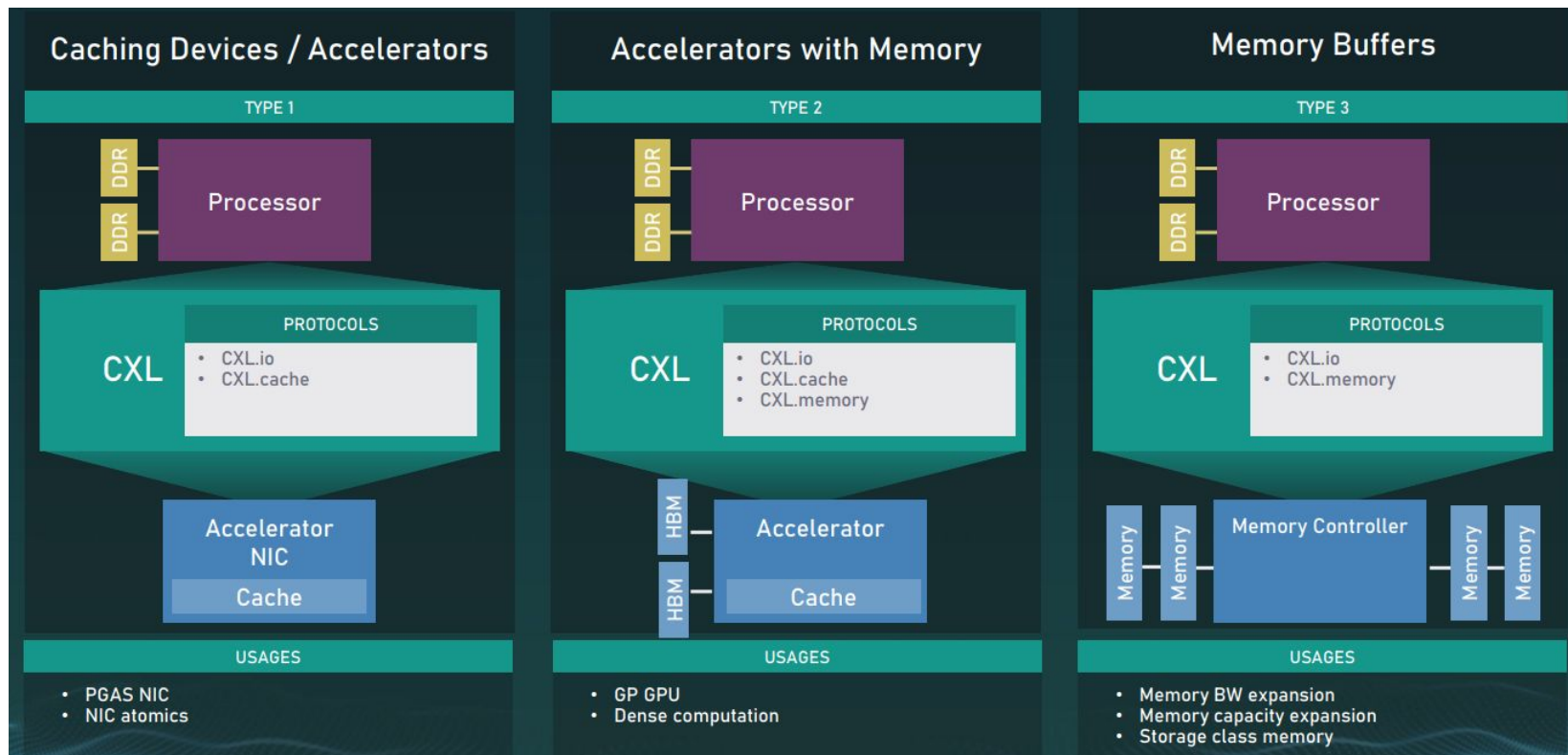
- Enables (only) CPU to access device/accelerator memory in a cacheable manner
- Useful in DRAM expansion
- Device is not initiating any communication

## CXL.cache

- The same as CXL.mem, but now devices can also access the CPU memory/caches
- Additional commands/requests for maintaining coherence among all copies



# Three Classes of Devices

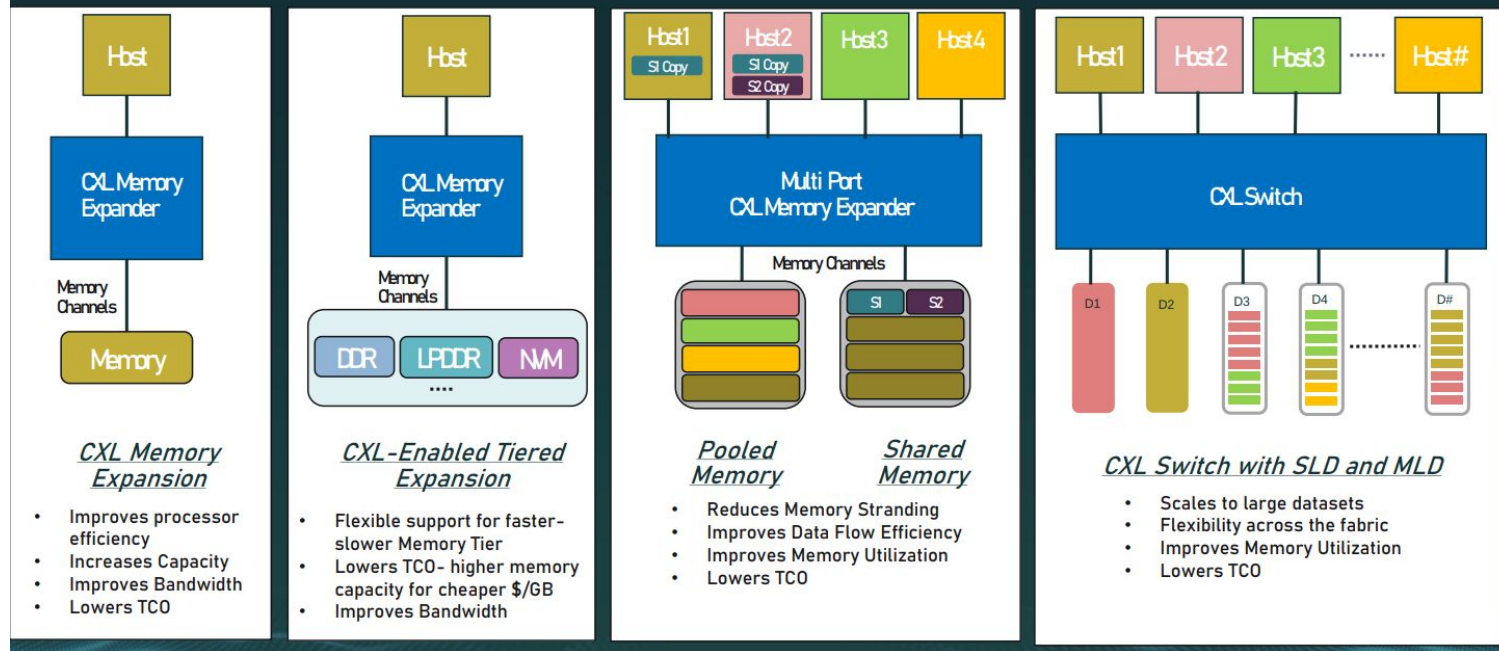


# Three Generations of CXL Protocols

| Features                                     | CXL 1.0 / 1.1 | CXL 2.0 | CXL 3.0 |
|--|---------------|---------|---------|
| Release date                                 | 2019          | 2020    | 1H 2022 |
| Max link rate                                | 32GTs         | 32GTs   | 64GTs   |
| Flit 68 byte (up to 32 GTs)                  | ✓             | ✓       | ✓       |
| Flit 256 byte (up to 64 GTs)                 |               |         | ✓       |
| Type 1, Type 2 and Type 3 Devices            | ✓             | ✓       | ✓       |
| Memory Pooling w/ MLDs                       |               | ✓       | ✓       |
| Global Persistent Flush                      |               | ✓       | ✓       |
| CXL IDE                                      |               | ✓       | ✓       |
| Switching (Single-level)                     |               | ✓       | ✓       |
| Switching (Multi-level)                      |               |         | ✓       |
| Direct memory access for peer-to-peer        |               |         | ✓       |
| Enhanced coherency (256 byte flit)           |               |         | ✓       |
| Memory sharing (256 byte flit)               |               |         | ✓       |
| Multiple Type 1/Type 2 devices per root port |               |         | ✓       |
| Fabric capabilities (256 byte flit)          |               |         | ✓       |

- CXL 3.0: Enabling composable systems with expanded fabric capabilities, October 6, 2022, [https://www.computeexpresslink.org/files/ugd/0c1418\\_998df4f459734f319e7a12cc2163b943.pdf](https://www.computeexpresslink.org/files/ugd/0c1418_998df4f459734f319e7a12cc2163b943.pdf)
- Good overview, [https://community.cadence.com/cadence\\_blogs\\_8/b/breakfast-bytes/posts/hot-chips-cxl-tutorial](https://community.cadence.com/cadence_blogs_8/b/breakfast-bytes/posts/hot-chips-cxl-tutorial)

# Evolving Use Cases

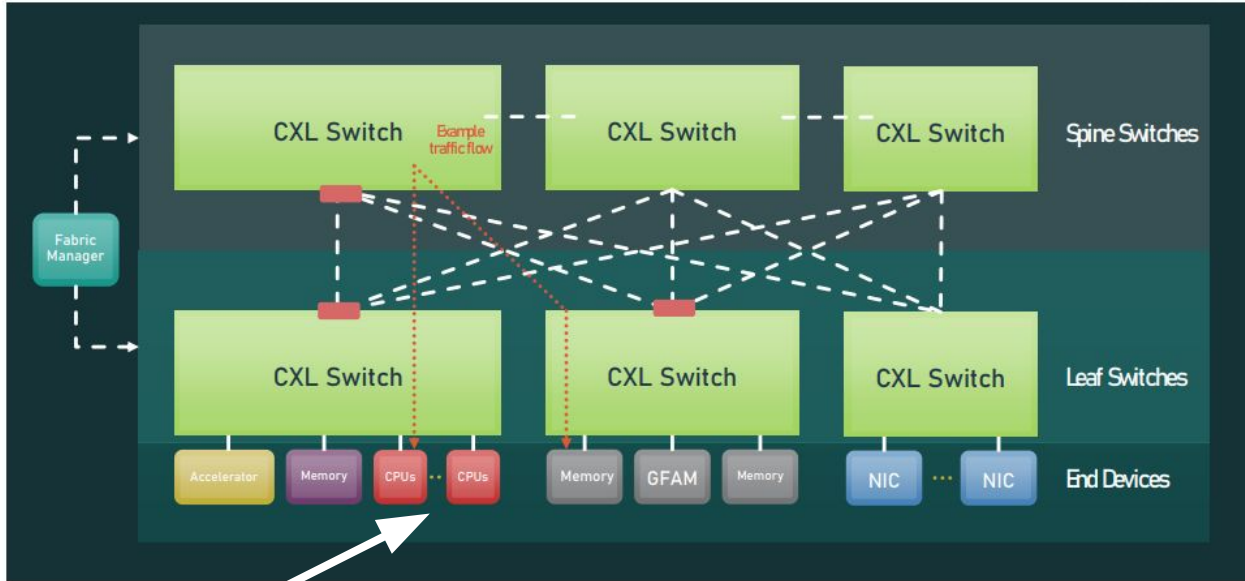


**What can we do?** Expansion of DRAM, CPU-Memory Decoupling (multiple generation of devices), Memory Pooling and sharing, Single Logical Device (SLD → Exclusive to one CXL root) to Multiple Logical Device (MLD, connected to multiple CXL roots), Memory hot swapping ...

A look into the CXL device ecosystem and the evolution of CXL use cases,

[https://0c141887-fbe4-4ec3-be17-adc8d70d3922.usrfiles.com/ugd/0c1418\\_037d4ba31f4b44cf9fcb37f5b36ae4d6.pdf](https://0c141887-fbe4-4ec3-be17-adc8d70d3922.usrfiles.com/ugd/0c1418_037d4ba31f4b44cf9fcb37f5b36ae4d6.pdf)

# Design a Distributed Cluster Running CXL



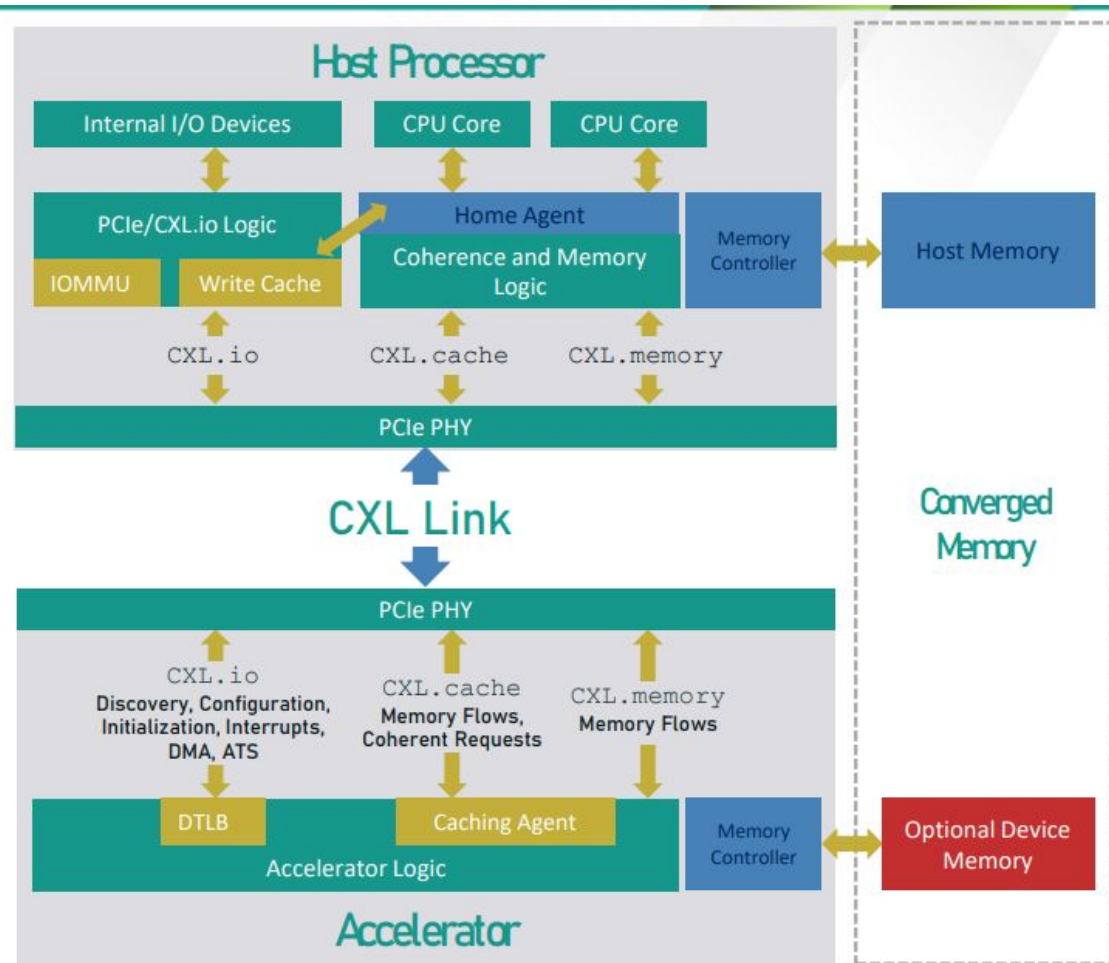
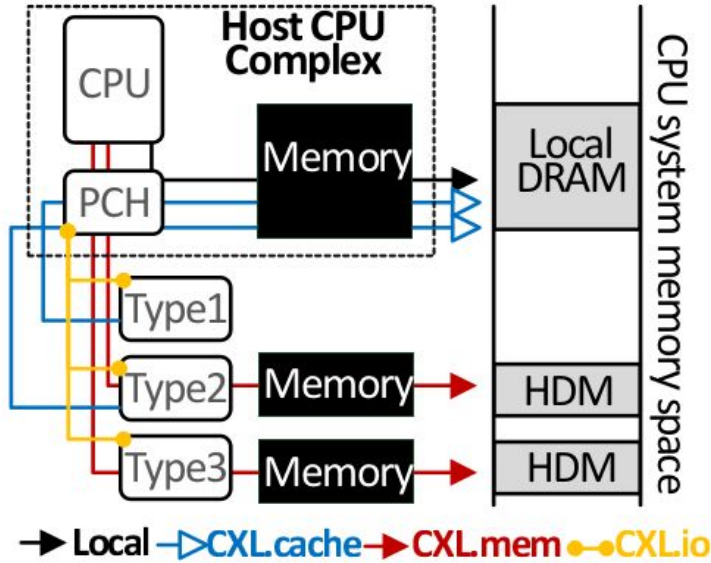
## CXL 3.0 Fabric Architecture

- Interconnected Spine Switch System
- Leaf Switch NIC Enclosure
- Leaf Switch CPU Enclosure
- Leaf Switch Accelerator Enclosure
- Leaf Switch Memory Enclosure

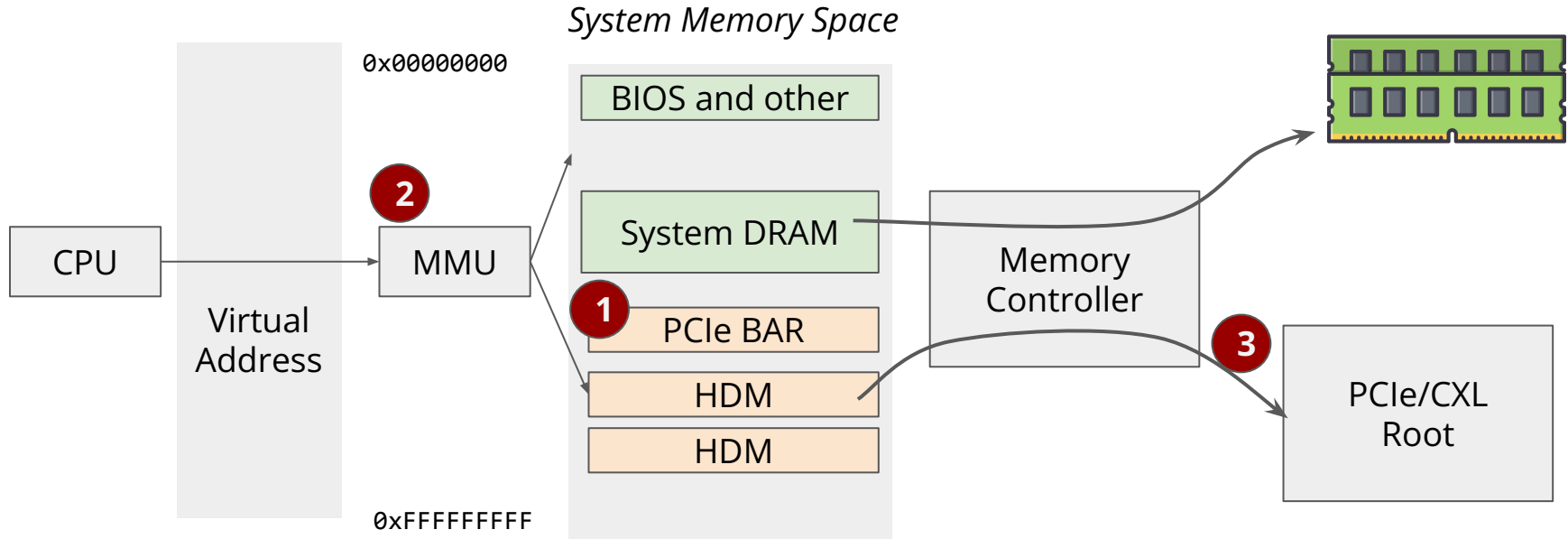


*Multiple type of devices, Global Fabric Attached Memory (GFAM)*

# CXL.mem



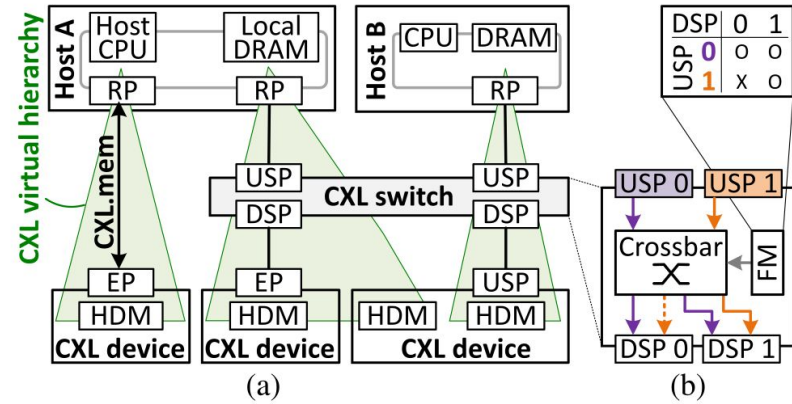
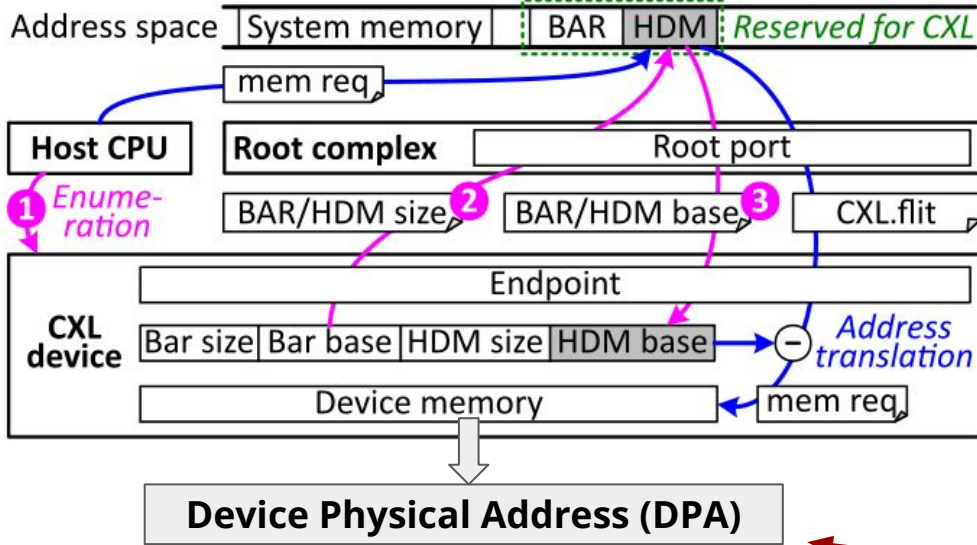
# CXL.mem Expansion Device Example



1. PCIe enumeration and BAR mapping with, Host-Managed Device Memory (HDM) areas
2. Setup MMU and allocate the DRAM physical address from this area (software support)
3. Access happens, and the request is routed to the PCIe/CXL root



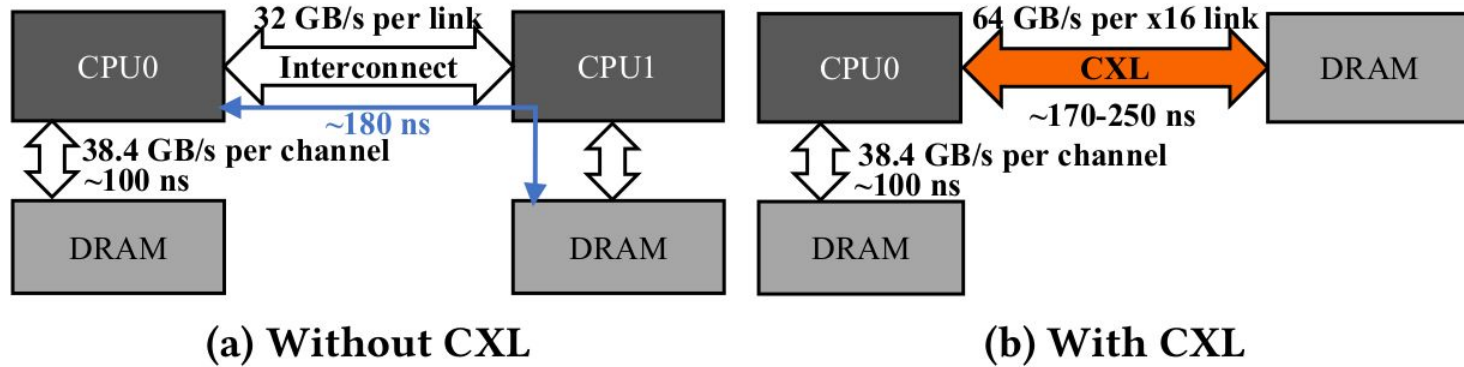
# CXL.mem Expansion Device Example



**DRAM Translation Layer DTL ;)**  
 See the ISCA'23 reference at the end of the slides

**Multiple configurations** (1) striping across multiple devices, ports, roots; (2) allocation units...

# Transparent Page Placement (TPP)

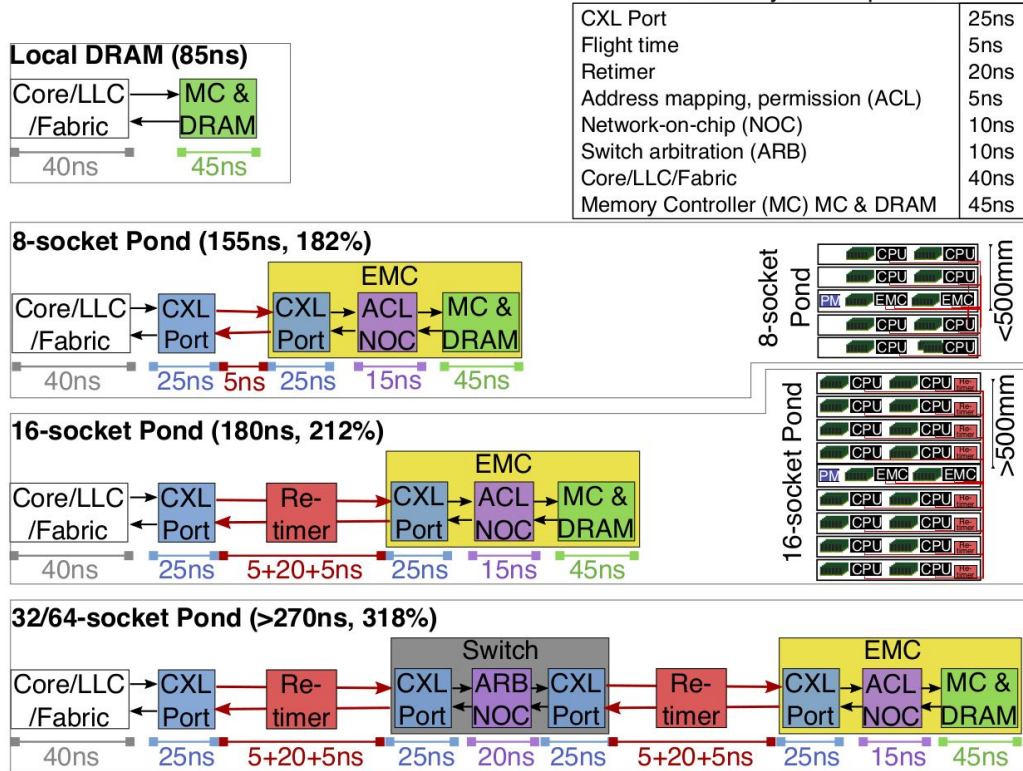


PCIe 6.0 latencies and bandwidth are approaching access to a remote NUMA CPU socket

**Challenge:** How to profile pages (at low-overheads) and put them in the right storage level in the CXL-enabled memory hierarchy



# POND (ASPLOS'23): How to Disaggregate VM Memory



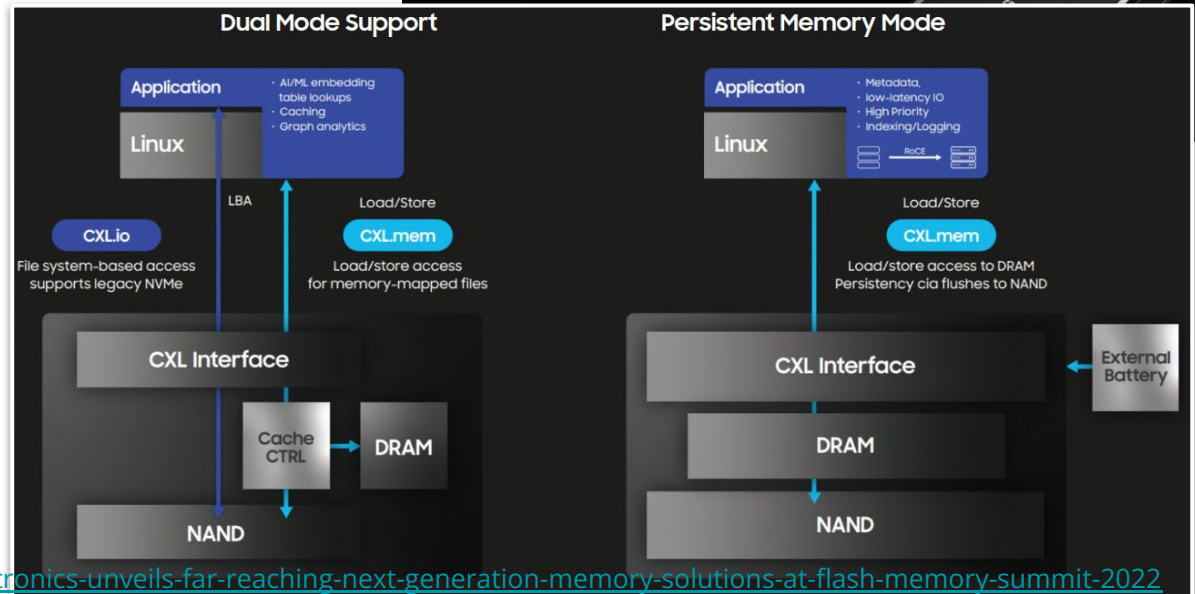
# Where does Storage Come into the Play?

*Any device can implement the CXL protocol*

- Use SSD as large capacity RAM
- **Byte\*-addressable**
- Persistent

\*64B addressable

Industry 1st CXL-based Storage  
Optimized for AI/ML



# Emerging work: Quantifying and Hiding Flash Latencies

## Hello Bytes, Bye Blocks: PCIe Storage Meets Compute Express Link for Memory Expansion (CXL-SSD)

Myoungsoo Jung  
Computer Architecture and Memory Systems Laboratory,  
Korea Advanced Institute of Science and Technology (KAIST)  
http://camlab.org

### ABSTRACT

Compute express link (CXL) is the first open multi-protocol method to support cache coherent interconnect for different processors, accelerators, and memory device types. Even though CXL manages data coherency mainly between CPU memory spaces and memory on attached devices, we argue that it can also be useful to reform existing block storage as cost-efficient, large-scale working memory. Specifically, this paper examines three different sub-protocols of CXL from a memory expander viewpoint. It then suggests which device type can be the best option for PCIe storage to bridge its local semantics to memory-compatible, byte semantics. We then discuss how to integrate a storage-integrated memory expander into an existing system and speculate how much effect it does have on the system performance. Lastly, we visit various CXL network topologies and explore a new opportunity to efficiently manage the storage-integrated, CXL-based memory expansion.

### 1 INTRODUCTION

Cache coherence interconnects are recently emerged to integrate different CPUs, accelerators, and memory components into a heterogeneous, single computing domain. Specifically, the interconnect technologies maintain data coherency between CPU memory and private memory attached to devices, defining a new type of globally shared memory and network space. While there have been several efforts to coherently connect different hardware components, such as Gen-Z [1] and CCIX [2], *Compute Express Link (CXL)* is the first open interconnect protocol supporting various types of processors and device endpoints [3]. CXL has absorbed Gen-Z [4] and has become one of the most promising interconnect interfaces thanks to its high-speed coherence control and full compatibility with the existing bus standard, PCIe. A broad spectrum

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting, storing, and retrieval is permitted. Copying beyond these limits, especially for redistribution, requires prior specific permission and/or fee. Request permissions from permissions@acm.org.

ACM ISBN 978-1-4503-9369-7/22/06... \$15.00

https://doi.org/10.1145/359643.359645

of datacenter-scale hardware such as CPU, GPU, FPGA, and domain-specific ASIC is thus expected to take significant advantage of CXL [5–7]. CXL consortium announced that it can also disaggregate memory by pooling DRAM and byte-addressable persistent memory (PMEM). While CXL can handle diverse computing resources and memory components, it sets block storage aside and leaves a question on whether the storage can reap the benefits of CXL or not. A primary question that storage designers and system architects may have is i) *why and what can the block storage benefit from CXL?* If there is an advantage, we should be able to answer the following questions: ii) *how can we connect the underlying block storage to the host's system memory bus?*, iii) *what kind of CXL device type should be used for the block storage and memory expander?*, and iv) *what does CXL need to improve for better utilization of the block storage?*

In this paper, we argue that CXL is helpful in leveraging PCIe-based block storage to incarnate a large, scalable working memory by answering all the four questions mentioned above. We believe CXL is a cost-effective and practical interconnect technology that can bridge PCIe storage's block semantics to memory-compatible, byte semantics. To this end, we should carefully integrate the block storage into its interconnect network by being aware of the diversity of device types and protocols that CXL supports. This paper first discusses what a mechanism makes the PCIe storage impractical and unable to be used for a memory expander (§2). Then, we explore all the CXL device types and their protocol interfaces to answer which configuration would be the best for the PCIe storage to expand the host's CPU memory (§3).

Even though CXL is not the most promising interface for the block storage in getting closer to CPU, it is non-trivial to speculate how much effect a storage-integrated memory expander does have on system performance. As there is no CPU and fabric for CXL yet, it is also unclear for the storage designers and system architects to see how CXL-enabled storage can be implemented and interact with CPU. To answer this, we discuss what a PCIe storage device needs to change, how it can be connected to the host over CXL, and how users can access the device through load/store instructions (§4). We then project the performance of the storage-integrated memory expander by prototyping CXL agents and controllers in different FPGA nodes, all connected by a PCIe network.

## Cache in Hand: Expander-Driven CXL Prefetcher for Next Generation CXL-SSDs

Miryong Kwon<sup>†</sup>, Sangwon Lee<sup>\*,†</sup>, Myoungsoo Jung<sup>\*,†</sup>  
<sup>\*</sup>Computer Architecture and Memory Systems Laboratory, KAIST  
<sup>†</sup>Panmesia, inc.

### ABSTRACT

Integrating compute express link (CXL) with SSDs allows scalable access to large memory but has slower speeds than DRAMs. We present ExpAND, an expander-driven CXL prefetcher that offloads last-level cache (LLC) prefetching from host CPU to CXL-SSDs. ExpAND uses a heterogeneous prediction algorithm for prefetching and ensures data consistency with CXL mem's back-invalidation. We examine prefetcher timeliness for accurate latency estimation. ExpAND, being aware of CXL multi-tiered switching, provides end-to-end latency for each CXL-SSD and precise prefetch time-limits estimations. Our method reduces CXL-SSD reliance and enables direct host cache access for most data. ExpAND enhances graph application performance by 3.5%, surpassing CXL-SSD pools with diverse prefetching strategies.

### 1 INTRODUCTION

Compute Express Link (CXL) is receiving considerable attention as an emerging interface that separates memory resources from computing servers, allowing users to access large-capacity memory scalably. In terms of capacity, storage class memory (SCM) technologies such as PRAM [1], Z-NAND [2], and XL-Flash [3] offer greater advantages over DRAMs. As a result, both industry and academia strive to introduce byte-addressable solid-state drives (SSDs) using the CXL protocol and SCM's memory instruction semantics. For instance, one method integrates CXL into Optane SSDs for hierarchical memory expansion, while several proof-of-concepts (PoCs) employ new flash like Z-NAND and XL-Flash to develop CXL-SSDs [4–6].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM ISBN 978-1-4503-9369-7/22/06... \$15.00

https://doi.org/10.1145/359643.359645

While CXL-SSDs target capacity needs for memory disaggregation, their backend media remains slower than DRAMs. Specifically, PRAMs are 7x slower than DRAMs [7], and the new flash technologies exhibit latencies 30x slower [2]. To address this, industrial PoCs employ SSD-side DRAM buffers as internal caches, resembling high-performance NVMe storage with larger internal DRAMs. Although these buffers effectively handle write latency issues, they struggle to mask the long read latency caused by SCM backend media. Unlike file system-managed block devices, CXL-SSDs should serve memory requests (load/store) without relying on the host-side storage stack. Concealing long read latency necessitates understanding execution behaviors of host applications and managing the corresponding CPU cache hierarchy, especially. Regrettably, these aspects are neglected by existing SSD technologies, as they have solely handled block requests thus far.

When CXL-SSDs are placed in the system memory space as host-managed device memory, existing CPU-side cache prefetching mechanisms can still be beneficial. However, two main unaddressed challenges prevent current prefetchers within the cache hierarchy from fully utilizing the advantages of LLC with CXL-SSDs: i) *hardware logic size constraints* in handling a wide range of memory access patterns possibly encountered in the extensive CXL memory pooling space, and ii) *latency variations* experienced by different CXL-SSDs located in diverse positions within the CXL switch network.

In particular, rule-based cache prefetchers, such as spatial [8–10] and temporal/prefetching algorithms [11–13], require tens of MB storage that is similar to the actual *last-level cache (LLC)* of a CPU [9]. As a result, modern CPUs employ a simpler stream cache prefetching algorithm [14], which unfortunately is unable to make the increased latency introduced by CXL-SSDs. Another contributing factor is the interconnect network topology used in CXL-based memory disaggregation. To boost memory capacity in a scalable way, CXL introduces a multi-level switch architecture where each level can potentially increase memory expander latency, depending on the target's position within the network. This is because the processing time taken by CXL switches at different levels cannot be overlooked. Consequently, existing prefetchers are

## Overcoming the Memory Wall with CXL-Enabled SSDs

Shao-Peng Yang  
Syracuse University  
Eeye Hyun Nam  
FADU Inc.  
Minjae Kim  
DGIST  
Eunji Lee  
Soongsil University  
Sanghyun Nam  
Soongsil University  
Sungjin Lee  
DGIST  
Juhyoung Park  
DGIST  
Bryan S. Kim  
Syracuse University  
Jin-yong Choi  
FADU Inc.

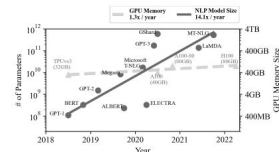
### Abstract

This paper investigates the feasibility of using inexpensive flash memory on new interconnect technologies such as CXL (Compute Express Link) to overcome the memory wall. We explore the design space of a CXL-enabled flash device and show that techniques such as caching and prefetching can help mitigate the concerns regarding flash memory's performance and lifetime. We demonstrate using real-world application traces that these techniques enable the CXL device to have an estimated lifetime of at least 3.1 years and serve 68–91% of the memory requests under a microsecond. We analyze the limitations of existing techniques and suggest system-level changes to achieve a DRAM-level performance using flash.

### 1 Introduction

The growing imbalance between computing power and memory capacity requirement in computing systems has developed into a challenge known as the memory wall [23, 34, 52]. Figure 1, based on the data from Ghahani et al. [34] and expanded with more recent data [11, 30, 43], illustrates the rapid growth in NLP (natural language processing) models (14.1x per year), which far outpaces that of memory capacity (1.3x per year). The memory wall forces modern data-intensive applications such as databases [8, 10, 14, 20], data analytics [1, 35], and machine learning (ML) [45, 48, 66] to either be aware of their memory usage [61] or implement user-level memory management [66] to avoid expensive page swaps [37, 53]. As a result, overcoming the memory wall in an application-transparent manner is an active research avenue; approaches such as creating an ML-centric system [45, 48, 61], building a memory disaggregation framework [36, 37, 52, 69], and designing new memory architecture [23, 42] are actively pursued.

We question whether it is possible to overcome the memory wall using flash memory — a memory technology that is typically used in storage due to its high density and capacity scaling [59]. While DRAM can only scale to gigabytes in capacity, a flash memory-based state-drive (SSD) is

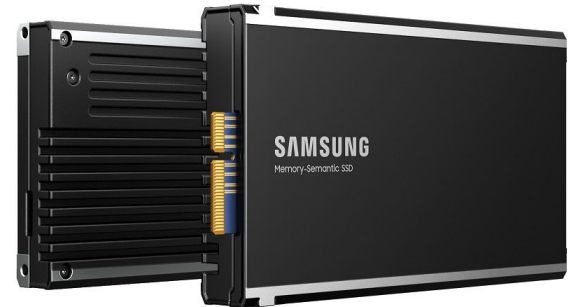


# Putting SSDs with CXL Memory Expander

Which type of device to use? Type-1, Type-2, or Type-3 when using SSD as memory expander?

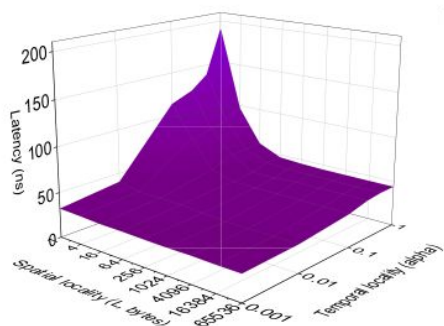
## Type-3:

- (in CXL 1.0, 2.0): Only one Type-1 or Type-2 device allowed per CXL root, hence Type-3 are more scalable.
- Type-1/2 can be more complex, caches, all load/store requests require checking the cache states of PCIe storage computing complex

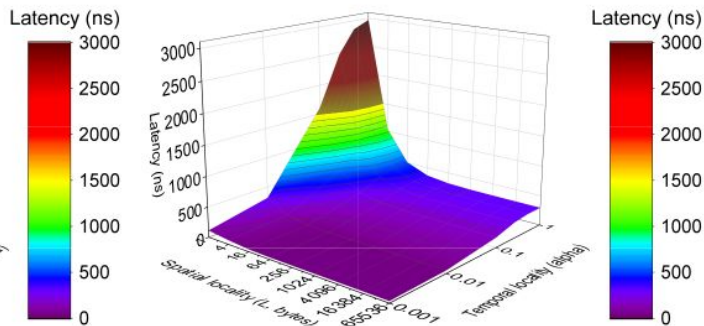


**Hence, a Type-3 device type is the ideal CXL device for a “memory expander”**

# CXL + Flash SSDs: Can Flash do it?



(a) LocalDRAM.



(b) CXL-SSD.

## Can we use NAND flash SSDs as memory expander?

- What latencies one get with the granularity mismatch?
  - Cache line : 64B, flash pages : 8-16 KiB
  - DRAM: 100s of nanoseconds, vs. flash in 10-100 microseconds
- What is the access pattern for common workloads?
- Can we optimize latencies in any manner? Prefetching, buffering, caching?
- How about flash P/E limitations? Can it endure small 64B writes?

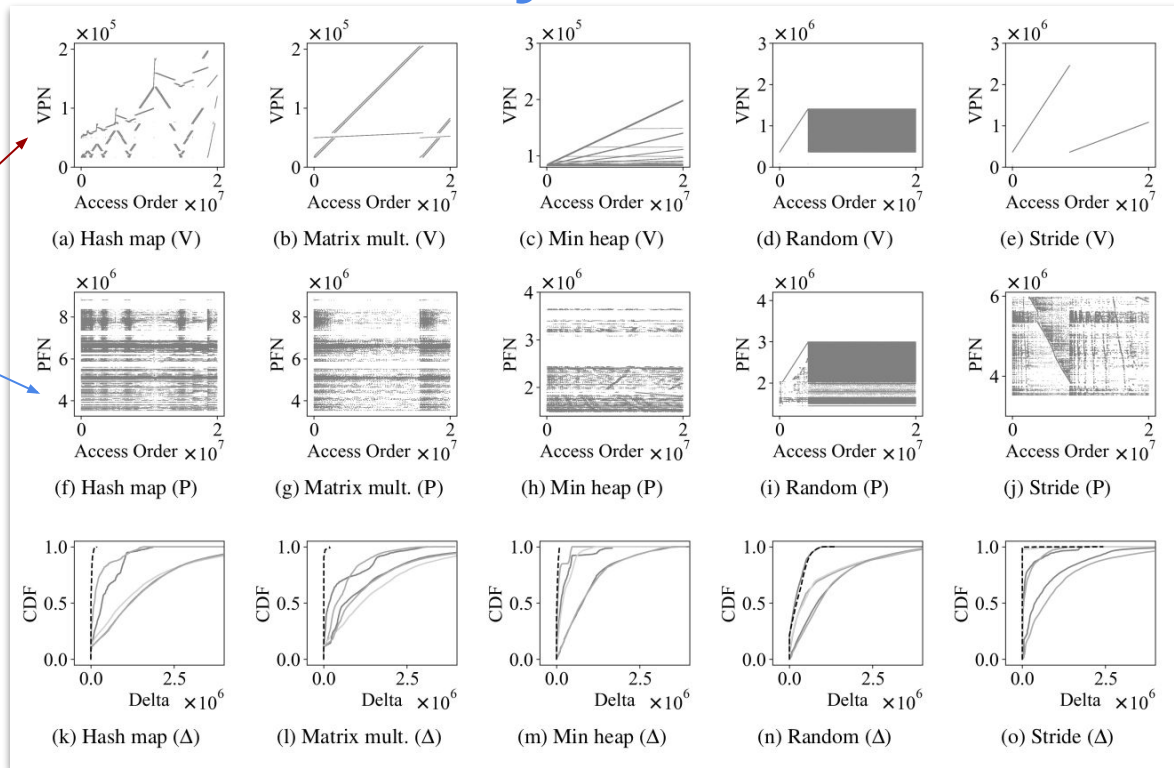


# CXL-Enabled SSDs - Virtual vs. Physical Addresses

⇒ Shows that the access pattern at the **virtual address level** do not correspond to the **physical address level**

**Why?**

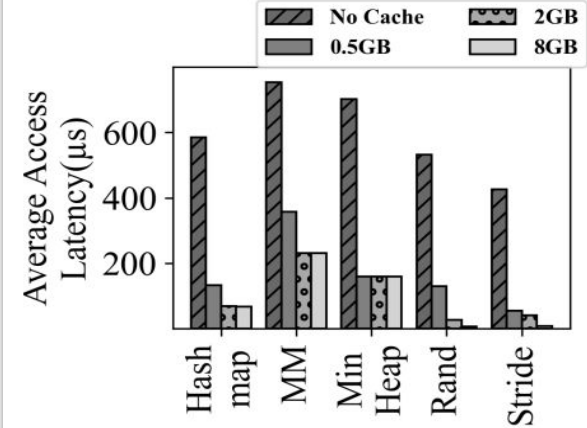
Just basic prefetching is not effective to hide latencies



# Impact of Caching

Inter-arrival time of 64B requests has a huge impact

- Queuing delays w/o cache
- Small amount of cache helps (0.5GB)



(a) Average access latency

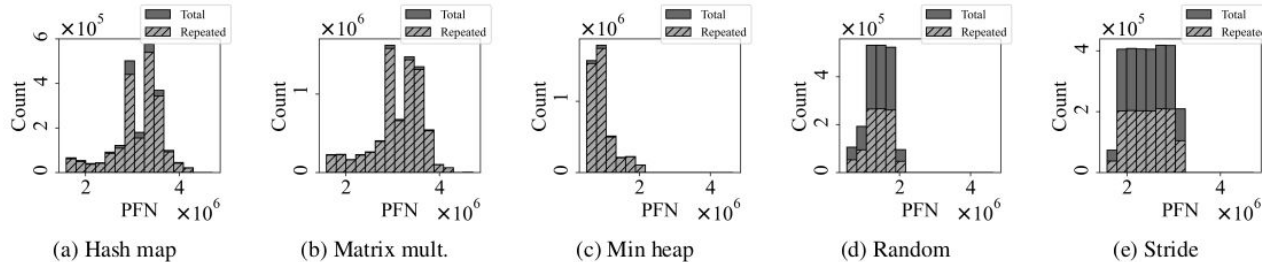
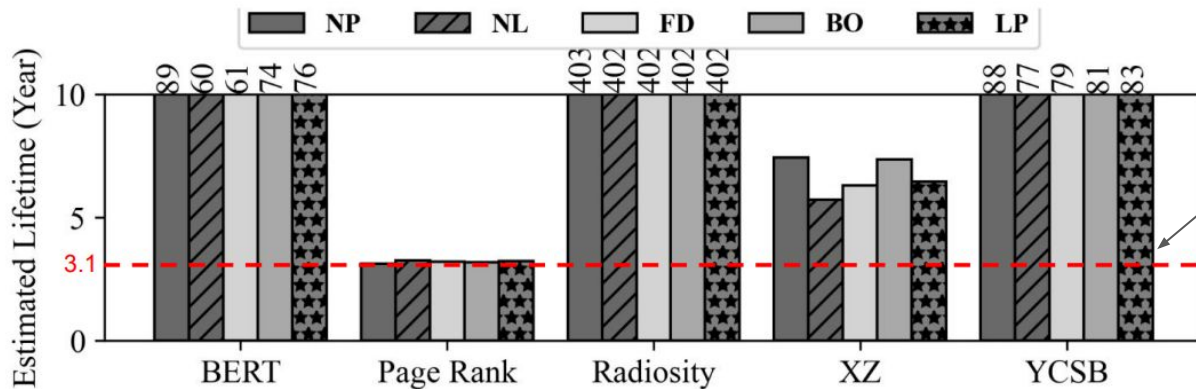


Figure 6: Flash memory read count for physical memory frames. The solid bar represents the total number of reads, while the shaded bar, the number of repeated reads. A repeated read is a read request to an outstanding read request.

Lots of repeated accesses for the same page!

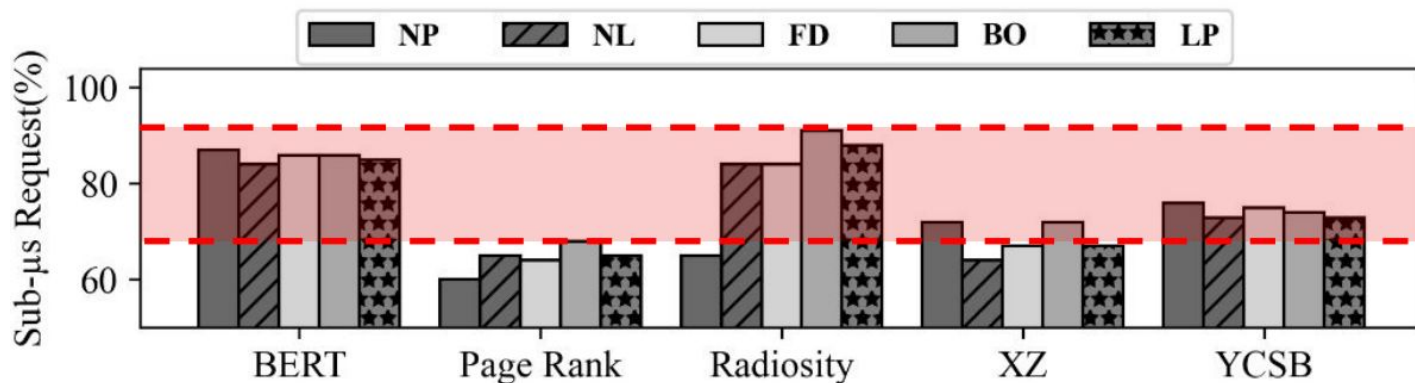
Multiple 64B requests go into the same flash page **(Keep track of it)**

# Workload-level Performance



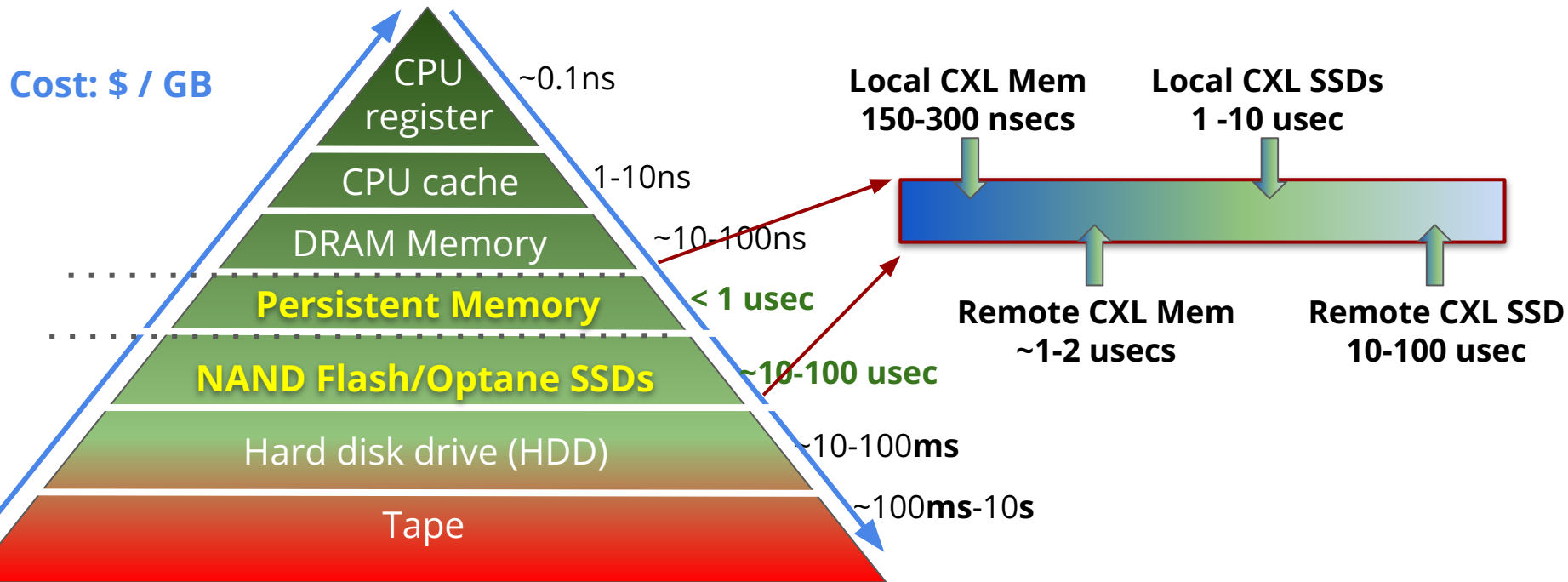
**3.1 years (min) with the workloads SLC flash, 100K P/E)**

**Can hide high flash latencies**



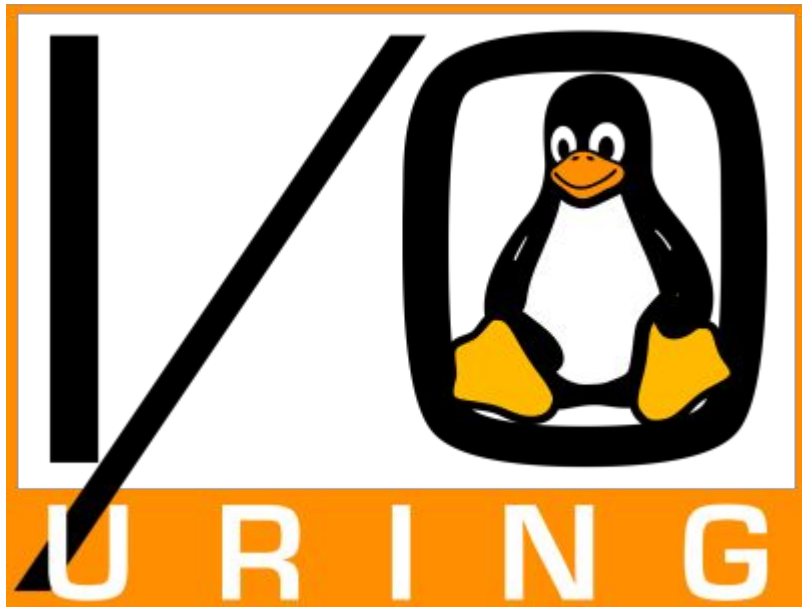


# The New(er) Triangle of Storage-Memory Continuum



Instead of discrete steps, it is a continuous spectrum now: **Continuum**

# io\_uring : What is it and why you should care?

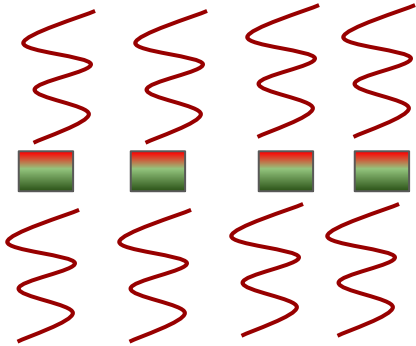


What is io\_uring?, [https://unixism.net/loti/what\\_is\\_io\\_uring.html](https://unixism.net/loti/what_is_io_uring.html)

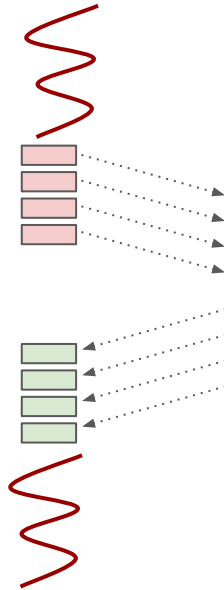
# The Long Debate: How to get Concurrency?

## Threads versus Events (Asynchronous)

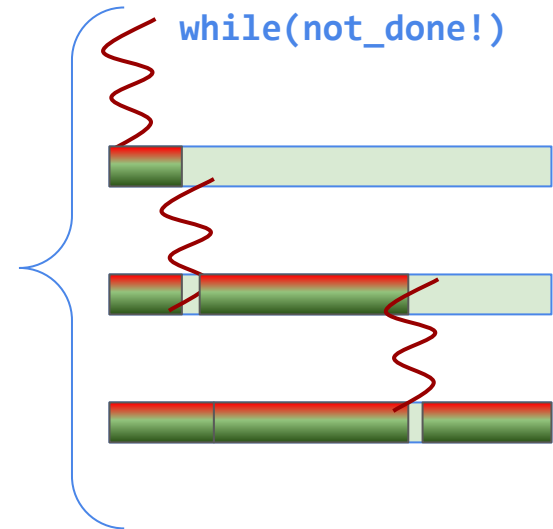
Blocking I/O



Asynchronous I/O



**Non-Blocking I/O** and **Asynchronous I/O** are two different things!



# Linux I/O Options

Standard POSIX I/O **blocking** read/write calls:

- <https://man7.org/linux/man-pages/man2/read.2.html>
- <https://man7.org/linux/man-pages/man2/write.2.html>

|                     | <u>BLOCKING</u> | <u>NON-BLOCKING</u>                               |
|---------------------|-----------------|---|
| <u>SYNCHRONOUS</u>  | READ/<br>WRITE  | READ/WRITE<br>(O_NONBLOCK)<br>I/O<br>MULTIPLEXING |
| <u>ASYNCHRONOUS</u> | —               | AIO   |

Make I/O calls **non-blocking** : set O\_NONBLOCK flag on the file descriptor

- <https://man7.org/linux/man-pages/man2/fcntl.2.html> (O\_NONBLOCK)

**Asynchronous I/O** on Linux : libaio and POSIX AIO

- <https://github.com/littledan/linux-aio>
- Example of how to use libaio: <https://github.com/axboe/fio/blob/master/engines/libaio.c>

# AIO Issues

## SIGNAL based delivery of completion

- Preemption and context switch
- Needs care for signal-safe function execution

Linux' AIO works truly "asynchronously" under very restricted conditions:

- works only with O\_DIRECT modes (alignment, and size restrictions)
- works only when the file's metadata is available  
(otherwise blocks until the metadata is fetched)
- can block based on device's queue capacity
- needs to memcpy of I/O metadata (~100 bytes)

Good introduction: [https://unixism.net/loti/async\\_intro.html](https://unixism.net/loti/async_intro.html) and [https://kernel.dk/io\\_uring.pdf](https://kernel.dk/io_uring.pdf)

Signal Handling in a Multi-Threaded Application in Linux, <https://www.baeldung.com/linux/signals-multi-threaded-app>  
Re: [PATCH 09/13] aio: add support for async openat(), <https://lwn.net/Articles/671657/>

Archive- [Article, Thread](#)  
link:

```
On Mon, Jan 11, 2016 at 2:07 PM, Benjamin LaHaise <bcr1@kvack.org> wrote:  
> Another blocking operation used by applications that want aio  
> functionality is that of opening files that are not resident in memory.  
> Using the thread based aio helper, add support for IOCB_CMD_OPENAT.
```

So I think this is ridiculously ugly.

AIO is a horrible ad-hoc design, with the main excuse being "other, less gifted people, made that design, and we are implementing it for compatibility because database people - who seldom have any shred of taste - actually use it".

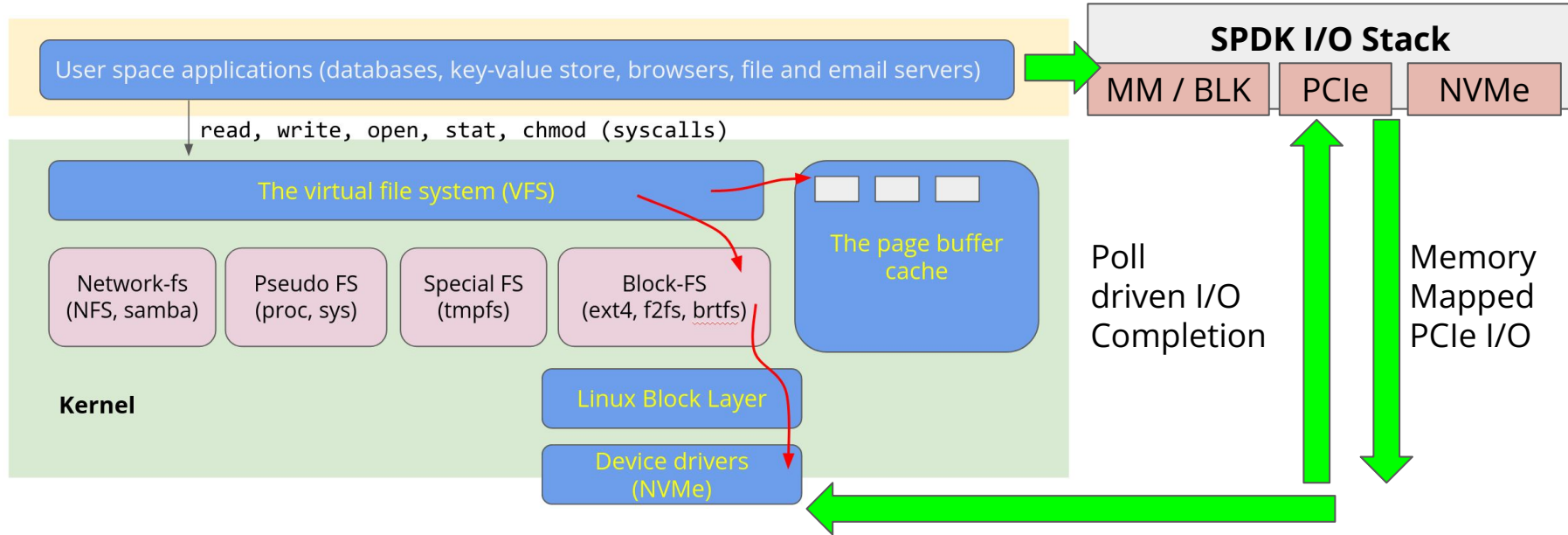
But AIO was always really really ugly.

# Cost of these Interfaces

TABLE I: Categories of system-call techniques

| Kind  | Mechanism    | Examples                   | per sys request |        |              |
|-------|--------------|----------------------------|-----------------|--------|--------------|
|       |              |                            | traps           | csw    | cost[ns]     |
| Sync  | Blocking     | read(), write()            | 1               | 2      | 955 ±1069    |
| Sync  | Non-Blocking | SOCK_NONBLOCK<br>& epoll() | [1, 3]          | [2, 6] | 1656 ±1318   |
| Async | Callback     | POSIX AIO [13]             | 1               | 2, 3   | 6224 ±12 232 |
| Async | Queue-based  | Linux AIO                  | ]0, 2]          | ]1, 4] | 1922 ±1467   |

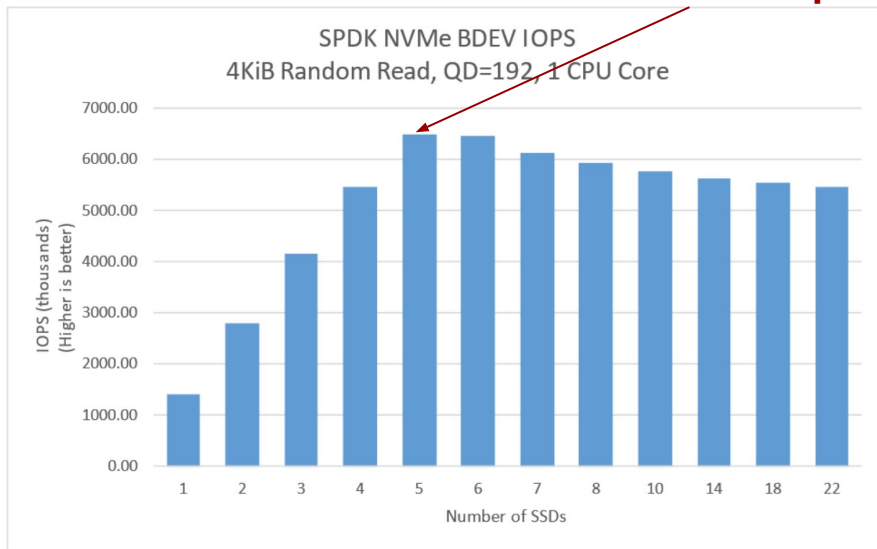
# Skip the OS Complexity: The SPDK Stack



- A user-space I/O framework for NVMe devices (only)
- Block-level abstraction (no file system, but there are research prototypes)
- Has user-space mapped drivers (<https://spdk.io/doc/userspace.html>)
- Designed for light-weight I/O, best performance (eschews many core OS features)

# SPDK can have the Highest Performance

**6 Million Ops/core**



**~30 Million Ops/server**



2 CPU sockets, Intel(R) Xeon(R) Gold 6348 CPU @ 2.60GHz

22x Kioxia® KCM61VUL3T20 3.2TBs (FW: 0105) (10 on CPU NUMA Node 0, 12 on CPU NUMA Node 1)

SPDK NVMe BDEV Performance Report Release 23.05, June 2023,

[https://ci.spdk.io/download/performance-reports/SPDK\\_nvme\\_bdev\\_perf\\_report\\_2305.pdf](https://ci.spdk.io/download/performance-reports/SPDK_nvme_bdev_perf_report_2305.pdf)



# Intricately Linked Issues

What is the system call interface

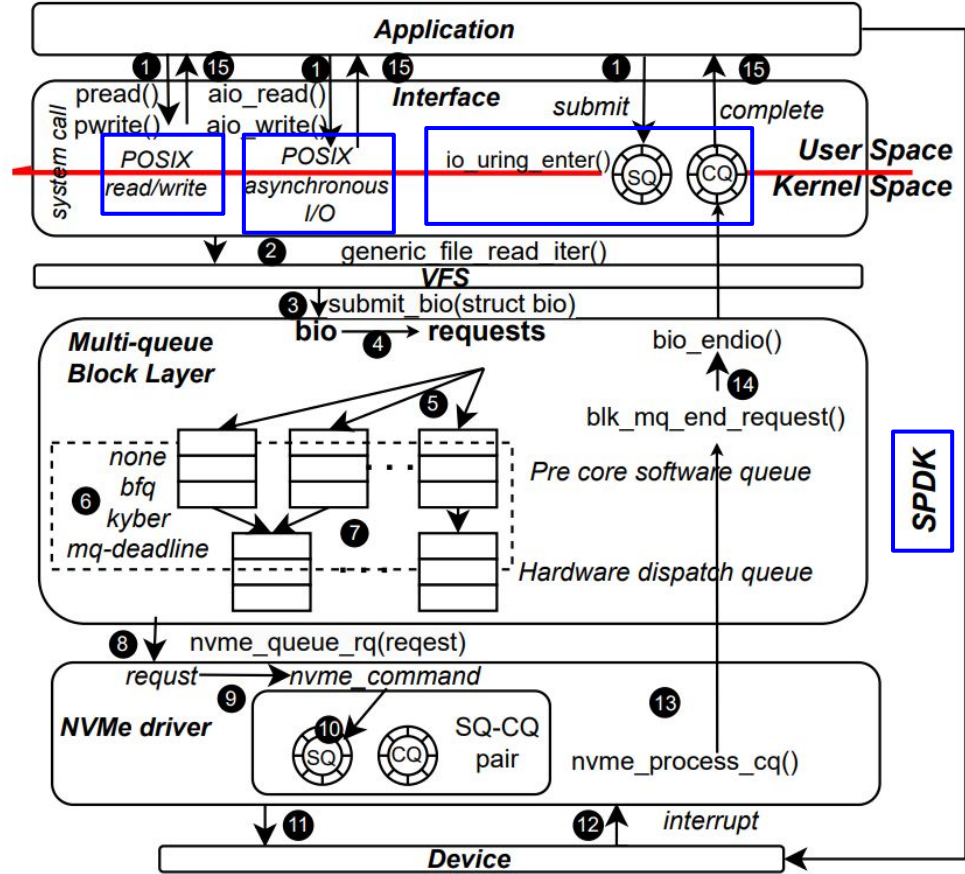
What is the kernel threading model

Signal vs queuing

What is the cost of scheduling, context switching

Management of concurrency

Programming languages (error handling)



# Background Reading on this Topic

Because the original of the following paper by Lauer and Needham is not widely available, we are reprinting it here. If the paper is referenced in published work, the citation should read: "Lauer, H.C., Needham, R.M., "On the Duality of Operating System Structures," in Proc. Second International Symposium on Operating Systems, IRIA, Oct. 1978, reprinted in Operating Systems Review, 13,2 April 1979, pp. 3-19.

## On the Duality of Operating System Structures

Hugh C. Lauer  
Xerox Corporation  
Palo Alto, California

Roger M. Needham\*  
Cambridge University  
Cambridge, England

### Abstract

Many operating system designs can be placed into one of two very rough categories, depending upon how they implement and use the notions of process and synchronization. One category, the "Message-oriented System," is characterized by a relatively small, static number of processes with an explicit message system for communicating among them. The other category, the "Procedure-oriented System," is characterized by a large, rapidly changing number of small processes and a process synchronization mechanism based on shared data.

In this paper, it is demonstrated that these two categories are duals of each other and that a system which is constructed according to one model has a direct counterpart in the other. The principal conclusion is that neither model is inherently preferable, and the main consideration for choosing between them is the nature of the machine architecture upon which the system is being built, not the application which the system will ultimately support.

This is an empirical paper, in the sense of empirical studies in the natural sciences. We have observed a number of samples from a class of objects and identified a classification of some of their properties. We have then generalized our classification and constructed abstract models to describe these properties. With the aid of these models, we were able to make some observations about the nature of the objects themselves, observations which are supported by other experimental evidence. Finally, we have drawn some conclusions about the class of objects which better aid our understanding of that class and the decisions which affect the design of members of that class.

The universe in this investigation is the class of operating systems, and the properties in which we are interested are the ways in which the concepts of process, synchronization, and interprocess communication occur within these systems and among their clients. There appear to be two general categories in this respect, which we designate the *Message-oriented Systems* and the *Procedure-oriented Systems*. Most systems which we have observed tend to be biased fairly strongly in favour of one or the other, rather than being neutral or indeterminate. Moreover,

\* This work was done while the author was on sabbatical leave at the Xerox Palo Alto Research Center during the summer of 1977.

## Why Threads Are A Bad Idea (for most purposes)

John Ousterhout  
Sun Microsystems Laboratories

john.ousterhout@eng.sun.com  
http://www.sunlabs.com/~ouster

### Introduction

- 1 **Threads:**
  - Grew up in OS world (processes).
  - Evolved into user-level tool.
  - Proposed as solution for a variety of problems.
  - Every programmer should be a threads programmer!
- 2 **Problem: threads are very hard to program.**
- 3 **Alternative: events.**
- 4 **Claims:**
  - For most purposes proposed for threads, events are better.
  - Threads should be used only when true CPU concurrency is needed.

Why Threads are A Bad Idea

September 28, 1995, slide 2

## Why Events Are A Bad Idea (for high-concurrency servers)

Rob von Behren, Jeremy Condit and Eric Brewer  
Computer Science Division, University of California at Berkeley  
[jrvb, jcondit, brewer]@cs.berkeley.edu  
http://cspicricio.cs.berkeley.edu/

ing has been highly touted in recent years as a highly concurrent application. If these systems, we now believe this. Specifically, we believe that threads are a bad idea for most purposes. Specifically, we believe that threads are a bad idea for most purposes. Specifically, we believe that threads are a bad idea for most purposes.

applications such as Internet processing databases present a to application designers. First, of concurrent tasks requires the nature. Second, these systems maximum capacity, which creates high sensitivity to scheduling it be handled with care to avoid conditions and subtle corner cases which makes debugging and code

e historically failed to meet these by researchers to conclude that ing is the best (or even only) performance in highly concurrent gives four primary arguments are:

- Inexpensive synchronization due to multithreading;
- Lower overhead for managing state (no application-level information; and
- More flexible control flow (not just calling

We have made extensive use of events in high-concurrency environments, including NSEDA [17], and Inktank's Traffic Server. With these systems, we realized that the proposed to event systems; many had been implemented with threads, and the rest as

Ultimately, our experience led us to conclude that event-based programming is the wrong choice for concurrent systems. We believe that (1) threads are a more natural abstraction for high-concurrency and that (2) small improvements to compilers runtime systems can eliminate the historical use events. Additionally, threads are more to compiler-based enhancements; we believe paradigm for highly concurrent applications package with better compiler support.

Section 2 compares events with threads, the common arguments against threads. Next, explains why threads are particularly natural for high-concurrency servers. Section 4 explores of compiler support for threads. In Section 5, our approach with a simple web server. Finally covers (some) related work, and Section 7

### 2 Threads vs. Events

The debate between threads and events is a very old one. Lauer and Needham attempted to end the discussion in 1978 by showing that message-passing systems and process-based systems are duals, both in terms of program structure and performance characteristics [10]. Nonetheless, in recent years many authors have declared the need for event-driven programming for highly concurrent systems [11, 12, 17].

HotOS IX: The 9th Workshop on Hot Topics in Operating Systems

19



## Scala Actors: Unifying thread-based and event-based programming\*

Philipp Haller<sup>†</sup>, Martin Odersky

EPFL, Switzerland

### ARTICLE INFO

Keywords:  
Concurrent programming  
Actors  
Threads  
Events

### ABSTRACT

There is an impedance mismatch between message-passing concurrency and virtual machines, such as the JVM. VMs usually map their threads to heavyweight OS processes. Without a lightweight process abstraction, users are often forced to write parts of concurrent applications in an event-driven style which obscures control flow, and increases the burden on the programmer.

In this paper we show how thread-based and event-based programming can be unified under a single actor abstraction. Using advanced abstraction mechanisms of the Scala programming language, we implement our approach on unmodified JVMs. Our programming model integrates well with the threading model of the underlying VM. © 2008 Elsevier B.V. All rights reserved.

### 1. Introduction

Concurrency issues have lately received enormous interest because of two converging trends: first, multi-core processors make concurrency an essential ingredient of efficient program execution. Second, distributed computing and web services are inherently concurrent. Message-based concurrency is attractive because it might provide a way to address the two challenges at the same time. It can be seen as a higher-level model for threads with the potential to generalize to distributed computation. Many message passing systems used in practice are instantiations of the actor model [28,2]. A popular implementation of this form of concurrency is the Erlang programming language [4]. Erlang supports massively concurrent systems such as telephone exchanges by using a very lightweight implementation of concurrent processes [3,36].

On mainstream platforms such as the JVM [34], an equally attractive implementation was, as yet, missing. Their standard concurrency constructs, shared-memory threads with locks, suffer from high memory consumption and context-switching overhead. Therefore, the interleaving of independent computations is often modeled in an event-driven style on these platforms. However, programming in an explicitly event-driven style is complicated and error-prone, because it involves an inversion of control [41,13].

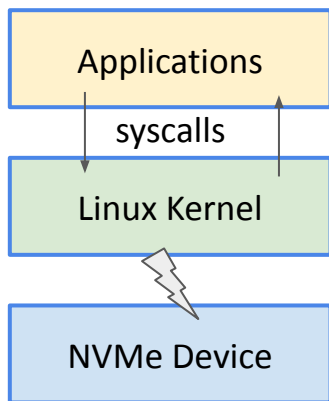
In previous work [24], we developed event-based actors which let one program event-driven systems without inversion of control. Event-based actors support the same operations as thread-based actors, except that the receive operation cannot return normally to the thread that invoked it. Instead the entire continuation of such an actor has to be a part of the receive operation. This makes it possible to model a suspended actor by a continuation closure, which is usually much cheaper than suspending a thread.

In this paper we present a unification of thread-based and event-based actors. An actor can suspend with a full thread stack (reactor) or it can suspend with just a continuation closure (reactor). The first form of suspension corresponds to thread-based, the second form to event-based programming. The new system combines the benefits of both models.

\* A preliminary version of the paper appears in the proceedings of COORDINATION 2007, LNCS 4867, June 2007.  
<sup>†</sup> Corresponding address: EPFL, Station 14, 1015 Lausanne, Switzerland. Tel.: +41 21 693 6883, fax: +41 21 693 6960.  
E-mail address: philipp.haller@epfl.ch (P. Haller).

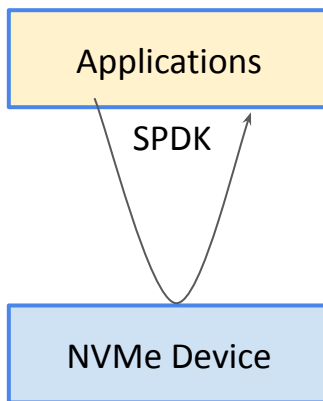
0304-3975/\$ - see front matter © 2008 Elsevier B.V. All rights reserved.  
doi:10.1016/j.tcs.2008.05.019

# Storage APIs: Recap



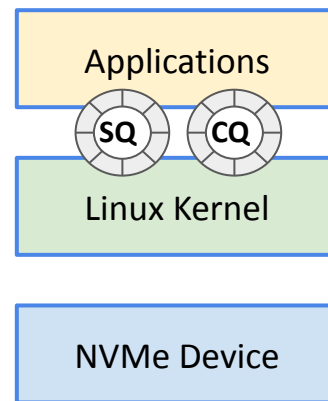
## Libaio:

- + Async I/O
- + Any files/FSeS
- + Any device: HDD, NVMe
  
- Async only with direct I/O
- Performance
- Metadata management



## SPDK:

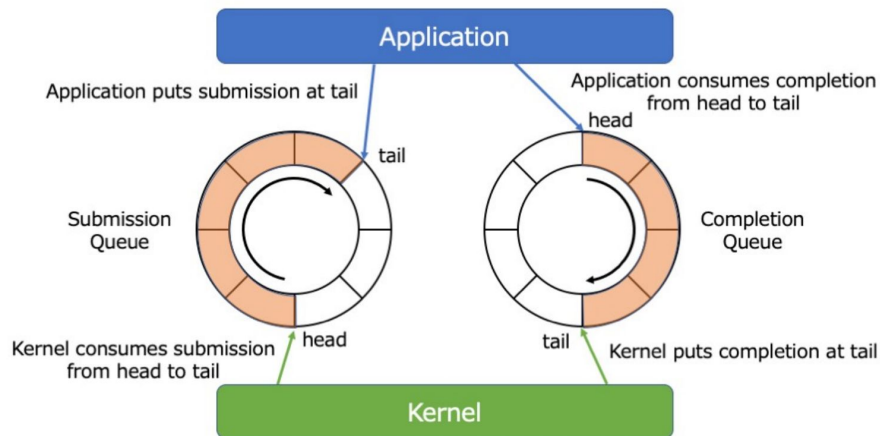
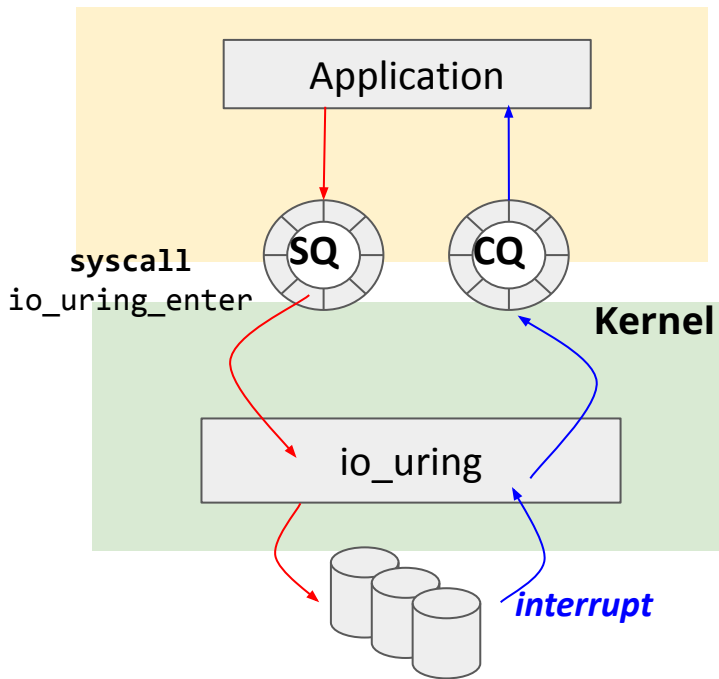
- + Performance
- + Close application integration
- + No syscall or interrupts
  
- Only NVMe
- No kernel assistance
- Scalability and brittle



**io\_uring**

**Best of both worlds?**

# io\_uring: A Structured Approach to Asynchronous I/O

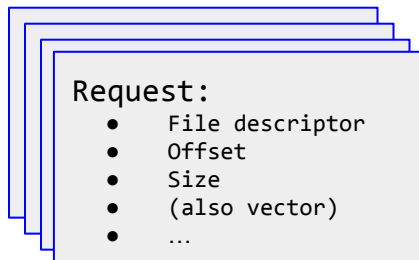
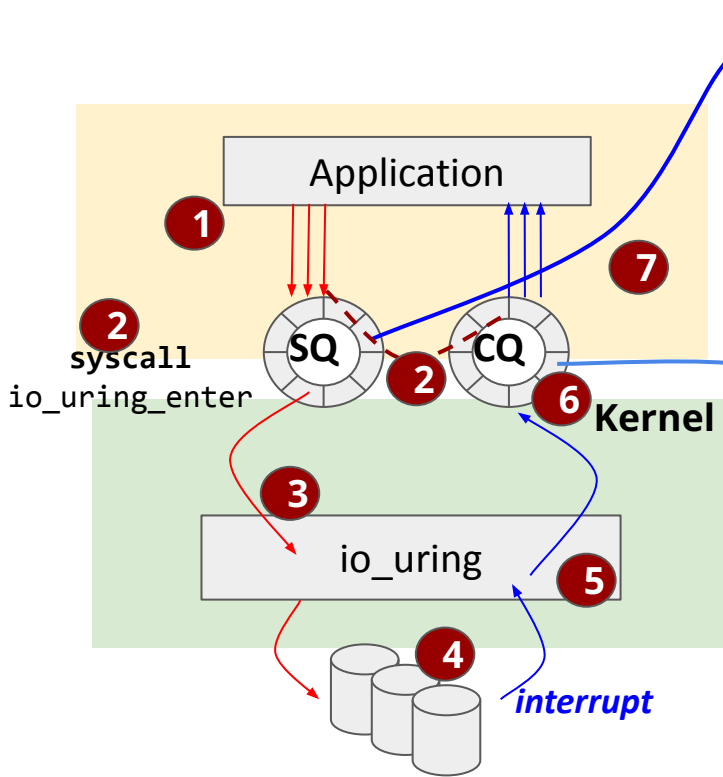


## Producer-consumer pattern

- SQ: producer = application (tail), consumer = kernel (head)
- CQ: producer = kernel (tail), consumer = application (head)

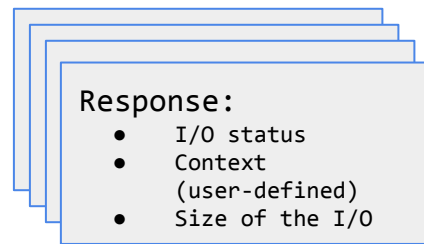
Head and tail pointers manipulation with exclusive write ownership

# io\_uring: A Structured Approach to Asynchronous I/O



Applications can

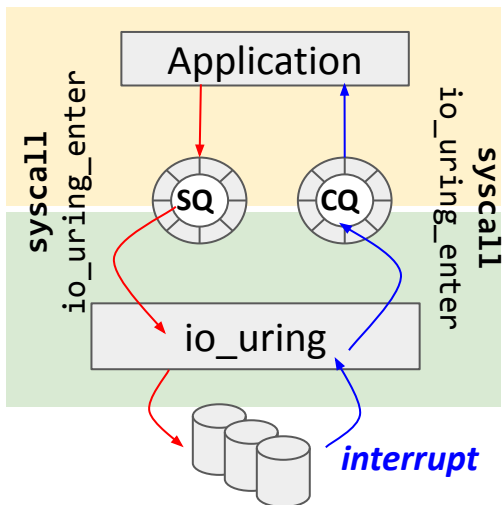
- **Async I/O**
- I/O on any fd type (+net)
- Queue requests (batch)
- Vector I/O
- Optimize (fixed FD, pin)



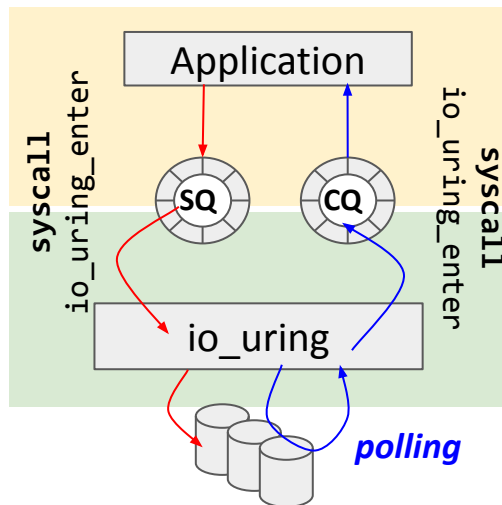
# The three new Syscalls

1. **io\_uring\_setup**: This call is for creating the ring structure (queue-depth, I/O completion and notification modes)
  - a. Completion polling by the kernel on the device (IORING\_SETUP\_\*\*IOPOLL
  - b. Kernel polling for submission (IORING\_SETUP\_\*\*SQPOLL, zero system call)
2. **io\_uring\_enter**: This call enters the kernel and tells it to process I/O requests (any type and extensible, not just storage I/O)
  - a. Networking, ZNS, Programmable storage and more
  - b. Replacement for the ioctl() call: a private interface between a device driver and application
3. **io\_uring\_register**: This call is for registering specific fd, buffers, file ranges that are being used frequently to put them on an optimized fast path

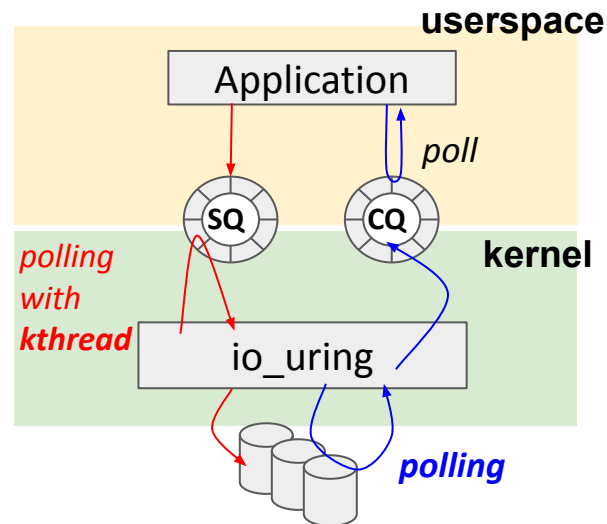
# Three Modes of Operations



(a) io\_uring (default)



(b) with completion polling



(c) with submission polling



## Understanding Modern Storage APIs: A systematic study of libaio, SPDK, and io\_uring

Diego Didona, Jonas Pfefferle  
Nikolas Ioannou, Bernard Metzler  
IBM Research Europe  
Zurich, Switzerland  
{ddj, jpf, nio, bmt}@ibm.zurich.com

Animesh Trivedi  
VU Amsterdam  
Amsterdam, Netherlands  
a.trivedi@vu.nl

### ABSTRACT

Recent high-performance storage devices have exposed software inefficiencies in existing storage stacks, leading to a new breed of I/O stacks. The newest storage API of the Linux kernel is `io_uring`. We perform one of the first in-depth studies of `io_uring`, and compare its performance and disadvantages with the established `libaio` and `SPDK` APIs. Our key findings reveal that (i) polling design significantly impacts performance, (ii) with enough CPU cores `io_uring` can deliver performance close to that of `SPDK`, and (iii) performance scalability over multiple CPU cores and devices requires careful consideration and necessitates a hybrid approach. Last, we provide design guidelines for developers of storage intensive applications.

### ACM Reference Format:

Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler and Animesh Trivedi. 2022. Understanding Modern Storage APIs: A systematic study of `libaio`, `SPDK`, and `io_uring`. In *The 15th ACM International Systems and Storage Conference (SYSTOR '22)*, June 13–15, 2022, Haifa, Israel. ACM, New York, NY, USA, 8 pages. <https://doi.org/10.1145/3534056.3534945>

### 1 INTRODUCTION

Modern non-volatile memory (NVM) storage technologies, like `Flash` and `Optane SSDs`, can support down to single digit `µsecond` latencies, and up to multi GB/s bandwidths with millions of I/O operations per second (IOPS). CPU performance improvements have stalled over the past years due to various manufacturing and technical limitations [5].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Short copying with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions.acm.org](https://permissions.acm.org).

SYSTOR '22, June 13–15, 2022, Haifa, Israel  
© 2022 Association for Computing Machinery.  
ACM ISBN 978-1-4503-9100-9/22...\$5.00  
<https://doi.org/10.1145/3534056.3534945>

As a result, researchers have put considerable effort into identifying new CPU-efficient storage APIs, abstractions, designs, and optimizations [2, 3, 11, 13, 15, 19, 22, 25, 26, 30, 31]. One specific API, `io_uring`, has drawn much attention from the community due to its versatile and high performance interface [5, 15, 16, 18, 27, 34]. `io_uring` was introduced in 2019 and has been merged in Linux v5.1. It brings together many well established ideas from the high performance storage and networking communities, such as asynchronous I/O, shared memory-mapped queues, and polling (Section 2) [9, 10, 31, 32].

With the addition of `io_uring`, Linux now has multiple ways of accessing a storage device. In this paper, we look at Linux Asynchronous I/O (`libaio`) [6, 24], the Storage Performance Development Kit (`SPDK`) from Intel® [13], and `io_uring` [15, 17, 18]. These APIs have different parameters, deployment models, and characteristics, which make understanding their performance and limitations a challenging task. The use of the `io_uring` API and its performance has been the focus of recent studies [7, 28, 33, 36]. However, to the best of our knowledge, there is no systematic study of these APIs that provides design guidelines for the developers of I/O intensive applications. There has also been an extensive body of work in studying system call overhead [29], implementing better interrupt management for I/O devices [30], leveraging polling for fast storage devices [38], using I/O speculation for second-scale devices such as NVM drives [35], and improving the performance of the Linux block layer in general [3, 39, 40]. These works are orthogonal to ours, since they explore designing new storage stacks, while we focus on the performance characteristics of state-of-the-art APIs that are readily available in Linux.

Our main contributions include (i) a systematic comparison of `libaio`, `io_uring`, and `SPDK`, that evaluates their latency, IOPS, and scalability behaviors; (ii) a first-of-its-kind detailed evaluation of the different `io_uring` configurations; and (iii) design guidelines for high-performance applications using modern storage APIs. Our key findings reveal that:

## Performance Characterization of Modern Storage Stacks: POSIX I/O, libaio, SPDK, and io\_uring

Zebin Ren  
z.ren@vu.nl  
Vrije Universiteit Amsterdam  
Amsterdam, Netherlands

Animesh Trivedi  
a.trivedi@vu.nl  
Vrije Universiteit Amsterdam  
Amsterdam, Netherlands

### Abstract

Linux storage stack offers a variety of storage I/O stacks and APIs such as POSIX I/O, asynchronous I/O (`libaio`), high-performance asynchronous I/O (emerging `io_uring`) or `SPDK`, the last of which completely bypasses the kernel. Despite their availability, there has not been a systematic study of their performance and overheads. In order to aid our understanding, in this work we systematically characterize performance, scalability and microarchitectural properties of popular Linux I/O APIs on high-performance storage hardware (Intel Optane SSDs). Our characterization reveals that: (1) at low I/O loads, all APIs perform competitively with each other, with polling helping the performance by 1.7x, but consuming 2.3x CPU instructions; (2) at high loads and scale, `io_uring` is more than an order of magnitude slower than `SPDK`; (3) at high loads and scale, the benchmarking tool (io) itself becomes a bottleneck; (4) state-of-practice Linux block I/O schedulers (BFQ, mq-deadline, and Kyber) introduce significant (up to 50%) overheads, and their use of global locks hinder their scalability. All artifacts from this work are available at <https://github.com/afgaresearch/Performance-Characterization-Storage-Stacks>.

**CCS Concepts:** • Software and its engineering → Secondary storage; Operating systems.

**Keywords:** Linux storage stack, `io_uring`, `SPDK`, Efficiency, Measurements

### ACM Reference Format:

Zebin Ren and Animesh Trivedi. 2023. Performance Characterization of Modern Storage Stacks: POSIX I/O, libaio, SPDK, and io\_uring. In *3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '23)*, May 8, 2023, Rome, Italy. ACM, New York, NY, USA, 11 pages. <https://doi.org/10.1145/3578353.3589545>



This work is licensed under a Creative Commons Attribution International 4.0 License.  
CHEOPS '23, May 8, 2023, Rome, Italy  
© 2023 Copyright held by the owner(s).  
ACM ISBN 978-1-4503-9100-9/23...\$5.00  
<https://doi.org/10.1145/3578353.3589545>

### 1 Introduction

Modern storage devices such as Intel Optane SSDs can deliver millions of IOPS (I/O operations per second) with single-digit microseconds (`µsec`) access latencies [7, 17]. Meanwhile, the CPU performance has remained relatively stable as Moore's Law driven performance gains stall [29]. Consequently, the stalled CPU performance with high-performance storage hardware has exposed many previously hidden software overheads in the storage stack implementations, thus leading to a series of efforts to redesign and optimize the storage stack focusing on lock contentions, polling, copy elimination, new interfaces, scheduling, context switches, asynchronous I/O paths, interrupt and system call eliminations [3, 18, 20, 25, 30, 36, 37, 39, 40, 45, 56, 59, 66, 68].

Beyond these optimizations, there have been many efforts to improve the user kernel and user-storage APIs and abstractions. Linux supports two popular and widely used APIs called (synchronous) POSIX file I/O calls [12, 13] and an asynchronous API called `libaio` [3]. Both of these APIs interact via system calls (syscalls) with the Linux kernel which can have high overheads [22, 38, 55]. More recently, Linux developers have introduced a new high-performance I/O API called `io_uring` [8]. It takes many established ideas from the high-performance networking domain (shared-memory queues, asynchronous I/O, polling, shared I/O contexts) and applies them to storage in a unified manner [61, 62]. These advancements are now merged in the Linux storage stack (since v5.1 kernel version), and have shown to deliver high performance and CPU efficiency [22]. All of these APIs (POSIX, `libaio`, `io_uring`) work within the kernel.

The Linux kernel with its generic code execution, functionalities, and features can also introduce significant overheads [51], thus leading to the design of kernel-bypassing user-space storage stacks [24, 34, 49, 74]. The Storage Performance Development Kit (`SPDK`) is one of the most popular and widely used user space I/O libraries, which can deliver up to 10 million IOPS using a single CPU core [2]. However, user space I/O libraries lack many kernel-supported features such as fine-grained isolation, access control, file systems, multi tenancy, and QoS support [48, 64].

In summary, over the past decade, the in-kernel and user-space I/O stacks have undergone a significant development phase. Despite sharing a common functional goal

Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding modern storage APIs: a systematic study of `libaio`, `SPDK`, and `io_uring`. In Proceedings of the 15th ACM International Conference on Systems and Storage (SYSTOR '22).

<https://doi.org/10.1145/3534056.3534945>

Zebin Ren and Animesh Trivedi. 2023. Performance Characterization of Modern Storage Stacks: POSIX I/O, `libaio`, `SPDK`, and `io_uring`. In Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '23). <https://doi.org/10.1145/3578353.3589545>



# Benchmarking Setup

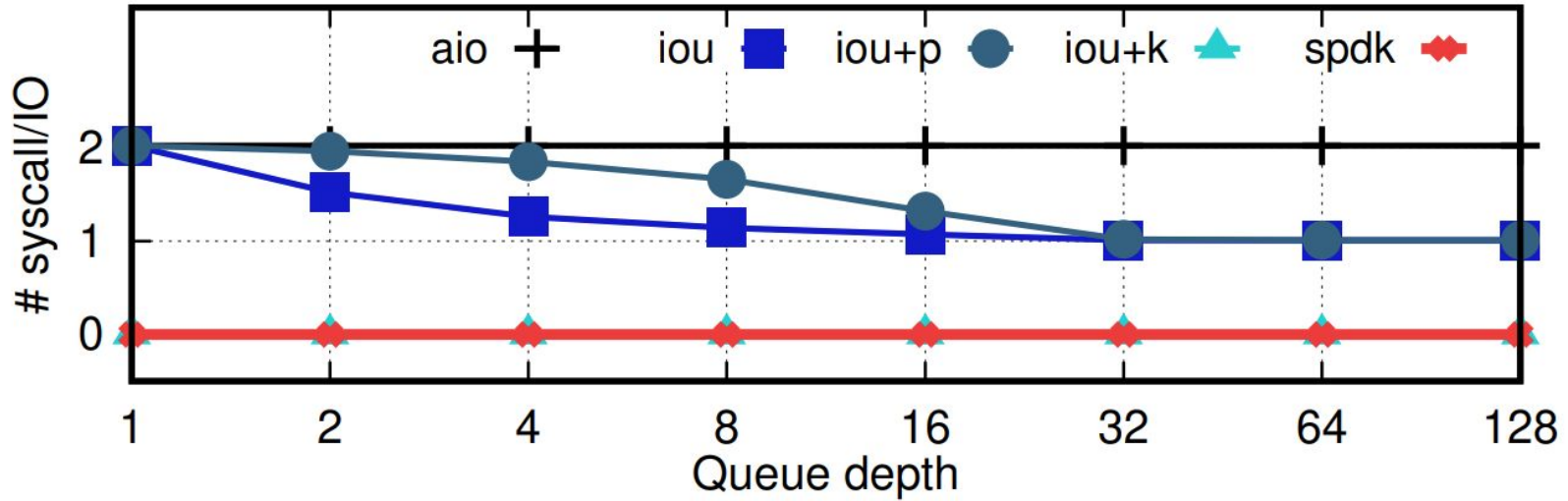
## Setup 1 [Systor'22]:

- 2x Intel® Xeon® E5-2630 (Sandy Bridge), 10 cores/socket ⇒ 20 CPU cores
- 20 Intel® DC P3600 400GB NVMe Flash SSDs ⇒ ~6 Million IOPS

## Setup 2 [CHEOPS'23]:

- 2x Intel® Xeon® Silver 4210R (Cascade Lake), 10 cores/socket ⇒ 20 CPU cores
- 7x Intel Corporation 900P NVMe Optane SSD ⇒ 4.2 Million IOPS

# Number of System Calls

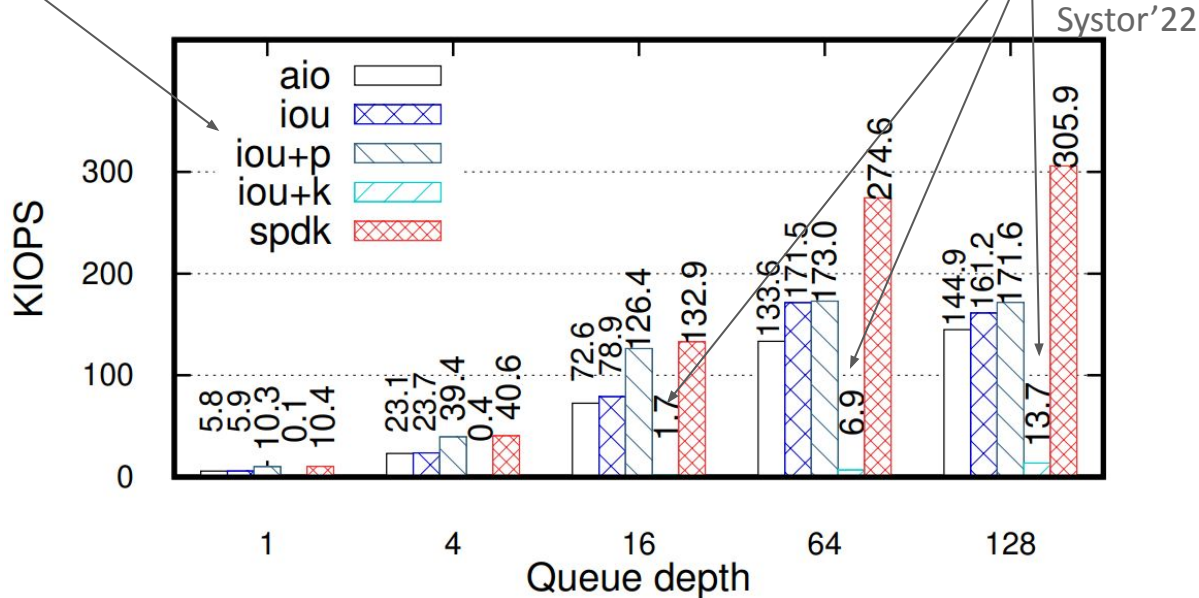


**Doing I/O with zero system calls!**

# Results: Efficiency (single CPU core)

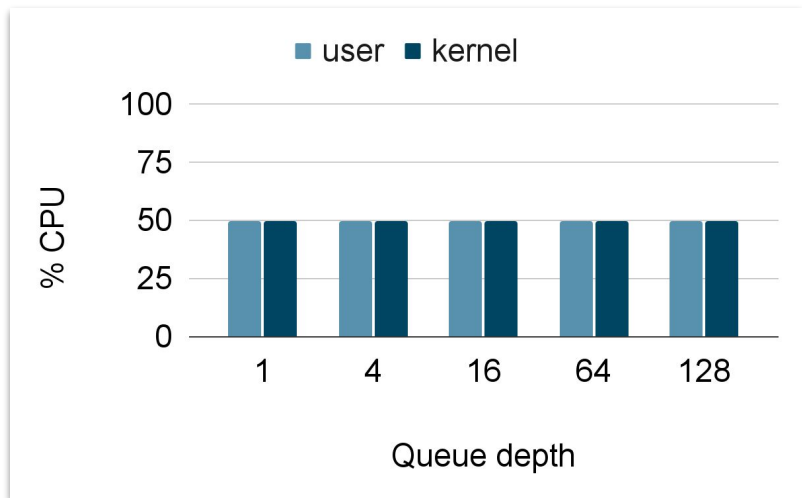
*io\_uring sits between libaio and SPDK*

*Performance collapses with the kernel polling*



# Analysis

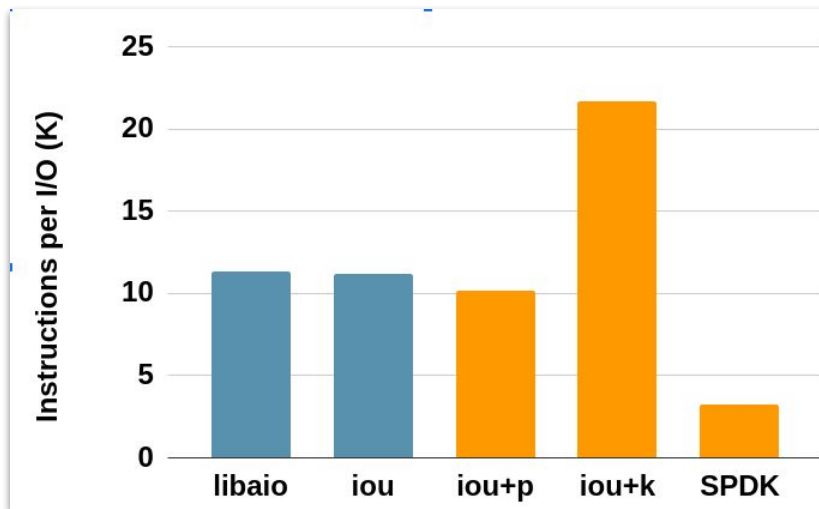
Systor'22



*[Interesting]* 8 milliseconds constant latency for all queue depths!

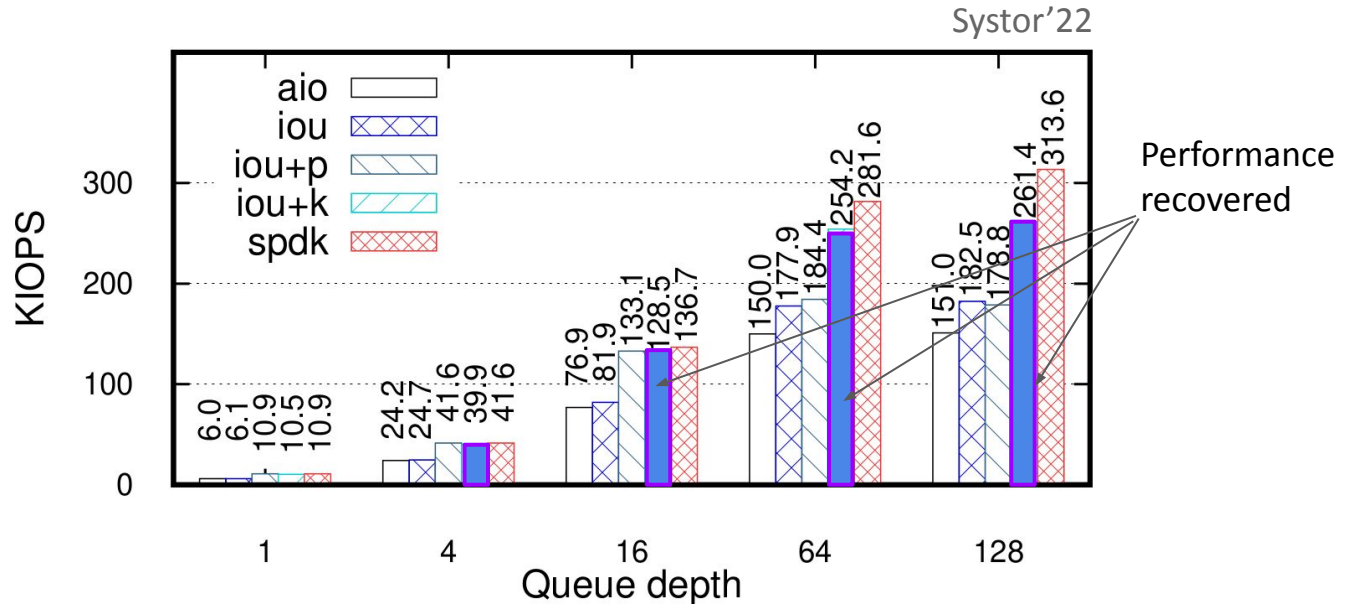
Poor scheduling, and CPU sharing - **Careful!**

CHEOPS'23



SPDK is still 5x more efficient

# Result: Efficiency with TWO CPU cores

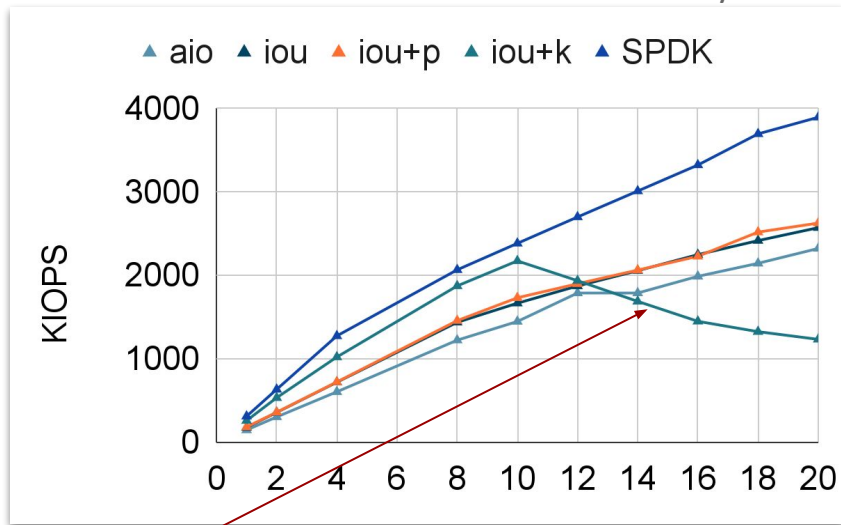


[ aio < iou < iou with polling < iou with kernel poll < SPDK ]

Normal service order can be resumed (**but** at the cost of 2x CPU cores)!

# Results: Scalability

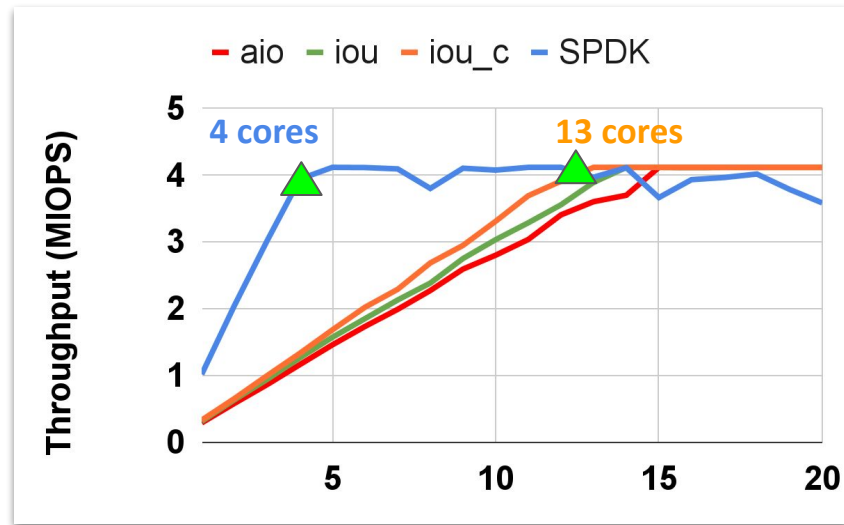
Systor'22



CPU Cores used

**io\_uring kernel polling:** Performance collapses when the number of poller CPU threads increases beyond the cores

CHEOPS'23



**CPU efficiency is still bad:** 10x more CPU cores needed

# io\_uring : Programming Ecosystem

- liburing : <https://github.com/axboe/liburing>
  - 3x syscall based programming can be tricky, hence, a high(er)-level library

## List of manual pages

|  |  |   |  |   |  |  |
|--|--|---|--|---|--|--|
| <ul style="list-style-type: none"><li>• [en] io_uring_check_version(3)</li><li>• [en] IO_URING_VERSION_MAJOR(3)</li><li>• [en] IO_URING_VERSION_MINOR(3)</li><li>• [en] __io_uring_buf_ring_cq_advance(3)</li><li>• [en] io_uring(7)</li><li>• [en] io_uring_buf_ring_add(3)</li><li>• [en] io_uring_buf_ring_advance(3)</li><li>• [en] io_uring_buf_ring_cq_advance(3)</li><li>• [en] io_uring_buf_ring_init(3)</li><li>• [en] io_uring_buf_ring_mask(3)</li><li>• [en] io_uring_check_version(3)</li><li>• [en] io_uring_close_ring_fd(3)</li><li>• [en] io_uring_cq_advance(3)</li><li>• [en] io_uring_cq_has_overflow(3)</li><li>• [en] io_uring_cq_ready(3)</li><li>• [en] io_uring_cqe_get_data(3)</li><li>• [en] io_uring_cqe_get_data64(3)</li><li>• [en] io_uring_cqe_seen(3)</li><li>• [en] io_uring_enter(2)</li><li>• [en] io_uring_enter2(2)</li><li>• [en] io_uring_for_each_cqe(3)</li><li>• [en] io_uring_free_buf_ring(3)</li><li>• [en] io_uring_free_probe(3)</li></ul> | <ul style="list-style-type: none"><li>• [en] io_uring_get_events(3)</li><li>• [en] io_uring_get_probe(3)</li><li>• [en] io_uring_get_sqe(3)</li><li>• [en] io_uring_major_version(3)</li><li>• [en] io_uring_minor_version(3)</li><li>• [en] io_uring_opcode_supported(3)</li><li>• [en] io_uring_peek_cqe(3)</li><li>• [en] io_uring_prep_accept(3)</li><li>• [en] io_uring_prep_accept_direct(3)</li><li>• [en] io_uring_prep_cancel(3)</li><li>• [en] io_uring_prep_cancel64(3)</li><li>• [en] io_uring_prep_close(3)</li><li>• [en] io_uring_prep_close_direct(3)</li><li>• [en] io_uring_prep_connect(3)</li><li>• [en] io_uring_prep_fadvise(3)</li><li>• [en] io_uring_prep_fallocate(3)</li><li>• [en] io_uring_prep_fgetxattr(3)</li><li>• [en] io_uring_prep_files_update(3)</li><li>• [en] io_uring_prep_fsync(3)</li><li>• [en] io_uring_prep_getsock(3)</li><li>• [en] io_uring_prep_link(3)</li><li>• [en] io_uring_prep_link_timeout(3)</li></ul> | <ul style="list-style-type: none"><li>• [en] io_uring_prep_linkat(3)</li><li>• [en] io_uring_prep_madvise(3)</li><li>• [en] io_uring_prep_mkdir(3)</li><li>• [en] io_uring_prep_mkdirat(3)</li><li>• [en] io_uring_prep_msg_ring(3)</li><li>• [en] io_uring_prep_msg_ring_cqe_flags(3)</li><li>• [en] io_uring_prep_msg_ring_fd(3)</li><li>• [en] io_uring_prep_msg_ring_fd_alloc(3)</li><li>• [en] io_uring_prep_multishot_accept(3)</li><li>• [en] io_uring_prep_multishot_accept_direct(3)</li><li>• [en] io_uring_prep_nop(3)</li><li>• [en] io_uring_prep_openat(3)</li><li>• [en] io_uring_prep_openat2(3)</li><li>• [en] io_uring_prep_openat2_direct(3)</li><li>• [en] io_uring_prep_poll_add(3)</li><li>• [en] io_uring_prep_poll_multishot(3)</li><li>• [en] io_uring_prep_poll_remove(3)</li><li>• [en] io_uring_prep_poll_update(3)</li><li>• [en] io_uring_prep_provide_buffers(3)</li><li>• [en] io_uring_prep_read(3)</li><li>• [en] io_uring_prep_read_fixed(3)</li></ul> | <ul style="list-style-type: none"><li>• [en] io_uring_prep_readv(3)</li><li>• [en] io_uring_prep_readv2(3)</li><li>• [en] io_uring_prep_recv(3)</li><li>• [en] io_uring_prep_recv_multishot(3)</li><li>• [en] io_uring_prep_recvmsg(3)</li><li>• [en] io_uring_prep_recvmsg3(3)</li><li>• [en] io_uring_prep_remove_buffers(3)</li><li>• [en] io_uring_prep_rename(3)</li><li>• [en] io_uring_prep_renameat(3)</li><li>• [en] io_uring_prep_send(3)</li><li>• [en] io_uring_prep_send_set_add(3)</li><li>• [en] io_uring_prep_send_zc(3)</li><li>• [en] io_uring_prep_send_zc_fixed(3)</li><li>• [en] io_uring_prep_sendmsg(3)</li><li>• [en] io_uring_prep_sendmsg_zc(3)</li><li>• [en] io_uring_prep_sendto(3)</li><li>• [en] io_uring_prep_setxattr(3)</li><li>• [en] io_uring_prep_shutdown(3)</li><li>• [en] io_uring_prep_socket(3)</li><li>• [en] io_uring_prep_socket_direct(3)</li><li>• [en] io_uring_prep_socket_direct_alloc(3)</li><li>• [en] io_uring_prep_splice(3)</li><li>• [en] io_uring_prep_stats(3)</li></ul> | <ul style="list-style-type: none"><li>• [en] io_uring_prep_symlink(3)</li><li>• [en] io_uring_prep_symlinkat(3)</li><li>• [en] io_uring_prep_sync_file_range(3)</li><li>• [en] io_uring_prep_tee(3)</li><li>• [en] io_uring_prep_timeout(3)</li><li>• [en] io_uring_prep_timeout_remove(3)</li><li>• [en] io_uring_prep_timeout_update(3)</li><li>• [en] io_uring_prep_unlink(3)</li><li>• [en] io_uring_prep_unlinkat(3)</li><li>• [en] io_uring_prep_write(3)</li><li>• [en] io_uring_prep_write_fixed(3)</li><li>• [en] io_uring_prep_writev(3)</li><li>• [en] io_uring_prep_writev2(3)</li><li>• [en] io_uring_queue_exit(3)</li><li>• [en] io_uring_queue_init(3)</li><li>• [en] io_uring_queue_init_params(3)</li><li>• [en] io_uring_recvmsg_cmsg_firsthdr(3)</li><li>• [en] io_uring_recvmsg_cmsg_nexthdr(3)</li><li>• [en] io_uring_recvmsg_name(3)</li><li>• [en] io_uring_recvmsg_wait(3)</li><li>• [en] io_uring_recvmsg_payload(3)</li><li>• [en] io_uring_recvmsg_payload_length(3)</li><li>• [en] io_uring_recvmsg_validate(3)</li></ul> | <ul style="list-style-type: none"><li>• [en] io_uring_register(2)</li><li>• [en] io_uring_register_buf_ring(3)</li><li>• [en] io_uring_register_buffers(3)</li><li>• [en] io_uring_register_buffers_sparse(3)</li><li>• [en] io_uring_register_buffers_tags(3)</li><li>• [en] io_uring_register_buffers_update_tag(3)</li><li>• [en] io_uring_register_eventfd(3)</li><li>• [en] io_uring_register_eventfd_async(3)</li><li>• [en] io_uring_register_file_alloc_range(3)</li><li>• [en] io_uring_register_files(3)</li><li>• [en] io_uring_register_files_sparse(3)</li><li>• [en] io_uring_register_files_tags(3)</li><li>• [en] io_uring_register_files_update_tag(3)</li><li>• [en] io_uring_register_iowa_aff(3)</li><li>• [en] io_uring_register_iowa_max_workers(3)</li><li>• [en] io_uring_register_iowa_fd(3)</li><li>• [en] io_uring_register_sync_cancel(3)</li><li>• [en] io_uring_setup(2)</li><li>• [en] io_uring_setup_buf_ring(3)</li><li>• [en] io_uring_sq_ready(3)</li></ul> | <ul style="list-style-type: none"><li>• [en] io_uring_sq_space_left(3)</li><li>• [en] io_uring_sqe_set_data(3)</li><li>• [en] io_uring_sqe_set_data64(3)</li><li>• [en] io_uring_sqe_set_flags(3)</li><li>• [en] io_uring_sqring_wait(3)</li><li>• [en] io_uring_submit(3)</li><li>• [en] io_uring_submit_and_get_events(3)</li><li>• [en] io_uring_submit_and_wait(3)</li><li>• [en] io_uring_submit_timeout(3)</li><li>• [en] io_uring_unregister_buf_ring(3)</li><li>• [en] io_uring_unregister_buffers(3)</li><li>• [en] io_uring_unregister_eventfd(3)</li><li>• [en] io_uring_unregister_files(3)</li><li>• [en] io_uring_unregister_iowa_aff(3)</li><li>• [en] io_uring_unregister_ring_fd(3)</li><li>• [en] io_uring_wait_cqe(3)</li><li>• [en] io_uring_wait_cqe_nr(3)</li><li>• [en] io_uring_wait_cqe_timeout(3)</li><li>• [en] io_uring_wait_cqes(3)</li></ul> |
|--|--|---|--|---|--|--|

- Active research in leveraging io\_uring in DBs, key-value store, etc.
- Applicability beyond storage as the “core” kernel-application interfacing API

# What you should know from this lecture

What is CXL and what key problems does it solve

What is different types of CXL protocols, device types, and generational features

What does flash + CXL allow us to do

What is asynchronous and non-block I/O, and what different APIs support them

What is io\_uring? What are the different operation completion modes it support

What are the performance implications of these modes

**The New(er) Triangle of Storage-Memory Continuum**



# To Conclude

**Storage Research is fundamentally changing and reshaping what kind of systems we can build tomorrow**

- Performance
- Abstractions
- Efficiency
- Programmability
- Cost
- Scalability

This course came out of this report ;)

## **Data Storage Research Vision 2025**

Report on NSF Visioning Workshop held May 30–June 1, 2018

George Amvrosiadis<sup>†</sup>, Ali R. Butt<sup>¶</sup>, Vasily Tarasov<sup>‡</sup>, Erez Zadok<sup>\*</sup>, Ming Zhao<sup>§</sup>

Irfan Ahmad, Remzi H. Arpaci-Dusseau, Feng Chen, Yiran Chen, Yong Chen, Yue Cheng, Vijay Chidambaram, Dilma Da Silva, Angela Demke-Brown, Peter Desnoyers, Jason Flinn, Xubin He, Song Jiang, Geoff Kuenning, Min Li, Carlos Maltzahn, Ethan L. Miller, Kathryn Mohror, Raju Rangaswami, Narasimha Reddy, David Rosenthal, Ali Saman Tosun, Nisha Talagala, Peter Varman, Sudharshan Vazhkudai Avani Waldani, Xiaodong Zhang, Yiyang Zhang, and Mai Zheng.

<sup>†</sup>Carnegie Mellon University, <sup>¶</sup>Virginia Tech, <sup>‡</sup>IBM Research,  
<sup>\*</sup>Stony Brook University, <sup>§</sup>Arizona State University

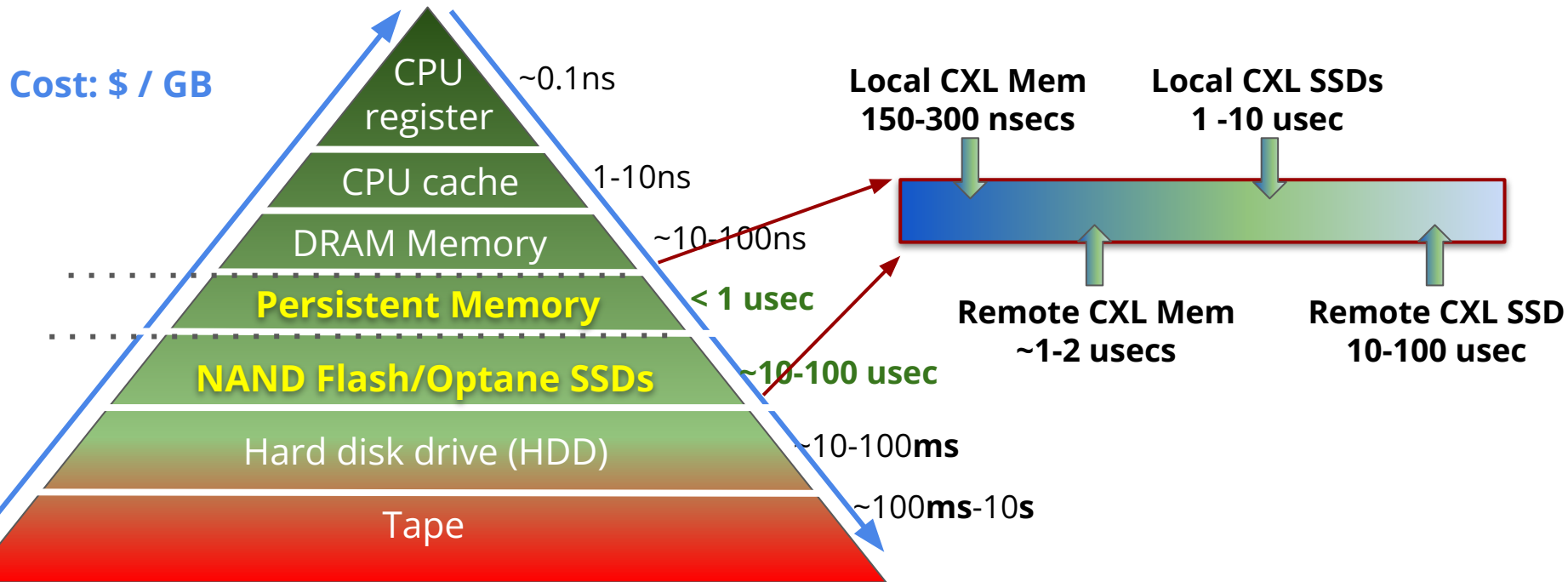
February 2019

### **Executive Summary**

With the emergence of new computing paradigms (e.g., cloud and edge computing, big data, Internet of Things (IoT), deep learning, etc.) and new storage hardware (e.g., non-volatile memory (NVM), shingled-magnetic recording (SMR) disks, and kinetic drives, etc.), a number of open challenges and research issues need to be addressed to ensure sustained storage systems efficacy and performance. The wide variety of applications demand that the fundamental design of storage systems should be revisited to support application-specific and application-defined semantics. Existing standards and abstractions need to be reevaluated; new sustainable data representations need to be designed to support emerging applications. To take advantage of hardware advancements, new storage software designs are also necessary in order to maximize overall system efficiency and performance.

Therefore, there is a urgent need for a consolidated effort to identify and establish a vision for storage systems research and comprehensive techniques that provide practical solutions to the storage issues facing the information technology community. To address this need, the National Science Foundation's (NSF) "Visioning Workshop on Data Storage Research 2025" brought together a number of storage researchers from academia, industry, national laboratories, and federal agencies to develop a collective vision for future storage research, as well as to prioritize

# The New(er) Triangle of Storage-Memory Continuum



Instead of discrete steps, it is a continuous spectrum now: **Continuum**

# Further Reading - CXL (1 or 2)

- CXL Consortium, <https://www.computeexpresslink.org/>
- CXL resources, <https://www.computeexpresslink.org/resource-library>
- Linux CXL driver code: <https://elixir.bootlin.com/linux/latest/source/drivers/cxl>
- Debendra Das Sharma, and others, An Introduction to the Compute Express Link (CXL) Interconnect, **2023**, <https://arxiv.org/abs/2306.11227>
- Hasan Al Maruf, and others. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In Proceedings of the 28th ACM ASPLOS **2023**. <https://doi.org/10.1145/3582016.3582063>
- Myoungsoo Jung. **2022**. Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In Proceedings of the 14th ACM HotStorage '22, <https://doi.org/10.1145/3538643.3539745>
- Miryeong Kwon, Sangwon Lee, and Myoungsoo Jung. 2023. Cache in Hand: Expander-Driven CXL Prefetcher for Next Generation CXL-SSD. In Proceedings of the 15th ACM HotStorage '23, <https://doi.org/10.1145/3599691.3603406>
- Huaicheng Li, and others. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In Proceedings of the 28th ACM ASPLOS 2023, <https://doi.org/10.1145/3575693.3578835>
- Shao-Peng Yang and others. Overcoming the Memory Wall with CXL-Enabled SSDs, USENIX ATC **2023**, <https://www.usenix.org/conference/atc23/presentation/yang-shao-peng>
- Donghyun Gouk and others, Direct Access, High-Performance Memory Disaggregation with DirectCXL, USENIX ATC **2022**, <https://www.usenix.org/conference/atc22/presentation/gouk>

# Further Reading - CXL (2 of 2)

- CXL-ANNS: Software-Hardware Collaborative Memory Disaggregation and Computation for Billion-Scale Approximate Nearest Neighbor Search, USENIX ATC 2023, <https://www.usenix.org/conference/atc23/presentation/jiang>
- Marcos K. Aguilera, and others. 2023. Memory disaggregation: why now and what are the challenges. SIGOPS Oper. Syst. Rev. 57, 1 (June **2023**), 38–46. <https://doi.org/10.1145/3606557.3606563>
- Hasan Al Maruf and Mosharaf Chowdhury. 2023. Memory Disaggregation: Advances and Open Challenges. SIGOPS Oper. Syst. Rev. 57, 1 (June **2023**), 29–37. <https://doi.org/10.1145/3606557.3606562>
- Jianguo Wang and Qizhen Zhang. **2023**. Disaggregated Database Systems. In Companion of the **2023** International Conference on Management of Data (SIGMOD '23). <https://doi.org/10.1145/3555041.3589403>
- Wenjing Jin, and others. DRAM Translation Layer: Software-Transparent DRAM Power Savings for Disaggregated Memory. In Proceedings of the 50th Annual International Symposium on Computer Architecture (**ISCA '23**). <https://doi.org/10.1145/3579371.3589051>
- What's the Difference Between CXL 1.1 and CXL 2.0? <https://www.electronicdesign.com/technologies/embedded/article/21249351/cxl-consortium-whats-the-difference-between-cxl-11-and-cxl-20>
- QEMU CXL setup, <https://www.qemu.org/docs/master/system/devices/cxl.html>
- How To Map a CXL Endpoint to a CPU Socket in Linux, <https://stevescargall.com/blog/2022/12/27/how-to-map-a-cxl-endpoint-to-a-cpu-socket-in-linux/>

# Further Reading - io\_uring (1 of 2)

- Efficient IO with io\_uring, [https://kernel.dk/io\\_uring.pdf](https://kernel.dk/io_uring.pdf)
- What's new with io\_uring, <https://kernel.dk/axboe-kr2022.pdf>
- An Introduction to the io\_uring Asynchronous I/O Framework, <https://blogs.oracle.com/linux/post/an-introduction-to-the-io-uring-asynchronous-io-framework>
- Zebin Ren and Animesh Trivedi. 2023. Performance Characterization of Modern Storage Stacks: POSIX I/O, libaio, SPDK, and io\_uring. In Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems (CHEOPS '23). Association for Computing Machinery, New York, NY, USA, 35–45. <https://doi.org/10.1145/3578353.3589545>
- Diego Didona, Jonas Pfefferle, Nikolas Ioannou, Bernard Metzler, and Animesh Trivedi. 2022. Understanding modern storage APIs: a systematic study of libaio, SPDK, and io\_uring. In Proceedings of the 15th ACM International Conference on Systems and Storage (SYSTOR '22). Association for Computing Machinery, New York, NY, USA, 120–127. <https://doi.org/10.1145/3534056.3534945>
- Simon A. F. Lund, Philippe Bonnet, Klaus B. A. Jensen, and Javier Gonzalez. 2022. I/O interface independence with xNVMe. In Proceedings of the 15th ACM International Conference on Systems and Storage (SYSTOR '22). Association for Computing Machinery, New York, NY, USA, 108–119. <https://doi.org/10.1145/3534056.3534936>
- Sidharth Sundar, William Simpson, Jacob Higdon, Caeden Whitaker, Bryan Harris, and Nihat Altiparmak. 2023. Energy Implications of IO Interface Design Choices. In Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '23). Association for Computing Machinery, New York, NY, USA, 58–64. <https://doi.org/10.1145/3599691.3603411>

# Further Reading - io\_uring (2 of 2)

- Ringing in a new asynchronous I/O API, <https://lwn.net/Articles/776703/>
- [PATCHSET v5] io\_uring IO interface, <https://lore.kernel.org/linux-block/20190116175003.17880-1-axboe@kernel.dk/>
- Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, and How to Exploit it: High-Performance I/O for High-Performance Storage Engines. Proc. VLDB Endow. 16, 9 (May 2023), 2090–2102. <https://doi.org/10.14778/3598581.3598584>
- Hugh C. Lauer and Roger M. Needham. 1979. On the duality of operating system structures. SIGOPS Oper. Syst. Rev. 13, 2 (April 1979), 3–19. <https://doi.org/10.1145/850657.850658>
- John Ousterhout, Why Threads Are A Bad Idea (for most purposes), <https://web.stanford.edu/~ouster/cgi-bin/papers/threads.pdf>
- Rob von Behren, Jeremy Condit, and Eric Brewer. 2003. Why events are a bad idea (for high-concurrency servers). In Proceedings of the 9th conference on Hot Topics in Operating Systems - Volume 9 (HOTOS'03). USENIX Association, USA, 4. <https://dl.acm.org/doi/10.5555/1251054.1251058>
- Philipp Haller, Martin Odersky, Scala Actors: Unifying thread-based and event-based programming, 2008, <https://doi.org/10.1016/j.tcs.2008.09.019>.
- A 5 part series on the asynchronous nature of I/O, OS, and concurrency: <https://blog.acolyer.org/2014/12/08/on-the-duality-of-operating-system-structures/>
- µTune: Auto-Tuned Threading for OLDI Microservices, <https://www.usenix.org/conference/osdi18/presentation/sriraman>
- Linux Asynchronous I/O, <https://oxnz.github.io/2016/10/13/linux-aio/>
- Linux-aio, <https://github.com/littledan/linux-aio>