

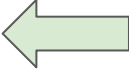
Storage Systems (StoSys)

XM_0092

Lecture 2: Host Interfacing and Software implications

Animesh Trivedi
Autumn 2023, Period 1

Syllabus outline

- ~~1. Welcome and introduction to NVM (today)~~
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

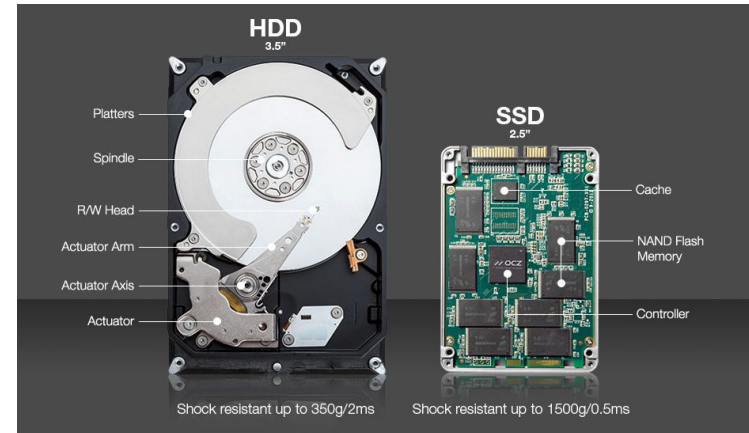
Ramifications of fast flash storage

First flash storage devices in mainstream computing showed up in mid-2000s (2005-2006-2007)

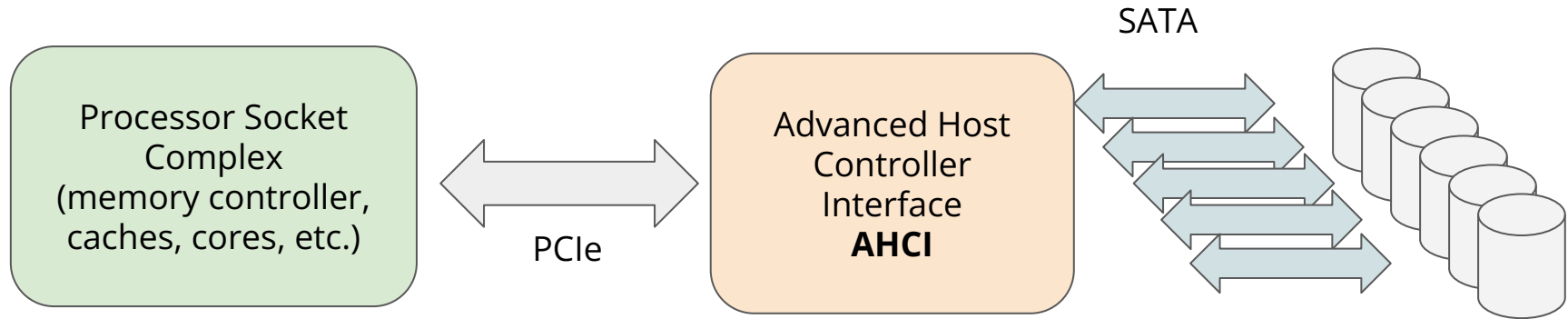
- Recall that flash NAND/NOR are already used extensively in embedded systems, ROM/BIOS, etc.

Typically (and often) a new technology is packed behind a known systems interface

First generation of flash devices were packaged as a fast “HDD” running with compatible SAS/SATA HDD protocols for data transfers



Classical HDD setup: AHCI

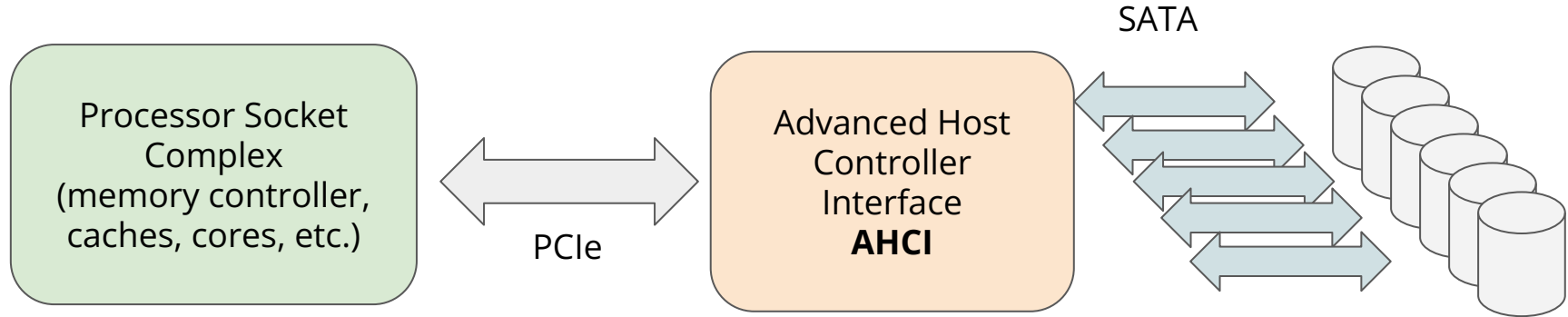


AHCI is implemented in motherboards

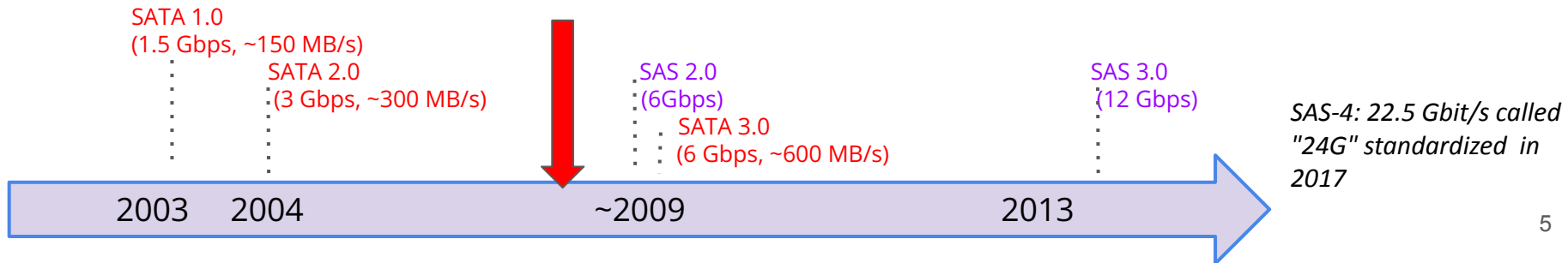
Single point to implement the **protocol translation** between PCIe and SATA

Support multiple devices types (HDD, optical drives, floppy drives?)

AHCI challenges



Single bottleneck for performance, SATA speeds (or SAS) were just not fast enough

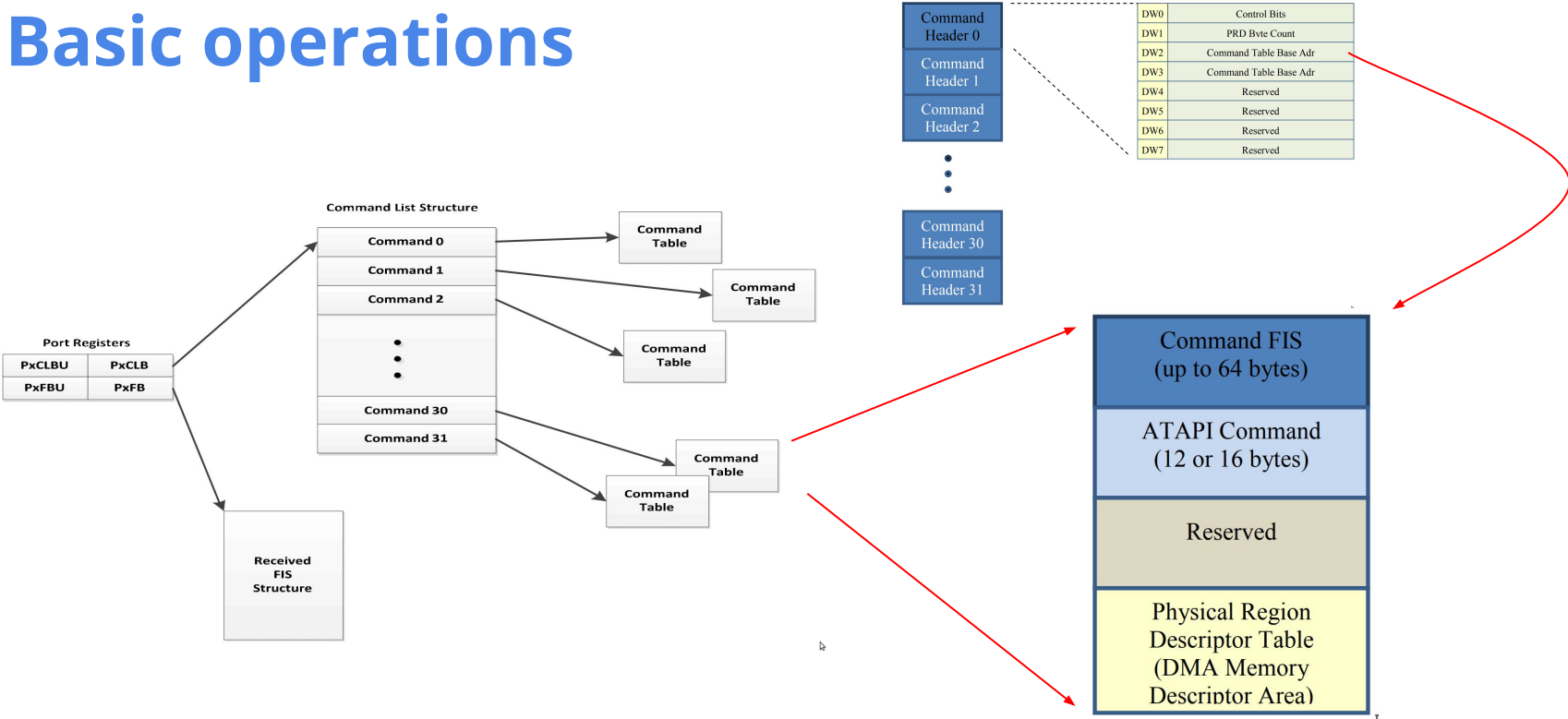


Challenges with the storage protocols

Beyond just the hardware speeds

- SAS/SATA protocols were too slow to evolve, took multi years for one standard to get to the next
 - See the time between different version 4-5 years, it has improved later
- The AHCI centralized complex became (**hardware**) performance bottleneck as I/O requests have an intermediary stop
 - Low latency
 - High IOPS
 - Complexity of implementation and revision with new generation of flash drives
 - Could take up to **6 microseconds** on the wire

Basic operations



Linux storage software stack components

Applications

VFS

ext4

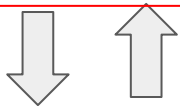
XFS

Btrfs

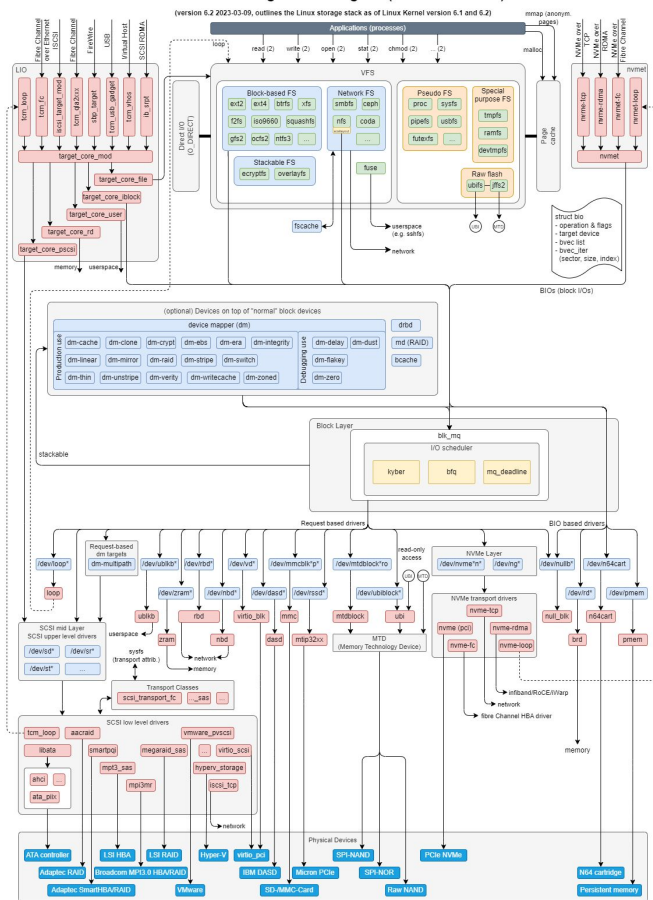
Block Layer

SCSI mid-layer

SCSI low-level drivers



The Linux Storage Stack Diagram (Linux Kernel 6.2)



https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram

High Performance Solid State Storage Under Linux, <https://storageconference.us/2010/Papers/MSST/Seppanen.pdf>, MSST 2010

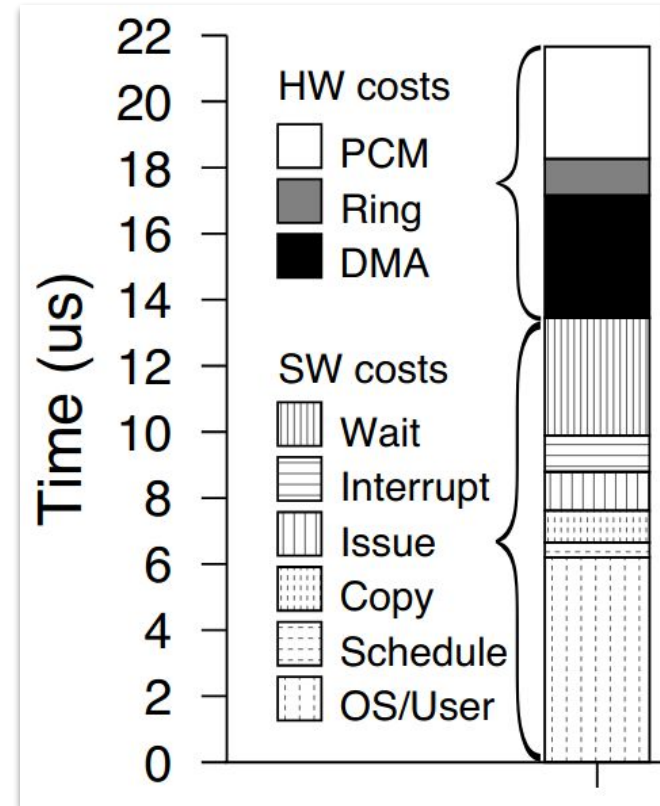
High software overheads

(2010) timeframe

- It takes ~20,000 instructions to issue and complete a 4kB I/O request in Linux
- Total time = 23 μ sec

In a setup with an experimental device (Moneta)

- ~2 μ sec is for the storage hardware, PCM
- 13 μ sec out of 23 μ sec is software overhead (62%)
- For RAID-HDD this is less than 1%



To summarize

2008-2009-2010 timeframe

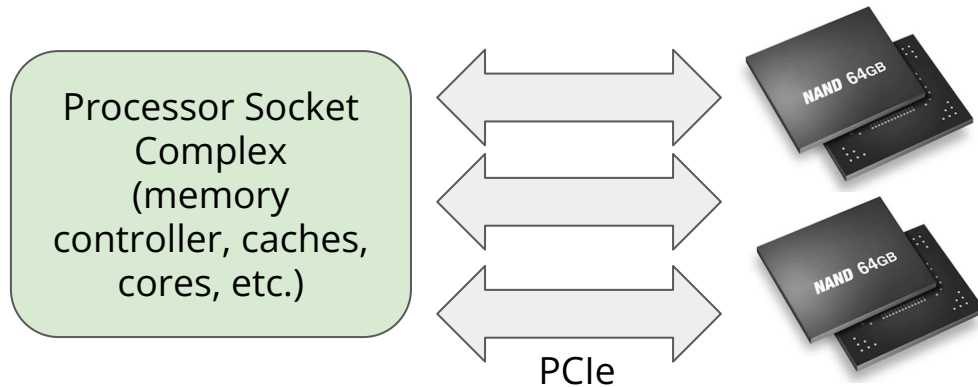
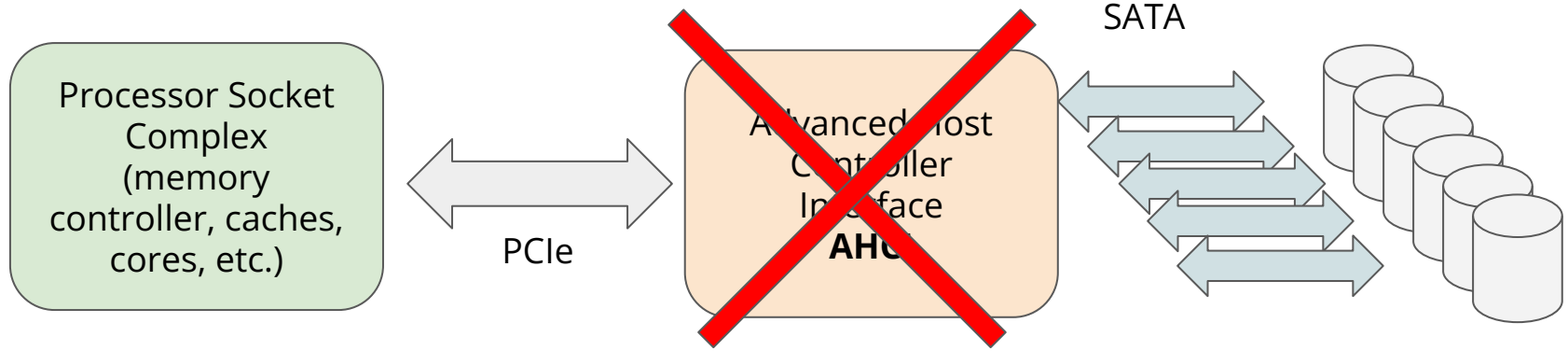
- Fast high-bandwidth flash SSDs were hitting the market

However, their performance was bottlenecked by the

- **Hardware overheads:** HDD-oriented AHCI interfaces and SAS/SATA protocols
- **Software overheads:** HDD-oriented software design decisions made for slow storage devices (i.e., HDDs) that needed revision

A radically new way of integrating new emerging storage was needed....

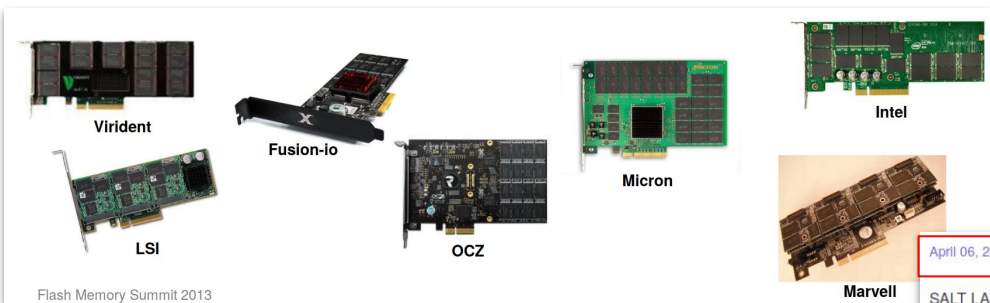
Connect NVM storage directly to the PCIe bus



Attach them directly on the PCIe bus, why?

- No HBA, directly to the CPU
- Scalable port width (1-16x lanes)
- High bandwidth/lane (~500MB/lane for v2.0, today 4.0 has 2GB/lane)
- Standard bus, supported by all
- Large configuration/data space
- Power efficient ...

Multiple competitive standards and proprietary solutions



ONE MILLION RAND IOPS (r/w mix)

April 06, 2009 12:47 PM Eastern Daylight Time

SALT LAKE CITY--(BUSINESS WIRE)--First graph, second sentence of release should read: built a system using five 320GB ioDrive Duos and six 160GB ioDrives (sted using five 320MB ioDrive Duos and six 160MB ioDrives.)

"Quad-Core AMD Opteron processors help drive efficiencies and reduce complexities with innovations that enable superior performance"

Tweet this

The corrected release reads:

FUSION-IO BREAKS STORAGE PERFORMANCE BARRIERS, EXCEEDING 1 MILLION IOPS AND 8 GB/S THROUGHPUT WITHIN A SINGLE HP PROLIANT SERVER

Fusion-io, the leader in application-centric, solid-state architecture and high-performance I/O solutions, working with HP, the world's largest technology company, today announced that they exceeded an astonishing 1 million IOPS (I/O Operations Per Second) and eight gigabytes per second (GB/s) sustained throughput using a single HP ProLiant server.

Working together in HP's ProLiant engineering labs in Houston, technologists from HP and Fusion-io built a system using five 320GB ioDrive Duos and six 160GB ioDrives in a single HP ProLiant DL785 G5 server, running with four Quad-Core AMD Opteron™ processors. This standard configuration allowed the engineers to reach an unprecedented eight GB/s sustained throughput, making it possible to achieve 1,009,384 IOPS using 2KB random 70/30 read/write mix, as measured using the fio benchmark.

Image yourself in 2009

- you spent all your life optimizing to squeeze out a bit of performance from storage
- A HDD does ~100s of random IOPS
- You can pack ~30s of them in a single server
 - ~a few thousand IOPS
- *Then comes Fusion IO ...*

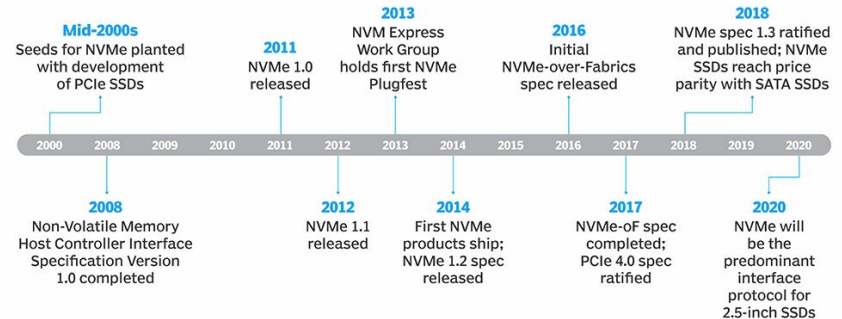
Emergence of NVM Express

NVM Express is a protocol specification regarding how host software communicates with NVM storage across the PCI Express (PCIe) bus

- A set of command and response
- Designed for high performance, highly parallel PCIe NVM storage devices
- Has scope to define lots of control commands for device management
 - FTL, firmware, temperate, errors, etc.

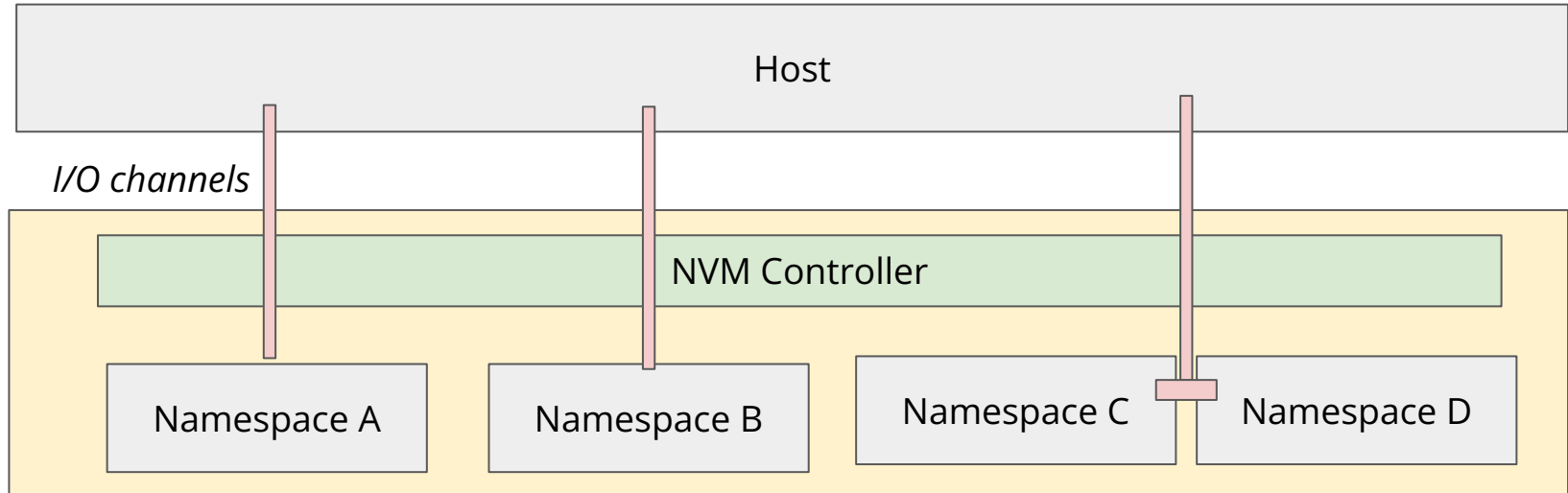


NVMe milestones



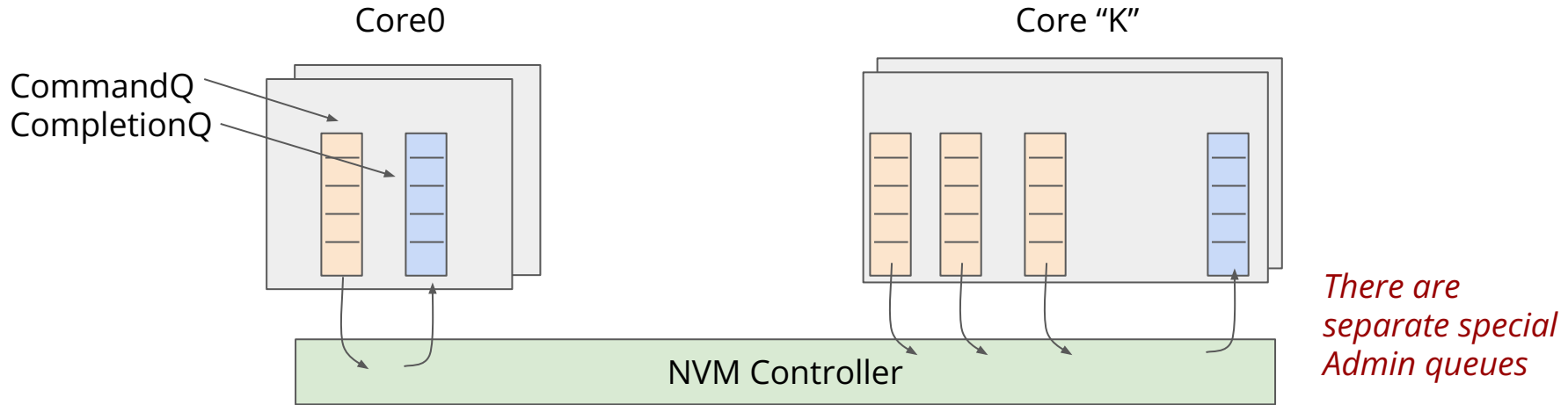
NVMe ideas - namespaces

Key challenge: how to exploit parallelism inside the device



Multiple independent partition of a device (block range start:end)
Independent I/O channels can be created in namespaces

NVMe ideas - Command/Completion queues



A command queue and completion queue based structure

- Small commands - 64 Bytes (no legacy stuff). Initially, 10 admin commands, and 3 I/O commands (r/w/f)
- 16 bytes completion structure
- 64K queues, 64K deep, outstanding requests
- A large number of interrupt mapping possible
- Any possible mapping of command:completion queue possible based on the architecture

In comparison, AHCI vs. NVMe

1. Aggregation vs Point-to-Point Architecture

- a. AHCI is a single point of aggregation vs NVMe has point-to-point PCIe lanes
- b. Helps with high bandwidth and scalable performance

2. Opportunities to exploit device parallelism

- a. HDDs are slow, and queuing up inside the device does not help much
- b. NVM SSDs are fast, and have lots of parallel parts, inside device queuing helps
- c. (Max) 64K queue, 64K deep (vs AHCI 32 ports/queues, 32 deep)
- d. Support for multiple interrupts (NVMe) vs single interrupt to AHCI

3. Lightweight device interaction and streamlined shared data structures

- a. **9 device registers** read/write (AHCI) vs **2 device register** (NVMe) read/writes for a command completion
- b. Possibility to amortize command issuing over multiple commands in one dispatch

4. Possibility to multipath and networking over PCIe (Ethernet)

Early prototyping (Chatham NVMe prototype)

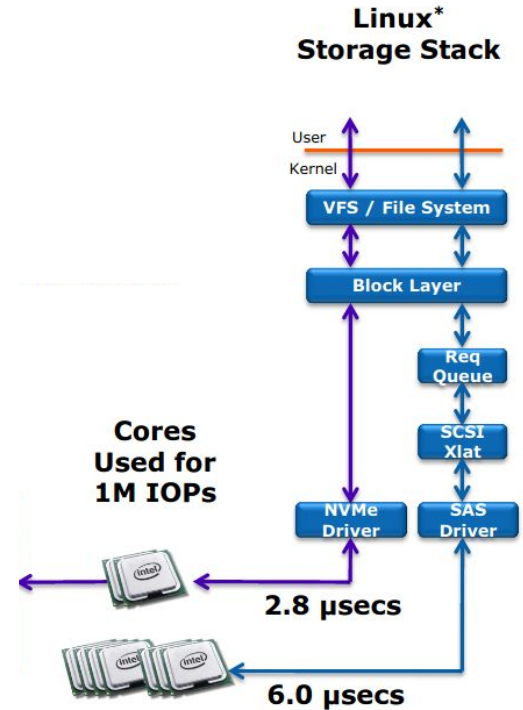
Chatham NVMe Prototype



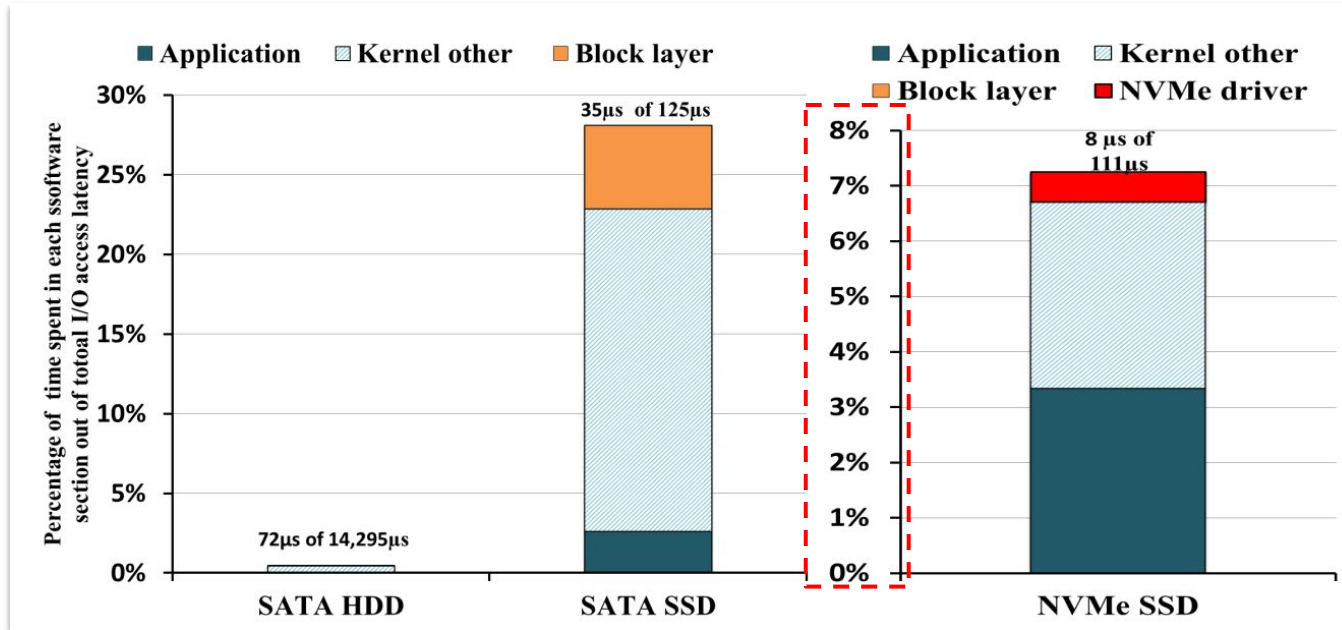
NVMe reduces latency overhead by more than 50%

- SCSI/SAS : 6.0 μ s 19,500 cycles
- NVMe : 2.8 μ s 9,100 cycles

PCIe removes the **hardware/link overheads**
NVMe removes the **protocol overheads**

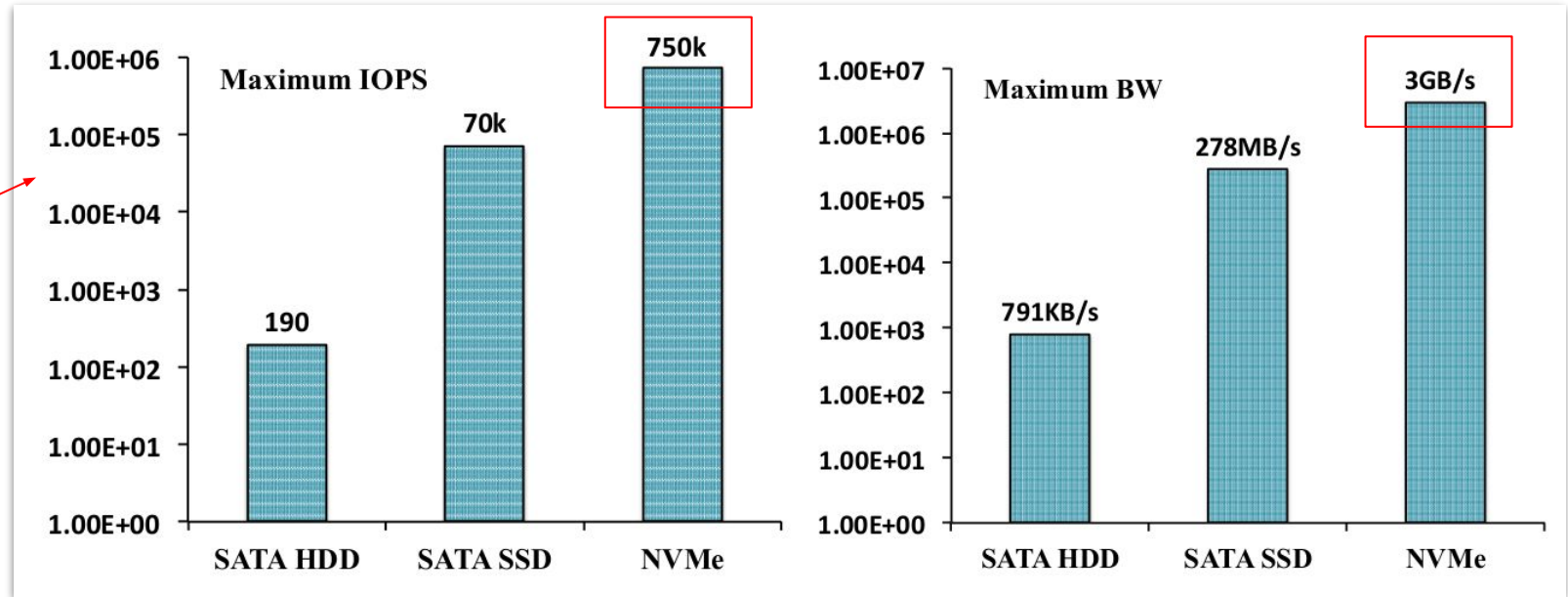


NVM Express : Reduces software overheads



- From HDD to SATA SSD the relative overhead of software is increased from 0.5% to 28%
- NVMe reduces the relative software overhead to ~7%

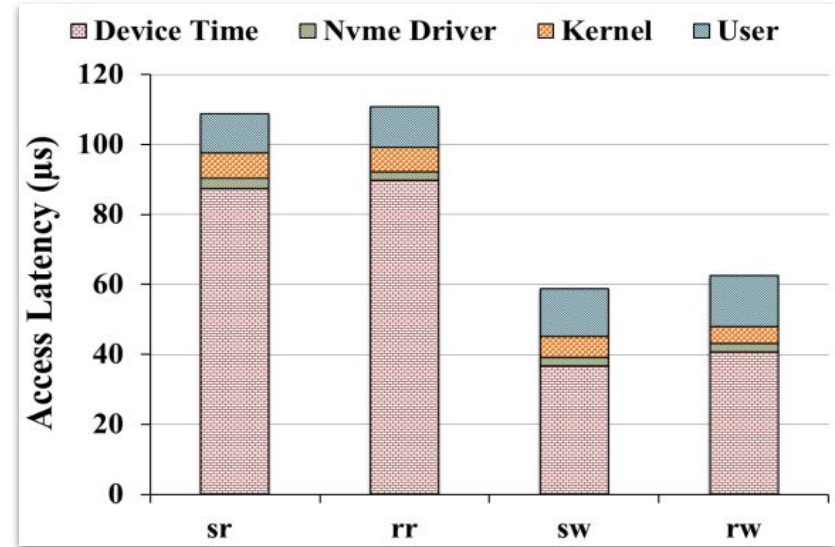
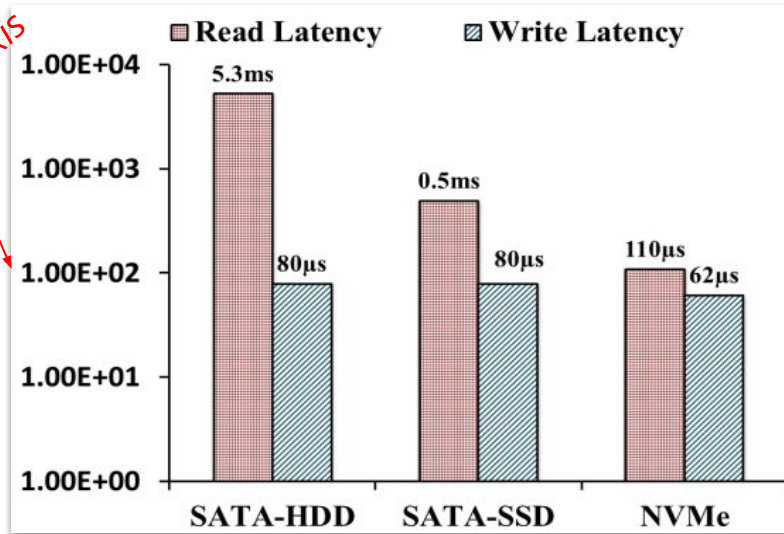
NVM Express: Improves IOPS and bandwidth



- 750K IOPS in a single NVMe device!
- 3 GB/sec bandwidth (bounded by the PCIe links)

NVM Express: I/O latencies

Log y axis



1-3 orders of magnitude better read performance (in comparison to SATA SSD and HDD)
Similar random and sequential read/write latencies (we will see later, one is better than the other)

Today: NVM Express

One of the most popular and de-facto standard for high-performance NVM storage devices

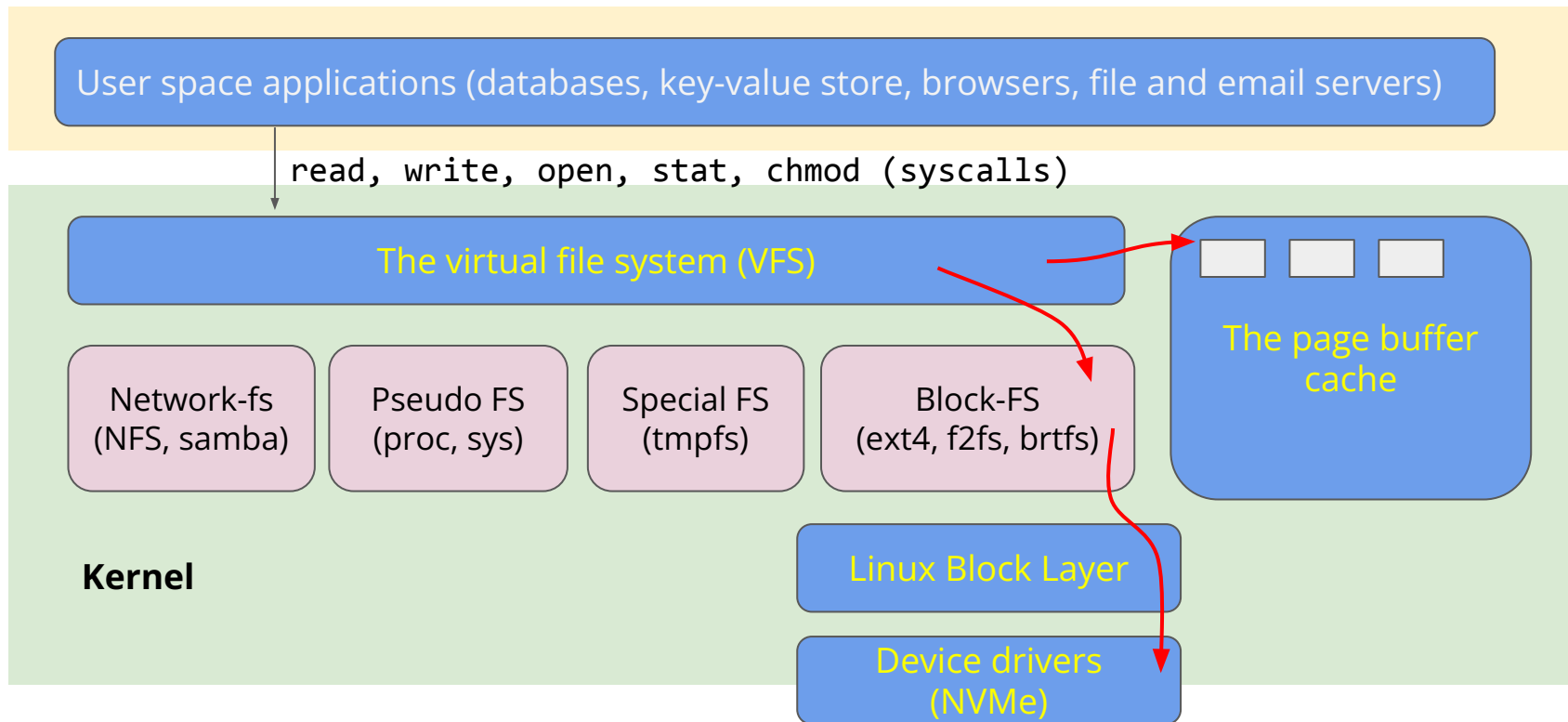
A comprehensive set of control, data command set, semantics, and response

Constantly being updated to include the demands from the industries and input from academia

In the project, you talk to a NVMe device using the NVMe command set (ZNS command set uses the NVMe command set)



The Linux storage stack - software (simplified)



Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems (2013)

Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems

Matias Bjorling¹

Jens Axboe¹

David Nellans¹

Philippe Bonnet²

¹IT University of Copenhagen
{mabj,phbo}@itu.dk

¹Fusion-io
{jaxboe,dnellans}@fusionio.com

ABSTRACT

The IO performance of storage devices has accelerated from hundreds of IOPS five years ago, to hundreds of thousands of IOPS today, and tens of millions of IOPS projected in five years. This sharp evolution is primarily due to the introduction of NAND-flash devices and their data parallel design. In this work, we demonstrate that the block layer within the operating system, originally designed to handle thousands of IOPS, has become a bottleneck to overall storage system performance, specially on the high NUMA-factor processors systems that are becoming commonplace. We describe the design of a next generation block layer that is capable of handling tens of millions of IOPS on a multi-core system equipped with a single storage device. Our experiments show that our design scales gracefully with the number of cores, even on NUMA systems with multiple sockets.

Categories and Subject Descriptors

D.4.2 [Operating System]: Storage Management—Secondary storage; D.4.8 [Operating System]: Performance—measurements

General Terms

Design, Experimentation, Measurement, Performance.

Keywords

Linux, Block Layer, Solid State Drives, Non-volatile Memory, Latency, Throughput.

1 Introduction

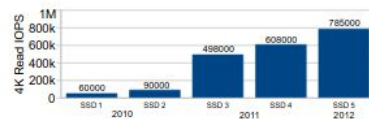


Figure 1: IOPS for 4K random read for five SSD devices.

(e.g., flash or phase-change memory [11, 6]) is transforming the performance characteristics of secondary storage. SSDs often exhibit little latency difference between sequential and random IOs [16]. IO latency for SSDs is in the order of tens of microseconds as opposed to tens of milliseconds for HDDs. Large internal data parallelism in SSDs disks enables many concurrent IO operations which, in turn, allows single devices to achieve close to a million IOs per second (IOPS) for random accesses, as opposed to just hundreds on traditional magnetic hard drives. In Figure 1, we illustrate the evolution of SSD performance over the last couple of years.

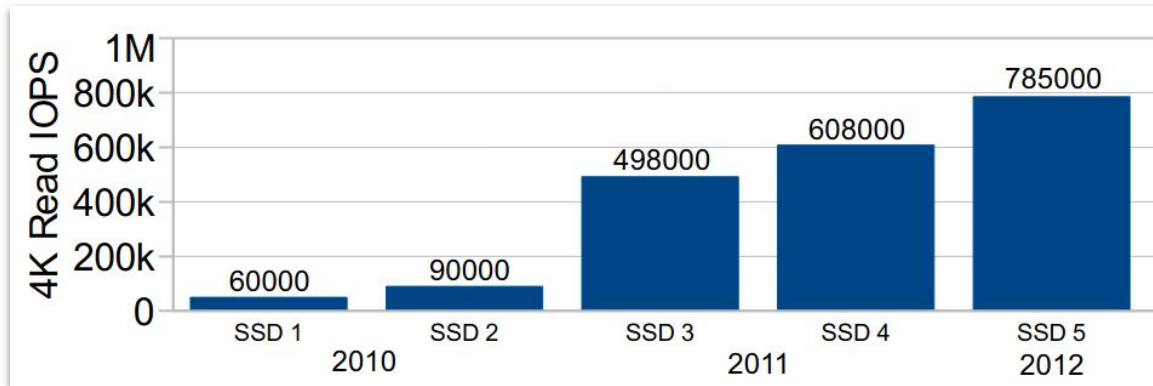
A similar, albeit slower, performance transformation has already been witnessed for network systems. Ethernet speed evolved steadily from 10 Mb/s in the early 1990s to 100 Gb/s in 2010. Such a regular evolution over a 20 years period has allowed for a smooth transition between lab prototypes and mainstream deployments over time. For storage, the rate of change is much faster. We have seen a 10,000x improvement

Key challenges

In the early 2010, lots of hardware/protocol optimizations were happening

Two trends were evident

1. Performance of NVMe SSD (i.e., flash) was improving rapidly
2. Single CPU performance was stalled
 - a. Most gains came from multi-core / multi-socket systems



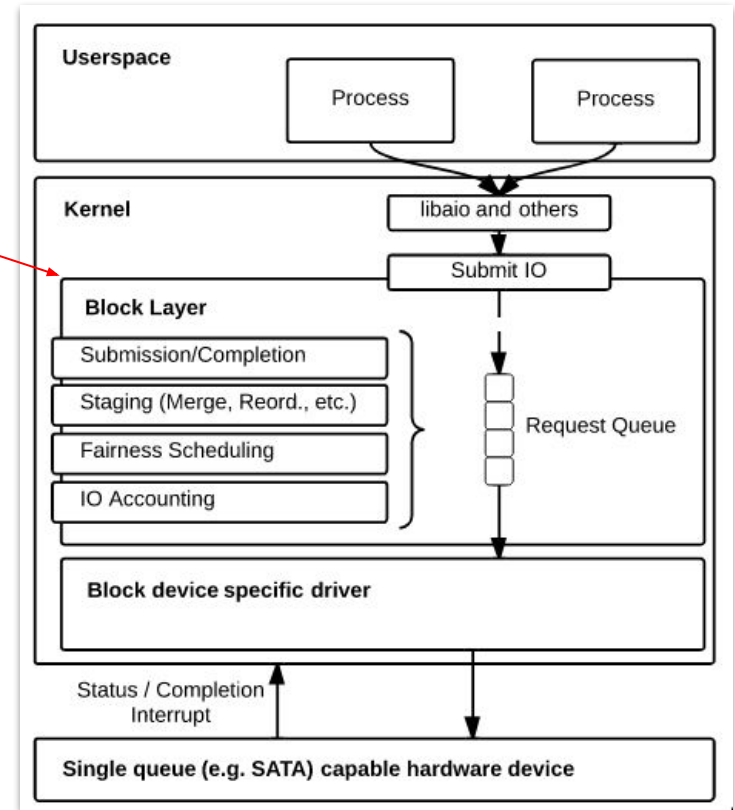
The Linux I/O stack

The Linux Block Layer

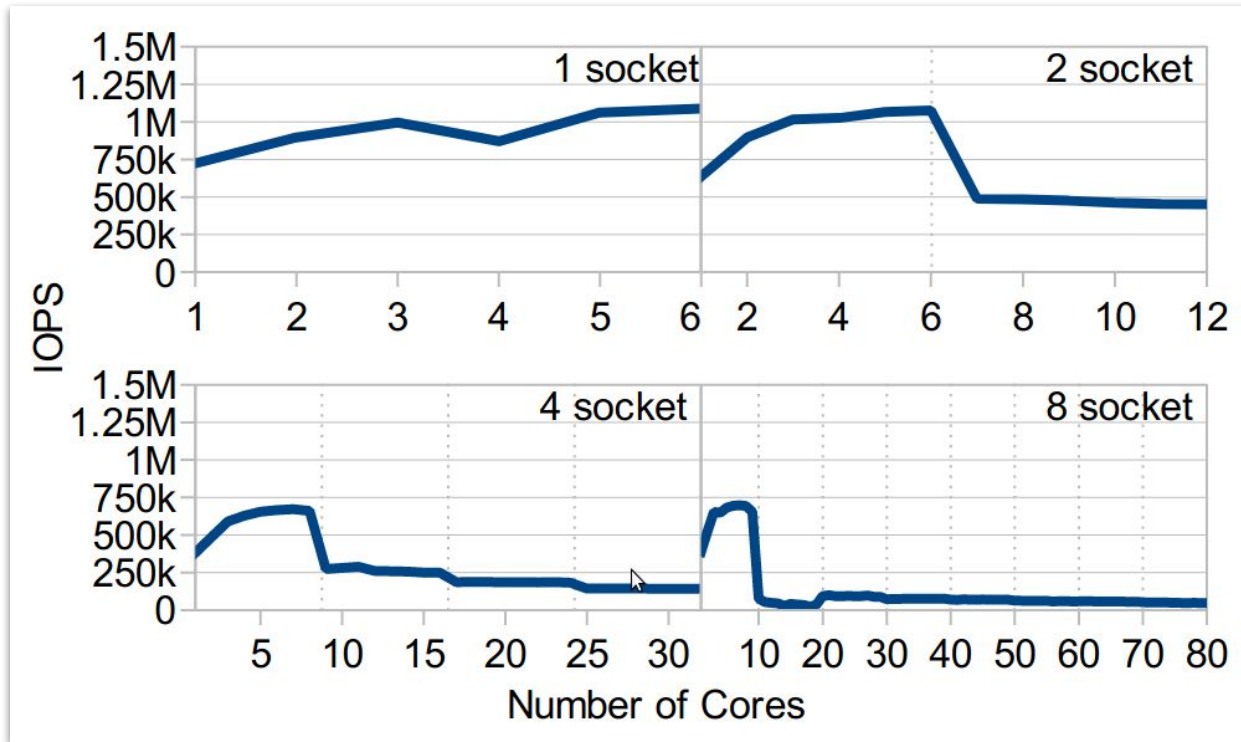
- Unified interface to application and device drivers
- Provides many common services like IO buffering, scheduling, fairness, accounting, error handling, etc.

An essential part of the storage I/O

Can the Linux Block I/O layer scale on multi-core machines to match the SSD performance?



Performance evaluation of the block I/O



Performance collapses as the number of cores / socket increases (*guesses?*)

Key reasons

1. Request queue locking

the single request queue becomes the single point of contention in multi core machine

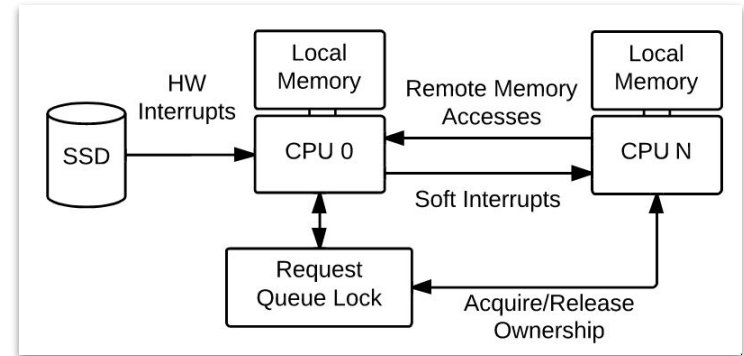
Can you think of why there **was** a single queue design?

2. Hardware interrupts

driver/stack was not ready to distribute load generated by interrupts on the `cpu0`

3. Remote Memory Access

Cross socket memory access to issue and complete a request, poor performance



Key proposals

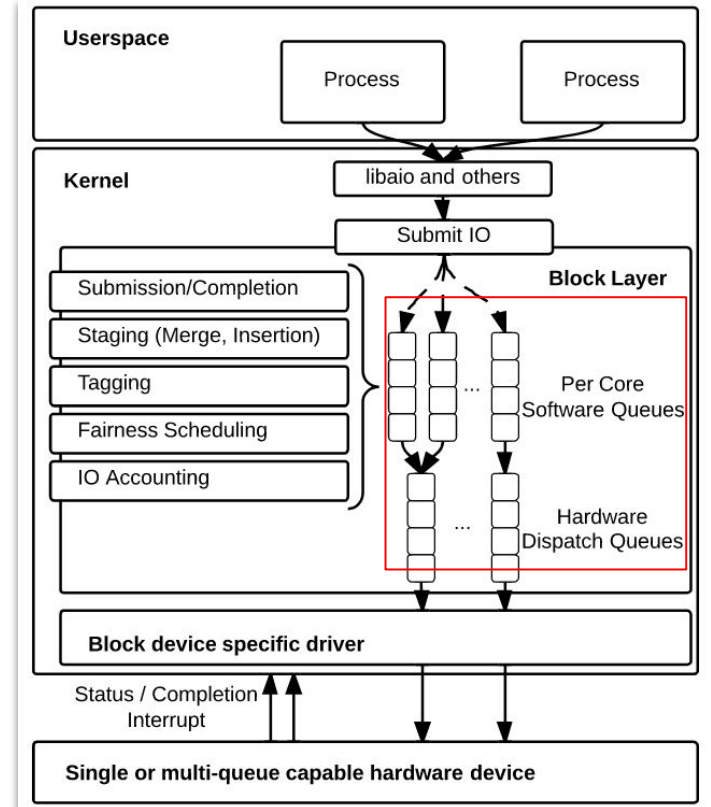
A two stage split **multi-queue** interface
(separation of concerns design principle)

Software Staging Queues

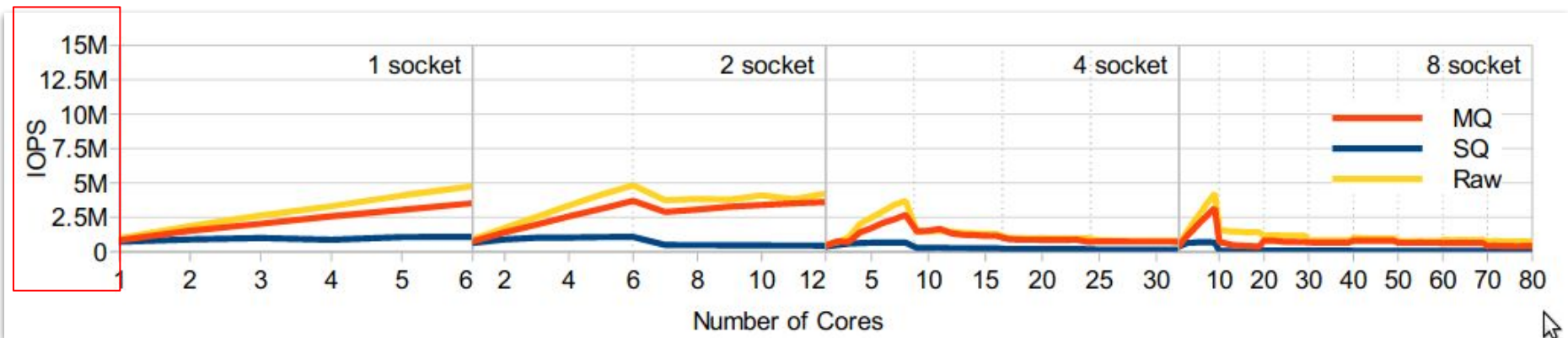
- Local to each core/socket → reduces contention and NUMA memory accesses
- Hooks to provide OS/software services
- Software manipulation does not need to sync between cores

Hardware Dispatch Queues

- Any number of queues supported by the device
- Use the queues close to the CPU core
- Helps to use and distribute interrupts
 - Interrupts/queue (simplified)



Significant baseline improvements



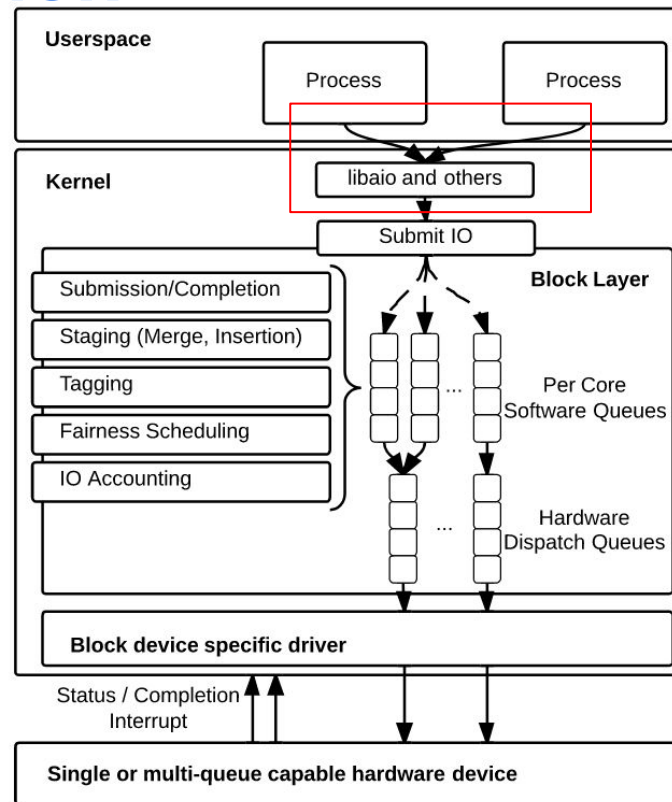
- General improvements across the spectrum (but still)
 - Raw is performance when the block layer is skipped
- Second socket leads to fall in performance for Single Queue (SQ)
 - Coherence and locking
- Multiqueue (MQ) follows the performance of raw closely

What else can we optimize here?

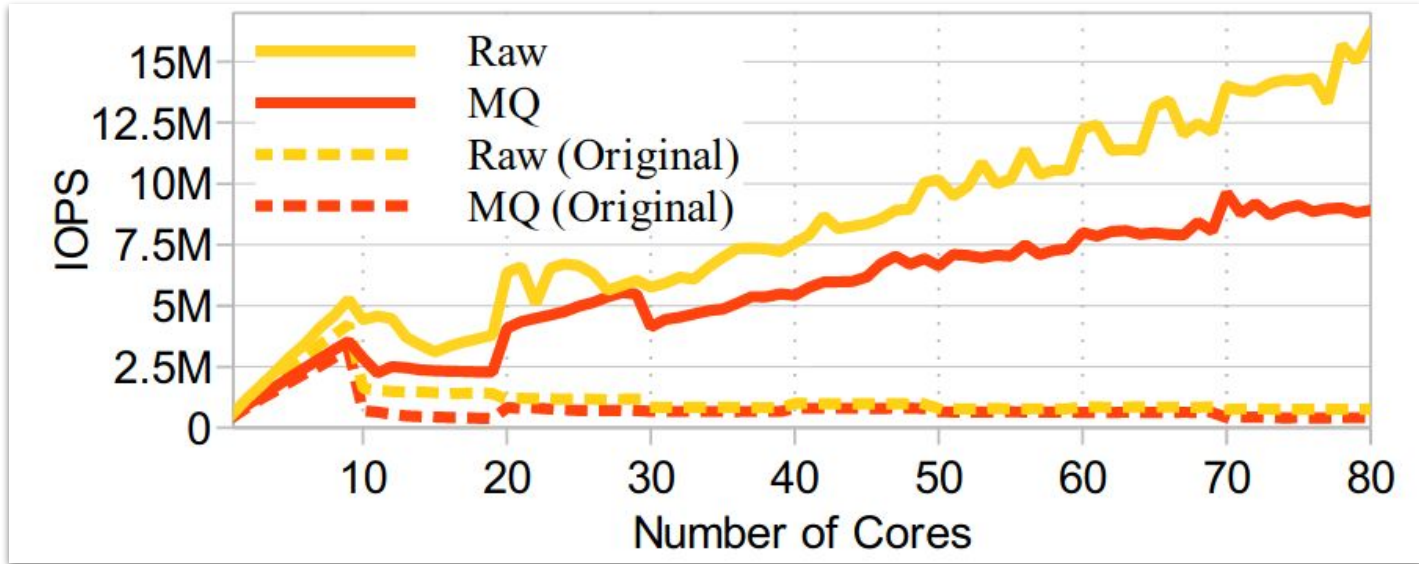
Application-level I/O submission

Optimizations in the application-level API and implementation (libaio)

- A global context list lock
 - Replaced by lockless list with CAS instructions
- A completion ring based notification to the user thread
 - Remove it
- Various shared variables throughout the stack
 - Reimplement them with per-core variables and CAS instructions



As a result ...



Managed to push IOPS close to 15 million IOPS

More importantly, made the Linux block layer scalable and ready for the future NVMe devices

When Poll is Better than Interrupt (2012)

When Poll is Better than Interrupt

Jisoo Yang

Dave B. Mintum

Frank Hady

{jisoo.yang | dave.b.minturn | frank.hady} (at) intel.com

Intel Corporation

Abstract

In a traditional block I/O path, the operating system completes virtually all I/Os asynchronously via interrupts. However, performing storage I/O with ultra-low latency devices using next-generation non-volatile memory, it can be shown that polling for the completion – hence wasting clock cycles during the I/O – delivers higher performance than traditional interrupt-driven I/O. This paper thus argues for the *synchronous completion* of block I/O first by presenting strong empirical evidence showing a stack latency advantage, second by delineating limits with the current interrupt-driven path, and third by proving that synchronous completion is indeed safe and correct. This paper further discusses challenges and opportunities introduced by synchronous I/O completion model for both operating system kernels and user applications.

1 Introduction

When an operating system kernel processes a block storage I/O request, the kernel usually submits and completes the I/O request asynchronously, releasing the CPU to perform other tasks while the hardware device completes the storage operation. In addition to the CPU

pass the kernel's heavyweight asynchronous block I/O subsystem, reducing CPU clock cycles needed to process I/Os. However, a necessary condition is that the CPU has to spin-wait for the completion from the device, increasing the cycles used.

Using a prototype DRAM-based storage device to mimic the potential performance of a very fast next-generation SSD, we verified that the synchronous model completes an individual I/O faster and consumes less CPU clock cycles despite having to poll. The device is fast enough that the spinning time is smaller than the overhead of the asynchronous I/O completion model.

Interrupt-driven asynchronous completion introduces additional performance issues when used with very fast SSDs such as our prototype. Asynchronous completion may suffer from lower I/O rates even when scaled to many outstanding I/Os across many threads. We empirically confirmed this with Linux,* and examine the system overheads of interrupt handling, cache pollution, CPU power-state transitions associated with the asynchronous model.

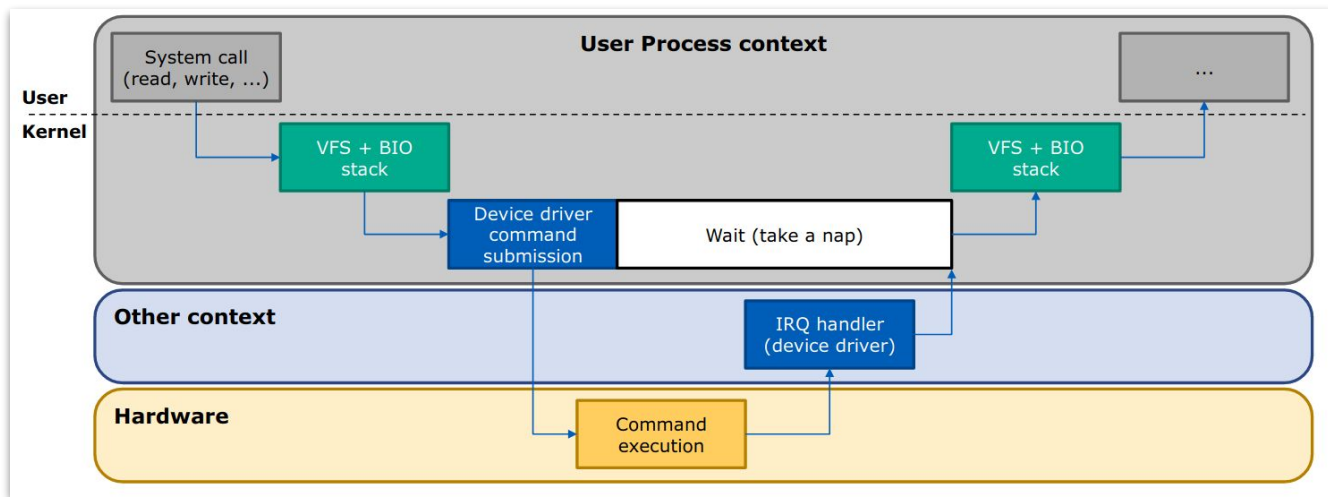
We also demonstrate that the synchronous completion model is correct and simple with respect to maintaining I/O ordering when used with application interfaces such

Any guesses, why?
Faster may be, but better? How?

The classic way of doing an I/O operation

Asynchronous I/O (*it is a loaded term*)

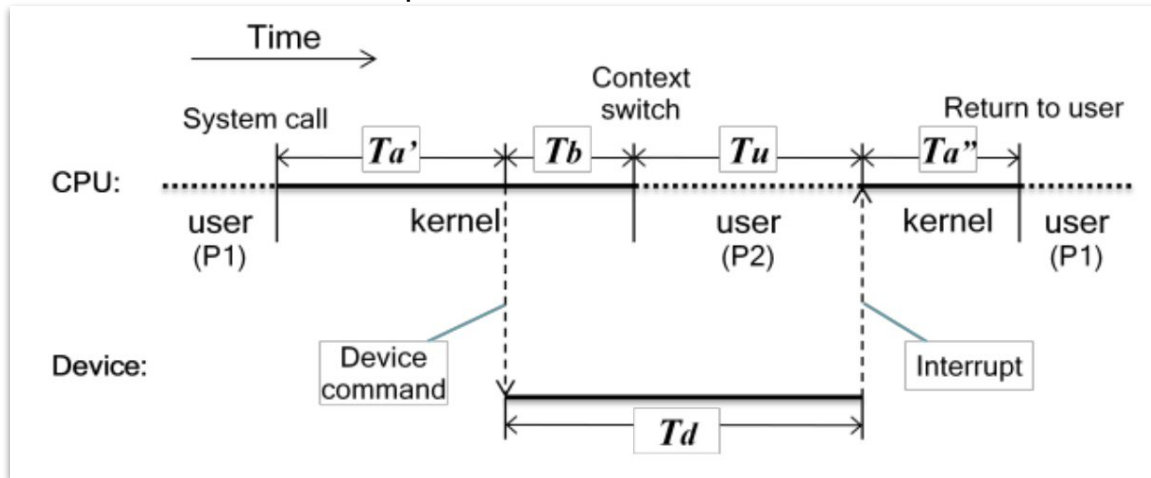
- Software issues a request
- And then switches to something else
- (At some point in future) Request completes and there is an interrupt for notification



The classic way of doing an I/O operation

Asynchronous I/O (*it is a loaded term*)

- Software issues a request
- And then switches to something else
- (At some point in future) Request completes and there is an interrupt for notification



For 4kB transfer, $T_a = T_{a'} + T_{a''} \sim 4.8 \text{ usec}$, $T_d = 4.1 \text{ usec}$, $T_b \text{ (sched)} = 1.4 \text{ usec}$, $T_u = 2.7 \text{ usec}$

Challenges with the classical way

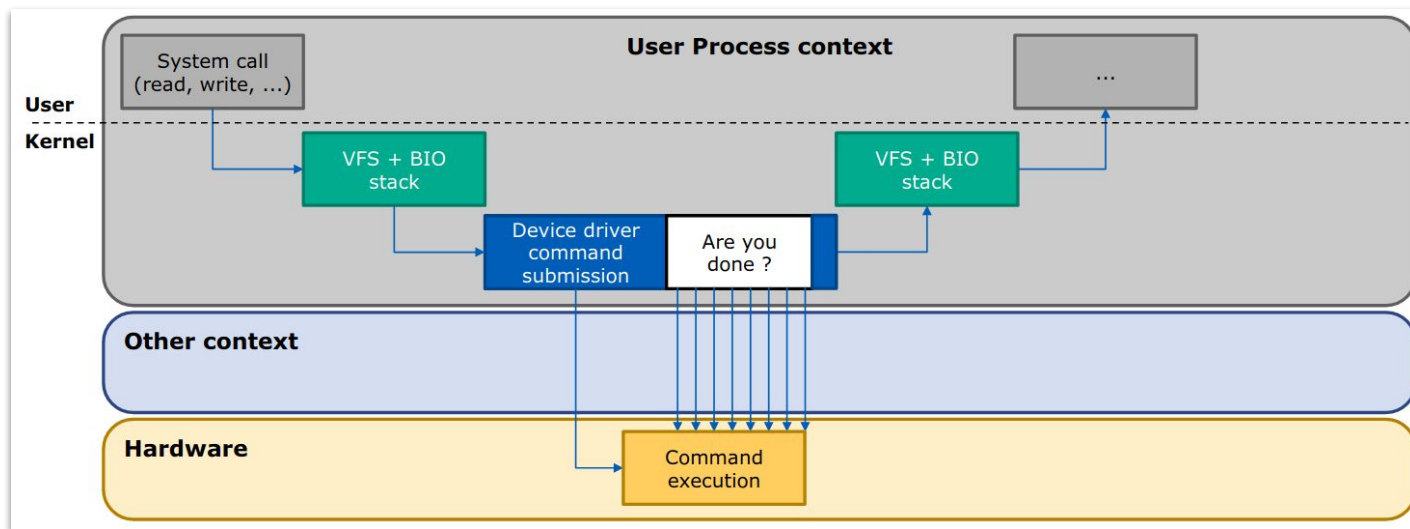
- Device latencies are improving significantly
 - 10s of useconds, overheads shifting from hardware to software
- Scheduling and context switching have latencies comparable to device I/O
 - Does it make sense to context switch when the I/O will be completing in that time?
System calls? (*Can we do zero system call I/O? - lecture 11*)
https://www.usenix.org/legacy/event/osdi10/tech/full_papers/Soares.pdf (OSDI 2010)
- Interrupt generation and processing take time
 - Interrupts destroy the current execution context
 - Poor cache profile, and instruction pipeline flushing
 - Interrupt storm / livelocks (high-performance networking problem in storage)
- Gap in the load, results in CPU entering the energy saving “C” states, thus, introducing latencies of 1-2 useconds

Can we do better?

Synchronous completion: Polling

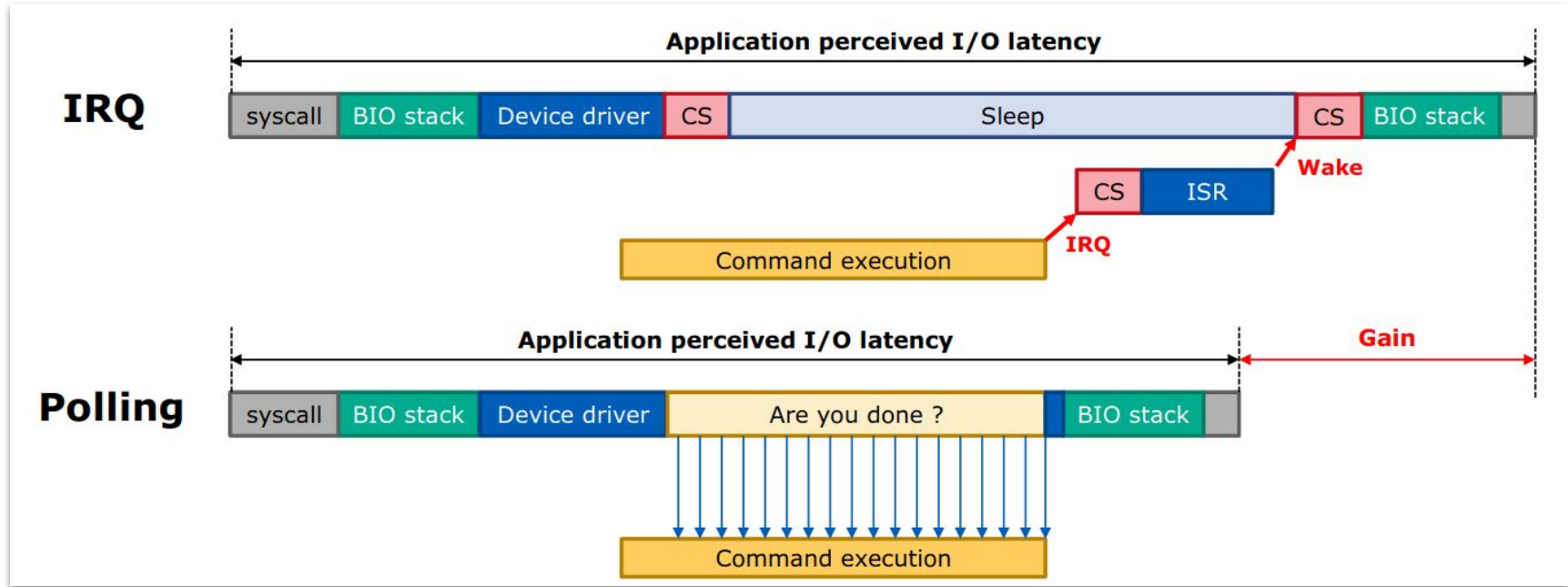
Constantly **poll** to check if the command is completed

- In-place command completion (hence, synchronous in order)
- Better performance

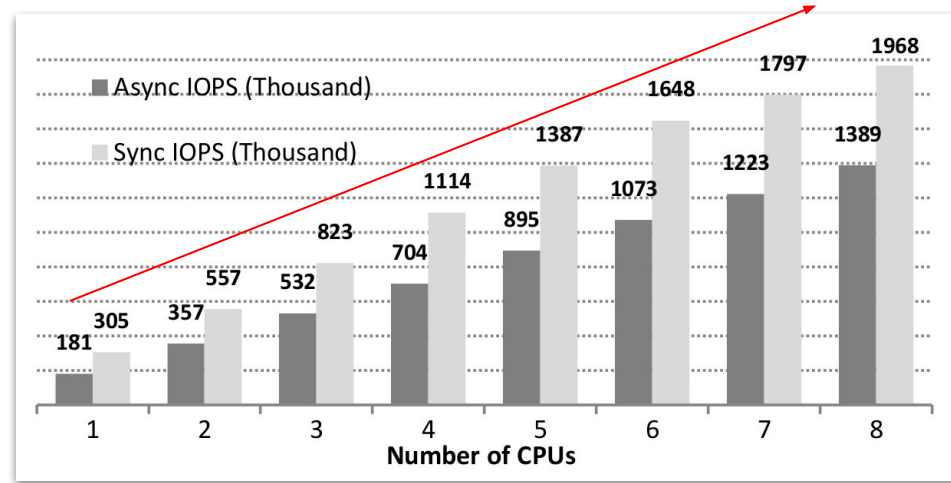
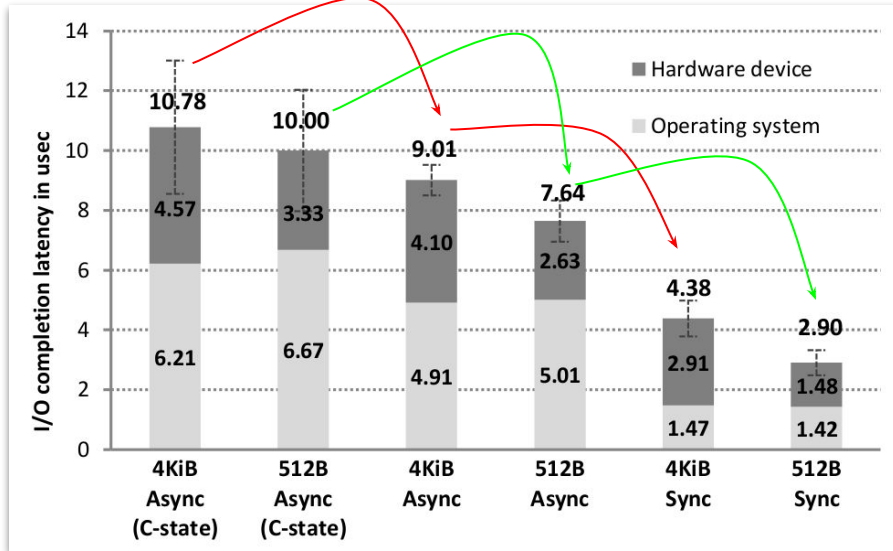


Asynchronous vs. Synchronous completion timeline

Where do the gains come from?



Performance : Sync. vs. Async. completions

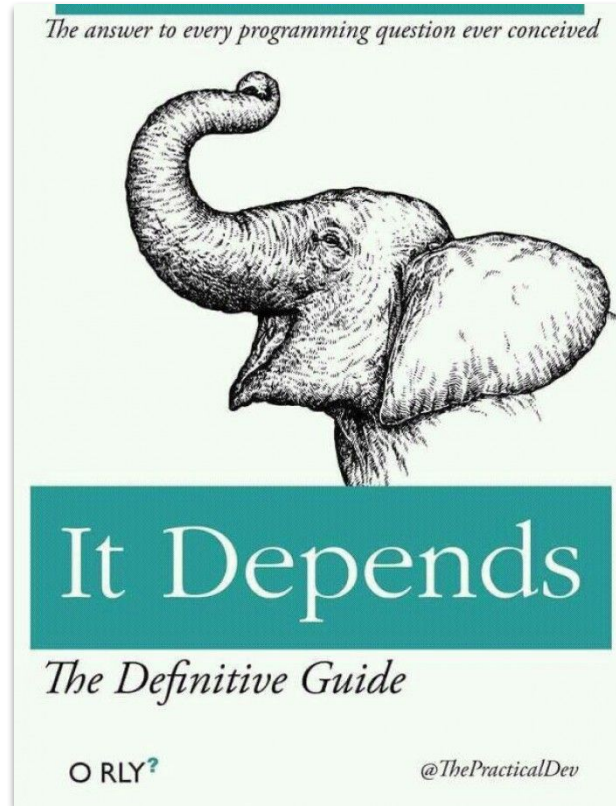


For a single I/O latency

- C-State introduces latencies
- Sync is faster than the async

As interrupts are not taken, less context switches → results in **better** utilization of the CPU cycles (even perhaps both of them use 100% of CPU cycles)

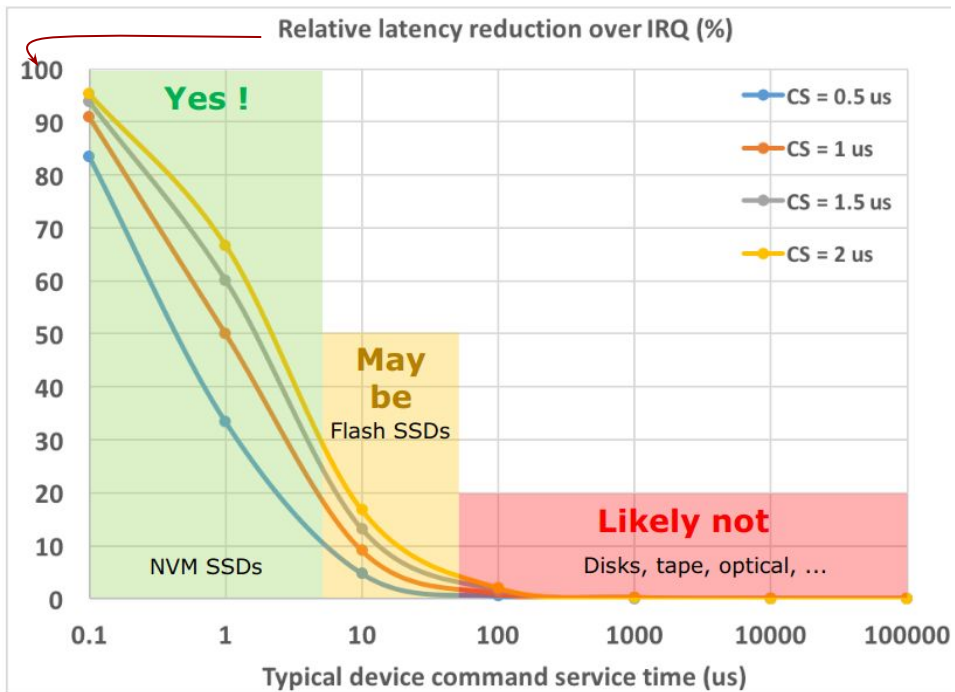
When to use poll?



When to use Poll - The utility spectrum

Like everything in systems, there is no all-good optimizations. When to poll?

- When device service time is comparable to software overheads
 - Cost of scheduling
 - Cost of taking interrupts
 - Device latencies
- Application overheads
 - Can kernel figure out always when to poll? How can application tell kernel to poll?
 - New APIs (io_uring)
 - Buffer management



Short answer: Measure and decide :-)

In Linux

```
atr@atr-XPS-13:~$ cat /sys/block/nvme0n1/queue/io_poll
1
atr@atr-XPS-13:~$ cat /sys/block/nvme0n1/queue/io_poll_delay
-1
atr@atr-XPS-13:~$ █
```

Part of the mainstream kernel

- 1 is enabled with io_poll
- Delay
 - -1: classical spin looping
 - 0 : hybrid strategy, kernel will figure out the best way for you
 - [any_value]: nanosecond time to delay between checking
- It would be an interesting thesis/research project to evaluate impact of these parameters on the “application” performance

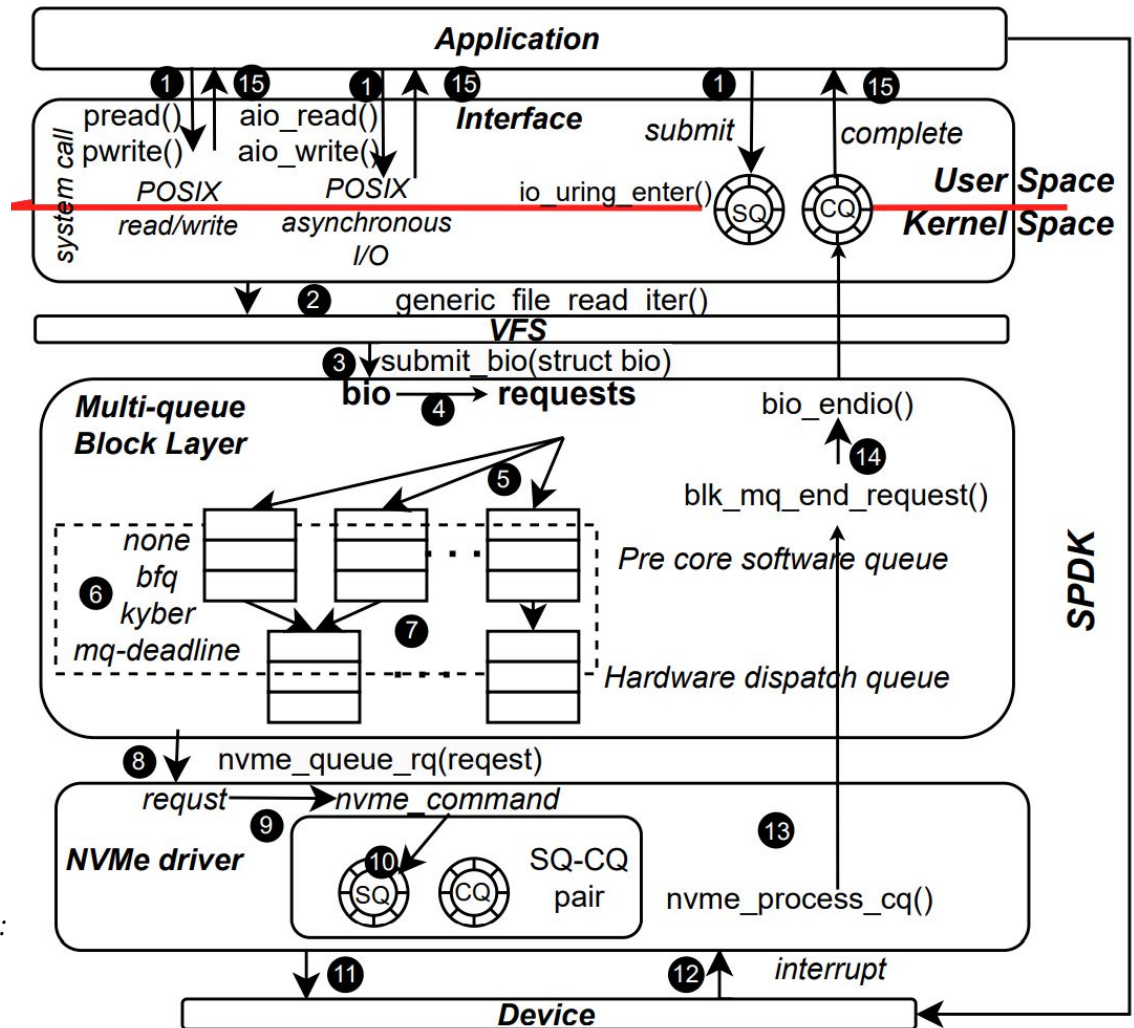
Complexity

How to optimize your storage I/O?

Multiple ways of doing I/O

Queues

Interactions, signals, interrupts



Zebin Ren and Animesh Trivedi,
Performance Characterization of Modern Storage Stacks: POSIX I/O, libaio, SPDK, and io_uring, CHEOPS 2023,
<https://dl.acm.org/doi/abs/10.1145/3578353.3589545>
<https://atlarge-research.com/pdfs/2023-cheops-iostack.pdf>

Implications of Fast NVM on Data Center

storage 1 of 24 TEXT ONLY

Non-volatile Storage

IMPLICATIONS OF THE DATACENTER'S SHIFTING CENTER

MIHIR NANAVATI,
MALTE SCHWARZKOPF,
JAKE WIRES, AND
ANDREW WARFIELD,
COHO DATA

For the entire careers of most practicing computer scientists, a fundamental observation has consistently held true: CPUs are significantly more performant and more expensive than I/O devices. The fact that CPUs can process data at extremely high rates, while simultaneously servicing multiple I/O devices, has had a sweeping impact on the design of both hardware and software for systems of all sizes, for pretty much as long as we've been building them.

This assumption, however, is in the process of being completely invalidated.

contributed articles

DOI:10.1145/3015146

Microsecond-scale I/O means tension between performance and productivity that will need new latency-mitigating ideas, including in hardware.

BY LUIZ BARROSO, MIKE MARTY, DAVID PATTERSON, AND PARTHASARATHY RANGANATHAN

Attack of the Killer Microseconds

THE COMPUTER SYSTEMS WE USE today make it easy for programmers to mitigate event latencies in the nanosecond and millisecond time scales (such as DRAM accesses at tens or hundreds of nanoseconds and disk I/Os at a few milliseconds) but significantly lack support for microsecond (μ s)-scale events. This oversight is quickly becoming a serious problem for programming warehouse-scale computers, where efficient handling of microsecond-scale events is becoming paramount for a new breed of low-latency I/O devices ranging from datacenter networking to emerging memories (see the first sidebar “Is the Microsecond Getting Enough Respect?”).

Processor designers have developed multiple techniques to facilitate a deep memory hierarchy that works at the nanosecond scale by providing a simple synchronous programming interface to the memory system. A load operation will logically block a thread's execution, with the program appearing to resume after the load completes. A host of complex microarchitectural techniques make high performance possible while supporting this intuitive programming model. Techniques include prefetching, out-of-order execution, and branch prediction. Since nanosecond-scale devices are so fast, low-level interactions are performed primarily by hardware.

At the other end of the latency-mitigating spectrum, computer scientists have worked on a number of techniques—typically software based—to deal with the millisecond time scale. Operating system context switching is a notable example. For instance, when a `read()` system call to a disk is made, the operating system kicks off the low-level I/O operation but also performs a software context switch to a different thread to make use of the processor during the disk operation. The original thread resumes execution sometime after the I/O completes. The long overhead of making a disk access (milliseconds) easily outweighs the cost of two context switches (microseconds). Millisecond-scale devices are slow enough that the cost of these software-based mechanisms can be amortized (see Table 1).

These synchronous models for interacting with nanosecond- and millisecond-scale devices are easier than the alternative of asynchronous models. In an asynchronous programming model, the program sends a request to a device and continue processing other work

Key insights

- A new breed of low-latency I/O devices, ranging from faster datacenter networking to emerging non-volatile memories and accelerators, motivates greater interest in microsecond-scale latencies.
- Existing system optimizations targeting nanosecond- and millisecond-scale events are inadequate for events in the microsecond range.
- New techniques are needed to enable simple programs to achieve high performance when microsecond-scale latencies are involved, including new microarchitecture support.

48 COMMUNICATIONS OF THE ACM | APRIL 2017 | VOL. 60 | NO. 4

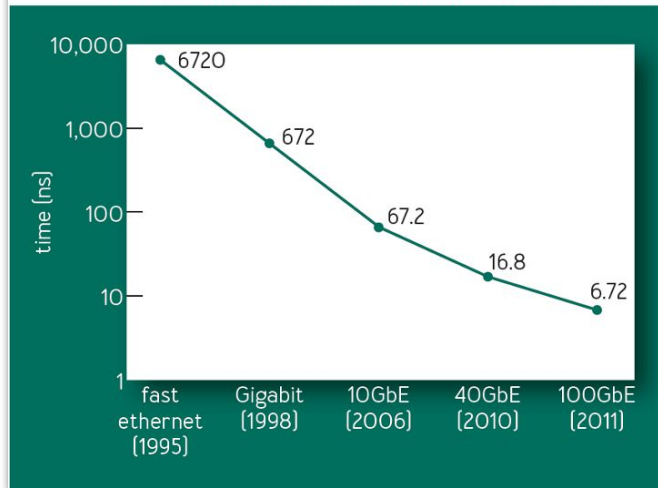
- Mihir Nanavati, Malte Schwarzkopf, Jake Wires, and Andrew Warfield. 2015. Non-volatile Storage: Implications of the Datacenter's Shifting Center. Queue 13, 9 (November-December 2015), 33–56. DOI:<https://doi.org/10.1145/2857274.2874238>
- Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. 2017. Attack of the killer microseconds. Commun. ACM 60, 4 (April 2017), 48–54. DOI:<https://doi.org/10.1145/3015146>

Historically

CPU have been improving (Moore's Law and Multi Core scalability)

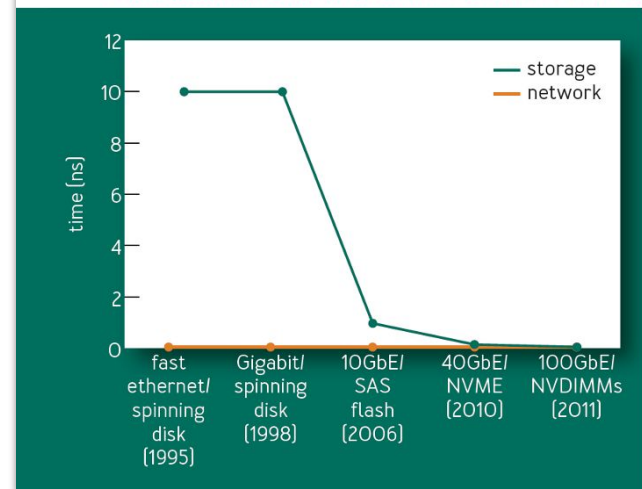
DRAM speeds have been improving (latency not so much)

FIGURE 1: PER-PACKET PROCESSING TIME WITH FASTER NETWORK ADAPTERS



Network performance over time (2-3 oom)

FIGURE 2: PROGRESSION IN SPEED OF SCMS COMPARED TO NETWORK ADAPTERS

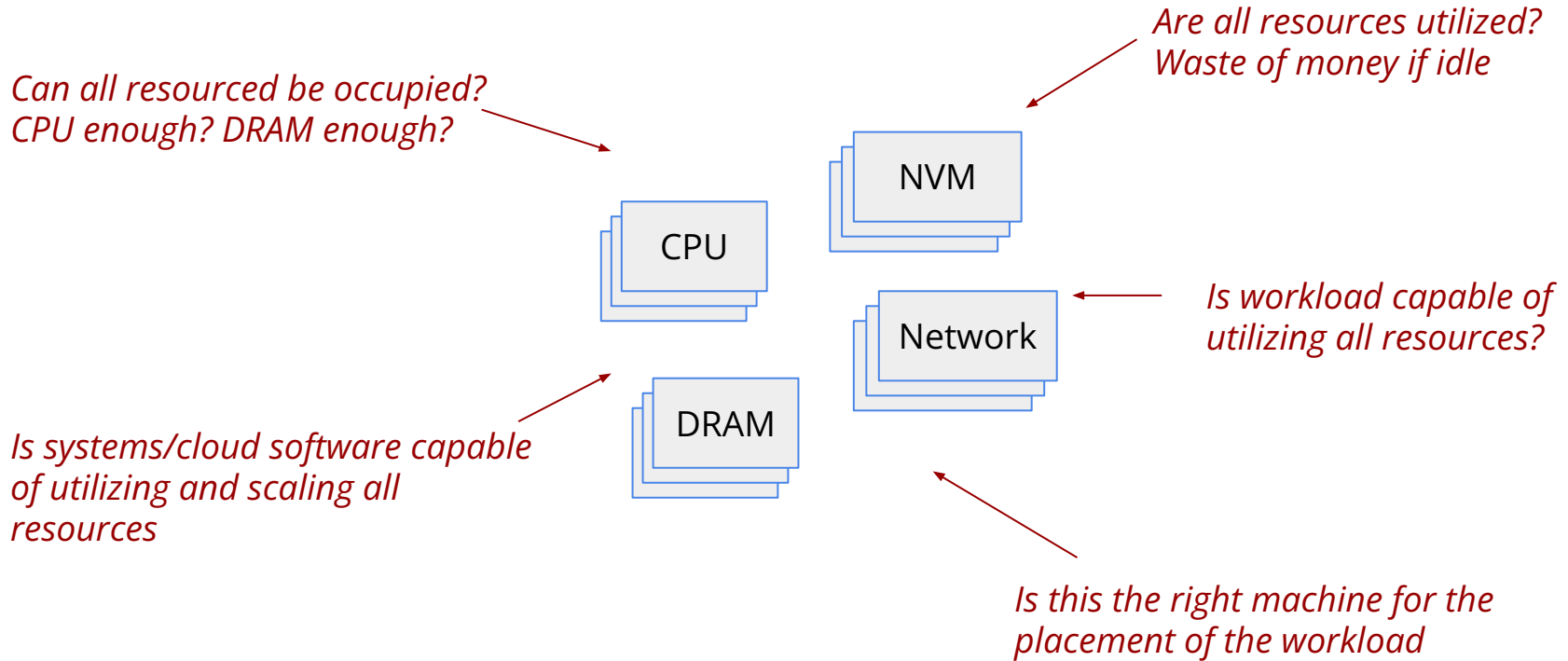


Storage performance over time (4-6 oom)

Trends in the data center

- 1. The age-old assumption that I/O is slow and computation is fast is no longer true*
 - this invalidates decades of design decisions that are deeply embedded in today's systems
 - Examples: use caching, prefetching, trade CPU for I/O (compression?)
- 2. The relative performance of layers in systems has changed by a factor of a thousand times over a very short time (this has never happened in computing before!)*
 - this requires rapid adaptation throughout the systems software stack
 - Examples: PCIe/NVMe storage that exposed overheads in the software stack
- 3. Piles of existing enterprise datacenter infrastructure—hardware and software—are about to become useless (or, at least, very inefficient)*
 - SCMs require rethinking the compute/storage balance and architecture from the ground up
 - Example: moving MySQL from SATA RAID to SSDs improves performance only by 5-7x, the raw devices might offer 10-100-1000x times better performance
 - For your RocksDB project and how to integrate that on ZNS?

A balancing act: Balanced Systems



It is an open-research problem

What you should know from this lecture

- What is NVM Express and why it was developed
- What are the main feature of NVM Express
 - Multiple, deep queues
 - Memory mapped I/O submission and completion
- What are the challenges with the scalability of the block layer on multi-core systems
- What is synchronous (poll) vs. asynchronous completion
 - Why would you poll on a storage stack
- What is Asynchronous I/O stack - what did they propose and why it was beneficial
- Changing trends inside a data center

Lecture Reading List

- High Performance Solid State Storage Under Linux, <https://storageconference.us/2010/Papers/MSST/Seppanen.pdf>, MSST 2010
- Xu and others, Performance Analysis of NVMe SSDs and their Implication on Real World Databases, ACM Systor 2015, <https://dl.acm.org/doi/10.1145/2757667.2757684>.
- A Comparison of NVMe and AHCI, <https://sata-io.org/sites/default/files/documents/NVMe%20and%20AHCI%20long.pdf>
- Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. 2013. Linux block IO: introducing multi-queue SSD access on multi-core systems. In Proceedings of the 6th International Systems and Storage Conference (SYSTOR '13). Association for Computing Machinery, New York, NY, USA, Article 22, 1–10.
- Jisoo Yang, Dave B. Minton, and Frank Hady. 2012. When poll is better than interrupt. In Proceedings of the 10th USENIX conference on File and Storage Technologies (FAST'12). USENIX Association, USA, 3.
- Gyun Lee, Seokha Shin, Wonsuk Song, Tae Jun Ham, Jae W. Lee, and Jinkyu Jeong. 2019. Asynchronous I/O stack: a low-latency kernel I/O stack for ultra-low latency SSDs. In Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '19). USENIX Association, USA, 603–616.
- Mihir Nanavati, Malte Schwarzkopf, Jake Wires, and Andrew Warfield. 2015. Non-volatile Storage: Implications of the Datacenter's Shifting Center. Queue 13, 9 (November-December 2015), 33–56. DOI:<https://doi.org/10.1145/2857274.2874238>
- Jaehyun Hwang, Midhul Vuppapapati, Simon Peter, Rachit Agarwal, Rearchitecting Linux Storage Stack for μ s Latency and High Throughput. OSDI 2021.
- Anastasios Papagiannis, Giorgos Saloustros, Manolis Marazakis, and Angelos Bilas. 2017. Iris: An optimized I/O stack for low-latency storage devices. SIGOPS Oper. Syst. Rev. 50, 2 (December 2016), 3–11. DOI:<https://doi.org/10.1145/3041710.3041713>
- Jie Zhang, and others. 2018. Flashshare: punching through server storage stack from kernel to firmware for ultra-low latency SSDs. In Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation (OSDI'18). USENIX Association, USA, 477–492.

Backup (not a part of the course)

Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs (2019)

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

Gyusun Lee[†], Seokha Shin^{*†}, Wonsuk Song[†], Tae Jun Ham[§], Jae W. Lee[§], Jinkyu Jeong[†]

[†]*Sungkyunkwan University*, [§]*Seoul National University*

{gyusun.lee, seokha.shin, wonsuk.song}@csi.skku.edu, {taejunham, jaewlee}@snu.ac.kr, jinkyu@skku.edu

Abstract

Today's ultra-low latency SSDs can deliver an I/O latency of sub-ten microseconds. With this dramatically shrunken device time, operations inside the kernel I/O stack, which were traditionally considered lightweight, are no longer a negligible portion. This motivates us to reexamine the storage I/O stack design and propose an *asynchronous* I/O stack (AIOS), where synchronous operations in the I/O path are replaced by asynchronous ones to overlap I/O-related CPU operations with device I/O. The asynchronous I/O stack leverages a lightweight block layer specialized for NVMe SSDs using the page cache without block I/O scheduling and merging, thereby reducing the sojourn time in the block layer. We prototype the proposed asynchronous I/O stack on the Linux kernel and evaluate it with various workloads. Synthetic FIO benchmarks demonstrate that the application-perceived I/O latency falls into single-digit microseconds for 4 KB random reads on Optane SSD, and the overall I/O latency is reduced by 15–33% across varying block sizes. This I/O latency reduction leads to a significant performance improvement of real-world applications as well: 11–44% IOPS increase on RocksDB and 15–30% throughput improvement on Filebench and OLTP workloads.

One way to alleviate the I/O stack overhead is to allow user processes to directly access storage devices [6, 16, 27, 28, 49]. While this approach is effective in eliminating I/O stack overheads, it tosses many burdens to applications. For example, applications are required to have their own block management layers [49] or file systems [15, 43, 49] to build useful I/O primitives on top of a simple block-level interface (e.g., BlobFS in SPDK). Providing protections between multiple applications or users is also challenging [6, 16, 28, 43]. These burdens limit the applicability of user-level direct access to storage devices [49].

An alternative, more popular way to alleviate the I/O stack overhead is to optimize the kernel I/O stack. Traditionally, the operating system (OS) is in charge of managing storage and providing file abstractions to applications. To make the kernel more suitable for fast storage devices, many prior work proposed various solutions to reduce the I/O stack overheads. Examples of such prior work include the use of polling mechanism to avoid context switching overheads [5, 47], removal of bottom halves in interrupt handling [24, 35], proposal of scatter/scatter I/O commands [37, 50], simple block I/O scheduling [3, 24], and so on. These proposals are effective in reducing I/O stack overheads, and some of those are adopted by mainstream OS (e.g., I/O stack for NVMe SSDs in Linux).

What are the Challenges

A new class of ultra-low latency devices

- Optane SSDs, Samsung Z-SSD
- < 10 usec latencies, 3+GB/s bandwidth

Pressure on the software stack to deliver performance, **do you get the raw device latencies when doing I/O?**

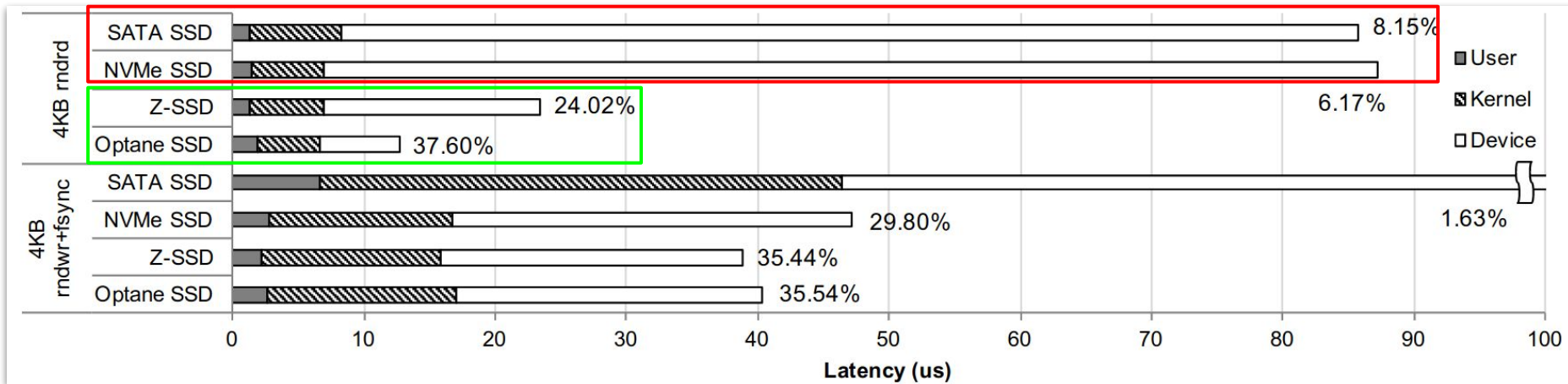
- *Understand that software and optimizations for 100usec will look very different than optimizations for 10usec*

Polling helps to eliminate the context switch overheads between the the time we issue an I/O request to the device and we get a response ...

But what is happening before that?



Quantify the Problem

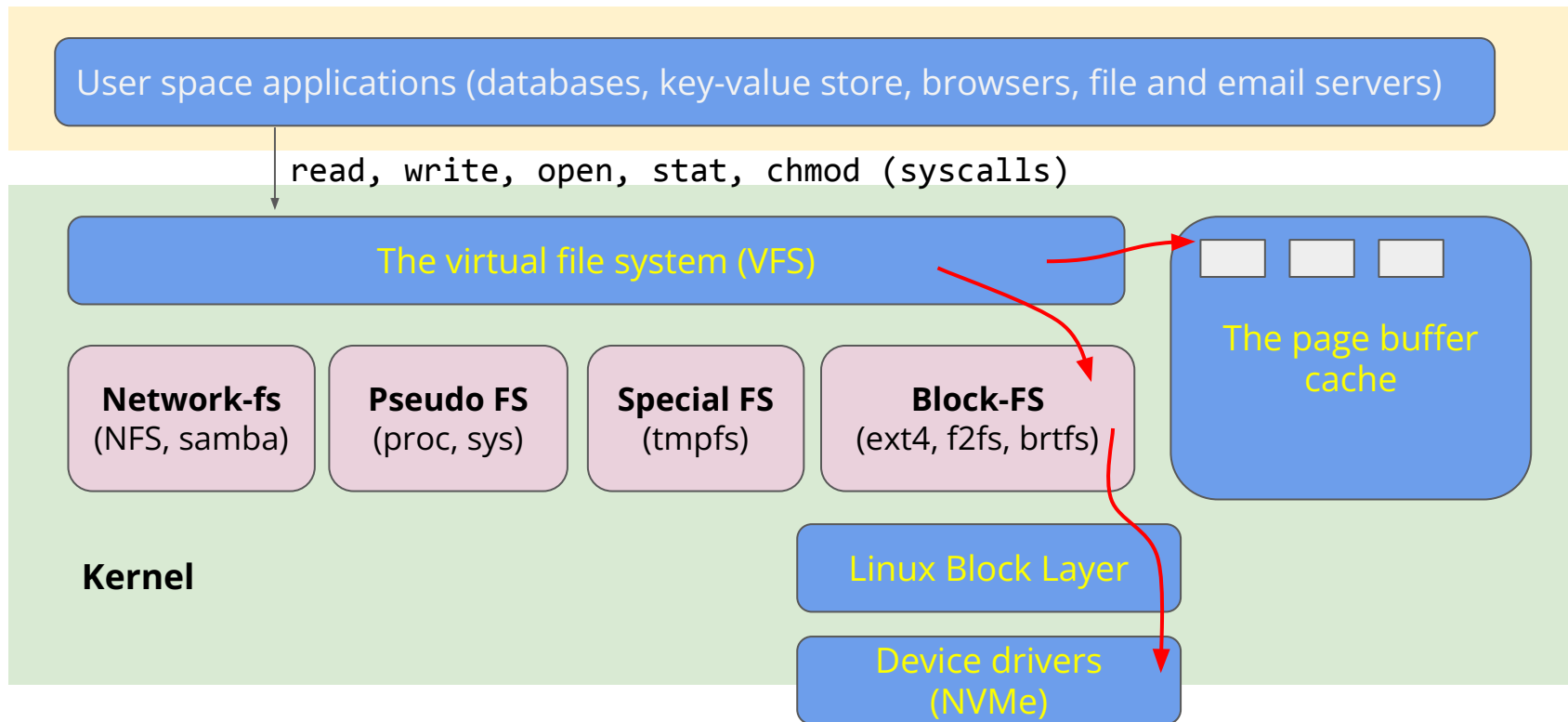


Ultra-low SSDs like Z-SSDs and Optane SSDs have

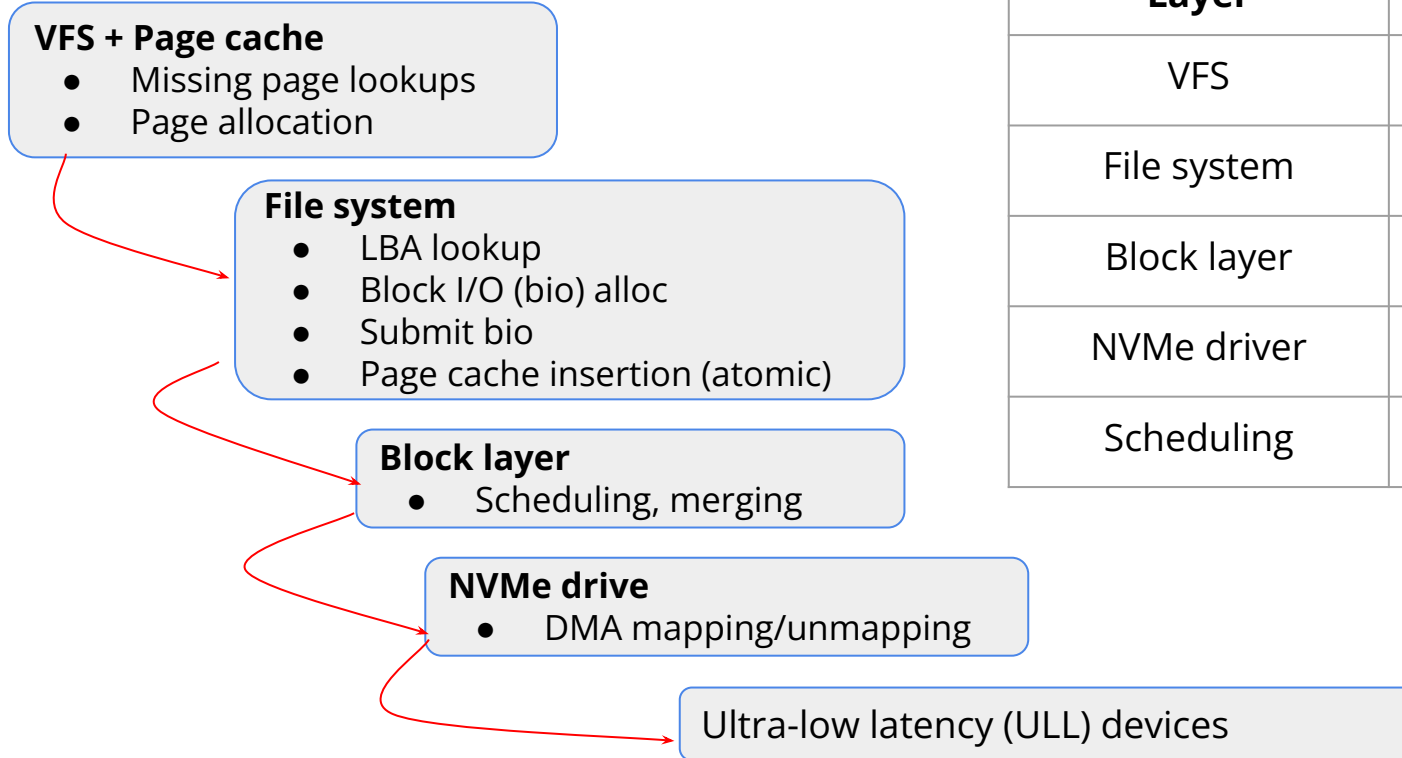
- Much smaller device-time for reads
- Smaller device-time for writes

Optane SSDs have 50-50 split between hardware and software time, *can we do better in software?*

The Linux Storage Stack - Software (simplified)



Deeper Dive on the read Path



Layer	% in kernel time
VFS	9-10.8%
File system	4.5-12%
Block layer	26-28.5%
NVMe driver	10-11%
Scheduling	25-41.5%

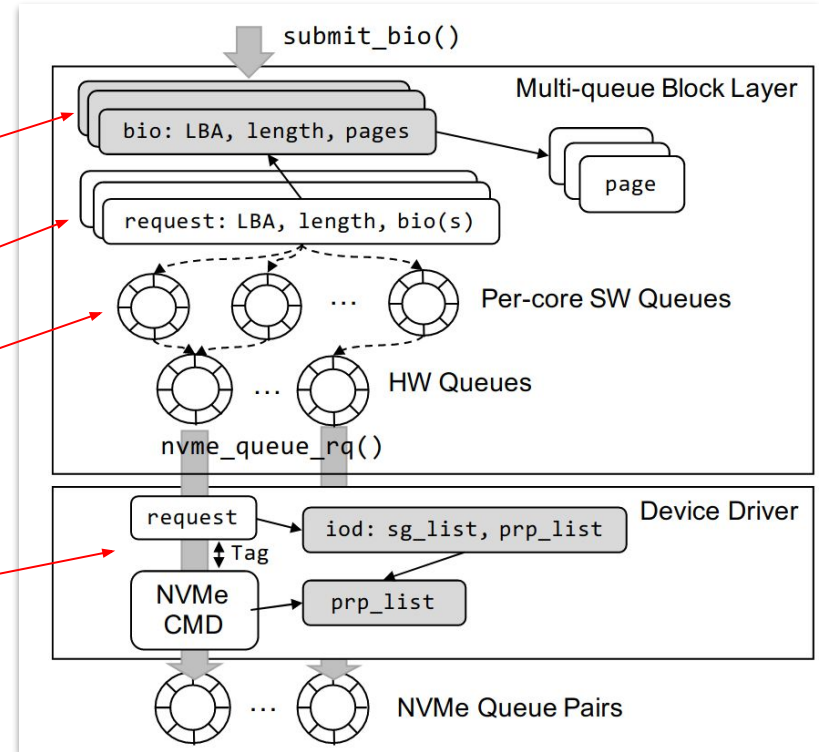
Block Layer Overheads

A lots of steps inside the block layer

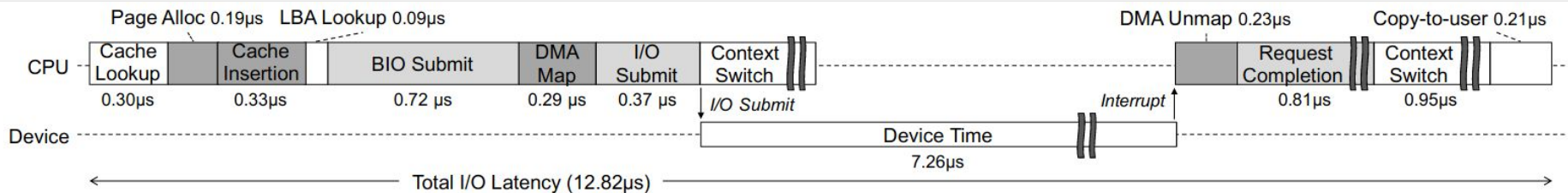
- Dynamic allocation of struct `bio`
 - Separate slab cache
- Transformation of a `bio` (kernel) to an I/O request (device)
- Passing through software and hardware queues (multiQ)
- IO descriptor object and preparing a DMA request for transfer
 - Memory mapping/unmapping

Lots of step (are they all necessary?)

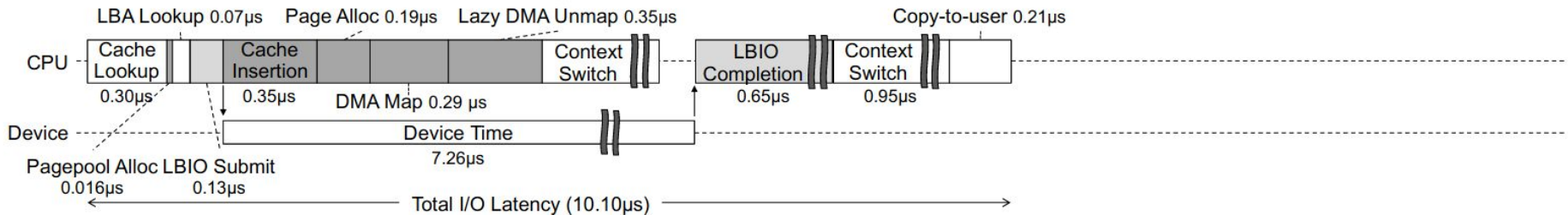
Dynamic objects in the shaded areas



Timeline Comparison - Vanilla vs. Proposed read Path



(a) Vanilla read path



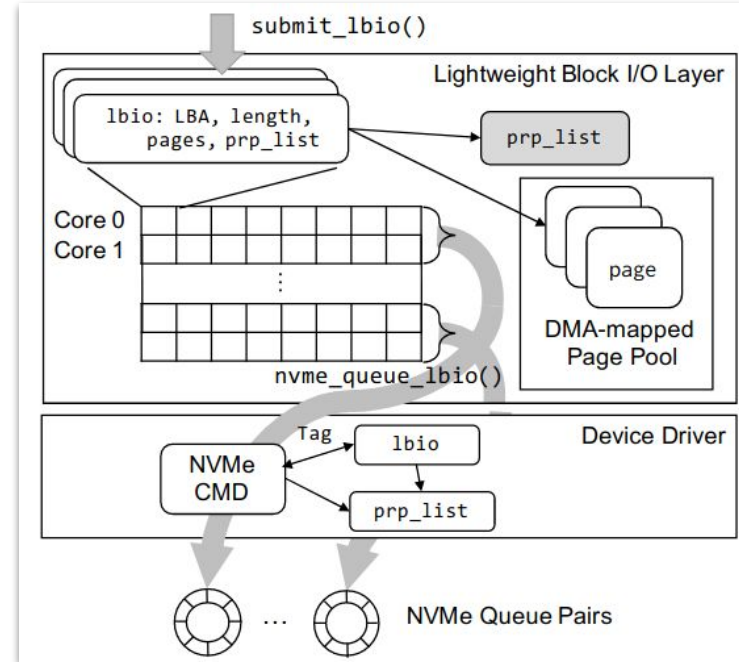
(b) Proposed read path

How to Make it Happen?

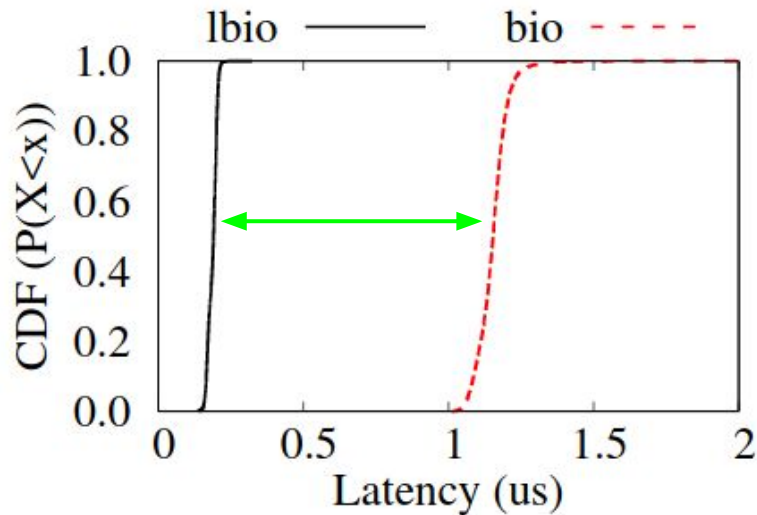
Lightweight block I/O layer (or LBIO)

- Simple, but very interesting idea
- Preallocate a bunch of object
 - Single lbio structure containing all information
 - Pre DMA-mapped page pool for I/O
 - Reduce locking and scheduling by mapping 1:1 pages to CPU cores and NVMe queues
 - Core x queue dimension

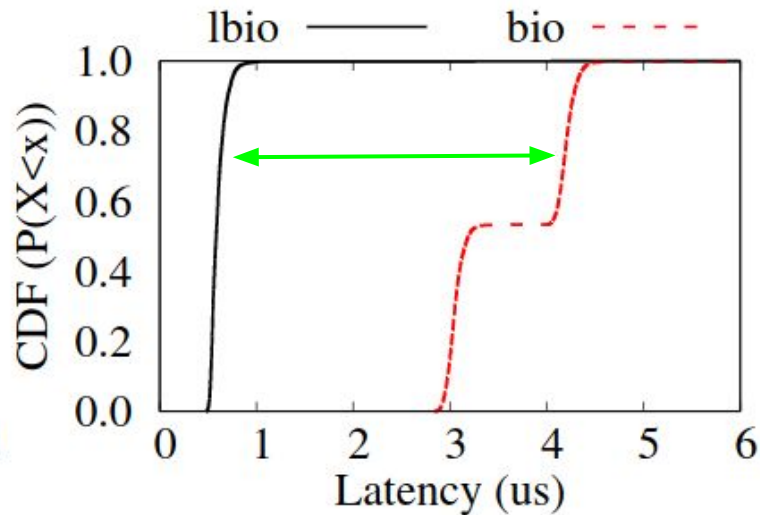
The idea is quite general and is used in many other systems like high-performance networking (RDMA - preallocation of buffers)



Optimized Lightweight Block layer



(a) 4 KB random read

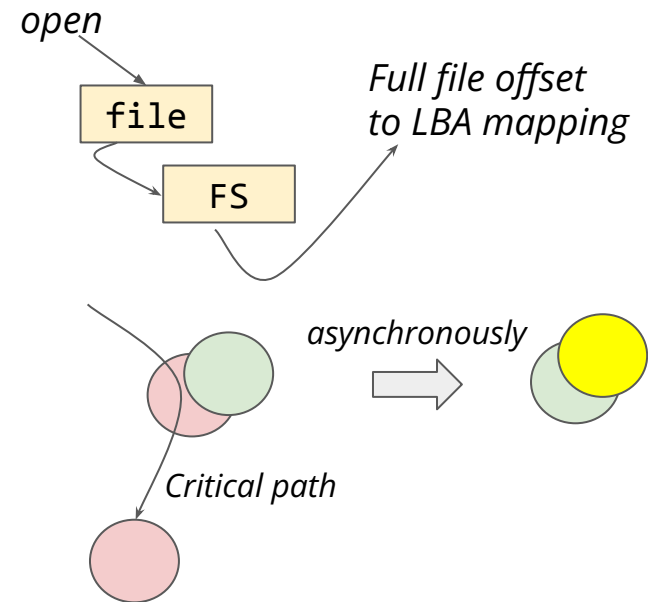


(b) 32 KB random read

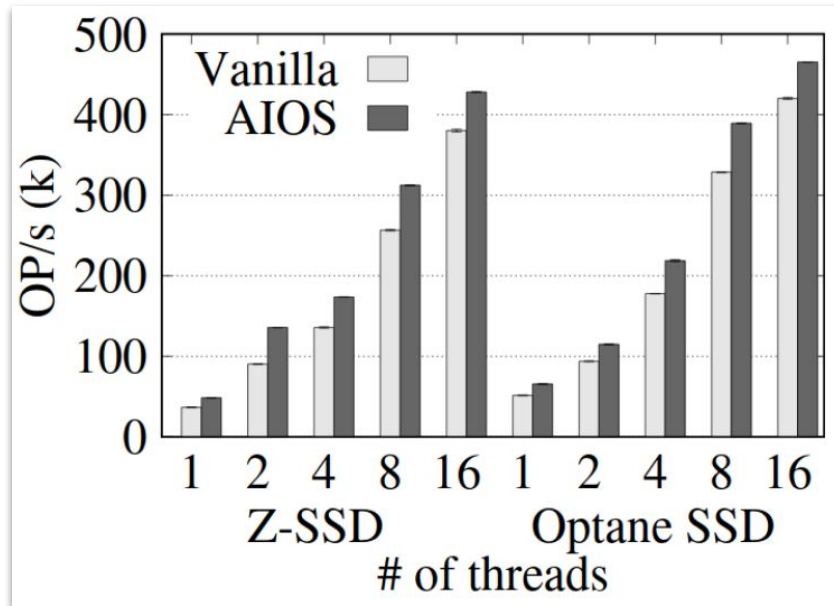
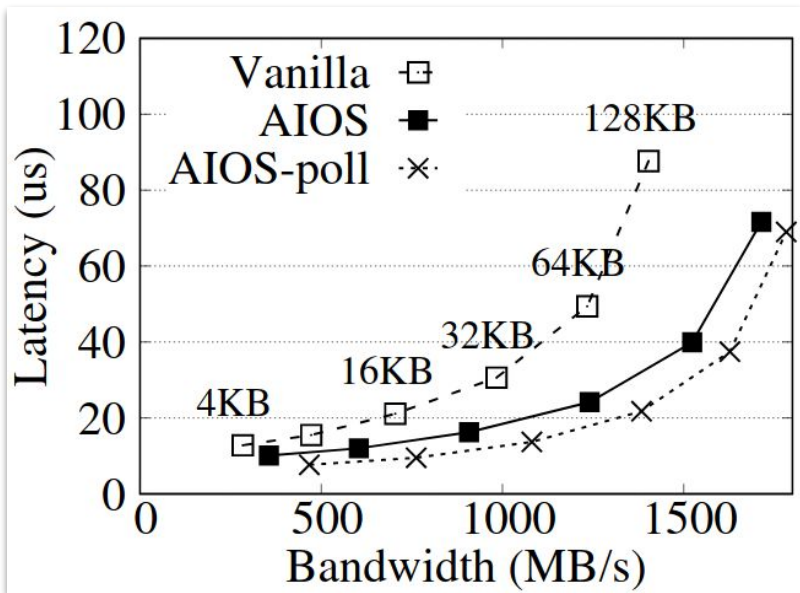
- 0.18–0.60 μ s I/O submission latency in lbio
- 83.4%–84.4% shorter than the original block layer

Rest of the File read Path

1. **How to do a fast file offset → LBA lookup?** Not all mappings might be in memory and file system needs to do further I/O to look them up
 - a. **Solution:** when a file is open, preload the whole mapping in the memory
 - b. Memory consumption? Can be done selectively
2. **How to manage DMA-mapped page pool?**
 - a. **Solution:** pick one, start using it, but asynchronously add another page
 - b. Solution: once I/O is finished, only unmap lazily when the new page is needed
3. **Atomic page-cache insertion before I/O**
 - a. **Solution:** well...there will be duplicate work, and we will discard it



Performance: Microbenchmark and RocksDB



- AIOS results in scalable latency gains with higher bandwidth
- RocksDB random read performance is improved by 11-32%