

Exploring the End-to-End Storage Stack on Modern Storage Hardware

Animesh Trivedi

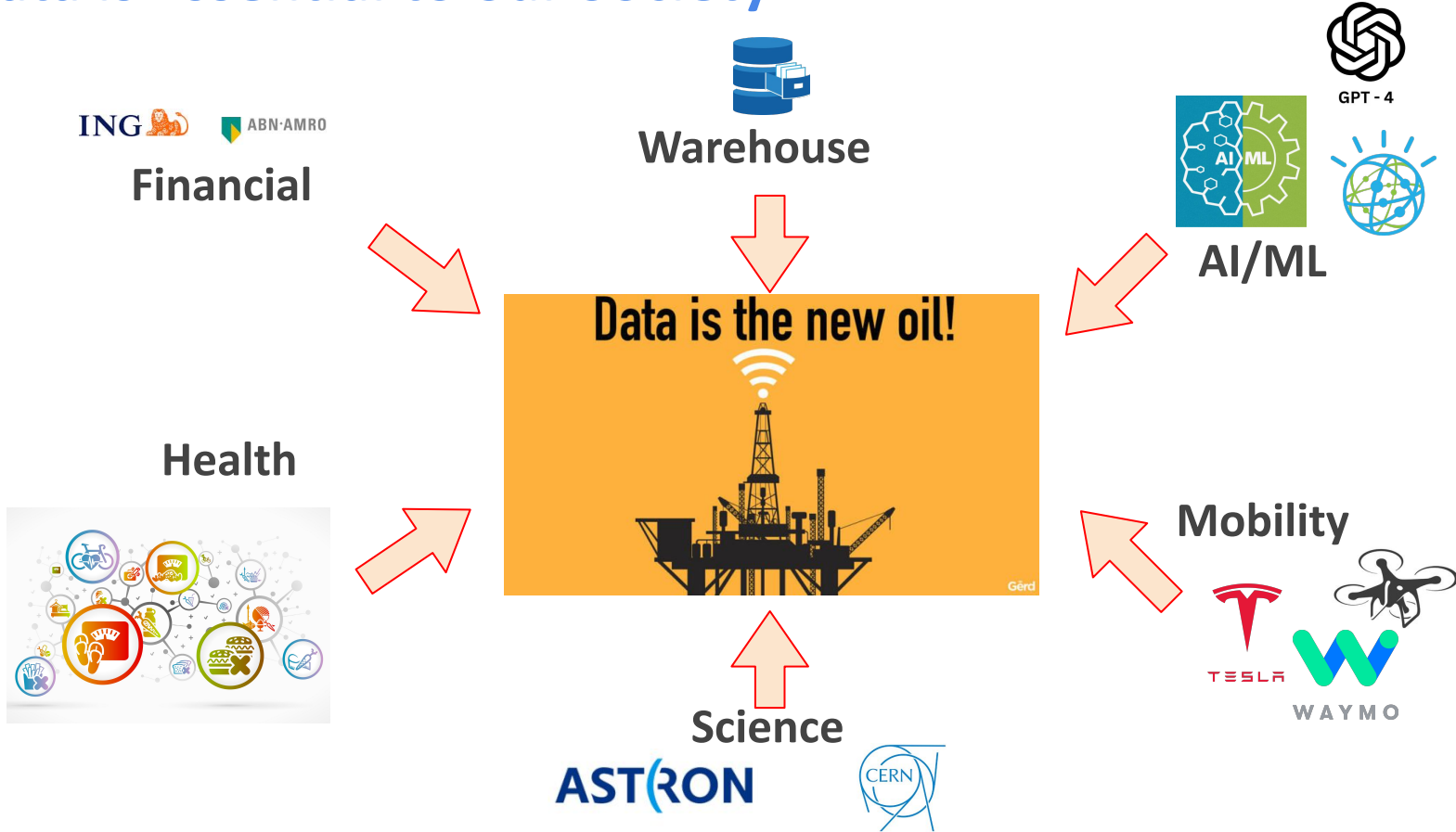
Assistant Professor (tenured)

<https://animeshtrivedi.github.io/>

December, 2023

(presenting on behalf of many in the research team at VU Amsterdam)

Data is Essential to our Society



If a Zettabyte does not Resonate

Assume (*illustration purposes only*):

- 1 grain of rice is 1 byte of data



If a Zettabyte does not Resonate

Assume (*illustration purposes only*):

- 1 grain of rice is 1 byte of data
- **Kilobyte**: a cup of rice



If a Zettabyte does not Resonate

Assume (*illustration purposes only*):

- 1 grain of rice is 1 byte of data
- **Kilobyte**: a cup of rice
- **Megabyte**: 8 bags of rice



If a Zettabyte does not Resonate

Assume (*illustration purposes only*):

- 1 grain of rice is 1 byte of data
- **Kilobyte**: a cup of rice
- **Megabyte**: 8 bags of rice
- **Gigabyte**: 3 semi-trucks



If a Zettabyte does not Resonate

Assume (*illustration purposes only*):

- 1 grain of rice is 1 byte of data
- **Kilobyte**: a cup of rice
- **Megabyte**: 8 bags of rice
- **Gigabyte**: 3 semi-trucks
- **Terabyte**: 2 container ships



If a Zettabyte does not Resonate

Assume (*illustration purposes only*):

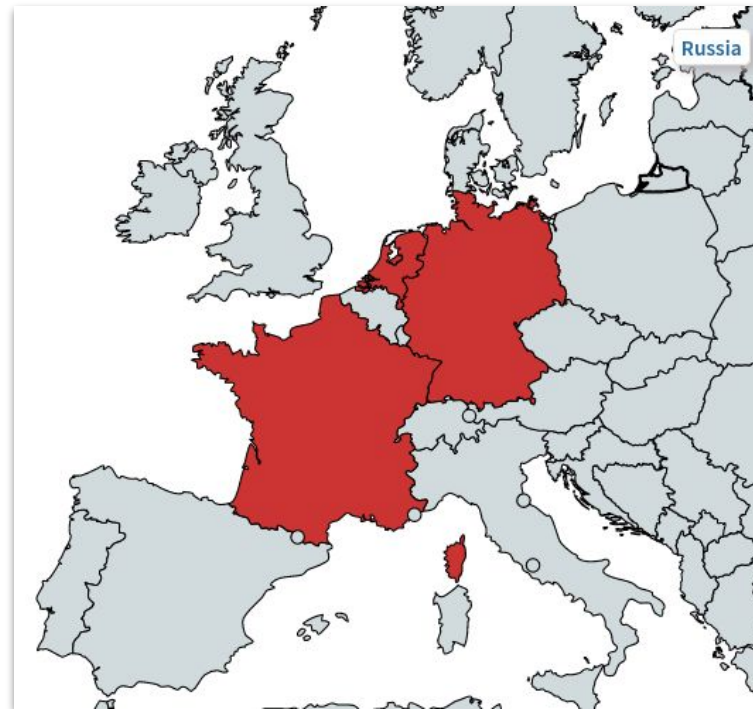
- 1 grain of rice is 1 byte of data
- **Kilobyte**: a cup of rice
- **Megabyte**: 8 bags of rice
- **Gigabyte**: 3 semi-trucks
- **Terabyte**: 2 container ships
- **Petabyte**: Covers Maastricht



If a Zettabyte does not Resonate

Assume (*illustration purposes only*):

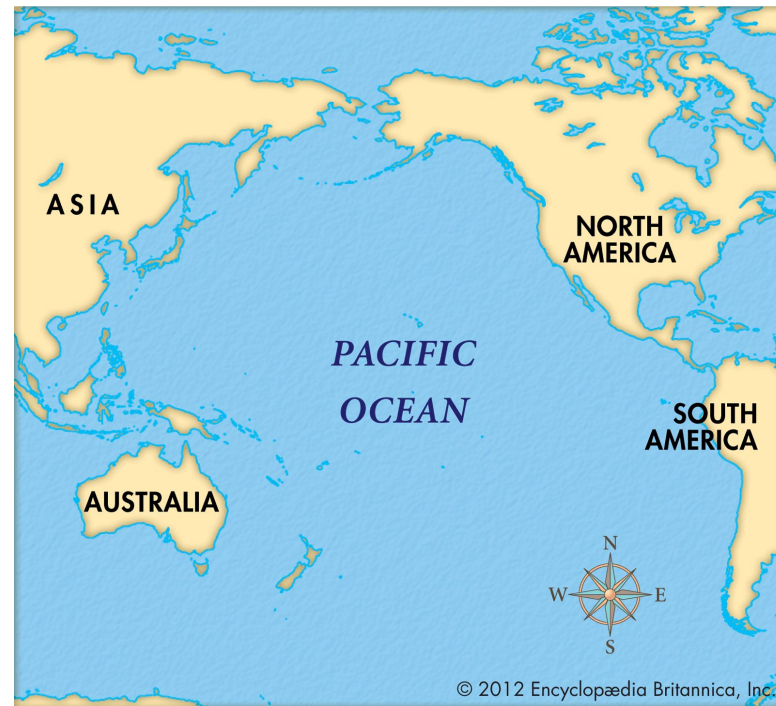
- 1 grain of rice is 1 byte of data
- **Kilobyte**: a cup of rice
- **Megabyte**: 8 bags of rice
- **Gigabyte**: 3 semi-trucks
- **Terabyte**: 2 container ships
- **Petabyte**: Covers Maastricht
- **Exabyte**: Covers NL + DE + FR



If a Zettabyte does not Resonate

Assume (*illustration purposes only*):

- 1 grain of rice is 1 byte of data
- **Kilobyte**: a cup of rice
- **Megabyte**: 8 bags of rice
- **Gigabyte**: 3 semi-trucks
- **Terabyte**: 2 container ships
- **Petabyte**: Covers Maastricht
- **Exabyte**: Covers NL + DE + FR
- **Zettabyte**: Fills up the Pacific Ocean



If a Zettabyte does not Resonate

Assume (*illustration purposes only*):

- 1 grain of rice is 1 byte of data
 - **Kilobyte**: a cup of rice
 - **Megabyte**: 8 bags of rice
 - **Gigabyte**: 3 semi-trucks
 - **Terabyte**: 2 container ships
 - **Petabyte**: Covers Maastricht
 - **Exabyte**: Covers NL + DE + FR
 - **Zettabyte**: Fills up the Pacific Ocean
 - **Yottabytes**: An Earth size rice ball
- We are here* →
- *By 2030*



Non-Volatile Memory (NVM) Storage to the Rescue...

tom's HARDWARE US Edition

Home Reviews Best Picks Raspberry Pi CPUs GPUs Coupons

TRENDING Ryzen 7 7800X3D Raptor Lake Ryzen 9 7950X3D

When you purchase through links on our site, we may earn an affiliate commission. [Here's how it works.](#)

Home > News

SK Hynix's New SSD Boasts 1.4 Million IOPS

By Aaron Klotz last updated May 20, 2022

Well over 1 Million IOPS

[f](#) [t](#) [in](#) [p](#) [v](#) [e](#) [c](#) [omments \(5\)](#)

(Image credit: Amazon)



Samsung Newsroom CORPORATE | PRODUCTS | INSIGHTS | PRESS RESOURCES

Samsung Develops High-Performance PCIe 5.0 SSD for Enterprise Servers

Korea on December 23, 2021

Audio [Share](#)

*Samsung's PCIe 5.0 SSD will provide nearly two times faster data transfer speeds and 30% enhanced power efficiency than the previous generation, resulting in lower server operating costs**

Samsung's PM1743 will feature a sequential read speed of up to 13,000 megabytes per second (MB/s) and a random read speed of 2,500K input/output operations per second (IOPS), offering 1.9x and 1.7x faster speeds over the previous PCIe 4.0-based products. Moreover, write speeds have been elevated significantly, with a sequential write speed of 6,600 MB/s and a random write speed of 250K IOPS, also delivering 1.7x and 1.9x faster speeds, respectively. These remarkable data transfer rates will allow enterprise server manufacturers deploying the PM1743 to enjoy a much higher level of performance.

server manufacturers to drive next-generation servers

gy, today announced that it has developed the (Component Interconnect Express) 5.0 interface

SAMSUNG PM1743



Novel Systems Designs (CXL)

Workload-specific Configurations

FlashNeuron: SSD-Enabled Large-Batch Training of Very Deep Neural Networks

Jonghyun Bae[†] Jongsung Lee[‡] Yunho Jin[†] Sam Son[†] Shine Kim[†] Hakbeom Jang[†]
 Tae Jun Han[†] Jae W. Lee[‡]
[†]Seoul National University [‡]Samsung Electronics

Abstract

Deep neural networks (DNNs) are widely used in various AI application domains such as computer vision, natural language processing, autonomous driving, and bioinformatics. As DNNs continue to get wider and deeper to improve accuracy, the limited DRAM capacity of a training platform like GPU often becomes the limiting factor on the size of DNNs and batch size—called *memory capacity wall*. Since increasing the batch size is a popular technique to improve hardware utilization, this can yield a suboptimal training throughput. Recent proposals address this problem by offloading some of the intermediate data (e.g., feature maps) to the host memory. However, they fail to provide robust performance as the training process on a GPU contends with applications running on a CPU for memory bandwidth and capacity. Thus, we propose FlashNeuron, the first DNN training system using an NVMe SSD as a backing store. To fully utilize the limited SSD write bandwidth, FlashNeuron introduces an offloading scheduler, which selectively offloads a set of intermediate data to the SSD in a compressed format without increasing DNN evaluation time. FlashNeuron causes minimal interference to CPU processes as the GPU and the SSD directly communicate for data transfers. Our evaluation of FlashNeuron with four state-of-the-art DNNs shows that FlashNeuron can increase the batch size by a factor of 12.4x to 14.0x over the maximum allowable batch size on an NVIDIA Tesla V100 GPU with 16GB DRAM. By employing a larger batch size, FlashNeuron also improves the training throughput by up to 37.8% (with an average of 30.3%) over the baseline using GPU memory only, while minimally disturbing applications running on CPU.

1 Introduction

Deep neural networks (DNNs) are the key enabler of emerging AI-based applications and services such as computer vision [19, 22, 38, 53, 54], natural language processing [2, 11, 13, 51, 67], and bioinformatics [46, 73]. With a relentless pursuit of higher accuracy, DNNs have become wider and deeper to increase model size [65]. It is because even a 1% accuracy loss (or gain) potentially affects the experience of millions of users if the AI application serves a billion of people [47].

DNNs must be trained before deployment to find optimal network parameters that minimize the error rate. Stochastic Gradient Descent (SGD) is the dominant algorithm used for DNN training [15]. In SGD, the entire dataset is divided into multiple (mini-)batches, and weight gradients are calculated and applied to the network parameters (weights) for each batch via backward propagation. Unlike inference, the training algorithm reuses the intermediate results (e.g., feature maps) produced by a forward propagation during the backward propagation, thus requiring a lot of memory space [55]. This GPU *memory capacity wall* [33] often becomes the limiting factor on DNN size and its throughput. Specifically, such a large memory capacity requirement forces a GPU device to operate at a relatively small batch size, which often adversely affects its throughput. The use of multiple GPUs can partially bypass the memory capacity wall because a careful use of multiple GPUs can achieve near-linear improvements in throughput [27, 28, 59]. However, such a throughput improvement comes with the linear increase in the GPU cost, which is often a major component of the overall system cost. As a result, the use of multiple GPUs often ends up with sub-optimal cost efficiency (i.e., throughput/system cost) as it does not change the fact that each GPU is not operating at its full capacity due to the limited per-GPU batch size.

This memory capacity problem in DNN training has drawn much attention from the research community. The most popular approach is to utilize the host CPU memory as a backing store to offload some of the tensors that are not immediately used [8, 9, 24, 42, 55, 62]. However, this *buffering-on-memory* approach fails to provide robust performance as the training process on the GPU contends with applications running on the CPU for memory bandwidth and capacity (e.g., data augmentation methods [5, 41, 57, 61] to boost training accuracy). Moreover, these proposals focus mostly on increasing batch size but less on improving training throughput. Therefore, they often yield a low training throughput as the cost of CPU-GPU data transfers outweighs a larger batch's benefits.

Thus, we propose FlashNeuron, the first DNN training system using a high-performance SSD as a backing store. While NVMe SSDs are a promising alternative to substitute or augment DRAM, they have at least an order of magnitude

Overcoming the Memory Wall with CXL-Enabled SSDs

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

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 Gholami 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 solid-state drive (SSD) is

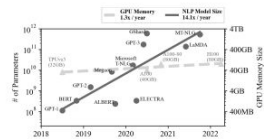


Figure 1: The trend in memory requirements for NLP applications [11, 30, 43]. The number of parameters increases by a factor of 14.1x per year, while the memory capacity in GPUs only grows by a factor of 1.3x every year.

in the terabyte scale [23], a sufficiently large capacity to address the memory wall challenge. The use of flash memory as main memory is enabled by the recent emergence of interconnect technologies such as CXL [3], Gen-Z [7], CCIX [2], and OpenCAPI [2], which allow PCIe (Peripheral Component Interconnect Express) devices to be accessed directly by the CPU through load/store instructions. Furthermore, these technologies promise excellent scalability as more PCIe devices can be attached across switches [13] unlike DIMM (Dual In-line Memory Module) used for DRAM.

However, there are three main challenges to using flash memory as CPU-accessible main memory. First, there is a granularity mismatch between memory requests and flash memory. This results in a significant traffic amplification on top of the existing need for indirection in flash [23, 33]; for example, a 64KB cache line flush to the CXL-enabled flash would result in 16KB flash memory page read, 64B update, and 16KB flash program to a different location (assuming a 16KB page-level mapping). Second, flash memory is still orders of magnitude slower than DRAM (tens of microseconds vs. tens of nanoseconds) [5, 24]. As a consequence, while the peak data transfer rate between the two technologies is similar [4, 15], the long flash memory latency hinders sustained performance as data-intensive applications can only endure

When Poll is More Energy Efficient than Interrupt

Bryan Harris and Nihat Altıparmak
 Dept. of Computer Science & Engineering
 University of Louisville
 {bryan.harris, n.nihat.altiparmak}@louisville.edu

ABSTRACT

Polling is commonly indicated to be a more suitable IO completion mechanism than interrupt for ultra-low latency storage devices. However, polling's impact on overall energy efficiency has not been thoroughly investigated. In this paper, contrary to common belief, we show that polling can also be more energy efficient than interrupt. To do so, we systematically investigate the energy efficiency of all available Linux IO completion mechanisms, including interrupt, classic polling, and hybrid polling using a real ultra-low latency storage device, a power meter, and various workload behaviors. Our experimental results indicate that although hybrid polling provides a good trade-off in CPU utilization, it is the least energy efficient, whereas classic polling is the most energy efficient for low latency IO requests. To the best of our knowledge, this is the first paper classifying polling as more energy efficient than interrupt for a real secondary storage device, and we hope that our observations will lead to more energy efficient IO completion mechanisms for new generation storage device characteristics.

CCS CONCEPTS

• Information systems → Storage power management; Storage class memory.

KEYWORDS

IO completion, energy efficiency

ACM Reference Format:

Bryan Harris and Nihat Altıparmak. 2022. When Poll is More Energy Efficient than Interrupt. In *14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '22)*, June 27–28, 2022, Virtual Event, USA. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3538643.3539747>

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 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/.
 HotStorage '22, June 27–28, 2022, Virtual Event, USA
 © 2022 Association for Computing Machinery.
 ACM ISBN 978-1-4503-9399-7/22/06...\$15.00
<https://doi.org/10.1145/3538643.3539747>

1 INTRODUCTION

With the most recent advancements in data storage technology, a new category of Solid-State Drives (SSDs) have emerged. These devices are referred to as Ultra-Low Latency (ULL) SSDs and are broadly classified as providing data access in less than 10 μ s [17]. Various vendors including Intel, Samsung, and Toshiba have representative ULL SSDs [3, 4, 20], where Intel's latest generation of the Optane SSD is advertised to deliver read IO in 5 μ s and write IO in 6 μ s [6]. ULL IO performance providing sub-10 μ s data access latency renders the performance of traditional, interrupt-based IO completion mechanism questionable. Both industry and academia suggested replacing interrupts with polling based IO completion methods for improved latency in such devices [11, 13, 15, 19, 22, 25–27], where polling has also been supported by the Linux kernel since version 4.4. However, one must also consider the relationship between IO performance and power consumption, as power saving methods may not be worth the resulting loss in IO performance.

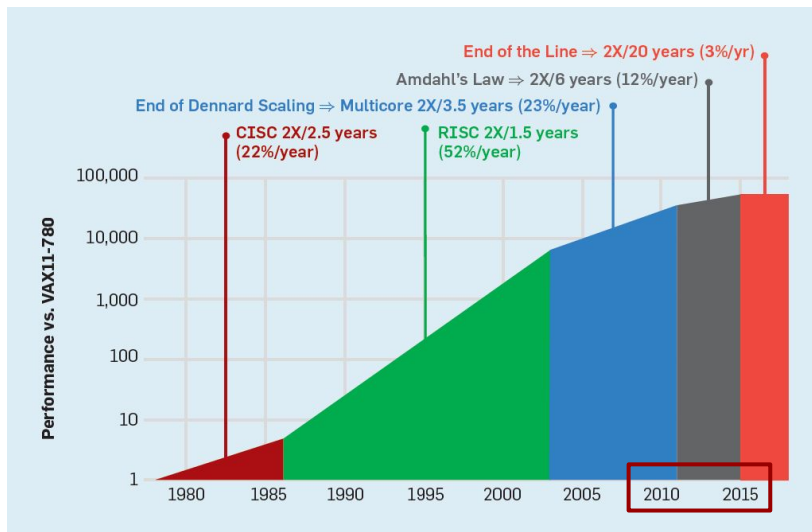
Despite greater performance, polling is commonly believed to be more costly and less energy efficient than interrupt since polling wastes CPU cycles. The primary assumption behind this is that reduced CPU usage directly correlates to reduced power consumption. Therefore, with kernel version 4.10, Linux introduced a hybrid polling mechanism, which sleeps the task before starting to poll so that less CPU cycles are wasted [13].

In this paper, we study the energy implications of the three IO completion mechanisms available in Linux, including interrupts, classic polling, and hybrid polling techniques, specifically for ULL disk IO. Our empirical evaluation using a real ULL device, a power meter, various workload behaviors, and the most recent long-term Linux kernel relies on IO performance measured per energy unit, bytes transferred per byte. Considering both performance and energy in a single metric, we make observations laying out the most energy efficient IO completion mechanisms. We hope that our observations and analysis can lead to more energy efficient storage stack designs in the future.

2 IO COMPLETION IN LINUX KERNEL

In this section, we outline the working mechanisms of available Linux IO completion mechanisms for the most commonly used Linux-native synchronous IO interface.

Rise of Domain-Specific Computing

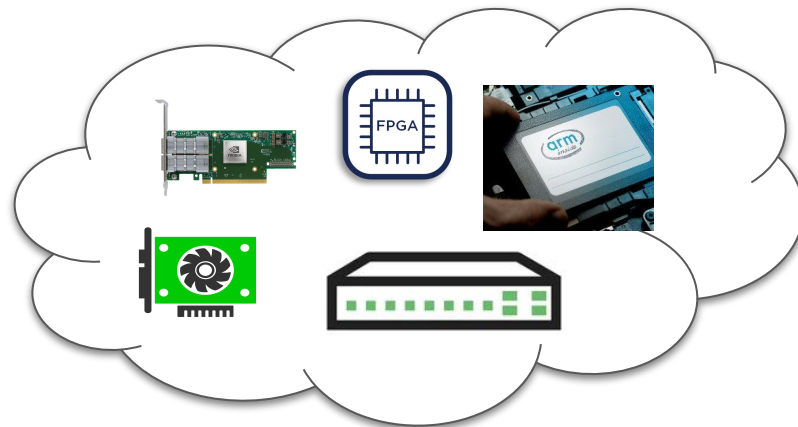


Stalled CPU-centric computing scaling

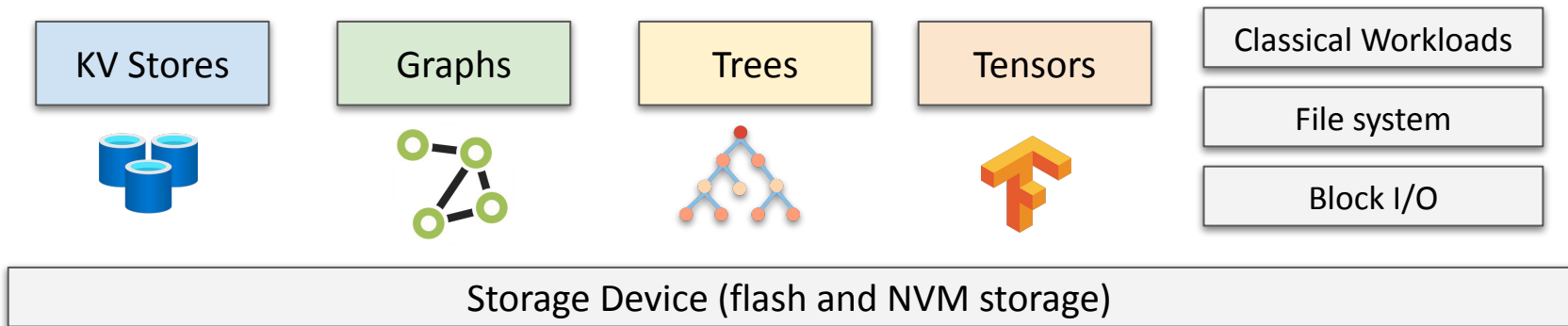


Rise of accelerator-centric computing

- + Specialized hardware
- + Energy/Perf. gains over the CPU



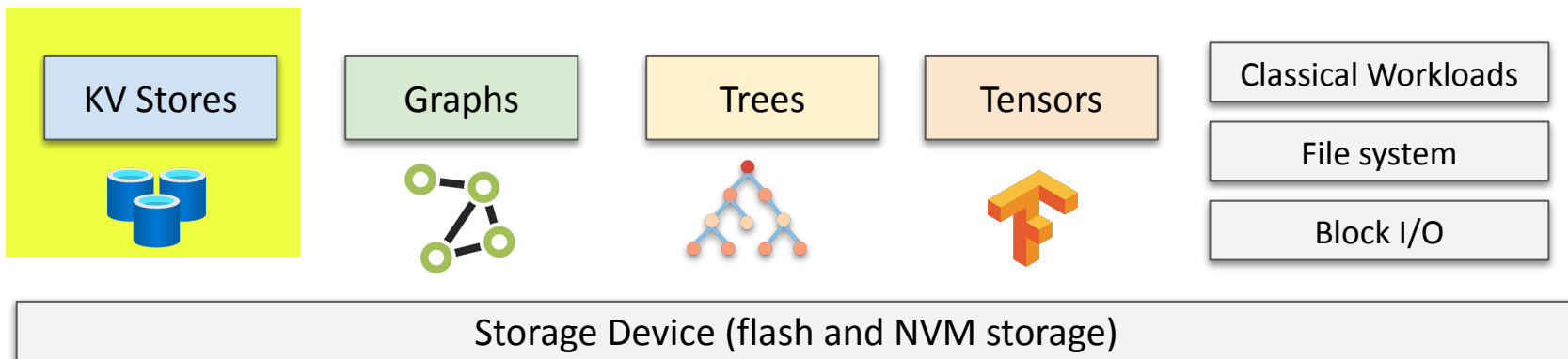
Position: Workload-Specialized Storage Software Will Emerge



Position: Workload-Specialized Storage Software Will Emerge



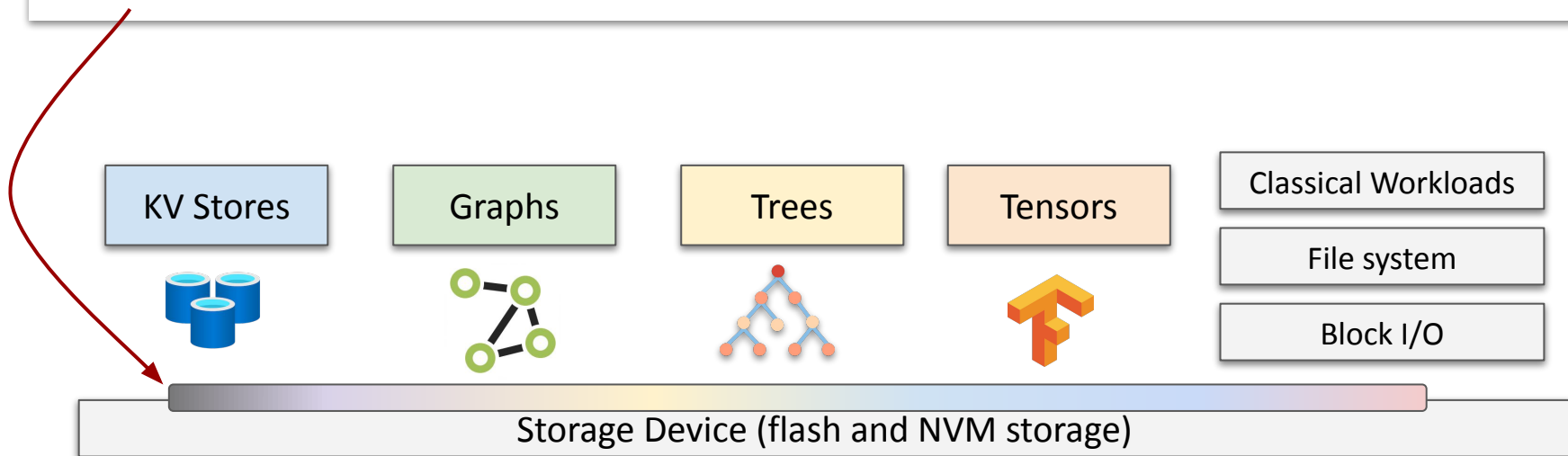
KV-SSDs



Position: Workload-Specialized Storage Software Will Emerge

Performance and scheduling overheads? [Systor'22, CHEOPS'23, ICPE'24 (under submission)]

New Interface: **Zone Namespace** to the rescue? [CLUSTER'23, CCGrid'24 (under submission)]

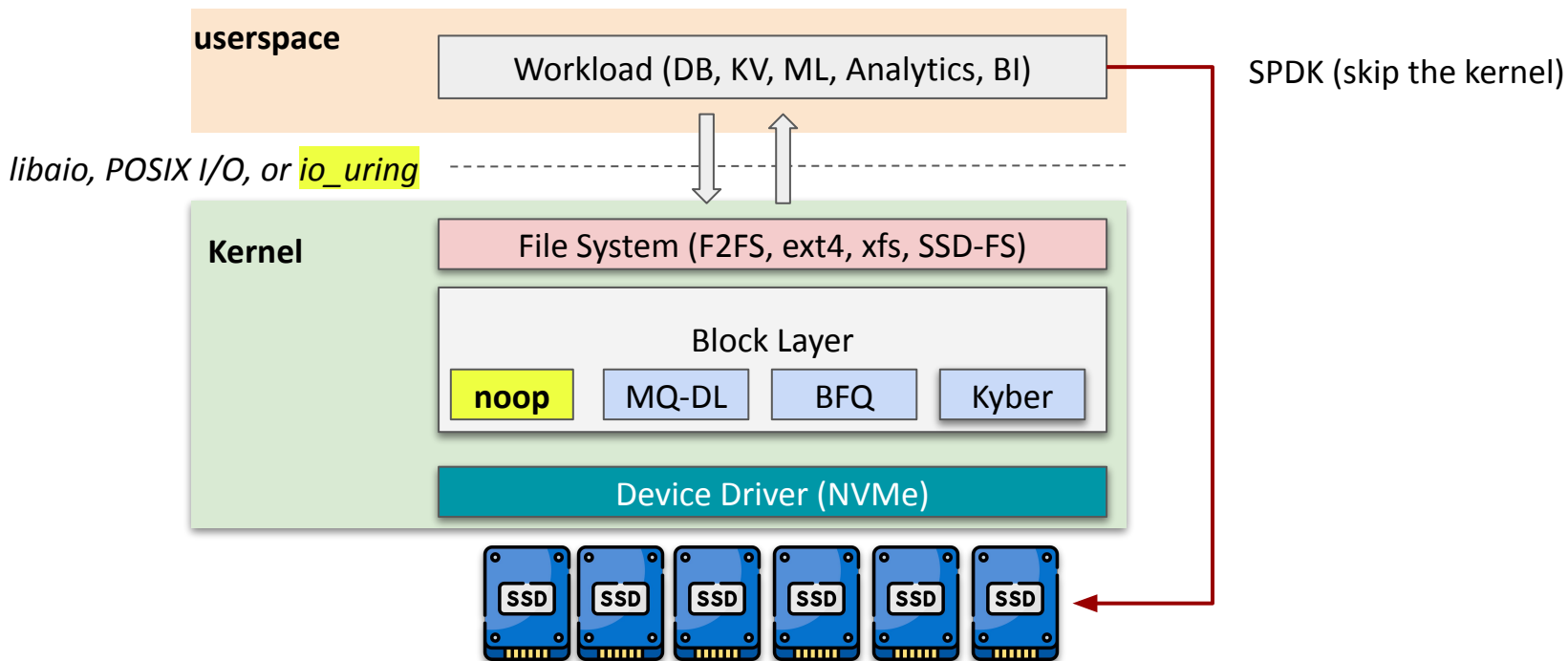


[Part - 1/3] : Performance and Scheduling Challenges

[Part - 2/3] : New Interfaces - Zone Namespace (ZNS) SSDs

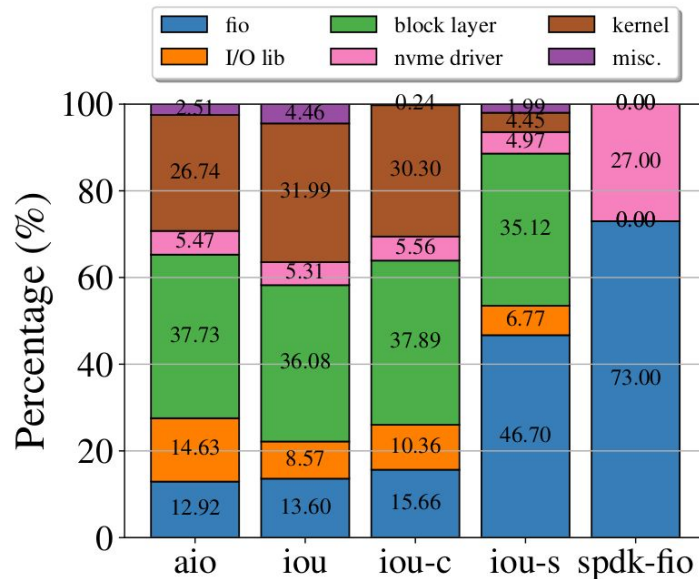
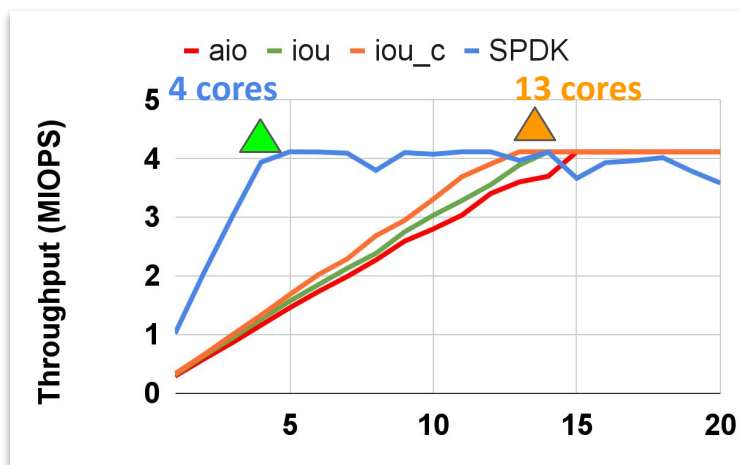
[Part - 3/3] : (WiP) Building Workload-Specialized Storage Stacks

Workload-NVMe Interaction



Results: Pure Performance

CHEOPS'23



There is a large gap (**10x**) in the CPU efficiency between SPDK and io_uring stacks

Linux kernel, with block I/O are the primary consumers of the CPU cycles

So What's Wrong with SPDK?

Takes a pure performance-based approach

Highly CPU inefficient (only poll, 100% CPU utilization)

Scaling performance can be fragile beyond CPU cores

Does not have a file system

Does not have multi-tenancy (only single process)

No support for any other kind of devices except NVMe

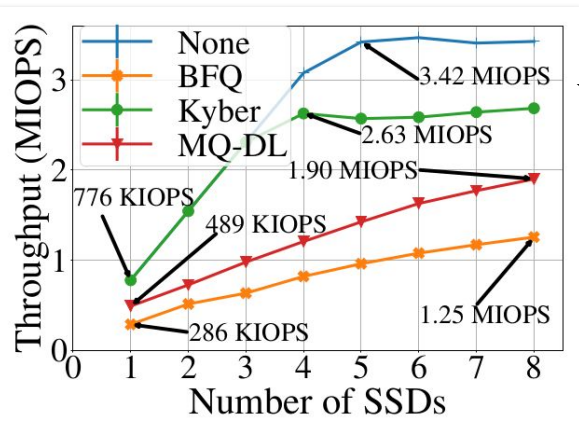
No provision for the kernel supported services:

- Caching, buffering, security
- **Importantly: Sharing and I/O Scheduling**



What are the Scheduling Challenges

(in review) ICPE'24



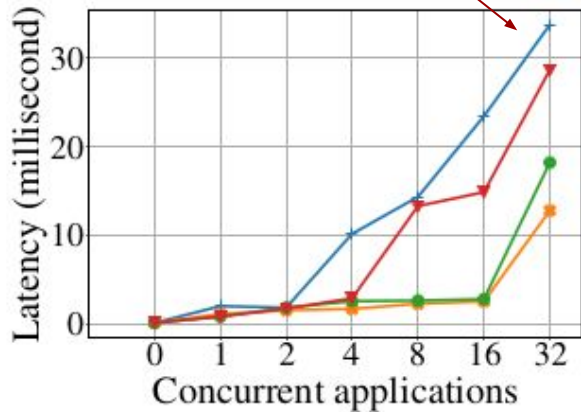
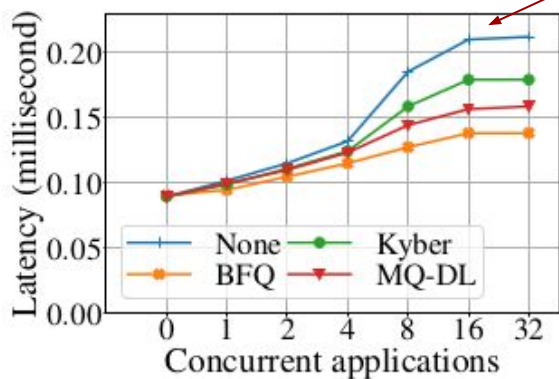
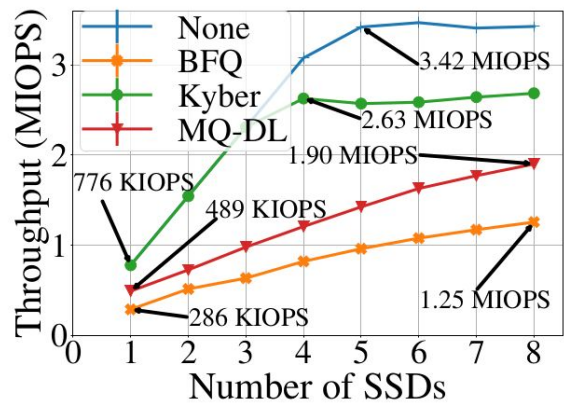
High performance scaling with the none I/O scheduler
1.3 - 2.7x slowdown with other schedulers

(a) IOPS performance of schedulers;

What are the Scheduling Challenges

P95 latencies degradation

(in review) ICPE'24

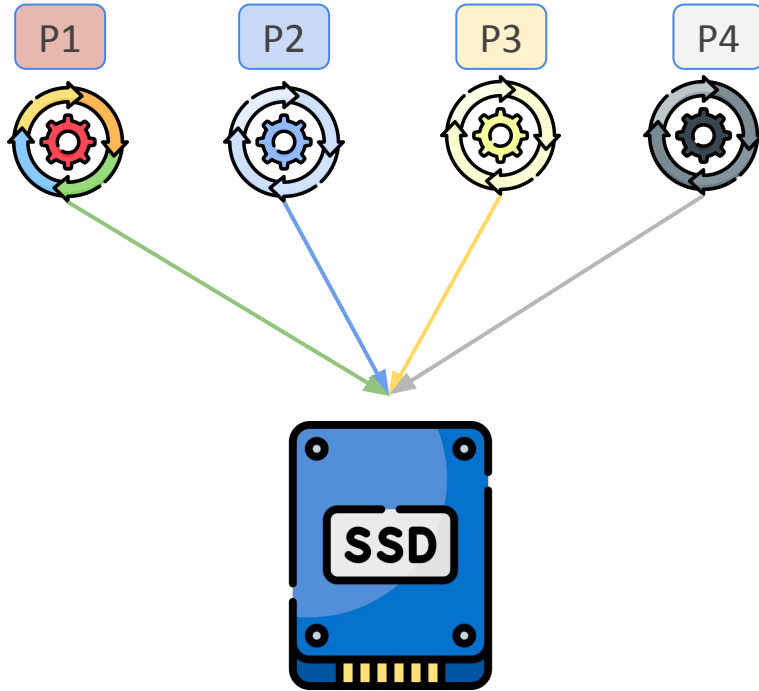


(a) IOPS performance of schedulers;

Latency (P95) with background (b) reads and (c) writes traffic

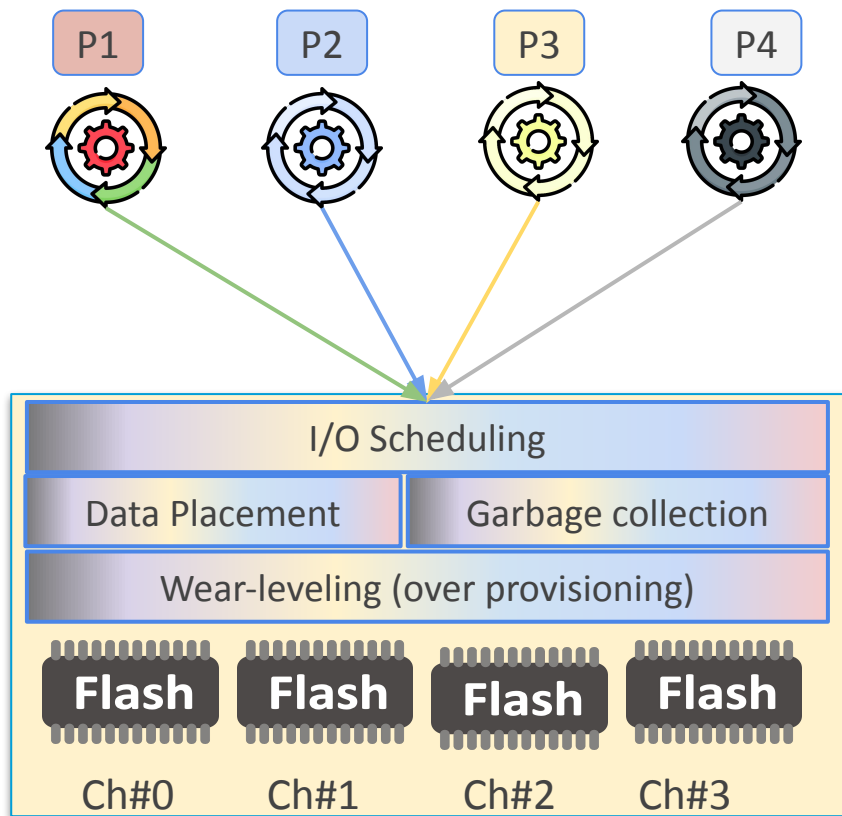
- No scheduling (NOOP) helps with pure performance scaling
- No scheduling (NOOP) has poor performance isolation with interfering tasks

The Interference Control (or Delivering Quality-of-Service)



I/O Scheduling interference and overheads

The Interference Control (or Delivering Quality-of-Service)



I/O Scheduling interference and overheads

Inside an SSD

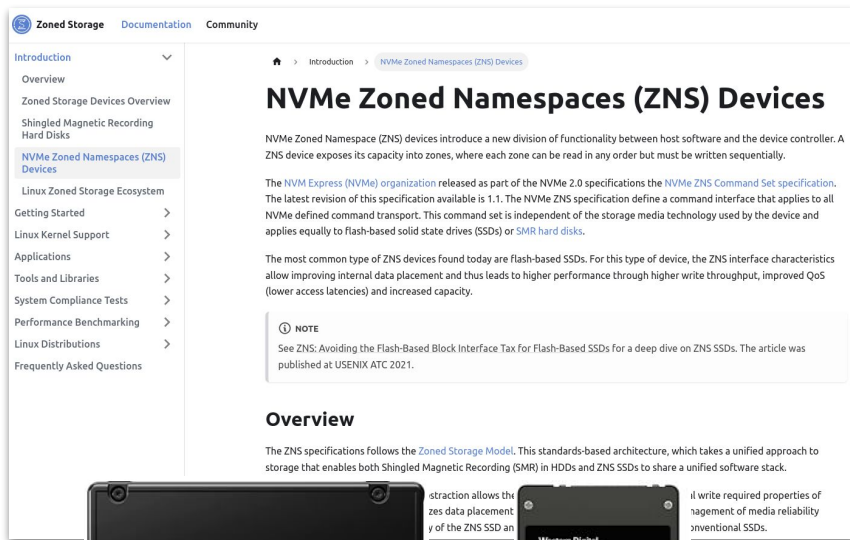
- Mixing of data (lifetime, workloads)
- I/O Scheduling
- Interference from GC
- Over provisioning
- Parallelism management
- ...

~~[Part - 1/3] : Performance and Scheduling Challenges~~

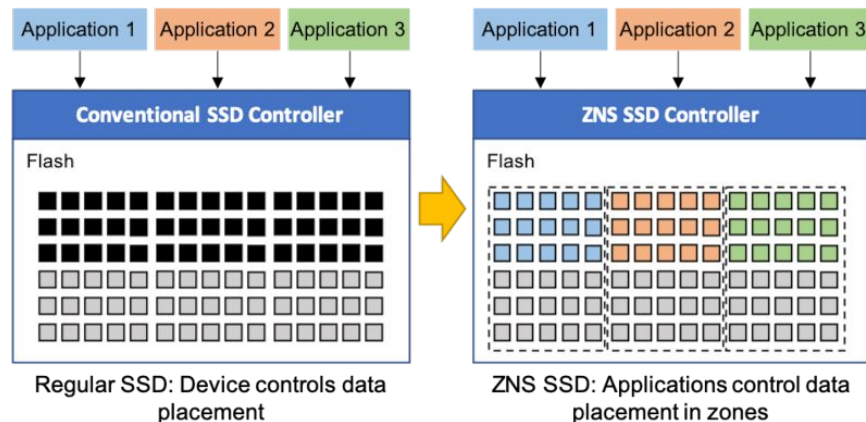
[Part - 2/3] : New Interfaces - Zone Namespace (ZNS) SSDs

[Part - 3/3] : (WiP) Building Workload-Specialized Storage Stacks

ZNS: The New Storage Interface and Capabilities



The screenshot shows the 'Zoned Storage' documentation website. The main heading is 'NVMe Zoned Namespaces (ZNS) Devices'. The text explains that ZNS devices introduce a new division of functionality between host software and the device controller, exposing capacity into zones. It mentions the NVMe Express (NVMe) organization's release of the NVMe ZNS Command Set specification as part of the NVMe 2.0 specifications. A note at the bottom states: 'See ZNS: Avoiding the Flash-Based Block Interface Tax for Flash-Based SSDs for a deep dive on ZNS SSDs. The article was published at USENIX ATC 2021.'



<https://zonedstorage.io/docs/introduction/zns>

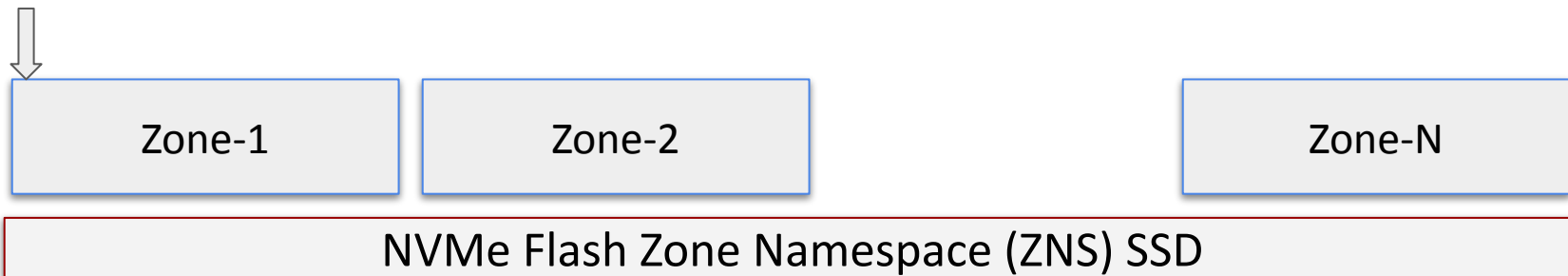
Standardized in the NVMe 1.4, July 2021

Zone Namespace (ZNS) Devices : The Operational Model

A ZNS SSD is divided into Zones

Each zone has its size and a **write pointer**

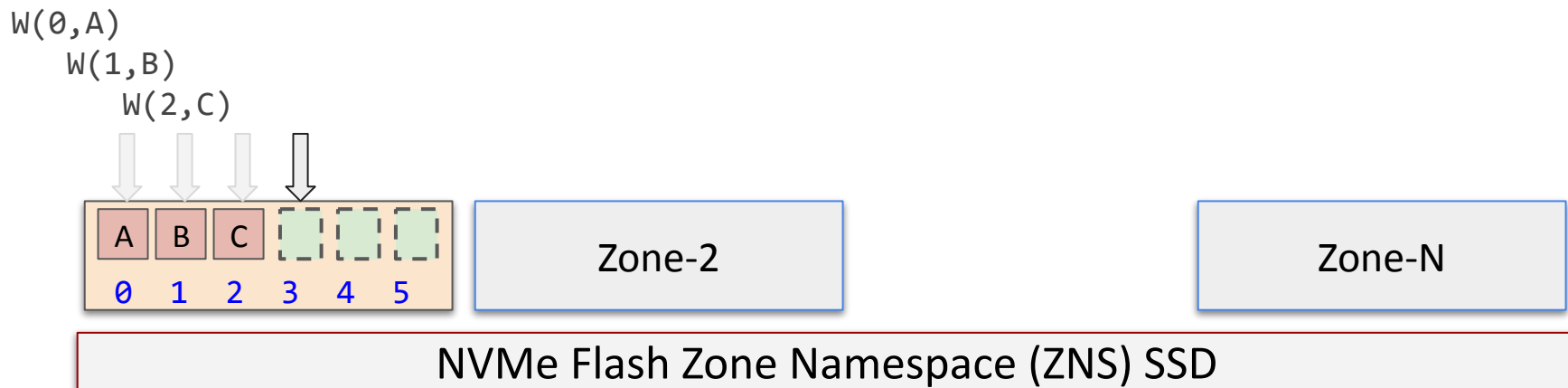
Write pointer



Zone Namespace (ZNS) Devices : The Operational Model

Each zone must be written sequentially

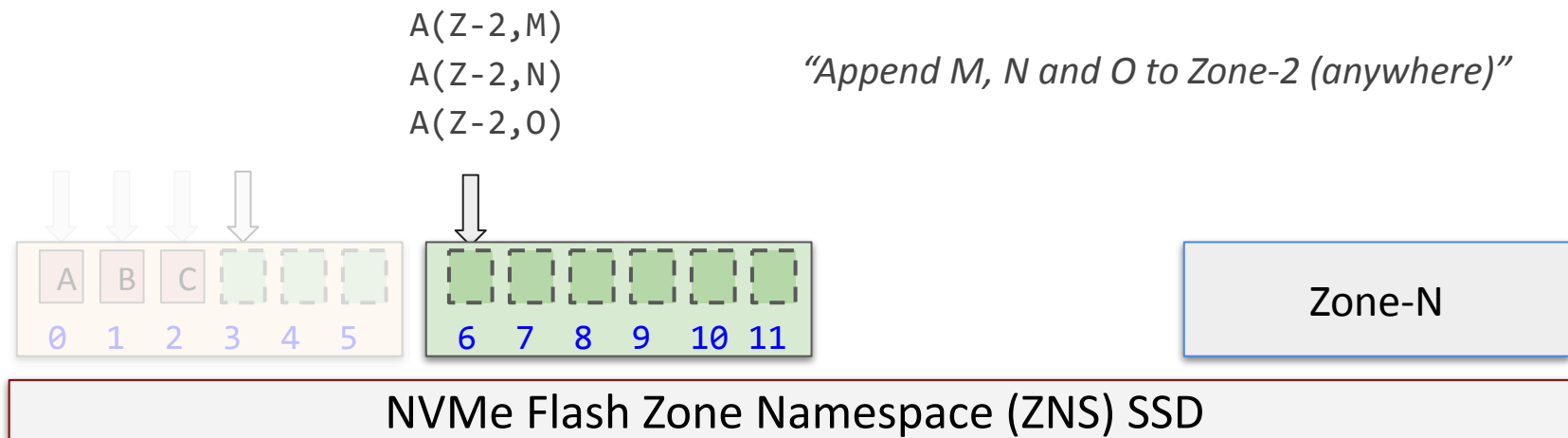
Limited intra-zone parallelism (only 1 write at a time)



Zone Namespace (ZNS) Devices : The Operational Model

New I/O Command: **Append**

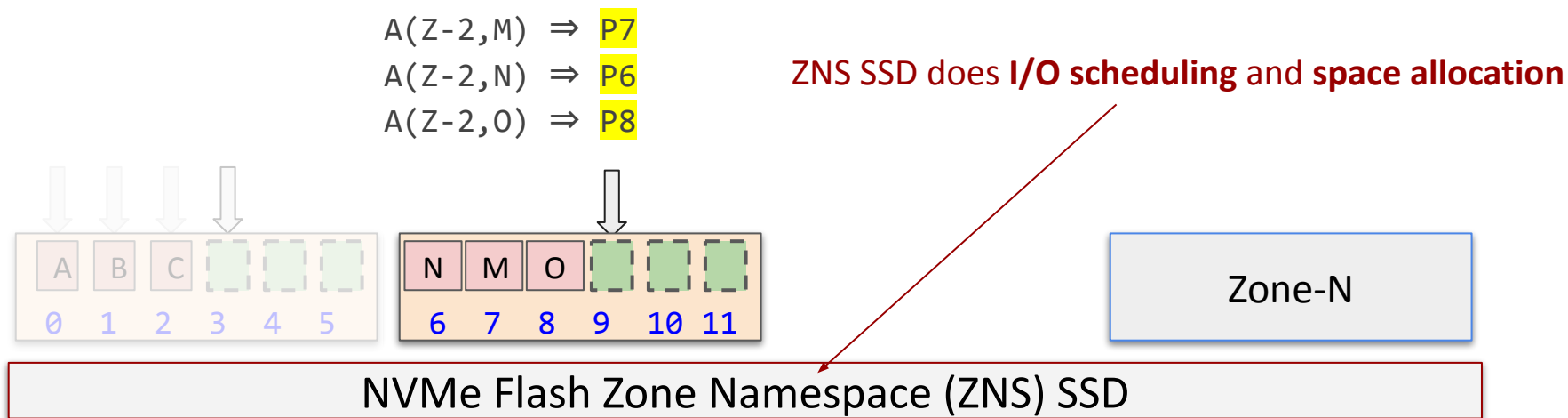
Multiple Append command can be issued to a zone (high intra-zone parallelism)



Zone Namespace (ZNS) Devices : The Operational Model

New I/O Command: **Append**

Multiple Append command can be issued to a zone (high intra-zone parallelism)

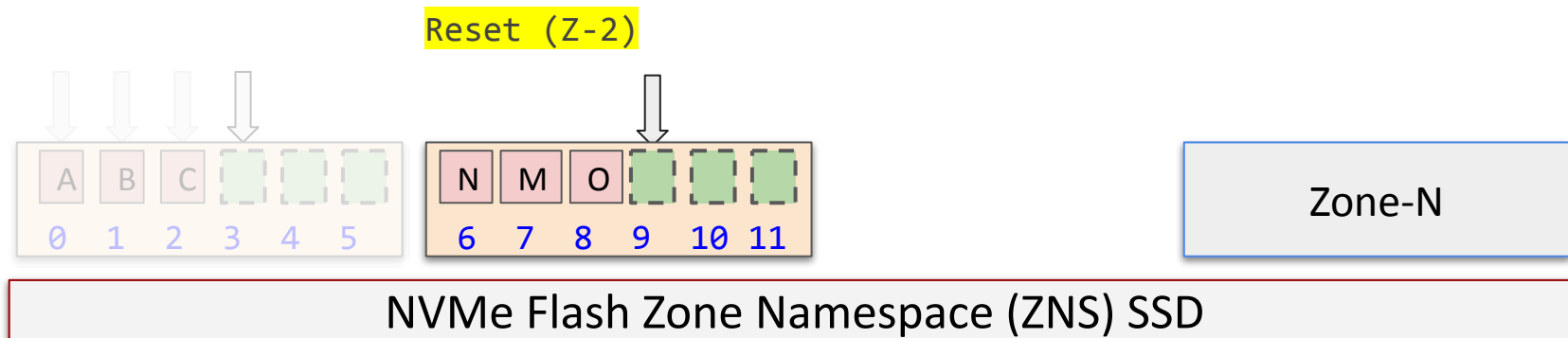


Zone Namespace (ZNS) Devices : The Operational Model

New zone-management commands: **Finish** and **Reset**

Finish: makes it read-only (release write resources)

Reset: garbage collect the zone

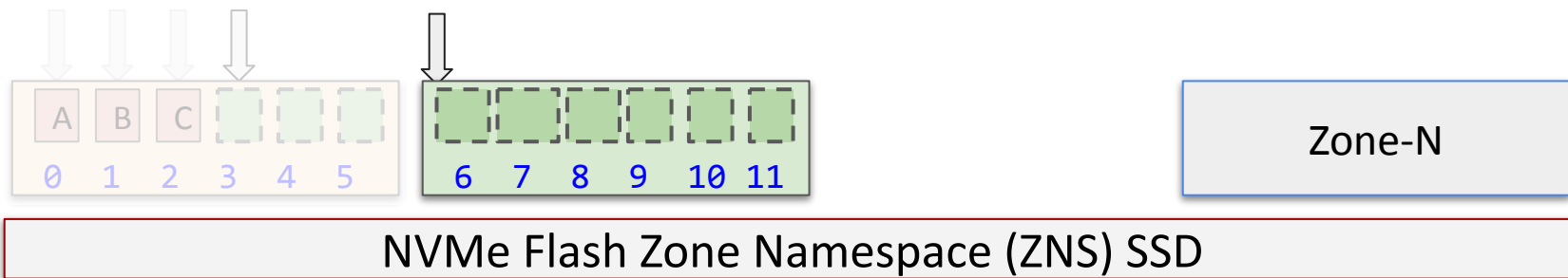


Zone Namespace (ZNS) Devices : The Operational Model

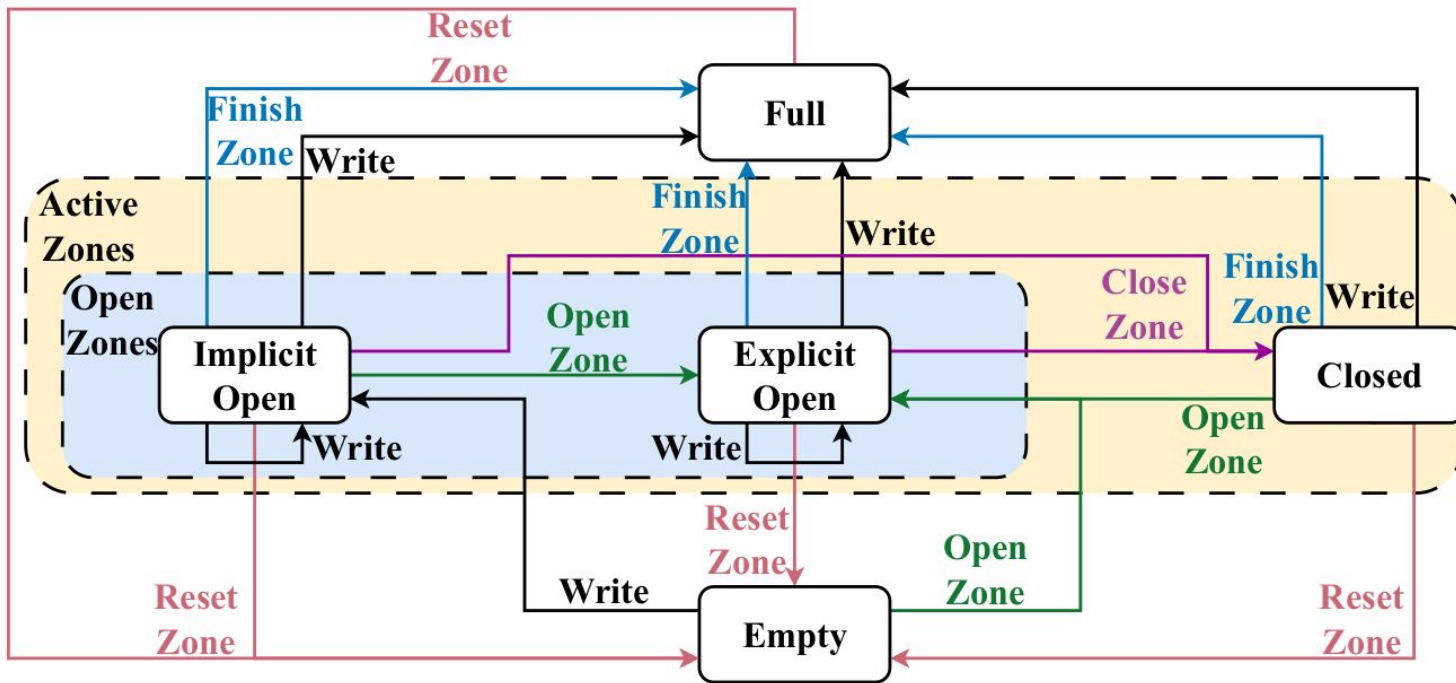
New zone-management commands: **Finish** and **Reset**

Finish: makes it read-only (release write resources)

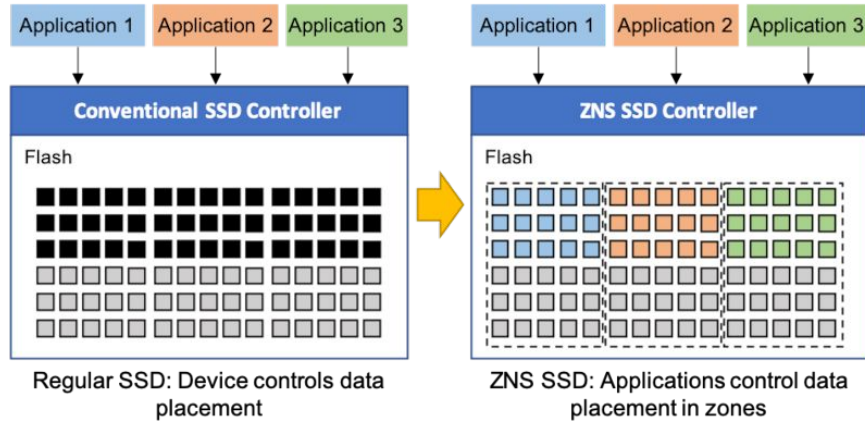
Reset: garbage collect the zone



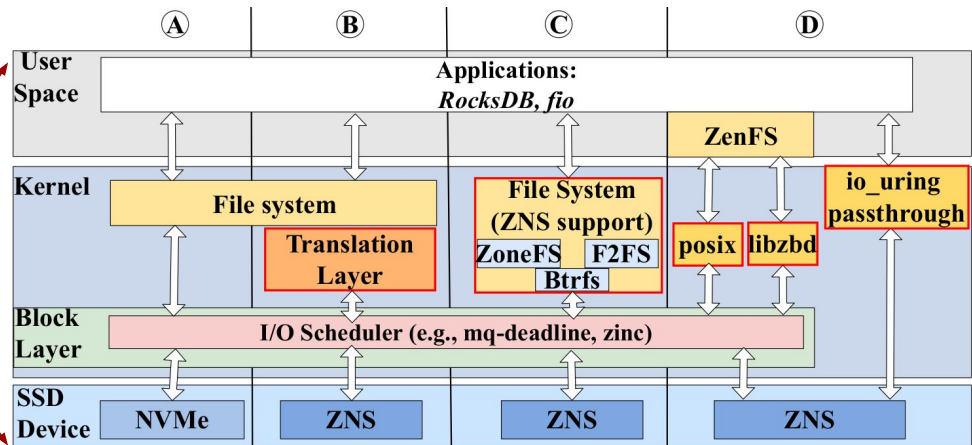
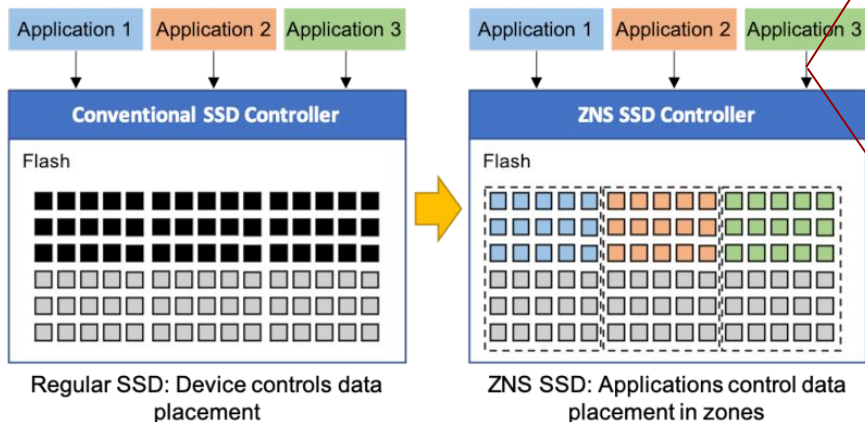
Zone Namespace (ZNS) Devices: The State Machine



State of the ZNS Software



State of the ZNS Software



Understanding NVMe Zoned Namespace (ZNS) Flash SSD Storage Devices,
 Nick Tehrany, Animesh Trivedi, <https://arxiv.org/abs/2206.01547> (2022).

Idea: Different zones helps to isolate workloads from each other and better Quality-of-Service (QoS)

But: There are multiple ways ZNS devices can be integrated

- Should I use **Append** or **Write**? How do I manage **parallelism**? Intra-zone or Inter-zone?
- What is the cost of **Reset** and **Finish**? And the state machine implementation
- Does ZNS deliver on its promise of **isolation**?

Performance Characterization of NVMe Flash Devices with Zoned Namespaces (ZNS)

Krijn Doekemeijer^{1*}, Nick Tehrani^{1,2}, Balakrishnan Chandrasekaran¹, Matias Björling³, and Animesh Trivedi¹

¹Vrije Universiteit Amsterdam, Amsterdam, the Netherlands

²Delft University of Technology, Delft, the Netherlands

³Western Digital, Copenhagen, Denmark

{k.doekemeijer, n.a.tehrani, b.chandrasekaran, a.trivedi}@vu.nl, matias.bjorling@wdc.com

Abstract—The recent emergence of NVMe flash devices with Zoned Namespace support, ZNS SSDs, represents a significant new advancement in flash storage. ZNS SSDs introduce a new storage abstraction of append-only zones with a set of new I/O (i.e., append) and management (zone state machine transition) commands. With the new abstraction and commands, ZNS SSDs offer more control to the host software stack than a non-zoned SSD for flash management, which is known to be complex (because of garbage collection, scheduling, block allocation, parallelism management, overprovisioning). ZNS SSDs are, consequently, gaining adoption in a variety of applications (e.g., file systems, key-value stores, and databases), particularly latency-sensitive big data applications. Despite this enthusiasm, there has yet to be a systematic characterization of ZNS SSD performance with its zoned storage model abstractions and I/O operations. This work addresses this crucial shortcoming. We report on the performance features of a commercially available ZNS SSD (13 key observations), explain how these features can be incorporated into publicly available state-of-the-art ZNS emulators, and recommend guidelines for ZNS SSD application developers. All artifacts (code and data sets) of this study are publicly available at <https://github.com/stonet-research/NVMeBenchmarks>.

Index Terms—Measurements, NVMe storage, Zoned Namespaces Devices

I. INTRODUCTION

The emergence of fast flash storage in data centers, HPC, and commodity computing has fundamentally caused changes in every layer of the storage stack, and led to a series of new developments such as a new host interface (NVMe Express, NVMe) [1], [2], [3], a high-performance block layer [4], [5], [6], [7], new storage I/O abstractions [8], [9], [10], [11], [12], [13], [14], and re/co-design of storage application stacks [15], [16], [17], [18], [19], [20], [21]. Today, flash-based solid-state drives (SSDs) can support very low latencies (i.e., a few microseconds), and multi GiB/s bandwidth with millions of I/O operations per second [22], [23], [24].

Despite these advancements, the conceptual model of a storage device remains unchanged since the introduction of hard disk drives (HDDs) more than half a century ago. A storage device supports only two necessary operations: write and read data in units of *sectors* (or blocks) [25]. Data can be read from and written to anywhere on the device, hence

supporting random and sequential I/O operations. Though this model works with conventional HDDs, it is not apt for flash-based storage devices as flash internally does not support overwriting data [26], [27], [28]. Flash devices offer the illusion of “overwritable” storage via the *flash translation layer* (FTL), a software component that runs within the device. The FTL enables easy integration of flash devices (by allowing them to masquerade as fast HDDs), albeit it introduces unpredictability in performance [29], [30], [31], [32], [33], [34] and complicates device lifetime management [35]. These challenges are defined as the *unwritten contracts* of SSDs [26]. As data centers have largely transitioned to SSDs for fast, reliable storage [36], [37], and modern big data applications have high QoS demands [38], [39], there is a dire need to address these unwritten contracts.

Researchers and practitioners advocate for open flash SSD interfaces beyond block I/O [40] to address these challenges. Examples include *Open-Channel SSDs* (OCSSD) [41], *multi-stream SSDs* [9], and, more recently, *Zoned Namespaces* (ZNS) [11]. The focus of this work is on NVMe devices that support ZNS, which are commercially available today [42], [43]. ZNS promises a low and stable tail latency [11] and a high device longevity, and, hence, addresses the needs of modern big data workloads. There is, unsurprisingly, a rich body of active and recent work on ZNS [44], [11], [45], [46], [47], [48], [49], [50], [51], [52], [53], [54], [55], [56]. Despite this enthusiasm, there has not been a systematic performance and operational characterization of ZNS SSDs. This lack of an extensive performance and operational characterization of ZNS SSDs severely limits the utilization and application of ZNS devices in big data workloads. In this work, we bridge this gap by presenting the performance characterization of a commercially-available NVMe ZNS device.

We complement this characterization of a physical device with an investigation of emulated ZNS devices, since they are widely used in research [51], [57], [58], [55]. Emulated devices enable researchers to explore the ZNS design space without being constrained by device-specific characteristics. Such unconstrained explorations are crucial since ZNS is a new interface and the selection of available configurations in a real SSD is, unsurprisingly, quite limited. The research validity of all of these works hinge on an emulator’s ability to mimic

ZINC - A ZNS Interference-aware NVMe Command Scheduler

^{1st} Nick Tehrani

Computer Science

Vrije Universiteit Amsterdam

Amsterdam, The Netherlands

n.a.tehrani@vu.nl

^{2nd} Krijn Doekemeijer

Computer Science

Vrije Universiteit Amsterdam

Amsterdam, The Netherlands

k.doekemeijer@vu.nl

^{3rd} Zebin Ren

Computer Science

Vrije Universiteit Amsterdam

Amsterdam, The Netherlands

z.ren@vu.nl

^{4th} Animesh Trivedi

Computer Science

Vrije Universiteit Amsterdam

Amsterdam, The Netherlands

a.trivedi@vu.nl

Abstract—NVMe Zoned Namespaces (ZNS) is a new NVMe standard designed to open up the rigid block-based host-device interface and offer a new zone-based device interface to host software. ZNS introduces a set of new I/O (append) and flash management (*reset*, *finish*, *open*, *close*) commands to host software (i.e., block layer, file system, or application). The flash management commands allow managing flash-based SSDs directly and offer a possibility for better interference management between flash- and user-issued I/O operations. In this paper, we demonstrate that, despite ZNS’s promises, its new commands create complex interference patterns with I/O and each other that lead to significant losses in application performance. We introduce a first-of-its-kind interference model for ZNS and use this model to report 3 interference observations made on a physical ZNS SSD. Based on our interference study, we propose ZINC, a ZNS interface-aware NVMe command scheduler that mitigates the impact of interference by prioritizing user I/O commands over flash management commands (configurable). ZINC delivers up to 56.87% lower interference in fio-based micro-benchmarks compared to the state-of-the-practice *mq-deadline*, and a 9.81% throughput improvement for RocksDB + ZenFS, a ZNS-enabled KV-store for ZNS SSDs. With concurrent *reset* operations, RocksDB’s throughput degrades from 80 KIOPS to 72 KIOPS with *mq-deadline*, but remains at 80 KIOPS with ZINC. We open-sourced ZINC at <https://anonymous.openscience/rzinc>

I. INTRODUCTION

The emergence of *solid-state drives* (SSDs) has presented a significant advancement in storage technology over prior technologies (e.g., hard disk drives), with modern SSDs capable of achieving millions of I/O operations per second, gigabytes of bandwidth per second, and sub-microsecond access latencies [40]. Despite their popularity [41], a perennial challenge with flash-based SSDs is delivering both predictable and consistent performance to a variety of workloads. The internal structure of flash-based SSDs is at the root of this challenge—flash requires significant management effort [31] (i.e., location mapping, garbage collection, parallelism management, bad block, and ECC) and flash management is typically only done in firmware, known as the *flash-translation layer* (FTL) running within SSDs [28]. Traditional block-based NVMe SSDs (non-ZNS) have an interface that only accounts for block-based NVMe *read* and *write* commands. This interface does not offer any insight or host-level control over data placement or garbage collection [29], these are all done inside of the SSD, run on the background and can be issued at

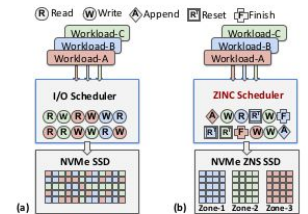


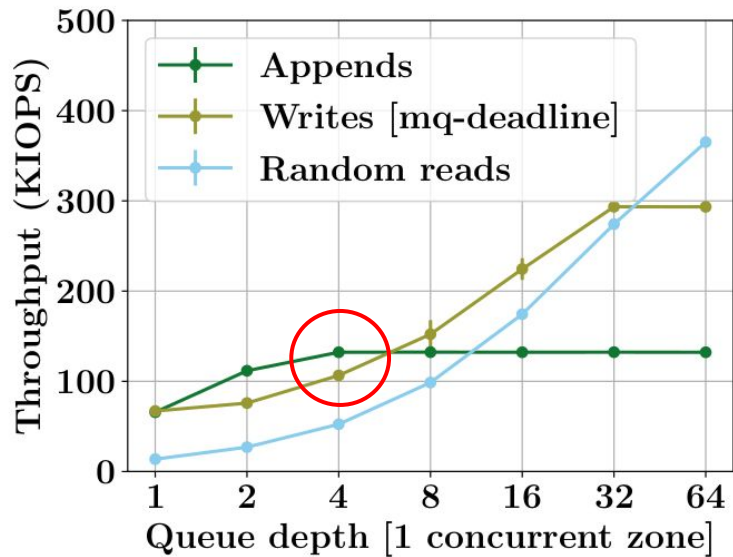
Figure 1: Conventional (a) NVMe vs. (b) ZINC scheduling.

any point in time. Consequently, when the firmware inside the SSD executes these background management commands, these commands interfere with the foreground host-issued I/O commands. There has been a series of efforts to curtail the impact of the interference (collectively termed as *Unwritten Contracts* [17]) with better resource allocation, scheduling techniques [3], [10], [13], [26], [34] and even host-device interfaces such as Open-Channel SSDs [23], StreamSSDs [13], and more recently, Zoned Namespaces (ZNS) SSDs [2], [42]. Among them, ZNS has attracted a significant amount of research interest [8], [9], [15], [20], [21], [33], [36] and has become an industry-standard with the NVMe 2.0 specification.

The unique aspect of NVMe ZNS devices is that they offer a new I/O abstraction—append-only zones—with a set of new I/O (append) and zone management (*reset*, *finish*, *open*, *close*) commands. Zone management commands closely imitate how flash chips are managed internally and offer more direct control over garbage collection and data placement within zones to the host systems software (i.e., block layer, file systems, or applications; see [8]). The zone interface provides opportunities, as the host system software now controls data placement by explicitly identifying which zone to store data in, thus following the “Grouping by Death Time” unwritten contract [17], [39]. Further on, the host also implicitly controls garbage collection, as a zone is the unit of garbage collection—a host can decide when a zone needs to be reclaimed by issuing an explicit ZNS command

*Equal contributions, joint first authors. Nick was with TU Delft during this work.

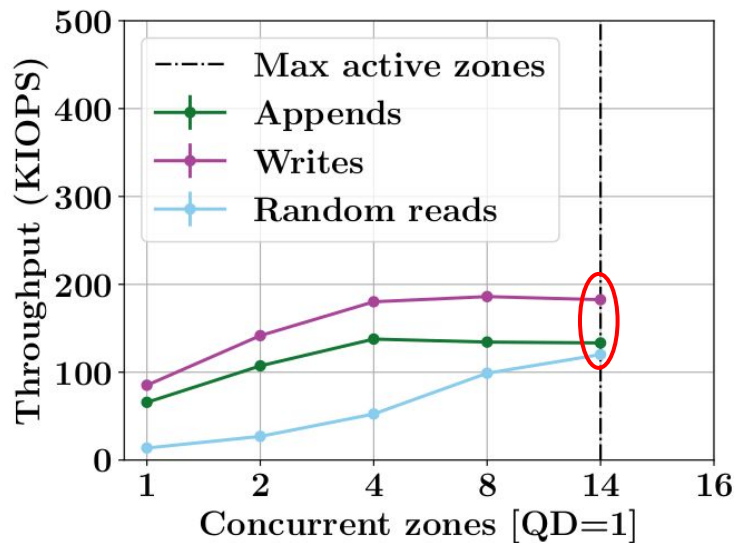
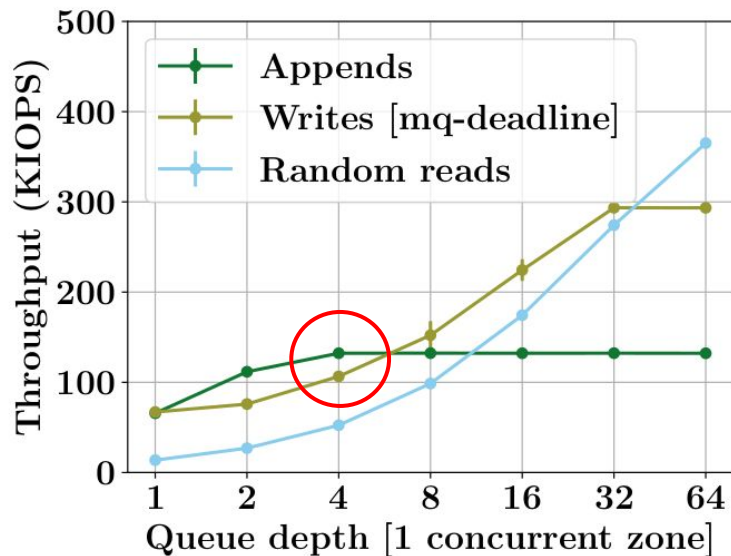
Result [1 / 3]: Write vs Append Parallelism Management



mq-deadline merges adjacent writes

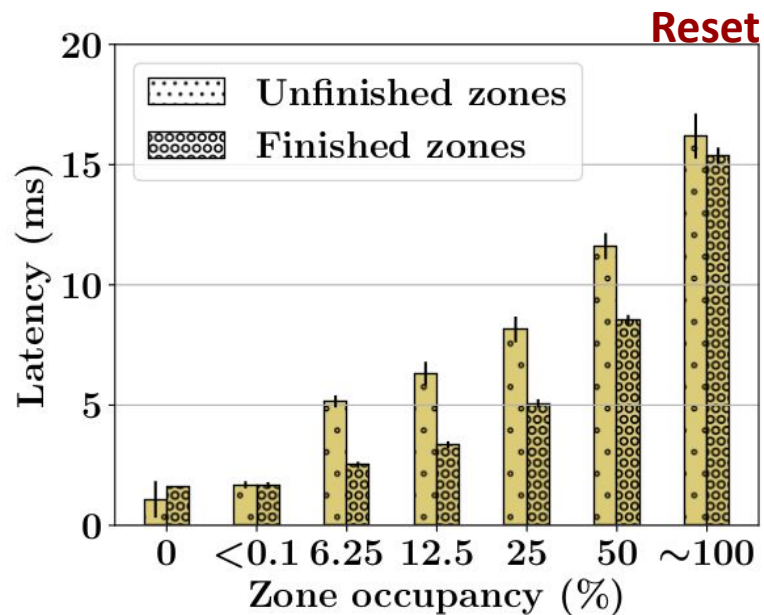
Single Zone Parallelism (intra-zone)

Result [1 / 3]: Write vs Append Parallelism Management

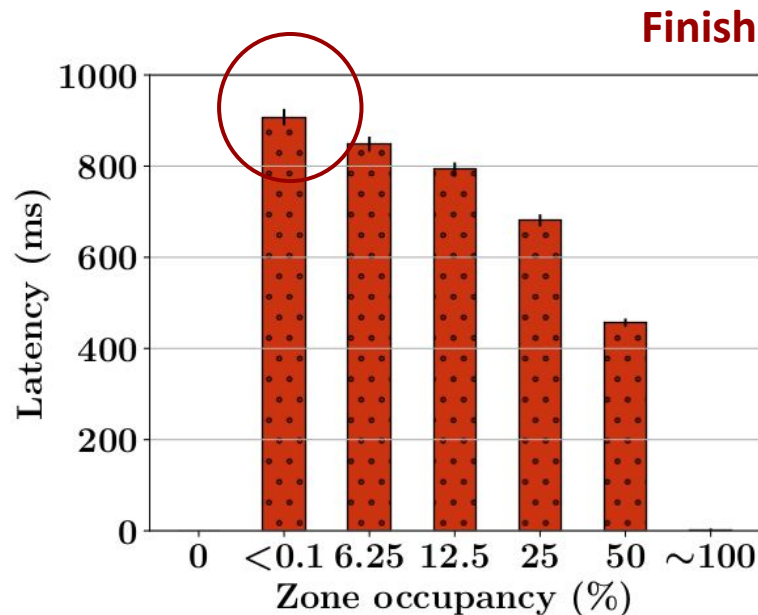
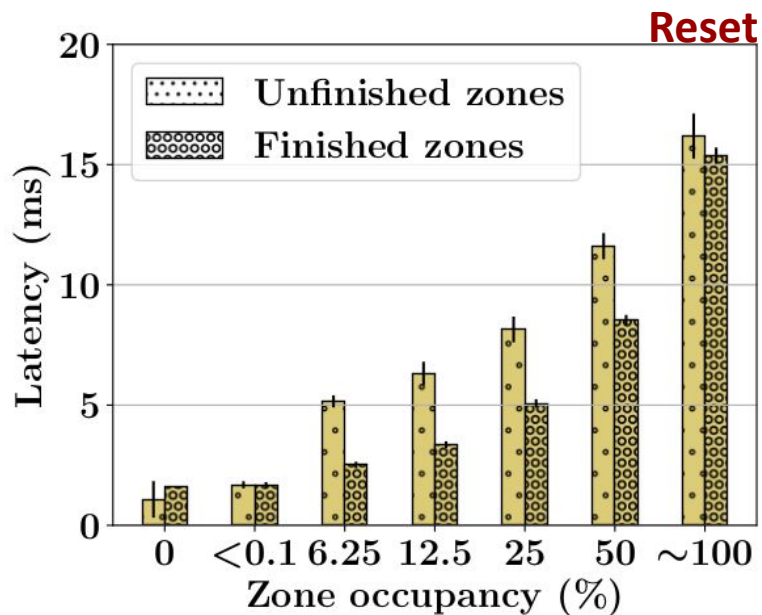


- Intra-Zone parallelism has higher performance
- Writes have better performance scalability than Appends (!)
- Append scalability is independent of intra- or inter-zone, but limited in performance

Result [2 / 3]: The Cost of Reset and Finish Operations

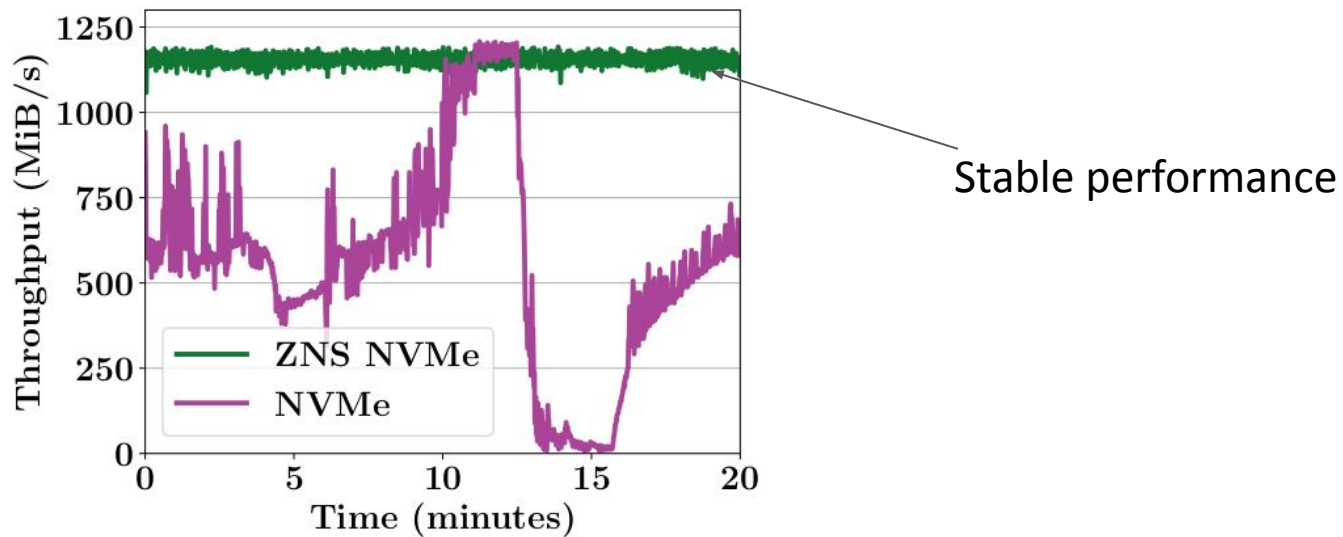


Result [2 / 3]: The Cost of Reset and Finish Operations



- The zone utilization --- Very important factor
- Finish is an extremely expensive operation (100 - 1,000s of milliseconds)
- Leverage intra-zone parallelism (*minimize half-written zones*)

Result [3 / 3]: Read-Write Isolation on ZNS



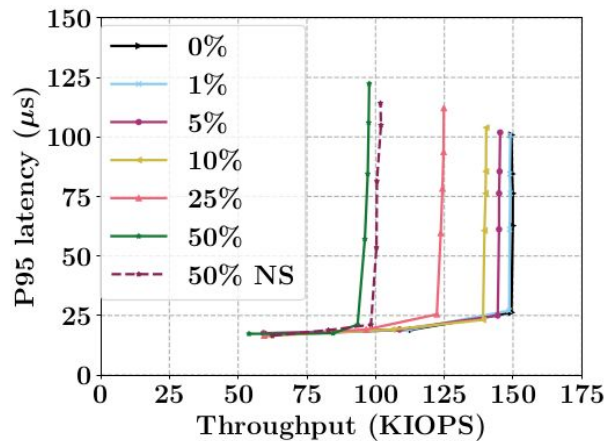
- ZNS provides good read-write isolation when operating on multiple zones
- Stable performance (in comparison to NVMe)

BUT WAIT

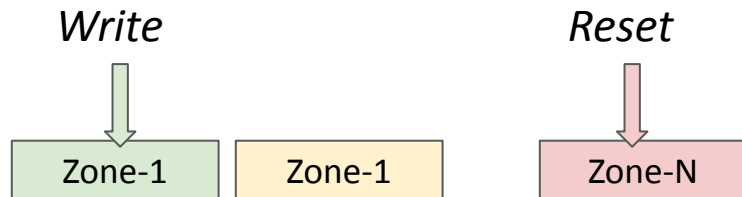


THERE'S MORE

New Interference: Reset on I/O Operations

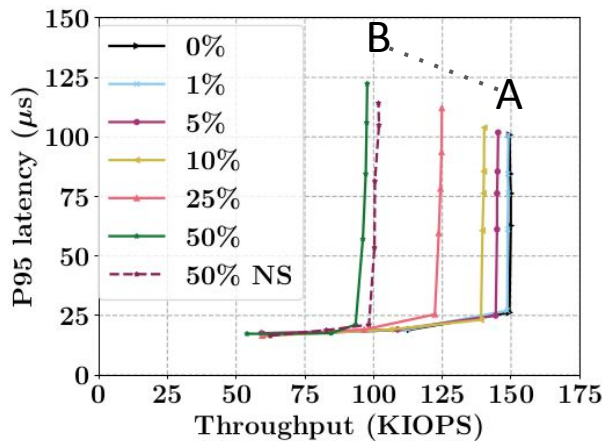


(a) on *write* with *inter-zone* concurrency.



Concurrent resets with a controlled rate on a different zone

New Interference: Reset on I/O Operations



(a) on *write* with *inter-zone* concurrency.

B(x2, y2)

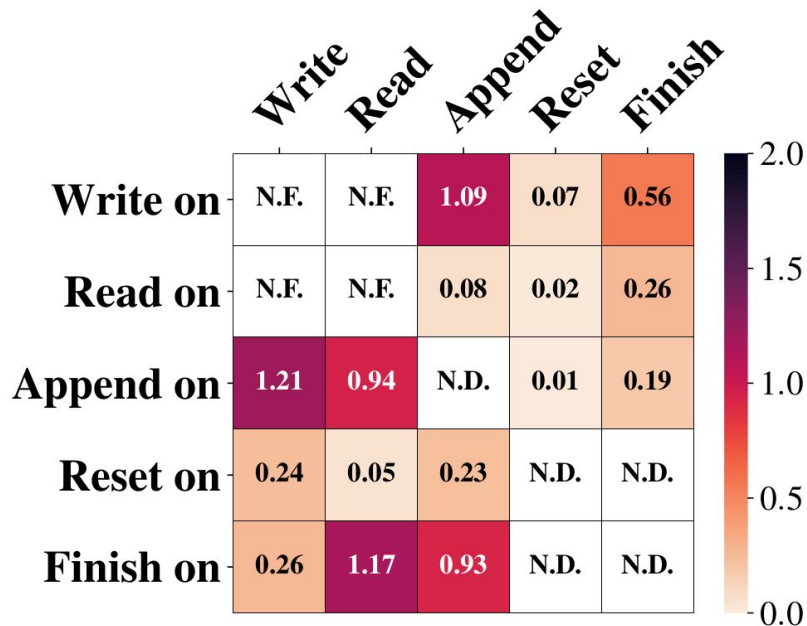


A(x1, y1)



Modeling and **quantifying interference** with a first-order Earth Mover's Distance (EMD)-style model

Interference Results : Micro- and Workload-level



RocksDB

0% Reset	50% Reset	
78.9 KIOPS	72.1 KIOPS	-8.7% drop

Workload-level interference!

ZINC: Zone-Interface aware NVMe ~~H/O~~ Command Scheduler

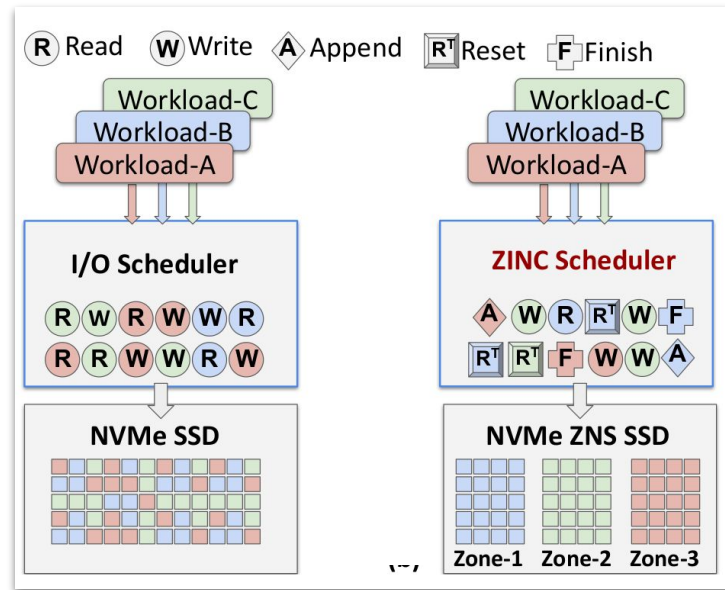
All NVMe commands need scheduling for QoS

ZINC is derived from mq-deadline scheduler with a Kyber-style Reset-throttling logic

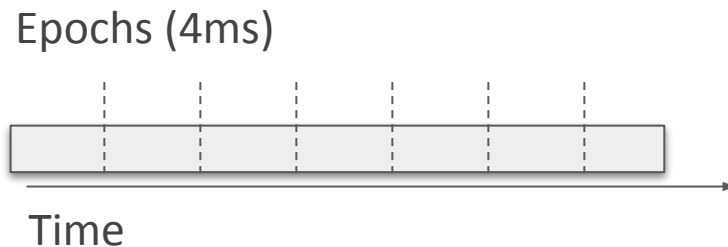
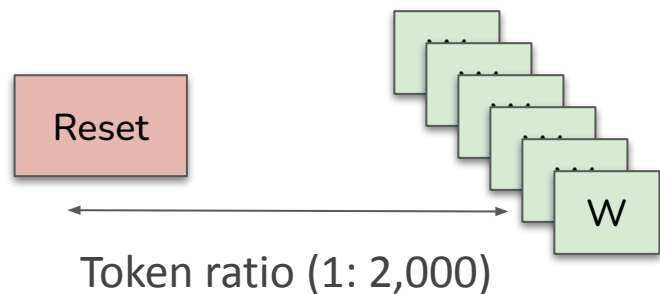
Extra code to connect the io_uring passthrough commands to the I/O scheduler in the Linux block layer

Open source based on v6.3

The paper is under review



The Design of ZINC

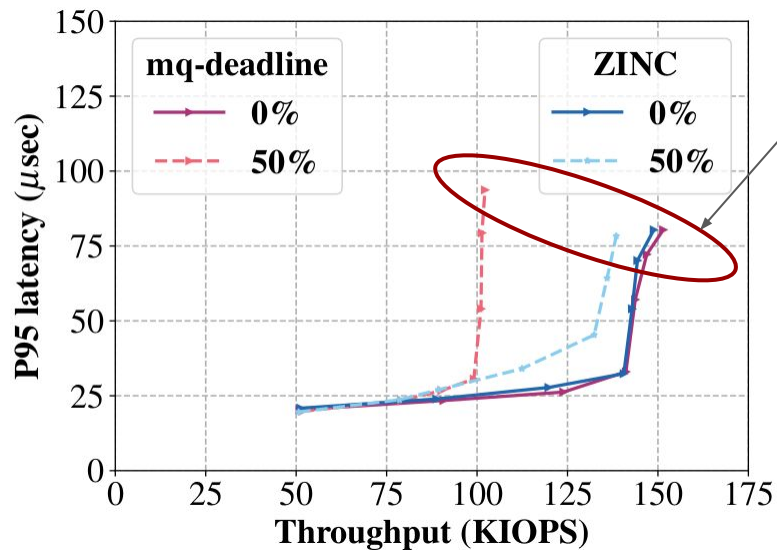


The decision process in the each epoch:

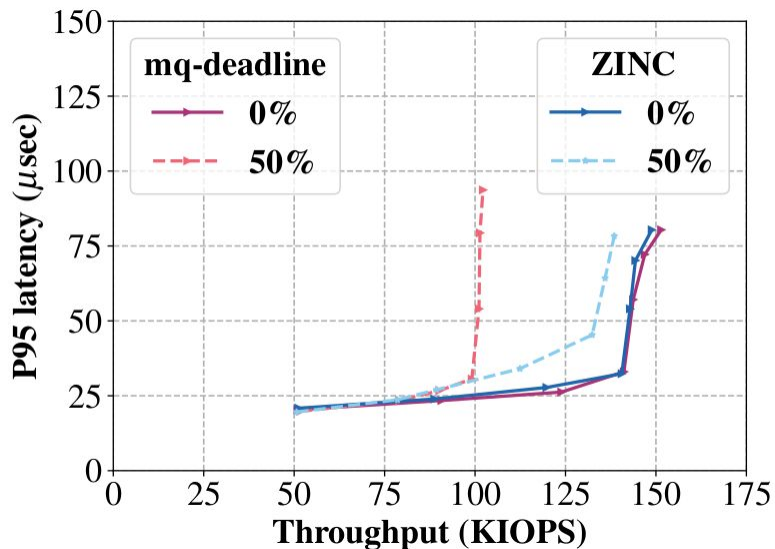
- (1) Are all the write tokens consumed? If yes, then issue the Reset command to the ZNS
- (2) If an epoch is reached and the Reset command is still held, then increase its priority
- (3) In any epoch if the Reset command has more priority than "x" (configurable count), then issue it immediately

Impact of using ZINC

Controlled degradation



Impact of using ZINC



RocksDB

	0% Reset	50% Reset
MQ-D	78.9 KIOPS	72.1 KIOPS
ZINC	78.2 KIOPS	80.0 KIOPS

~11% gain

- ZINC helps to control the interference between NVMe I/O and NVMe zone-management commands
- ZINC helps to deliver workload-level performance gains

~~[Part - 1/3] : Performance and Scheduling Overheads~~

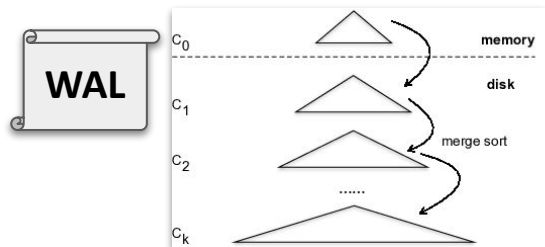
~~[Part - 2/3] : New Interfaces - Zone Namespace (ZNS) SSDs~~

[Part - 3/3] : (WiP) Building Workload-Specialized Storage Stacks

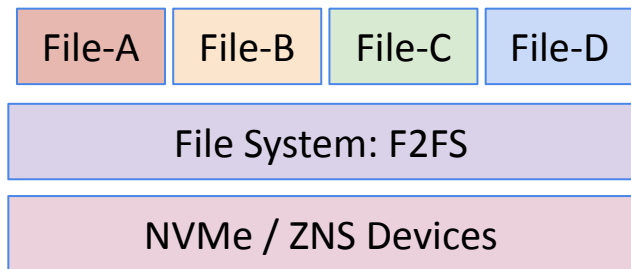
Constructing an End-to-End Picture



RocksDB



Workload



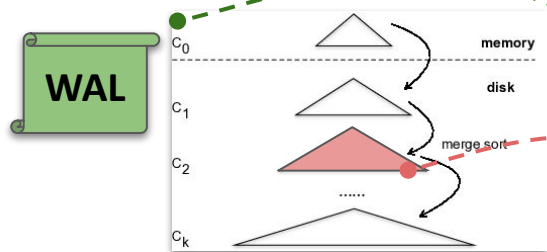
Storage stack

Constructing an End-to-End Picture



RocksDB

The WAL writing is more important than the L7 compaction



WAL

File-A

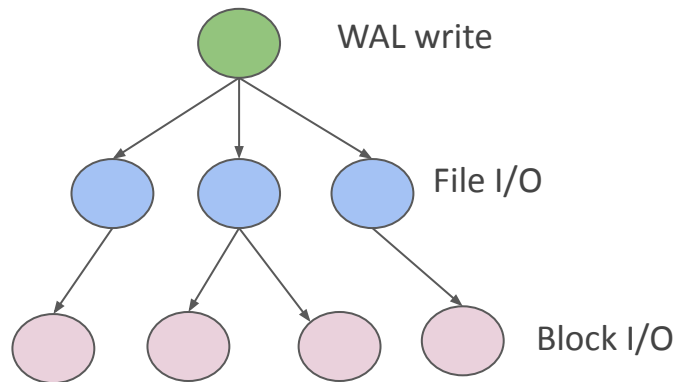
File-B

File-C

File-D

File System: F2FS

NVMe / ZNS Devices

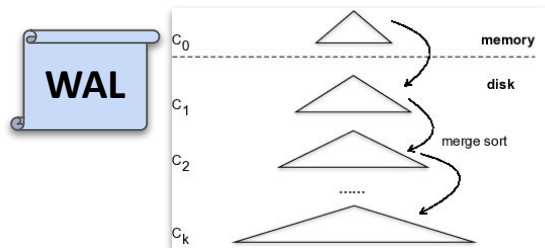


These two block I/O operations are not equal for the workload
The LBA address `0x5ABDE345` belongs to what?

Constructing an End-to-End Picture



RocksDB



File-A

File-B

File-C

File-D

File System: F2FS

NVMe / ZNS Devices



ZNS-Tools: e-BPF powered whole-stack tracing framework

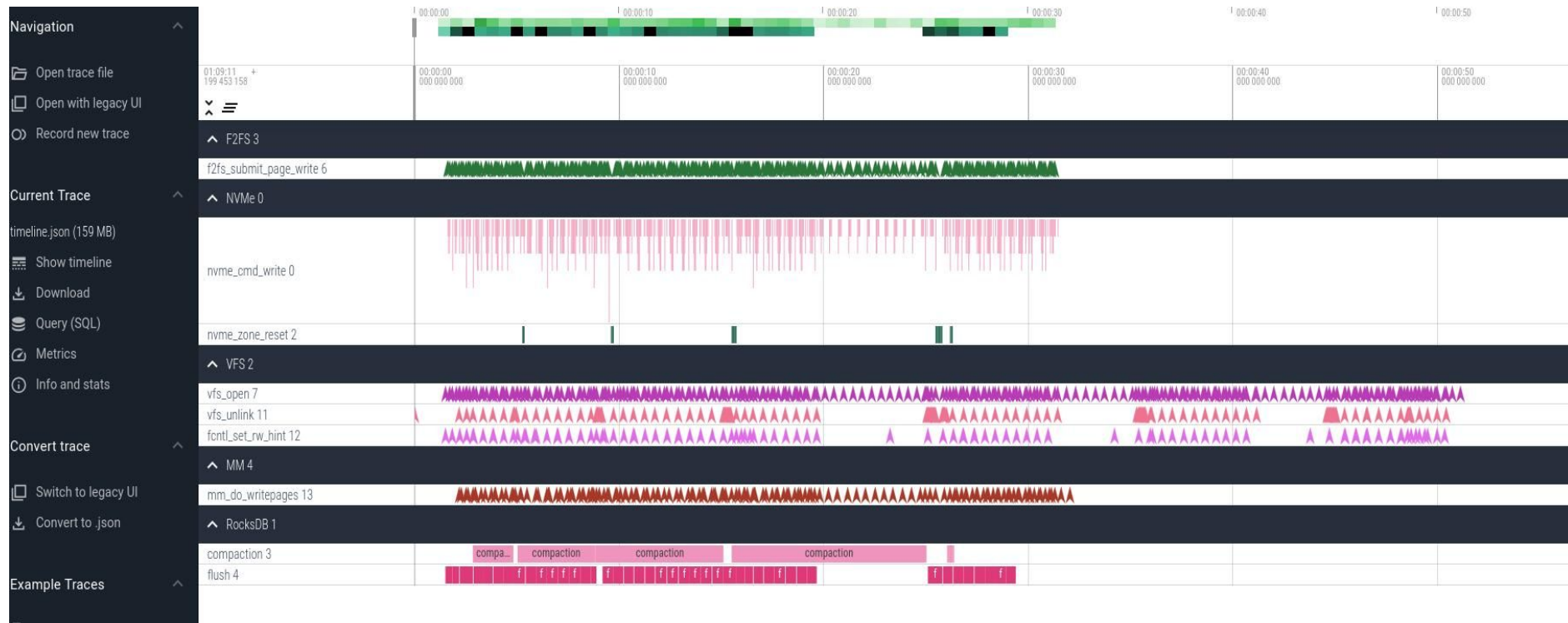
Collects traces for workload-level data operations

- **Workload:** RocksDB (WAL, compaction, GC)
- **File system:** F2FS (log, GC)
- **Block layer:** ZNS (read, write, reset scheduling)
- **Device Driver:** NVMe (command issuing)

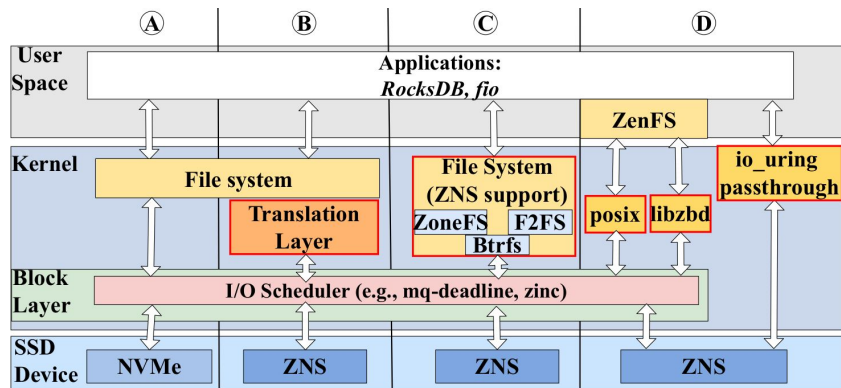
Builds offline location and movement profile

<https://github.com/stonet-research/zns-tools>

ZNS-tools: I/O and Trace visualization



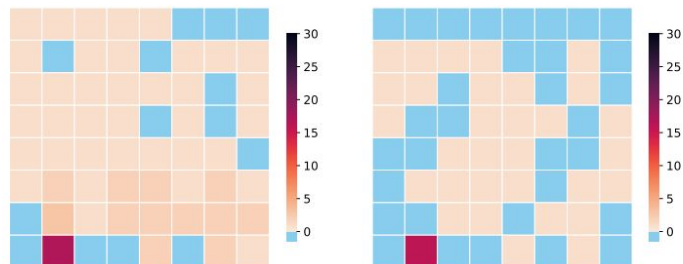
ZNS-tools: Zone Utilization



Same workload: YCSB-A

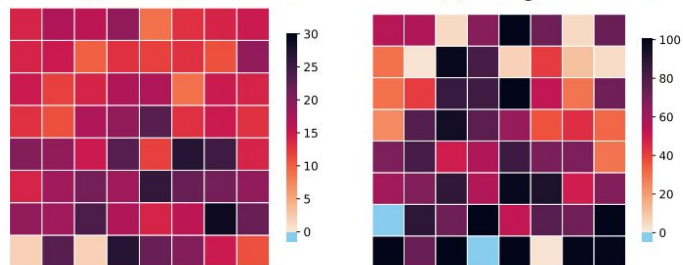
Very different ZNS utilization and placement

Data grouping interference!



(a) RocksDB + F2FS

(b) MongoDB + F2FS

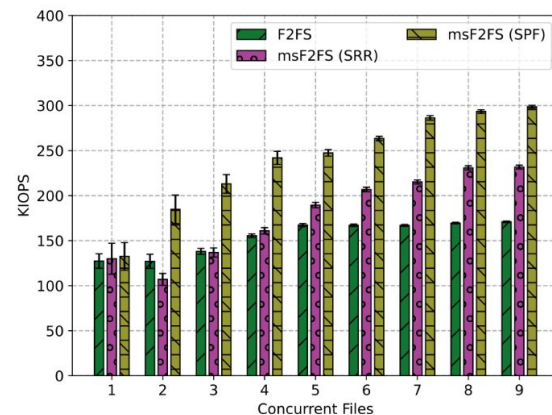
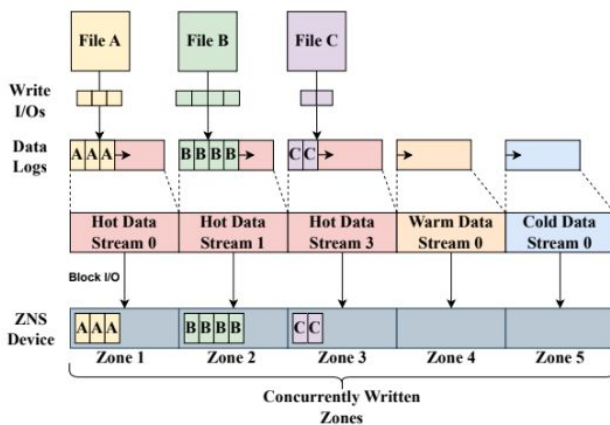
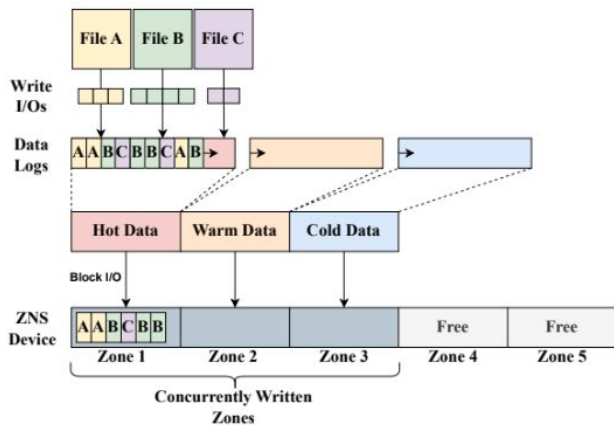


(c) PostgreSQL + F2FS

(d) RocksDB + (aged) F2FS

Result: Not all ZNS software stacks are equal, hence software specialization matters!

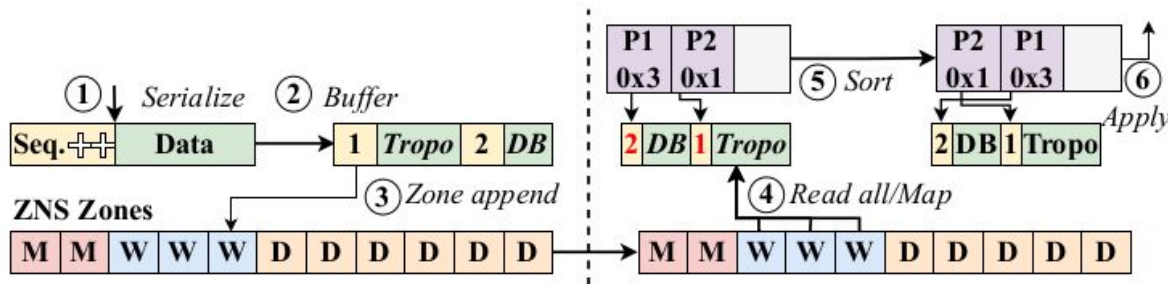
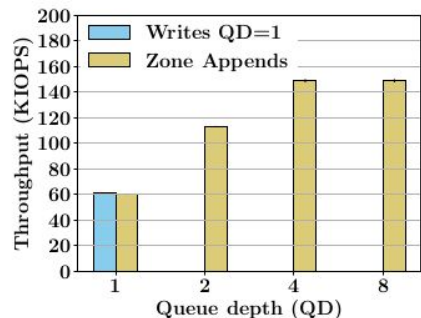
[1 / 2] Workload-Specialized Control - msF2FS



Multistream F2FS (msF2FS) optimized for NVMe ZNS devices

- Gives control over: file \Rightarrow F2FS zone \Rightarrow ZNS Zone sharing (exclusive, sharing)
- Physical separation in zones
- Performance scaling with inter-zone parallelism (F2FS does not support Appends)
- <https://github.com/stonet-research/msF2FS>

[2 / 2] Workload-Specialized Control - **zWAL**

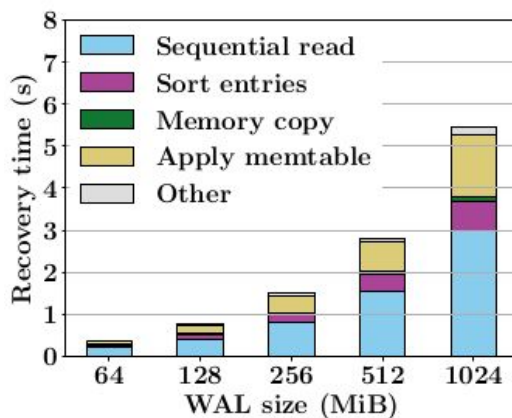
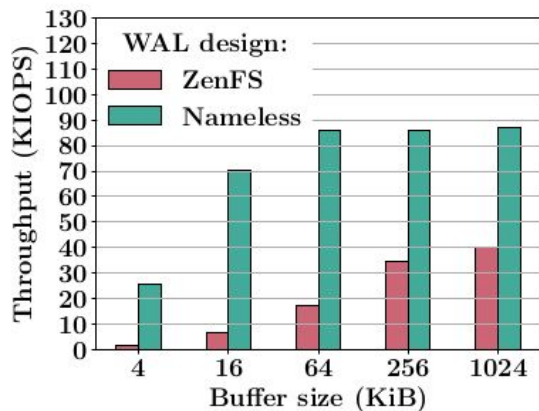
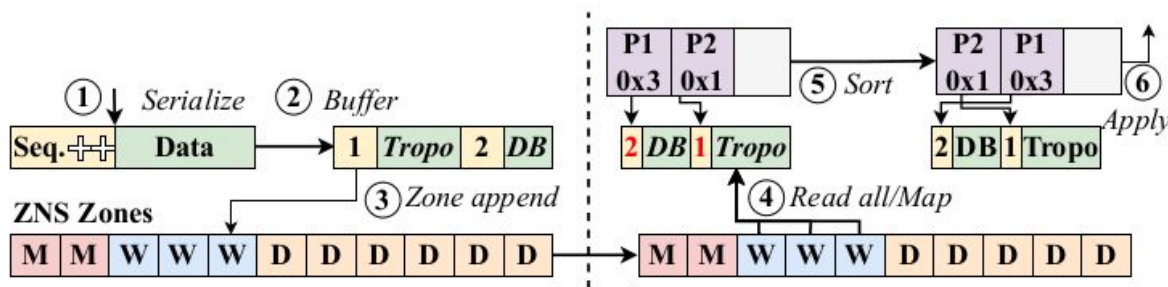
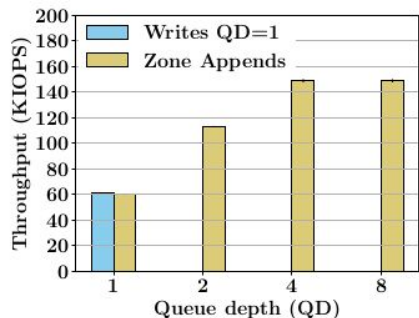


zWAL: A ZNS-native **Write-Ahead-Log (WAL)** design with Appends (parallel I/O)

Idea: Write in any order with ordering information with Append, then sort out later when reading

Open-sourced code: <https://github.com/Krien/ZenFS-append/tree/appends>

[2 / 2] Workload-Specialized Control - **zWAL**



(a) Micro-benchmarks

(b) Replay cost

(c) YCSB workload

~~[Part - 1/3] : Performance and Scheduling Overheads~~

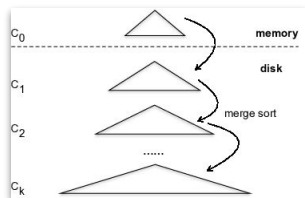
~~[Part - 2/3] : New Interfaces - Zone Namespace (ZNS) SSDs~~

~~[Part - 3/3] : (WiP) Building Workload-Optimized Storage Stacks~~

Workload-specialized QoS in an End-to-End Manner



RocksDB



File-A

File-B

File-C

File-D

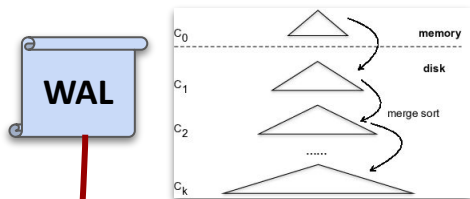
Disaggregated storage setup



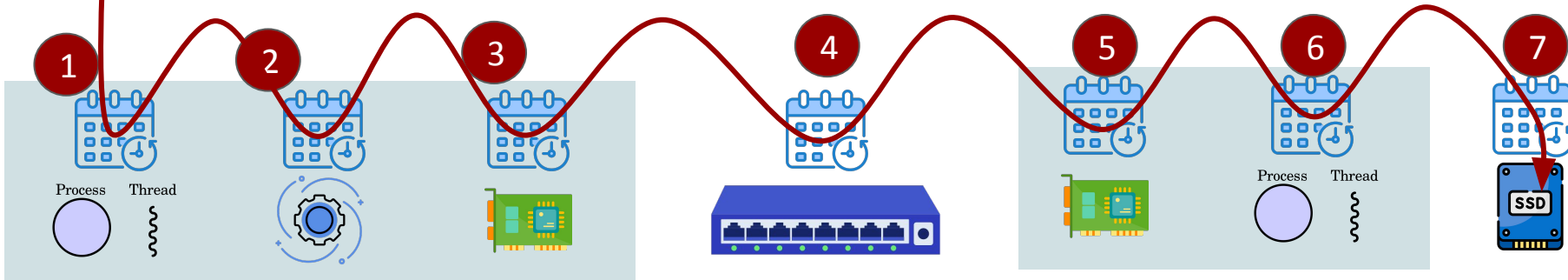
Workload-specialized QoS in an End-to-End Manner



RocksDB



File-A File-B File-C File-D



Workload-specialized QoS in an End-to-End Manner

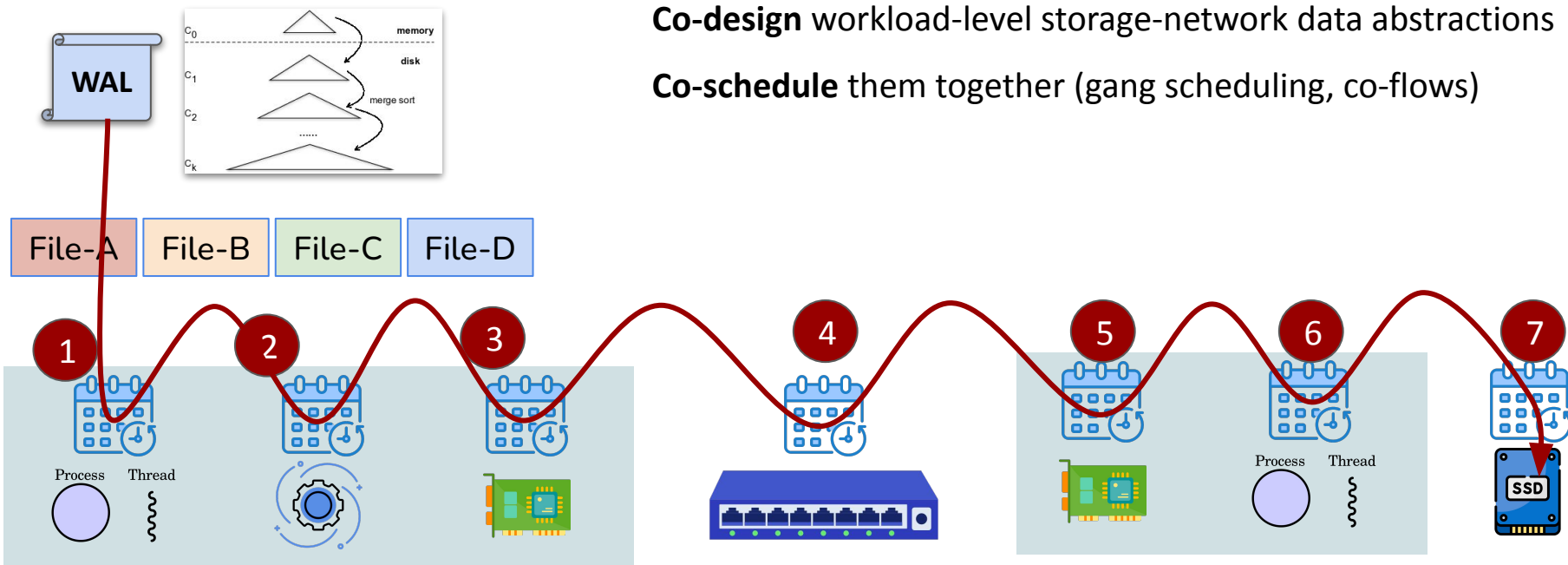


RocksDB

End-to-End abstraction for QoS:

Co-design workload-level storage-network data abstractions

Co-schedule them together (gang scheduling, co-flows)



Conclusion

Vision: use your favorite workload-specialized data structure I/O stack!

The era of workload-specialized storage stacks is here

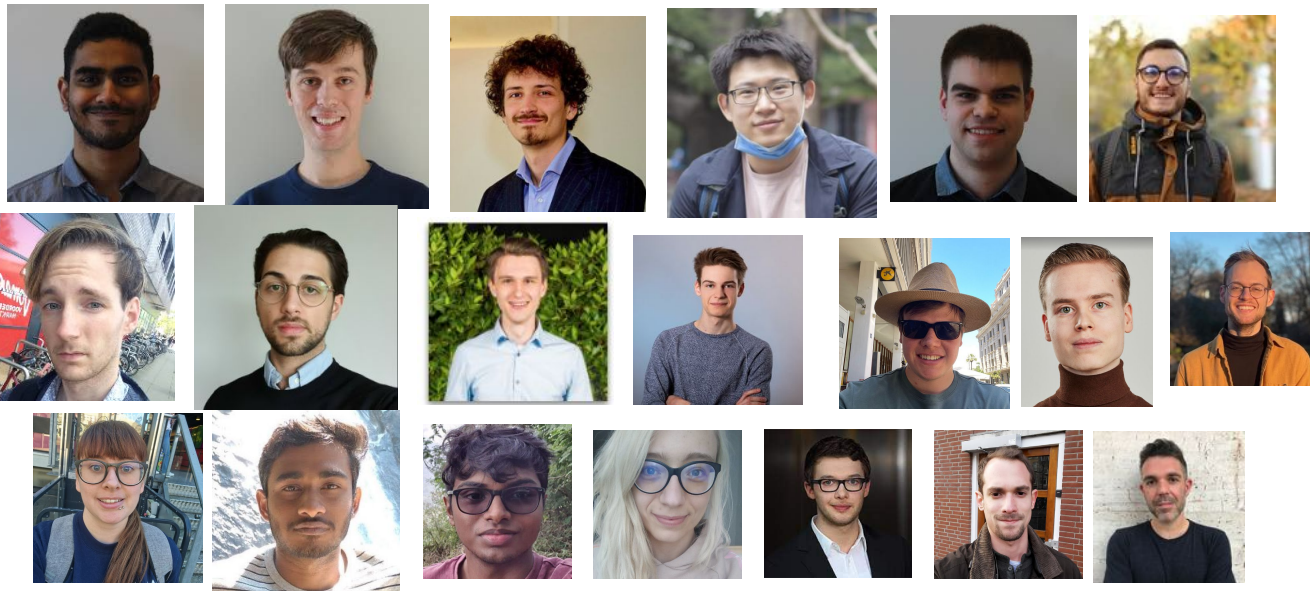
We are exploring:

- Workload-specialized storage software abstractions
- Mapping software interfaces to the available hardware interfaces
 - NVMe ZNS, KV-SSD, CXL (new)

WiP: [Network (CXL) + Storage = Disaggregation] File system, Key-value store, and ML workloads

Thank you!

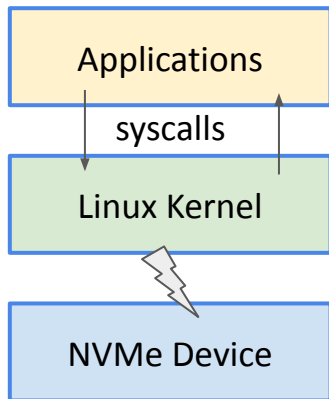
<https://stonet-research.github.io/>



Acknowledgments: Work generously funded by **the Dutch Research Council (NWO)** grants and donations from **Xilinx, Western Digital, Mellanox, AWS, and VU Amsterdam.**

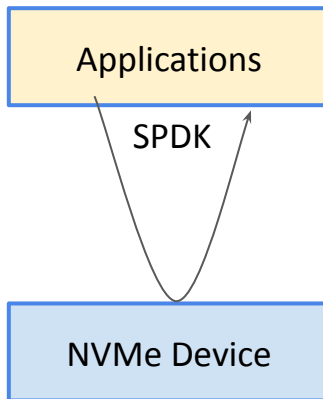
Backup

Revisiting Storage APIs: Rise of io_uring



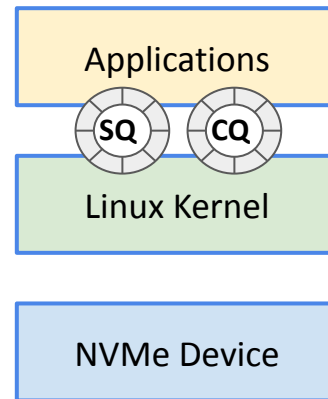
Libaio:

- + 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

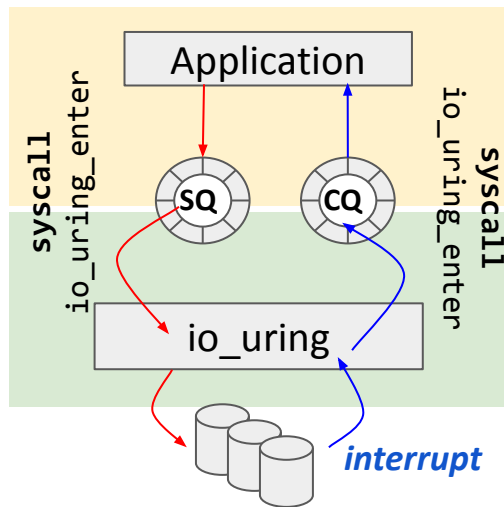


io_uring

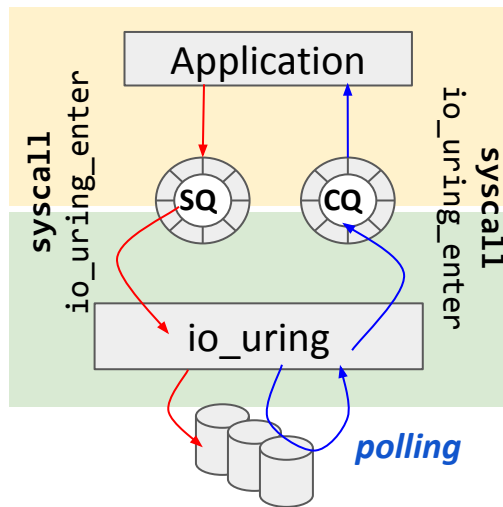
- + Command-based interface
- + Extensible

Best of both worlds?

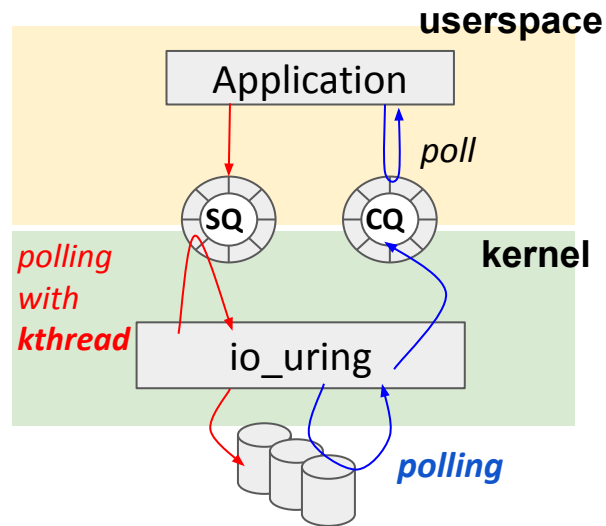
Three Modes of io_uring API



(a) default with syscalls



(b) [iou+p] with completion polling



(c) [iou+k] with submission polling

Benchmarking Setup

Setup 1 [Systor'22]:

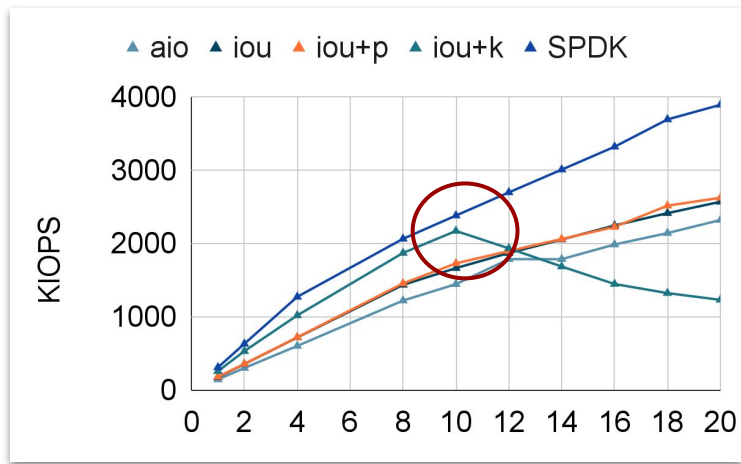
- 2x Intel® Xeon® E5-2630 (Sandy Bridge), 10 cores/socket ⇒ 20 CPU cores
- 20 Intel® DC P3600 400GB NVMe Flash SSDs ⇒ ~6 Million IOPS

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

Results: Scalability

Systor'22

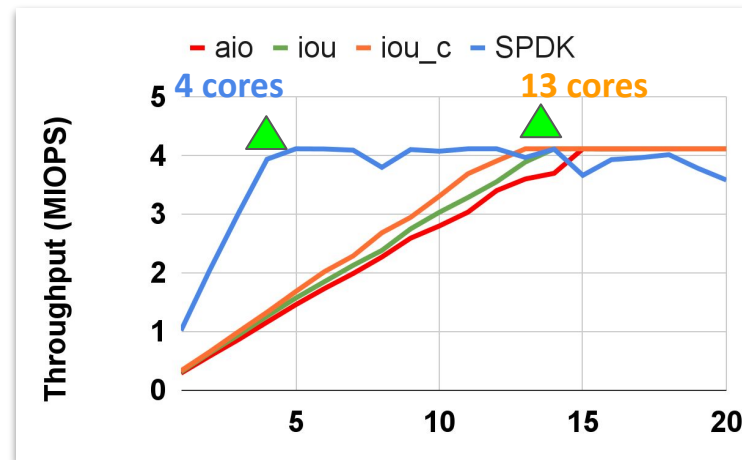
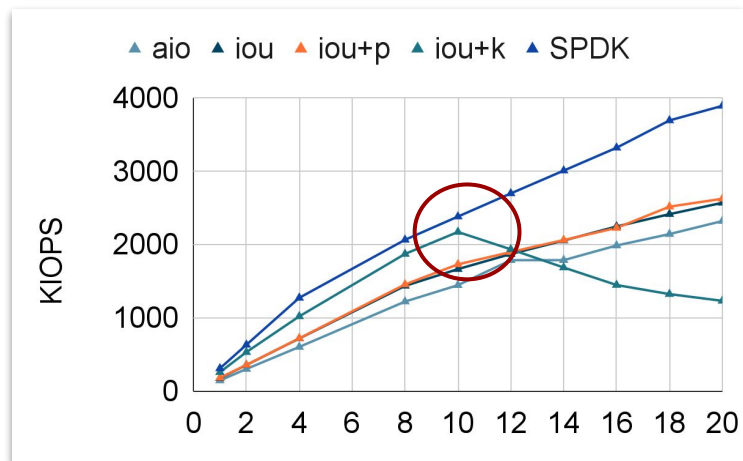


io_uring kernel polling: Performance collapses when not enough cores to poll

Results: Scalability

Systor'22

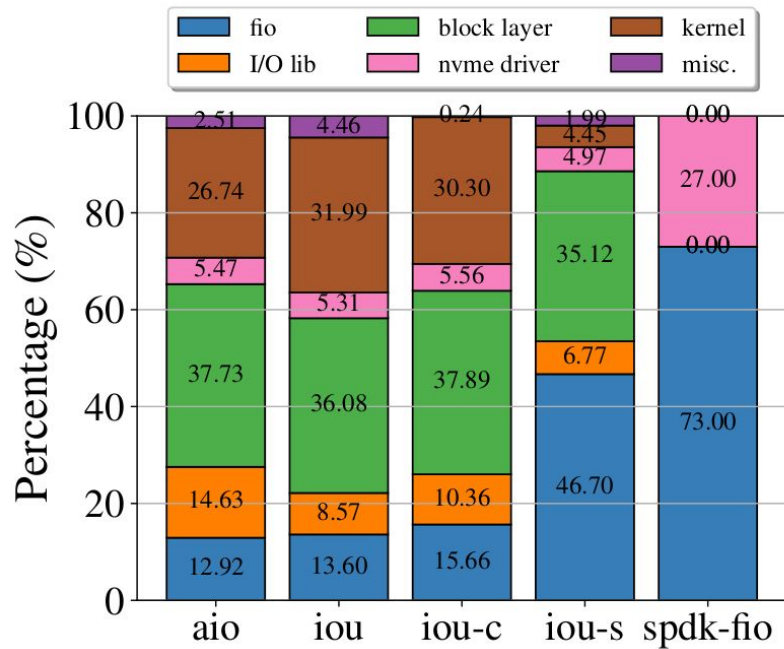
CHEOPS'23



io_uring kernel polling: Performance collapses when not enough cores to poll

CPU efficiency is still bad: 10x more CPU cores needed to match the SPDK performance

Results: CPU Profile



The Block layer takes a big chunk of the CPU cycles

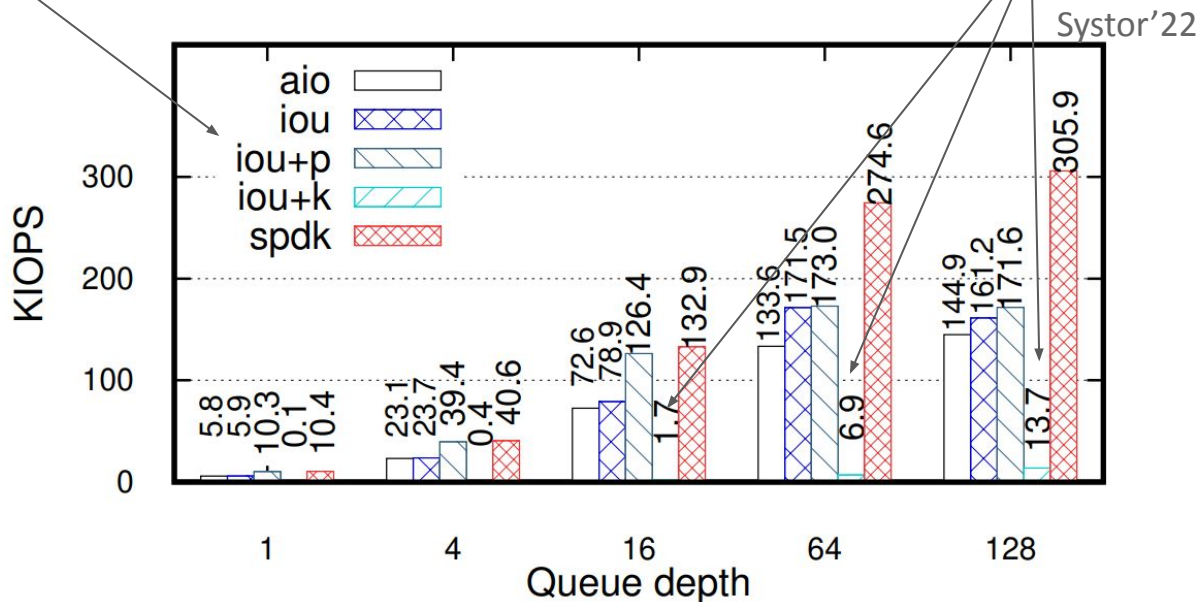
The kernel overheads with blocking interfaces

For SPDK, fio itself becomes the bottleneck

Results: Efficiency (single CPU core)

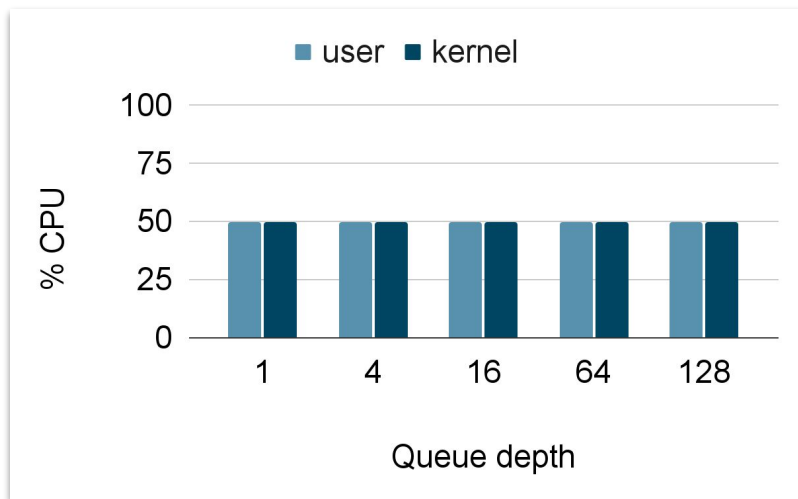
io_uring sits between libaio and SPDK

Performance collapses with the kernel polling



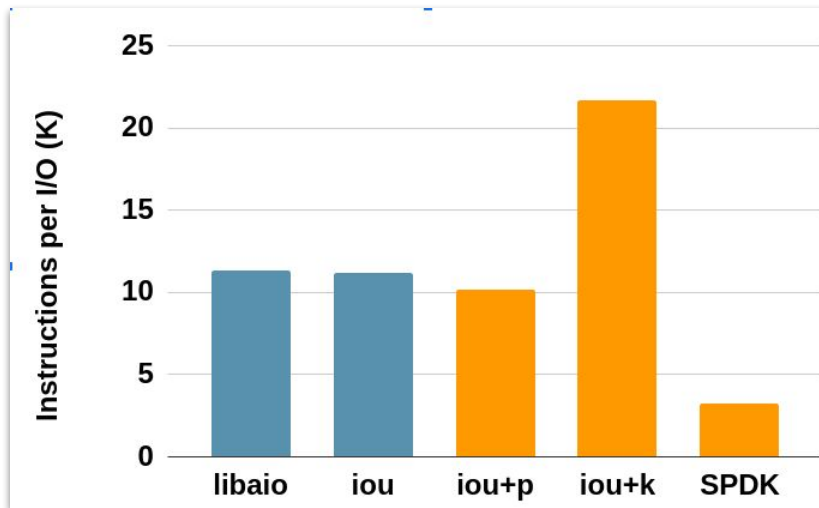
Analysis: CPU Profile

Systor'22



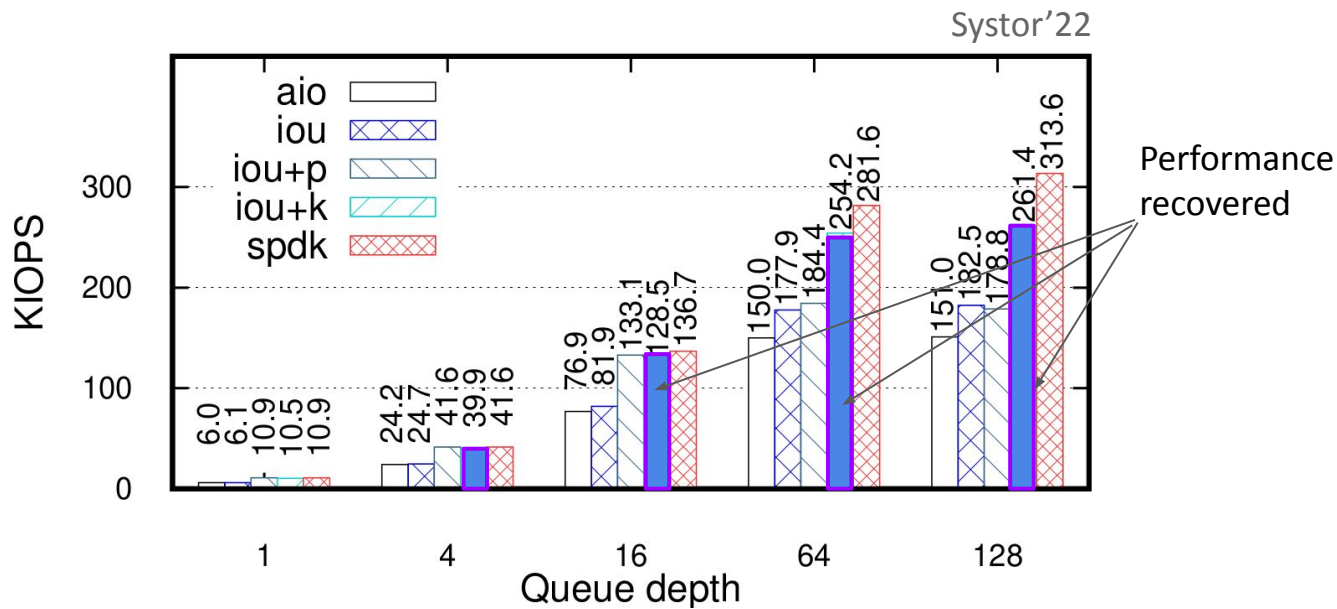
Poor scheduling, and CPU sharing - **Careful!**

CHEOPS'23



SPDK is still 5x more efficient

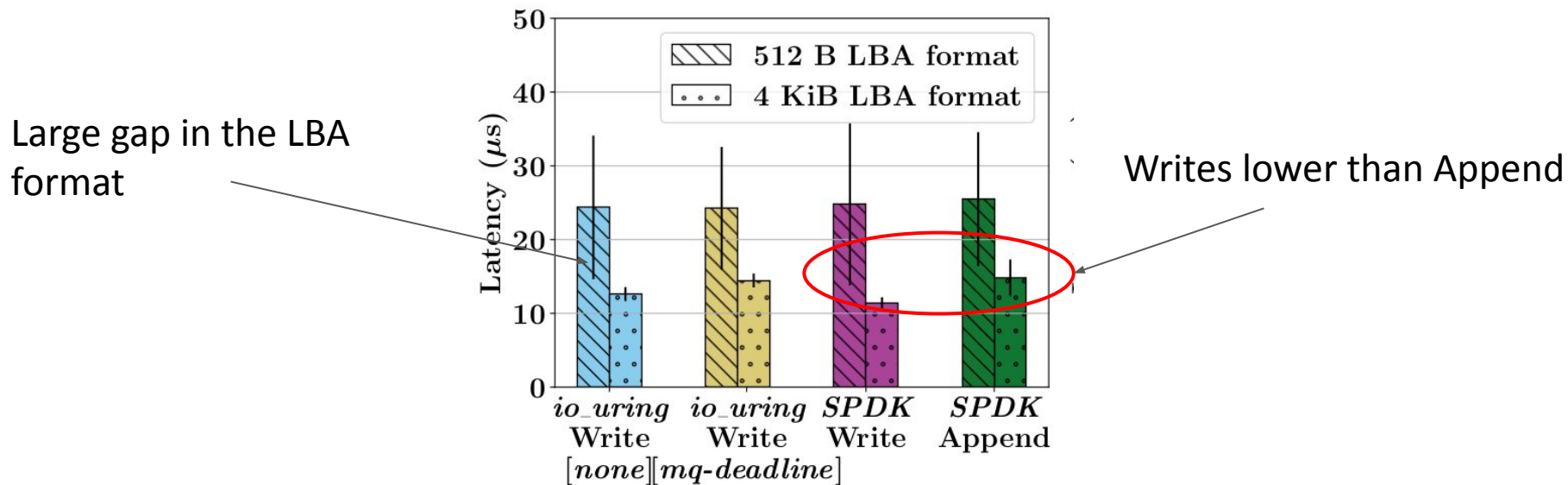
Results: Efficiency with TWO CPU cores



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

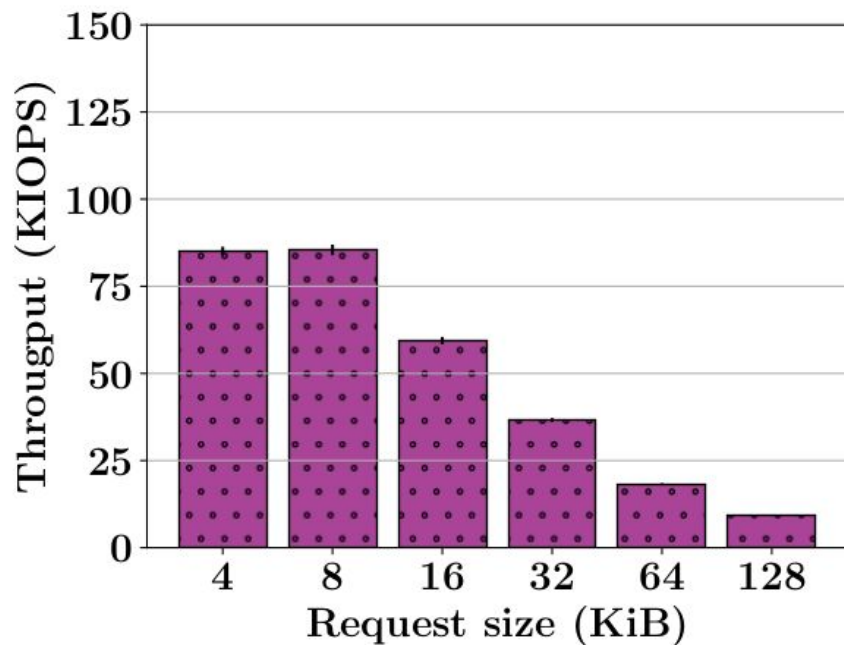
Normal service order can be resumed (**but** at the cost of 2x CPU cores)!

Result [1 / 4]: Write vs Append Latencies

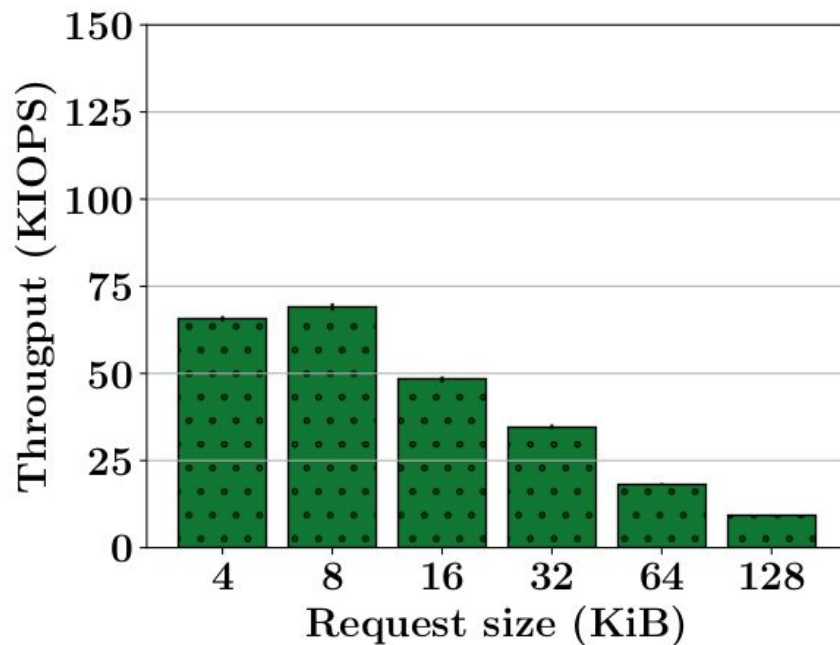


- 4KiB block size has lower latencies (up to 2x)
- Writes have lower latencies than Append operations in our experiments
- SPDK has lower latencies than the Linux I/O stack (none, mq-deadline)

Write and Append: Bandwidth

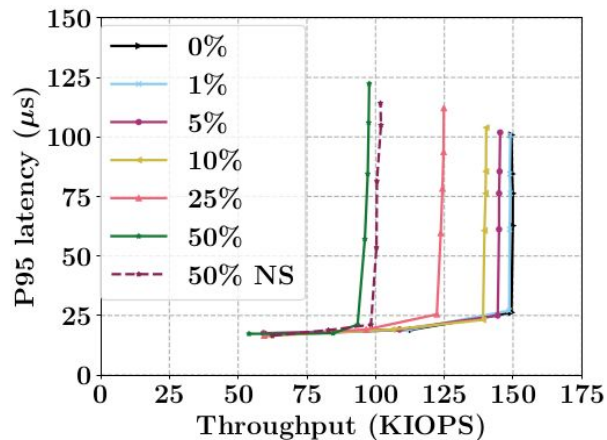


(a) *write*

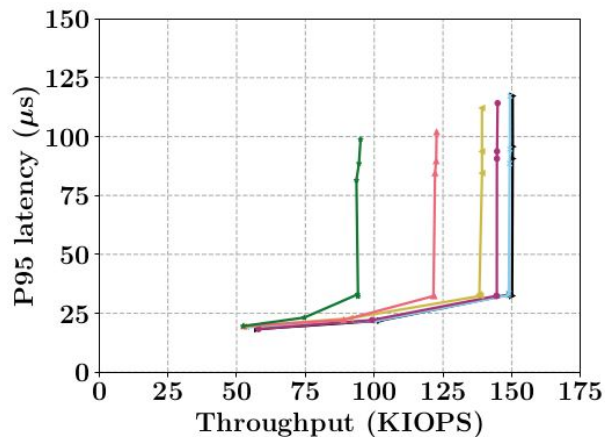


(b) *append*

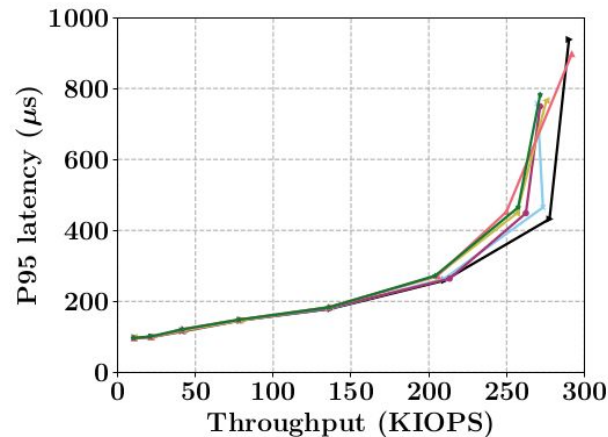
New Interference: Reset on I/O Operations



(a) on *write* with inter-zone concurrency.



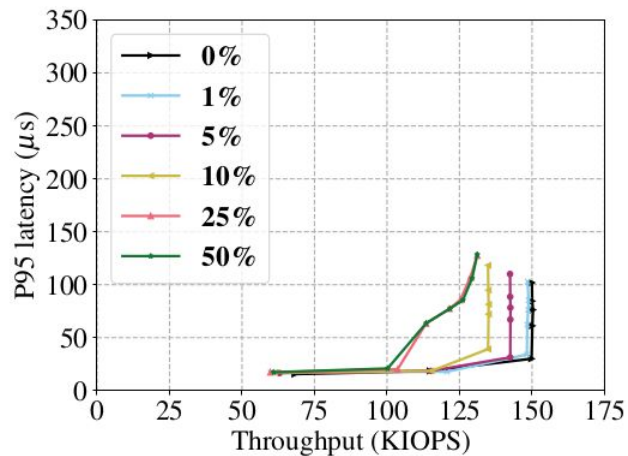
(b) on *append* with intra-zone concurrency.



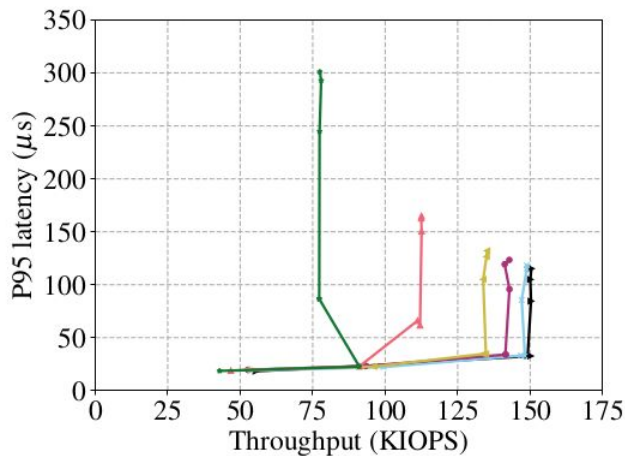
(c) on *read* with intra-zone concurrency.

- Concurrent Reset commands slow down I/O (write, append, reads)
- Namespace based isolation does not help

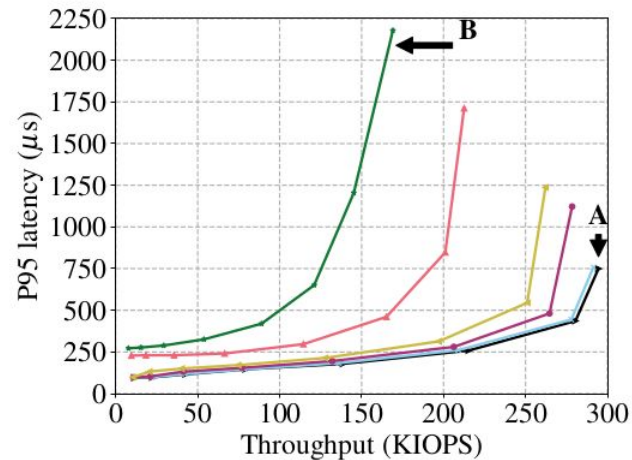
New Interference: Finish on I/O Operations



(a) on write with inter-zone concurrency.

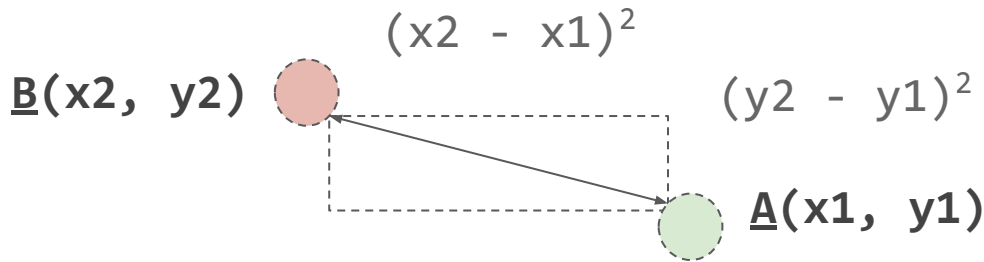


(b) on append with intra-zone concurrency.



(c) on read with intra-zone concurrency.

Make a first-order linear model using the EMD distance:



ZINC Interference Model

$$Z^{Inter} = \frac{1}{n} \sum_{i=1}^n \sqrt{\alpha \times (\Delta T_i)^2 + \beta \times (\Delta L_i)^2} \quad (1)$$

With:

$$[\alpha + \beta = 1, \quad 0 \leq \alpha \leq 1, \quad 0 \leq \beta \leq 1]$$

$$\Delta T_i = \left(\frac{T_i^{int} - T_i^{iso}}{T_i^{iso}} \right) \quad (2)$$

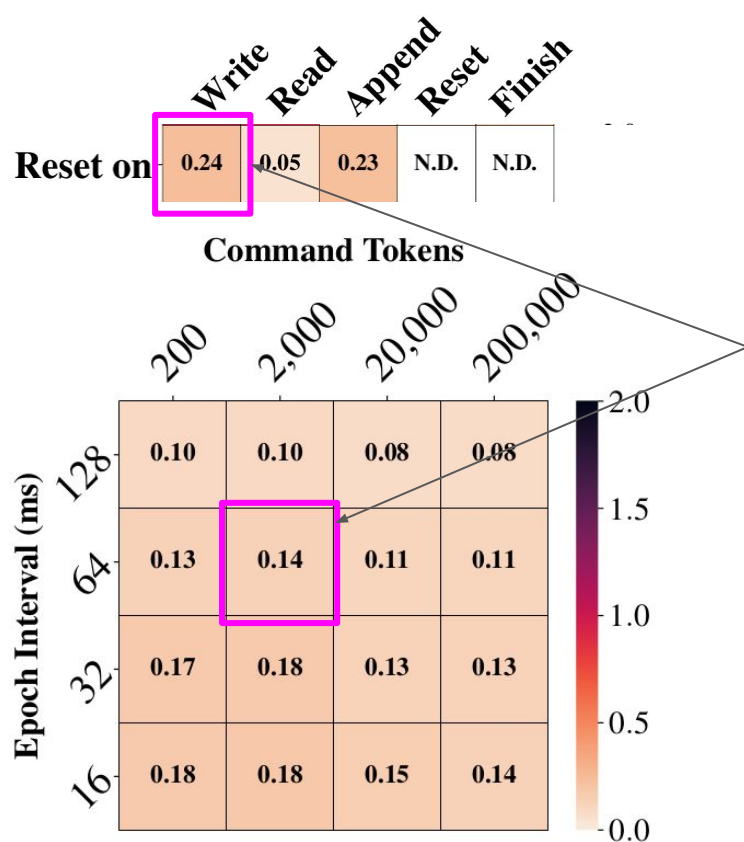
$$\Delta L_i = \left(\frac{L_i^{int} - L_i^{iso}}{L_i^{iso}} \right) \quad (3)$$

Impact of using ZINC



RocksDB

0% Reset	50% Reset	
78.9 KIOPS	72.1 KIOPS	-8.7% drop
78.2 KIOPS	80.0 KIOPS	+2.3% gain



- ZINC helps to control the interference between I/O & zone-management commands
- ZINC helps to deliver workload-level performance gains

ZINC: Reset Profile

