Empowering Storage Systems Research with NVMeVirt

Jaehoon Shim

(*mattjs@snu.ac.kr*)

Systems Software & Architecture Lab.

Seoul National University

Spring 2024

NVMeVirt: A Versatile Software-defined Virtual NVMe Device

(Sang-Hoon Kim et al., USENIX FAST, 2023)

Why Do We Need NVMeVirt?

- Emulators can facilitate advanced storge research by ***actualizing*** novel device concepts
  - Open-Channel SSD, NVM SSD, KVSSD, Zoned Namespace (ZNS) SSD, computational storage, …
  - Can implement the concepts in software
    - No need to wait until they become available at retailor shops
    - $$$

- Cannot support some I/O models and storage configurations that are frequently used for building modern storage systems

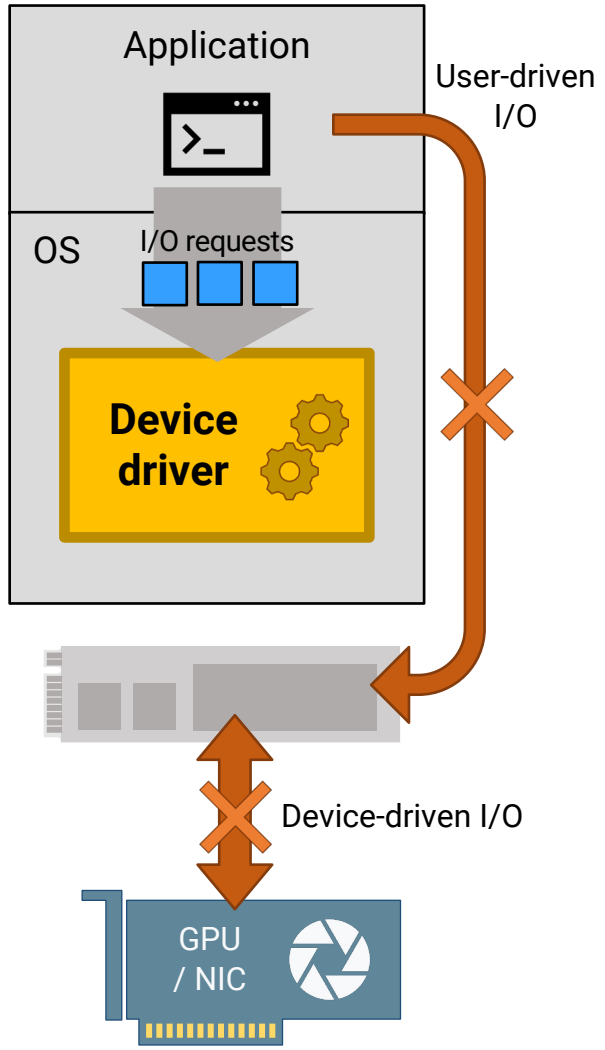
	Simulators		Emulators				
	Trace-driven [30, 36]	**Full-system [10, 22, 49]**	**VM-based [12, 32, 55]**	**Block-driver level [56]**	**NVMe-driver level [35]**	**HW platforms [21, 28]**	**NVMeVirt**
Deployable in real environments	No	Yes	Yes	Yes	Yes	Yes	**Yes**
Execution speed	Fast	Very slow	Slow	Fast	Fast	Real-time	**Real-time**
NVMe Multi-queue support	No	Yes	Yes	No	Yes	Yes	**Yes**
NVMe interface modification	Impossible	Easy	Easy	Impossible	Easy	Difficult	**Easy**
Low-latency device support	Possible	Possible	Difficult	Possible	Possible	Difficult	**Possible**
Kernel bypassing with SPDK	No	No	Yes	No	No	Yes	**Yes**
PCI peer-to-peer DMA support	No	No	No	No	No	Yes	**Yes**
NVMe-oF target offloading	No	No	No	No	No	Yes	**Yes**

Rely on collected traces / extremely slow

Expensive / hard to modify

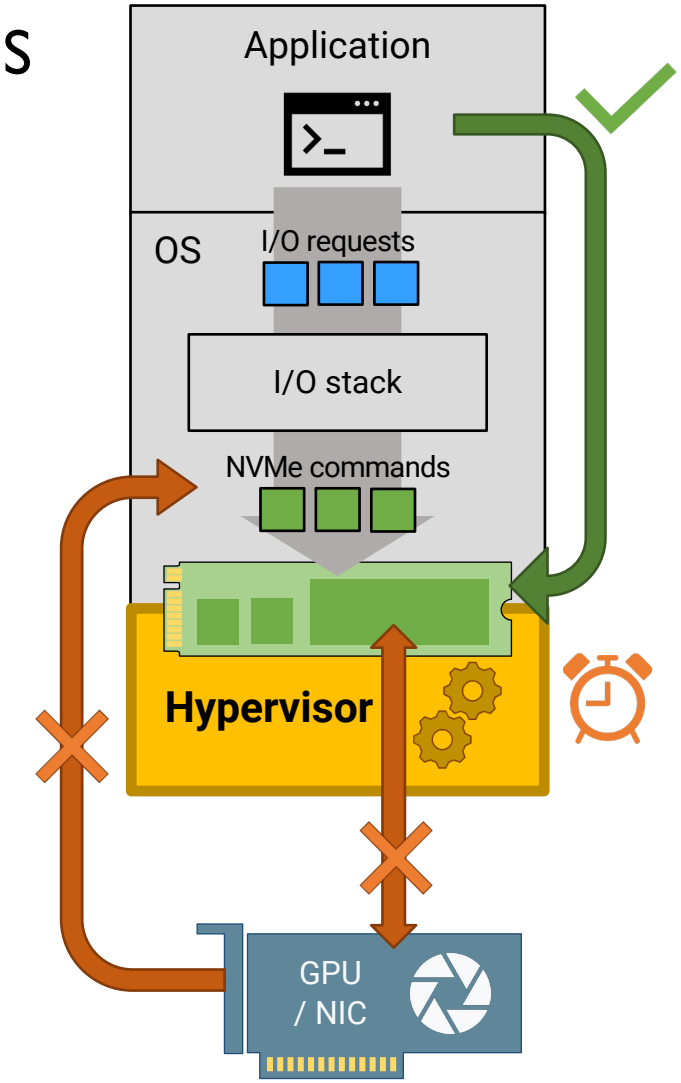
Previous: Device Driver-level Approaches

- Catch I/O requests at the block/NVMe device driver and emulate the requests
  - David[FAST11], FlexDrive[HPCC16], …
- Can only process '*regular*' I/O requests
- Unable to support user-driven I/O: Kernel bypassing with SPDK
- Neither for device-driven I/O
  - RDMA target for NVMe-oF, PCI peer-to-peer DMA



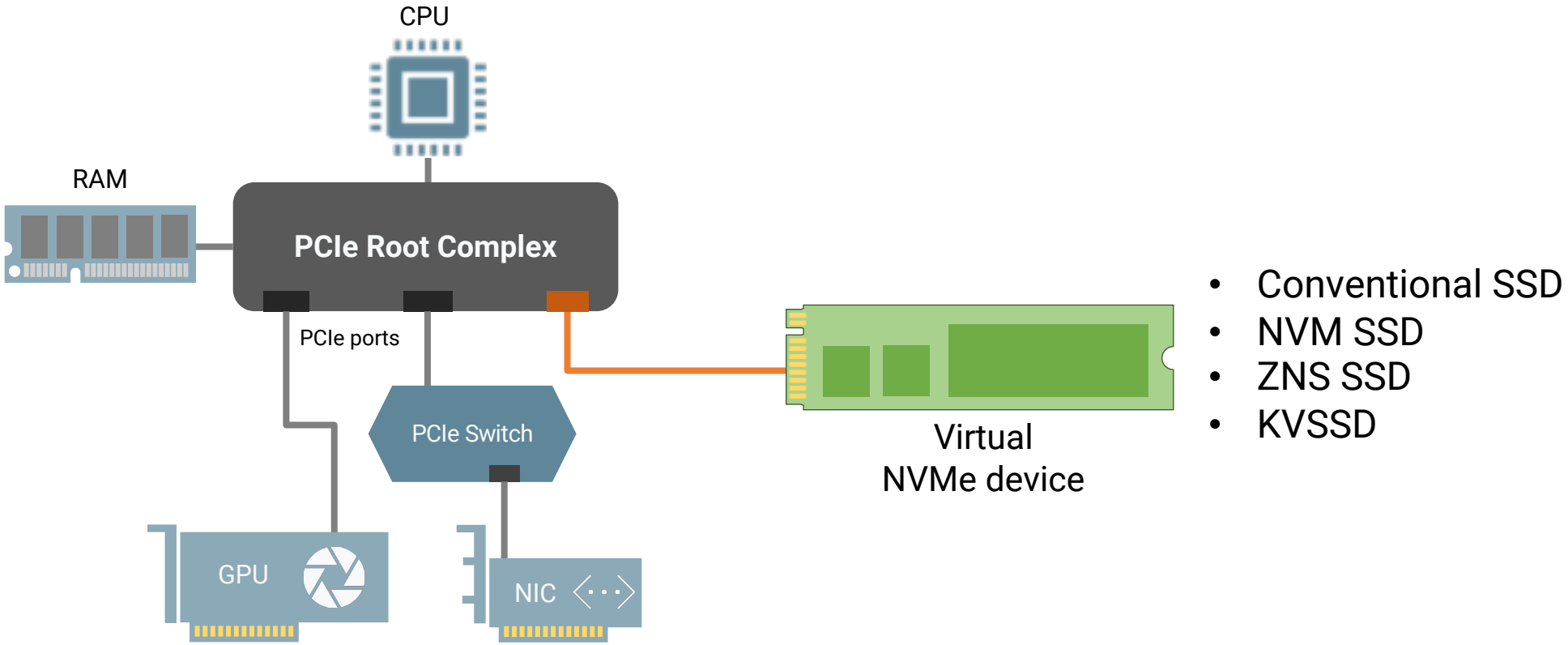
Previous: Virtualization-based Approaches

- Hypervisor emulates a virtual device exposed to the guest OS
  - VSSIM[MSST13], FEMU[FAST18], ZNS+[OSDI21], …
- Can support the user-driven I/O
- Cannot support device-driven I/O configurations
  - ~~No way to contact the virtual device from real devices on the host~~
  - Complicated memory layout in VM environments makes RDMA infeasible
- Virtualization overhead limits and/or impacts on the performance characteristics of target devices



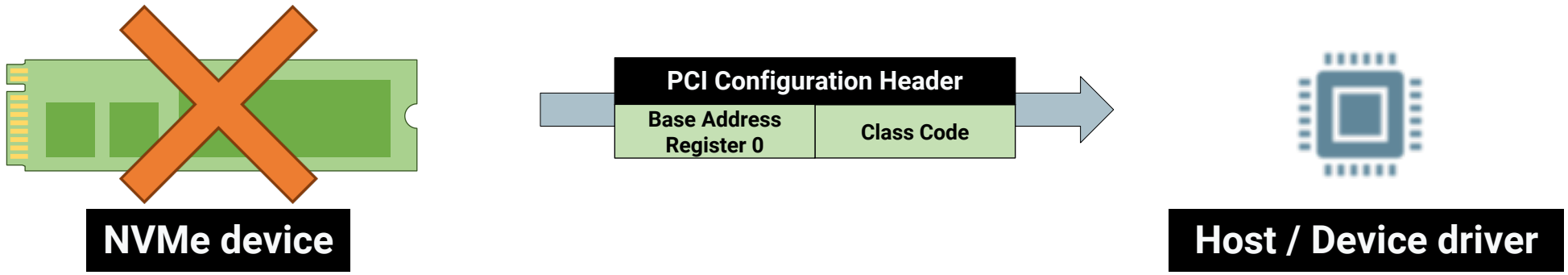
NVMeVirt: Virtual NVMe Device in Software

- A light-weight kernel module that presents a native NVMe device to the entire system
  - Support any storage configurations!



Challenges for Virtual PCI/NVMe Devices

- Challenge 1: How to create a virtual PCI device instance in software?
  - The real device initiates the initialization
  - We don't have the physical device that can initiate the initialization
  - We don't want to mess up with the existing PCI subsystem implementation



Challenges for Virtual PCI/NVMe Devices

- Challenge 1: How to create a virtual PCI device instance in software?
  - The real device initiates the initialization
  - We don't have the physical device that can initiate the initialization
  - We don't want to mess up with the existing PCI subsystem implementation

- Solution: Make a PCI device instance indirectly through PCI bus
  - Create a virtual PCI bus that presents the PCI configuration header of virtual device to the PCI subsystem
  - No modification is needed in the Linux kernel

Challenges for Virtual PCI/NVMe Devices

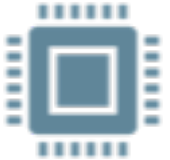
- Challenge 2: Cannot rely on the PCI mechanism to detect the requests from the host-side
  - Updates to the control block and doorbells are notified to the device as PCI transactions

NVMe device

	NVMe Control Block	
00h	Controller Capabilities	
08h	Version	Interrupt Mask Set
10h	Interrupt Mask Clear	Controller Configuration
18h	Reserved	Controller Status
20h	Subsystem Reset	Admin Q Attributes
28h	Admin SQ Base Address	
30h	Admin CQ Base Address	
38h	CMB Location	CMB Size

	Doorbells			
1000h				
1010h				
1020h				

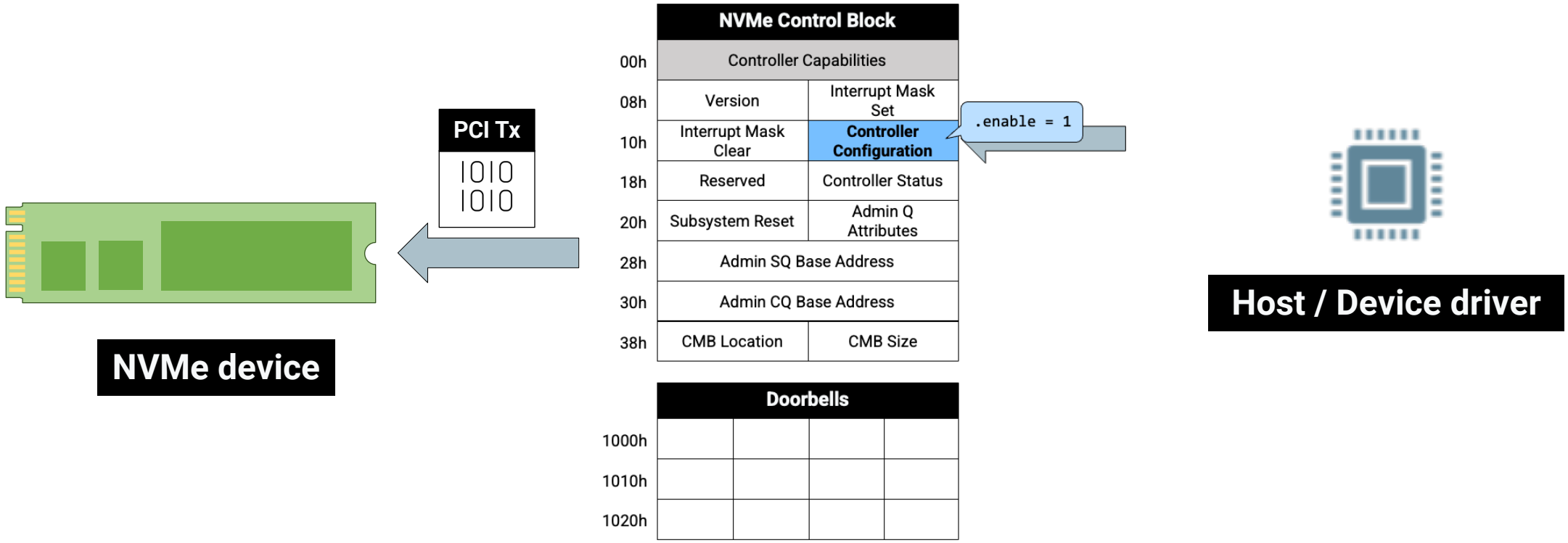
Device memory mapped to the host's address space



Host / Device driver

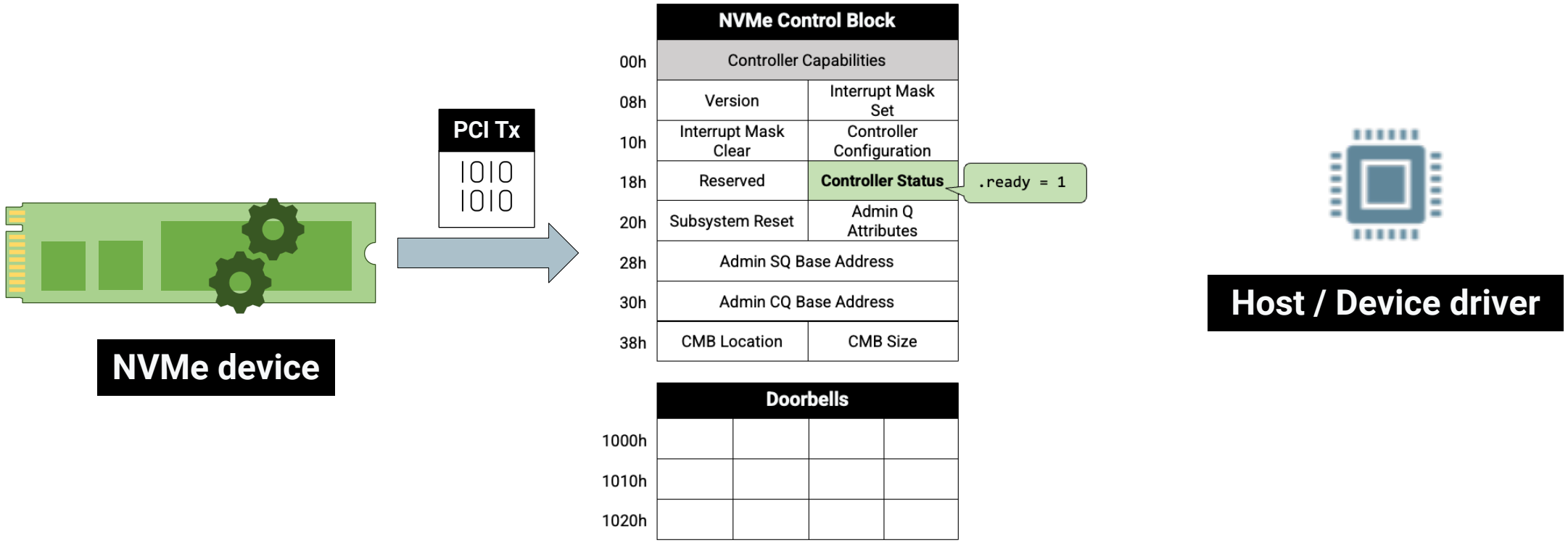
Challenges for Virtual PCI/NVMe Devices

- Challenge 2: Cannot rely on the PCI mechanism to detect the requests from the host-side
  - Updates to the control block and doorbells are notified to the device as PCI transactions



Challenges for Virtual PCI/NVMe Devices

- Challenge 2: Cannot rely on the PCI mechanism to detect the requests from the host-side
  - Updates to the control block and doorbells are notified to the device as PCI transactions



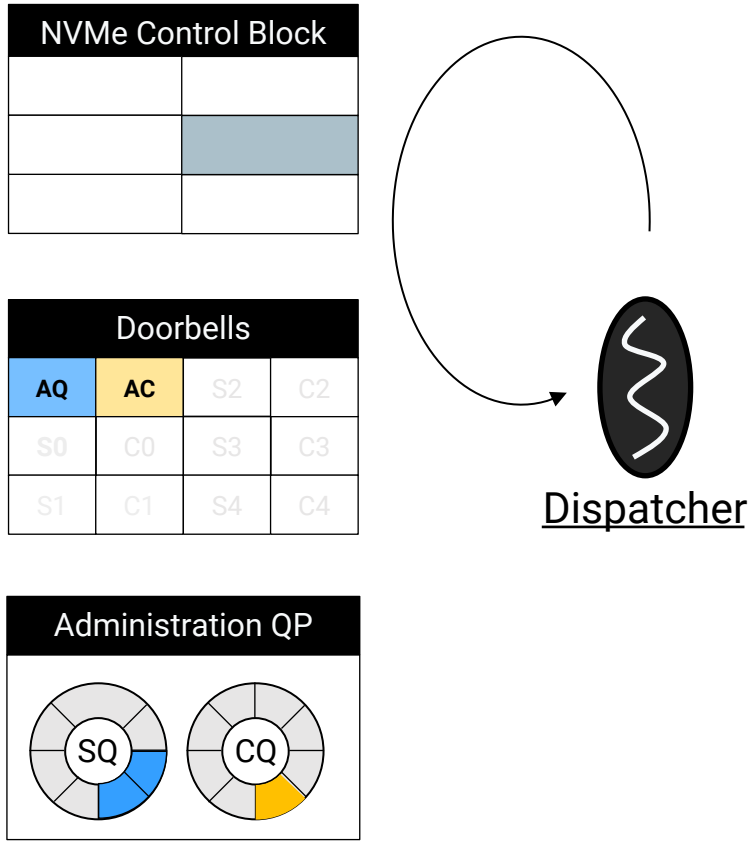
Challenges for Virtual PCI/NVMe Devices

- Challenge 2: Cannot rely on the PCI mechanism to detect the requests from the host-side
  - Updates to the control block and doorbells are notified to the device as PCI transactions

  → Changes are applied silently as normal memory writes

- Solution: Dedicate a thread that scans the control block and doorbells to find any updates

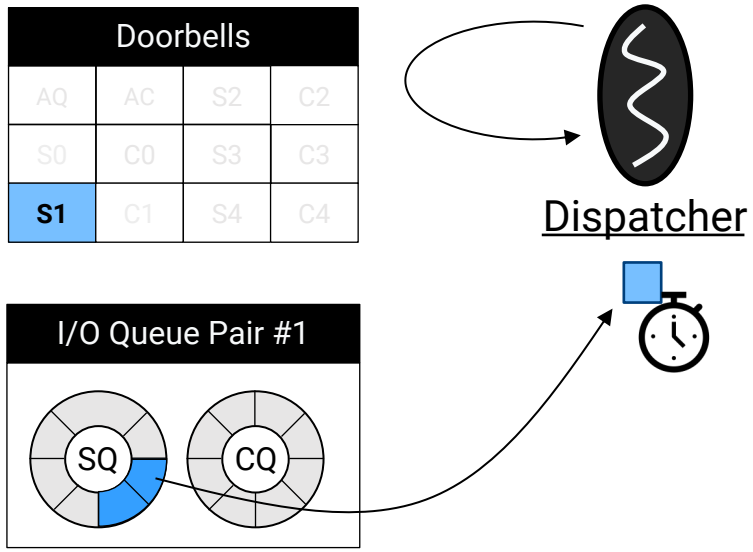
Emulating NVMe Device: Configuration Requests



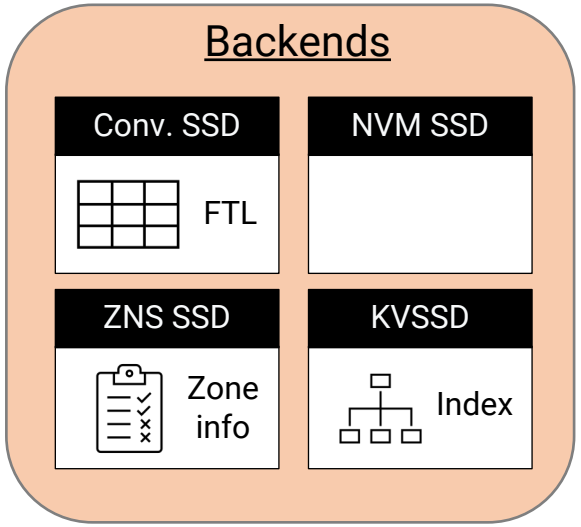
- Dispatcher directly processes configuration requests from admin Q
  - Enable/shutdown device
  - Identify device and namespaces
  - Setup administration queue pair
  - Set/get features (e.g., # of queues)
  - Allocate/deallocate I/O queues
- Handle completion doorbells
  - Perform housekeeping

Emulating NVMe Device: I/O Requests

- I/O requests are divided into backend operations
  - According to the configured backend type

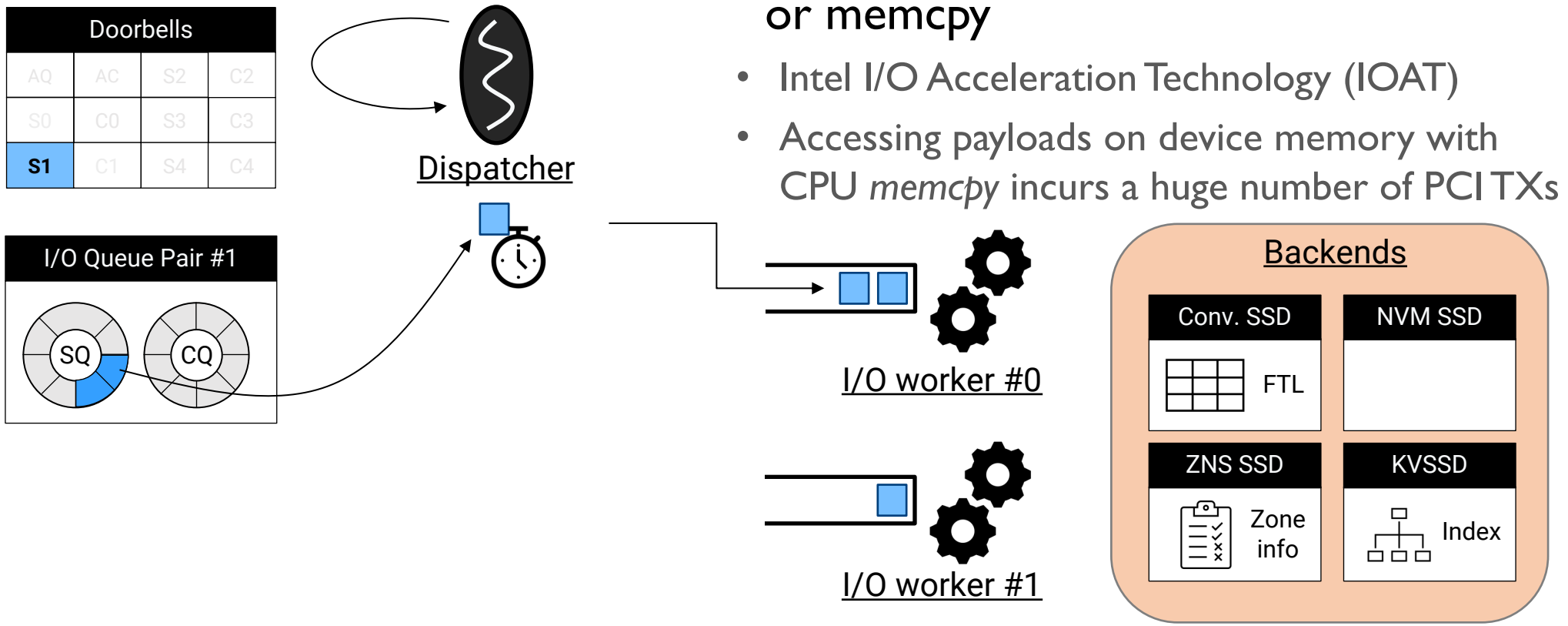


- Attach timestamps on the backend operations
  - Requested time, expected completion time



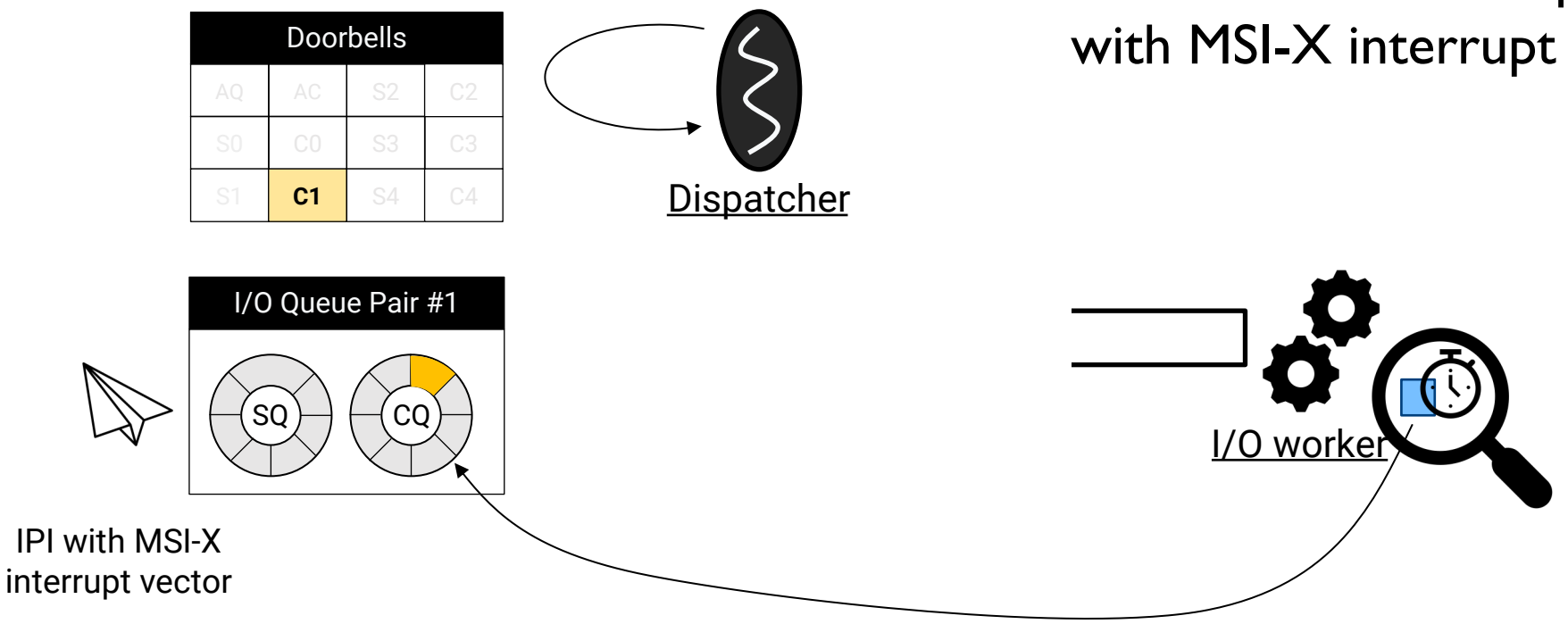
Emulating NVMe Device: I/O Requests

- Backend operations are dispatched to I/O workers
- I/O worker moves data using DMA engine or memcpy
  - Intel I/O Acceleration Technology (IOAT)
  - Accessing payloads on device memory with CPU *memcpy* incurs a huge number of PCI TXs



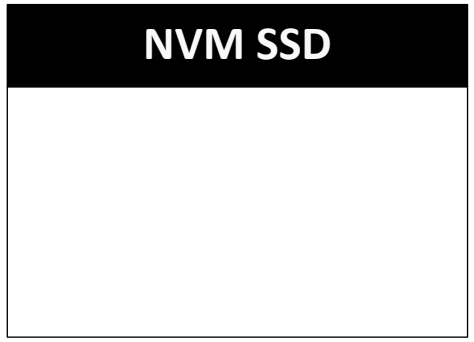
Emulating NVMe Device: I/O Requests

- I/O worker compares the current and expected completion time
- Notifies the I/O completion through IPI with MSI-X interrupt vector

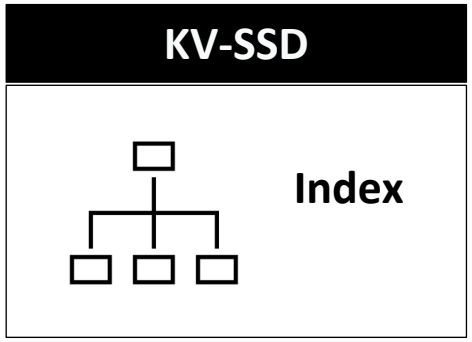


Performance Models

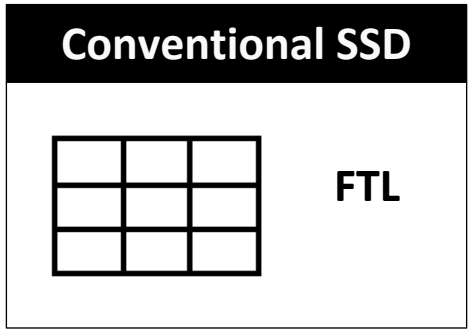
- Expected completion time is calculated by the performance model
  - Needs to handle various sizes and operations



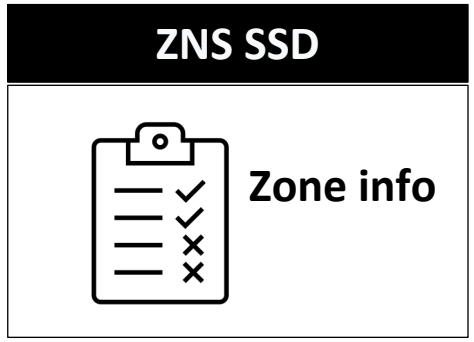
- In-place update, No GC
- OptaneDC-like SSDs



- SNIA KV APIs: store, retrieve, exist, delete, iterate
- OpenMPDK-compliant
- Hash-based index



- Page-mapping FTL
- GC
- Write buffer
- Multiple FTL instances



- Zone management
- Write buffer
- Multiple FTL instances

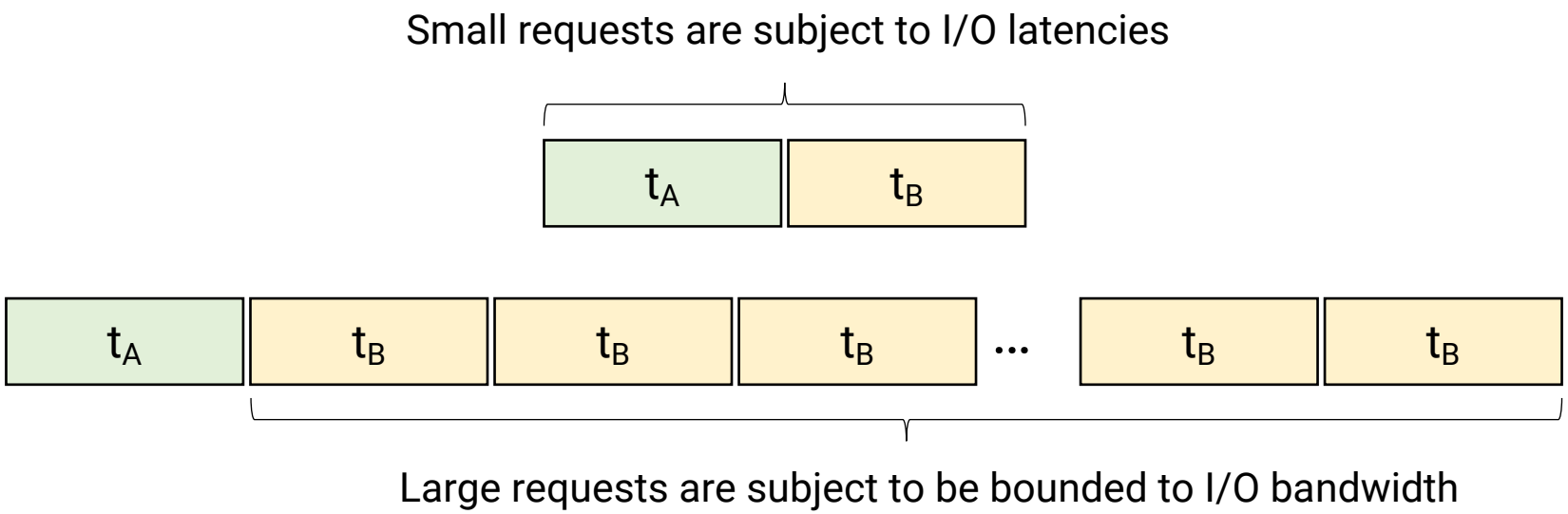
Simple model

Parallel model

Simple Model

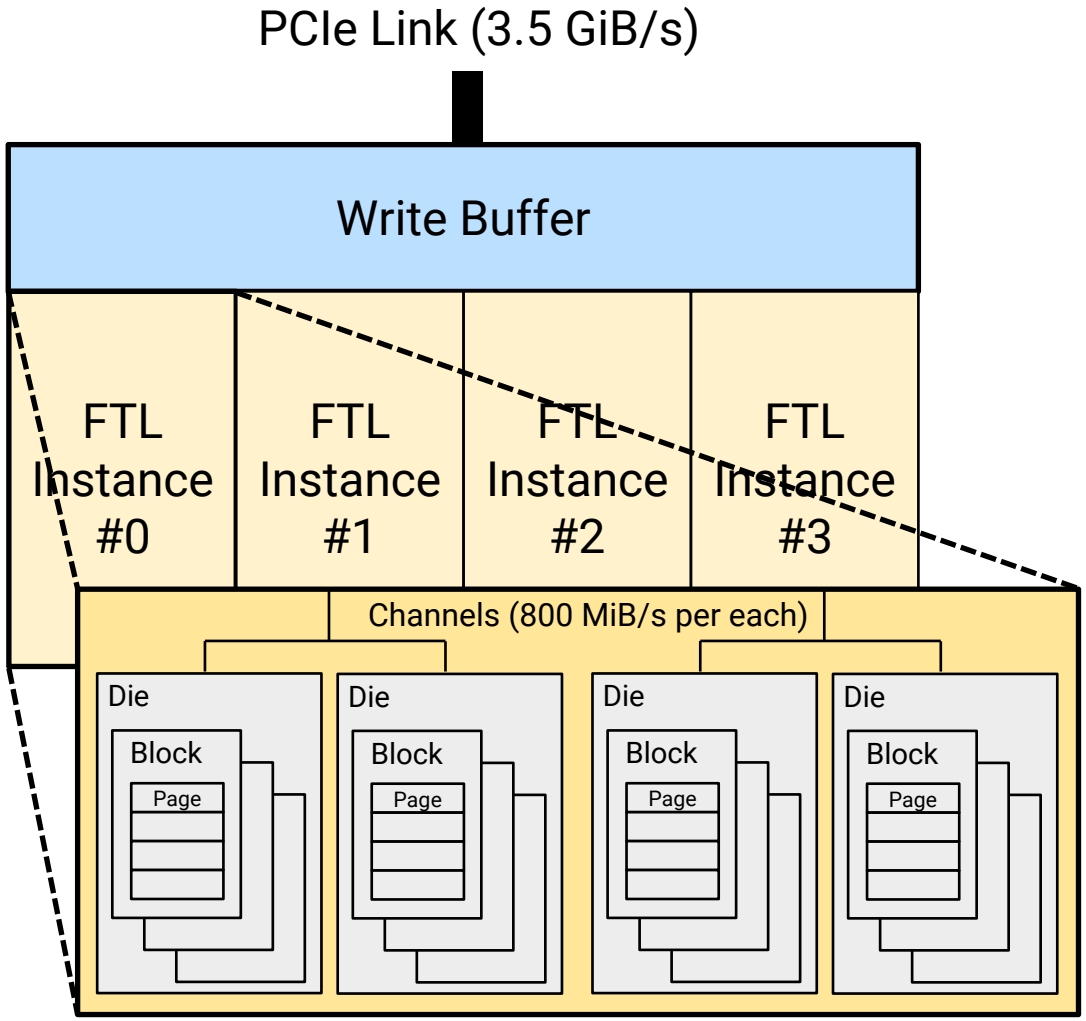
- Models a set of parallel I/O units where each I/O unit handles a sequence of I/O operations
  - Timing parameters are computed from ***target_latency*** and ***target_bandwidth***
  - Target latency and bandwidth can be independently specified
  - e.g. OptaneDC SSD: read 12µs @ 2.4GiB, write 14µs @ 2.0GiB/s

- Used for OptaneDC-like NVM SSDs and KV-SSDs



Parallel Model

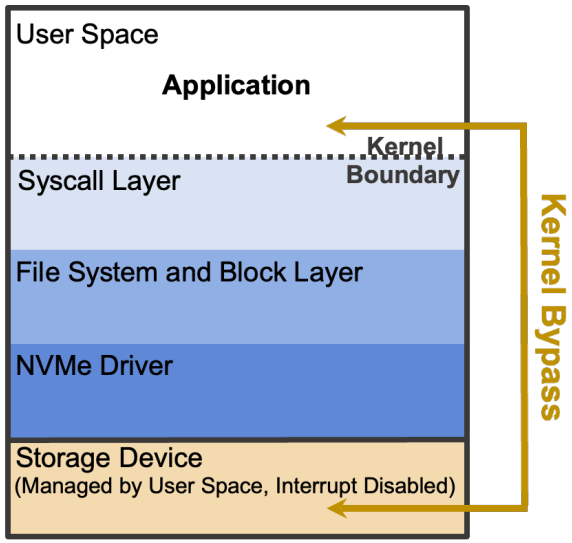
- Models complicated SSD internals
  - A full-scale page-mapped FTL with GC
  - Model the on-device write buffer
  - Model the parallel architectures in modern SSDs
    - Multiple FTL instances
    - Multiple dies and channels that operate independently
    - Use superblock as the operation unit
    - PCIe link and channels with limited aggregate bandwidth
  - Reserved N% of space as OP area
  - Support small-/large-zone ZNS SSDs
  - Based on token-based scheduling



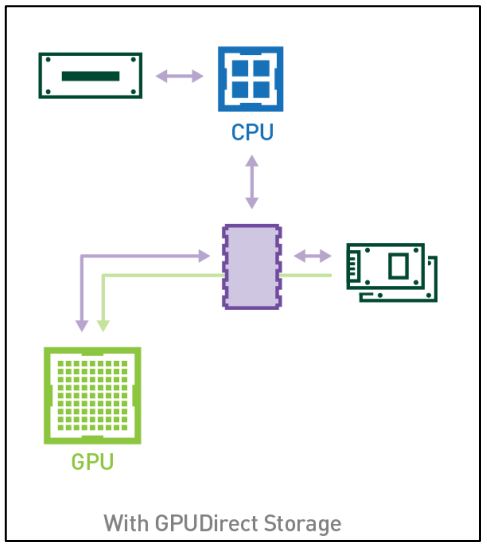
Supporting Modern Storage Environments

- Kernel bypassing using SPDK

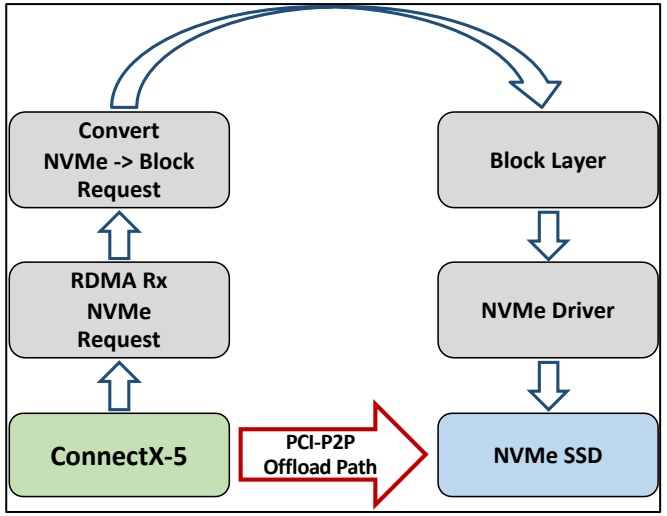
- Compatible with other PCIe devices like GPUs, NICs, and FPGAs
  - PCI P2P communication
  - NVMe-oF target offloading



SPDK

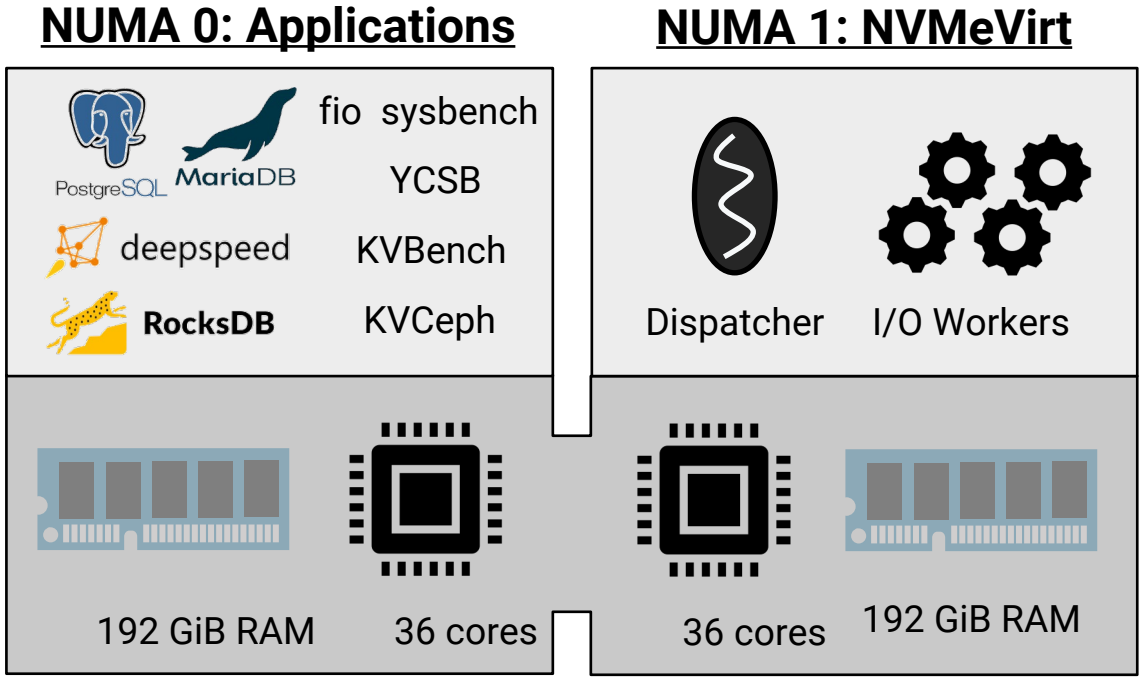


GPUDirect Storage



NVMe-oF Target Offload

Evaluation



- Implemented in the Linux kernel 5.15 (~9,000 LoC)
- Intel Xeon Gold 6240 x2
- 394 GiB RAM
- Debian Bullseye 11.5
- MariaDB 10.5
- PostgreSQL 13

Samsung 970 PRO

- Conventional SSD
- 512 GB

Intel P4800X

- OptaneDC NVM SSD
- 350 GB

Samsung KVSSD

- 3.84 TB

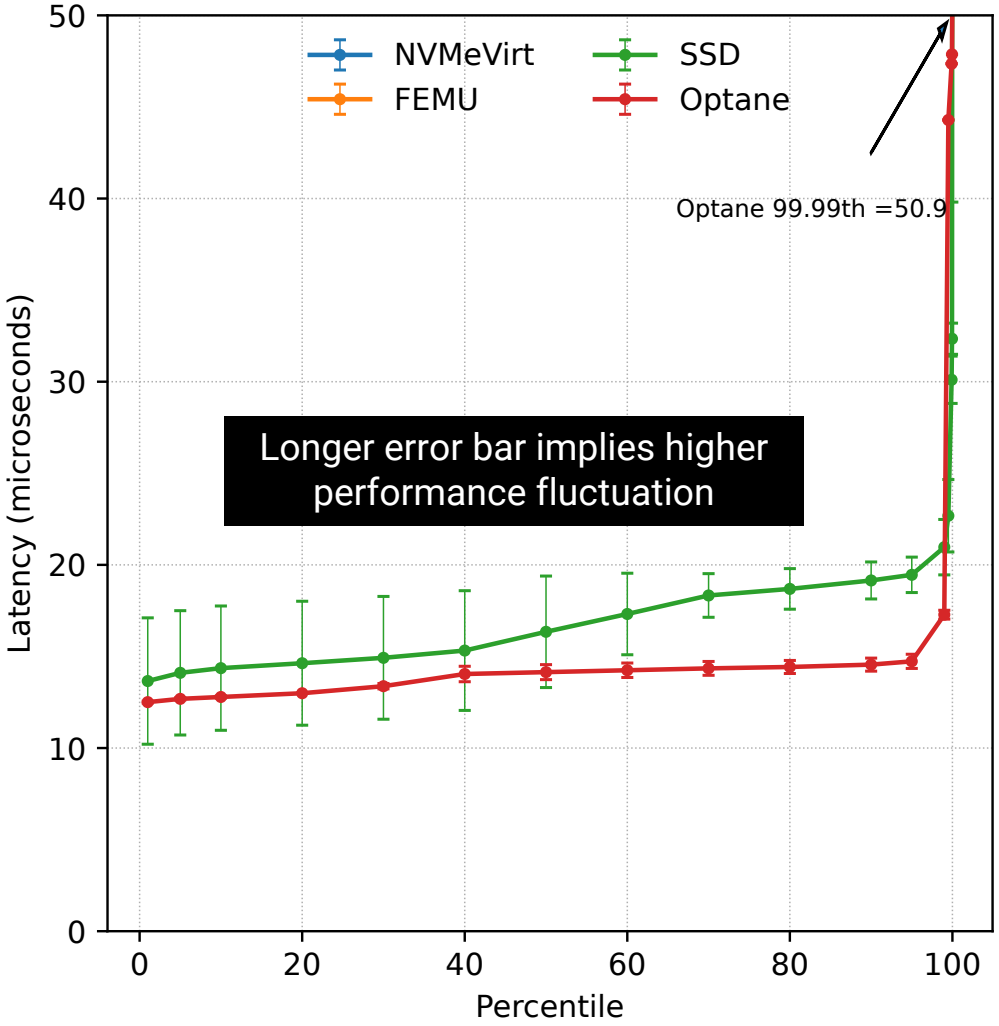
Prototype ZNS SSD

- 96 MiB zones
- 192 KiB write unit
- 32 TB

Ultrastar DC ZN540

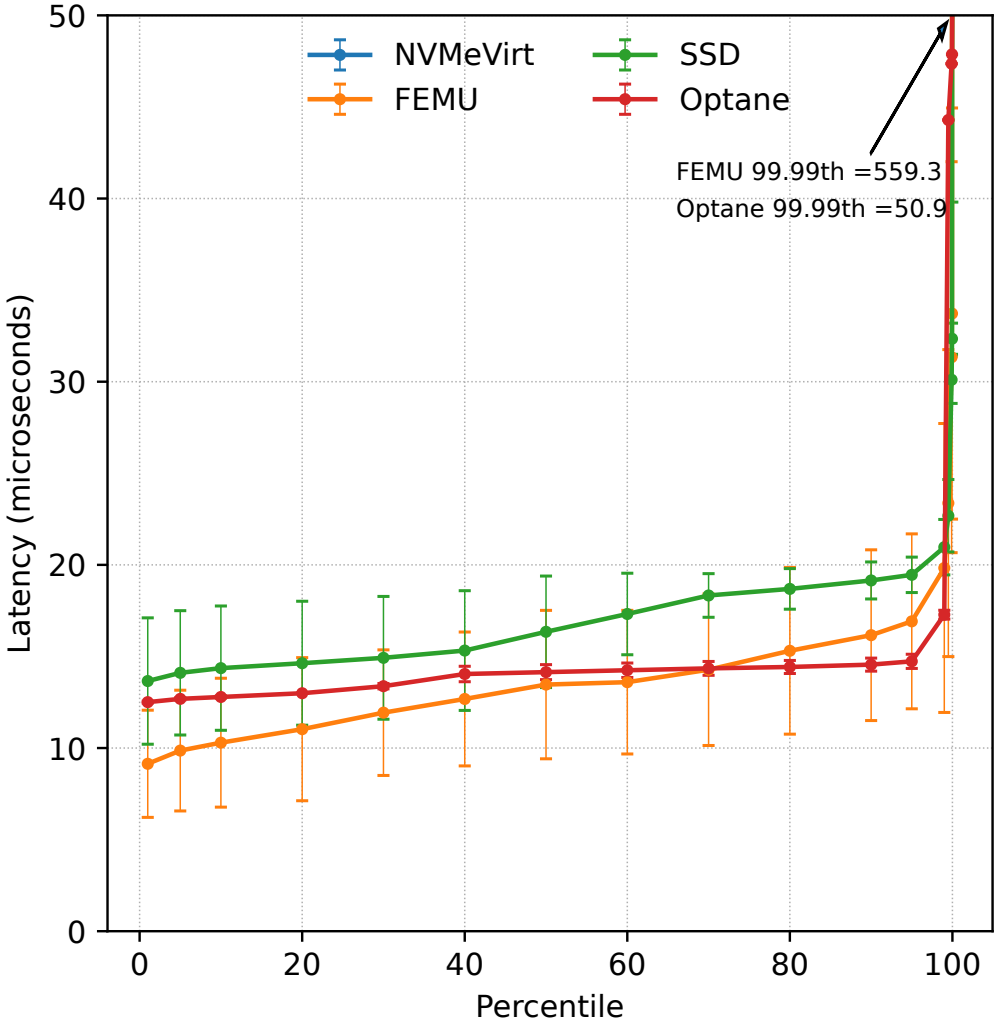
- 2048 MiB zones
- Support write buffer
- 4 TB

Emulation Quality: Performance Variance



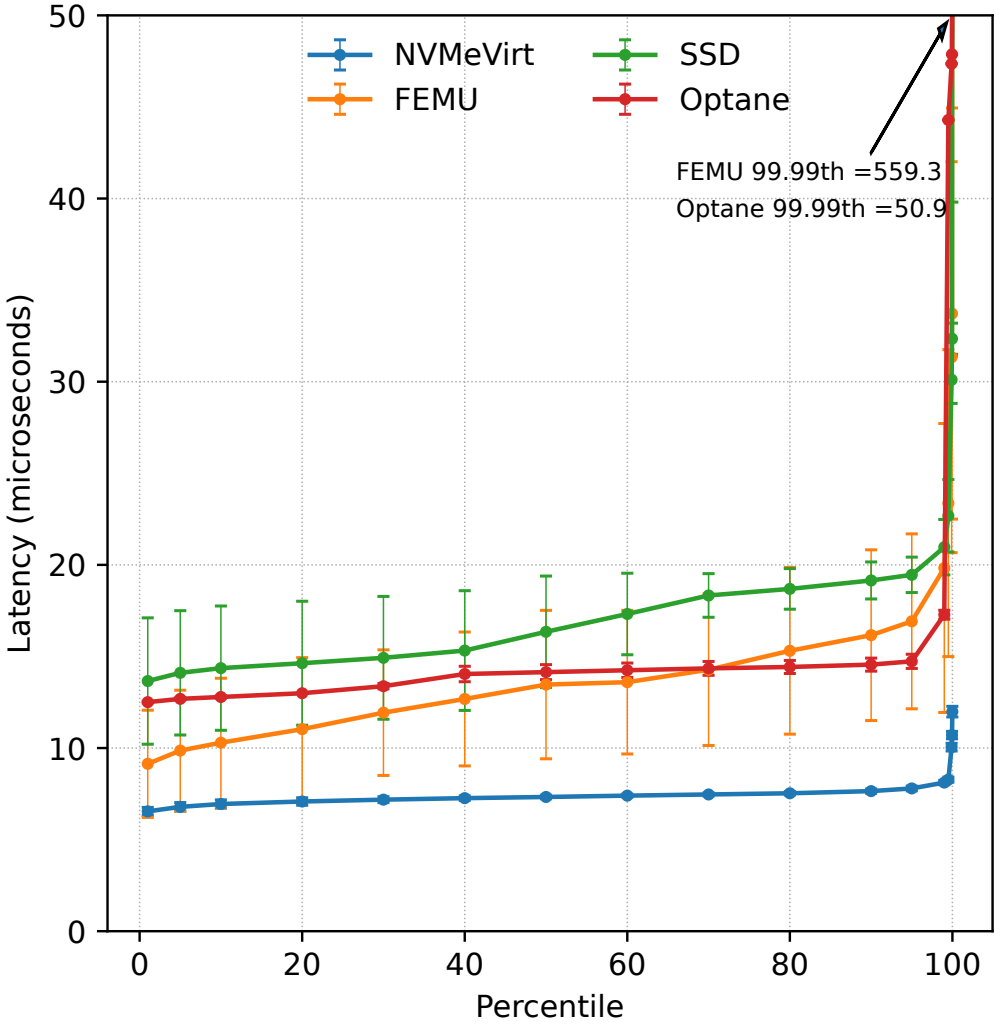
- Distribution of percentiles for 10 runs
  - Each run does 4 KiB random writes with fio
  - Error bar indicates the standard deviation for the percentile

Emulation Quality: Performance Variance



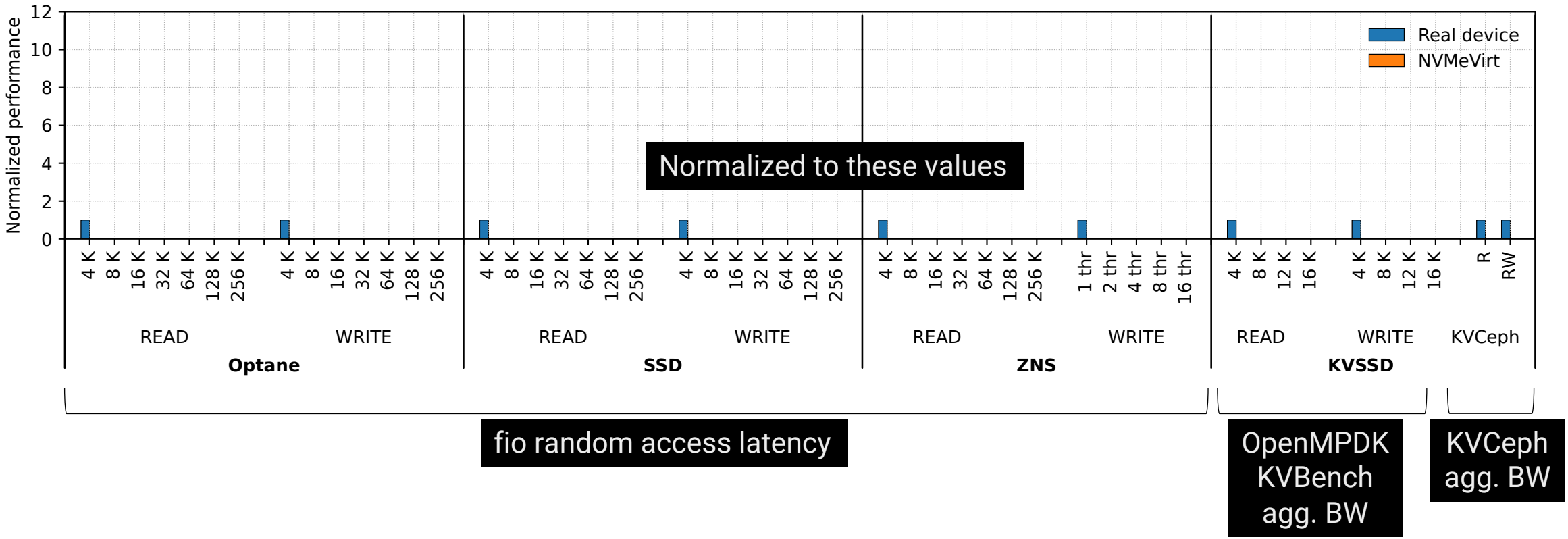
- Distribution of percentiles for 10 runs
  - Each run does 4 KiB random writes with fio
  - Error bar indicates the standard deviation for the percentile
- FEMU exhibits a long tail latency and high run-by-run performance fluctuation
- FEMU would not be able to consistently emulate high-performance NVM SSDs

Emulation Quality: Performance Variance

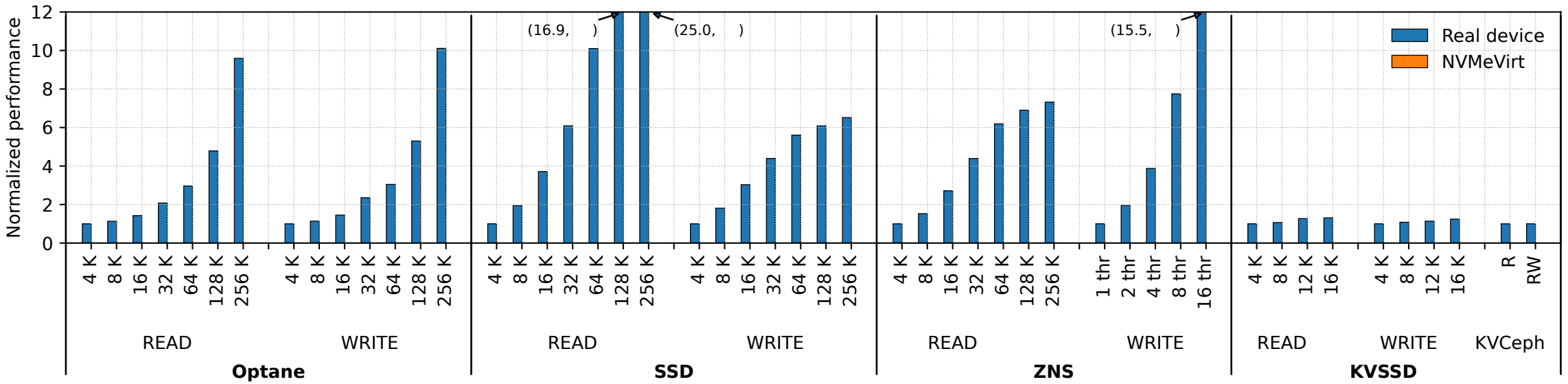


- Distribution of percentiles for 10 runs
  - Each run does 4 KiB random writes with fio
  - Error bar indicates the standard deviation for the percentile
- FEMU exhibits a long tail latency and high run-by-run performance fluctuation
- FEMU would not be able to consistently emulate high-performance NVM SSDs
- NVMeVirt provides low latency with little performance variation

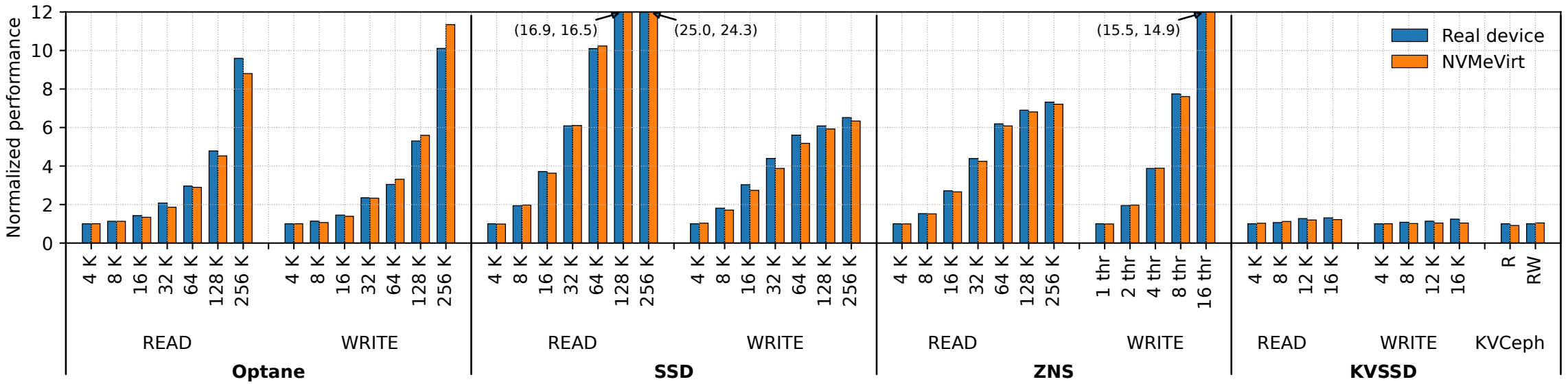
Performance Comparison to Real Devices



Performance Comparison to Real Devices



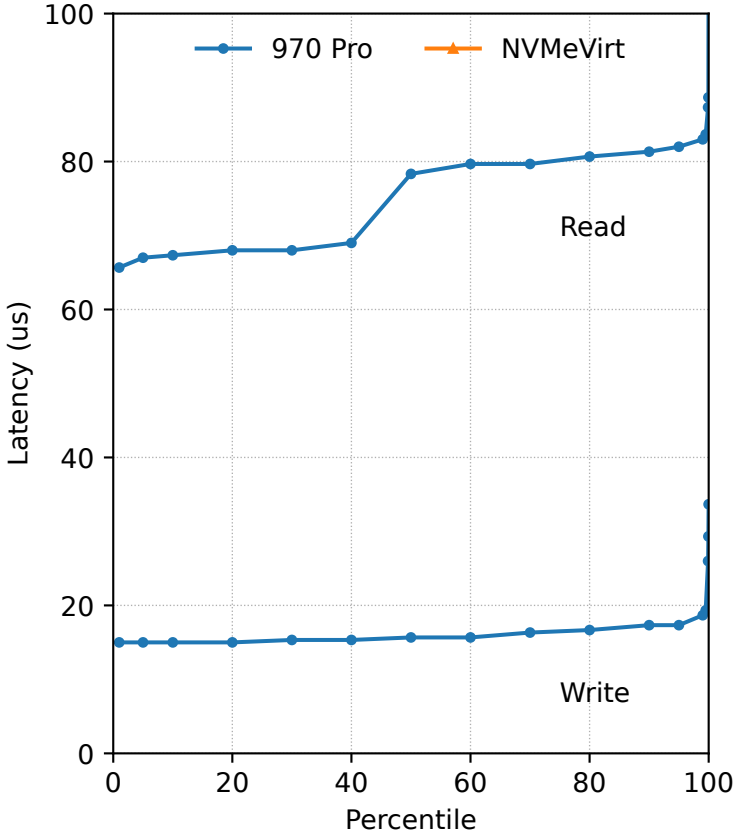
Performance Comparison to Real Devices



NVMeVirt can replicate the real devices' performance closely

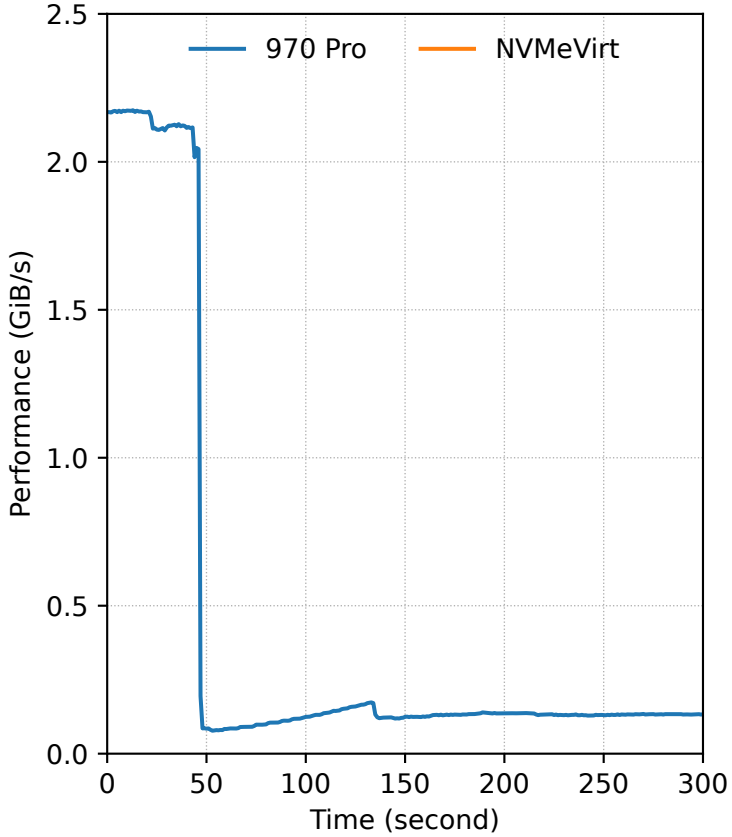
Harmonic mean of performance differences = 1.17%

Characteristics Compared to Real Devices



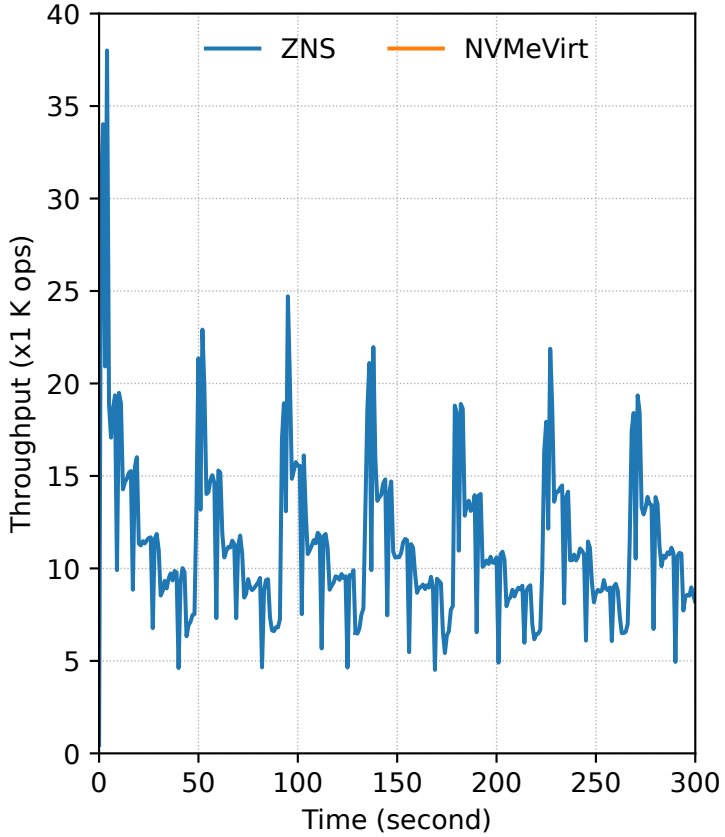
Distributions of latencies

- fio 16 KiB



Performance impact of GC

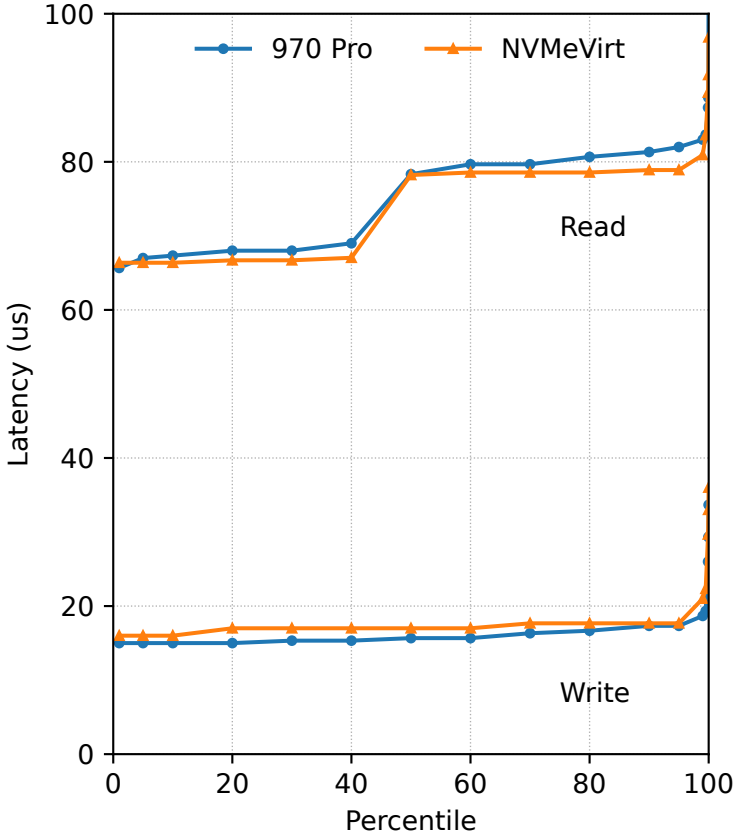
- Fill storage with sequential writes
- Perform random writes to trigger GC



Throughput over time

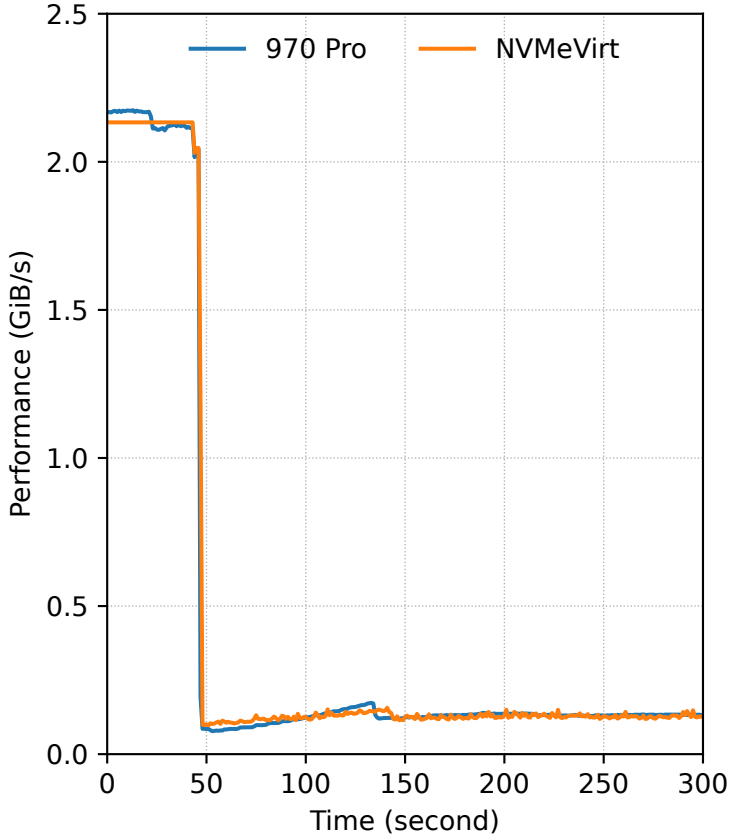
- YCSB-A on RocksDB (50:50 read:update)

Characteristics Compared to Real Devices



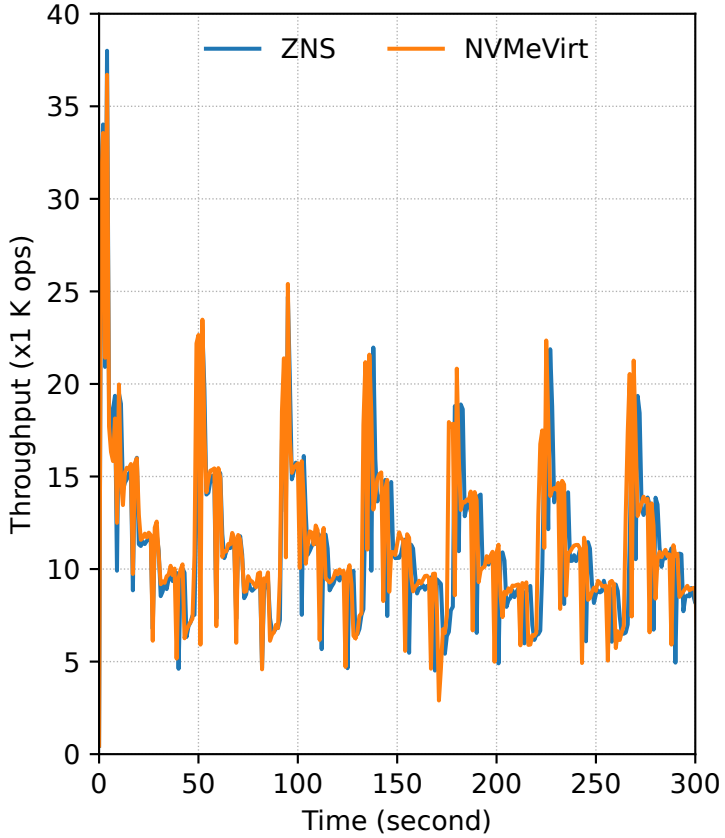
Distributions of latencies

- fio 16 KiB



Performance impact of GC

- Fill storage with sequential writes
- Perform random writes to trigger GC

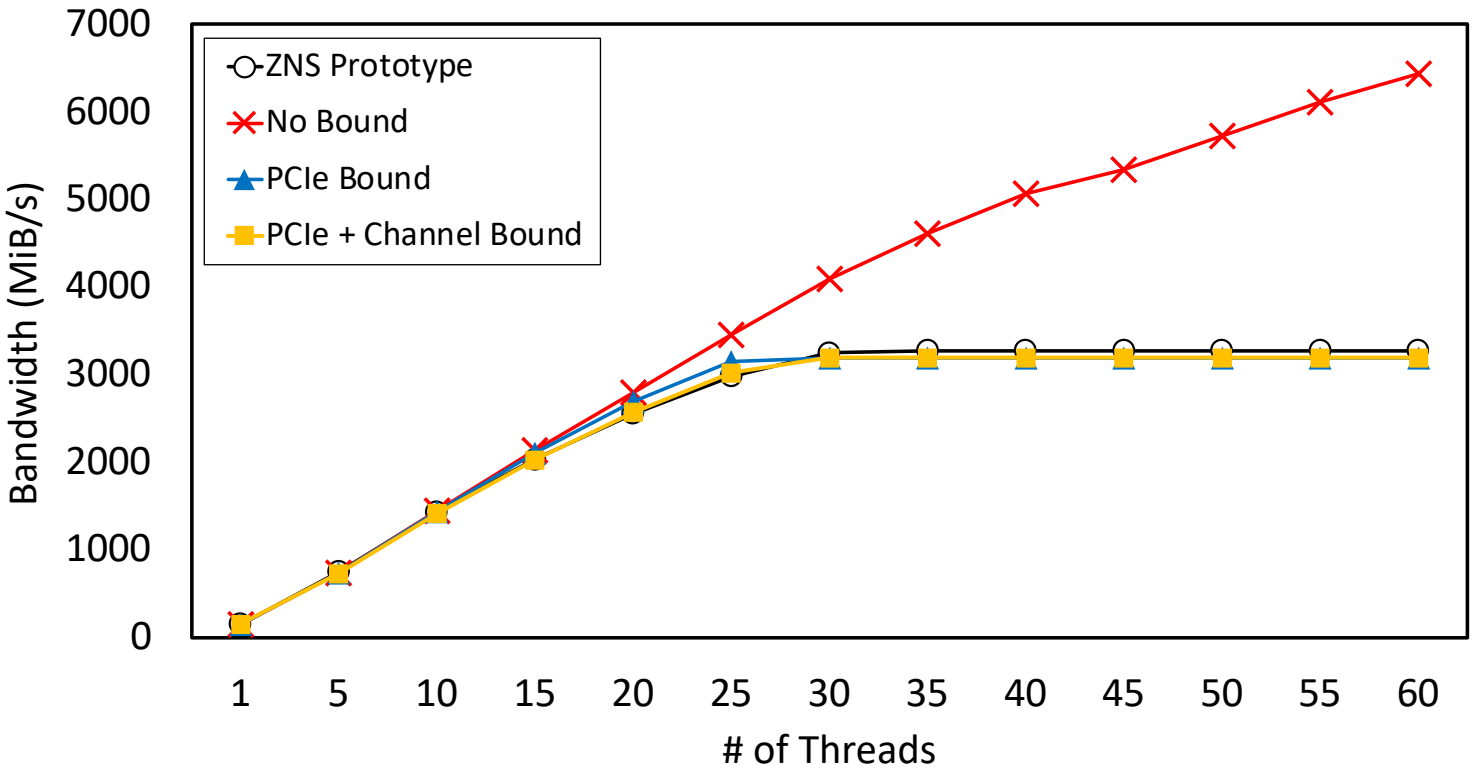


Throughput over time

- YCSB-A on RocksDB (50:50 read:update)

Bandwidth Depending on Contention Modeling

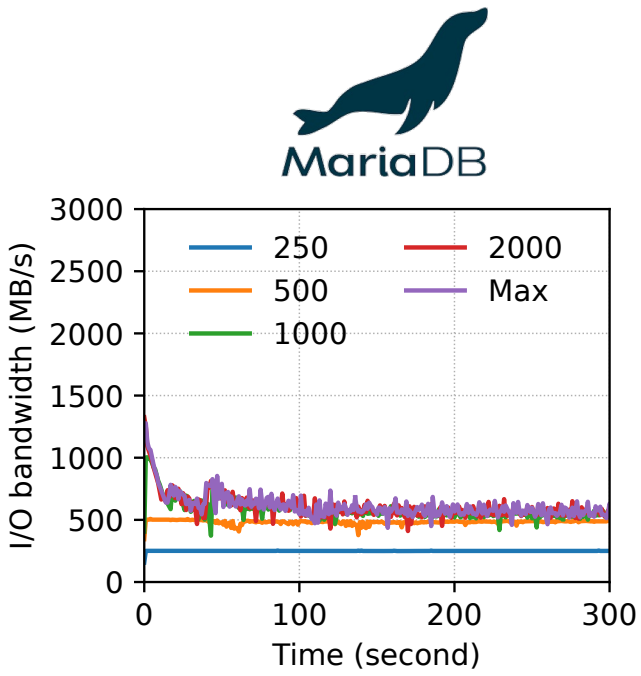
- Fio with increasing number of threads and difference contention modeling

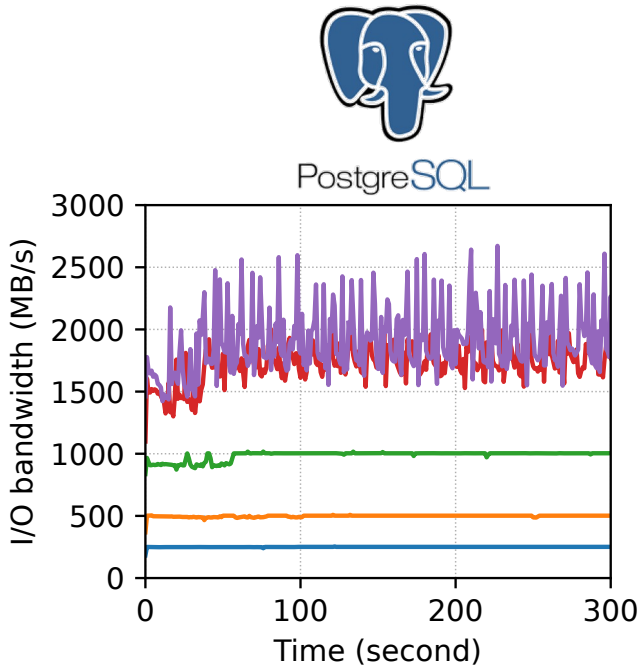


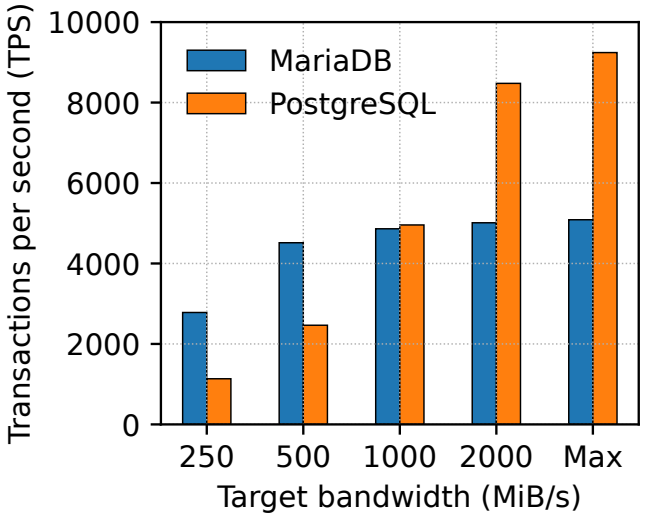
16KiB random read bandwidth

Case Study: DBMS

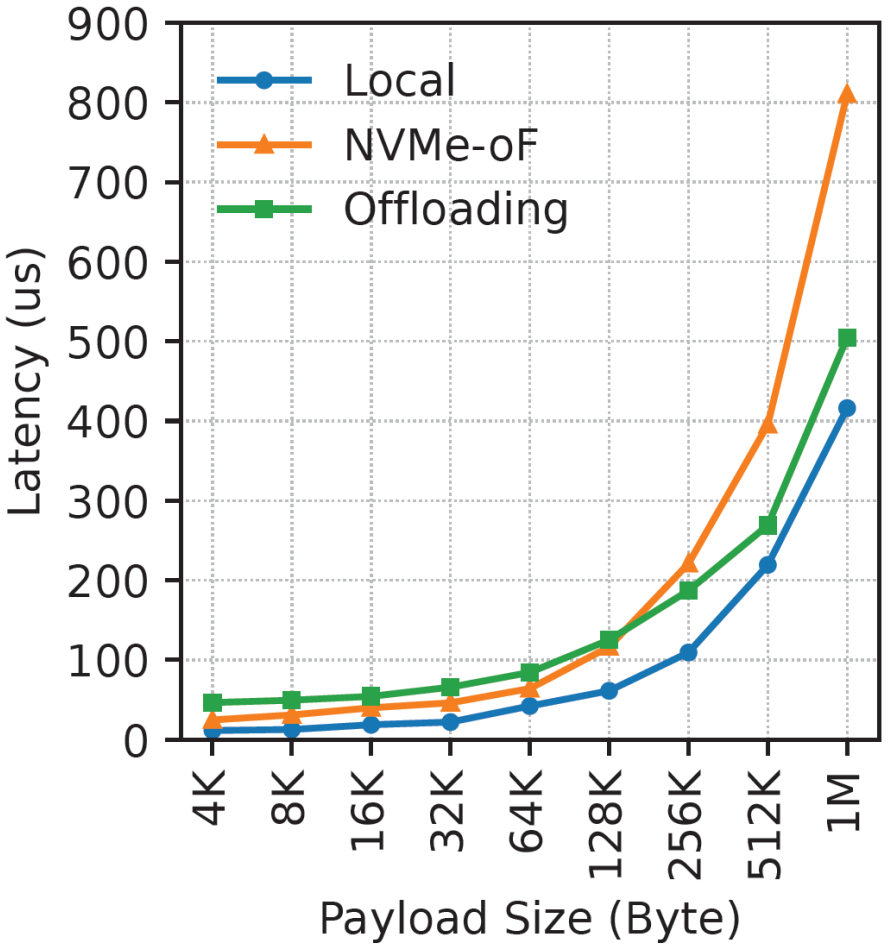
- Sysbench with various bandwidth limits



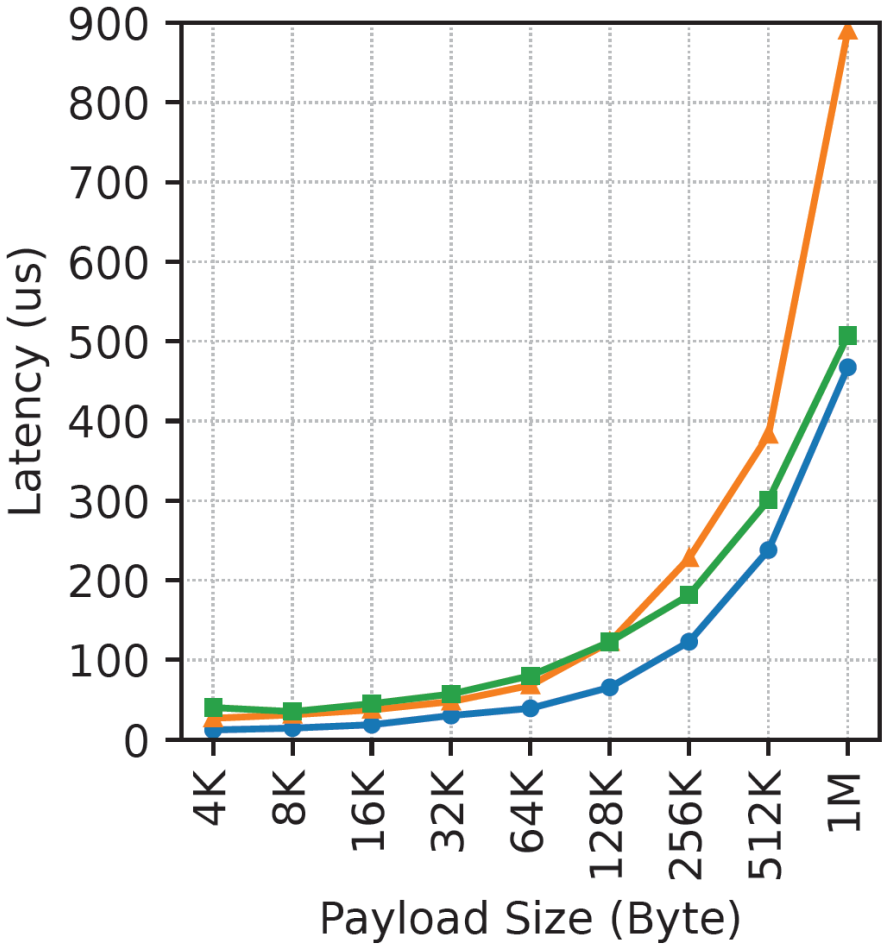




NVMe-oF Write Latency



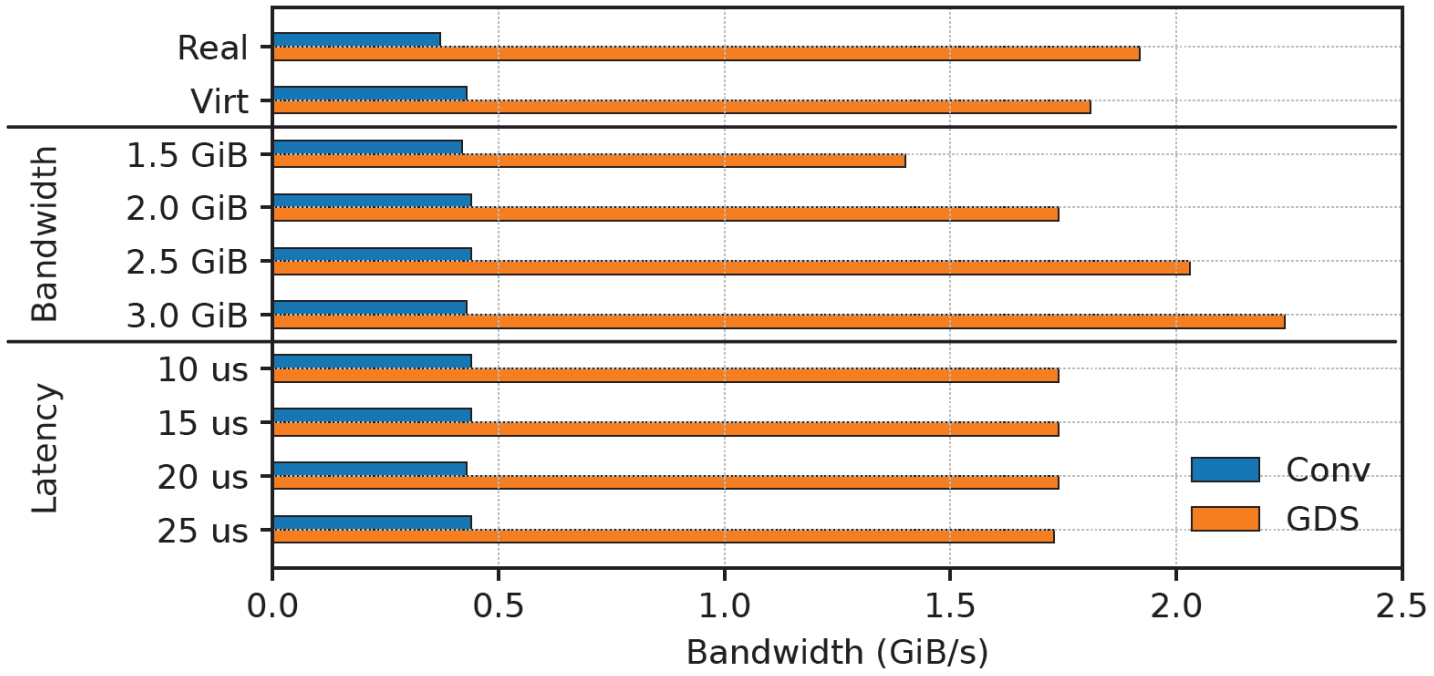
Optane



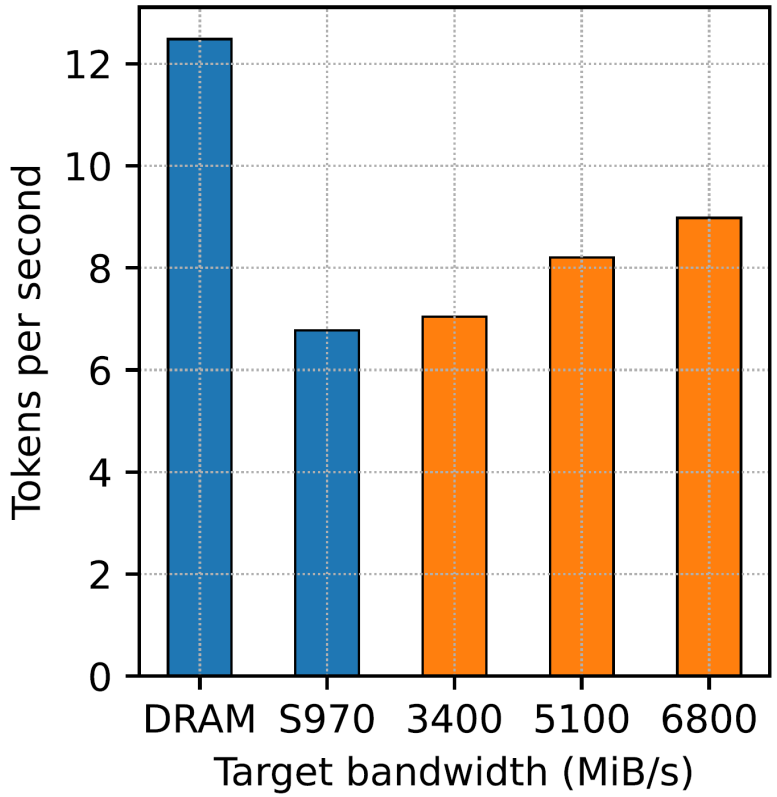
NVMeVirt

GPUDirect Storage (GDS)

- Checkpointing performance of Megatron Deepspeed



- Inference performance of OPT-30B



Use Cases

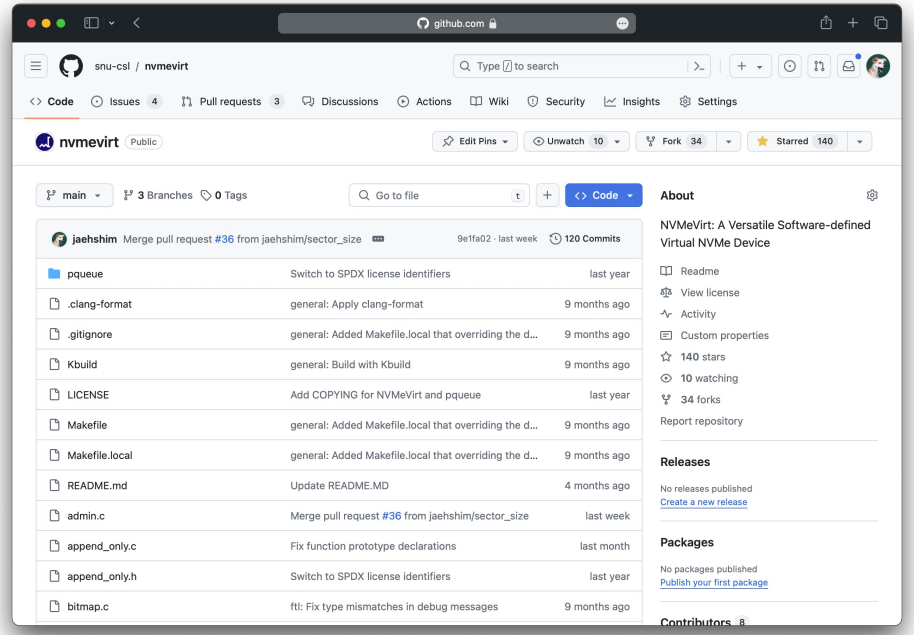
- Fast prototyping for new NVMe interface extensions
- Finding and improving software bottlenecks in the storage stack
- Analyzing application's scalability on future high-performance storage devices
- Investigating performance impact of hardware parameters (e.g. MDTS)
- Developing a new device-centric architecture
- Benchmarking and performance testing & monitoring

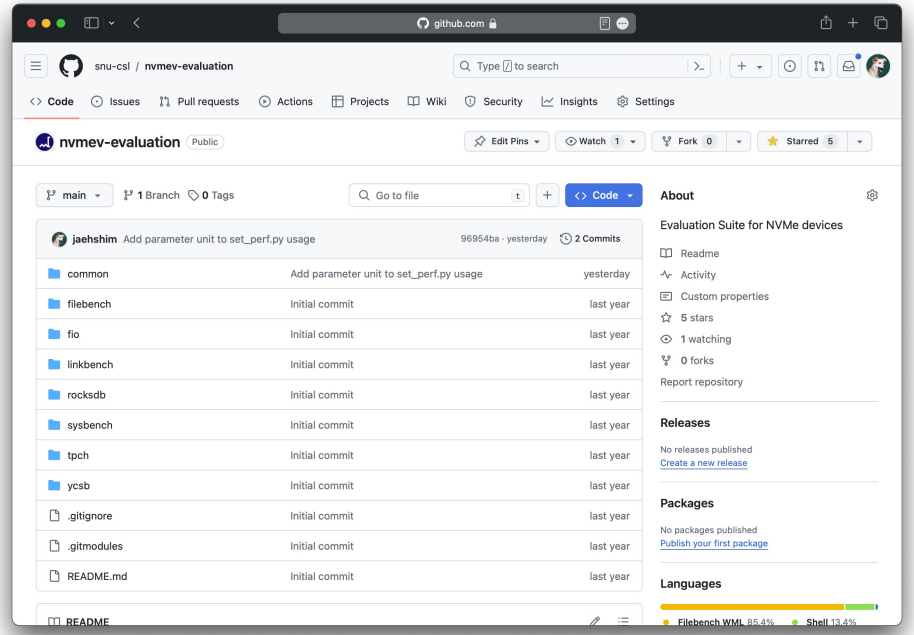
Code Walkthrough

Code Repository

- Codes are currently available at Github!

- NVMeVirt: https://github.com/snu-csl/nvmevirt
- NVMeV-Evaluation: https://github.com/snu-csl/nvmev-evaluation





NVMeVirt Code Architecture

- nvme.h
- nvmev.h
- ssd_config.h
- pci.c(.h)
- admin.c
- main.c
- io.c
- dma.c(.h)

common

- simple_ftl.c(.h)

NVM

- kv_ftl.c(.h)
- nvme_kv.h
- append_only.c(.h)
- bitmap.c(.h)

key-value

- conv_ftl.c(.h)
- ssd.c(.h)
- channel_model.c(.h)

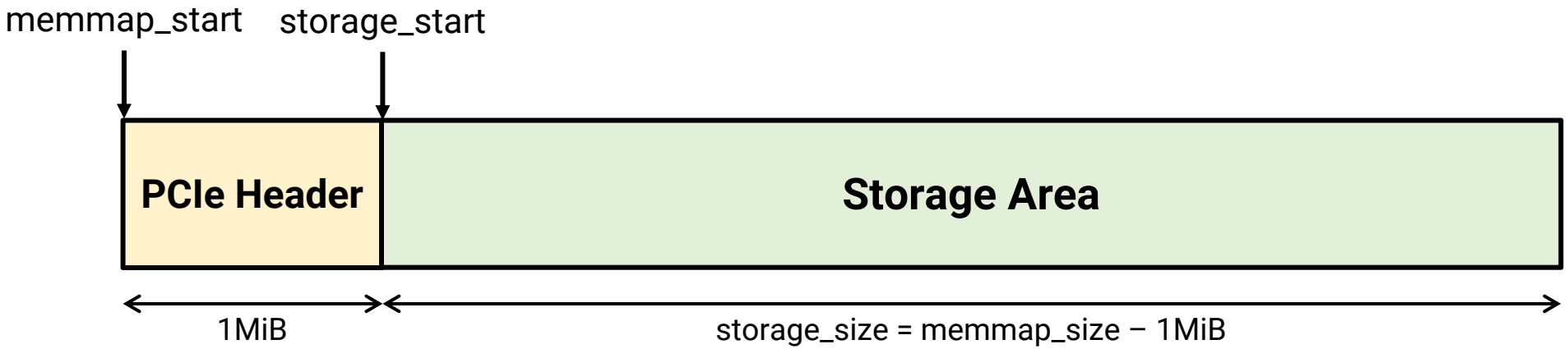
conventional

- zns_ftl.c(.h)
- nvme_zns.h
- ssd.c(.h)
- channel_model.c(.h)
- zns_mgmt_recv.c
- zns_mgmt_send.c
- zns_read_write.c

zoned namespace

(0) Backend Data Storage Initialization

- Extensive amount of memory is needed for data storage
- On system initialization, a part of physical address space must be reserved for NVMeVirt
- Beginning of the reserved memory area are used for NVMe control block and PCI resources



(0) Namespace Initialization

- Initialize Namespaces depending on the configuration of NR_NAMESPACES and NS_SSD_TYPE

```c
void NVMEV_NAMESPACE_INIT(struct nvmev_dev *nvmev_vdev)
{
    unsigned long long remaining_capacity = nvmev_vdev->config.storage_size;
    void *ns_addr = nvmev_vdev->storage_mapped;
    const int nr_ns = NR_NAMESPACES; // XXX: allow for dynamic nr_ns
    const unsigned int disp_no = nvmev_vdev->config.cpu_nr_dispatcher;
    int i;
    unsigned long long size;

    struct nvmev_ns *ns = kmalloc(sizeof(struct nvmev_ns) * nr_ns, GFP_KERNEL)

    for (i = 0; i < nr_ns; i++) {
        if (NS_CAPACITY(i) == 0)
            size = remaining_capacity;
        else
            size = min(NS_CAPACITY(i), remaining_capacity);

        if (NS_SSD_TYPE(i) == SSD_TYPE_NVM)
            simple_init_namespace(&ns[i], i, size, ns_addr, disp_no);
        else if (NS_SSD_TYPE(i) == SSD_TYPE_CONV)
            conv_init_namespace(&ns[i], i, size, ns_addr, disp_no);
        else if (NS_SSD_TYPE(i) == SSD_TYPE_ZNS)
            zns_init_namespace(&ns[i], i, size, ns_addr, disp_no);
        else if (NS_SSD_TYPE(i) == SSD_TYPE_KV)
            kv_init_namespace(&ns[i], i, size, ns_addr, disp_no);
        else
            BUG_ON(1);

        remaining_capacity -= size;
        ns_addr += size;
        NVMEV_INFO("ns %d/%d: size %lld MiB\n", i, nr_ns, BYTE_TO_MB(ns[i].size
    }

    nvmev_vdev->ns = ns;
    nvmev_vdev->nr_ns = nr_ns;
    nvmev_vdev->mdts = MDTS;
}
```

main.c

(0) Backend Type Specific Initialization

- Register dedicated modeling function for each type of backends
- Register backend-dedicated I/O handling function if needed
  - KVSSD needs special handling/identifying function for key-value pairs
- Initialize backend specific metadata
  - Page mapping table, KV hash table, zone metadata, etc

```c
void simple_init_namespace(struct nvmev_ns *ns, uint32_t id, uint64_t size, void *mapped_addr,
                uint32_t cpu_nr_dispatcher)
{
    ns->id = id;
    ns->csi = NVME_CSI_NVM;
    ns->size = size;
    ns->mapped = mapped_addr;
    ns->proc_io_cmd = simple_proc_nvme_io_cmd;

    return;
}
```

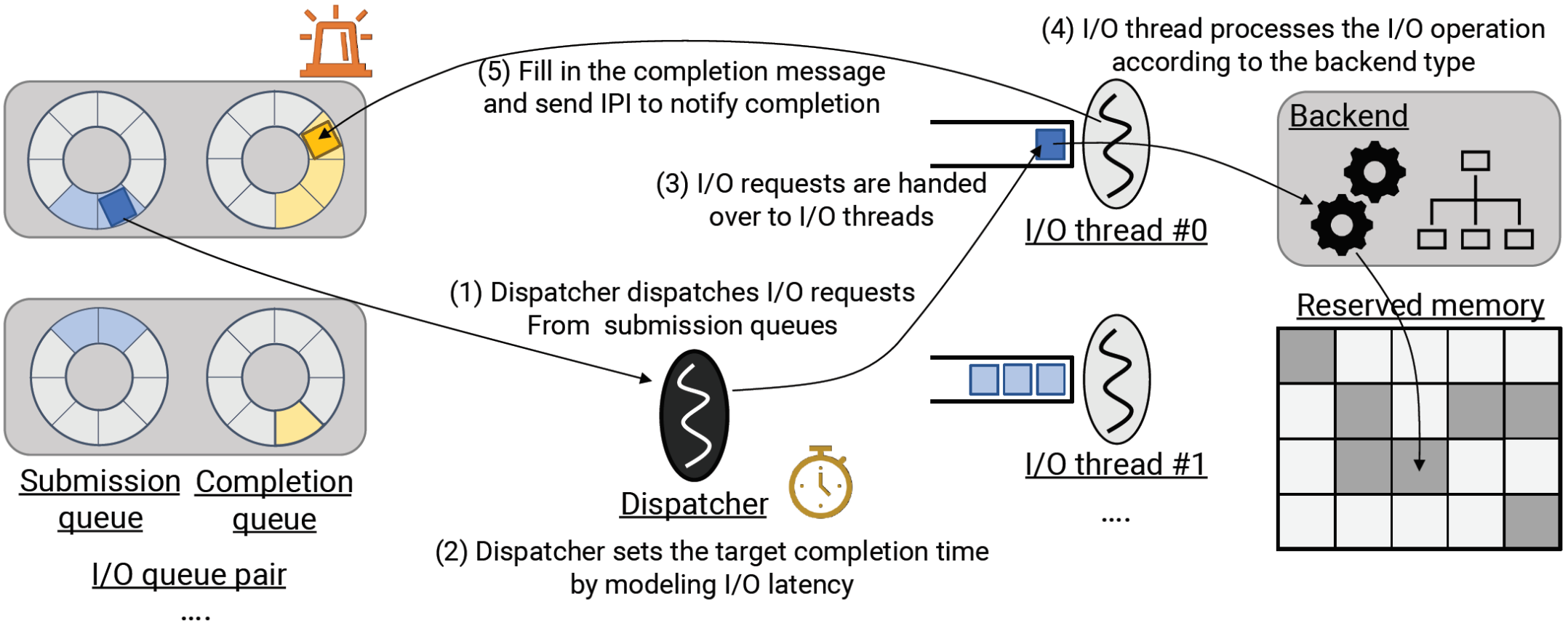
simple_ftl.c

```c
void kv_init_namespace(struct nvmev_ns *ns, uint32_t id, uint64_t size,
                void *mapped_addr, uint32_t cpu_nr_dispatcher)
{
    struct kv_ftl *kv_ftl;
    int i;

    kv_ftl = kmalloc(sizeof(struct kv_ftl), GFP_KERNEL);

    kv_ftl->kv_mapping_table =
        memremap(nvmev_vdev->config.storage_start + nvmev_vdev->config.storage_size,
            KV_MAPPING_TABLE_SIZE, MEMREMAP_WB);

    /* Config KV Mapping Tables */
    ...

    ns->id = id;
    ns->csi = NVME_CSI_NVM; // Not specifying to KV. Need to support NVM commands too.
    ns->ftls = (void *)kv_ftl;
    ns->size = size;
    ns->mapped = mapped_addr;
    /*register io command handler*/
    ns->proc_io_cmd = kv_proc_nvme_io_cmd;
    /*register CSS specific io command functions*/
    ns->identify_io_cmd = kv_identify_nvme_io_cmd;
    ns->perform_io_cmd = kv_perform_nvme_io_cmd;

    return;
}
```

kv_ftl.c

NVMeVirt I/O Handling

- One dispatcher + multiple I/O threads (each pinned to a core)



(1) Dispatch Requests From SQ

- When the SQ doorbell value has changed, dispatcher dispatches requests from the corresponding SQ

```c
static void nvmev_proc_dbs(void)
{
    ...

    // Submission queues
    for (qid = 1; qid <= nvmev_vdev->nr_sq; qid++) {
        if (nvmev_vdev->sqes[qid] == NULL)
            continue;
        dbs_idx = qid * 2;
        new_db = nvmev_vdev->dbs[dbs_idx];
        old_db = nvmev_vdev->old_dbs[dbs_idx];
        if (new_db != old_db) {
            nvmev_vdev->old_dbs[dbs_idx] = nvmev_proc_io_sq(qid, new_db, old_db);
        }
    }

    ...
}
```

main.c

```c
int nvmev_proc_io_sq(int sqid, int new_db, int old_db)
{
    struct nvmev_submission_queue *sq = nvmev_vdev->sqes[sqid];
    int num_proc = new_db - old_db;
    int seq;
    int sq_entry = old_db;
    int latest_db;

    if (unlikely(!sq))
        return old_db;
    if (unlikely(num_proc < 0))
        num_proc += sq->queue_size;

    for (seq = 0; seq < num_proc; seq++) {
        size_t io_size;
        if (!__nvmev_proc_io(sqid, sq_entry, &io_size))
            break;

        if (++sq_entry == sq->queue_size) {
            sq_entry = 0;
        }
        sq->stat.nr_dispatched++;
        sq->stat.nr_in_flight++;
        sq->stat.total_io += io_size;
    }
    sq->stat.nr_dispatch++;
    sq->stat.max_nr_in_flight = max_t(int, sq->stat.max_nr_in_flight, sq->stat.nr_in_flight);

    latest_db = (old_db + seq) % sq->queue_size;
    return latest_db;
}
```

io.c

(2) Set Target Completion Time

- Calculate and attach timestamps to the I/O request

```c
static size_t __nvmev_proc_io(int sqid, int sq_entry, size_t *io_size)
{
    struct nvmev_submission_queue *sq = nvmev_vdev->sqes[sqid];
    unsigned long long nsecs_start = __get_wallclock();
    struct nvme_command *cmd = &sq_entry(sq_entry);
#if (BASE_SSD == KV_PROTOTYPE)
    uint32_t nsid = 0; // Some KVSSD programs give 0 as nsid for KV IO
#else
    uint32_t nsid = cmd->common.nsid - 1;
#endif
    struct nvmev_ns *ns = &nvmev_vdev->ns[nsid];

    struct nvmev_request req = {
        .cmd = cmd,
        .sq_id = sqid,
        .nsecs_start = nsecs_start,
    };
    struct nvmev_result ret = {
        .nsecs_target = nsecs_start,
        .status = NVME_SC_SUCCESS,
    };

    if (!ns->proc_io_cmd(ns, &req, &ret))
        return false;
    *io_size = (cmd->rw.length + 1) << 9;

    __enqueue_io_req(sqid, sq->cqid, sq_entry, nsecs_start, &ret);

    __reclaim_completed_reqs();

    return true;
}
```

io.c

```c
bool conv_proc_nvme_io_cmd(struct nvmev_ns *ns, struct nvmev_request *req, struct nvmev_result *ret)
{
    struct nvme_command *cmd = req->cmd;

    NVMEV_ASSERT(ns->csi == NVME_CSI_NVM);

    switch (cmd->common.opcode) {
    case nvme_cmd_write:
        if (!conv_write(ns, req, ret))
            return false;
        break;
    case nvme_cmd_read:
        if (!conv_read(ns, req, ret))
            return false;
        break;
    case nvme_cmd_flush:
        conv_flush(ns, req, ret);
        break;
    default:
        NVMEV_ERROR("%s: command not implemented: %s (0x%x)\n", __func__,
                nvme_opcode_string(cmd->common.opcode), cmd->common.opcode);
        break;
    }

    return true;
}
```

conv_ftl.c

(3) Send Requests to I/O Worker

- Enqueue the timestamped I/O request to work queue

```c
static size_t __nvmev_proc_io(int sqid, int sq_entry, size_t *io_size)
{
    struct nvmev_submission_queue *sq = nvmev_vdev->sqes[sqid];
    unsigned long long nsecs_start = __get_wallclock();
    struct nvme_command *cmd = &sq_entry(sq_entry);
#if (BASE_SSD == KV_PROTOTYPE)
    uint32_t nsid = 0; // Some KVSSD programs give 0 as nsid for KV IO
#else
    uint32_t nsid = cmd->common.nsid - 1;
#endif
    struct nvmev_ns *ns = &nvmev_vdev->ns[nsid];

    struct nvmev_request req = {
        .cmd = cmd,
        .sq_id = sqid,
        .nsecs_start = nsecs_start,
    };
    struct nvmev_result ret = {
        .nsecs_target = nsecs_start,
        .status = NVME_SC_SUCCESS,
    };

    if (!ns->proc_io_cmd(ns, &req, &ret))
        return false;
    *io_size = (cmd->rw.length + 1) << 9;

    __enqueue_io_req(sqid, sq->cqid, sq_entry, nsecs_start, &ret);

    __reclaim_completed_reqs();

    return true;
}
```

io.c

```c
static void __enqueue_io_req(int sqid, int cqid, int sq_entry,
                             unsigned long long nsecs_start, struct nvmev_result *ret)
{
    struct nvmev_submission_queue *sq = nvmev_vdev->sqes[sqid];
    struct nvmev_io_worker *worker;
    struct nvmev_io_work *w;
    unsigned int entry;

    worker = __allocate_work_queue_entry(sqid, &entry);
    if (!worker)
        return;

    w = worker->work_queue + entry;

    w->sqid = sqid;
    w->cqid = cqid;
    w->sq_entry = sq_entry;
    w->command_id = sq_entry(sq_entry).common.command_id;
    w->nsecs_start = nsecs_start;
    w->nsecs_enqueue = local_clock();
    w->nsecs_target = ret->nsecs_target;
    w->status = ret->status;
    w->is_completed = false;
    w->is_copied = false;
    w->prev = -1;
    w->next = -1;

    w->is_internal = false;
    mb(); /* IO worker shall see the updated w at once */

    __insert_req_sorted(entry, worker, ret->nsecs_target);
}
```

io.c

(4) Perform I/O operation

- Process the I/O operation according to the backend type
  - Write/read data to/from reserved memory using memcpy or DMA engine

```
volatile unsigned int curr = worker->io_seq;
while (curr != -1) {
    struct nvmev_io_work *w = &worker->work_queue[curr];

    if (w->is_copied == false) {
        ...
        else if (io_using_dma) {
            __do_perform_io_using_dma(w->sqid, w->sq_entry);
        } else {
#if (BASE_SSD == KV_PROTOTYPE)
            struct nvmev_submission_queue *sq =
                nvmev_vdev->sqes[w->sqid];
            ns = &nvmev_vdev->ns[0];
            if (ns->identify_io_cmd(ns, sq_entry(w->sq_entry))) {
                w->result0 = ns->perform_io_cmd(
                    ns, &sq_entry(w->sq_entry), &(w->status));
            } else {
                __do_perform_io(w->sqid, w->sq_entry);
            }
#else
            __do_perform_io(w->sqid, w->sq_entry);
#endif
        }
        w->is_copied = true;
    }
```

nvmev_io_worker() in io.c

```
static unsigned int __do_perform_io(int sqid, int sq_entry)
{
    ...
    offset = cmd->slba << 9;
    length = (cmd->length + 1) << 9;
    remaining = length;

    while (remaining) {
        ...
        if (prp_offs == 1) {
            paddr = cmd->prp1;
        } else if (prp_offs == 2) {
            paddr = cmd->prp2;
            if (remaining > PAGE_SIZE) {
                paddr_list = kmap_atomic_pfn(PRP_PFN(paddr)) + (paddr & PAGE_OFFSET_MASK);
                paddr = paddr_list[prp2_offs++];
            }
        } else {
            paddr = paddr_list[prp2_offs++];
        }
        ...
        if (cmd->opcode == nvme_cmd_write || cmd->opcode == nvme_cmd_zone_append) {
            memcpy(nvmev_vdev->ns[nsid].mapped + offset, vaddr + mem_offs, io_size);
        } else if (cmd->opcode == nvme_cmd_read) {
            memcpy(vaddr + mem_offs, nvmev_vdev->ns[nsid].mapped + offset, io_size);
        }
        remaining -= io_size;
        offset += io_size;
    }
}
```

io.c

(5) Fill in the Completion Message

- Compare the timestamp with current time
- Fill in the completion message and send IPI to notify completion

```
    if (w->nsecs_target <= curr_nsecs) {
        ...
        __fill_cq_result(w);
        ...
        w->is_completed = true;
    }
    curr = w->next;
}

for (qidx = 1; qidx <= nvmev_vdev->nr_cq; qidx++) {
    ...
    if (spin_trylock(&cq->irq_lock)) {
        if (cq->interrupt_ready == true) {
            cq->interrupt_ready = false;
            nvmev_signal_irq(cq->irq_vector);
        }
    }
}
```

Fill the CQ head with command ID, status, results, etc

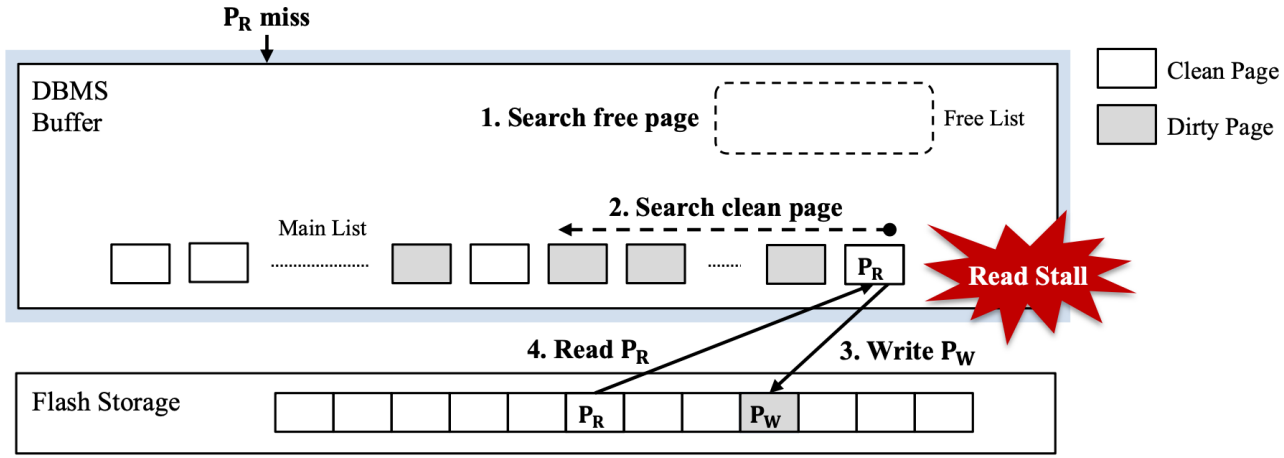
IPI with MSI-X interrupt vector

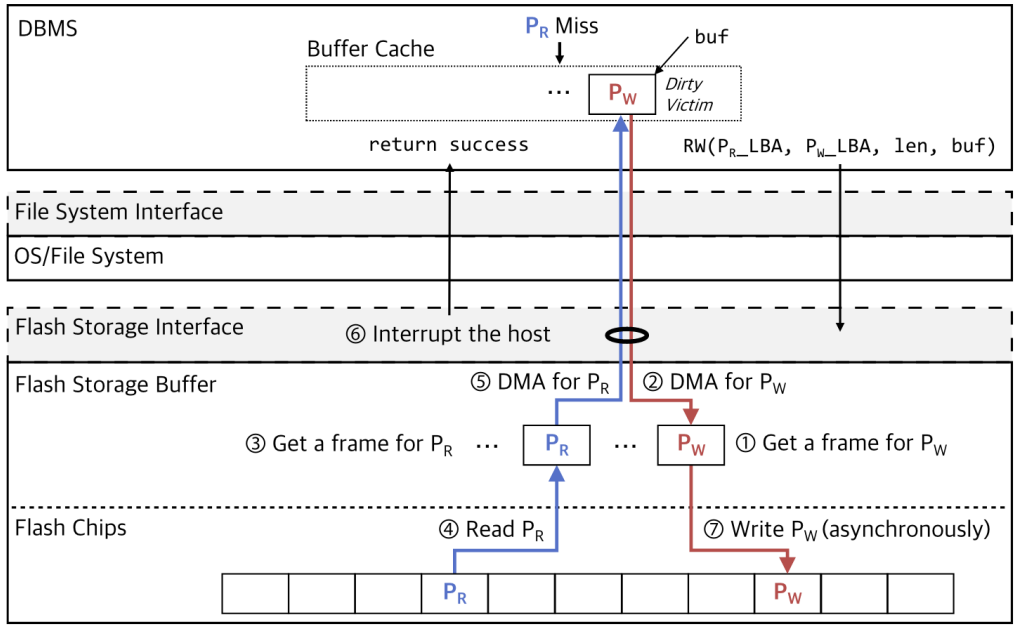
nvmev_io_worker() in io.c

Use Case 1: Implementing New NVMe Command with NVMeVirt

RW command

- Your Read is Our Priority in Flash Storage [VLDB 2022]
- Shared buffer resource between P_R and P_W incurs read stall due to serialized write and read
- RW (Fused Read and Write) command allows the DBMS to request both read and write requests via one I/O call
- Implemented using OpenSSD





Step 1. Extend NVMe Command Set

- Referenced the author's code in: https://github.com/meeeejin/rw-rbuf
- Add RW command opcode to NVME_OPCODES

```c
/* Prepare the read command */
struct nvme_passthru_cmd rw_cmd = {
    .opcode   = 0x66,
    .flags    = 0,
    .rsvd1    = 0,
    .nsid     = 0,
    .cdw2     = 0,
    .cdw3     = 0,
    .metadata = 0,
    .addr     = (__u64)(uintptr_t) buf_w,
    .metadata_len = 0,
    .data_len = (__u32) n_w,
    .cdw10    = (__u32) write_lba,
    .cdw11    = 0,
    .cdw12    = (__u32) nlb_w,
    .cdw13    = (__u32) nlb_r,
    .cdw14    = (__u32) read_lba,
    .cdw15    = 0,
    .timeout_ms = 0,
    .result   = 0,
};
```

os0file.cc in rw-rbuf

```c
#define NVME_OPCODES(op)                \
    op(nvme_cmd_flush, 0x00)            \
    op(nvme_cmd_write, 0x01)            \
    op(nvme_cmd_read, 0x02)             \
    op(nvme_cmd_write_uncor, 0x04)      \
    op(nvme_cmd_compare, 0x05)          \
    op(nvme_cmd_write_zeroes, 0x08)     \
    op(nvme_cmd_dsm, 0x09)              \
    op(nvme_cmd_verify, 0x0c)           \
    op(nvme_cmd_resv_register, 0x0d)    \
    op(nvme_cmd_resv_report, 0x0e)      \
    op(nvme_cmd_resv_acquire, 0x11)     \
    op(nvme_cmd_resv_release, 0x15)     \
    op(nvme_cmd_zone_mgmt_send, 0x79)   \
    op(nvme_cmd_zone_mgmt_recv, 0x7a)   \
    op(nvme_cmd_zone_append, 0x7d)      \
    op(nvme_cmd_kv_store, 0x81)         \
    op(nvme_cmd_kv_append, 0x83)        \
    op(nvme_cmd_kv_retrieve, 0x90)      \
    op(nvme_cmd_kv_delete, 0xA1)        \
    op(nvme_cmd_kv_iter_req, 0xB1)      \
    op(nvme_cmd_kv_iter_read, 0xB2)     \
    op(nvme_cmd_kv_exist, 0xB3)         \
    op(nvme_cmd_kv_batch, 0x85)         \
    op(nvme_cmd_read_and_write, 0x66)   \
```

nvmev.h

Step 2. Add RW Target Completion Calculation

- Currently only supported by NVM SSD type

```
switch (cmd->common.opcode) {
case nvme_cmd_write:
case nvme_cmd_read:
    ret->nsecs_target = __schedule_io_units(
        cmd->common.opcode, cmd->rw.slba,
        __cmd_io_size((struct nvme_rw_command *)cmd), __get_wallclock());
    break;
case nvme_cmd_flush:
    ret->nsecs_target = __schedule_flush(req);
    break;
case nvme_cmd_read_and_write: {
    uint64_t write_target;
    unsigned long write_lba;
    unsigned long read_lba;
    unsigned int length;

    write_lba = cmd->common.cdw10[0];
    read_lba = cmd->common.cdw10[4];
    length = (cmd->common.cdw10[2] + 1) << 9;

    write_target = __schedule_io_units(
        nvme_cmd_write, write_lba, length, __get_wallclock());
    ret->nsecs_target = __schedule_io_units(
        nvme_cmd_read, read_lba, length, write_target);
    break;
}
default:
    NVMEV_ERROR("%s: command not implemented: %s (0x%x)\n", __func__,
            nvme_opcode_string(cmd->common.opcode), cmd->common.opcode);
    break;
}
```

simple_proc_nvme_io_cmd() in simple_ftl.c

Step 3. Add RW I/O Handling

- Add a special I/O handling mechanism for RW command

```
...
if (cmd->opcode == nvme_cmd_read_and_write) {
    struct nvme_common_command *common = &sq_entry(sq_entry).common;
    offset = common->cdw10[0] << 9;
    read_offset = common->cdw10[4] << 9;
    length = (common->cdw10[2] + 1) << 9;
    remaining = length;
    NVMEV_INFO("Write offset: %ld, read offset: %ld, length: %ld\n", offset, read_offset, length);
} else {
    offset = cmd->slba << 9;
    length = (cmd->length + 1) << 9;
    remaining = length;
}
```

__do_perform_io() in io.c

```
while (remaining) {
    ...
    if (prp_offs == 1) {
        paddr = cmd->prp1;
    } else if (prp_offs == 2) {
        paddr = cmd->prp2;
        if (remaining > PAGE_SIZE) {
            paddr_list = kmap_atomic_pfn(PRP_PFN(paddr)) +
                         (paddr & PAGE_OFFSET_MASK);
            paddr = paddr_list[prp2_offs++];
        }
    } else {
        paddr = paddr_list[prp2_offs++];
    }
    ...
    if (cmd->opcode == nvme_cmd_write ||
        cmd->opcode == nvme_cmd_zone_append) {
        memcpy(nvmev_vdev->ns[nsid].mapped + offset, vaddr + mem_offs, io_size);
    } else if (cmd->opcode == nvme_cmd_read) {
        memcpy(vaddr + mem_offs, nvmev_vdev->ns[nsid].mapped + offset, io_size);
    } else if (cmd->opcode == nvme_cmd_read_and_write) {
        /* perform write first */
        memcpy(nvmev_vdev->ns[nsid].mapped + offset, vaddr + mem_offs, io_size);
        /* then, perform read */
        memcpy(vaddr + mem_offs, nvmev_vdev->ns[nsid].mapped + read_offset, io_size);
        read_offset += io_size;
    }

    kunmap_atomic(vaddr);

    remaining -= io_size;
    offset += io_size;
}
```

__do_perform_io() in io.c

RW Command Functionality Test

- All codes are in https://github.com/jaehshim/nvmevirt/tree/rw

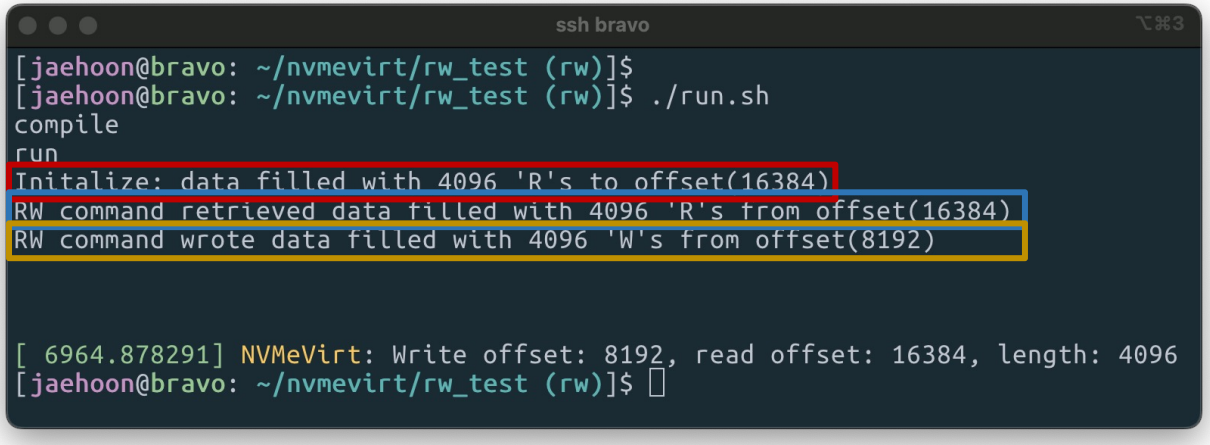
rw_test/rw_test.cc

```
/* Initalize page to read */
ret = pwrite(fd, read_data, buffer_size, read_offset);
printf("Initalize: data filled with %d 'R's to offset(%d)\n", ret, read_offset);
```

```
/* Initalize RW command */
struct nvme_passthru_cmd nvmeCmd;
struct nvme_passthru_cmd *m_nvmeCmd = &nvmeCmd;
memset(m_nvmeCmd, 0, sizeof(struct nvme_passthru_cmd));
m_nvmeCmd->nsid = 1;
m_nvmeCmd->opcode = 0x66;
m_nvmeCmd->addr = (__u64) write_data;
m_nvmeCmd->data_len = buffer_size;
m_nvmeCmd->cdw10 = write_offset / 512;
m_nvmeCmd->cdw12 = buffer_size / 512 - 1;
m_nvmeCmd->cdw13 = buffer_size / 512 - 1;
m_nvmeCmd->cdw14 = read_offset / 512;

ret = ioctl(fd, NVME_IOCTL_IO_CMD, m_nvmeCmd);
if (ret < 0) {
    printf("ioctl failed!! %d\n", ret);
}
```

```
/* Verify RW command READ */
num = 0;
for (int i = 0; i < buffer_size; i++) {
    if (write_data[i] == 'R')
        num++;
}
printf("RW command retrieved data filled with %d 'R's from offset(%d)\n", num, read_offset);
```

```
/* Verify RW command WRITE*/
ret = pread(fd, buffer, buffer_size, write_offset);
num = 0;
for (int i = 0; i < buffer_size; i++) {
    if (buffer[i] == 'W')
        num++;
}
printf("RW command wrote data filled with %d 'W's from offset(%d)\n", num, write_offset);
```



NVMeVirt Configuration Parameters

Extracting Simple Model Parameters

- ./set_perf.py [unit size read latency] [unit size write min latency] [read max bandwidth] {write max bandwidth}
- Parameters can be extracted by using FIO or spec chart
  - I/O unit size SYNC I/O, large size ASYNC I/O, etc

common/set_perf.py

```
minimum_latency = [ 6610, 9870 ] # Minimum nanoseconds for read and write
#minimum_latency = [ 0, 0 ] # Minimum nanoseconds for read and write

if len(sys.argv) == 2 and sys.argv[1] == "max":…
else:
    if len(sys.argv) < 4:…

    target_latency = [ float(sys.argv[1]), float(sys.argv[2]) ]
    target_bw = [ float(sys.argv[3]), float(sys.argv[3]) ]
    if len(sys.argv) == 5:
        target_bw[1] = float(sys.argv[4])


    io_unit_shift = 12
    io_unit_size = 1 << (io_unit_shift - 10)

    # The number of operations per second = bandwidth / per-operation size
    nr_ops = [ bw * 1024 / io_unit_size for bw in target_bw ]

    # Per-operation latency = 1 / # of ops
    per_op_latency = [ 1 / ops * 1000000000 for ops in nr_ops]

    delay_initial = [ max(target * 1000 - per_op - m, 1) for (target, per_op, m) in zip(target_latency, per_op_latency, minimum_latency) ]

os.system("sudo sh -c \"echo %d %d %d > /proc/nvmev/read_times\"" % (delay_initial[0], per_op_latency[0], 0))
os.system("sudo sh -c \"echo %d %d %d > /proc/nvmev/write_times\"" % (delay_initial[1], per_op_latency[1], 0))
os.system("sudo sh -c \"echo %d %d > /proc/nvmev/io_units\"" % (1, io_unit_shift))
```

NVMeVirt SSD Configuration Parameters

- Parallel model parameters are determined at compile time

- Every model parameters are tunable to fit into various vendors' SSDs
  - Number of FTLs
  - Number of channels
  - NAND package organization
  - Channel/PCIe bandwidth
  - Read, program, erase latency
  - Size of OP area
  - Size of write buffer

ssd_config.h

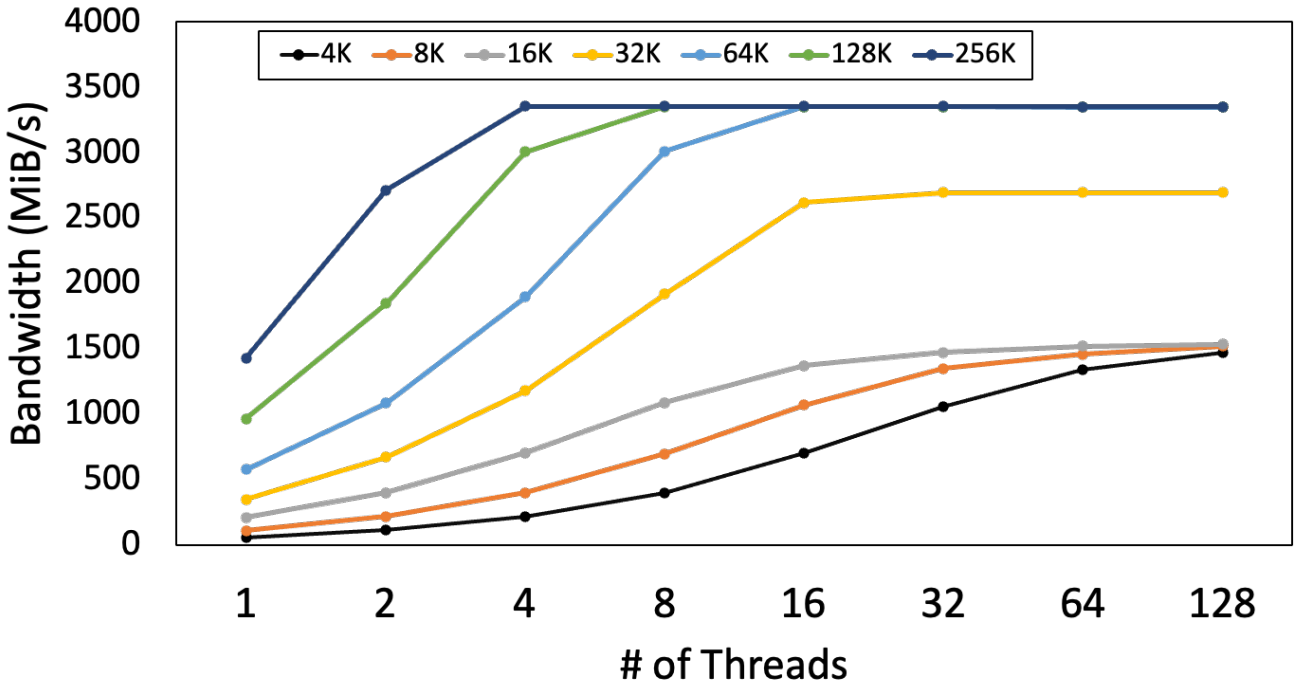
```
#elif (BASE_SSD == SAMSUNG_970PRO)
#define NR_NAMESPACES 1

#define NS_SSD_TYPE_0 SSD_TYPE_CONV
#define NS_CAPACITY_0 (0)
#define NS_SSD_TYPE_1 NS_SSD_TYPE_0
#define NS_CAPACITY_1 (0)
#define MDTS (6)
#define CELL_MODE (CELL_MODE_MLC)

#define SSD_PARTITIONS (4)
#define NAND_CHANNELS (8)
#define LUNS_PER_NAND_CH (2)
#define PLNS_PER_LUN (1)
#define FLASH_PAGE_SIZE KB(32)
#define ONESHOT_PAGE_SIZE (FLASH_PAGE_SIZE * 1)
#define BLKS_PER_PLN (8192)
#define BLK_SIZE (0) /*BLKS_PER_PLN should not be 0 */
static_assert((ONESHOT_PAGE_SIZE % FLASH_PAGE_SIZE) == 0);

#define MAX_CH_XFER_SIZE KB(16) /* to overlap with pcie transfer */
#define WRITE_UNIT_SIZE (512)

#define NAND_CHANNEL_BANDWIDTH (800ull) //MB/s
#define PCIE_BANDWIDTH (3360ull) //MB/s

#define NAND_4KB_READ_LATENCY_LSB (35760 - 6000) //ns
#define NAND_4KB_READ_LATENCY_MSB (35760 + 6000) //ns
#define NAND_4KB_READ_LATENCY_CSB (0) //not used
#define NAND_READ_LATENCY_LSB (36013 - 6000)
#define NAND_READ_LATENCY_MSB (36013 + 6000)
#define NAND_READ_LATENCY_CSB (0) //not used
#define NAND_PROG_LATENCY (185000)
#define NAND_ERASE_LATENCY (0)

#define FW_4KB_READ_LATENCY (21500)
#define FW_READ_LATENCY (30490)
#define FW_WBUF_LATENCY0 (4000)
#define FW_WBUF_LATENCY1 (460)
#define FW_CH_XFER_LATENCY (0)
#define OP_AREA_PERCENT (0.07)

#define GLOBAL_WB_SIZE (NAND_CHANNELS * LUNS_PER_NAND_CH * ONESHOT_PAGE_SIZE * 2)
#define WRITE_EARLY_COMPLETION 1
```

Extracting Parallel Model Parameters

- Use spec chart to know SSD package parameters
  - https://www.techpowerup.com/ssd-specs/samsung-970-pro-512-gb.d54

- Use random read performance to extract NAND related performance parameters
  - Writes are too much optimized

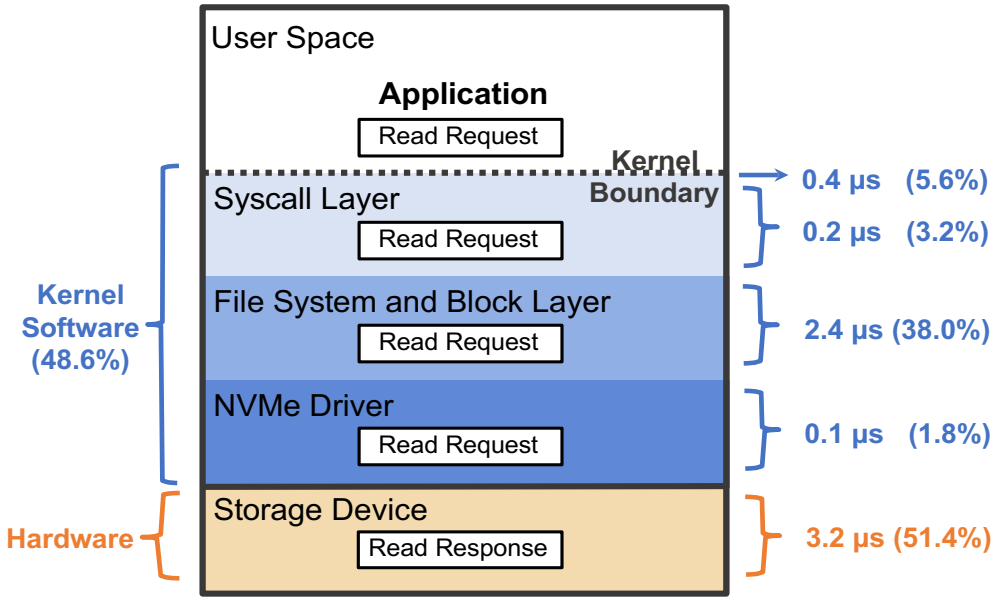
- Performance parameters
  - PCIe bandwidth
  - Channel bandwidth
  - NAND read latency
  - Number of channels

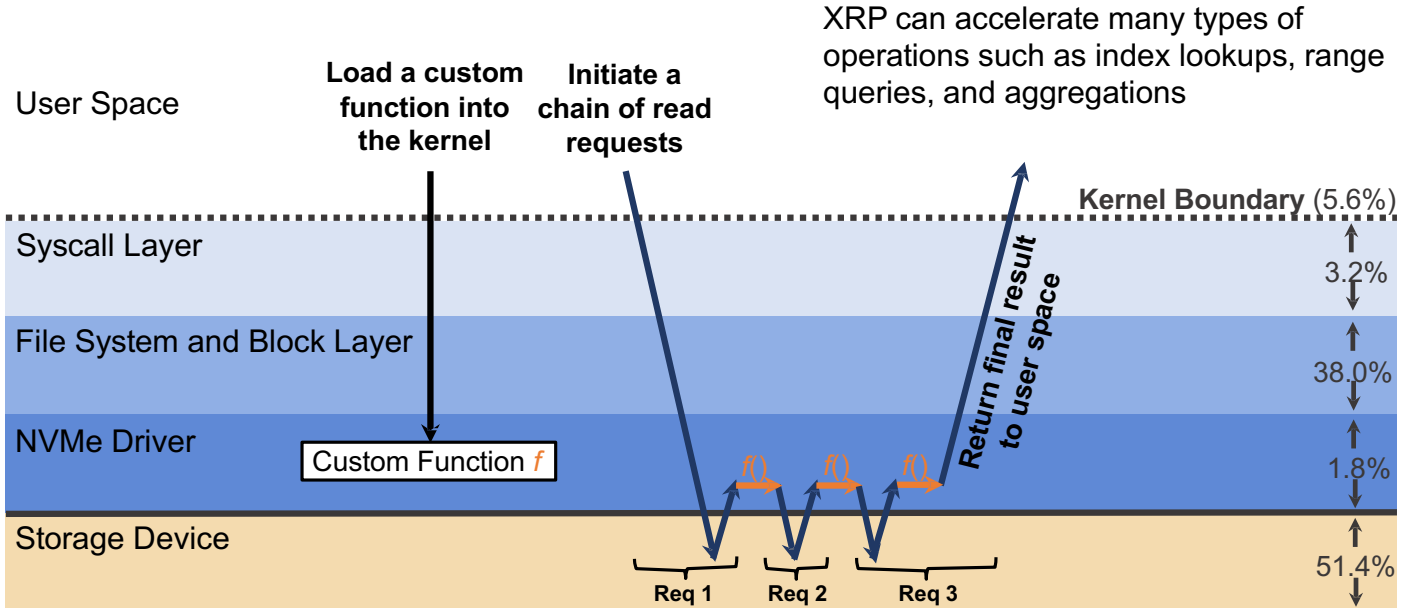


Use Case 2:
XRP Evaluation

XRP

- XRP: In-Kernel Storage Functions with eBPF [OSDI '22]
- Kernel software overhead accounts for ~50% of read latency on Optane SSD Gen 2 (P5800X)
  - Optane SSD Gen 2 latency < 6us



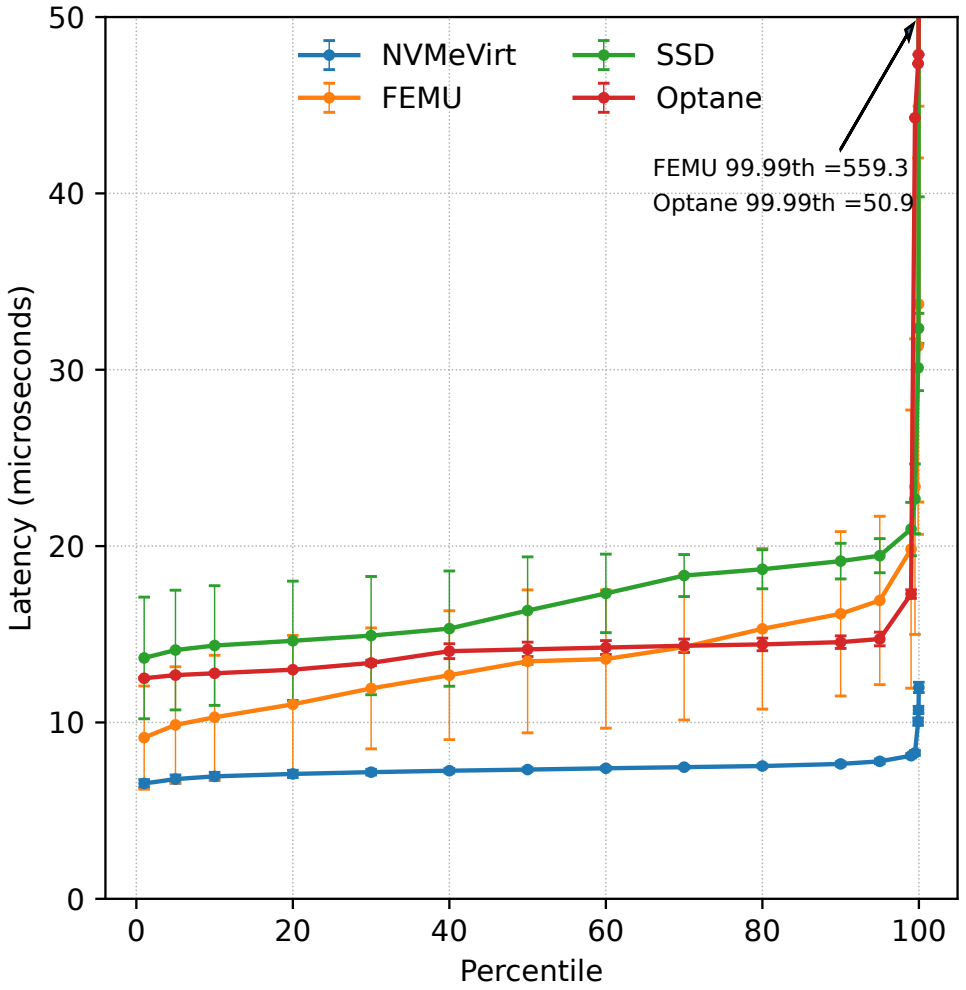


Hurdle for Reproducing XRP Results

- Optane SSD P5800X is hard to access
- Other accessible SSDs:
  - Samsung 970 PRO model's 4K random read performance: 75us
  - Optane SSD Intel P4800X model's 4K random read performance: 12us
- Optane SSD P5800X emulation is possible with NVMeVirt

```
fio_test: (groupid=0, jobs=1): err= 0: pid=82582: Wed Dec 20 09:44:44 2023
  read: IOPS=157k, BW=76.8MiB/s (80.5MB/s)(448MiB/5842msec)
    clat (nsec): min=5368, max=114117, avg=6092.56, stdev=262.75
     lat (nsec): min=5398, max=114147, avg=6122.12, stdev=262.90
```

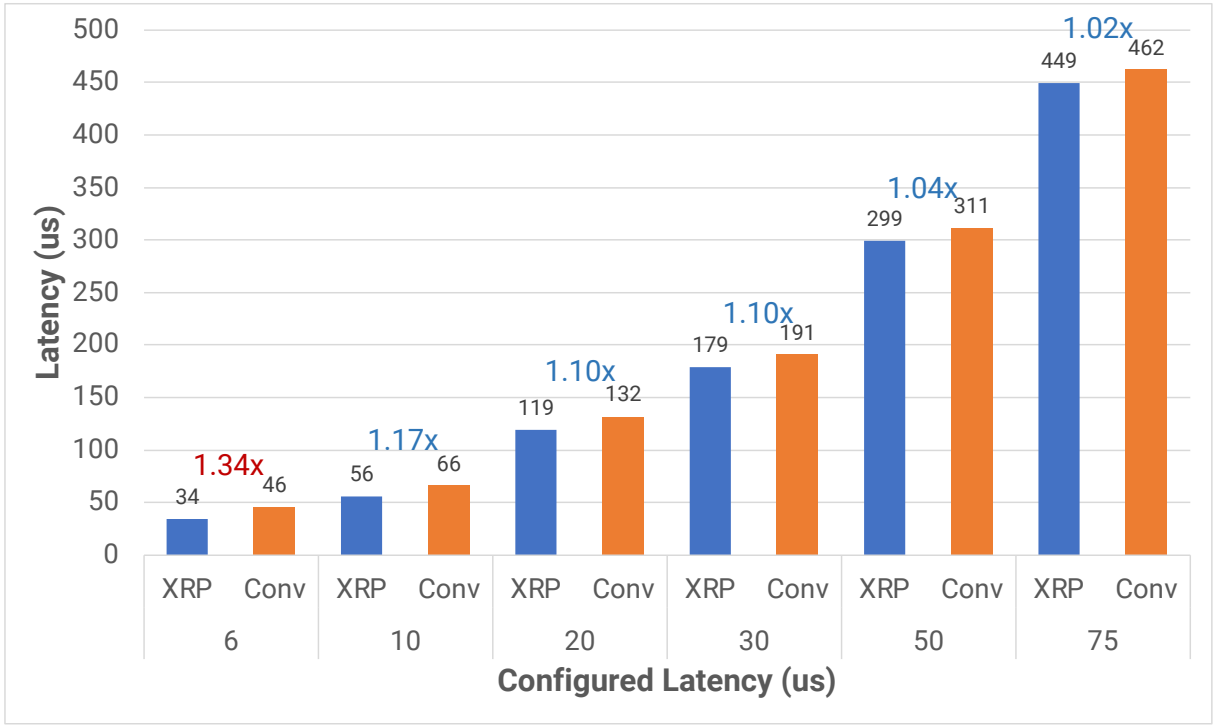
512B random read latency



XRP Evaluation Using NVMeVirt

- Run BPF-KV using various SSD performance configurations

Average Lookup Latency (us)						
# Ops	paper-read()	paper-XRP	read()	XRP	paper	Evaluation
1	13.40	10.70	14.57	12.72	1.25	1.14
2	20.60	14.20	21.96	17.84	1.45	1.23
3	27.40	18.00	29.33	23.14	1.52	1.27
4	34.00	21.70	36.60	28.54	1.57	1.28
5	41.50	25.40	46.00	34.00	1.63	1.35



Installing and Using NVMeVirt

Kernel Setup

- Recommended Linux kernel version: v5.15.x and higher

- Reserve a part of the main memory for NVMeVirt

```
GRUB_CMDLINE_LINUX="memmap=32G\\\$16G"
```

- Update grub and reboot your system

```
$ sudo update-grub
$ sudo reboot
```

Compiling NVMeVirt

- Select the target device type by manually editing the Kbuild

```
# Select one of the targets to build
CONFIG_NVMEVIRT_NVM := y
#CONFIG_NVMEVIRT_SSD := y
#CONFIG_NVMEVIRT_ZNS := y
#CONFIG_NVMEVIRT_KV := y

obj-m   := nvmev.o
nvmev-objs := main.o pci.o admin.o io.o dma.o
ccflags-y += -Wno-unused-variable -Wno-unused-function

ccflags-$(CONFIG_NVMEVIRT_NVM) += -DBASE_SSD=INTEL_OPTANE
nvmev-$(CONFIG_NVMEVIRT_NVM) += simple_ftl.o

ccflags-$(CONFIG_NVMEVIRT_SSD) += -DBASE_SSD=SAMSUNG_970PRO
nvmev-$(CONFIG_NVMEVIRT_SSD) += ssd.o conv_ftl.o pqueue/pqueue.o channel_model.o

ccflags-$(CONFIG_NVMEVIRT_ZNS) += -DBASE_SSD=WD_ZN540
#ccflags-$(CONFIG_NVMEVIRT_ZNS) += -DBASE_SSD=ZNS_PROTOTYPE
ccflags-$(CONFIG_NVMEVIRT_ZNS) += -Wno-implicit-fallthrough
nvmev-$(CONFIG_NVMEVIRT_ZNS) += ssd.o zns_ftl.o zns_read_write.o zns_mgmt_send.o zns_mgmt_recv.o channel_model.o

ccflags-$(CONFIG_NVMEVIRT_KV) += -DBASE_SSD=KV_PROTOTYPE
nvmev-$(CONFIG_NVMEVIRT_KV) += kv_ftl.o append_only.o bitmap.o
```

Using NVMeVirt

- Attach the emulated NVMe SSD by loading the NVMeVirt kernel module

```
$ sudo insmod ./nvmev.ko \
  memmap_start=16G \      # e.g., 1M, 4G, 8T
  memmap_size=32G   \     # e.g., 1M, 4G, 8T
  cpus=7,8                # List of CPU cores to process I/O requests
                            (must have at least 2)
```

Dispatcher core, I/O core 1, I/O core 2, …

```
[ 6519.269608] NVMeVirt: Version 1.10 for >> NVM SSD <<
[ 6519.269611] NVMeVirt: Storage: 0x400100000-0xc00000000 (32767 MiB)
[ 6519.278201] NVMeVirt: ns 0/1: size 32767 MiB
[ 6519.278231] PCI host bridge to bus 0001:10
[ 6519.278233] pci_bus 0001:10: root bus resource [io  0x0000-0xffff]
[ 6519.278234] pci_bus 0001:10: root bus resource [mem 0x00000000-0x3fffffffffff]
[ 6519.278235] pci_bus 0001:10: root bus resource [bus 00-ff]
[ 6519.278240] pci 0001:10:00.0: [0c51:0110] type 00 class 0x010802
[ 6519.278242] pci 0001:10:00.0: reg 0x10: [mem 0x400000000-0x400003fff 64bit]
[ 6519.278244] pci 0001:10:00.0: enabling Extended Tags
[ 6519.278335] NVMeVirt: Virtual PCI bus created (node 0)
[ 6519.278525] NVMeVirt: nvmev_io_worker_0 started on cpu 8 (node 0)
[ 6519.278686] NVMeVirt: nvmev_io_worker_1 started on cpu 9 (node 0)
[ 6519.278857] NVMeVirt: nvmev_io_worker_2 started on cpu 10 (node 0)
[ 6519.278869] NVMeVirt: nvmev_dispatcher started on cpu 7 (node 0)
[ 6519.278936] nvme nvme1: pci function 0001:10:00.0
[ 6519.278959] NVMeVirt: Virtual NVMe device created
[ 6519.281500] nvme nvme1: 24/0/0 default/read/poll queues
```

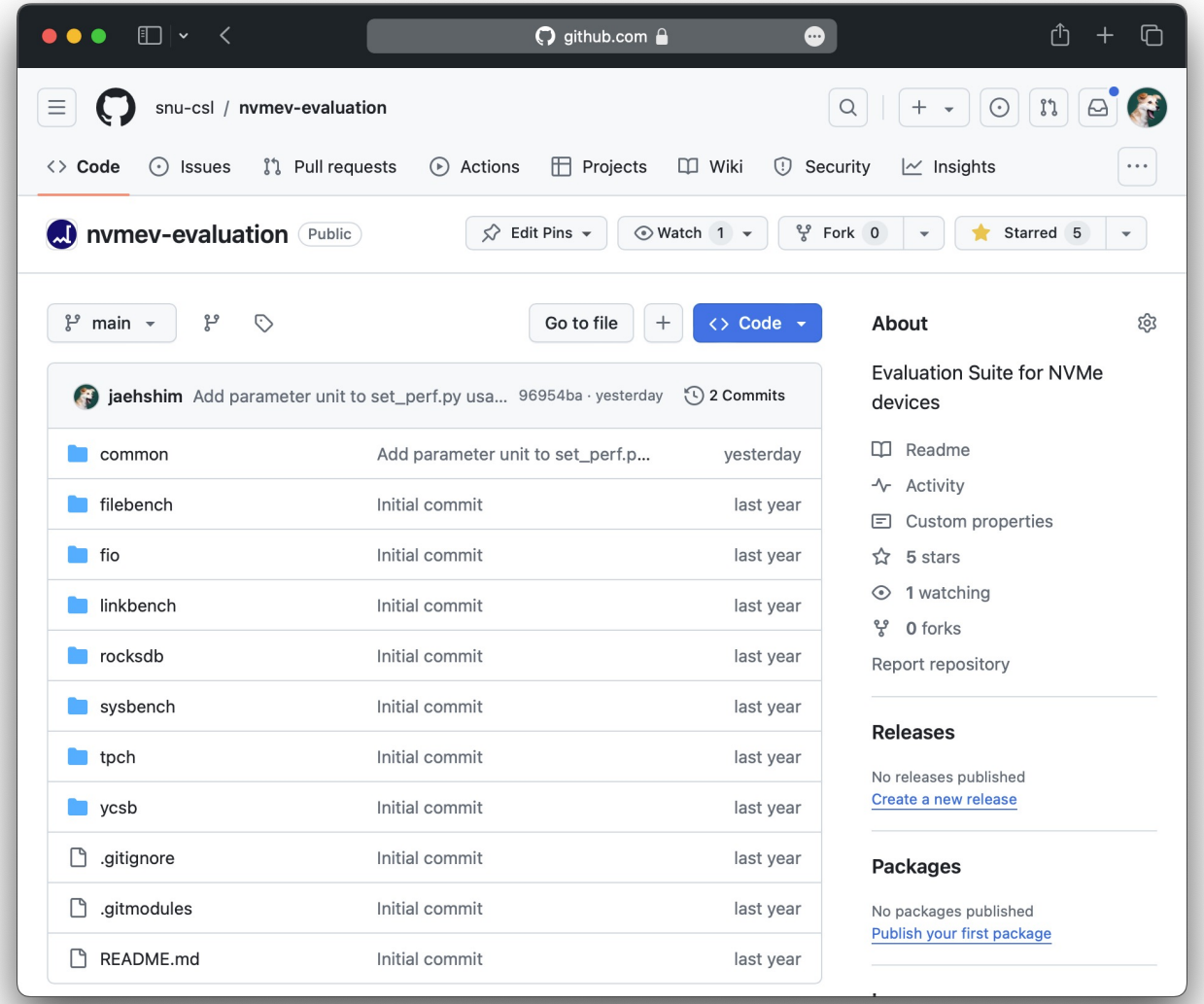
cpus=7,8,9,10

NVMeV-Evaluation

- Evaluation Suite for NVMe devices
  - Filebench, fio, RocksDB, sysbench, TPC-H, etc

- Scripts for loading NVMeVirt module, configuring performance, running benchmarks, and collecting results

- Currently configuring performance is only possible for simple model



Evaluation Script

```
function eval_nvmev() {
    if [ $1 == "max" ]; then
        $COMMON/set_perf.py max
    elif [ $# -lt 3 ]; then
        echo "Usage $0 [read latency in us] [write latency in us] [io_units] ...."
        exit -1
    else
        $COMMON/set_perf.py $1 $2 $3
    fi

    echo "#############################################################"
    echo "###                NVMeVirt Evaluator                     ###"
    echo "#############################################################"
    echo "# read latency  : ${1} us"
    echo "# write latency : ${2} us"
    echo "# bandwidth     : ${3} MB/s"
    echo
    echo -n "Start measuring at "
    date

    taskset -p -c $CPU_AFFINITY $$ &> /dev/null
    sleep 1

    do_measure $*

    echo -n "Finish measuring at "
    date
    echo "# read latency  : ${1} us"
    echo "# write latency : ${2} us"
    echo "# io units      : ${3}"
    echo "#############################################################"
    echo; echo; echo; echo
}
```

common/eval_nvmev.sh

```
function do_measure() {
    # Start measuring
    $COMMON/abort_eval.sh

    sudo sh -c "echo 0 > /proc/nvmev/stat"

    $COMMON/stat_cpu_utilization.sh &> /dev/null &
    $COMMON/stat_io_utilization.sh $DEV &> /dev/null &
    $COMMON/stat_nvmev.sh &> /dev/null &

    # Run actual test
    do_test $* 2>&1 | tee result

    # Finish collecting results
    for p in `jobs -p`; do
        for c in `pgrep -P $p`; do # Kill children
            sudo sh -c "kill -9 $c"
        done
        sudo sh -c "kill -9 $p"
    done

    # Collect results
    mkdir -p results &> /dev/null
    do_collect $*
}
```

common/eval_nvmev.sh