STORAGE DEVELOPER CONFERENCE

SDC 21

BY Developers FOR Developers

Virtual Conference
September 28-29, 2021

Asynchronous I/O passthru in NVMe-Native Applications

Presented by

Kanchan Joshi
Samsung Semiconductor India Research (SSIR)

Simon Lund
Samsung Semiconductor Denmark Research (SSDR)

Javier González
Samsung Semiconductor Denmark Research (SSDR)

Foreword & Acknowledgement

- This has elements that are under discussion in LKML
  - And few have not been discussed yet
  - Mechanism, Opcode, API etc. may change in future

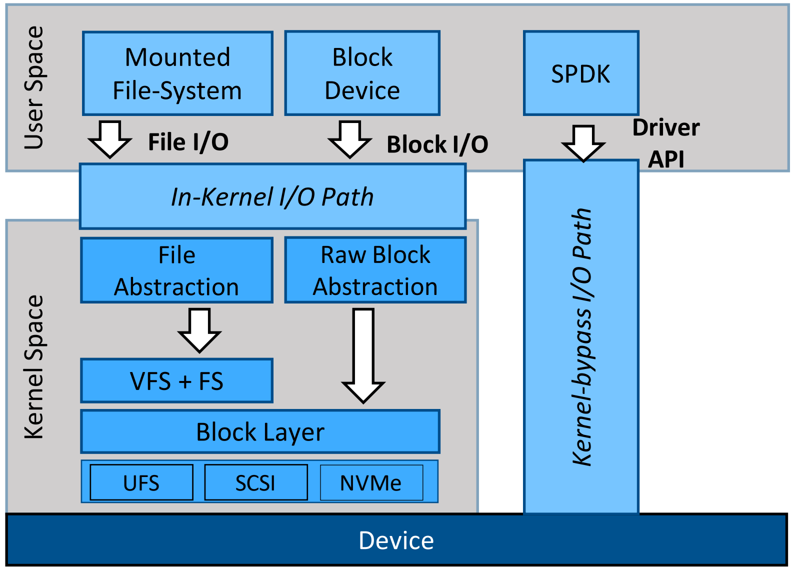
- The work presented here is a community effort
  - Feedback, ideas and code have come from many contributors!
  - Jens Axboe, Christoph Hellwig, Keith Busch to name a few

Agenda

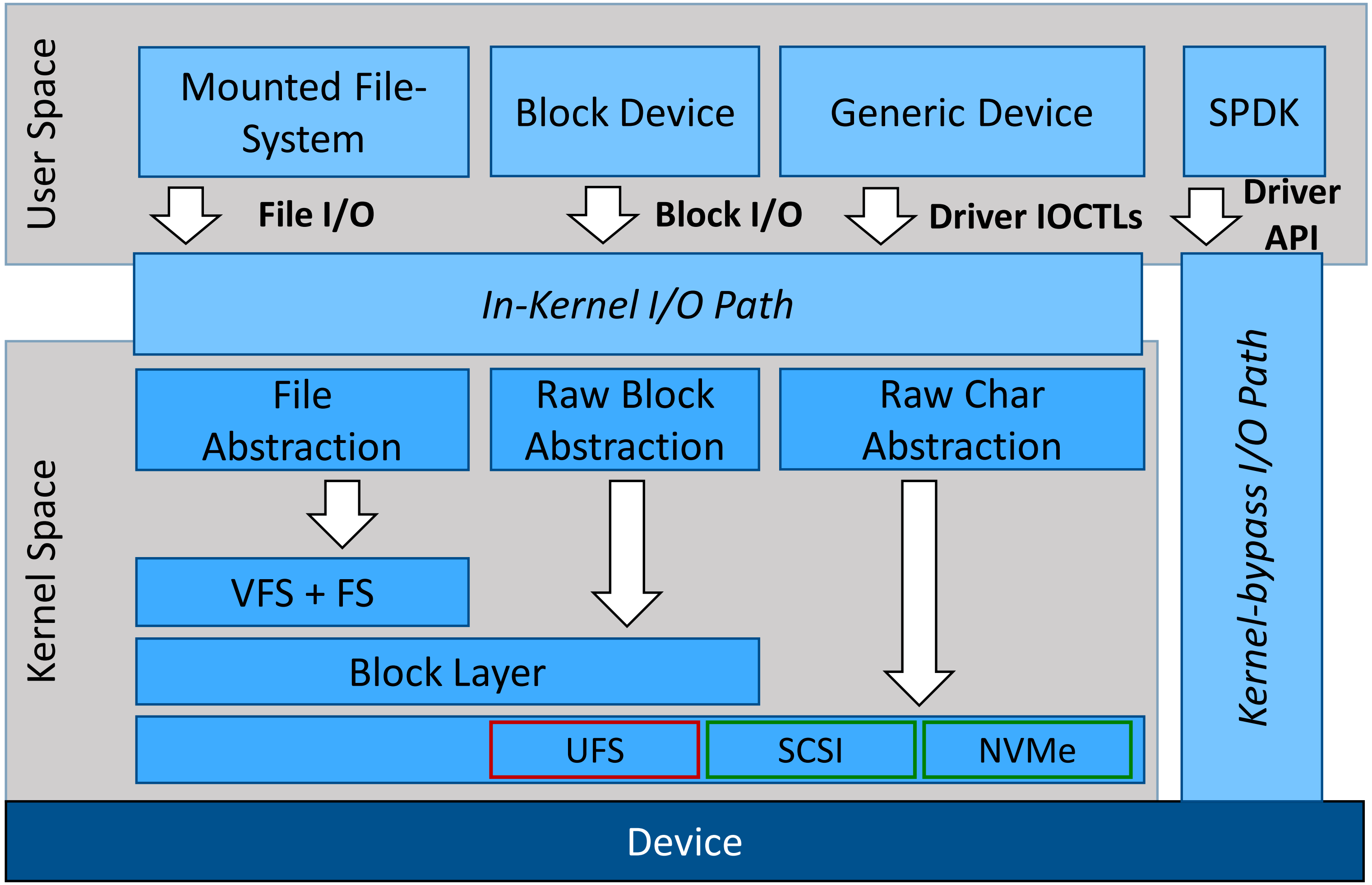
1. NVMe Generic Device in the Linux Kernel
   - Enable an in-kernel passthru I/O Path
   - Support all NVMe device features
2. Async IOCTLs in the Linux Kernel
   - Provide a performant and scalable I/O path for driver passthru
   - Generic layer in io_uring. Specific support for NVMe
3. Application enablement through xNVMe
   - Provide a storage API with cross I/O Path and cross OS support
   - Characterization with real-world numbers

Raw Block in Linux

- Lowest API for block I/O in Linux
  - Control over LBA address space
  - Control over raw I/O properties (e.g., async/sync, direct/cached, queue depth)
  - Block device (namespace) granularity
- A common block abstraction comes with (natural) limitations
  - Unsupported data protection schemes (PI DIF/DIX)
  - Constrains on new device types (e.g., NVMe ZNS)
- Rise of SPDK
  - Enable domain-specific I/O paths and block devices
  - Pave the way for a low-latency storage stack
  - Support fast innovation in end-to-end deployments
  - Becoming generic comes with redundancy



NVMe Generic Device



- Generic Device
  - Always available
  - In-kernel passthru
  - Kernel security (e.g., cgroups)
  - Char device per namespace
  - Upstream in NVMe (5.13)
    - IOCTL I/O
    - Tool support ongoing

Consuming the NVMe Generic Device

- Enumeration
  - Nvme-cli can list [1]
  - Nvme-cli can issue I/O (already upstream)

```
nvme-cli $./nvme list
Node                  SN                   Model
--------------------- -------------------- ----------------
/dev/ng0n1            deadbeef             QEMU NVMe Ctrl
/dev/ng0n2            deadbeef             QEMU NVMe Ctrl
/dev/ng0n3            deadbeef             QEMU NVMe Ctrl
/dev/nvme0n1          deadbeef             QEMU NVMe Ctrl
/dev/nvme0n2          deadbeef             QEMU NVMe Ctrl
/dev/nvme0n3          deadbeef             QEMU NVMe Ctrl
```

- How application can use
  - Send any nvme command via passthru interface
  - Current transport - via NVMe Driver IOCTL
  - Future transport - via io_uring

```
static const struct file_operations nvme_ns_chr_fops = {
        .owner          = THIS_MODULE,
        .open           = nvme_ns_chr_open,
        .release        = nvme_ns_chr_release,
        .unlocked_ioctl = nvme_ns_chr_ioctl,
        .compat_ioctl   = compat_ptr_ioctl,
};
```

- How to enable over fabrics (NVMe-oF)
  - Automatic, when block-interface (/dev/nvme0n1) is up
  - When it is not, enable passthru controller (CONFIG_NVME_TARGET_PASSTHRU)'

```
# Set device nvme0 as the controller we want to expose over the fabric
echo -n /dev/nvme0 > /sys/kernel/config/nvmet/subsystems/testnqn/passthru/device_path
echo 1 > /sys/kernel/config/nvmet/subsystems/testnqn/passthru/enable
```

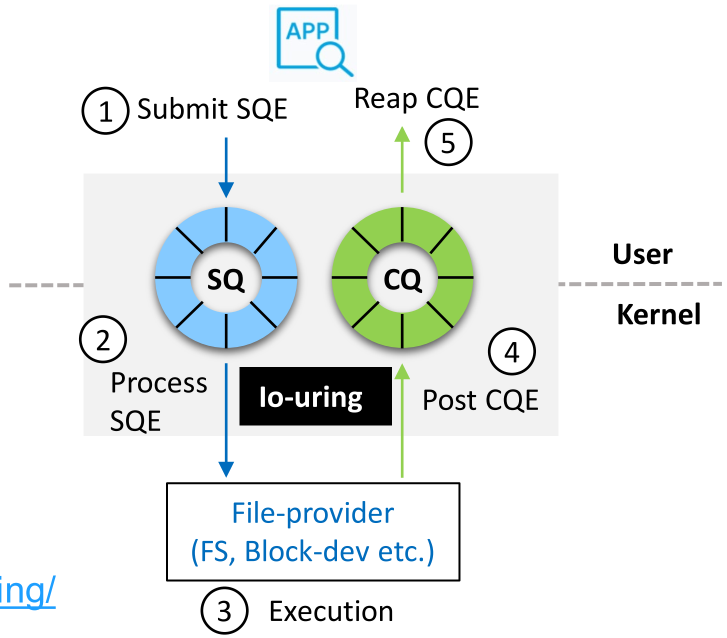
[1] https://github.com/joshkan/nvme-cli/tree/standalone_list_ng

Async IOCTLs

….the io_uring way

What is io_uring (in a nutshell)

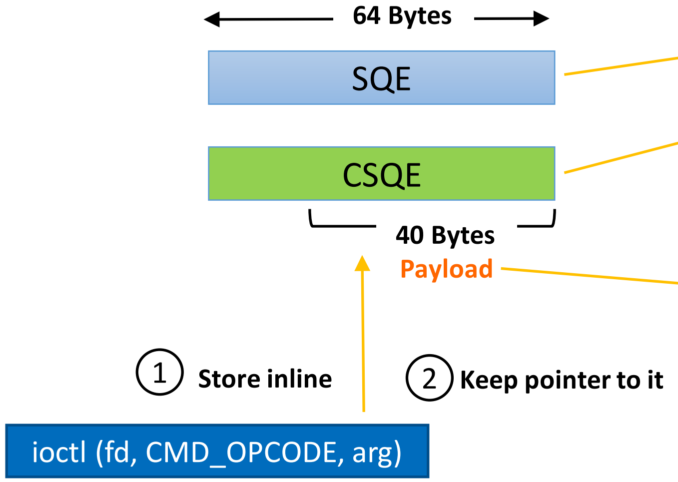
- Scalable asynchronous IO infrastructure
  - File IO as well as Network IO
  - Async without needing O_DIRECT (unlike “linux aio”)
  - Extensible - rapidly adding async variants of sync syscalls
    - mkdir, link, symlink: few recent ones
- User-Kernel communication scheme
  - App/Kernel communicate over shared ring-buffers (SQ and CQ)
    - Reduce syscalls & copies
    - Prepare IO: Get SQE from SQ ring, and fill it up (fill more to make a batch)
    - Submit IO: By calling *io_uring_enter*
    - Complete IO: Reap CQE from CQ ring
  - Submission can be offloaded (no syscall)
  - Completion can be polled (interrupt-free IO)
- Faster IO through io_uring https://kernel-recipes.org/en/2019/talks/faster-io-through-io_uring/



STORAGE DEVELOPER CONFERENCE SDC 21

Asynchronous IOCTL: user-interface

- ‘uring cmd’: IOCTL-like async facility
  - New opcode IORING_OP_URING_CMD
  - A new ‘command’ SQE (CSQE) to be used
    - CSQE = Specialized SQE with 40 bytes of free-space
    - Useful for avoiding allocation (for IOCTL cmd) cost
    - Can be used in other way too (e.g. pointer to larger IOCTL cmd)
  - Submit CSQE and reap completion, as usual




```
struct uring_cmd_ioc {
        __u32   ioctl_cmd;
        __u32   unused1;
        __u64   unused2[4];
};

static int get_bs(struct io_uring *ring, const char *dev)
{
        struct io_uring_cqe *cqe;
        struct io_uring_sqe *sqe;
        struct io_uring_cmd_sqe *csqe;
        struct uring_cmd_ioc *ucmd;
        int ret, fd;

        fd = open(dev, O_RDONLY);

        sqe = io_uring_get_sqe(ring);
        csqe = (void *) sqe;
        memset(csqe, 0, sizeof(*csqe));
        csqe->hdr.opcode = IORING_OP_URING_CMD;
        csqe->hdr.fd = fd;
        csqe->user_data = 0x1234;
        csqe->op = BLOCK_URING_OP_IOCTL;
        ucmd = (void *) &csqe->pdu;
        ucmd->ioctl_cmd = BLKBSZGET;

        io_uring_submit(ring);
        io_uring_wait_cqe(ring, &cqe);
        printf("bs=%d\n", cqe->res);
        io_uring_cqe_seen(ring, cqe);
        return 0;
}
```

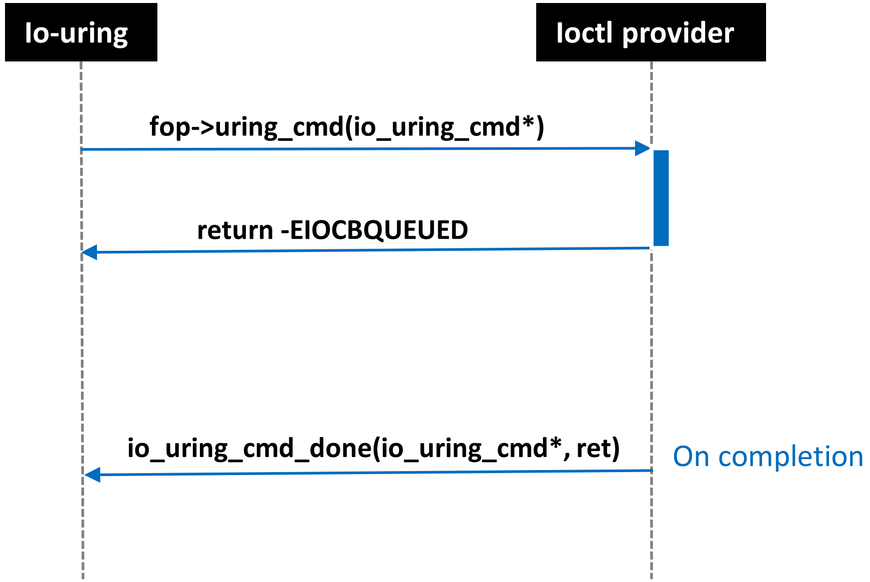

Asynchronous IOCTL: under the hood

- io_uring prepares 'struct io_uring_cmd'

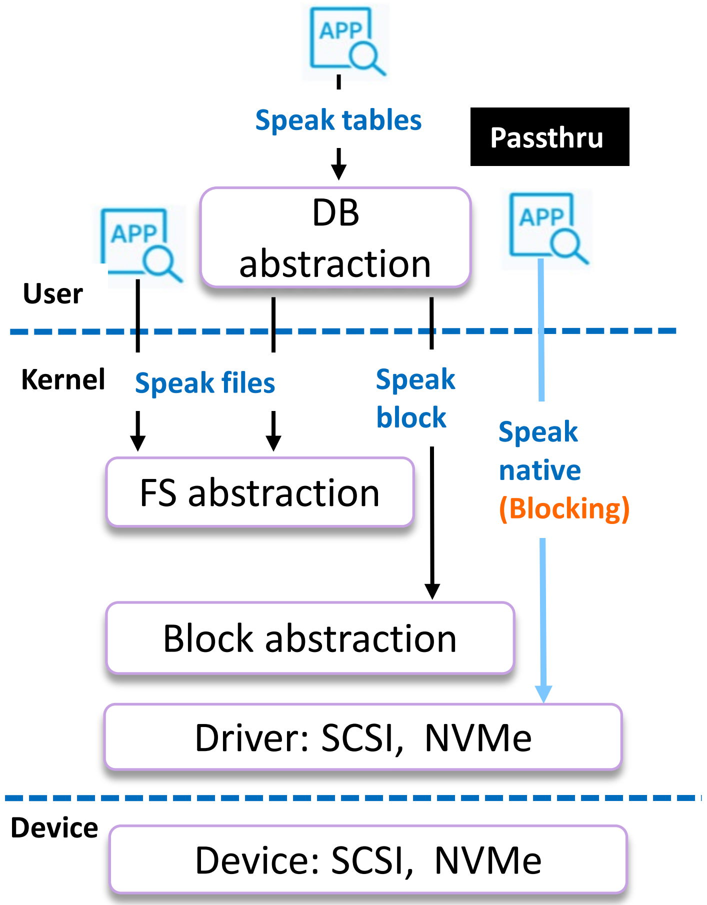
```
+struct io_uring_cmd {
+        struct file     *file;
+        __u16           op;
+        __u16           unused;
+        __u32           len;
+        __u64           pdu[5];
+};
```

```
struct file_operations {
        struct module *owner;
@@ -2059,6 +2068,8 @@ struct file_operations {
                        struct file *file_out, loff_t pos_out,
                        loff_t len, unsigned int remap_flags);
        int (*fadvise)(struct file *, loff_t, loff_t, int);
+
+       int (*uring_cmd)(struct io_uring_cmd *, enum io_uring_cmd_flags);
 } __randomize_layout;
```

- Provider (FS, driver etc.) need to implement async behavior
  - Implement new method uring_cmd in *struct file_operations* (fop, in short)
  - Io_uring submits IOCTL by calling uring_cmd method
  - Provider does what it should (for submission), and returns *without blocking*
  - Provider can return the result immediately
  - Or returns in future, by calling *io_uring_cmd_done*()
  - Io_uring puts result into CQE and posts it to the CQ ring
- Jens v4 series: https://lore.kernel.org/linux-nvme/20210317221027.366780-1-axboe@kernel.dk/



NVMe passthru interface



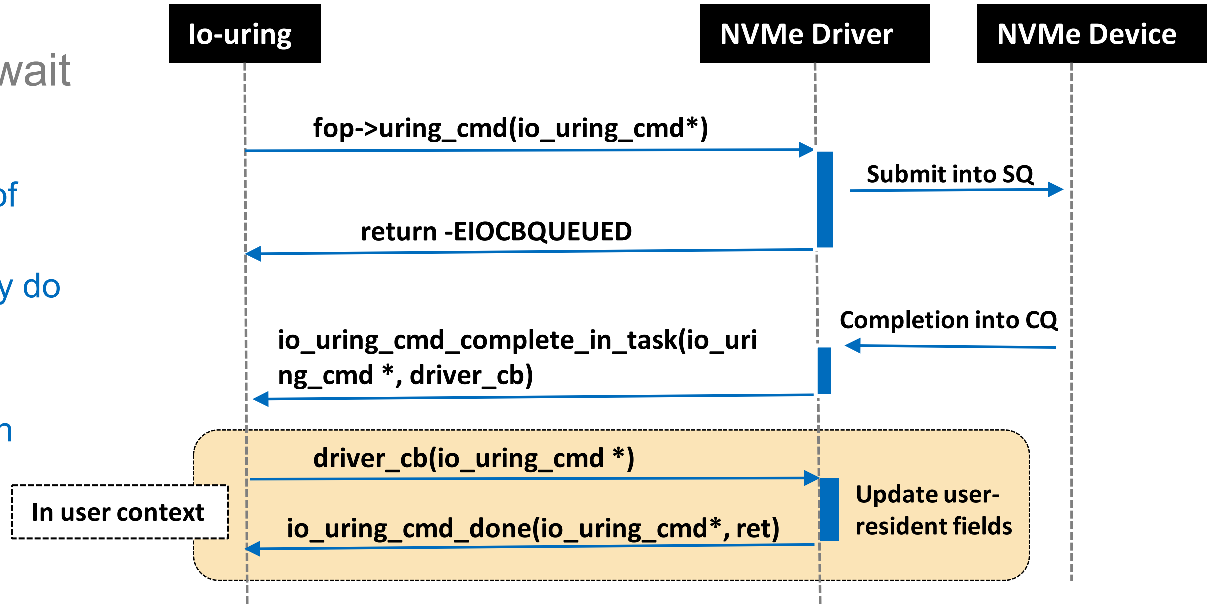
- NVMe passthru interface – as of today
  - Good part
    - In-kernel path that cuts through layers of abstraction
    - Enables new device-features to be consumed (in native form) readily
      - Block/file generic in-kernel interfaces/users, and user-space interfaces may take time to evolve
  - Bad part
    - Transport (from user to kernel) is only via synchronous ioctl()
    - That renders it virtually useless for fast I/O path
- NVMe passthru interface – of future (hopefully)
  - Scalable enough to leverage performance-aspect of NVMe features (beyond read/write)
  - Move along performance advancements of io_uring
  - TL;DR: much more useful passthru interface!

NVMe passthru: async transport

- NVMe ioctl() operation
  - Sync-over-Async
    - Device interface is 'naturally' async
      - Host submit commands into NVMe SQ, at time T
      - Device posts completion into NVMe CQ, at time T+ Δ T
    - Driver puts the submitter go into blocking-wait until completion arrives

```
static const struct file_operations nvme_ns_chr_fops = {
        .owner          = THIS_MODULE,
        .open           = nvme_ns_chr_open,
        .release        = nvme_ns_chr_release,
        .unlocked_ioctl = nvme_ns_chr_ioctl,
        .compat_ioctl   = compat_ptr_ioctl,
        .uring_cmd      = nvme_ns_chr_async_ioctl,
};
```

- nvme uring_cmd() operation
  - Decouples completion from submission; no blocking-wait
  - The 'async-update-to-user-memory' problem
    - user-resident fields (in ioctl cmd) may need to updated as part of completion
    - But completion, when arriving in interrupt-context, can not safely do that!
    - Thankfully Kernel has task-work infra
    - Driver, while in interrupt context, schedules update to be done in submitter's context



STORAGE DEVELOPER CONFERENCE SDC21

Async NVMe passthru

- Example: read from /dev/ng0n1

Allocate and setup nvme passthru command

Prepare CSQE for uring-cmd

Setup passthrough ioctl & cmd pointer inside uring-cmd

```
/* this overlays struct io_uring_cmd pdu (40 bytes) */
struct nvme_uring_cmd {
        __u32   ioctl_cmd;
        __u32   unused1;
        void    *argp;
};
/* issue passthru command to read from device into buf */
void nvme_passthru_read(struct io_uring *ring, void *buf)
{
        struct io_uring_sqe *sqe = NULL;
        struct io_uring_cqe *cqe = NULL;
        struct io_uring_cmd_sqe *csqe;
        struct nvme_passthru_cmd *ptcmd;
        struct nvme_uring_cmd *ncmd;
        int fd;

        fd = open("/dev/ng0n1", O_RDONLY);

        ptcmd = (struct nvme_passthru_cmd *)malloc(sizeof(struct nvme_passthru_cmd));
        prepare_pt_cmd(ptcmd, buf);

        sqe = io_uring_get_sqe(ring);
        csqe = (void *)sqe;
        csqe->hdr.fd = fd;
        csqe->hdr.opcode = IORING_OP_URING_CMD;
        csqe->user_data = 0x1234;

        ncmd = (void *) &csqe->pdu;
        ncmd->ioctl_cmd = NVME_IOCTL_IO64_CMD;
        ncmd->argp = (void *)ptcmd;

        io_uring_submit(ring);
        io_uring_wait_cqe(ring, &cqe);

        printf("res=%d\n", cqe->res);
        io_uring_cqe_seen(ring, cqe);
        free(ptcmd);
}
```

- Tidbits for ZNS users
  - Async zone-reset; Currently possible only via zone-mgmt ioctl
  - Zone-append at higher-qd

Is Async enough

…can we take this further?

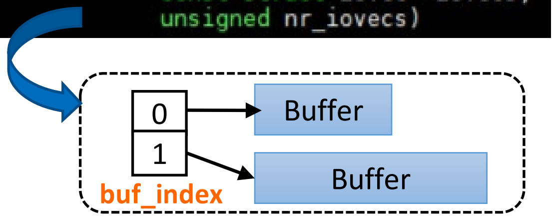
Features for faster I/O

Feature	What it does	Io_uring	Uring-passthru
Register-files	Reference fd once and reuse	☑	☑
SQPoll	Offload IO submission	☑	☑
Fixed-buffer	Map IO buffer once and reuse	☑	☒
Async polling	Interrupt-free completion	☑	☒

Uring passthru: fixed-buffer

- What & how it helps
  - Fixed-buffer or pre-mapped buffer
    - User-buffer need to be pinned before IO, and unpinned on completion
    - Reduce the pin/unpin cost: pin once and reuse the same buffer
    - io_uring allows application to
      - Pin N buffers upfront (using *io_uring_register*)
      - Specify IO (fixed-buffer IO) by using any of the pre-mapped buffer

```
int io_uring_register_buffers(struct io_uring *ring,
                              const struct iovec *iovecs,
                              unsigned nr_iovecs)
```



- Passthru with fixed-buffer
  - io_uring side
    - New opcode IORING_OP_URING_CMD_FIXED
    - Buffer are registered as before, and *sqe->buf_index* to be used for IO
    - Provide infra (to driver) for accessing the registered buffer
  - NVMe side
    - Instead of pin/unpin, talk to io_uring to reuse 'previously pinned' buffer

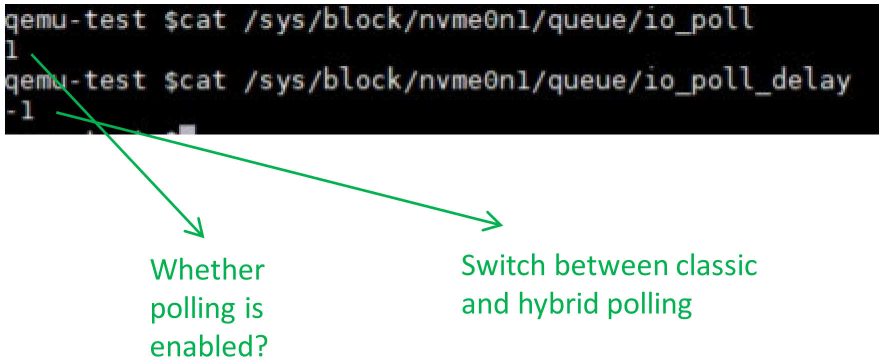
```
sqe = io_uring_get_sqe(ring);
csqe = (void *)sqe;
csqe->hdr.fd = fd;
csqe->hdr.opcode = IORING_OP_URING_CMD_FIXED;
csqe->buf_index = buf_index;
csqe->user_data = 0x1234;

ncmd = (void *) &csqe->pdu;
ncmd->ioctl_cmd = NVME_IOCTL_IO64_CMD;
ncmd->argp = (void *)ptcmd;
```

```
int io_uring_cmd_import_fixed(void *ubuf, unsigned long len,
                              int rw, struct iov_iter *iter, void *ioucmd)
```


I/O polling: Sync vs Async

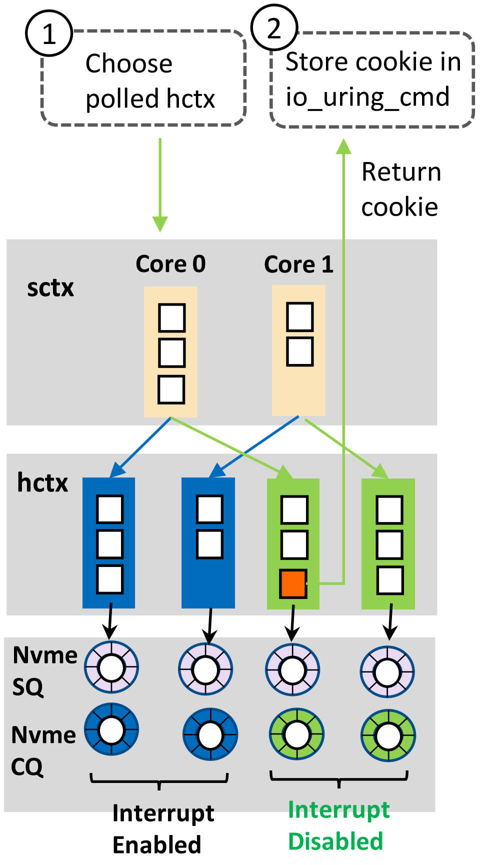
- Kernel I/O Polling
  - Allows interrupt-free IO; particularly useful for ultra-low-latency devices
  - Submitter actively checks for completion (busy-waiting)
  - Sync Polling
    - Application goes about spinning for completion just after submission
    - Hybrid polling: sleep for some time (relax the cpu) while looking for completion
    - Syscall: preadv2(), pwritev2() with RWF_HIPRI flag
  - Async Polling
    - What choices do we have after submitting an IO – 1. spin 2. sleep+spin 3. do_more_work
    - Async polling enables the third option i.e. submit more IO, or do other app-specific processing
    - Polling is decoupled from submission; Hybrid polling can still be configured into this model
    - Syscall: io_uring needs to be setup with IORING_SETUP_IOPOLL. All I/Os to such ring are polled



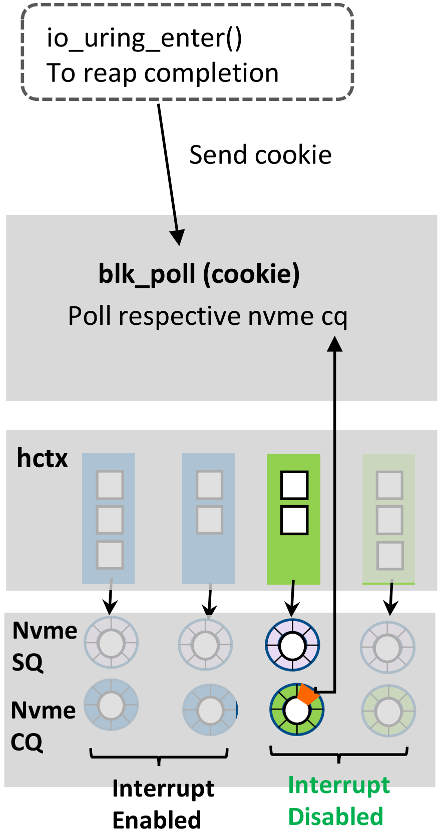
STORAGE DEVELOPER CONFERENCE SDC21

Uring passthru: Async Polling

- Submission



- Completion (polled)

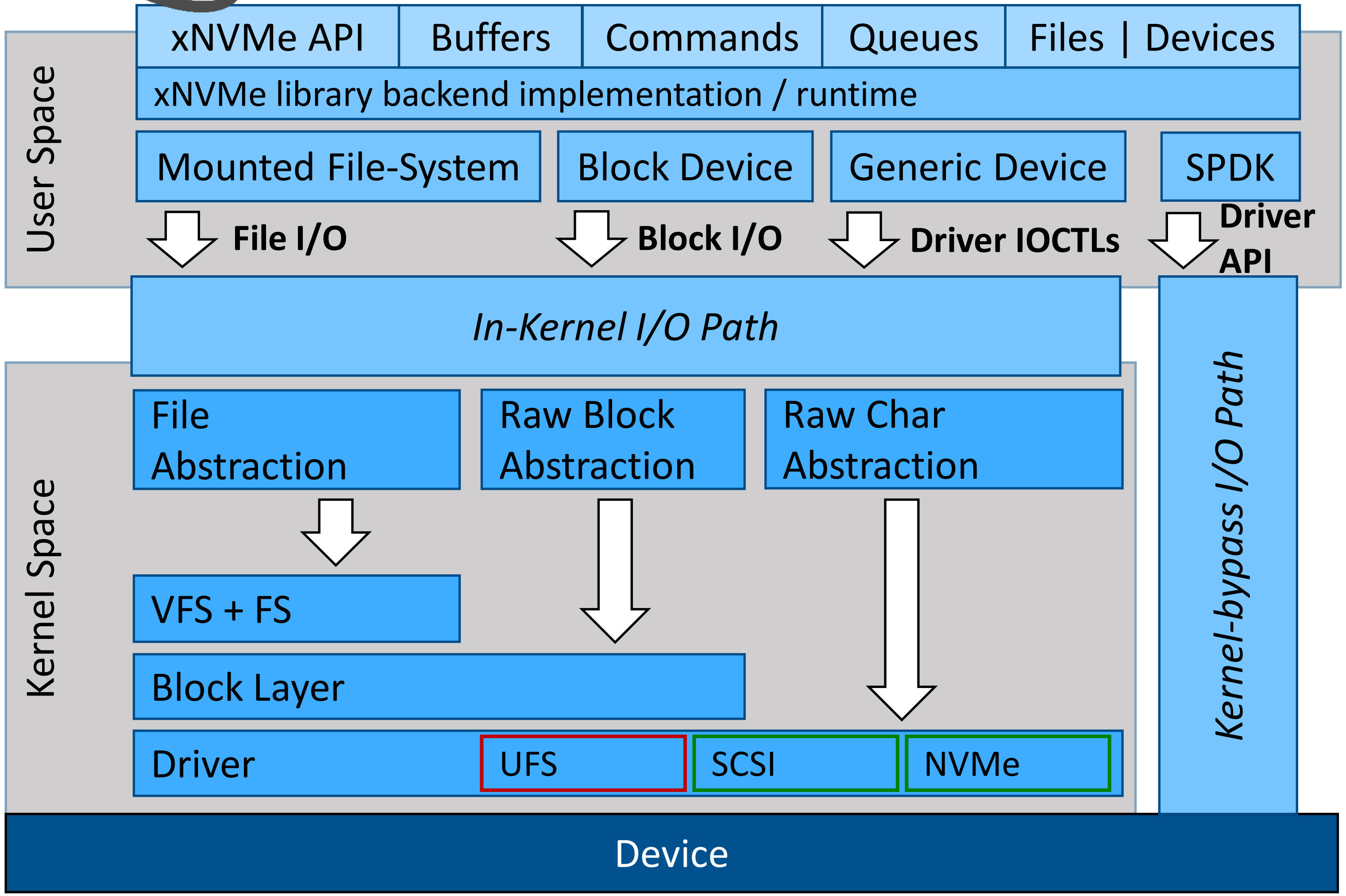


 |

Features for faster I/O

Feature	What it does	Io_uring	Uring-passthru
Register-files	Reference fd once and reuse	☑	☑
SQPoll	Offload IO submission	☑	☑
Fixed-buffer	Map IO buffer once and reuse	☑	☑
Async polling	Interrupt-free completion	☑	☑
Bio-recycling*	In-kernel cache to reduce per-io alloc & free	☑	☒

Using the Char Device



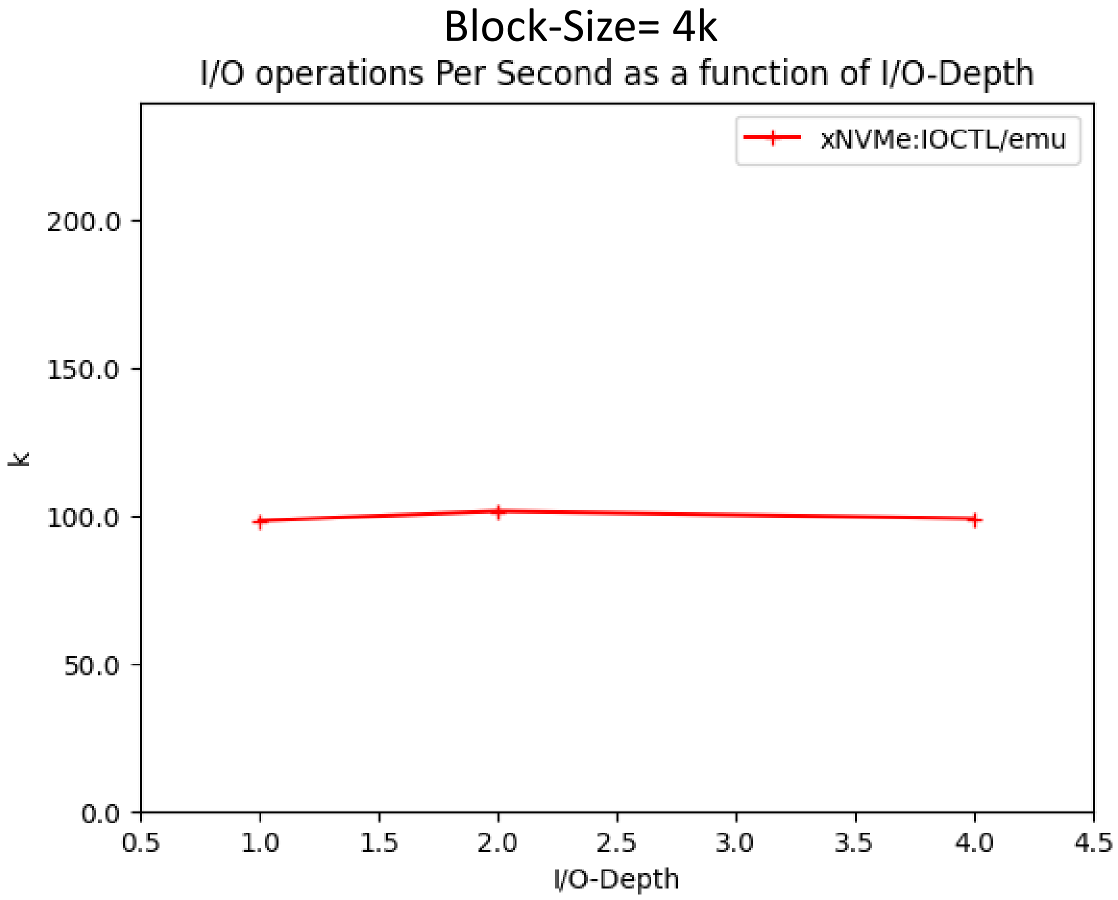
- IO / Command Library
  - Change I/O Path without changing a single-line of code
  - Synchronous API, blocking until completion
  - Asynchronous API using queues and callbacks
  - Knobs to tune the underlying implementation / runtime

xNVMe Tool Demo

```
root@box-tux01:~# xnvme enum
xnvme_enumeration:
  - {uri: '0000:01:00.0', dtype: 0x2, nsid: 0x1, csi: 0x0}
  - {uri: '/dev/nvme1n1', dtype: 0x2, nsid: 0x1, csi: 0x0}
  - {uri: '/dev/ng1n1', dtype: 0x2, nsid: 0x1, csi: 0x0}
root@box-tux01:~#
```

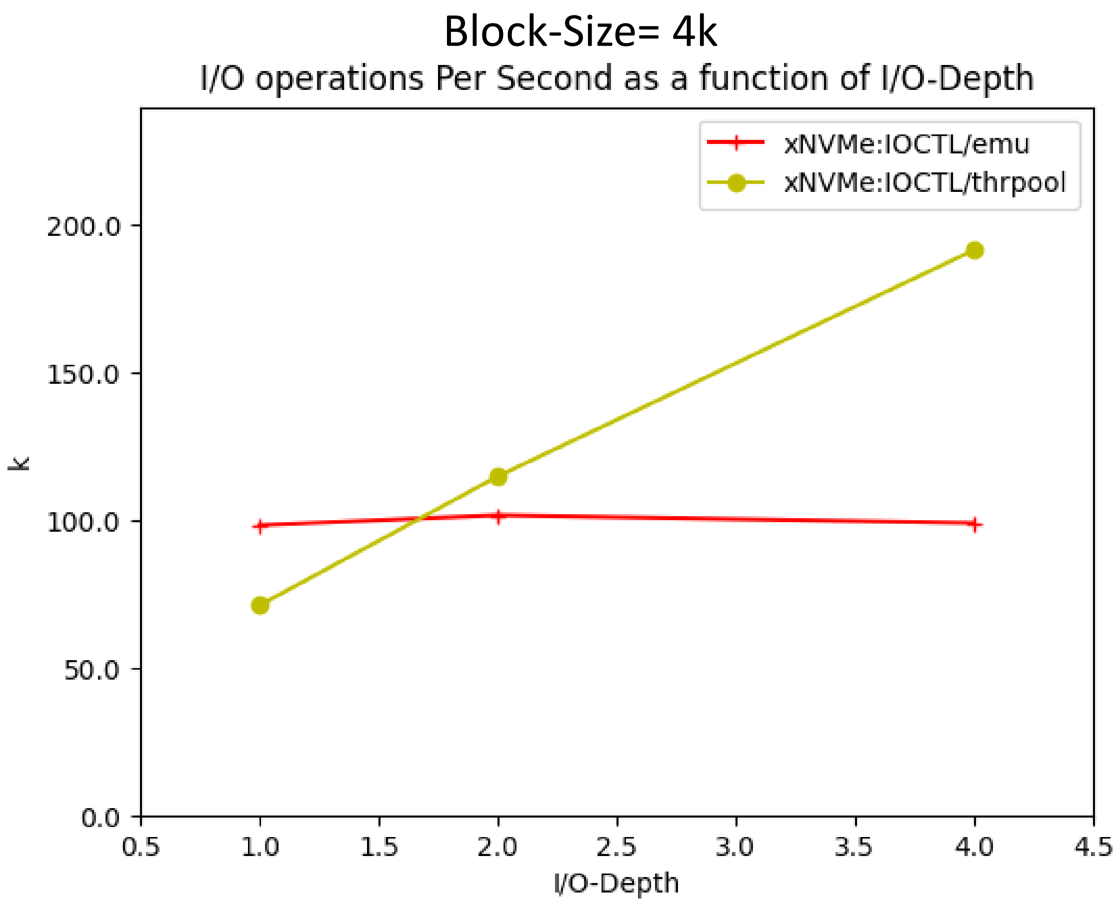
- Device enumeration
  - `xnvme enum`
- Device inspection
  - `xnvme idfy-ns 0000:01:00.0 --dev-nsid 0x1`
  - `xnvme idfy-ns /dev/ng0n1`
  - `xnvme idfy-ns /dev/nvme0n1`
- fio invocation
  - `fio … --filename=/dev/nvme0n1 --xnvme_async=io_uring`
  - `fio … --filename=/dev/ng0n1   --xnvme_async=io_uring_cmd`
  - `fio … --filename=0000:01:00.0 -xnvme_dev_nsid=0x1`

xNVMe Performance & Scalability



- NVMe Passthru (today)
  - Driver IOCTL
    - ➔ scale: none

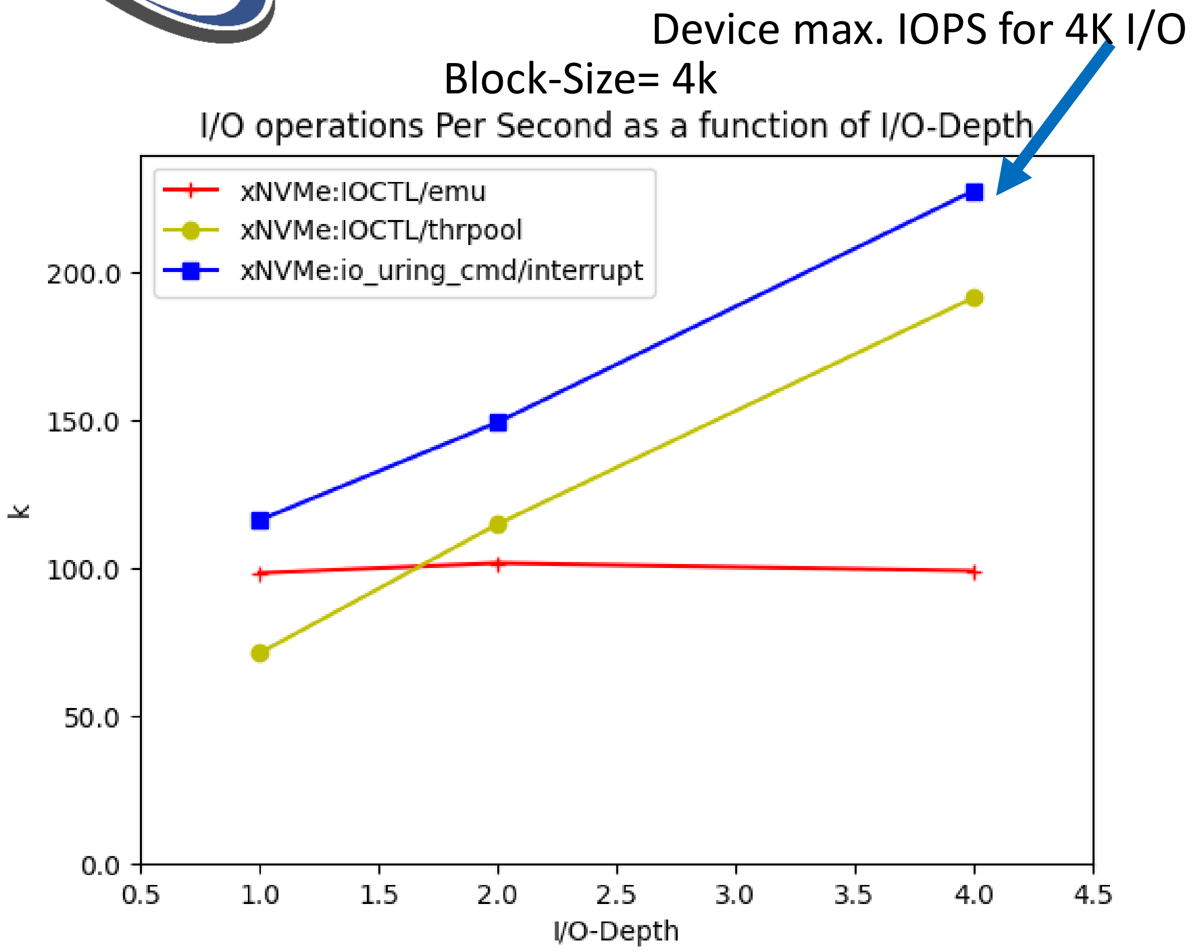
xNVMe Performance & Scalability



- NVMe Passthru (today)
  - Driver IOCTL
    - ➔ scale: none
  - Driver IOCTL + threadpool
    - ➔ scale: but high overhead

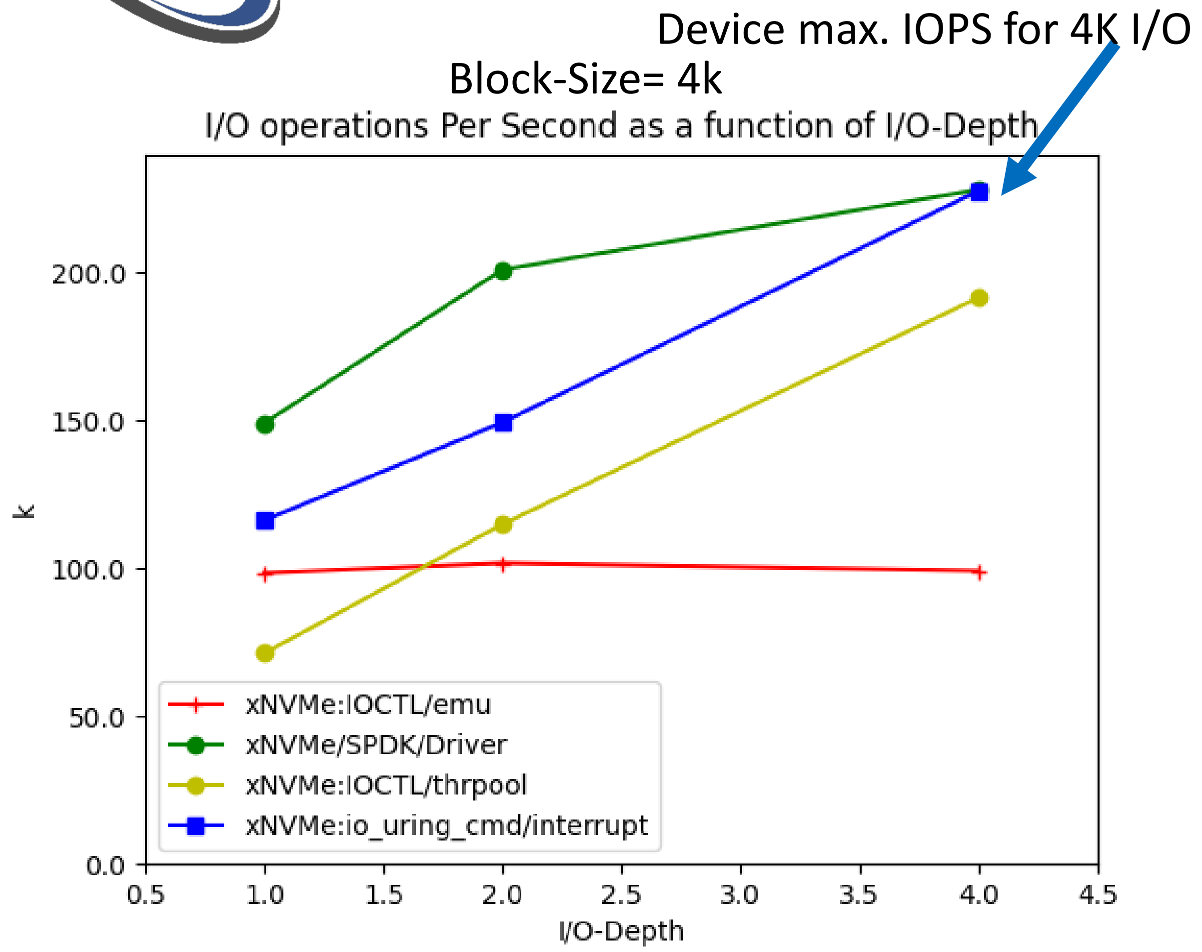
STORAGE DEVELOPER CONFERENCE SDC 21

xNVMe Performance & Scalability



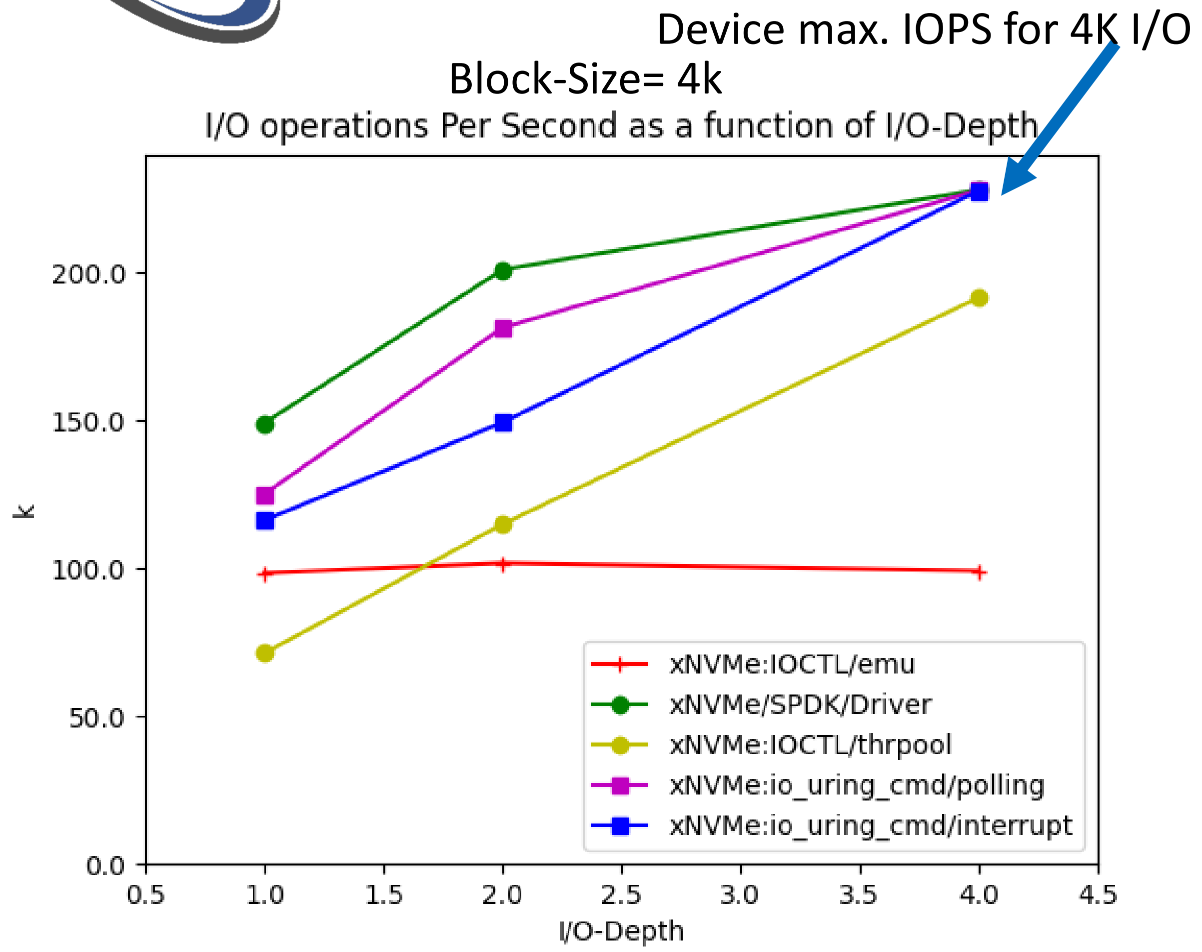
- NVMe Passthru (today)
  - Driver IOCTL
    - ➔ scale: none
  - Driver IOCTL + threadpool
    - ➔ scale: but high overhead
- NVMe Passthru (future)
  - io_uring_cmd() – v5 patchset
    - ➔ scale: efficiently

xNVMe Performance & Scalability



- NVMe Passthru (today)
  - Driver IOCTL
    - ➔ scale: none
  - Driver IOCTL + threadpool
    - ➔ scale: but high overhead
- NVMe Passthru (future)
  - io_uring_cmd() – v5 patchset
    - ➔ scale: efficiently
  - A gap to reach SPDK Driver

xNVMe Performance & Scalability



- NVMe Passthru (today)
  - Driver IOCTL
    - ➔ scale: none
  - Driver IOCTL + threadpool
    - ➔ scale: but high overhead
- NVMe Passthru (future)
  - io_uring_cmd() – v5 patchset
    - ➔ scale: efficiently
  - A gap to reach SPDK Driver
  - io_uring_cmd() – v6 patchset
    - ➔ scale: reduce the gap with polling
    - **Further**: fixedbufs, sqthread_poll, etc.

Upstreaming & Ecosystem

- NVMe Generic Device
  - Available since 5.13
- Async IOCTLs
  - Ongoing upstreaming effort
  - Current working branch
    - Kernel Patches: https://github.com/joshkan/nvme-uring-pt
    - Features: async nvme passthru, fixed-buffer and polling support for passthru
- xNVMe
  - Supported I/O Paths: psync, POSIX aio, libaio, io_uring, NVMe Generic, NVMe Driver IOCTLs, SPDK NVMe Driver, Windows: IO Control Ports and IOCTLs
  - Supported Operating Systems: Linux, FreeBSD, Windows
  - Latest release: https://github.com/OpenMPDK/xNVMe

Talk to us

- Join our Discord Channel
  - Samsung Memory Open-Source
- Email us
  - Kanchan Joshi <joshi.k@samsung.com>
  - Simon Lund <simon.lund@samsung.com>
  - Javier González <javier.gonz@samsung.com>

Please take a moment to rate this session.

Your feedback is important to us.

