I/O Passthru: Upstreaming a flexible and efficient I/O Path in Linux

Kanchan Joshi, Anuj Gupta, Javier González, Ankit Kumar, Krishna Kanth Reddy, Arun George, and Simon Lund, *Samsung Semiconductor;*
Jens Axboe, *Meta Platforms Inc.*

https://www.usenix.org/conference/fast24/presentation/joshi

This paper is included in the Proceedings of the 22nd USENIX Conference on File and Storage Technologies.

February 27–29, 2024 • Santa Clara, CA, USA

978-1-939133-38-0

Open access to the Proceedings of the 22nd USENIX Conference on File and Storage Technologies is sponsored by

I/O Passthru: Upstreaming a flexible and efficient I/O Path in Linux

Kanchan Joshi[1], Anuj Gupta[1], Javier González[1], Ankit Kumar[1], Krishna Kanth Reddy[1], Arun George[1], Simon Lund[1], and Jens Axboe[2]

[1]Samsung Semiconductor
[2]Meta Platforms Inc

Abstract

New storage interfaces continue to emerge fast on Non-Volatile Memory Express (NVMe) storage. Fitting these innovations in the general-purpose I/O stack of operating systems has been challenging and time-consuming. The NVMe standard is no longer limited to block-I/O, but the Linux I/O advances historically centered around the block-I/O path. The lack of scalable OS interfaces risks the adoption of the new storage innovations.

We introduce *I/O Passthru*, a new I/O Path that has made its way into the mainline Linux Kernel. The key ingredients of this new path are *NVMe char interface* and *io_uring command*. In this paper, we present our experience building and upstreaming I/O Passthru and report on how this helps to consume new NVMe innovations without changes to the Linux kernel. We provide experimental results to (i) compare its efficiency against existing *io_uring* block path and (ii) demonstrate its flexibility by integrating data placement into Cachelib. FIO peak performance workloads show 16-40% higher IOPS than block path.

1 Introduction

The Non-Volatile Memory Express (NVMe) protocol has been the unquestionable catalyst for the broad adoption of NAND-based storage devices and Solid-State Drives (SSDs). NVMe continues to bring new capabilities in terms of performance and functionality. Low latency, high bandwidth SSDs (such as Intel optane [61], Kioxia's FL6 [11], Samsung's ZNAND [25]) use NVMe as the protocol of choice. Functionality expansion comes from new commands and command sets that make NVMe viable for unconventional block storage. In the past few years, several non-block storage interfaces have gained popularity and, lately, standardization in NVMe. Specifically, in data-placement solutions, Open-Channel SSDs [37, 44, 47] gained popularity in the academia and industry and eventually opened the door for the standardization of Zoned Namespaces (ZNS) in NVMe. As of today, NVMe standardizes several new interfaces, including Multi-Stream (NVMe Directives) [19], Key-Value [19], Zoned Namespaces [19], and Flexible Data Placement [17]; more interfaces such as Computational Storage [23] are still under development.

It is relevant to note that all of these new interfaces require vertical integration across different storage stack layers (driver, block-layer, file systems) and define new user interfaces to accommodate new device interfaces. Such changes are not always welcomed, as they go against the principle of maintaining a stable and general-purpose operating system. Linux Kernel goes to great lengths to abstract the hardware and never breaks the user-space. This presents a difficult trade-off as robustness and maintenance of the operating system lock horns with early enablement and adoption of NVMe innovations.

In this paper, we present *I/O Passthru*, a novel I/O path in mainline Linux kernel that (i) allows the deployment of any new NVMe feature much faster as it is devoid of extra abstractions and (ii) provides an efficient and feature-rich user-interface. To summarize, our main contributions are the following:

- We build a new NVMe passthrough I/O path which provides higher flexibility and efficiency than the block I/O path (Section 4). We provide examples of how this path enables NVMe interfaces, such as flexible data placement, computational storage, and end-to-end data protection (Section 6).
- We introduce io_uring command, a generic facility to implement asynchronous IOCTLs in the Linux kernel. We detail its API and design (Section 4.2.1).
- We get this path upstream in the Linux Kernel (Section 5) and integrate it into user-space software, including SPDK, xNVMe, liburing, fio and nvme-cli (Section 5.2).
- We elaborate on factors that influence the efficiency of I/O and evaluate the proposed path. FIO peak-performance workloads show 16-40% higher IOPS (Section 7.1).

2 Motivation and Background

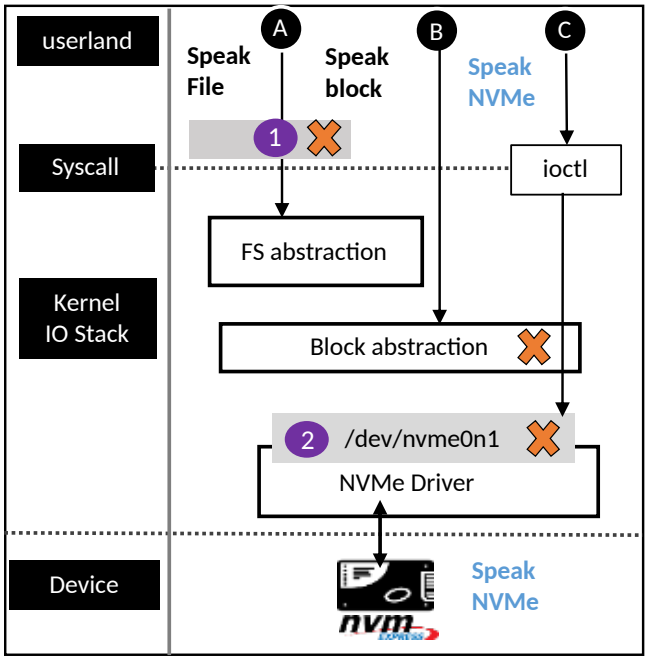


Figure 1: Abstraction layers across different I/O paths and its failures undermining usability, availability and efficiency

2.1 NVMe innovations vs Kernel abstractions

The primary motivation is the fast-paced growth of NVMe innovations and the Linux kernel's agility, or lack thereof, to consume those. **NVMe, initially meant to support only block storage, is no longer tied to it.** This is possible after the introduction of an entity named command-set in NVMe standard [19]. NVMe 2.0 specification defines three command sets:

- NVM command set corresponds to block storage. Nevertheless, it continues to grow newer ways of interacting with storage. For example, (i) data-placement methods like multi-stream and FDP [17] involve passing hints with write, (ii) copy command that does not involve host buffers and performs in-device copy instead.
- Zoned namespace command set exposes zones to the Host and presents new I/O commands such as zone-append and zone-management send/receive.
- Key-value command set does away with fixed-size logical blocks and speaks keys/values instead.

Moreover, new command sets for computational-storage are shaping up. Command sets convey the divorce of NVMe from block-only storage, thereby ensuring faster future innovation.

As for the Linux kernel, generic abstractions are at the foundation. Figure 1 briefly describes various I/O paths in the Linux kernel. NVMe driver collaborates with the block layer, abstracts NVMe protocol, and presents a block device `/dev/nvme0n1` to the upper layer. This block device interface helps the file system to be NVMe agnostic. For example, the file system sends a write operation to the block device by forming a `bio` with `REQ_OP_WRITE`, which is translated to a protocol-specific write command by the underlying driver. The block interface is the bedrock that file systems use to create file abstraction. File systems collaborate with VFS to provide specific implementations of certain user-space APIs that are invoked as system calls. For the syscall users, the file system itself is abstracted. This is shown as path (A) in the figure. Path (B) is a subset when the block device is operated directly without any file system. Figure 1 outlines specific problems with the existing I/O paths:

- Many **new NVMe commands do not fit into the existing user interface.** Adding a new system call requires a more generic use case than the NVMe-specific one. Furthermore, a new syscall is discouraged as it has to be supported indefinitely [1]. Consequently, there is an increase in NVMe interfaces that still need a user interface in Linux. For example, zone-append [35], a variant of nameless writes [27] tailored for zoned storage which is supported by the block layer for in-kernel users, but lacks a user-space API due to the unconventional semantics. Also, while we count on several mentioned technologies to improve in-device data-placement decisions, we still do not have a streamlined way to communicate placement information with the existing write APIs. Finally, despite several efforts to open-source support for copy-offload [43, 54] given the existing hardware support [38, 45], we are yet to see mainline support with a matching user interface.
- One way to alleviate the user-interface scarcity is by using the NVMe passthrough path, shown as (C) in the figure. This path is devoid of file/block abstractions. Applications can send the NVMe command using the `ioctl` syscall [5]. However, this path comes at the **cost of efficiency**, as `ioctl` is a synchronous operation and not a good fit, particularly for the highly parallel NVMe storage. Apart from blocking nature, `ioctl` (and therefore passthrough I/O path) is far from various advancements (outlined in the Section 2.2) that have gone into regular read/write I/O path.
- All three paths (A), (B), and (C) rely on the block interface. However, **availability of block-interface is not guaranteed**. There are various situations when the block interface goes haywire. For example, (i) if a namespace is configured to transfer data and metadata as extended LBA(logical block address), the block device is marked with zero-capacity, which prevents further block/file I/O, (ii) ZNS device without zone-append is marked read-only, (iii) ZNS device with non-power of two zone-size is marked hidden, and (iv) any non-block command-set, e.g., Key-value, can not be operated with the block interface.

2.2 I/O advances with io_uring

io_uring is the latest and most feature-rich asynchronous I/O subsystem in Linux [29]. It operates at the boundary of user-space/kernel and covers storage and network I/O. The communication backbone is a pair of ring buffers, Submission Queue (SQ)/Completion Queue (CQ), shared between user-space and kernel. The application creates these rings by calling `io_uring_setup` call. It prepares the I/O by extracting an entry from SQ called `SQE`. It fills up the SQE and submits the I/O by calling `io_uring_enter` system call [6]. Finally, it obtains the completion by extracting an `CQE` entry from CQ. io_uring brings various advancements in the I/O path, some of which are outlined below:

- **Batching:**Allow submission of multiple I/O requests in one shot with a single system call.
- **SQPoll:**Syscall-free submissions. The application can offload the submission of I/O to a kernel thread that io_uring creates.
- **IOPoll:**Completion can be polled by setting IORING_SETUP_IOPOLL flag on the ring. This gives interrupt-free I/O.
- **Chaining:**Allows to establish ordering/dependencies among multiple commands at the time of submission. For example, write followed by a read (i.e., copy semantics) and commonly used sequence open-read-close. This is possible by chaining adjacent SQEs with the flag IOSQE_IO_LINK and submitting the entire chain in a go with a single syscall.

Ever since its inception, io_uring has added async variants of various sync system calls [52]. Two methods for turning a sync operation into async are outlined below.

- **Worker-based async**: spawn a worker thread and delegate sync operation to it. The advantage is the low implementation effort. However, this causes overheads and does not scale.
- **True-async**: fast and scalable as it does not involve worker threads. It relies on ensuring that the submitter does not block during the submission. Implementation effort grows as all components participating in the operation (e.g., file system, driver) should provide wait-free compliance.

io_uring employs both methods depending on the operation. For more common read/write operations, it uses true-async and falls back to worker-based async if need be. For known blocking operations (e.g., mkdirat, fsync), worker-based async is used in the first place itself.

3 Design considerations

3.1 Limitations of existing NVMe passthrough

The upstream Linux NVMe driver presents a passthrough interface to applications using these ioctl-driven opcodes:

- `NVME_IOCTL_IO64_CMD` is used to send NVMe I/O commands.
- `NVME_IOCTL_ADMIN64_CMD` is used to send NVMe admin commands.

Both these ioctls operate on `struct nvme_passthru_cmd64`, shown in Listing 1.

```
1  struct nvme_passthru_cmd64 {
2    __u8  opcode;
3    __u8  flags;
4    __u16 rsvd1;
5    __u32 nsid;
6    __u32 cdw2;
7    __u32 cdw3;
8    __u64 metadata;
9    __u64 addr;
10   __u32 metadata_len;
11   union {
12     __u32 data_len;
13     __u32 vec_cnt;
14   };
15   __u32 cdw10;
16   __u32 cdw11;
17   __u32 cdw12;
18   __u32 cdw13;
19   __u32 cdw14;
20   __u32 cdw15;
21   __u32 timeout_ms;
22   __u32   rsvd2;
23   __u64 result;
24 };
```

Listing 1: control structure that user-space sends for sync passthorugh

User-space forms the command using this structure, which is 80 bytes in size. Upon submission, the NVMe driver copies this to kernel-space using `copy_from_user` operation. It maps the data (line 9) and metadata buffer (line 8) and eventually submits the NVMe command to the device. The caller is put to wait until completion arrives. On completion, the primary result is sent to the user-space using the ioctl return value, and another one is updated into the result field (line 23). For the latter, the driver does a `put_user` operation.

While this interface allows to bypass the abstractions, it suffers several limitations:

- It is tied to the block device, which itself is fragile.
- Ioctl, due to its blocking interface, harms both scalability and efficiency. Figure 3 shows that io_uring random read scales perfectly, while ioctl driven read stays flat.
- As the above sequence outlines, there is a per-command overhead of copying command and result between user and kernel space.
- This interface can only be used by the root user.

3.2 Design goals

Based on the shortcomings mentioned in Section 2.1 and 3.1, we set the main design goals as follows:

- **Block I/O independence**: The block interface cannot represent the non-block command sets that NVMe has.

The new interface should have higher flexibility and cover all NVMe command sets regardless of their non-block semantics.

- **Catch-all user interface**: Adding a new syscall in Linux every time NVMe gets a new command is impractical. For every existing and future NVMe command, the solution should ensure a user interface without coining it.
- **Efficient and scalable**: NVMe represents fast and parallel storage. The new interface should have the same or higher efficiency and scalability than the direct block I/O path.
- **General accessibility**: The solution should not only be locked to the root/admin user.
- **Upstream acceptance**: The solution should become part of the official Linux repositories. This ensures that adopters do not have to reinvent or maintain off-tree code.

4 I/O Passthru in Kernel: Architecture and Implementation

The proposed I/O path is shown in Figure 2, with the label (D). It consists of a new char-interface `/dev/ng0n1` as the backend, which interfaces with io_uring using newly introduced `io_uring_command`.

We also considered Linux AIO but chose io_uring for two reasons. First, it is more efficient and feature-rich, as outlined in Section 2.2. Second, it is a more active subsystem in the upstream Linux kernel. The following sections detail the design and implementation by grouping those into three attributes - (i) availability, (ii) efficiency, and (iii) accessibility.

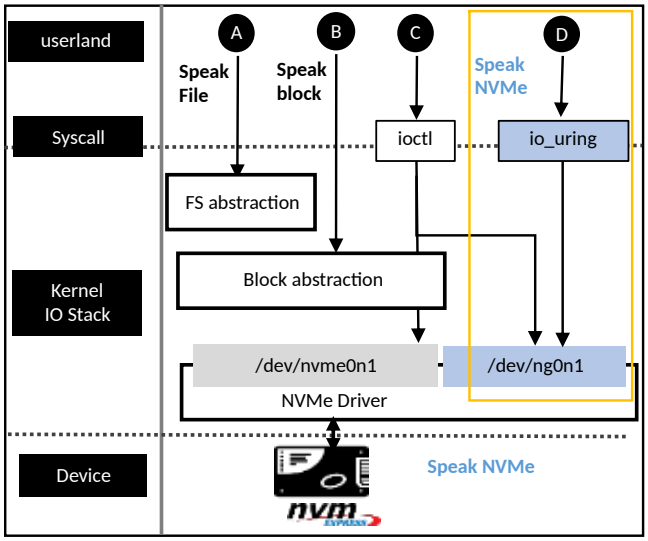


Figure 2: New passthrough I/O path, marked with (D) and enclosed with the dotted rectangle.

4.1 Availability: NVMe generic char interface

NVMe generic device solves the availability problem associated with the block device. We modify the NVMe driver to create a character device node for each namespace found on the NVMe device, and more importantly, this is done regardless of any unsupported feature that may break the block device (Section 2.1). Char device is also created for unknown command sets, e.g., anything other than NVM and ZNS. Therefore, the char interface is bound to appear for future command sets without requiring any further code changes to the NVMe driver. While the block device follows the naming convention `/dev/nvme<X>n<Y>`, the char device follows `/dev/ng<X>n<Y>`. The term `ng` refers to NVMe-generic as it applies to any NVMe command set. Listing 2 shows the `file_operations` for this character device. User-space can send any NVMe command through the character device using `ioctl`. Line 6 shows the `ioctl` handler in the NVMe driver. More importantly, the NVMe char device also talks to io_uring to enable a bunch of advances. This is shown in line 8 via the `uring_cmd` handler and elaborated in the next section.

```
1  const struct file_operations nvme_ns_chr_fops =
2  {
3    .owner          = THIS_MODULE,
4    .open           = nvme_ns_chr_open,
5    .release        = nvme_ns_chr_release,
6    .unlocked_ioctl = nvme_ns_chr_ioctl,
7    .compat_ioctl   = compat_ptr_ioctl,
8    .uring_cmd      = nvme_ns_chr_uring_cmd,
9    .uring_cmd_iopoll = nvme_ns_chr_uring_cmd_iopoll
       ,
10 };
```

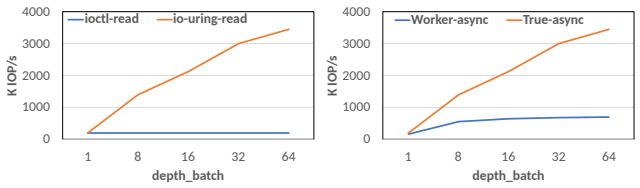
Listing 2: `file_operations` for NVMe char device

4.2 Infusing the efficiency & scalability

Solving the efficiency limitation of NVMe passthrough requires solving the more fundamental problem in Linux - coining an efficient alternative of ioctl. This alternative must be generic enough to be applied beyond the NVMe use case. To that end, we added three new facilities in io_uring:io_uring command, Big SQE, and Big CQE. Then, we outline how we employ these facilities to construct a new NVMe passthrough path. To further reduce the per I/O overhead, we wire up two more capabilities: fixed-buffer and completion-polling.

4.2.1 io_uring command

A relatively simple way to introduce ioctl-like capability in io_uring is to use the worker-based-async approach (Section 2.2). However, that will be anything but scalable. Figure 3 shows io_uring scaling for 512b random-read with and without the worker thread. With the default true-async approach, throughput soars as queue depth increases and reaches 3.5M IOPS. However, with the worker approach, the throughput



a. ioctl-read v/s io-uring-read b. defer-to-worker v/s true-async

Figure 3: Performance comparison

does not increase beyond 500K. Therefore, we go by the true-async design approach to add this new facility named `io_uring command`. User interface involves preparing `SQE` with a new operation code `IORING_OP_URING_CMD`. The command is to be placed inline within the `SQE`. This relieves the application from the command allocation and provides zero-copy communication as `SQE` is shared between user and kernel space. Regular `SQE` has 16 bytes of free space that the application can use for housing the command. The application gets to reap the result from the `CQE`. Regular `CQE` provides a signed 4-byte value as the result.

Big SQE: Regular SQE with 16 bytes of free space is not enough as the NVMe passthrough command is about 80 bytes in size (listing 1). Therefore, we introduce the facility to create the ring with a larger SQE. `Big SQE` is double the size of regular `SQE` and provides 80 bytes of free space. The application can set up the ring with `Big SQE` by specifying the flag `IORING_SETUP_SQE_128`.

Big CQE: Some NVMe commands return more than one result to the user-space. For example, the zone-append command returns the location where the write landed. And io_uring regular CQE lacks the ability to return more than one result. To tackle that, we introduce `Big CQE`, which is double the size of regular `CQE` and provides 16 bytes of extra space to return additional information to user-space. The flag `IORING_SETUP_CQE_32` allows the application to set up the ring with `Big CQE`.

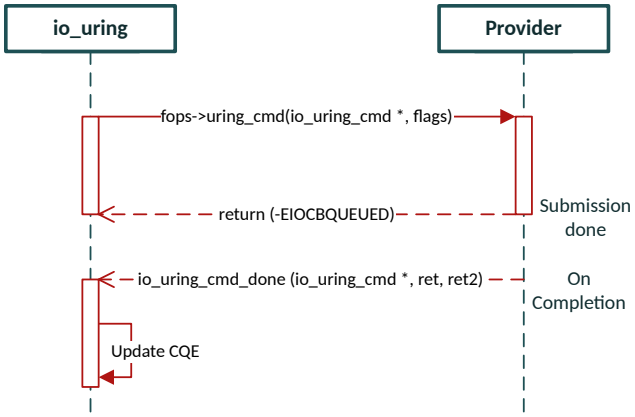


Figure 4: `uring_cmd` communication flow overview

We implemented `io_uring command` to be generic to support any underlying command. The **command provider** can be any kernel component (e.g., file system, driver) that collaborates with io_uring. While the NVMe driver is the first command-provider that got into the kernel, other examples include ublk [48] and network sockets [49]. The communication between io_uring and command-provider follows the true-async design approach (Section 2.2), and this is outlined in Figure 4. During the submission, io_uring processes the `SQE` and prepares another `struct io_uring_cmd` (Listing 3) that is used for all further communication. io_uring invokes the command-provider by `->uring_cmd` handler of file_operations. The provider does what is necessary for the submission and returns to io_uring without blocking. Actual completion is decoupled from submission and is rather done when the provider calls `io_uring_cmd_done` with the primary and auxiliary result. The primary result is placed into regular `CQE`, and the auxiliary result goes to `Big CQE`.

```
struct io_uring_cmd {
  struct file   *file;
  const void    *cmd;
  union {
    /* to defer completions to task context */
    void (*task_work_cb) (struct io_uring_cmd *cmd
    );
    /* for polled completion */
    void *cookie;
  };
  u32 cmd_op;
  u32 flags;
  u8 pdu[32]; /* available inline for free use */
};
```

Listing 3: `struct io_uring_cmd` for in-kernel communication

4.2.2 Asynchronous processing

For the new io_uring driven passthrough we add the following opcodes in NVMe driver:

- `NVME_URING_CMD_IO` : for NVMe I/O commands.
- `NVME_URING_CMD_IO_VEC`: vectored variant of the above.
- `NVME_URING_CMD_ADMIN`: for NVMe admin commands.
- `NVME_URING_CMD_ADMIN_VEC`: vectored variant of the above.

Vectored variants. Allow multiple data buffers to be passed from user-space, similar to what is possible for classical I/O using readv/writev syscalls. The above four operations expect a new `struct nvme_uring_cmd` as input. This is shown in Listing 4.

```
struct nvme_uring_cmd {
    __u8  opcode;
    __u8  flags;
    __u16 rsvd1;
    __u32 nsid;
    __u32 cdw2;
```

```
    __u32 cdw3;
    __u64 metadata;
    __u64 addr;
    __u32 metadata_len;
    __u32 data_len;
    __u32 cdw10;
    __u32 cdw11;
    __u32 cdw12;
    __u32 cdw13;
    __u32 cdw14;
    __u32 cdw15;
    __u32 timeout_ms;
    __u32   rsvd2;
};
```

Listing 4: control structure that user-space sends for uring passthorugh

Zero copy. User-space creates this structure within the `Big SQE` itself, eliminating the need for `copy_from_user`. Also, the auxiliary result is returned via `Big CQE`, so `put_user` is avoided. Therefore, this structure does not have a result field embedded into it. This ensures zero-copy in the control path.
Zero memory-allocations. Unlike sync passthrough, we ensure that command completion is decoupled from submission and the submitter is not blocked. This asynchronous processing requires some fields to be persistent (until completion), so these fields cannot be created on the stack. Dynamically allocating these fields will add to the latency of I/O. We avoid dynamic allocation by reusing the free space `pdu` inside the `struct io_uring_cmd` (Listing 3, line 12) for such bookkeeping.

4.2.3 Fixed-buffer

I/O buffers must be locked into the memory for any data transfer. This adds to the per I/O cost as buffers are pinned and unpinned during the operation. However, this can be optimized if the same buffers are used for I/O repeatedly. Therefore, io_uring can pin several buffers upfront using `io_uring_register`. The application can use these buffers for I/O using opcodes such as IORING_OP_READ_FIXED or IORING_OP_WRITE_FIXED.

We introduce this capability for `uring_cmd` using a new flag `IORING_URING_CMD_FIXED` instead. The application specifies this flag and buffer index in the `SQE`. Within the kernel, the NVMe driver checks the presence of this flag. If found set, it does not attempt to lock the buffer. Instead, it talks to io_uring to reuse the previously locked region. To that end, we add a new in-kernel API `io_uring_cmd_import_fixed` that any command provider can use.

4.2.4 Completion polling

io_uring allows the application to do interrupt-free completions for read/write I/O. This helps in reducing the context-switching overhead as the application engages in active polling rather than relying on the interrupts. NVMe driver, when loaded with `polled_queues = N` parameter, sets up N polled queue-pair (SQ and CQ) for which NVMe device does not generate the interrupt on command-completion. Since io_uring decouples submission from completion, async polling for completion is possible. This is more useful than sync polling, as the application can do other work rather than spinning on the CPU just after a single submission.

We extend async polling for uring_cmd too. For this, two things are done differently during submission in the NVMe driver - (i) polled-queue is chosen for command submission, (ii) a submission identifier `cookie` is stored in `struct io_uring_cmd` (line 8, Listing 3). Two identifiers are required to pinpoint the particular command during completion: (i) queue-identifier, in which the command is submitted, and (ii) command-identifier within that queue. These two identifiers are combined into a single 4-byte entity referred to as `cookie`.

For completion, a new callback `uring_cmd_iopoll` (line 9, listing 2) is added that implements the polling loop for matching completion. It extracts the `cookie` from `struct io_uring_cmd` and uses that to look for the matching completion entry in the NVMe completion queue.

4.3 Accessibility: from root-only to general

Linux uses discretionary access control (DAC) as the default way to manage object access. File mode is a numeric representation that specifies who (file owner, member of a group, or anyone else) is allowed to do what (read, write, or execute).

The VFS uses file mode to do the first level of permission checks when the application requests to open the file. The second level of check is done by the NVMe driver when the application issues the command using the opened file handle. However, the NVMe driver guards all passthrough operations by a coarse-granular CAP_SYS_ADMIN check that disregards the file mode completely.

Listing 5 shows an example in which ng0n1 has a less restrictive file mode, i.e., 0666, compared to ng0n2.

```
$ ls -l --time-style=+ /dev/ng*
crw-rw-rw- 1 root root 242, 0 /dev/ng0n1
crw------- 1 root root 242, 1 /dev/ng0n2
```

Listing 5: example file-mode for char device

Even though ng0n1 has been marked to allow unprivileged read/write operations, nothing goes through. Instead, it behaves the same as ng0n2. The all-or-nothing CAP_SYS_ADMIN check renders the passthrough interface limited to the root user.

We modify the NVMe driver to implement a fine-granular policy that takes file-mode and command type into account for access control. This policy is defined as follows:

- When CAP_SYS_ADMIN is present, everything is allowed as before. Otherwise, the command type (admin command or I/O command) is checked.

- Any I/O command that can write/alter the device is only allowed if file-mode contains write permission.
- Any I/O command that only reads/obtains the information from the device is allowed.
- Admin commands such as identify-namespace and identify-controller are allowed. This is because these commands provide information that is necessary to form the I/O command. Other admin commands are not allowed.

Beyond DAC, the `uring_cmd` also supports mandatory access control (MAC). A new Linux Security Module (LSM) hook is defined for `uring_cmd` and SELinux [15] and Smack [16] implement the respective policy for the hook.

4.4 Block layer: To bypass or not

Does NVMe passthrough mean bypassing the block layer? It is a common misconception. Passthrough is rather about not placing another layering over the device. The NVMe generic char-device, introduced in this work, does away with the block abstraction altogether and presents cleaner semantics than passthrough over the block-device. Figure 5 shows how I/O Passthru interacts with the block layer during the submission. The block layer implements many common functionalities,

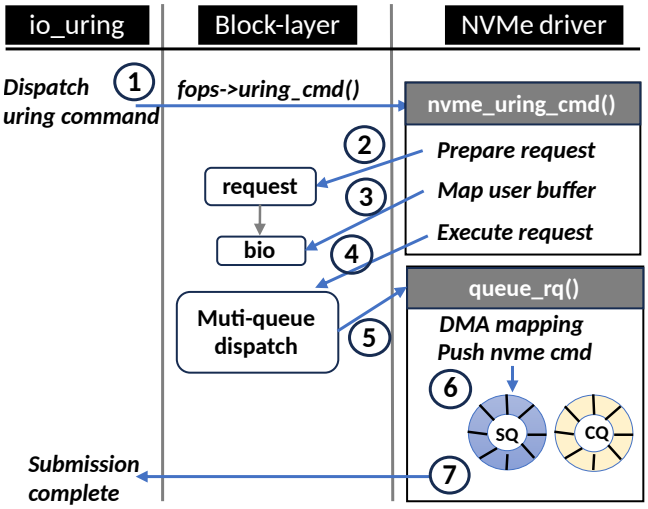


Figure 5: Integration with the block layer

either entirely or in collaboration with the underlying storage driver. Bypassing the block layer is not practical as it requires either reinventing or giving away the functionalities, turning the passthrough toothless. Table 1 presents the comparison.

- **Abstract device limits**. For example, the block layer makes it possible to send larger read I/O on a device that does not support single read commands to be larger than 64KB. For this, the block layer splits the larger read into many 64KB commands. Passthrough, by definition, does not abstract the device limits.

Feature	Block I/O	Passthrough I/O
`Abstract device limits`	Yes	No
`Scheduler bypass`	No	Yes
`Core to queue mapping`	Yes	Yes
`Command tags mgmt.`	Yes	Yes
`Timeout value`	Global	Per I/O
`Abort`	Yes	Yes

Table 1: Block layer functionalities: Block & Passthrough path

- **I/O scheduler**. Since I/O schedulers can merge the incoming I/Os, they are skipped for passthrough I/O. This is not a spoilsport, as not using the I/O scheduler performs best on NVMe SSDs. Generally, NVMe SSDs have deep queues and employ good internal I/O scheduling to meet SLAs. Prior studies have shown that Linux I/O schedulers (BFQ, mq-deadline, kyber) add significant overheads (up to 50%) and hamper scalability [12, 57]. Enterprise Linux distributions such as RHEL and SLES keep 'none' as the default scheduler for NVMe.
- **Muti queue**. Block-layer abstracts the device queues within the Blk-MQ infrastructure [36] and enables those to be shared among available cores. Passthrough also leverages this infrastructure.
- **Tag management**. The block layer manages the outstanding commands for each hardware queue. It manages the allocation/freeing of command IDs (tags) so that the driver does not need to implement flow control.
- **Command-timeout & Abort**.If a command takes longer than expected, the block layer can detect the timeout and abort the outstanding command. Passthrough supports user-specified timeout value (Listing 4, line 18), while block-path uses a hard-coded value for timeout.

5 Upstream

5.1 Kernel I/O Passthru Support

Table 2 shows the upstream progression of the proposed I/O path. All the constituent parts have made it to the official Linux kernel repository [14].

Feature	Kernel
Char-interface: initial support	5.13
Char-interface: any command-set	6.0
io_uring command	5.19
uring-passthrough for NVMe	5.19
Efficiency knobs (polling, fixed-buffer)	6.1
Unprivileged access for passthrough	6.2

Table 2: Upstream progression in the Linux kernel

5.2 Userspace I/O Passthru Support

5.2.1 xNVMe integration

xNVMe [51] is a cross-platform user-space library aimed at providing I/O interface independence to applications. xNVMe API abstracts multiple synchronous and asynchronous backends, including io_uring, libaio, and spdk. Application coded using xNVMe API can seamlessly switch among xNVMe's backends. We extend xNVMe to support a new asynchronous backend named `io_uring_cmd`. This backend works with NVMe character device `/dev/ngXnY`.

5.2.2 SPDK integration

SPDK contains a block-device layer, `bdev`, that implements a consistent block-device API over various devices underneath. For example, `NVMe bdev` is based on the NVMe driver of SPDK. `AIO bdev` and `uring bdev` are other examples that are implemented over Linux aio and io_uring respectively. We add a new bdev `xNVMe` in SPDK (shown in Figure 6). This single bdev allows to switch among AIO, io_uring, and io_uring_cmd. This bdev became part of SPDK since release version 22.09 [24].

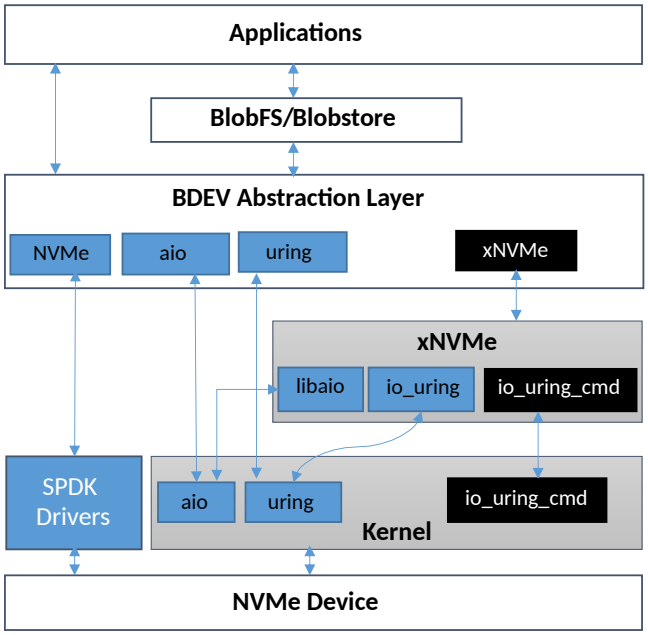


Figure 6: Overview of SPDK stack, and bdev_xnvme module

5.2.3 Tooling

nvme-cli [18] is modified to list character interface `/dev/ngXnY`. Any operation that nvme-cli can do on block-interface `/dev/nvmeXnY` can also be done on char-interface `/dev/ngXnY`.

Fio [30]: We add a new io-engine named `io_uring_cmd`. The user must pass a `cmd_type` when using this engine. This provides the flexibility to support other types of passthrough commands in the future. For NVMe passthrough, `cmd_type` is to be set as `nvme`, and the filename should be specified as `/dev/ngXnY`. This new ioengine is part of the Fio release since version 3.31. Fio repository contains a utility `t/io_uring` [32], which comes in handy to evaluate peak performance obtained via io_uring. We extend this utility so that io_uring NVMe passthrough can also be evaluated for peak performance.

Liburing [31] is the library that provides a simpler interface to io_uring applications. It is extended to support big-SQE and big-CQE. Moreover, we add a bunch of tests that issue uring-passthrough commands on the NVMe character device `/dev/ngXnY` [40].

6 Enabling NVMe interfaces with I/O Passthru

In this section, we present examples showing how the flexibility and efficiency of I/O Passthru help consume some NVMe features that are otherwise challenging to use in Linux.

6.1 Flexible Data Placement

Flexible data placement (FDP) is the latest host-guided data placement method in the NVMe standard. The ratified proposal [17] adds concepts such as reclaim unit (RU) and placement identifier (PID). RU is analogous to the SSD garbage-collection unit, and the host can place logical block addresses into RU by specifying PID in the write command. LBAs written with one-placement-identifier are not mixed with LBAs written with another placement-identifier. This helps to separate different data lifetimes and reduces write amplification in the SSD.

When multi-stream support was standardized in NVMe as directives, the Linux kernel developed the write-hint-based infrastructure that allowed applications to send the placement hints along with writes. However, this infrastructure is no longer functional as its core pieces have been purged from the mainline kernel [39]. I/O Passthru comes to the rescue as applications can send placement hints without worrying about vertical integration of FDP to various parts of the kernel storage stack. We demonstrate this with Cachelib, which can leverage FDP via I/O Passthru (Section 7.2). Also, FIO io_uring_cmd ioengine has supported FDP since version 3.34.

6.2 Computational Storage

Computational storage is a new architecture that allows the host to offload various compute operations to the storage, reducing data movement and energy consumption. The NVMe standardization is underway, and it involves presenting two new namespaces.

- **Memory namespace** refers to the subsystem-local-memory (SLM), a byte-addressable memory to enable the local processing of the SSD data. The host needs to issue new NVMe commands to (i) Transfer data between host-memory and SLM and (ii) Copy data between NVM namespace and SLM.
- **Compute namespace** represents various compute programs executed on the data residing in SLM. The host orchestrates the local data processing using a new set of NVMe commands: execute-program, load-program, activate-program, etc.

Supporting computational storage in Kernel is challenging because these new namespaces come with non-block semantics and various new unconventional commands. However, the generic char interface (`/dev/ngXnY`) comes up fine for both SLM and Compute namespace. All new NVMe commands can be issued efficiently with the I/O Passthru interface. Overall, this enables user-space to leverage computational storage without any changes to the Kernel.

6.3 End-to-End Data Protection

E2E data protection detects data integrity issues early and prevents corrupted data from being stored on the disk. Many NVMe SSDs have the ability to store extra metadata (8, 16, 32, 64 bytes) along with the data. This metadata can be interleaved with the data buffer (referred to as DIF) or in a separate buffer (referred to as DIX) [20]. This ability comes in handy to support erasure-coding, too. All or a portion (first or last bytes) of this metadata can contain protection information (PI) that contains checksum, reference tag, and application tag. The NVMe SSD controller verifies the PI contents while writing and reading.

Kernel support for E2E data protection [55] is limited, as shown in Figure 7. DIF is not supported as passing unaligned (e.g., 4096+8 bytes) data buffers is inconvenient. The block layer supports DIX as metadata is kept in a separate buffer. However, DIX is only supported if protection information resides in the first bytes of metadata. Also, PI is block-layer generated, and user-space applications cannot pass it due to a lack of interface.

I/O Passthru does not face buffer alignment checks or user-interface issues. The passthrough command structure allows applications to pass metadata buffer and length (Listing 4, lines 8 & 10). We have added DIF and DIX support in FIO io_uring_cmd ioengine.

7 Experiments

Table 3 summarizes our experimental setup. We conducted the experiments in three parts.

In the first part, we compare the efficiency of the new passthrough I/O path against the block I/O path on a direct-attached NVMe SSD. This is an apples-to-apples comparison

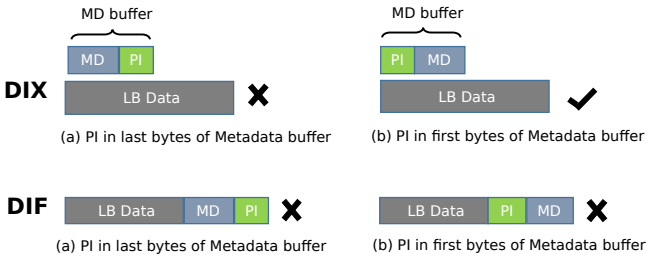


Figure 7: Block device limitations for DIF and DIX cases

between block interface `/dev/nvmeXnY` and char-interface `/dev/ngXnY`, as both are driven by io_uring. We exclude the sync passthrough path as it is known not to scale due to being `ioctl`-driven. We use Fio and t/io_uring utility, which is particularly suitable for peak-performance determination due to its low overhead. Both these are configured to run an unbuffered random read workload.

In the second part, we demonstrate the flexibility of the I/O Passthru interface by applying it in the real-world application Cachelib [3]. Cachelib is an open-source Caching engine from Meta which leverages RAM and SSD in the solution. Due to the nature of the workloads, Cachelib deployments can incur SSD Write Amplification (WAF > 2) on high SSD utilization scenarios. Therefore, the SSD utilization was limited to 50 percent in many production deployments. NVMe FDP tackles this problem of high WAF by segregating I/Os of different longevity types in the physical NAND media. We use the Samsung SSD that supports data placement using the NVMe FDP commands. Atop Cachelib, we run the production workload and compare the write-amplification against the case when FDP is not enabled.

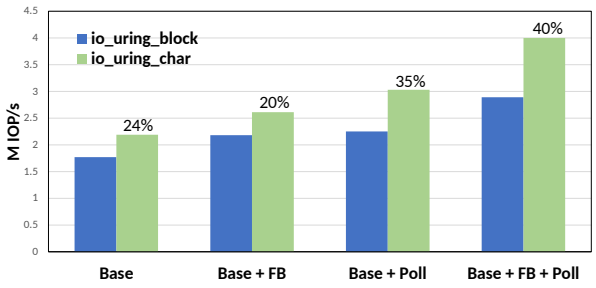
In the third part, we compare the scalability of block and passthrough I/O against the user-space SPDK NVMe driver.

Hardware	**Model**
CPU	AMD Ryzen 9 5900X 12-Core
Memory	DDR4 16 GB
Board	MSI MEG X570 GODLIKE
Storage	[1] Intel Optane P5800X, 400GB Spec: 5M (512b RR), 1.6M (4K RW) [2] Samsung FDP SSD, 7.5 TB
Software	**Version**
OS	Ubuntu 22.04 LTS
Kernel	Linux 6.2
fio	3.35
Cachelib	0.10.2

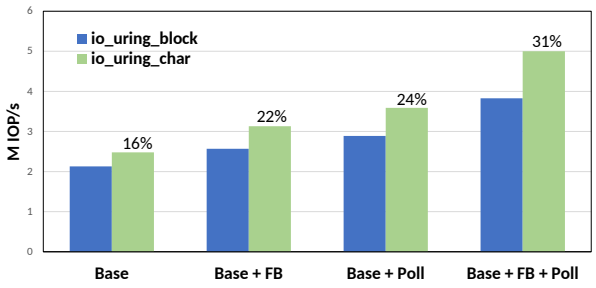
Table 3: Experimental configuration

7.1 Efficiency Characterization

The SSD used for this evaluation is notably optimized for 512b random reads and can show up to 5M IOPS as per its specification [41]. This is why we focus only

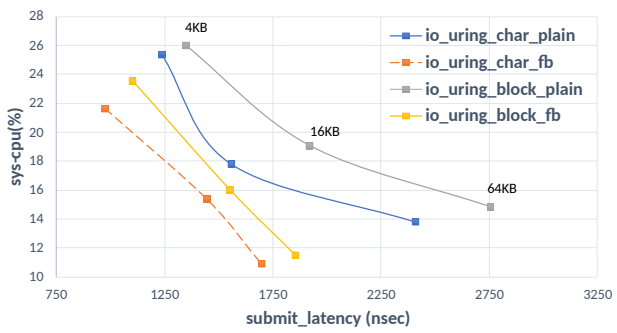


(a) Peak performance comparison on general kernel config

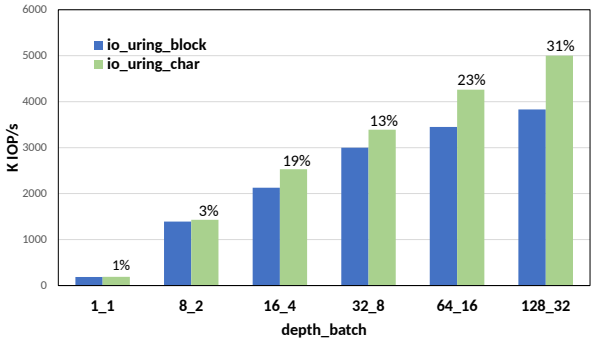


(b) Peak performance comparison on optimized kernel config

Figure 8: io_uring_char vs io_uring_block peak performance comparison



(a) Fixed-buffers effect on cpu-util and slat with increasing block size



(b) Scalability across queue-depths on optimized kernel config

Figure 9: io_uring_char vs io_uring_block scalability across queue-depths and fixed-buffers effect

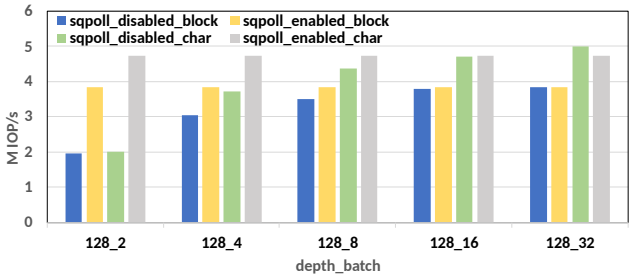


Figure 10: SQPoll and batching effect

on read-only workload, as this helps to reveal the software overheads and impact of optimizations readily. We use two kernel configurations to refine the test setup for overhead/efficiency measurements. The first one is the default configuration. The second one is a more performance-friendly configuration with CONFIG_RETPOLINE and CONFIG_PAGE_TABLE_ISOLATION options disabled. The kernel added these options to mitigate the Spectre [46] and Meltdown [50] hardware vulnerabilities. However, these come at the cost of performance overhead [9, 56].

Peak performance using single CPU core: We saturate the SSD to its maximum read performance, i.e., 5M IOPS. To that end, we measure the individual and combined impact of two knobs that elevate efficiency - (i) FB, which refers to fixed-buffers (Section 4.2.3), and (ii) poll, which refers to completion polling (Section 4.2.4). For this test, `t/io_uring` is bounded to a single CPU core, and it issues 512b random read at queue-depth 128 with batch size set to 32. Figure 8(a) shows the result on default kernel config. Fixed-buffer shows higher IOPS as the processing overhead of mapping buffers is minimized. Poll also shows improved numbers as interrupt-processing and context-switching overhead goes away. The performance of the io_uring passthrough path is better than the io_uring block path in all four cases. When both the knobs (FB and Poll) are combined, performance reaches its peak. Block I/O reaches up to 2.9M IOPS, while passthrough red I/O goes 35% higher and reaches 3.9M IOPS. However, SSD is capable of higher IOPS. Therefore, we repeat the test with an optimized kernel config. Figure 8(b) shows the results. There is notable improvement across all metrics. Block I/O elevates to 3.83M IOPS, while passthrough I/O goes 31% higher and saturates the SSD at 5M IOPS.

The reason is that I/O submission via the io_uring passthrough path involves less processing than the io_uring block path. It skips the attempts to split, merge, and i/o

scheduling. Table 4 shows the execution time of the respective io_uring handlers in the kernel (optimized config). The time these functions take corresponds to the time taken to submit the request to the device. The block-path handler, `io_read`, takes 209 nanoseconds for a single submission. The passthrough handler is 31% leaner and takes 144 nanoseconds for the submission.

Handler (io_uring)	Execution Time (nsec)
`io_read` (block)	209
`io_uring_cmd` (passthrough)	144

Table 4: Profiling of submission path

Scalability across queue-depths: We use `t/io_uring` to issue 512b random reads and vary the queue-depths (ranging from 1 to 128) and batch sizes (ranging from 1 to 32). Figure 9(b) shows the IOPS comparison between block and passthrough paths. At single queue-depth, utilization of the device bandwidth is lowest, and both paths yield the same performance. This is expected and denotes the synchronous I/O performance. As the queue-depth amplifies, parallel-processing capabilities of software and hardware get leveraged better, exhibiting a consistent increase in IOPS. A leaner submission path matters more when I/O requests arrive at a higher rate. At queue-depth 16, passthrough can process 19% more requests, which goes up to 31% at 128 queue-depth. **Cpu utilization and submission-latency**: comparison when fixed-buffer is enabled for block and passthrough I/O path. For this test, we use fio random read workload with single queue-depth and varying block sizes - 4 KB, 16 KB, and 64 KB. Figure 9(a) shows the result. In general, submission latency increases with the larger record size. This is because during the submission, (i) physical pages (usually 4KB in size) backing the I/O buffer need to be locked, and (ii) DMA (direct memory address) mapping for these pages needs to be done. A larger I/O buffer involves more physical pages, so it takes more time to perform the aforementioned steps. With a smaller block size, the submission and completion rate is high. But as we shift to large record sizes, the workload becomes more I/O bound. Therefore, CPU utilization is higher for 4KB record size. Fixed-buffer variants (of block and passthrough path) exhibit reduced submission latency and CPU cost of the I/O. io_uring char with fixed-buffer produces the most optimal combination of submission latency and CPU utilization.

SQPoll and batching: We use `t/io_uring` to issue 512b random reads with queue-depth set to 128 and vary batch sizes (ranging from 2 to 32). To reduce the contention and variance across multiple runs, we affine the sqpoll thread on a CPU core, which differs from the core to which t/io_uring is bounded. This is achieved by specifying the `IORING_SETUP_SQ_AFF` flag during io_uring's ring setup phase. Figure 10 compares the block and passthrough path with the SQPoll option disabled/enabled. SQPoll helps eliminate system call costs. With lesser batching (which would lead to more syscalls), enabling SQPoll results in better performance for both block and passthrough paths. With a batch size of 2, we get a 136% better performance by enabling the SQPoll option for io_uring passthrough. Both batching and SQPoll provide a means to reduce the syscall cost, but SQPoll requires an extra CPU core so that its active polling loop does not collide with the application thread that needs to submit the I/O.

7.2 Data-placement in Cachelib

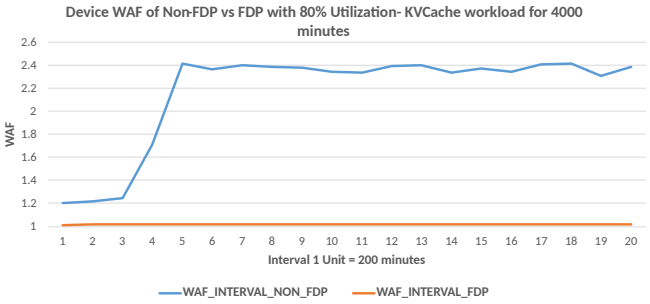


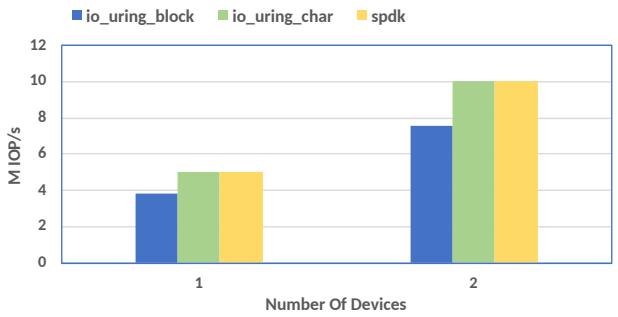
Figure 11: Cachelib WAF comparison: with and without FDP

Cachelib internally uses two I/O engines for data handling in SSD storage: **BigHash** and **BlockCache**. BigHash handles data of small sizes and random writes in 4K sizes. The large item engine, BlockCache, issues a sequential flash-friendly workload for data management. The leading cause of high WAF is the intermixing of these two I/O patterns in the physical NAND media and the resultant impact on SSD garbage collection. NVMe FDP commands allow the Host to send write hints to the SSD to avoid intermixing within the physical media. We modified Cachelib to use the I/O Passthru interface to send different placement identifiers with BigHash and BlockCache writes. The changes are being discussed for inclusion in the Cachelib upstream repository.

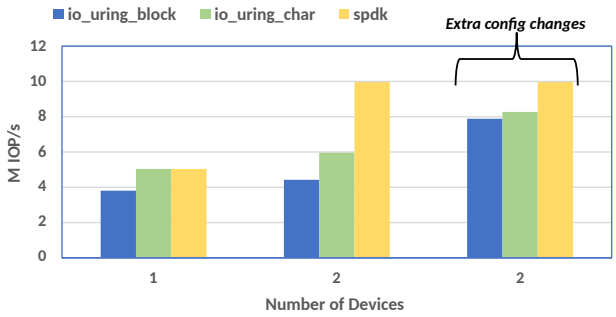
The evaluation was done using the built-in Cachebench tool. Cachebench can replay the Meta production workloads available from the Cachelib website [4]. We have used the write-only KVCache production workload for the experiments. We ran the workload for about 66 hours, and the resulting WAF comparison is shown in Figure 11. Without the placement hints, the intermixing occurs, and SSD WAF soars above 2. However, with the placement hints, intermixing reduces, and WAF remains close to 1.

7.3 Comparison against SPDK

We extend the peak performance test on two drives and compare scalability among io_uring_block, io_uring_char, and SPDK paths. We use the SPDK `perf` tool, which has minimal overhead during benchmarking. We used a distinct CPU core for each device in the first experiment. So, two cores are used for two devices. Figure 12(a) shows the comparison. The



(a) Peak performance with multiple devices



(b) Single core peak performance with multiple devices

Figure 12: io_uring_char vs io_uring_block vs spdk performance comparison

block path shows 7.55M IOPS with two devices, whereas the passthrough and SPDK paths show 10M IOPS. The second experiment examines the per-core scalability by forcing a single CPU core for both devices. Figure 12(b) shows the comparison. SPDK continues to saturate both the devices. Block path shows 4.4M, while passthrough reaches 6M IOPS.

Overall, I/O Passthru reduces the per-core efficiency gap but is still far from kernel-bypass solutions like SPDK. There are multiple reasons:

- The SPDK application (`perf` in this case) gets single-user luxury due to exclusive ownership of the NVMe device. It does not involve any extra code that must be written to ensure sharing and synchronization among multiple device users.
- I/O Passthru needs to use the block layer for its features (Section 4.4), such as hardware-queue abstraction, tag management, timeout/abort support, etc. The features come at the expense of extra processing in the I/O path.
- A few features do not fit the passthrough path, e.g., writeback-throttling and Block-cgroups. Turning off these features (by altering kernel config) cuts the processing and improves the I/O performance. The forthcoming 6.8 kernel skips these for passthrough I/O and does not require config changes. Beyond these, there are more general kernel configs that affect the I/O performance nonetheless. We turn off the forced preemption [7] and set the timer frequency to 100 Hz [8]. Figure 12(b) shows that, with extra config changes, block I/O performance improves to 7.9M, and passthrough I/O improves to 8.3M. Given the numerous kernel configuration choices, we feel more performance tuning is possible than we have explored here.

8 Discussion

8.1 I/O Passthru versus File systems

Relevance against file systems. Does passthrough make sense when Linux offers many stable and mature file systems such as XFS, BTRFS, and Ext4? We see two reasons to think that it does:

First, the maturity of these file systems comes in the way of embracing the emerging hardware. Production file systems get stability after going through battle-testing for a decade or so. Therefore, this stability is prioritized over adopting novel storage interfaces. In some cases, storage interfaces either change over a short period or do not get widespread adoption. Such cases pose the risk of bloated code and put a maintenance burden on the file system maintainers. Passthrough helps to consume new storage innovation readily in user space where real-world usefulness can be established. The compelling innovations can then find their way into robust file systems and other mature parts of the kernel.

Second, some large-scale storage systems have drifted away from file systems due to a multitude of reasons involving low performance, less control, and rigidity towards new hardware. Ceph [59] moved away from file systems and developed a new storage backend, BlueStore, which stores data directly on the raw storage device. BlueStore is the default storage backend, and it has been reported that 70% of Ceph users use this in production [26]. Aerospike also uses SSD directly using Linux direct I/O [2]. SPDK-based solutions do away with file systems. I/O Passthru presents a new choice to design storage backends with higher control, performance, and agility to embrace new hardware.

Performance against file systems. Kernel file systems create extra functionality above the block device, so their performance is usually capped by what is possible for block I/O. But FS-driven buffered I/O can perform better than block/passthrough I/O when it completes from DRAM (page cache) without causing thrashing. Table 5 compares filesystem buffered I/O performance with passthrough I/O. We use two types of fio random-read workloads, which vary in size - 8G and 32G. Both workloads spawn eight jobs, each doing 1GB I/O in the first case and 4GB I/O in the second case. FS buffered I/O performs better when the I/O size is less than the DRAM size (i.e., 16G) but worse when the workload cannot fit in the DRAM.

Read K-IOPS	I/O Size	
	8GB	32GB
Ext4 (buffered)	5767	2046
XFS (buffered)	5817	1915
Passthrough	4536	4524

Table 5: Randread performance comparison

8.2 Multi-tenancy and SQ/CQ limits

I/O Passthru does not involve dedicating the resources to a single application. Each io_uring ring (SQ/CQ pair) is a piece of preallocated memory that the application gets. This allocation is subject to the per-process limits. The application can use the same ring to do I/O on multiple files, as each SQE takes a distinct file handle as input. As for NVMe SQ/CQ, the upper limit comes from the NVMe SSD. The block layer abstracts available NVMe SQ/CQ under the per-core queues. The application that gets scheduled on a particular core submits its I/O to the underlying NVMe SQ mapped to that core. The architecture ensures that multiple applications run concurrently without reserving the hardware resources.

9 Related Work

SPDK allows applications to skip the abstraction layers and work directly with NVMe devices. The application needs to link with SPDK NVMe-driver to make use of it. However, SPDK NVMe-driver is a user-space library that maps the entire PCI bar to a single application. SPDK users face challenges when having to support multi-tenant deployment. The SPDK NVMe driver can operate only in polled mode. Also, storage is highly virtualized in a cloud environment, and root/admin access to raw PCIe devices is not feasible.

The abbreviation "ng" for the NVMe generic interface is inspired by "sg," which represents the SCSI generic interface. The sg driver, part of the Linux kernel SCSI subsystem, creates the SCSI generic interface [10]. The sg driver allows user applications to send SCSI commands to the underlying SCSI device. This communication from the user-space is done on character device node `/dev/sgX`, with syscalls such as write, read, and ioctl. Synchronous communication is done via `SG_IO` ioctl, which is analogous to `NVME_IOCTL_IO64_CMD` ioctl provided by NVMe(Section 4.2.2). Asynchronous communication using the sg interface is unhandy as it does not interface with io_uring or Linux AIO [13]. Instead, this requires pairing two system calls (read followed by a write) and signal handling [21,22]. `io_uring command` opens up an excellent way to upgrade the async communication mechanism of the SCSI generic interface.

Netlink sockets allow exchanging information in an async fashion between kernel and user-space [53, 58]. However, the netlink interface is designed for networking use cases and not for generic file I/O [34]. Some prior works proposed asynchronous ioctl via io_uring.

Pavel [33] and Hao [60] implemented by calling synchronous VFS ioctl handler in the io_uring worker context. This was anything but efficient (as Figure 3 shows). Kanchan et al. [42] early approach was tied to block-device and had allocation overhead. Jens [28] proposed a more generic and efficient approach involving `SQE` overlay. However, the `SQE` overlay did not forge ahead as (i) it provided 40 bytes of free space, which was insufficient for NVMe passthrough commands, and (ii) it brought certain plumbing unpleasantness in io_uring code. These were overcome after the introduction of `Big SQE` and cemented the proposal described in this paper.

10 Conclusion

Many new storage features/interfaces do not fit well within the block layer and face adoption changes due to the absence of appropriate syscall interfaces in Linux. Consequently, early adopters are left with two options: (i) use synchronous NVMe passthrough on block interface that may or may not exist, or (ii) switch to kernel-bypass solutions. We create a new alternative by adding a new passthrough path in the kernel. This path combines an always-available NVMe character interface with io_uring. Overall, this opens up an efficient way to use any current/future NVMe feature with the mainline kernel itself, i.e., all NVMe features with zero code in the kernel. We integrate this path to various user-space libraries/tools and present examples of how this can ease the enablement of FDP SSD, End-to-end data protection, and computational storage. As for efficiency, results demonstrate that the new passthrough path outperforms the existing block I/O path.

We also introduce an alternative of `ioctl` within io_uring. The `io_uring_command` infrastructure ensures that io_uring capabilities are not limited to existing mechanisms (i.e., classical read/write or other established syscalls) but will also be available to apply on new primitives. As is the case between host-system and storage, there will always be a requirement to communicate between user-space and kernel in a way that has not been imagined before. New pathways will remain in need. We hope `io_uring_command` will significantly ease up building efficient pathways between user-space and kernel.

Acknowledgement

We would like to thank our shepherd Sungjin Lee and the anonymous reviewers for their valuable feedback. We extend our appreciation to Joel Granados, Minwoo Im, Stefan Roesch, Vincent Kang Fu, Luis Chamberlain, and Christoph Hellwig for their contributions to I/O Passthru.

References

[1] Adding new system call. *https://docs.kernel.org/process/adding-syscalls.html*.

[2] Aerospike using raw storage. *https://docs.aerospike.com/server/operations/plan/ssd/ssd_setup*.

[3] Cachelib. https://github.com/facebook/CacheLib.

[4] Evaluating ssd hardware for facebook workloads. https://cachelib.org/docs/Cache_Library_User_Guides/Cachebench_FB_HW_eval/.

[5] ioctl, man page. https://man7.org/linux/man-pages/man2/ioctl.2.html.

[6] io_uring_enter, man page. https://man7.org/linux/man-pages/man2/io_uring_enter.2.html.

[7] Kernel config for preemption. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/kernel/Kconfig.preempt.

[8] Kernel config for timer frequency. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/kernel/Kconfig.hz.

[9] Kernel pti and its overhead. https://www.kernel.org/doc/html/latest/x86/pti.html.

[10] Kernel. scsi generic. https://docs.kernel.org/scsi/scsi-generic.html.

[11] Kioxia fl6 ssd. https://www.kioxia.com/en-jp/business/ssd/enterprise-ssd/fl6.html.

[12] Linux 5.6 i/o scheduler benchmarks. https://www.phoronix.com/review/linux-56-nvme.

[13] Linux aio. https://github.com/littledan/linux-aio.

[14] Linux kernel repository. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git.

[15] Lsm - selinux. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/admin-guide/LSM/SELinux.rst.

[16] Lsm - smack. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/Documentation/admin-guide/LSM/Smack.rst.

[17] Nvme 2.0 - tp 4146, fdp. https://nvmexpress.org/wp-content/uploads/NVM-Express-2.0-Ratified-TPs_12152022.zip.

[18] nvme-cli utility. https://github.com/linux-nvme/nvme-cli.

[19] Nvme command set. https://nvmexpress.org/developers/nvme-command-set-specifications.

[20] Nvme data protection. https://www.ripublication.com/ijaer19/ijaerv14n7_10.pdf.

[21] Scsi generic, async usage. https://tldp.org/HOWTO/SCSI-Generic-HOWTO/async.html.

[22] Scsi generic, theory of operation. https://tldp.org/HOWTO/SCSI-Generic-HOWTO/theory.html.

[23] Snia. computational storage. https://www.snia.org/computational.

[24] spdk-release. https://github.com/spdk/spdk/releases/tag/v22.09.

[25] Z-ssd. https://semiconductor.samsung.com/ssd/z-ssd/.

[26] AGHAYEV, A., WEIL, S., KUCHNIK, M., NELSON, M., GANGER, G. R., AND AMVROSIADIS, G. File systems unfit as distributed storage backends: lessons from 10 years of ceph evolution. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019), pp. 353–369.

[27] ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND PRABHAKARAN, V. Removing the costs of indirection in flash-based {SSDs} with nameless writes. In *2nd Workshop on Hot Topics in Storage and File Systems (HotStorage 10)* (2010).

[28] AXBOE, J. async ioctl. https://lwn.net/Articles/844875/.

[29] AXBOE, J. io_uring doc. https://kernel.dk/io_uring.pdf.

[30] AXBOE, J., ET AL. Flexible i/o tester. https://github.com/axboe/fio.

[31] AXBOE, J., ET AL. Liburing. https://github.com/axboe/liburing.

[32] AXBOE, J., ET AL. t/io_uring utility. https://github.com/axboe/fio/blob/master/t/io_uring.c.

[33] BEGUNKOV, P. async ioctl. https://lore.kernel.org/all/f77ac379ddb6a67c3ac6a9dc54430142ead07c6f.1576336565.git.asml.silence@gmail.com/.

[34] BERGMANN, A. How to not invent kernel interfaces. In *LinuxConf Europe 2007 Conference and Tutorials, 2.-5. září 2007* (2007).

[35] BJØRLING, M. Zone append: A new way of writing to zoned storage. *Santa Clara, CA, February. USENIX Association.[Cited on page.]* (2020).

[36] BJØRLING, M., AXBOE, J., NELLANS, D., AND BONNET, P. Linux block io: Introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference* (New York, NY, USA, 2013), SYSTOR '13, Association for Computing Machinery.

[37] BJØRLING, M., GONZALEZ, J., AND BONNET, P. LightNVM: The linux Open-Channel SSD subsystem. In *15th USENIX Conference on File and Storage Technologies (FAST 17)* (Santa Clara, CA, Feb. 2017), USENIX Association, pp. 359–374.

[38] GONZÁLEZ, J. Zoned namespaces: Standardization and linux ecosystem. *SDC EMEA* (2020).

[39] HELLWIG, C. Remove write-hint. https://lore.kernel.org/all/20220304175556.407719-2-hch@lst.de/.

[40] IM, M. generic per-namespace chardev. https://lore.kernel.org/linux-nvme/20210421074504.57750-2-minwoo.im.dev@gmail.com/.

[41] INTEL. Optane p5800x spec. https://www.intel.com/content/www/us/en/products/docs/memory-storage/solid-state-drives/data-center-ssds/optane-ssd-p5800x-p5801x-brief.html.

[42] JOSHI, K. async ioctl. https://lore.kernel.org/linux-nvme/20210127150029.13766-1-joshi.k@samsung.com/.

[43] JOSHI, K., AND S, S. Towards copy-offload in linux nvme. *SDC* (2021).

[44] JUNG, T., LEE, Y., AND SHIN, I. Openssd platform simulator to reduce ssd firmware test time. *Life Science Journal 11*, 7 (2014).

[45] KNIGHT, F. Storage data movement offload. *NetApp, Sep* (2011).

[46] KOCHER, P., HORN, J., FOGH, A., GENKIN, D., GRUSS, D., HAAS, W., HAMBURG, M., LIPP, M., MANGARD, S., PRESCHER, T., ET AL. Spectre attacks: Exploiting speculative execution. *Communications of the ACM 63*, 7 (2020), 93–101.

[47] KWAK, J., LEE, S., PARK, K., JEONG, J., AND SONG, Y. H. Cosmos+ openssd: Rapid prototype for flash storage systems. *ACM Trans. Storage 16*, 3 (jul 2020).

[48] LEI, M. ublk io_uring_cmd. https://lore.kernel.org/linux-block/20220628160807.148853-2-ming.lei@redhat.com/.

[49] LEITAO, B. uring_cmd network-socket. https://lore.kernel.org/lkml/20230627134424.2784797-1-leitao@debian.org/.

[50] LIPP, M., SCHWARZ, M., GRUSS, D., PRESCHER, T., HAAS, W., HORN, J., MANGARD, S., KOCHER, P., GENKIN, D., YAROM, Y., ET AL. Meltdown: Reading kernel memory from user space. *Communications of the ACM 63*, 6 (2020), 46–56.

[51] LUND, S. A., BONNET, P., JENSEN, K. B., AND GONZALEZ, J. I/o interface independence with xnvme. In *Proceedings of the 15th ACM International Conference on Systems and Storage* (2022), pp. 108–119.

[52] LWN. The rapid growth of io_uring. https://lwn.net/Articles/810414/.

[53] NEIRA-AYUSO, P., GASCA, R. M., AND LEFEVRE, L. Communicating between the kernel and user-space in linux using netlink sockets. *Software: Practice and Experience 40*, 9 (2010), 797–810.

[54] PETERSEN, M. K. Copy offload. here be dragons. https://oss.oracle.com/~mkp/docs/xcopy.pdf.

[55] PETERSEN, M. K. Dif, dix and linux data integrity. *Oracle, downloaded* (2010), 25.

[56] PROUT, A., ARCAND, W., BESTOR, D., BERGERON, B., BYUN, C., GADEPALLY, V., HOULE, M., HUBBELL, M., JONES, M., KLEIN, A., ET AL. Measuring the impact of spectre and meltdown. In *2018 IEEE High Performance extreme Computing Conference (HPEC)* (2018), IEEE, pp. 1–5.

[57] REN, Z., AND TRIVEDI, A. Performance characterization of modern storage stacks: Posix i/o, libaio, spdk, and io_uring. In *Proceedings of the 3rd Workshop on Challenges and Opportunities of Efficient and Performant Storage Systems* (2023), pp. 35–45.

[58] ROSEN, R. Netlink sockets. In *Linux Kernel Networking*. Springer, 2014, pp. 13–35.

[59] WEIL, S. A., BRANDT, S. A., MILLER, E. L., LONG, D. D., AND MALTZAHN, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation* (2006), pp. 307–320.

[60] XU, H. async ioctl. https://lore.kernel.org/all/1604303041-184595-1-git-send-email-haoxu@linux.alibaba.com/.

[61] ZHANG, J., LI, P., LIU, B., MARBACH, T. G., LIU, X., AND WANG, G. Performance analysis of 3d xpoint ssds in virtualized and non-virtualized environments. In *2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS)* (2018), IEEE, pp. 1–10.

A Artifact Appendix

Abstract

The evaluated artifact is provided in a git repository and contains the scripts used for running the experiments presented in this paper.

Scope

The artifact contains the scripts to reproduce the results obtained in Figure 8, Figure 9, Figure 10 and Figure 11.

Contents

The artifact contains the steps to build and install linux, links to patches for kernel and userspace contributions. It also contains the scripts used for performance benchmarks and cachelib experiments in the benchmark and cachelib-experiments subdirectory respectively. Also, each subdirectory has a separate README file, specifying the usage instructions.

Hosting

The artifact is available at https://github.com/anuj7781/io-passthru. All necessary instructions are provided in the README.md file. We encourage the users to use the latest version of the repository, since it may include bug fixes.

Requirements

The experiments can be run on any Linux machine (with 6.2 kernel). The benchmark experiments can be run on any NVMe drive, while the cachelib experiments can be run only on a FDP device. In order to reproduce the results, one needs to use the setup mentioned in Table 3.

Notes