I/O Interface Independence with xNVMe

Simon A. F. Lund
Samsung
Copenhagen, Denmark
simon.lund@samsung.com

Philippe Bonnet
IT University of Copenhagen
Copenhagen, Denmark
phbo@itu.dk

Klaus B. A. Jensen
Samsung
Copenhagen, Denmark
k.jensen@samsung.com

Javier Gonzalez
Samsung
Copenhagen, Denmark
javier.gonz@samsung.com

ABSTRACT

The tight coupling of data-intensive systems and I/O interface has been a problem for years. A database system, relying on an specific I/O backend for direct asynchronous I/Os such as libaio, inherits its limitations in terms of portability, expressiveness and performance. The emergence of high-performance NVMe Solid-State Drives (SSDs), enabling new command sets, compounds this problem. Indeed, efforts to streamline the I/O stack have led to the introduction of new, complex and idiosyncratic I/O interfaces such as SPDK, io_uring or asynchronous ioctls. What is the appropriate I/O interface for a given system? How can applications effectively leverage SSD and end-to-end I/O interface innovations? Is I/O interface lock-in a necessary evil for data-intensive systems and storage services? Our answer to the latter question is no. Our answer to the former questions is xNVMe, a cross-platform user-space library that provides I/O-interface independence to user-space software. In this paper, we present the xNVMe API, we detail its design and we show that xNVMe has a negligible cost atop the most efficient I/O interfaces on Linux, FreeBSD and Windows.

CCS CONCEPTS

• **Software and its engineering → Secondary storage**.

KEYWORDS

I/O interface, NVMe SSD, SPDK, io_uring

SYSTOR '22, June 13–15, 2022, Haifa, Israel

ACM ISBN 978-1-4503-9380-5/22/06.
https://doi.org/10.1145/3534056.3534936

ACM Reference Format:
Simon A. F. Lund, Philippe Bonnet, Klaus B. A. Jensen, and Javier Gonzalez. 2022. I/O Interface Independence with xNVMe. In *The 15th ACM International Systems and Storage Conference (SYSTOR '22), June 13–15, 2022, Haifa, Israel.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3534056.3534936

1 INTRODUCTION

I/O performance is of the essence for data-intensive systems (e.g., database systems, key-value stores, or HPC burst buffers). Such systems rely on NVMe SSDs to deliver I/O throughput of several GB/sec and I/O latency of a few microseconds. Leveraging such high-performance SSDs requires a streamlined storage stack [4] that provides user-space software with (i) control over allocation and layout policies, (ii) control over I/O scheduling, and (iii) an I/O path without redundancies or missed optimization opportunities. These systems rely on an I/O interface to manage I/O commands, payload memory and asynchronous accesses [14]. In Linux, this could be libaio, aio, io_uring [1], SPDK [19] or asynchronous ioctls [9].

Today, picking an I/O interface is a difficult choice that has a deep impact on system design.

Picking a single I/O interface is problematic because the entire system inherits its limitations in terms of portability, expressiveness or performance. Let us take an example. ScyllaDB relies on a log-structured merge tree and flushes immutable chunks of data to disk. It basically requires an I/O interface that supports asynchronous appending writes. ScyllaDB uses Seastar as a data plane that integrates networking and storage [12]. Seastar in turn relies on libaio to issue appending I/Os [11]. The problem is that libaio supports a limited number of file systems (ext4, jfs, xfs). Also, libaio quietly defaults to synchronous I/Os in case appending I/Os are not supported by the underlying file system (e.g., XFS on CentOS 7.1). Finally, because of libaio, ScyllaDB cannot take advantage of NVMe ZNS SSDs to support asynchronous appending writes efficiently. Picking a single I/O interface is so problematic that even the SPDK block device

layer (bdev) does not rely solely on the SPDK I/O interface. It also supports `io_uring`.

Supporting multiple I/O interfaces is expensive in terms of initial development and maintenance, as new features and possibly new I/O interfaces become available. For example, SPDK bdev needs to be significantly updated to leverage the new asynchronous ioctls interface being upstreamed in the Linux kernel.

In fact, today, I/O interfaces restrict the portability and performance of data-intensive systems. Worse, these systems can only benefit from SSD or I/O interface innovations at a significant cost in terms of software refactoring. This problem puts early technology adopters in a difficult position. Counting on an efficient I/O interface that enables end-to-end optimizations without requiring significant changes to stable projects is imperative for the success of new storage protocols.

We denote **I/O interface independence** the following property of a data-intensive system: *changing I/O interface does not require refactoring the rest of the system.* I/O interface independence addresses the problems listed above, as a system can easily change I/O interface when its requirements evolve or when new I/O interfaces become available. Our hypothesis is that I/O interface independence can be achieved at negligible performance cost.

In this paper, we detail the design of xNVMe, first introduced in [18], a minimal spanning layer in user-space that bridges the existing range of I/O interfaces. xNVMeprovides a universal abstraction for submitting I/Os to NVMe-based SSDs, thus implementing I/O interface independence.

Our contributions are the following:

(1) We introduce xNVMe as the narrow waist of the NVMe-based storage stack. We detail its API and its design.
(2) We evaluate the performance overhead of xNVMe on NVMe SSDs and show that introducing xNVMe results in a negligible costs in the worst cases and a significant improvement in the best cases atop the most efficient I/O interfaces on Linux, FreeBSD and Windows.

xNVMe is open-source. Code, experiments and results[1], as well as documentation[2] are available online.

2 API

Let us first focus on how programmers submit I/Os. An I/O is issued on an NVMe device (directly by opening the device file or through a file) via an I/O interface. Each I/O is a command involving a payload in memory (a buffer to/from which data is transferred) and an address on the device. The nature of the command and address depend on the NVMe namespace. As we stated in the Introduction, the problem in terms of I/O independence is to define an API that is as simple as can be, uniform across a range of different I/O interfaces and extensible so that new features can easily be added. There is a fundamental difference between synchronous and asynchronous I/Os. Let us consider them in turn.

2.1 Synchronous I/Os

The program in Figure 1 is the simplest example of synchronous I/O on a ZNS drive using xNVMe. We note that error handling in xNVMe is uniform across all I/O interfaces.

```
1  struct xnvme_opts opts = xnvme_opts_default();
2  struct xnvme_dev *dev;
3  const struct xnvme_geo *geo;
4  uint32_t nsid = cli;
5  struct xnvme_spec_znd_descr zone = { 0 };
6  size_t buf_nbytes;
7  char *buf = NULL;
8
9  dev = xnvme_dev_open("/dev/nvme0n1", &opts);
10 nsid = xnvme_dev_get_nsid(dev);
11 geo = xnvme_dev_get_geo(dev);
12
13 xnvme_znd_descr_from_dev(dev, 0x0, &zone);
14
15 buf_nbytes = zone.zcap * geo->lba_nbytes;
16 buf = xnvme_buf_alloc(dev, buf_nbytes, NULL);
17 memset(buf,0,buf_nbytes);
18
19 struct xnvme_cmd_ctx ctx;
20 for (uint64_t i = 0; i < zone.zcap; ++i) {
21   ctx = xnvme_cmd_ctx_from_dev(dev);
22   void *payload = buf + i * geo->lba_nbytes;
23   xnvme_nvm_write(&ctx, nsid, zone.zslba + i, 0,
       payload, NULL);
24 }
25
26 xnvme_buf_free(dev, buf);
27 xnvme_dev_close(dev);
```

Figure 1: Synchronous writes on a ZNS device.

Once a device is opened (line 9), its namespace id and geometry can be derived from it (lines 10 and 11) as well as the zone descriptor starting at the given LBA (here 0x0 at line 13).

xNVMe provides a primitive for buffer allocation (line 16). This makes it possible for xNVMe to rely on any DMA transferrable memory made available by the underlying I/O interface. The number of bytes allocated to the buffer (line 15) is the capacity of the zone multiplied by the size of each logical block.

The program fills a zone with synchronous writes (lines 19-24). The loop iterates over the capacity of the zone. For

[1] https://github.com/OpenMPDK/xNVMe
[2] https://xnvme.io/docs

each block, a synchronous command context is derived from the device (line 21), then the `xnvme_nvm_write` primitive is used to generate a command writing a range of logical blocks denoted by (a) a base LBA (the zone start LBA to which is added the product of the counter and block size) and (b) the number of LBAs to add to this base address to get the last LBA written (0 indicates that a single block is written). `xnvme_nvm_write` abstracts the call to `xnvme_cmd_pass` (see Section 3.1).

After the writes are performed, the program frees up the allocated buffer and closes the device (lines 26 and 27).

This program can run unchanged on any I/O interface that supports ZNS. Only the device `opts` needs to be adjusted. Porting this code from ZNS to a block device requires deleting references to zones: (a) line 12 is deleted, (b) `buf_nbytes` is directly obtained from `geo->lba_nbytes` on line 15, and (c) to write a range of contiguous LBAs, the range and the base LBA would have to be given explicitly instead of `zone.cap` and `zone.zslba`. Such a program can run unchanged on any I/O interface that supports block devices, as long as the device `opts` is updated.

2.2 Asynchronous I/Os

Programming asynchronous I/Os requires more work. Indeed, this requires explicit management of submissions and completions. In xNVMe, we provide a command queue that can be associated to a device. We detail our design in Section 3.3. Command queues abstract submission and completion queues. Completions are managed through callbacks, that need to be defined separately, and then associated to a queue. The code in Figure 2 defines a callback function. It defines first a struct for the callback arguments and then it defines the callback itself. The signature, defined by xNVMe requires the callback to take as first argument a pointer to a command context and as second argument a pointer to the callback arguments. Again, we omit all error handling from the code. The callback simply increases a counter of completed I/Os and puts back the command context on the queue, so that it can be reused.

The program in Figure 3 runs a single asynchronous read from a block device. I/O processing itself is organized around the following steps: (i) a device is opened based on a URI and instrumented via `opts` (line 2, libaio is used as a I/O interface), (ii) a queue is created with a given queue depth and associated to the device, the callback functions are attached to the queue (lines 12,13), (iii) the payload buffer is allocated (lines 15-17), (iv) the asynchronous command context is derived from the queue (line 18). The read is submitted through the `xnvme_nvm_read` command (line 21, the read targets a single LBA at address 0x0). This command abstracts the call

```
1  struct cb_args {
2    uint32_t completed;
3    uint32_t submitted;
4  };
5
6  static void
7  cb_pool(struct xnvme_cmd_ctx *ctx, void *cb_arg)
8  {
9    struct cb_args *cb_args = cb_arg;
10   cb_args->completed += 1;
11   xnvme_queue_put_cmd_ctx(ctx->async.queue, ctx);
12 }
```

Figure 2: Callback handling the completion of asynchronous reads.

to `xnvme_cmd_pass`. (v) we consider that the read is succesfully submitted, so the program waits for completion (line 25). (vi) In the last step, the program prints the number of submitted and completed I/Os, deallocates buffers and closes the device.

2.3 Discussion

Even if they are very simple, these two programs illustrate the key features of the xNVMe API. First, the opts argument makes it possible for programmers to declare which I/O interface is used to access a SSD. Porting a given program to a compatible I/O interface requires just changing `opts`, e.g `opts = {.async="io_uring"}`. Second, porting a program between NVM and ZNS Command Sets focuses on the necessary management of the zone write pointer. Most primitives are re-used across NVMe namespaces.

This simplicity is due to the fact that xNVMe encapsulates I/O interface dependencies behind the device, queue and command context abstractions. Thanks to this encapsulation, programs using xNVMe tend to be much more concise than programs using I/O interfaces directly. For instance, the SPDK bdev xNVMe module, providing I/O interface independence over multiple I/O interfaces, is implemented in 481 source lines of code (SLOC), which is significantly less than the legacy module implemented specifically for `io_uring` in 657 SLOC.

NVMe is in a constant evolution. The evolution includes new command sets and existing ones expanded. With such advances in NVMe device capabilities, the challenge becomes how the software ecosystem can adopt and make efficient use of them. xNVMe is designed to rapidly take advantage of any extension of the underlying NVMe device regardless of the intermediate operating system support. For example, we are preparing a module that will expose the NVMe computational storage functionalities to applications, with command-construction functions and helper functions similar to those

```c
struct cb_args cb_args = { 0 };
struct xnvme_opts opts = { .async = "libaio" };
struct xnvme_dev *dev = NULL;
uint32_t nsid;
char *buf = NULL;
size_t buf_nbytes;
struct xnvme_queue *queue = NULL;
const int qd = 16;

dev = xnvme_dev_open("/dev/nvme1n1", &opts);
nsid = xnvme_dev_get_nsid(dev);
xnvme_queue_init(dev,qd,0,&queue);
xnvme_queue_set_cb(queue,cb_pool,&cb_args);
buf_nbytes = xnvme_dev_get_geo(dev)->nbytes;
buf = xnvme_buf_alloc(dev,buf_nbytes,NULL);
memset(buf,0,buf_nbytes);
struct xnvme_cmd_ctx *ctx;
ctx = xnvme_queue_get_cmd_ctx(queue);

// Submit a read command
xnvme_nvm_read(ctx,nsid,0x0,0,buf,NULL);
cb_args.submitted += 1;

// Done submitting, wait for outstanding I/Os
xnvme_queue_drain(queue);

printf("cb_args:submitted:%u,completed:%u}",
       cb_args.submitted,cb_args.completed);

xnvme_buf_free(dev,buf);
xnvme_queue_term(queue);
xnvme_dev_close(dev);
```

Figure 3: Asynchronous read from a block device.

provided for the NVM and ZNS command sets. This is an essential value of xNVMe: all the low-level plumbing is handled, so it is possible to define new NVMe commands in about 30 SLOC. Here is for example, in Figure 4, the implementation of the ZNS append command.

3 DESIGN

We consider the following requirements for xNVMe: *(a) Performance*: xNVMe should have minimum performance penalty with respect to the underlying I/O interfaces; *(b) Simplicity*: xNVMe should not increase the complexity of data-intensive systems. *(c) Uniformity*: xNVMe should minimise the number of lines of codes that must be changed to port an application from one I/O interface to another, ideally across operating systems; *(d) Extensibility*: Supporting new NVMe features e.g., KV command set or computational storage); or adding new I/O interfaces (e.g., DirectStorage on windows) should require minimal changes to xNVMe.

```c
int
xnvme_znd_append(struct xnvme_cmd_ctx *ctx,
    uint32_t nsid,uint64_t zslba,uint16_t nlb,
    const void *dbuf,const void *mbuf)
{
  void *cdbuf = (void *)dbuf;
  void *cmbuf = (void *)mbuf;

  size_t dbuf_nbytes =
    cdbuf ? ctx->dev->geo.lba_nbytes*(nlb+1) : 0;
  size_t mbuf_nbytes =
    cmbuf ? ctx->dev->geo.nbytes_oob*(nlb+1) : 0;

  ctx->cmd.common.opcode =
    XNVME_SPEC_ZND_OPC_APPEND;
  ctx->cmd.common.nsid = nsid;
  ctx->cmd.znd.append.zslba = zslba;
  ctx->cmd.znd.append.nlb = nlb;

  return xnvme_cmd_pass(ctx,cdbuf,dbuf_nbytes,
    cmbuf,mbuf_nbytes);
}
```

Figure 4: Implementation of the ZNS append command in xNVMe.

These requirements drive our design. We make the common case (reads and writes) fast and intuitive. Note that our definition of the NVMe common case includes (i) synchronous and asynchronous I/Os on (ii) NVM and ZNS command sets. By default, xNVMe is set up to reduce I/O latency. Configuration is needed to trade increased I/O latency for reduced CPU utilization.

3.1 Architecture

We design xNVMe around a single module, denoted libxnvme. This module is responsible for handling NVMe devices, managing memory and executing commands. libxnvme is positioned in between NVMe abstractions exposed to the upper layers as illustrated in Section 2 (administration commands, NVM or ZNS command sets) and abstractions of the underlying I/O interfaces (across platforms). This position of libxnvme as a minimally sufficient spanning layer supporting NVMe command sets across I/O interfaces guarantees that, by construction, xNVMe is a narrow waist that satisfies the hourglass theorem [2]. Figure 5 presents the xNVMe architecture.

At its core, libxnvme provides a generic command interface (see Figure 6). This function takes as parameters a pointer to a command context (detailled in subsection 3.2.2), together with pointers to payload data (dbuf) and metadata (mbuf), with their associated sizes. It is a generic interface

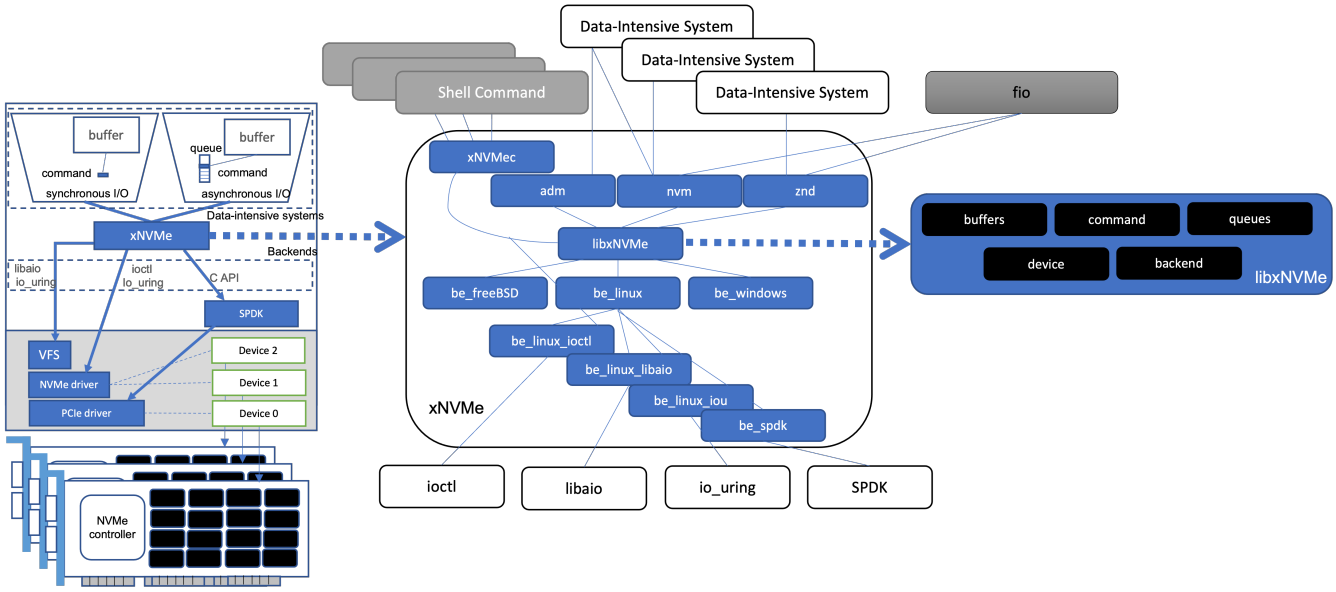

Figure 5: Positioning of xNVMe in the storage stack (left). Architecture of xNVMe (center). Architecture of libxnvme (right).

```
int
xnvme_cmd_pass(struct xnvme_cmd_ctx *ctx,
               void *dbuf, size_t dbuf_nbytes,
               void *mbuf, size_t mbuf_nbytes);
```

Figure 6: Generic command interface at the core of libxnvme.

for passing commands to the NVMe driver, independently of both the NVMe command-set/namespace and the underlying I/O interface. xnvme_cmd_pass is also agnostic to the I/O type, i.e., synchronous or asynchronous. The rest of libxnvme makes it possible to (a) generate a command context by opening a device through a given I/O interface and creating queues for asynchronous I/Os and (b) allocate buffers for the payload in DMA transferrable memory if this is supported by the underlying I/O interface or else in virtual memory. libxnvme is thus a deep module [17] whose interface defines five interrelated abstractions: devices, backends, queues, commands and buffers.

The full complexity of the NVMe standard is represented through an additional interface, libxnvme_spec, not shown on the figure, layered atop libxnvme. Reads and writes are available for the NVM and the ZNS namespaces through specific modules that access libxnvme and libxnvme_spec.

xNVMe makes it easy to create shell commands through the xnvmec[3] module to (i) enumerate devices or obtain NVMe identification about a controller or a namespace, (ii) issue commands, including read/write/append or retrieve information from a ZNS drive, or (iii) issue commands, including read/write or retrieve information from an NVMe SSD exposing an NVM namespace (i.e., a block device) are provided together with the xNVMe library. We have also implemented a xNVMe IO Engine[4] for fio that we use for our experiments (see Section 4).

3.2 libxnvme

We now detail the key characteristics of libxnvme.

3.2.1 Runtime Instrumentation. When a file or device is opened, then the library runtime does a best-effort to select an appropriate implementation to handle the given URI. That is, when giving a file-path to a regular file, the runtime can use operating system managed I/O interfaces such as libaio, io_uring, POSIX aio, and IOCP depending on operating system and interface support. Similar decisions are made by the runtime when device files, such as char devices, block devices, and specifically NVMe are probed to check which OS managed ioctl interface they respond to. Lastly when using non-operating system managed interfaces, the user can provide a fabrics address x.y.z.k:4422 or a PCIe address 0000:00:05.0, in which case the library runtime will use corresponding lower-level interfaces to communicate with a device over fabrics or communicate directly with a device over PCIe using a user-space NVMe driver.

[3]https://xnvme.io/docs/next/capis/xnvmec

[4]https://xnvme.io/docs/latest/tools/fio

In case the user wants to explicitly control which interface the library runtime will use, then it is possible to do runtime instrumentation by providing an `option-struct` when opening a device. Thus, changing an application from using e.g. `io_uring` instead of libaio is as simple as assigning the option-struct:

```
# Change async interface via runtime options

# Explicitly select libaio
struct xnvme_opts = {.async = "libaio" };

# Explicitly select io_uring
struct xnvme_opts = {.async = "io_uring" };
```

Figure 7: Change of I/O interface via runtime options.

The code in Figure 7 is the only change required to switch the underlying I/O interface from one interface to another. This runtime instrumentation is optional, flexible and extensible.

3.2.2 Command Context. The generic command interface `xnvme_cmd_pass` at the heart of `libxnvme` takes a command context as parameter. The examples of synchronous and asynchronous programs above manipulate command contexts as opaque data structures. Command contexts are used to carry the state associated to a command across modules within xNVMe. This data structure is presented in Figure 8.

It contains (i) a 64B representation of the NVMe submission queue entry and a 16B representation of the completion queue entry, (ii) pointer to the target device, together with pointers to a queue and callback function/arguments for an asynchronous I/O, (iii) options and reserved field, and (iv) a list entry enabling the inclusion of the structure in singly linked lists, such as command-queues and resource pools.

The command context refers to devices (`xnvme_dev`) and queues (`xnvme_queue`). These data structures are specific to each backend. To achieve polymorphism, we rely on two different mechanisms. `xnvme_dev` contains information about namespace, geometry and identity together with a backend interface, i.e., a collection of function pointers that encapsulate the device management, memory management and I/O processing capabilities of the underlying I/O interface. `xnvme_queue` is composed of a 24B base, which is common to all backends and contains a pointer to `xnvme_dev`, 232B of backend-specific state and storage space for a request pool, i.e., an array of `xnvme_cmd_ctx` defined as a flexible array member. The request pool makes it possible to maintain a queue of commands in virtual memory in case the underlying backend does not support it (e.g., ioctls). It also serves as a unified storage pool for pairs of submission queue and completion queue entries.

```
struct xnvme_cmd_ctx {
  // Command to be processed
  struct xnvme_spec_cmd cmd;
  // Completion result
  struct xnvme_spec_cpl cpl;
  // Device associated with the command
  struct xnvme_dev *dev;
  // Context for async IOs
  struct {
    // Queue
    struct xnvme_queue *queue;
    // callback function
    xnvme_queue_cb cb;
    // callback function arguments
    void *cb_arg;
  } async;
  // Options filled by helper functions
  uint32_t opts;
  // reserved for library internals
  uint8_t be_rsvd[4];
  // singly-linked list entry
  SLIST_ENTRY(xnvme_cmd_ctx) link;
};
```

Figure 8: xNVMe command context.

3.3 Backends

`libxnvme` relies on the functionalities from the underlying I/O interfaces for device management, memory management and queues. For the sake of space limitation, we focus on how SPDK, `io_uring` and ioctls are integrated in xNVMe.

3.3.1 SPDK. The SPDK backend, denoted `be_spdk`, is built on top of the C API provided by the SPDK NVMe driver.

Device management corresponds to approximately 1000 LOC in `be_spdk`. This complexity comes from the necessity to configure and initialize the SPDK environment, enumerate controllers/targets and available namespaces across the supported transport mechanisms, including PCIe, TCP and RDMA and match them with the controller and namespace as named by the user provided URI. This complexity is hidden from xNVMe programmers.

Memory management directly uses `spdk_dma_malloc` to allocate buffers in DMA-transferrable memory. Queue initialization directly allocates submission-completion queue pairs with `spdk_nvme_ctrlr_alloc_io_qpair`. A pointer to the allocated queue pair is stored in the backend-reserved portion of the associated `xnvme_queue`. Note that this design preserves the zero-copy property of SPDK. Note also that as part of the ~1000LOC is a queue-pair, protected by a mutex, along with a callback function, these parts are hidden from the user, but they serve the synchronous API provided to

xNVMe programmer, such that they can use the NVMe driver without creating a queue, callback functions and arguments.

I/O submission taps into lowest-level submission interfaces provided by the driver. It maps the user-defined command with data and metadata and payloads as contiguous DMA-transferable buffers, to context, payload data and metadata together with a callback function and its arguments, passing the command context down with it, such that it is available when completions are processed.

3.3.2 `io_uring`. The `io_uring` backend is `be_linux_iou`. It is one of several backends associated to I/O interfaces provided by the Linux kernel.

Device management is common to all Linux backends. The state of the Linux backends contains a file descriptor associated to the device file derived from the device URI (directly when accessing a raw device, indirectly when accessing a file). This device file contains all necessary information about the device identity and geometry.

Memory management is mapped onto malloc or any other dynamic memory allocator in virtual memory. Queue initialization results in the creation of a `io_uring` queue pair. A queue pair, denoted a ring (`struct io_uring`), is stored in the reserved portion of `xnvme_queue`. The command contexts are either setup manually by the user or using the helper functions provided in the API. The command portion of the command context provides the information needed to setup submission queue entries directly in `io_uring`. It takes about 10 instructions.

I/O submission relies on `io_uring_submit`, which takes a ring as parameter. By default, `io_uring` is configured with submission queue polling disabled, and is enabled via `opts={.poll_sq=1}`. Submission queue entries are then copied to the software queue and then on a hardware dispatch queue within the block layer, outside of the programmer's control.

3.3.3 ioctls. The ioctls backend, denoted `be_linux_ioctl`, shares device management and buffer management features with `be_linux_iou`. **I/O submission** is handled by wrapping around the kernel NVMe driver ioctl interface, allowing submission of arbitrary I/O and admin commands. What is special about this backend is that ioctls are synchronous. As a result, asynchronous I/Os must be implemented in the `be_linux_ioctl` backend. First, in terms of **memory management**, a queue pair is created in virtual memory. The request pool is used as a staging area for the submission queue. Second, a pool of threads is created to (i) implement submission queue polling and submit a synchronous ioctl as soon as an entry is available in the submission queue, and (b) completion handling is then done by invoking the callback as the ioctl call unblocks.

3.3.4 asynchronous ioctls. The `be_linux_ioucmd` backend is an extension of `be_linux_iou` and thus shares all the interface implementations for device, buffer, and queue management. The key differentiating area is **I/O submission**. A special purpose command is used, which enables the use of the `ioctl()` syscall via `io_uring`, and specifically using the NVMe driver ioctl() interface, thereby replacing the need for a theadpool in user-space with the asynchronous ability of the `io_uring` interface. Greater detail on the system interface is available on the Linux kernel mailing-list and presented at the Storage Developer Conference 2021 [9].

3.4 Discussion

Throughout this section, we have illustrated xNVMe's simplicity, uniformity across I/O interfaces and its extensibility. Does xNVMe hide all differences across I/O interfaces? No. A fundamental difference between `io_uring` and SPDK concerns hardware queues. With `io_uring`, it is the kernel that manages hardware queues, while with SPDK it is the application. This has a clear impact on I/O latency, as there are fewer memory copies with SPDK. The price to pay is a dependency between the application and the physical characteristics of the underlying SSD. Indeed, the number of queues that can be opened on a SSD is limited by the characteristics of the SSD controller (e.g., 128 queues for the Elpis controller introduced by Samsung with the 980 Pro in 2020, as opposed to 8 queues on Intel Optane SSDs we used in our experiments). This dependency is a feature of SPDK that is not abstracted by xNVMe.

Another dependency exists between the application and the maximum I/O size supported by an NVMe SSD. With xNVMe, we recommend that the application gets the maximum transfer size from the underlying devices and use it to bound the size of the I/Os it submits. This way applications can (a) be portable and (b) minimize the number of I/Os submitted when flushing large amounts of data. Note that SPDK simply fails when an I/O is larger than the maximum size supported by a SSD. In contrast, kernel-based I/O interfaces split large I/Os into multiple NVMe requests.

Finally, a comment on the FreeBSD and Windows implementations. Getting xNVMe to work across platforms required careful management of device names, as different operating system follow different naming conventions. Also, the toolchain was an issue on Windows as system-level APIs are only available through the Microsoft SDK, while fio, which we use in the evaluation (see the next section), only compiles with mingw.

4 PERFORMANCE EVALUATION

4.1 Experimental Framework

Table 1 lists the key hardware and software components of our evaluation system. It is comprised of low-cost commodity hardware such that reconstructing the environment and reproducing the experiments is possible with little effort and cost, today and in the near future. The data sheet of the Optane NVME SSD advertises low and predictable I/O latency (~7usec). Sequential read latency should be the same as random-read latency.

Hardware	**Model**
CPU	Intel Core i5-9400 2.90GHz
Memory	Corsair 2x 16GB DDR4 3200Mhz CL18
Board	MSI MPG Z390I GAMING EDGE AC
SSD	Intel Optane Memory M10 Series (MEMPEK1J016GAL)
Software	**Model**
FreeBSD	Version 12.1
Linux	Debian Bullseye / Kernel 5.14
Windows 10	Version 21H1
fio	Version 3.27
gcc	Version 10.2.1
clang	Version 12.0.1
SPDK	Version 21.04
xnvme	Version 0.0.26

Table 1: Experimental Framework System

We use the Flexible I/O tester (fio) to run our experiments and measure latency as well as throughput. Fio provides support for a range of I/O interfaces (denoted I/O engines) including libaio, posixasio, windowsaio, `io_uring`, and `SPDK`. The I/O engines for libaio and `io_uring` are implemented by Jens Axboe, the author of fio and the maintainer of the Linux kernel block layer with contributions from a wealth of authors in the Linux storage community. The SPDK engine is implemented by the core maintainers of SPDK with contributions from authors in the SPDK open source community. We thus consider fio as a reference implementation that represents the state-of-the-art for I/O handling over existing I/O interfaces.

We developed a fio module that relies on `xNVMe` to access the underlying I/O Interfaces. We can thus run the same fio scripts to compare `xNVMe` perfomances with the reference implementation obtained with the native I/O engines.

We use a workload composed of random reads spanning the entire device. This should provide the least interference from device-side behaviour such as caching, media wear, and read-ahead. An extensive comparison of the performance of different SSDs accessed through a range of I/O interfaces on different platforms with various workloads is beyond the scope of this paper.

The native `io_uring` engine can be tuned. Fio makes it possible to turn on or off submission queue thread (sqt) and

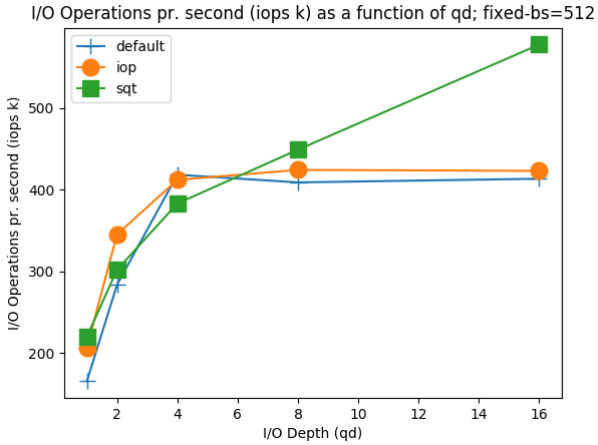

Figure 9: IOPS as a function of queue-depth for fio random-read job using the io_uring engine with default options, submission queue thread (sqt), and I/O completion polling (iop).

I/O completion polling (iop). We compare the throughput obtained when increasing queue depth with a fixed block size of 512B. Figure 9 shows that the default setup provides the best performance at queue depth 4. When queue depth is lower, then I/O completion polling provides best performance as it avoids context switches. For queue depths higher than 4, only submission queue thread provides scalable performance, as the number of system calls remains constant. We use `io_uring` with submission queue thread as a baseline for our experiments.

Another `io_uring` parameter concerns the layout of command payloads in memory. It is possible to (a) allocate contiguous memory or (b) use I/O vectors to scatter input or gather output over multiple non-contiguous buffers. With I/O vectors, it is possible to register buffers that can be reused over multiple I/Os. Our experiments with submission queue polling at queue depth 1 with a block size of 512B, show that using vectored I/O incurs a latency penalty of 75 nsec compared to non-vectored I/O. Applying buffer-registration mitigates the penalty at the cost of significantly increasing cognitive load for the programmer. We do not use I/O vectors in our experiments.

4.2 Experimental Results

Our experiments focus on the performance penalty introduced by `xNVMe`. Let us start by establishing a baseline. To this end, we configure `xNVMe` to use a nil-backend. The nil-backend does everything the actual I/O interface backends do, except that it does not actually submit I/Os to a lower-layer I/O interface but returns immediately. At queue depth

1, with a block size of 4K, we measure an average latency of 90 nsec. The minimum latency reported by fio is 82 nsec, and the maximum is 15844 nsec with standard deviation 74 nsec. In order to calibrate this result, and to confirm that the measured variance originates in fio and not xNVMe, we run the same script using the fio built-in NULL IO engine. I/Os are generated by fio, but they are not submitted. We measure an average latency of 36 nsec, with a minimum latency of 8 nsec, a maximum latency of 17916 and standard deviation of 48 nsec. We conclude that (1) xNVMe does not impact variance, so we consider average latencies in the rest of this section, and (2) the baseline overhead of xNVMe is $90 - 36 = 54$ nsec per I/O, i.e. 0.8% of the advertised I/O latency for the Optane SSD we use in our experiments.

Let us now explore how xNVMe performs when accessing an SSD through the following I/O interfaces: libaio, POSIX aio, Windows IOCP, `io_uring` and SPDK. We consider a queue depth of 1 and a block size of 4KB. Figure 10 shows the result, and Table 2 details the underlying latency results.

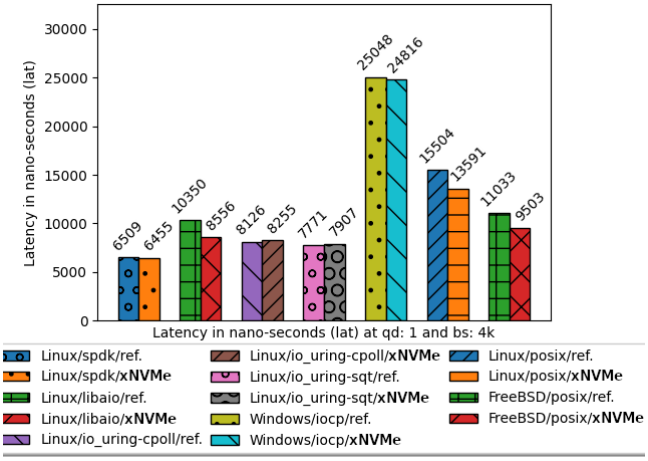

Figure 10: I/O latency when accessing the NVMe SSD through SPDK, `io_uring`, libaio, POSIX aio, and Windows IOCP with or without xNVMe.

Our expectation was to observe the same latency with xNVMe as with the reference implementation plus a penalty, in the form of increased latency, consistent with the baseline above. However, only the `io_uring` comparison is close to matching this expectation. For libaio and SPDK the results tell a different story: latency is better with the xNVMe I/O engine. This seems to defy logic, and raises the questions: how can xNVMe utilizing the SPDK NVMe driver be faster than the SPDK io-engine utilizing the exact same NVMe driver? And similarly how can the difference for libaio be so dramatically different?

fio random read at QD1 and BS 4				
fio I/O Engine	I/O Interface	Latency (nsec)	Std.Dev.	Dist
External SPDK	SPDK/driver	6509	1562	2
External xNVMe	SPDK/driver	6455	1559	10
	Difference	~54 nsec **less**		
Fio built-in	io_uring cpoll	8126	1552	45
External xNVMe	io_uring cpoll	8255	1538	40
	Difference	~129 nsec **more**		
Fio built-in	io_uring sqt	7771	1595	28
External xNVMe	io_uring sqt	7907	1597	31
	Difference	~136 nsec **more**		
Fio built-in	libaio	10350	1517	126
External xNVMe	libaio	8556	1506	70
	Difference	~1794 nsec **less**		
Fio built-in	POSIX(Linux)	15504	1570	113
External xNVMe	POSIX(Linux)	13591	1495	187
	Difference	~1913 nsec **less**		
Fio built-in	POSIX(FreeBSD)	11033	1757	65
External xNVMe	POSIX(FreeBSD)	9503	1503	40
	Difference	~1530 nsec **less**		
Fio built-in	IOCP(Windows)	25048	2778	280
External xNVMe	IOCP(Windows)	24816	1799	106
	Difference	~ **same**		

Table 2: xNVMe vs reference implementations, average latency over three runs of the experiment reported, with recorded standard deviation and the distance in nsec between the highest and lowest of the three runs

In the case of SPDK the reference implementation has features not implemented in the xNVMe engine, such as supporting NVMe Extended LBAs Formats and Protection Information. Due to this, the reference implementation needs to do handle Linux offset and length conversion to LBAs generically as two divisions. To solve the same task xNVMe uses shifts. Additionally, xNVMe calls in slightly lower in the NVMe driver than the reference implementation in fio does, thus skipping a function stack frame. It is these very minor details that shave off the few nanoseconds difference we observe.

For the libaio result, the reason is of a different nature. xNVMe actually uses the libaio I/O interface in a much different way than the reference implementation. Since libaio is interrupt driven, a typical use of the interface would be to submit and wait for a signal before calling io_getevents. xNVMe takes a different approach and spins on io_getevents. For the POSIX aio, the same characteristics apply, the interface is interrupt-driven, and the xNVMe approach is here too to check completion status rather than wait. As the results show, this provides improved latency. But what is not visible in the results is that the xNVMe implementation, unlike the reference implementation, is fully utilizing a core to do this.

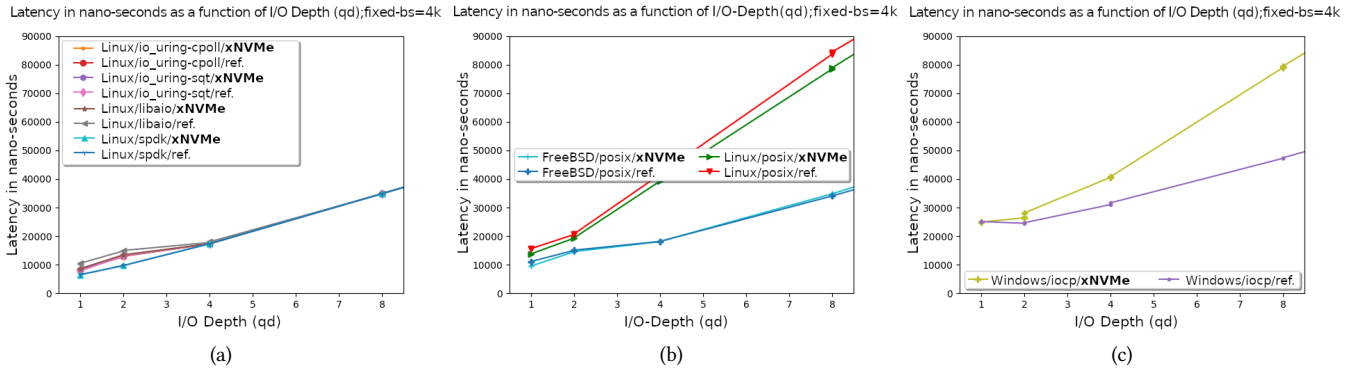

Figure 11: I/O latency with xNVMe vs fio built-in I/O engines as a function of (a) SPDK and Linux specific system interfaces, (b) POSIX aio on Linux and FreeBSD, and (c) IOCP on Windows.

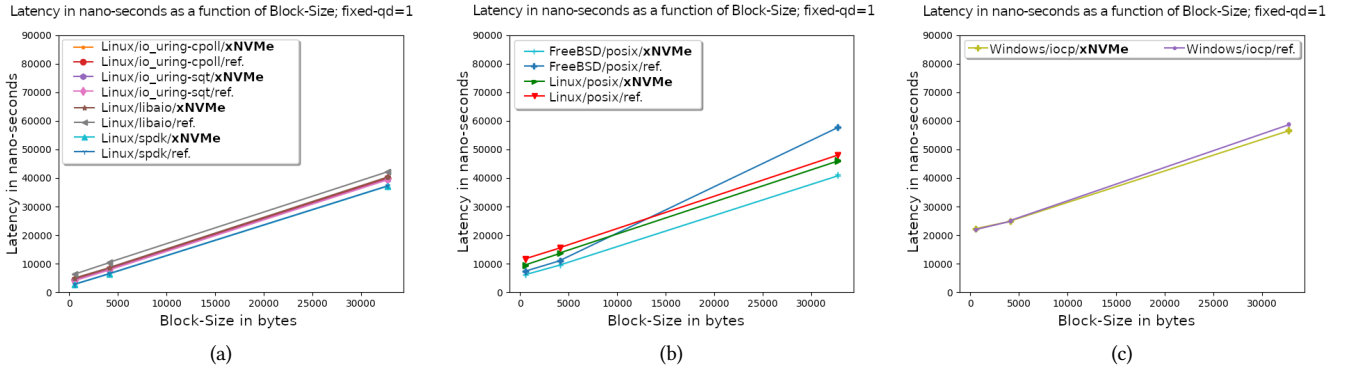

Figure 12: I/O latency with xNVMe vs fio built-in I/O engines as a function of block size. (a) SPDK and Linux specific system interfaces, (b) POSIX aio on Linux and FreeBSD, and (c) IOCP on Windows.

As a sanity check, we now vary queue depth and block size. Figure 11 and Figure 12 show the results. A perfect result would illustrate the xNVMe and reference implementions as lines parallel to each other and thus, that the xNVMe overhead does not degrade with queue depth or block size.

First, we observe that this is the case when varying iodepth, with one exception: the Windows IOCP interface. Although the starting point at iodepth 1 is the same, then the latency increases well beyond the reference implementation at higher iodepths. This has been identified as a shortcoming in implementation which will be addressed in future work.

Second, we observe that scaling block-size, does produce near perfect results for the xNVMe implementations, and thus the xNVMe overhead is constant in all accounts in this regard.

Third, several phenomena are observable in the plots which are not related to the overhead of xNVMe but still worth mentioning. Figure 11(b) shows that FreeBSD implementation of POSIX aio scales significantly better with 4K I/O than the POSIX aio implementation on Linux. This can be attributed to the fact that Linux aio is delegated to a GNU libc thread-emulation implementation whereas FreeBSD provides system calls and a kernel implementation of the interface. Figure 12(b) shows that the FreeBSD implementation of POSIX aio is having challenges scaling as a function of block-size. We observe in Figure 11(a) that when queue depth reaches 4, the cost of the interface is negligible with respect to the time it takes to complete I/Os on the device. There are no longer any performance differences across I/O interfaces.

5 RELATED WORK

The ADIOS2 library [6] provides a uniform I/O framework to supercomputer applications across different storage media (e.g., file, wide-area-network, in-memory staging, etc.). Both ADIOS2 and xNVMe aim to decouple applications from the underlying storage interface. While xNVMe focuses on NVMe, ADIOS2 supports a wide range of storage media. Conversely,

xNVMe makes no assumption about the nature of the data being stored, while ADIOS2 focuses on self-describing data variables and attributes, which are most relevant for scientific applications on supercomputers.

The introduction of ZNS [16] has led to increased focus on the impact of NVMe on data-intensive systems. In their NVMe webcast focused on the Linux ZNS ecosystem, Gonzalez and Demoal present two complementary efforts [7]: (1) introducing ZNS to legacy applications via POSIX I/O (e.g., via f2fs [3, 15] or zonefs [5]), (2) supporting data-intensive systems that access ZNS SSDs with minimal interferences. Our work contributes to the latter.

In addition to SPDK, a range of libraries are available to directly access ZNS SSDs via the Linux kernel. Libnvme [10] provides an API for NVMe device enumeration. Libzbd [13] provides an API for simplifying the access to ZNS devices. With libnvme and libzbd, I/Os are actually submitted with unistd, libaio or io_uring. For instance, ZenFS [8] implements a ZNS-aware backend for RocksDB on top of libzbd and unistd. In contrast, xNVMe provides a single API for managing and accessing ZNS devices over different I/O interfaces. All three libraries support the definition of shell commands.

Intel's oneAPI[5] is a uniform programming model over accelerator devices. OneAPI's design follows the hourglass model with Level Zero as its narrow waist. Despite the obvious difference between compute and I/Os, the design of Level Zero and xNVMe present many similarities in terms of device management, command contexts, command queues and memory management. These similarities are a confirmation that xNVMe is well designed to handle NVMe computational storage.

6 CONCLUSION

The advent of high-performance NVMe SSD has triggered a profound redesign of the storage stack. Data-intensive systems need to access NVMe SSDs directly, without interferences from the operating system. We introduced xNVMe, a user-space library that provides a uniform API for device management, memory management and I/O submission over existing I/O interfaces. By construction, xNVMe is the narrow waist of the NVMe storage stack. We showed that it provides programmer with a simple and extensible framework at negligible performance penalty.

ACKNOWLEDGEMENTS

We would like to thank the reviewers and our shepherd, Abutalib Aghayev, for their help improving the paper. Philippe Bonnet received funding from the European Union's Horizon 2020 Research and Innovation Programme (grant agreement No 957407/Daphne).

[5]https://www.oneapi.io/

REFERENCES

[1] Jens Axboe. 2019. *Efficient IO with io_uring.* Linux Kernel Report. https://kernel.dk/io_uring.pdf
[2] Micah Beck. 2019. On the Hourglass Model. *Communicatons of the ACM* 62, 7 (2019).
[3] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *USENIX Annual Technical Conference.*
[4] Adrian Caulfield and Steven Swanson. 2013. Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage. *IEEE Computer* 46 (2013).
[5] Damien Le Moal and Ting Yao. 2020. zonefs: Mapping POSIX File System Interface to Raw Zoned Block Device Accesses. In *Usenix Vault.* https://www.usenix.org/conference/vault20/presentation/lemoal.
[6] William F. Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip E. Davis, Jong Choi, Kai Germaschewski, Kevin A. Huck, Axel Huebl, Mark Kim, James Kress, Tahsin M. Kurç, Qing Liu, Jeremy Logan, Kshitij Mehta, George Ostrouchov, Manish Parashar, Franz Poeschel, David Pugmire, Eric Suchyta, Keichi Takahashi, Nick Thompson, Seiji Tsutsumi, Lipeng Wan, Matthew Wolf, Kesheng Wu, and Scott Klasky. 2020. ADIOS 2: The Adaptable Input Output System. A framework for high-performance data management. *SoftwareX* 12 (2020).
[7] Javier Gonzalez and Damien Lemoal. 2020. *NVMe Zoned Namespace SSDs and the Linux Zoned Storage Ecosystem.* https://www.youtube.com/watch?v=lcYdE_S5o8Q&t=531s.
[8] Hans Holmberg. 2020. In *SNIA Storage Developer Conference.* https://snia.org/sites/default/files/SDC/2020/074-Holmberg-ZenFS-Zones-and-RocksDB.pdf.
[9] Kanchan Joshi, Javier Gonzalez, and Simon Lund. 2021. Enabling Asynchronous I/O Passthru in NVMe-Native Applications. In *Storage Developer Conference 2021.* https://www.snia.org/educational-library/enabling-asynchronous-i-o-passthru-nvme-native-applications-2021.
[10] Keith Busch. 2020. Linux User Library for NVM Express. In *Usenix Vault.* https://www.usenix.org/conference/vault20/presentation/busch.
[11] Avi Kivity. 2016 (accessed April 2022). *Qualifying Filesystems for Seastar and ScyllaDB.* https://www.scylladb.com/2016/02/09/qualifying-filesystems/.
[12] Avi Kivity. 2017 (accessed April 2022). *Different I/O Access Methods for Linux, What We Chose for ScyllaDB, and Why.* https://www.scylladb.com/2017/10/05/io-access-methods-scylla/.
[13] Damien Lemoal. 2020 (accessed April 2022). *libzbd.* https://github.com/westerndigitalcorporation/libzbd.
[14] Alberto Lerner and Philippe Bonnet. 2021. Not Your Grandpa's SSD: The Era of Co-Designed Storage Devices. In *Proceedings of the 2021 International Conference on Management of Data.*
[15] Naohiro Aota. 2020. File System Support for Zoned Block Devices. In *Usenix Vault.* https://www.usenix.org/conference/vault20/presentation/aota.
[16] NVMe Standard. 2020. *NVME TP 4053 - Zoned Namespaces.* https://nvmexpress.org/wp-content/uploads/NVM-Express-1.4-Ratified-TPs-1.zip
[17] John Ousterhout. 2018. *A Philosophy of Software Design.* Yaknyam Press.
[18] Simon Lund. 2020. Programming Emerging Storage Interfaces. In *Usenix Vault.* https://www.usenix.org/conference/vault20/presentation/lund.

[19] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul. 2017. SPDK: A Development Kit to Build High Performance Storage Applications. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 154–161. https://doi.org/10.1109/CloudCom.2017.14 ISSN: 2330-2186.