SPDK: A development kit to build high performance storage applications

Ziye Yang, James R Harris, Benjamin Walker, Daniel Verkamp, Changpeng Liu,
Cunyin Chang, Gang Cao, Jonathan Stern, Vishal Verma, Luse E Paul
Intel
{ziye.yang, james.r.harris, benjamin.walker, daniel.verkamp, changpeng.liu,
cunyin.chang, gang.cao, jonathan.stern, vishal4.verma, paul.e.luse}@intel.com

Abstract

There is strong demand on building high performance storage service upon emerging fast storage devices (e.g., NVMe SSDs). Unfortunately, current storage software stack cannot satisfy such requirements and the software overhead becomes a major bottleneck for developing high performance storage applications.

According to our performance profiling results, most storage software overhead is caused by kernel I/O stacks due to context switch, data copy, interrupt, resource synchronization and etc. To address these issues, we provide SPDK (storage performance development kit), a set of tools and libraries for writing high performance, scalable, user-mode storage applications. It achieves high performance by moving the necessary drivers into user space and operating them in a polled mode instead of interrupt mode, which eliminates kernel context switch and interrupt handling overhead and also provides lockless resource access. Integrated with SPDK, performance competitive storage applications can be build upon those fast storage devices. In our experiments, the per cpu core IOPS of NVMe device driver in SPDK is about 6X to 10X better than the kernel NVMe driver. Some storage applications are also developed upon SPDK's framework and obtain much better per-core performance compared with the existing solutions. For example, SPDK NVMe over fabrics (NVMe-oF) target is 10X more efficient than Linux Kernel NVMe-oF target solution.

Keywords

User space NVMe driver; asynchronous polled mode driven I/O; High performance storage application;

1. Introduction

Nowadays, many companies are renting IT infrastructures from cloud providers instead of building their own due to many benefits including easy management, high scalability, low cost and etc. Among those service, cloud storage is one of the significant ones which cannot be ignored by the tenants. And there is strong demand for cloud storage providers to build high quality storage service to meet the requirements from frontend applications, e.g., providing high throughput/IOPS and low latency but consuming controllable resources.

As is known, the quality of a storage service is determined by both software and hardware, thus cloud providers have to frequently refresh those components in those two categories. In the era with low speed storage devices, the performance of a storage service is mainly determined by the hardware components (e.g., hardware disk drives) since the software overhead is trivial compared with the hardware. So there is no strong motivation to upgrade the existing software

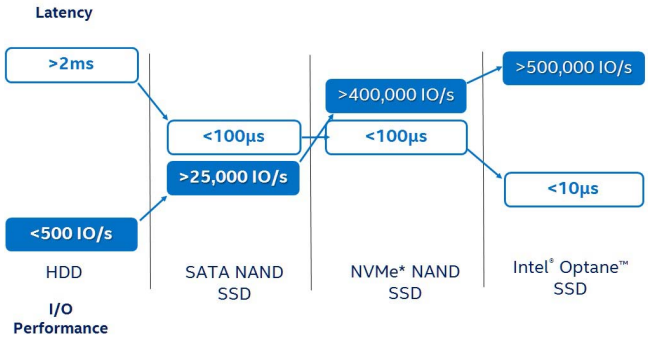

Fig. 1. I/O performance and latency of Hard drives

architecture. With the emergence of fast storage device drives (e.g., NVMe SSDs), such situation is changed. The software overhead is becoming a major bottleneck for improving the performance of storage applications since the latency of single I/O operation consumed by the hardware drives are reduced into several microseconds. Figure 1 shows the latency and IOPS distribution of different hardware drives. Historically, storage media access was much slower than CPU execution and memory access. But currently the latency of 3D Xpoint [1] media is approaching DRAM latencies gradually, which blurs the boundary between storage media and CPU/memory access latency. With this trend, storage software stacks become the obstacle for storage service performance improvement.

According to our analysis on the storage software stacks, kernel software I/O stack occupies large amount of execution time, which includes the following overhead, e.g., context switch, data copy between kernel and user space, interrupt, shared resource competition in I/O stack and etc. To eliminate the kernel I/O stack overhead, SPDK (Storage Performance Development Kit) [2, 3] is invented, which provides a set of user space libraries and tools for designing and implementing high performance scalable storage applications. Currently, SPDK provides user space NVMe driver and IOAT driver with more significant performance improvement than existing ones. For example, the per cpu core IOPS of SPDK NVMe device driver is about 6X to 10X better than the kernel NVMe driver. Moreover, some customized storage applications(e.g., NVMe-oF target) are also provided in SPDK. For example, SPDK

2330-2186/17 $31.00 © 2017 IEEE
DOI 10.1109/CloudCom.2017.14

NVMe-oF target is 10X more efficient than the Linux kernel NVMe-oF Target while both attached with same fast storage devices (e.g., NVMe SSDs).

To summarize, following contributions are made in this paper:

- We propose SPDK, the libraries of which can be leveraged by OEMs (Original Equipment manufacturer) and cloud storage service providers to build high performance storage applications or optimize their existing storage applications.
- We present the detailed mechanism inside SPDK and state why SPDK can improve the performance.
- We demonstrate the performance of storage applications built on SPDK are better than the existing solutions, e.g., SPDK NVMe-oF target/initiator is much efficient than Linux kernel NVMe-oF target/initiator.

The remainder of this paper is organized as follows. Section 2 presents the design and implementation of SPDK library. Section 3 provides the performance evaluation on SPDK user space NVMe driver. And Section 4 gives a case study on SPDK NVMe-oF target and initiator. Then section 5 discusses some related work. Finally, we conclude this paper in Section 6.

2. System Design and implementation

SPDK is a collection of libraries which can be used to build customized high performance storage applications. There are mainly four main components (shown in Figure 2)in SPDK, i.e., drivers, app scheduling, storage devices and storage protocols.

The **app scheduling** component provides an application event framework for writing asynchronous, polled-mode, shared-nothing server applications by leveraging SPDK's libraries, more details are described in Section 2.1; The foundation of SPDK is the **drivers** component, SPDK has a user space polled mode NVMe driver which provides zero-copy, highly parallel and direct access to NVMe SSDs for user space applications. Similarly, SPDK provides a user space polled driver to operate I/OAT DMA engine present on many Intel Xeon-based platforms with all of the same properties as the NVMe driver; the **storage devices** layer abstracts the device exported by drivers and provides the user space block I/O interface to storage applications above. SPDK can export blockdevs (i.e., bdevs) constructed by user space NVMe drivers, Linux asynchronous I/O (libaio), Ceph rados block device API and etc. Thus storage applications can avoid directly working on low level drivers. Also in this layer, we provide blobstore/blobfs, which is aimed to provide user space file I/O interface to applications. Currently, blobstore/blobfs can be integrated with rocksdb [4]; the **storage protocols** layer contains the accelerated applications implemented upon SPDK framework to support various different storage protocols, e.g., iSCSI target for iSCSI service acceleration, vhost-scsi/blk targets for accelerating virtio-scsi/blk in VMs and NVMe-oF target for accelerating NVMe commands over fabrics.

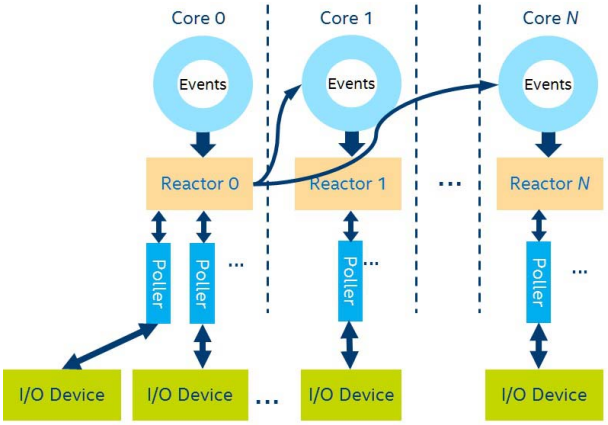

Fig. 3. SPDK application event framework

2.1. App scheduling

Simple server applications can be written in a single-threaded fashion. Thus straightforward code can maintain states without any locking or other synchronization. However, for scaling up (e.g., to allow more simultaneous connections), the application may need to use multiple threads. In the ideal case where each connection is independent from all other connections, the application can be scaled by creating additional threads and assigning connections to them without introducing cross-thread synchronization. Unfortunately, in many real-world cases, the connections are not entirely independent and cross-thread shared state is necessary. Thus SPDK provides an event framework to address this issue.

The event framework shown in Figure 3 is provided to write asynchronous, polled-mode, shared-nothing server applications. This framework is intended to be optional, and most other SPDK components are designed to be integrated into an application without specifically depending on this library. The framework defines several concepts, i.e., **reactors**, **events**, and **pollers** and it spawns one thread per core (i.e., reactor) and connects the threads with lockless queues. Messages (i.e., events) can then be passed between the threads. On modern CPU architectures, message passing is often much faster than traditional locking.

Events. To accomplish cross-thread communication while minimizing synchronization overhead, the framework provides message passing in the form of events. The event framework runs one event loop thread per CPU core. These threads are called reactors, and their main responsibility is to process incoming events from a queue. Each event consists of a bundled function pointer and its arguments, destined for a particular CPU core. Events are created by using spdk_event_allocate() and executed by using spdk_event_call(). Unlike a thread-per-connection server design, which achieves concurrency by depending on the operating system to schedule many threads issuing blocking I/O onto a limited number of cores, the event-driven model requires use of explicitly asynchronous

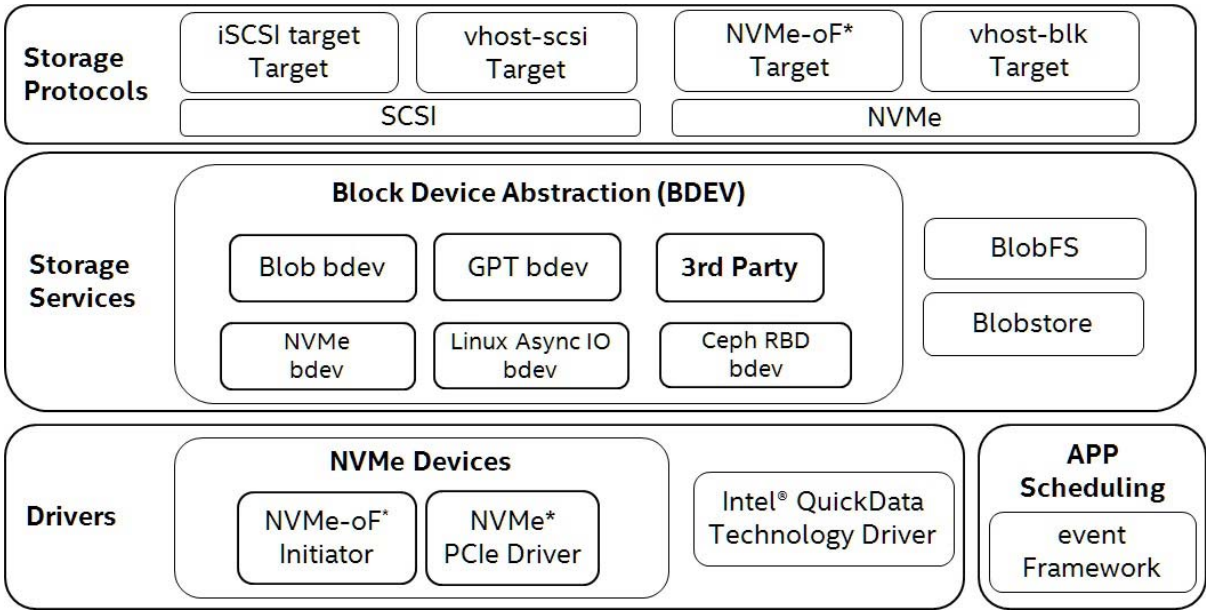

Fig. 2. Main components in SPDK

operations to achieve concurrency. Asynchronous I/O may be issued with a non-blocking function call, and completion is typically signaled using a callback function.

Reactors. Each reactor has a lock-free queue for incoming events to that core, and threads from any core may insert events into the queue of any other core. The reactor loop running on each core checks for incoming events and executes them in first-in, first-out order as they are received. Event functions should never block and should preferably execute very quickly, since they are called directly from the event loop on the destination core.

Pollers. The framework also defines another type of function called a poller. Pollers may be registered with the spdk_poller_register() function. Pollers, like events, are functions with arguments that can be bundled and sent to a specific core to be executed. However, unlike events, pollers are executed repeatedly until unregistered. The reactor event loop intersperses calls to the pollers with other event processing. Pollers are intended to poll hardware as a replacement for interrupts. Normally, pollers are executed on every iteration of the main event loop. Pollers may also be scheduled to execute periodically on a timer if low latency is not required.

2.2. User space polled drivers

SPDK moves the device driver implementation in user space instead of kernel space. The benefit is that the substantial overhead of system calls, data copies between kernel and user space can be eliminated. Regardless of other storage features in user space, user space drivers can waste fewer CPU cycles than kernel to complete corresponding read and write I/O operations. Moreover, SPDK uses polling [5] instead of interrupt. Polling allows the software to save the expense of invoking the kernel interrupt handler and reduce the context switching. It also allows users to determine how much CPU time for each task instead of letting the kernel scheduler decide, thus the software thread can be always available when new I/O arrives.

Asynchronous I/O mode. SPDK uses the asynchronous read and write I/O mode to complete the I/O requests. Applications can call SPDK's asynchronous read/write interface to send I/O requests, then use the corresponding I/O completion check functions to pull the completed I/Os. When I/O queue depth is larger than 1, such methodology is more efficient than the synchronous mode (i.e., sending one I/O request and immediately waiting for the I/O completion). Generally, it reduces the average I/O latency and increases the IOPS.

Lockless architecture. Traditionally, shared resources need to be accessed inside the kernel I/O stack while completing an I/O request. For example, there exists racing for NVMe I/O queues among different CPU cores in kernel NVMe driver. To eliminate the overhead caused by resource sharing, SPDK adopts a lockless architecture which requires each thread to access its own resources, e.g., memory, I/O submission queue, I/O completion queue and etc. Thus the I/O performance can be greatly improved.

2.3. Storage services

Storage services contains SPDK blobfs/blobstore and user space block device (BDEV)layer.

Blobstore. The blobstore is a persistent, power-fail safe block allocator designed to be used as the local storage system backing a higher level storage service, typically in

lieu of a traditional filesystem. These higher level services can be key/value stores(e.g., Rocksdb) or local databases(Mysql), dedicated appliances (SAN, NAS), or distributed storage systems (e.g., Cassandra). It is not designed to be a general purpose filesystem and it is intentionally not POSIX compliant. To avoid confusion, no reference to files or objects will be made at all, instead using the term 'blob'. The blobstore is designed to allow asynchronous, uncached, parallel reads and writes to groups of blocks on a block device called 'blobs'. Blobs are typically large, measured in at least hundreds of kilobytes, and are always a multiple of the underlying block size.

Blobfs. The blobfs is a very simple filesystem built on blobstore. BlobFS currently supports only a flat namespace for files with no directory support. Filenames are currently stored as xattrs in each blob. This means that filename lookup is an O(n) operation. An SPDK btree implementation is underway which will be the underpinning for BlobFS directory support in a future release. Writes to a file must always append to the end of the file. Support for writes to any location within the file will be added in a future release. Currently, BlobFS is only tested with Rocksdb.

BDEV. The purpose of SPDK bdev is to abstract the device identified by user space drivers (e.g., NVMe) and other third part libraries to export the block service interface to applications. Block storage consumed by SPDK applications is provided by the SPDK bdev layer. Currently, SPDK bdev consists of a driver module API for implementing bdev drivers which enumerate and claim SPDK block devices and performance operations (read, write, unmap, etc.) on those devices; bdev drivers for NVMe, Linux AIO and Ceph RBD, blobdev (i.e., blobstore as bdev), Gpt bdev (i.e., construct bdevs upon existing bdevs via detecting GPT partitions); Configuration interface via SPDK configuration files or JSON RPC.

3. User space NVMe driver implementation and evaluation

In this section, implementation details and evaluation of user space NVMe driver are provided. Also we focus on demonstrating the per cpu core performance of SPDK user space NVMe driver. The experiments are conducted on two types of SSDs, i.e., Intel 2D NAND NVMe SSDs (DC P3700) and Intel 3D Xpoint NVMe SSDs (DC P4800X). And we compare the performance between SPDK NVMe driver and Linux Kernel NVMe driver.

3.1. Implemention details of user space NVMe driver

SPDK implements a user space asynchronous polled NVMe driver, this library runs in Linux user space. The only requirement is to unload Linux kernel NVMe driver and bind the specified NVMe device to UIO or VFIO driver in Linux kernel. And this library controls NVMe devices by directly mapping the PCI BAR into the local process and performing MMIO. I/O is submitted asynchronously via queue pairs. Table 1 shows the detailed public API functions implemented in SPDK.

Asynchronous I/O. I/O is submitted to an NVMe namespace using nvme_ns_cmd_xxx functions. The NVMe driver submits the I/O request as an NVMe submission queue entry on the queue pair specified in the command. The function returns immediately, prior to the completion of the command. The application must poll for I/O completion on each queue pair with outstanding I/O to receive completion callbacks by calling spdk_nvme_qpair_process_completions().

Lockless queue qpair. Since NVMe queue pairs (struct spdk_nvme_qpair) provide parallel submission paths for I/O, thus I/O may be submitted on multiple queue pairs simultaneously from different threads. Queue pairs implemented in SPDK library contain no locks or atomics, so a given queue pair may only be used by a single thread at a time. This requirement is not enforced by the NVMe driver (doing so would require a lock), and violating this requirement results in undefined behavior.

Number of queue qpair restriction. The number of queue pairs allowed is dictated by the NVMe SSD itself. The specification allows for thousands, but most devices support between 32 and 128. The specification makes no guarantees about the performance available from each queue pair, but in practice the full performance of a device is almost always achievable using just one queue pair. For example, if a device claims to be capable of 450,000 I/O per second at queue depth (QD)=128, in practice it does not matter if the driver is using 4 queue pairs each with QD=32, or a single queue pair with QD=128. Given the above, the easiest threading model for an application using SPDK is to spawn a fixed number of threads which is pinned to separate CPU core in a pool and dedicate a single NVMe queue pair to each thread.

The NVMe driver takes no locks in the I/O path, so it scales linearly in terms of performance per thread as long as a queue pair and a CPU core are dedicated to each new thread. In order to take full advantage of this scaling, applications should consider organizing their internal data structures such that data is assigned exclusively to a single thread, and SPDK's application event framework can be used.

3.2. Performance evaluation on NAND NVMe SSDs

Table 2 shows our experiment configuration for testing 2D NAND NVMe SSDs on a single Xeon server host. In our experiment, we use the performance comparison tool (i.e., nvme perf [6]) in SPDK and run this tool with a single CPU core upon either Kernel NVMe driver or user space NVMe driver in SPDK.

TABLE 2. Configuration on testing NAND SSDs

Host configuration	
CPU	2x Xeon E5-2695v4
Memory	64GB DDR4 Memory, 8x 8GB DDR4 2133 MT/s
Hard drive	8 Intel's NVMe P3700 800G SSDs
OS info	CentOS 7.2 (kernel version 4.10.0)

TABLE 1. Public API of SPDK user space NVMe driver

Key Functions	Decriptions
spdk_nvme_probe()	Attach the userspace NVMe driver to found NVMe device.
spdk_nvme_ctrlr_alloc_io_qpair()	Allocate an I/O queue pair (submission and completion queue).
spdk_nvme_ctrlr_get_ns()	Get a handle to a namespace for the given controller.
spdk_nvme_ns_cmd_read()	Submit a read I/O to the specified NVMe namespace.
spdk_nvme_ns_cmd_write()	Submit a data set management request to the specified NVMe namespace.
spdk_nvme_ns_cmd_dataset_management()	Submit a data set management request to the specified NVMe namespace.
spdk_nvme_ns_cmd_flush()	Submit a flush request to the specified NVMe namespace.
spdk_nvme_qpair_process_completions()	Process any outstanding completions for I/O submitted on a queue pair.
spdk_nvme_ctrlr_cmd_admin_raw()	Send the given admin command to the NVMe controller.
spdk_nvme_ctrlr_process_admin_completions()	Process any outstanding completions for admin commands.

Figure 4 shows the performance testing results, the nvme perf tool [6] utilizes one core and generated 4KB random read I/O on 1, 2 and 8 NVMe SSDs with QD=128. We can see that SPDK user space NVMe driver linearly increases the performance by driving NVMe SSDs from 1 to 8, however there is no performance gain for Kernel NVMe driver while driving more devices. It clearly demonstrates user space NVMe driver can accomplish more NVMe I/Os with same CPU resource. As is well known, the maximal 4KB random read IOPS of a single Intel P3700 800G SSD is 450K IOPS, and our SPDK user space NVMe driver can easily achieve the hardware performance limitation of 8 NVMe SSDs (i.e., 3600K IOPS) via only one CPU core. In details, The performance gain by SPDK user space NVMe driver is due to:

- **Asynchronous Polling mode benefit**. For high frequency read and write I/Os, polling driven I/O is much more efficient than interrupt driven I/O.
- **No system call and data copy overhead**. There is no data copy and context switch (caused by system call) between kernel and user space for completing read and write I/O operations.
- **I/O stack reducing**. For completing an I/O request, the total function call number of user space NVMe driver is much less than kernel NVMe driver.
- **Lockless architecture**. User space NVMe driver can bind the NVMe I/O queues into independent threads, thus each thread can operate on dedicated I/O queues which eliminates the resource competition such as synchronization.

Figure 5 shows the single I/O operation execution time for 4KB random read. We divide the total time into two parts, i.e., submit and complete. From the diagram, it shows that SPDK user space NVMe driver reduces the both I/O submit and complete time compared with Linux kernel NVMe driver.

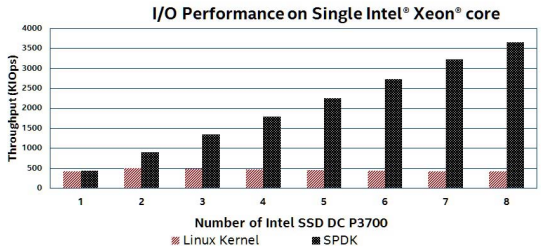

Fig. 4. Single core 4KB Random Read performance comparison

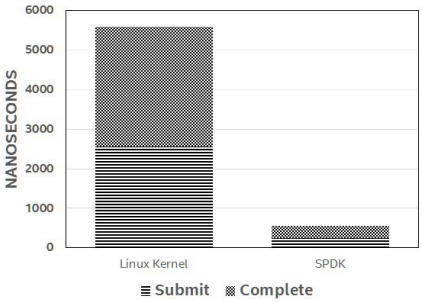

Fig. 5. I/O submit and complete time comparison

3.3. Performance evaluation on 3D Xpoint SSDs

Table 3 shows our experiment configuration for testing 3D Xpoint NVMe SSDs on a single Xeon server host. In our experiment, we use FIO-2.18 to conduct performance testing with the following configurations, i.e., 4KB random read, with queue depth from 1 to 32, numjobs=1, direct mode. For testing SPDK, we provide plugin into FIO thus FIO can be directly tested on NVMe SSDs bound on spdk user space NVMe driver.

TABLE 3. Configuration on testing 3D Xpoint SSDs

Host configuration	
CPU	2x Xeon E5-2695v4
Memory	64GB DDR4 Memory, 8x 8GB DDR4 2133 MT/s
Hard drive	1 Intel's NVMe P4800X 375G SSD
OS info	Ubuntu 16.04.1 (kernel version 4.10.1)

Figure 6 shows the IOPS and latency of both SPDK NVMe driver and the Kernel NVMe driver at different QDs. We limit both drivers to use just one CPU core on the test system. SPDK has higher IOPS at all QDs and saturates the SSD at QD=8. The kernel NVMe driver exhausts all the CPU cycles on a single thread beyond QD=8. And the kernel driver reaches CPU saturation on 1 core and needs another core to reach

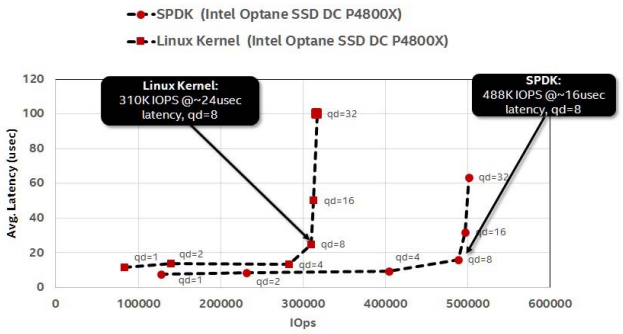

Fig. 6. Single core 4KB Random Read Performance comparison on Intel Optane SSD

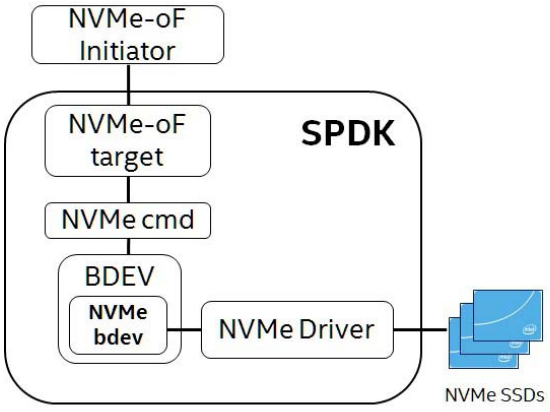

Fig. 7. General architecture of SPDK NVMe-oF target

device saturation. Also the latency of SPDK NVMe driver is better than Kernel at different QDs. Generally, SPDK NVMe driver is highly efficient, it uses few CPU cycles to submit and complete I/Os with good performance in both IOPS and latency aspects. So the per cpu core performance of SPDK NVMe driver is much better than Linux Kernel NVMe driver, which can save more CPU cycles for other tasks.

4. Case study: SPDK NVMe-oF target/initiator

SPDK NVMe-oF target is a user space application that presents block devices over the network using RDMA. It requires an RDMA-capable NIC with its corresponding OFED software package installed to run. The target should work on all flavors of RDMA, but it is currently tested against Mellanox NICs (RoCEv2) and Chelsio NICs (iWARP). And we also have NVMe-oF initiator inside user space NVMe driver which supports connecting to remote NVMe-oF targets and interacting with them in the same manner as local NVMe SSDs. Figure 7 shows the general architecture of SPDK NVMe-oF target, the NVMe-oF target is composed of NVMf library, NVMe block device layer (i.e., NVMe bdev) and NVMe user space driver. The NVMf library is designed to receive the request, extract the NVMe command, send it to the NVMe bdev and finally returns the results to NVMe-oF initiator via asynchronous function calls. To verify the efficiency of SPDK NVMe-oF target/initiator, we compare the performance with Linux kernel target/initiator on different NVMe SSDs.

4.1. Performance evaluation on NAND SSDs

Table 4 shows the experimental configuration of both target and initiator hosts, each initiator is directly connected to the target, and the target exports a 12 NAND SSDs (Intel DC P3700) for test. We use FIO [7] as benchmark to test with the following configurations, e.g., 4KB Random read I/O, 2 RDMA queue pair per remote SSD, numjobs=4 with QD=32 per job on each SSD.

TABLE 4. Experimental configuration on NAND SSDs

Target configuration	
CPU	2x Intel Xeon E5-2695v4
Memory	64GB DDR4 memory, 8x 8GB DDR4 2400 MT/s
OS info	Ubuntu 16.04.1, Linux kernel 4.10.1
NIC info	3x 25GbE Mellanox with two ports
SSDs	12 Intel's NVMe DC P3700 800G SSDs
Initiator configuration	
CPU	2x Intel Xeon E5-2695v4
Memory	64GB DDR4 memory, 8x 8GB DDR4 2400 MT/s
OS info	Fedora 25, Linux kernel 4.9.11
NIC info	25GbE Mellanox with two ports

Figure 8 shows the efficiency comparison between SPDK and Linux Kernel NVMe-oF targets. For both SPDK and kernel, they can achieve the line rate of 3 NICs (i.e., 150 Gb/s), however SPDK only requires 3 CPU cores compared to the 30 CPU cores used by Linux kernel. The benefit is gained by:

- **Utilizing SPDK NVMe driver**. SPDK user space NVMe driver is much more efficient than related Linux Kernel driver.
- **RDMA Queue Pair Polling**. We directly use RDMA

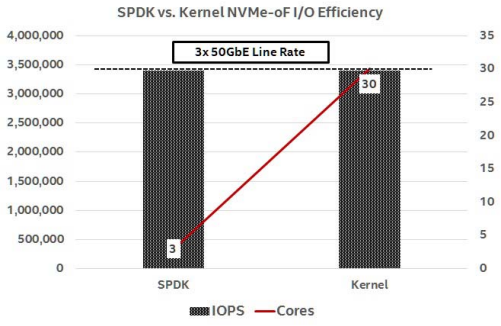

Fig. 8. Efficiency comparison between SPDK and Kernel NVMe-oF targets

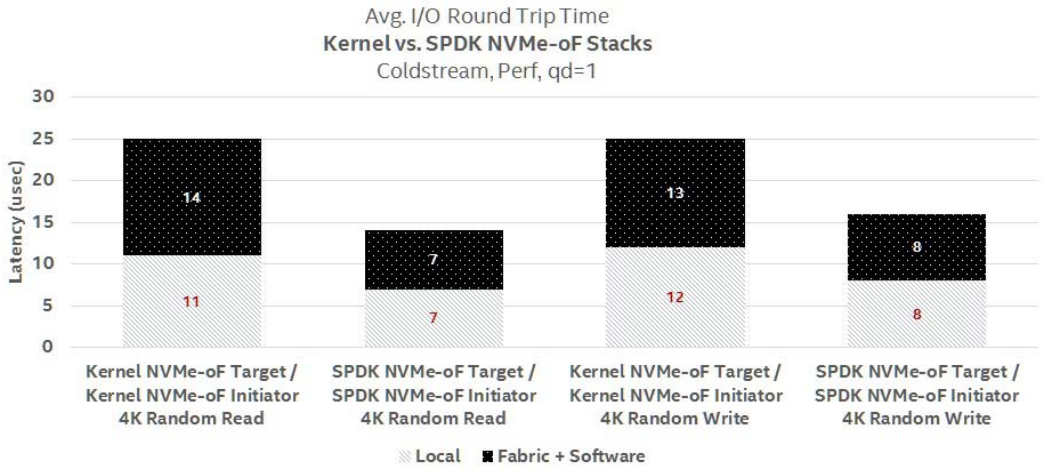

Fig. 9. Latency comparison between SPDK and Kernel NVMe-oF target/initiator

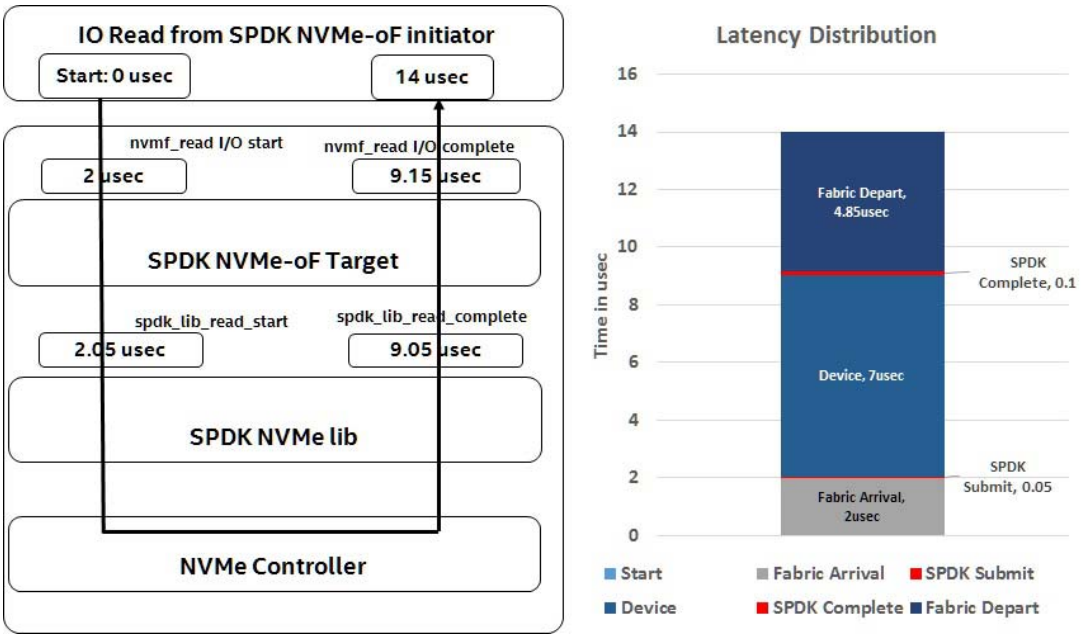

Fig. 10. Latency analysis of SPDK NVMe-oF target and initiator

queue pair polling which eliminates the interrupt overhead.

- **Lockless connection handling**. Each connection from NVMe-oF initiator is pinned to dedicated CPU cores to handle via SPDK app event framework, thus the synchronization overhead among threads are eliminated.

4.2. Performance evaluation on 3D Xpoint SSDs

Table 5 shows the experimental configuration of both target and initiator hosts, they are directly connected, and the target exports a 3D Xpoint SSD (Intel optane DC P4800X) for test.

TABLE 5. Experimental configuration on Intel P4800X SSDs

Target and initiator configuration	
CPU	2x Intel Xeon E5-2695v4
Memory	64GB DDR4 memory, 8x 8GB DDR4 2400 MT/s
OS info	Ubuntu 16.04.1, Linux kernel 4.10.1
NIC info	1x 25GbE Mellanox with two ports
SSD on target	1 Intel's NVMe DC P4800X 375G SSDs

We use nvme perf [6] in initiator host while testing both SPDK and kernel NVMe-oF target.

Figure 9 shows the latency comparison between kernel and

SPDK NVMe-oF target/initiator on both 4KB random read and write with QD=1. We divide the average round trip time into two parts, e.g., Local and Fabric part. Local part means that the local I/O latency (via either the SPDK NVMe driver or kernel driver) on target host; Fabric part means the time consumed on the fabric (via either SPDK and Kernel NVMe-oF initiator). We can see that SPDK hits an average of 7 usec of latency on read and 8 usec on write workload for local I/O part, and consumes 7 usec of latency on read and 8 usec on write workload on Fabric part. Based on the total time comparison, SPDK reduces the average NVMe-oF latency on DC P4800 NVMe SSD by 44% for read workload and 36% for write workload.

Figure 10 shows the detailed latency distribution for the read operation. The total latency (14 usec) can be divided into five parts, i.e., Fabric arrival, SPDK I/O submit, NVMe I/O executed in device, SPDK I/O complete, Fabric departure. From the diagram, we can see that **SPDK submit** only consumes 50 nsec, and **SPDK complete** consumes 100 nsec. The software overhead is very minimal which proves the effectiveness of SPDK library design.

5. Related work

Storage I/O stack optimization. There are many works to optimize storage I/O stacks for fast storage devices(e.g., SSDs) in both kernel and user spaces. For example in kernel space, Bjørling et al. [8] provide multiple queues for multi-cores to improve the I/O performance on SSD; Shin et al. [9] propose to optimize the I/O paths for SSD; Yu et al. [10] put efforts on I/O subsystem optimization. In summary, most kinds of the work are invented to address the existing kernel I/O stack design and implementation issues. In user space, Yang et al. [5] proves the efficiency of polling mode for frequent I/Os on fast storage devices(e.g., NVMe SSDs); and there are also some works to implement user space file systems [11] on fast storage devices for performance optimization. Compared with those storage I/O optimization works, SPDK optimizes the storage I/O for fast storage devices in user space with the polling mode, which has no dependency on kernel space I/O stack. Moreover, the NVMe library in SPDK can be utilized to construct user space file system (e.g., blobfs) by eliminating using the interface provided by fuse [12]. Currently there also exists user space I/O framework [13] upon both SPDK and kernel for NVMe I/O optimization.

Relationship with DPDK. Compared with DPDK, SPDK uses the similar user space polled mode idea[14], e.g., there is user space polled mode NIC driver in DPDK. Also SPDK can use the EAL (environment abstraction layer) in DPDK to abstract CPU, memory and other hardware resources. However, SPDK has its own environment API to invoke DPDK, it means that we can use SPDK without DPDK if the developers implement their own environment according to SPDK's environment API.

6. Conclusion

This paper presents SPDK, a set of libraries and tools for designing and implementing high performance scalable storage applications. Based on SPDK, OEMs and cloud storage providers can optimize their own storage solutions.

To verify the practicality and feasibility of SPDK, several experiments were conducted on standard server platforms with fast storage devices, i.e., NVMe SSDs. According to the experimental results, the per cpu core performance (IOPS/core) of user space NVMe driver is about 6X to 10X better than kernel space NVMe driver. Moreover, SPDK user space NVMe-oF target is 10X more efficient than Linux kernel NVMe-oF target. Nowadays, more and more companies are starting to evaluate and even use SPDK, which demonstrates the value of SPDK in real product environments. And we will continue developing SPDK to provide more functionalities and friendly API to storage software developers.

References

[1] 3d xpoint. https://en.wikipedia.org/wiki/3D_XPoint.

[2] Storage performance development kit. http://www.spdk.io/.

[3] Spdk github. https://github.com/spdk/spdk.

[4] Rocksdb. https://github.com/facebook/rocksdb.

[5] Jisoo Yang, Dave B. Minturn, and Frank Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*.

[6] Nvme perf. https://github.com/spdk/spdk/tree/master/examples/nvme/perf.

[7] Flexible i/o tester. https://github.com/axboe/fio.

[8] Matias Bjørling, Jens Axboe, David Nellans, and Philippe Bonnet. Linux block io: introducing multi-queue ssd access on multi-core systems. In *Proceedings of the 6th International Systems and Storage Conference*, page 22. ACM, 2013.

[9] Woong Shin, Qichen Chen, Myoungwon Oh, Hyeonsang Eom, and Heon Y. Yeom. Os i/o path optimizations for flash solid-state drives. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, 2014.

[10] Young Jin Yu, Dong In Shin, Woong Shin, Nae Young Song, Jae Woo Choi, Hyeong Seog Kim, Hyeonsang Eom, and Heon Young Yeom. Optimizing the block i/o subsystem for fast storage devices. volume 32.

[11] Yongseok Son, Nae Young Song, Hyuck Han, Hyeonsang Eom, and Heon Young Yeom. Design and evaluation of a user-level file system for fast storage devices. *Cluster Computing*, 18(3), September 2015.

[12] File system in user space. https://github.com/libfuse/libfuse.

[13] Hyeong-Jun Kim, Young-Sik Lee, and Jin-Soo Kim. Nvmedirect: A user-space I/O framework for application-specific optimization on nvme ssds. In *8th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2016, Denver, CO, June 20-21, 2016.*, 2016.

[14] Data plane development kit. http://dpdk.org/.