SPDK NVMe BDEV Performance Report Release 20.10

Testing Date: November 2020

Performed by Karol Latecki (karol.latecki@intel.com)

Acknowledgments:

James Harris (james.r.harris@intel.com)

John Kariuki (john.k.kariuki@intel.com)

Contents

Audience and Purpose

This report is intended for people who are interested in comparing the performance of the SPDK block device layer vs the Linux Kernel (`5.4.14-100.fc30.x86_64`) block device layer. It provides performance and efficiency information between the two block layers under various test workloads.

The purpose of the report is not to imply a single “correct” approach, but rather to provide a baseline of well-tested configurations and procedures with repeatable and reproducible results. This report can be viewed as information regarding best known method/practice when performance testing the SPDK NVMe block device.

Test setup

Hardware configuration

Table 1: Test setup hardware configuration

Item	Description
Server Platform	Intel WolfPass **R2224WFTZS**
Motherboard	S2600WFT
CPU	2 CPU sockets, Intel(R) Xeon(R) Gold 6230N CPU @ 2.30GHz Number of cores 20 per socket, number of threads 40 per socket Both sockets populated
Memory	10 x 32GB Micron DDR4 36ASF4G72PZ-2G6H1R Total 320 GBs Memory channel population:

P1	**P2**
CPU1_DIMM_A1	CPU2_DIMM_A1
CPU1_DIMM_B1	CPU2_DIMM_B1
CPU1_DIMM_C1	CPU2_DIMM_C1
CPU1_DIMM_D1	CPU2_DIMM_D1
CPU1_DIMM_E1	CPU2_DIMM_E1

Operating System	Fedora 30
BIOS	02.01.0010 (06.01.2020)
Linux kernel version	5.4.14-100.fc30.x86_64
SPDK version	SPDK 20.07
Fio version	3.19
Storage	**OS:** 1x 120GB Intel SSDSC2BB120G4 **Storage**: 24x Intel® P4610™ 1.6TBs (FW: VDV10152) (6 on CPU NUMA Node 0, 18 on CPU NUMA Node 1)

BIOS Settings

Table 2: Test setup BIOS settings

Item	Description
BIOS	VT-d = Enabled CPU Power and Performance Policy = <Performance> CPU C-state = No Limit CPU P-state = Enabled Enhanced Intel® Speedstep® Tech = Enabled Turbo Boost = Enabled Hyper Threading = Enabled

Storage distribution across NUMA Nodes and PCIe Switches

Wolfpass server platforms PCIe Lanes are not symmetrically distributed between CPU NUMA nodes, which is an important factor in performance tests. Additionally, the total amount of PCIe Lanes available was not enough to accommodate for all installed 24 NVMe drives. Therefore, appropriate setup of PCIe riser cards and PCIe switches was used. For more information on PCIe capabilities of the platform please refer to its technical specification.

Table 3: Test platform NVMe storage setup

Item	Description	
PCIe Riser cards	Risers 1&2: 2x Intel A2UL16RISER2 (PCI gen 3 1x16 Riser) ○ Installed in Riser Slot #1 ○ Installed in Riser Slot #2 Riser 3: 1 x A2UX8X4RISER (PCI gen 3 1x8 Riser) ○ Installed in Riser Slot #3	
PCIe Switches	5 x Intel 4-Port PCIe Gen3 x8 Switch AIC AXXP3SWX08040 Installed in: ○ PCIe Switch 1: Riser Slot #1 port 1 (using CPU1 PCIe Lanes) ○ PCIe Switch 2: Riser Slot #1 port 2 (using CPU2 PCIe Lanes) ○ PCIe Switch 3: Riser Slot #2 port 1 (using CPU2 PCIe Lanes) ○ PCIe Switch 4: Riser Slot #2 port 2 (using CPU2 PCIe Lanes) ○ PCIe Switch 5: Riser Slot #3 port 1 (using CPU2 PCIe Lanes)	
NVMe Drives distribution across the system	Nvme0 – 1	Motherboard ports (CPU1 PCIe Lanes)
	Nvme2 – 3	Motherboard ports (CPU2 PCIe Lanes)
	Nvme4 – 7	PCIe Switch 1 (CPU1 PCIe Lanes)
	Nvme8 – 11	PCIe Switch 2 (CPU2 PCIe Lanes)
	Nvme12 – 15	PCIe Switch 3 (CPU2 PCIe Lanes)
	Nvme16 – 19	PCIe Switch 4 (CPU2 PCIe Lanes)
	Nvme20 – 23	PCIe Switch 5 (CPU2 PCIe Lanes)

SSD Preconditioning

An empty NAND SSD will often show read performance far beyond what the drive claims to be capable of because the NVMe controller knows that the device is empty and completes the read request successfully without performing any data transfer. Therefore, prior to running each performance test case we preconditioned the SSDs by writing 128K blocks to the device sequentially to fill the SSD capacity (including the over-provisioned areas) twice and force the internal state of the device into some known state. Additionally, the 4K 100% random writes performance decreases from one test to the next until the NAND management overhead reaches steady state because the wear-levelling activity increases dramatically until the SSD reaches steady state. Therefore, to obtain accurate and repeatable results for the 4K 100% random write workload, we ran the workload for 90 minutes before starting the benchmark test and collecting performance data. For a highly detailed description of exactly how to force an SSD into a known state for benchmarking see the SNIA Solid State Storage Performance Test Specification.

Kernel & BIOS Spectre-Meltdown information

Host server system uses 5.4.14 kernel version which is available from the DNF repository. The default Spectre-Meltdown mitigation patches for this kernel version have been left enabled.

Introduction to SPDK Block Device Layer

SPDK Polled Mode Driver

The NVMe PCIe driver is something that is usually expected to be part of the system Kernel and your application would interact with the driver via the system call interface. SPDK takes a different approach. SPDK unbinds the NVMe devices from the kernel and binds the hardware queues to a userspace NVMe driver. From that point, your application will access the device queues directly from userspace.

The SPDK NVMe driver is a C library that may be linked directly into an application that provides direct, zero-copy data transfer to and from NVMe SSDs. It is entirely passive, meaning that it spawns no threads and only performs actions in response to function calls from the application. The library controls NVMe devices by directly mapping the PCI BAR into the local process and performing MMIO. The SPDK NVMe driver is asynchronous, which means that the driver submits the I/O request as an NVMe submission queue entry on a queue pair and the function returns immediately, prior to the completion of the NVMe command. The application must poll for I/O completion on each queue pair with outstanding I/O to receive completion callbacks.

SPDK Block Device Layer

SPDK further provides a full block stack as a user space library that performs many of the same operations as a block stack in an operating system. The SPDK block device layer often simply called **bdev**, is a C library intended to be equivalent to the operating system block storage layer located above the device drivers in traditional kernel storage stack.

The bdev module provides an abstraction layer that provides common APIs for implementing block devices that interface with different types of block storage device. An application can use the APIs to enumerate and claim SPDK block devices, and then perform asynchronous I/O operations (such as read, write, unmap, etc) in a generic way without knowing if the device is an NVMe device or SAS device or something else. The SPDK NVMe bdev module can create block devices for both local PCIe-attached NVMe device and remote devices exported over NVMe-oF.

In this report, we benchmarked the performance and efficiency of the bdev for the local PCIe-attached NVMe devices use case. We also demonstrated the benefits of the SPDK approaches, like user-space polling, asynchronous I/O, no context switching etc. under different workloads.

FIO Integration

SPDK provides an FIO plugin for integration with Flexible I/O benchmarking tool. The quickest way to generate a configuration file with all the bdevs for locally PCIe-attached NVMe devices is to use the gen_nvme.sh script with "—json-with-subsystems" option as shown in Figure 1.

```
[user@localhost spdk]$ sudo scripts/gen_nvme.sh --json-
with-subsystems | jq
{
  "subsystems": [
    {
      "subsystem": "bdev",
      "config": [
        {
          "method": "bdev_nvme_attach_controller",
          "params": {
            "trtype": "PCIe",
            "name": "Nvme0",
            "traddr": "0000:1a:00.0"
          }
        },


        [...]

          "method": "bdev_nvme_attach_controller",
          "params": {
            "trtype": "PCIe",
            "name": "Nvme22",
            "traddr": "0000:de:00.0"
          }
        },
        {
          "method": "bdev_nvme_attach_controller",
          "params": {
            "trtype": "PCIe",
            "name": "Nvme23",
            "traddr": "0000:df:00.0"
          }
        }
      ]
    }
  ]
```

Figure 1 : Example NVMe bdev configuration file

Add SPDK bdevs to the fio job file, by setting the *ioengine=spdk_bdev* and adding the *spdk_conf* parameter whose value points to the NVMe bdev configuration file.

The example fio configuration file in figure 2, shows how to define multiple fio jobs and assign NVMe bdevs to each job. Each job is also pinned to a CPU core on the same NUMA node as the NVMe SSDs that the job will access.

Finally, to use the bdev fio plugin specify the LD_PRELOAD when running fio.

LD_PRELOAD=<path to spdk repo>/examples/bdev/fio_plugin/fio_plugin fio <fio job file>

```
[global]
direct=1
thread=1
time_based=1
norandommap=1
group_reporting=1
ioengine=spdk_bdev
spdk_conf=/tmp/bdev.conf

rw=randread
rwmixread=70
bs=4096
numjobs=1
runtime=240
ramp_time=60

[filename0]
iodepth=192
cpus_allowed=0
filename=Nvme0n1
filename=Nvme1n1
filename=Nvme4n1
filename=Nvme5n1
filename=Nvme6n1
filename=Nvme7n1

[filename1]
iodepth=192
cpus_allowed=21
filename=Nvme2n1
filename=Nvme3n1
filename=Nvme8n1
filename=Nvme9n1
filename=Nvme10n1
filename=Nvme11n1

[filename2]
iodepth=192
cpus_allowed=22
filename=Nvme12n1
filename=Nvme13n1
filename=Nvme14n1
filename=Nvme15n1
filename=Nvme16n1
filename=Nvme17n1

[filename3]
iodepth=192
cpus_allowed=23
filename=Nvme18n1
filename=Nvme19n1
filename=Nvme20n1
filename=Nvme21n1
filename=Nvme22n1
filename=Nvme23n1
```

Figure 2: Example SPDK Fio BDEV configuration file

Test Case 1: SPDK NVMe BDEV IOPS Test

Purpose: The purpose of this test case was to measure the maximum performance in IOPS/Core of the NVMe block layer on a single CPU core. We used different benchmarking tools (SPDK bdevperf vs. SPDK FIO BDEV plugin vs SPDK NVMe perf) to understand the overhead of benchmarking tools. Measuring IOPS was key in this test case, so latency measurements were either disabled or skipped.

The following Random Read/Write workloads were used:

- 4KB 100% Random Read
- 4KB 100% Random Write
- 4KB Random 70% Read 30% Write

For each workload we followed the following steps:

1) Precondition SSDs as described in "Test Setup" chapter.
2) Run each test workload: Start with a configuration that has 24 Intel P4610x NVMe devices and decrease the number of SSDs by 2 on each subsequent run.
    - This shows us the IOPS scaling as we add SSDs till the maximum IOPS/Core is reached.
    - Starting with 24 SSDs and reducing the number of SSDs on subsequent eliminates having to precondition between runs because all SSDs used in the subsequent run were used in the previous run so they should still be in a steady state.
3) Repeat three times. The data reported is the average of the 3 runs.

Table 4: SPDK NVMe BDEV IOPS Test configuration

Item	Description
Test case	SPDK NVMe BDEV IOPS/Core Test
Test configuration	**FIO Version**: fio-3.19 **Number of NVMe SSDs**: scaled as follows: 24, 22, … 2, 1. Decreasing 2 SSDs on each test run. **SPDK_BDEV_IO_CACHE_SIZE** changed from 256 to 2048. **NUMA optimization:** The test platform has PCIe lanes unevenly distributed between NUMA nodes, most of the NVMe SSDs (18 out of total 24) are located on NUMA node 1. Therefore, a CPU Core from NUMA node 1 was selected as primary core for test, in order to reduce the overhead of cross-numa operations.
Bdevperf configuration	`./bdevperf -c bdev.conf -q ${iodepth} -o ${block_size} -w ${rw} -M ${rwmixread} -t 300 -m 20 -p 20`

FIO configuration	`[global]`
	`ioengine=spdk_bdev`
	`spdk_conf=bdev.conf`
	
	`gtod_reduce=1`
	`direct=1`
	`thread=1`
	`norandommap=1`
	`time_based=1`
	`ramp_time=60s`
	`runtime=240s`
	
	`bs=4k`
	`rw={randread, randwrite, randrw}`
	`rwmixread={100,70,0}`
	`iodepth={32, 64, 128, 256}`
	`numjobs=1`

SPDK NVMe BDEV Single Core Throughput

The first test was performed using SPDK bdevperf, which is lightweight benchmarking tool that adds minimal latency to the I/O path. The charts below show the Single core IOPS results for the SPDK Block Layer with increasing number of NVMe SSDs.

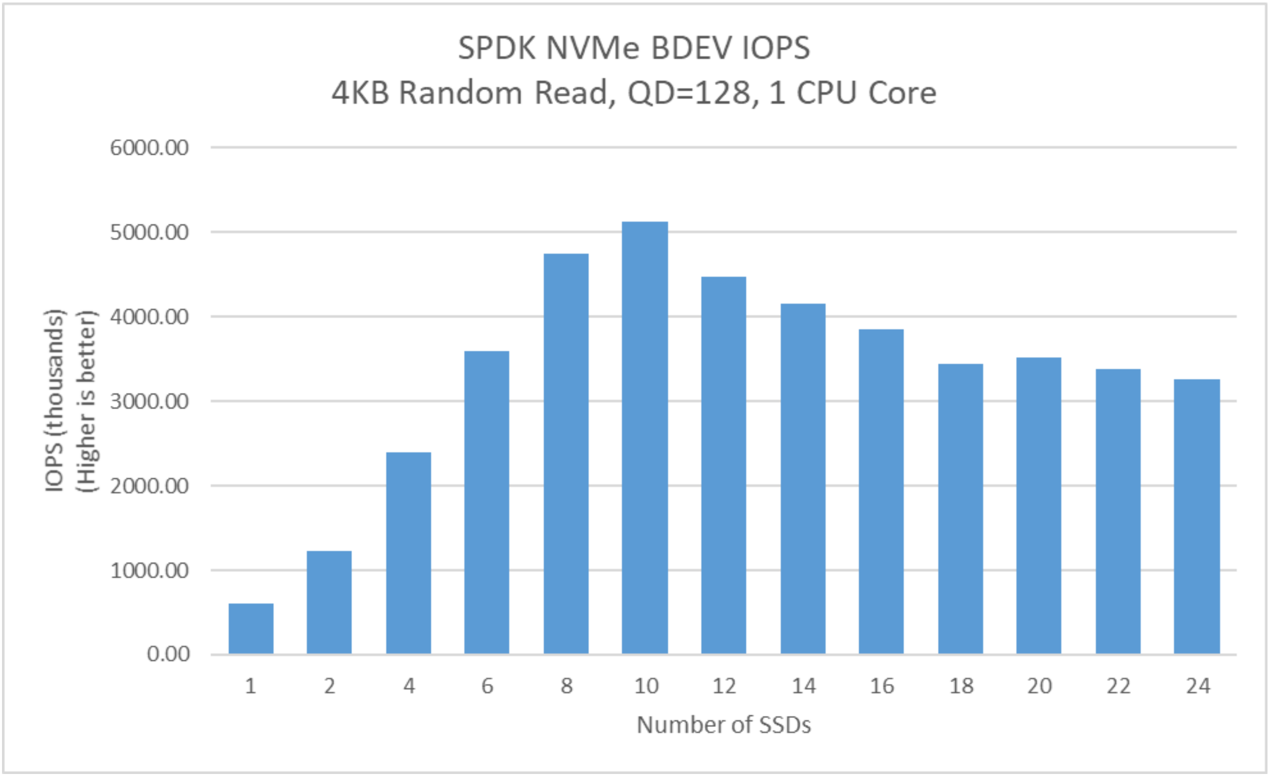

Figure 3: SPDK NVMe BDEV IOPS scalability with addition of SSDs (4KB Random Read, 1CPU Core, QD=128, using bdevperf tool)

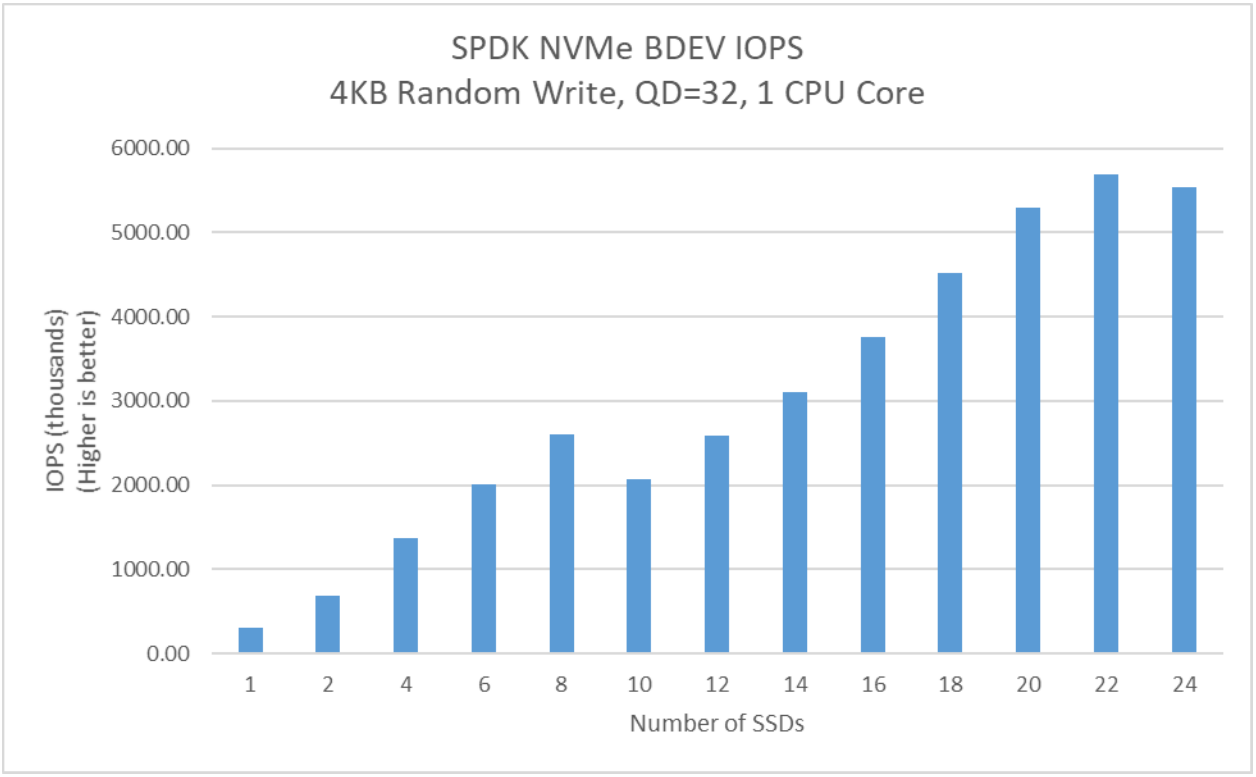

Figure 4: SPDK NVMe BDEV IOPS scalability with addition of SSDs (4KB Random Write, 1CPU Core, QD=32, using bdevperf tool)

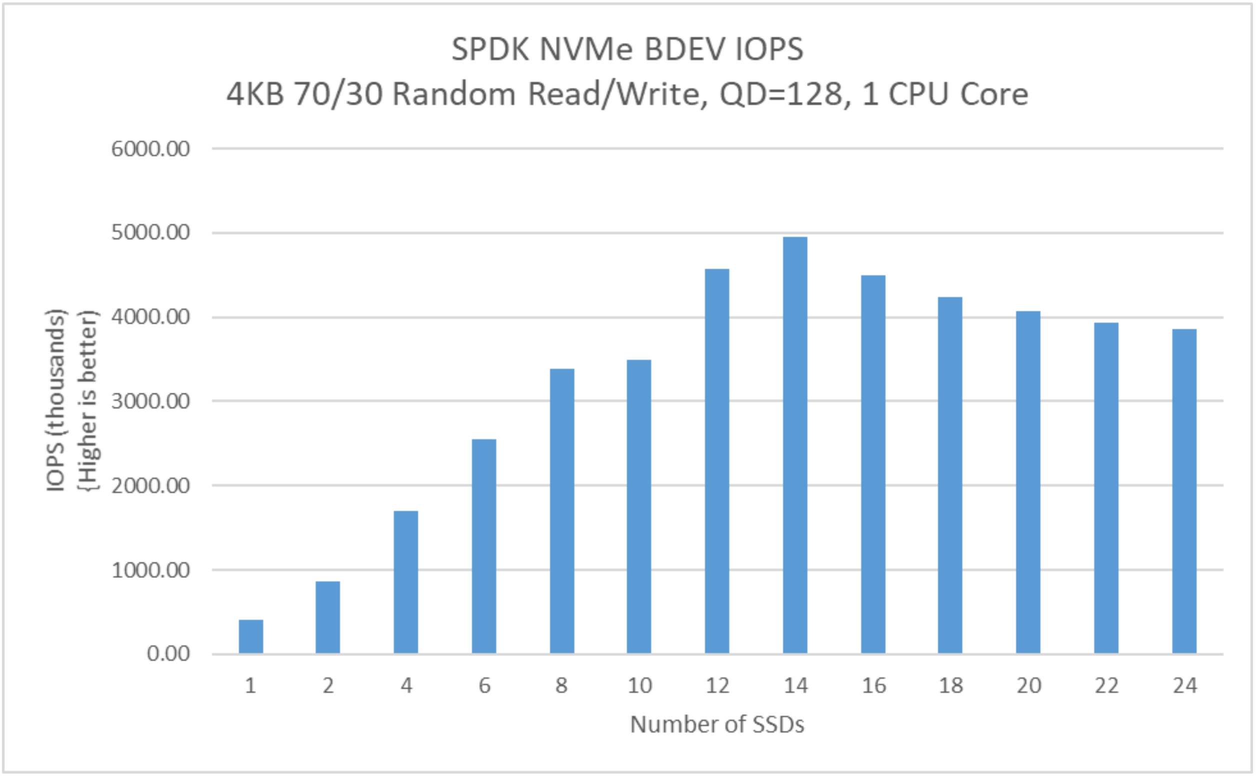

Figure 5: SPDK NVMe BDEV IOPS scalability with addition of SSDs (4KB 70/30 Random Read/Write, 1CPU Core, QD=128, using bdevperf tool)

Bdevperf vs. FIO IOPS/Core results

SPDK provides the bdevperf benchmarking tool that provides minimal capabilities needed to define basic workloads and collects a limited amount of data. The FIO benchmarking tool provides a lot of great features to enable users to quickly define workloads, scale the workloads and collect many data points for detailed performance analysis, however, at cost of higher overhead. This test compares the performance in IOPS/core of the bdevperf and FIO benchmarking tools.

Table 5: IOPS/Core performance; SPDK FIO bdev plugin vs SPDK bdevperf (Blocksize=4KB, 1 CPU Core)

Workload	SDPK Fio BDEV Plugin (IOPS, thousands)	SPDK Bdevperf (IOPS, thousands)	Performance gain
4KB Random Read, QD=128, 10 SSDs	2867.11	5131.86	79.0%
4KB Random Write, QD=32, 22 SSDs	2967.80	5685.03	91.6%
4KB 70/30 Random Read/Write, QD=128, 14 SSDs	2455.09	4948.27	101.6%

The overhead of the benchmarking tools is important when you are testing a system that is capable of millions of IOPS/Core. Using a benchmarking tool that has minimal overhead like the SPDK bdevperf yields up to 101% more IOPS/Core than FIO.

NVMe BDEV vs. Polled-Mode Driver IOPS/Core

In this test case, we compared the throughput of the NVMe BDEV with that of the polled-mode driver. How to read this data? The SPDK block layer provides several key features at a cost of up to 20% more CPU utilization. If you are building a system with many SSDs that is capable of millions of IOPS, you can take advantage of the block layer features at the cost of approximately 1 additional CPU core for every 5 I/O cores. Comparison was done using SPDK Bdevperf and Nvmeperf test tools.

Table 6: SPDK NVMe Bdev vs SPDK NVMe PMD IOPS/Core (Blocksize=4KB, 1 CPU Core)

Workload	SPDK Bdevperf (IOPS, thousands)	SPDK Nvmeperf (IOPS, thousands)	Performance gain
4KB Random Read, QD=128, 12 SSDs	5131.86	6054.35	18%
4KB Random Write, QD=32, 20 SSDs	5534.58	6652.14	20%

Conclusions

1) The SPDK NVMe block device comes at a cost of about 20% less I/O operations as compared to SPDK NVMe Polled-Mode Driver.

2) Performance scales linearly with addition of NVMe SSDs up to 8 and 14 SSDs for Random Read and Random Read/Write workloads, reaching around 5.1 and 4.9 million IOPS respectively.

3) Performance scaling is close to linear for Random Write workloads up 20 NVMe SSDs, reaching around 5.6M IOPS.

4) For all workloads there is a noticeable performance degradation with addition of more NVMe SSDs after peak performance point has been reached.

Test Case 2: SPDK NVMe BDEV I/O Cores Scaling

Purpose: The purpose of this test case is to demonstrate the I/O throughput scalability of the NVMe BDEV module with the addition of more CPU cores to perform I/O. The number of CPU cores used was scaled as 1, 2, 3, 4 and 5.

Test Workloads: We use the following Random Read/Write mixes

- 4KB 100% Random Read
- 4KB 100% Random Write
- 4KB Random 70% Read 30% Write

Table 7: SPDK NVMe BDEV I/O Cores Scalability Test

Item	Description
Test case	Test SPDK NVMe BDEV I/O Cores Scalability Test
Test configuration	**Number of CPU Cores:** 1, 2, 3, 4, 5 **Number of NVMe SSDs:** 6 per each CPU Core used in test **NUMA optimization:** The test platform has PCIe lanes unevenly distributed between NUMA nodes, most of the NVMe SSDs (18 out of total 24) are located on NUMA node 1. Therefore, only CPU Cores from NUMA node 1 were selected for test, in order to reduce the overhead of cross-numa operations.

Bdev perf configuration	`spdk/test/bdev/bdevperf/bdevperf --json bdev.conf \`
	`       -q 128 -o 4096 -w randrw -M ${MIXREAD} \`
	`       -t 300 -m ${CORE_MASK} -p ${PRIMARY_CORE}`

Results

Table 8: SPDK NVMe BDEV I/O Cores Scalability Test (4KB 100% Random Read IOPS at QD=128; 4KB 100% Random Write IOPS at QD=32; 4KB 70/30 Random Read/Write IOPS at QD=128)

		IOPS (thousands)		
CPU Cores	**NVMe SSDs**	**Random Read QD=128**	**Random Write QD=32**	**70/30 Random Read/Write QD=128**
1	**6**	3652.04	1244.75	2094.05
2	**12**	7285.59	2576.54	4419.27
3	**18**	10382.87	3977.72	6293.92
4	**24**	10975.49	5529.03	8426.85
5	**24**	10966.01	6096.72	8467.05

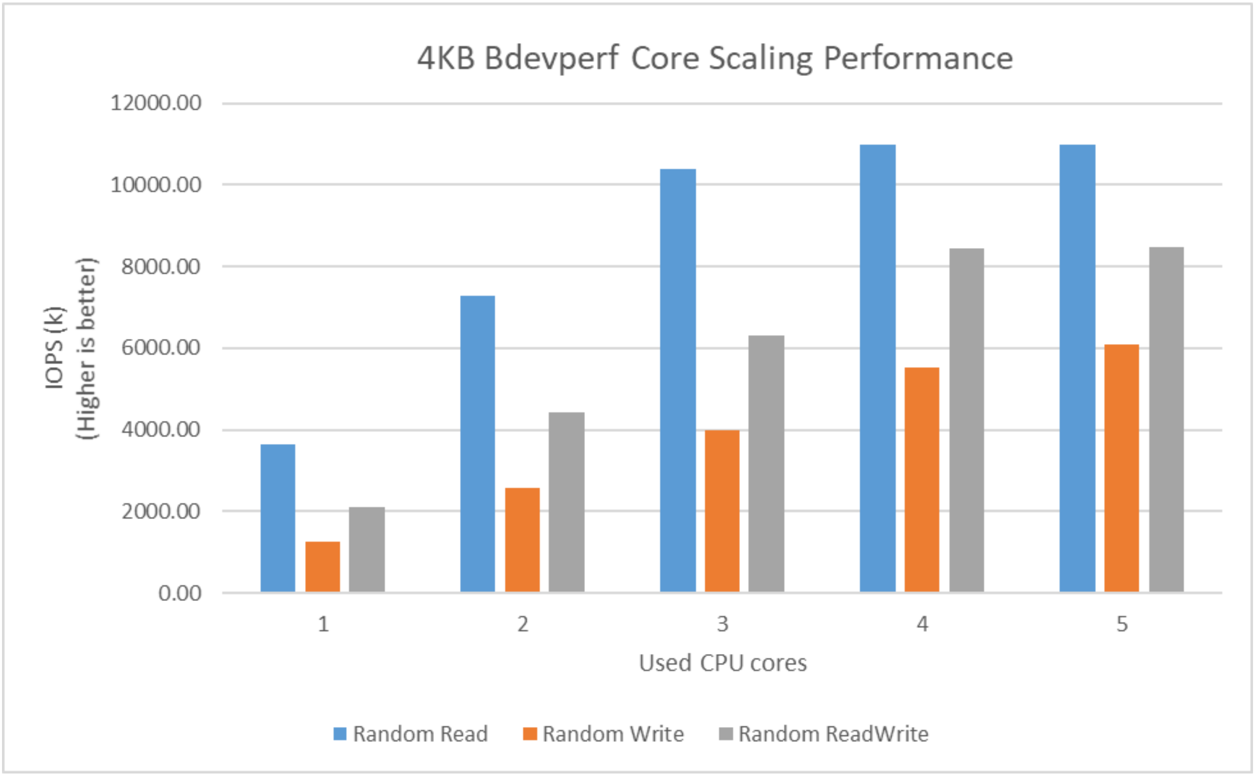

Figure 6: SPDK NVMe BDEV I/O Cores Scalability with addition of SSDs (4KB 100% Random Read IOPS at QD=128; 4KB 100% Random Write IOPS at QD=32; 4KB 70/30 Random Read/Write IOPS at QD=128)

Conclusions

1. The IOPS for the 4 KiB random read workload scales up linearly with the addition of I/O cores until the PCIe switches in platform are saturated (at about 10.5M IOPS; see "Test setup" chapter for more information).

2. The IOPS for the 4 KiB random write workload scaling is close to linear. At 4 and 5 CPU cores the IOPS exceeded the expected NVMe SSDs throughput for this workload which is about 4.8M IOPS, we suspect this is due to a not perfect preconditioning process, which wears off over

time. However, the results were repeatable and showed SPDK’s high scalability with addition of I/O cores.

3. The IOPS for the 4 KiB random 70/30 read/write workload scales up linearly with the addition of I/O cores up to 4 CPU cores. At this point peak performance of 8.5 million IOPS is reached and increasing the number of cores to 5 does not improve performance.

Test Case 3: SPDK NVMe BDEV Latency

This test case was carried out to understand latency characteristics while running SPDK NVMe bdev and its comparison to Linux Kernel NVMe block device layer. We used SPDK FIO BDEV Plugin instead of the SPDK Bdevperf tool, as it allowed us to gather detailed latency metrics. FIO was ran for 15 minutes targeting a single block device over a single NVMe drive. This test compares consistency between latency of the SPDK and Linux Kernel block layers over time in a histogram. The Linux Kernel block layer provides I/O polling capabilities to eliminate overhead such as context switch, IRQ delivery delay and IRQ handler scheduling. This test case includes a comparison of the I/O latency for the Kernel vs. SPDK.

Test Workloads: We use the following workloads:

- 4KB 100% Random Read
- 4KB 100% Random Write

Table 9: SPDK NVMe BDEV Latency Test

Item	Description
Test case	Test SPDK NVMe BDEV Latency Test
Test configuration	**FIO Version:** fio-3.19 **Number of CPU Cores:** 1 **Number of NVMe SSDs:** 1
SPDK NVMe Driver Configuration	ioengine=spdk_bdev
Linux Kernel Default (libaio) Configuration	ioengine=libaio

Linux Kernel io_uring	ioengine=io_uring
	
	System NVMe block device configuration:
	`echo 0 > /sys/block/nvme0n1/queue`
	`echo 0 > /sys/block/nvme0n1/rq_affinity`
	`echo 2 > /sys/block/nvme0n1/nomerges`
	`echo -1 > /sys/block/nvme0n1/io_poll_delay`

FIO configuration (common part)	`[global]`
	`direct=1`
	`thread=1`
	`time_based=1`
	`norandommap=1`
	`group_reporting=1`
	
	`rw={randread | randwrite}`
	`bs=4096`
	`runtime=900`
	`ramp_time=120`

	`numjobs=1` `log_avg_msec=15` `write_lat_log=/tmp/tc3_lat.log`
FIO configuration (SPDK specific)	`[global]` `ioengine=spdk_bdev` `spdk_conf=/tmp/bdev.conf` `[filename0]` `iodepth=1` `cpus_allowed=20` `filename=Nvme0n1`
FIO configuration (Linux Kernel common)	`[global]` `ioengine={libaio | io_uring}` `[filename0]` `iodepth=1` `cpus_allowed=20` `filename=/dev/nvme18n1`
FIO configuration (Linux Kernel io_uring specific)	`[global]` `fixedbufs=1` `hipri=1` `registerfiles=1` `sqthread_poll=1`

The Linux block layer implements I/O polling on the completion queue. Polling can remove context switch(cs) overhead, IRQ delivery and IRQ handler scheduling overhead[1].

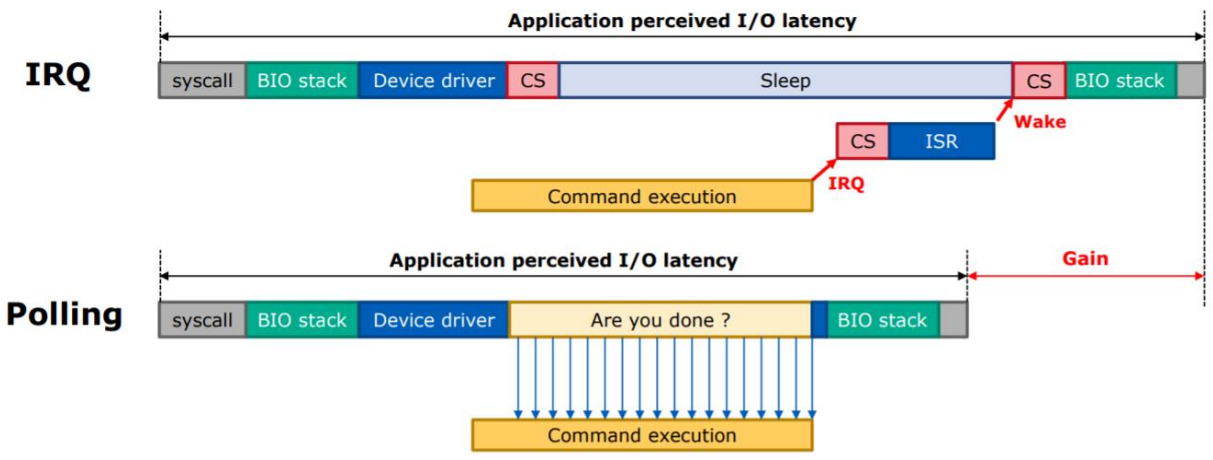

Figure 7: Linux Block Layer I/O Optimization with Polling. Source [1]

Furthermore, the Linux block I/O polling provides a mechanism to reduce the CPU load. In the *Classic Polling* model, the CPU spin-waits for the command completion and utilizes 100% of a CPU core [1]. There's also an adaptive hybrid polling which reduces the CPU load by putting the I/O polling thread to sleep for about half of the command execution time, but the polling thread must be woken up before the I/O completes with enough heads-up time for a context switch[1]. Hybrid pooling mode was not used for testing in this document.

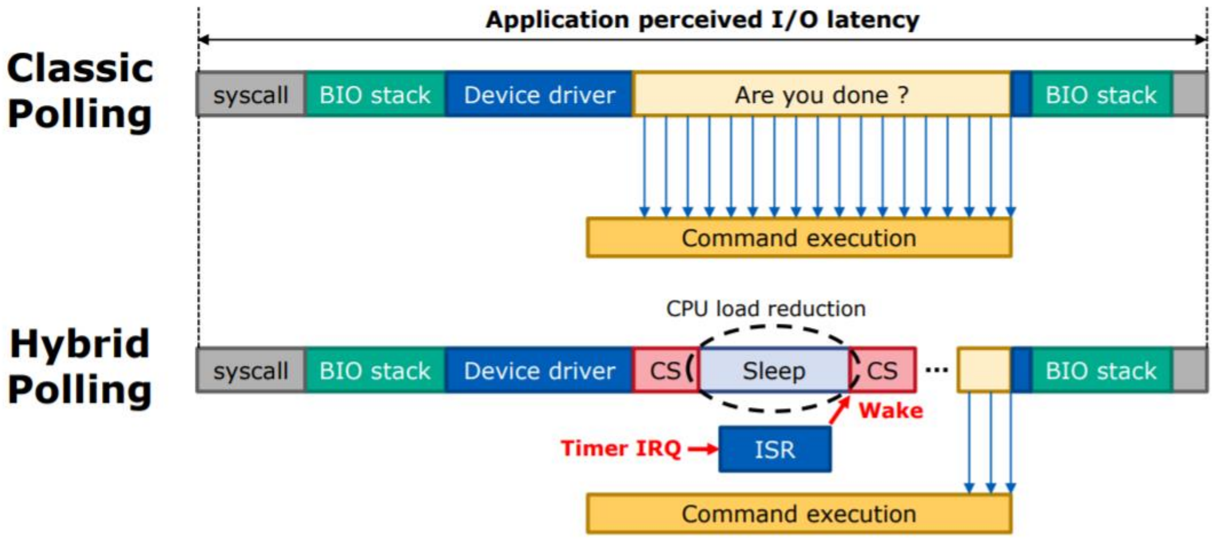

Figure 8: Linux Block I/O Classic and Hybrid Polling latency breakdown. Source [1]

The data in tables and charts compares the I/O latency for a various 4KB workloads performed using the SPDK BDEV vs. Linux block layerI/O model libaio and io_uring with polling mode enabled.

Average and tail latency comparison

Table 10: SPDK bdev vs. Linux Kernel latency comparison (4KB Random Read, QD=1, runtime=900s)

Latency metrics (usec)	SDPK Fio BDEV Plugin	Linux Kernel (libaio)	Linux Kernel (io_uring)
Average	72.89	89.59	74.06
P90	98.82	110.08	99.84
P99	144.38	154.62	146.43
P99.99	292.86	350.21	216.06
Stdev	23.28	23.74	22.91
Average submission latency	0.11	5.78	0.00
Average completion latency	72.78	83.43	74.03

Table 11: SPDK bdev vs. Linux Kernel latency comparison (4KB Random Write, QD=1, runtime=900s)

Latency metrics (usec)	SDPK Fio BDEV Plugin	Linux Kernel (Default libaio)	Linux Kernel (io_uring)
Average	10.25	13.75	11.37
P90	19.33	17.28	19.84
P99	37.12	33.54	36.10
P99.99	81.41	72.19	82.43
Stdev	7.10	5.54	6.57
Average submission latency	0.15	2.05	0.00
Average completion latency	10.10	11.57	11.26

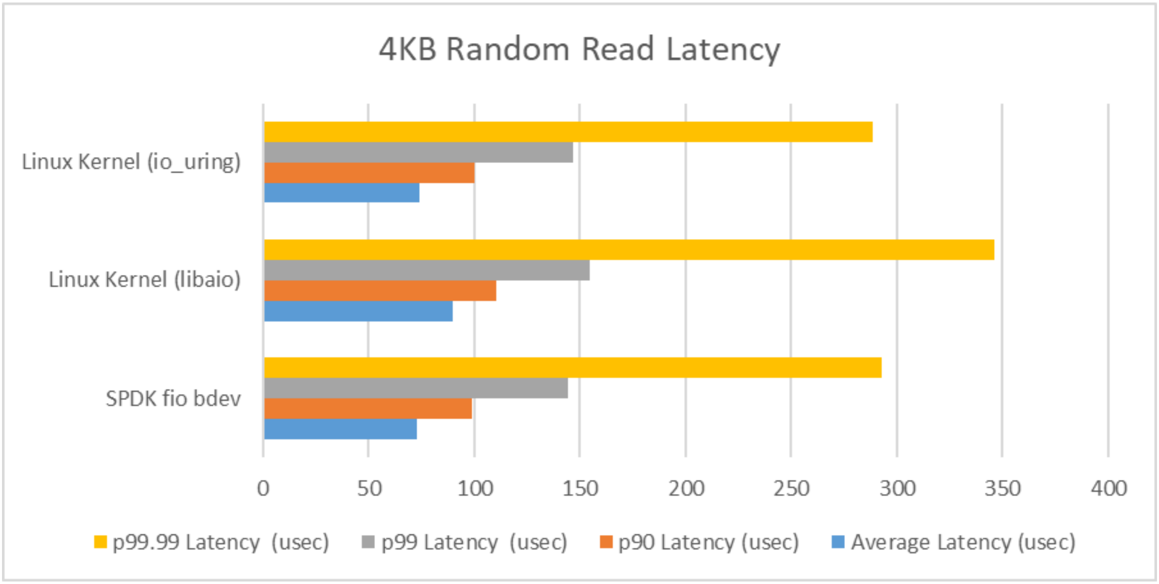

Figure 9: SPDK bdev vs Linux Kernel Latency comparison (4KB Random Read)

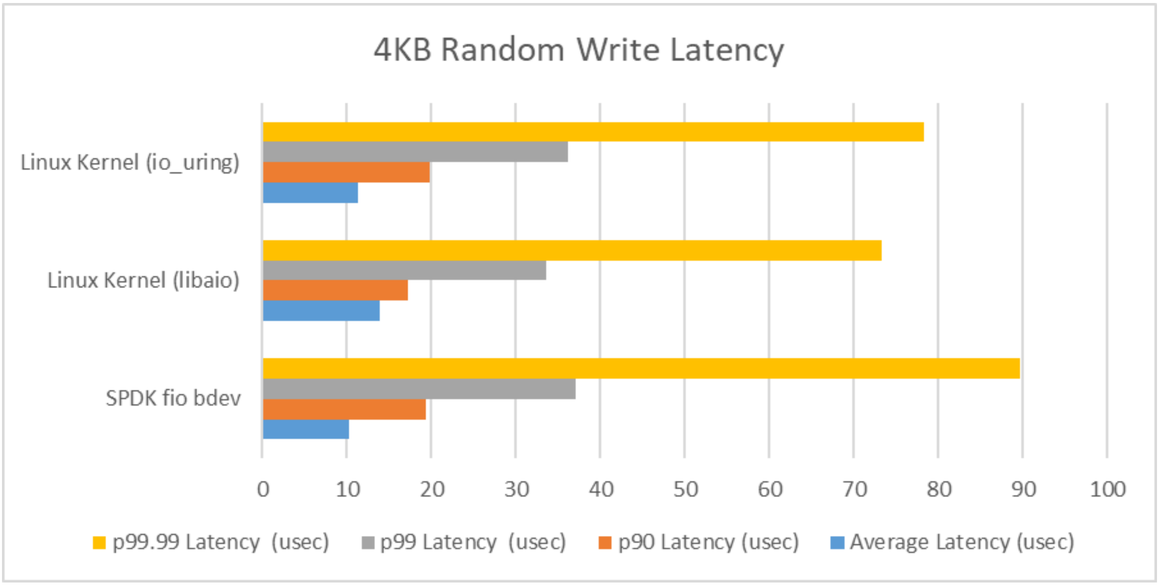

Figure 10: SPDK bdev vs Linux Kernel Latency comparison (4KB Random Write)

Linux Kernel libaio Histograms

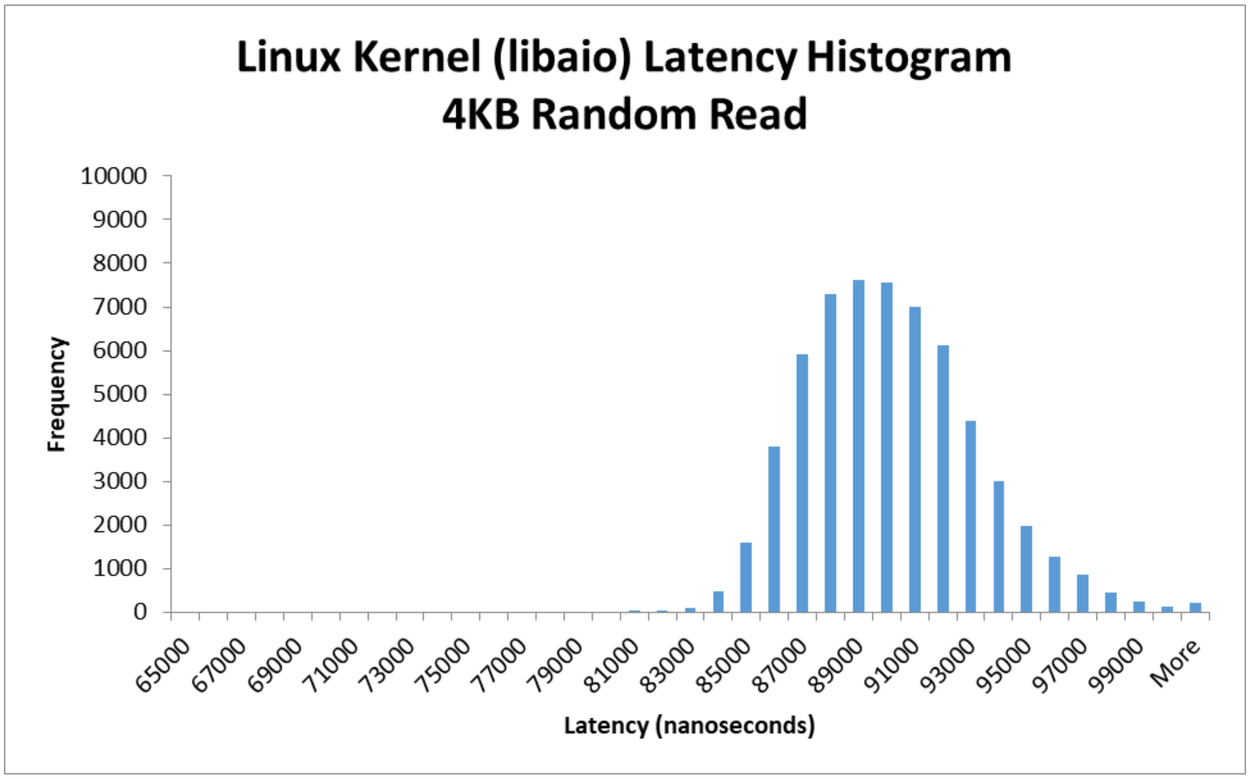

Figure 11: Linux Kernel (Default libaio) 4KB Random Read Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec)

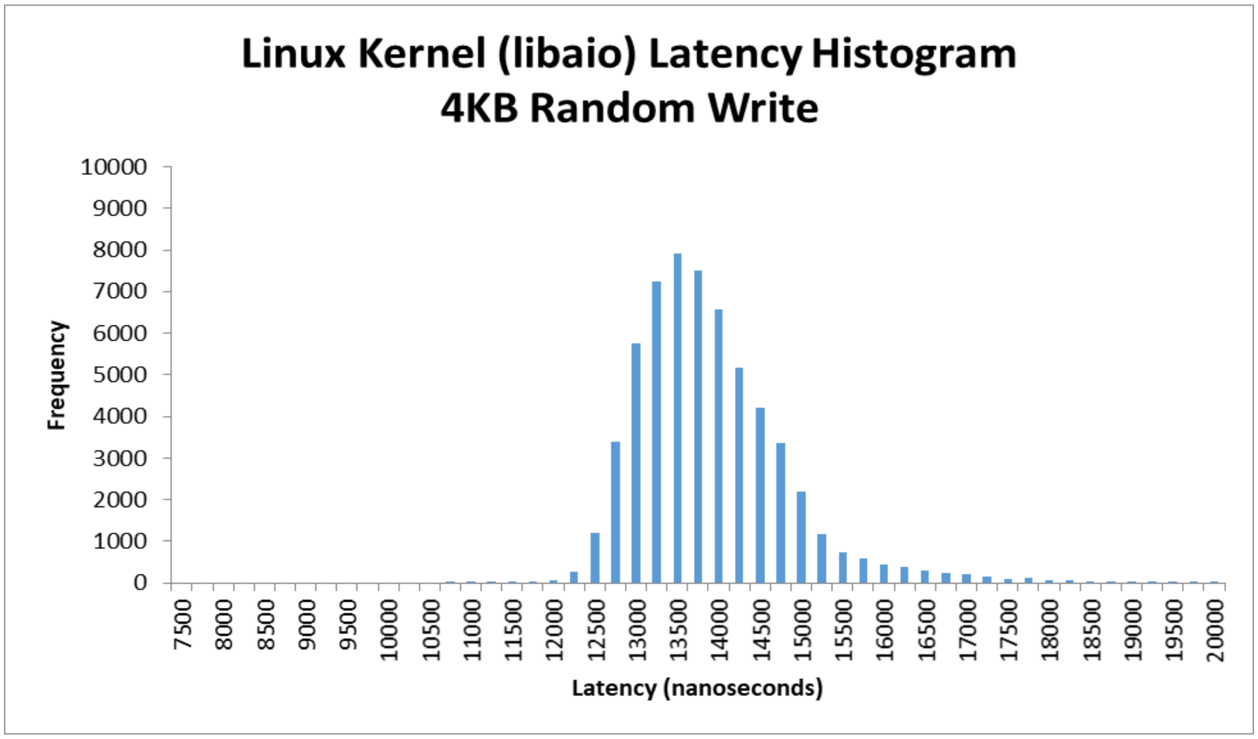

Figure 12: Linux Kernel (Default libaio) 4KB Random Write Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec)

Linux Kernel io_uring Histograms

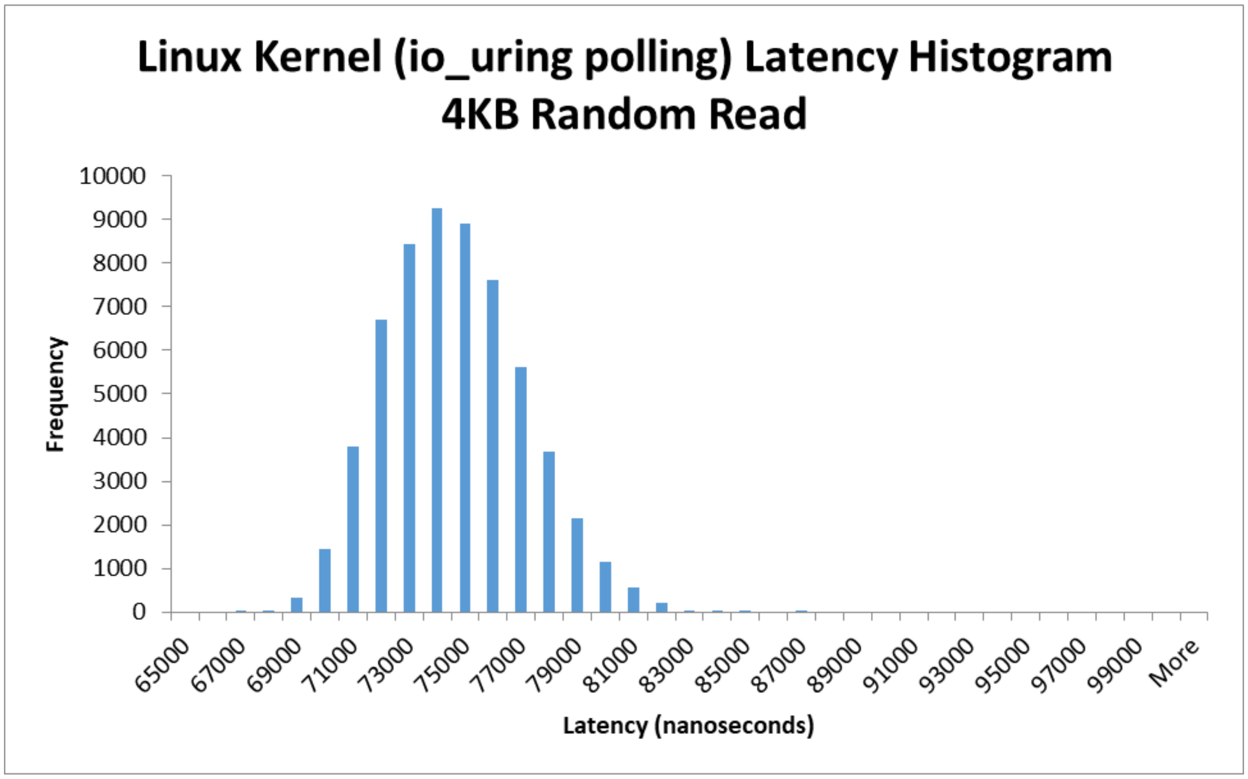

Figure 13: Linux Kernel (io_uring polling) 4KB Random Read Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec)

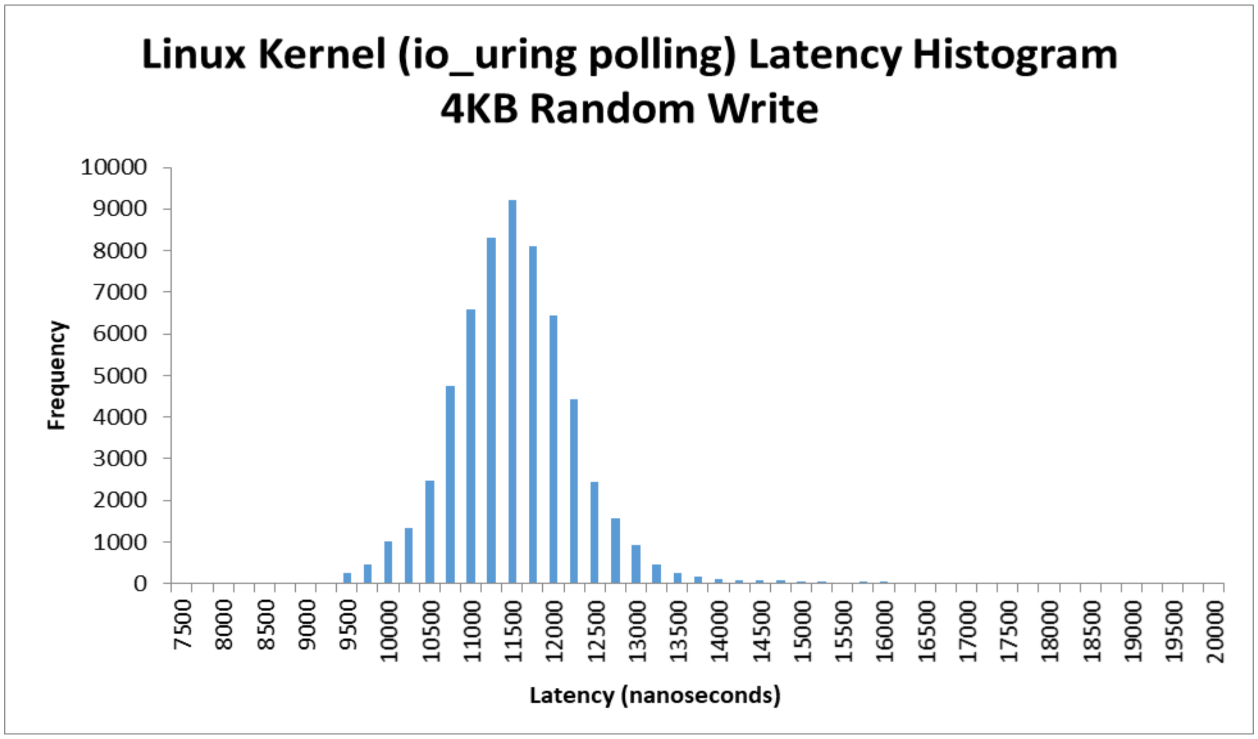

Figure 14: Linux Kernel (io_uring polling) 4KB Random Write Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec)

SPDK FIO Bdev Histograms

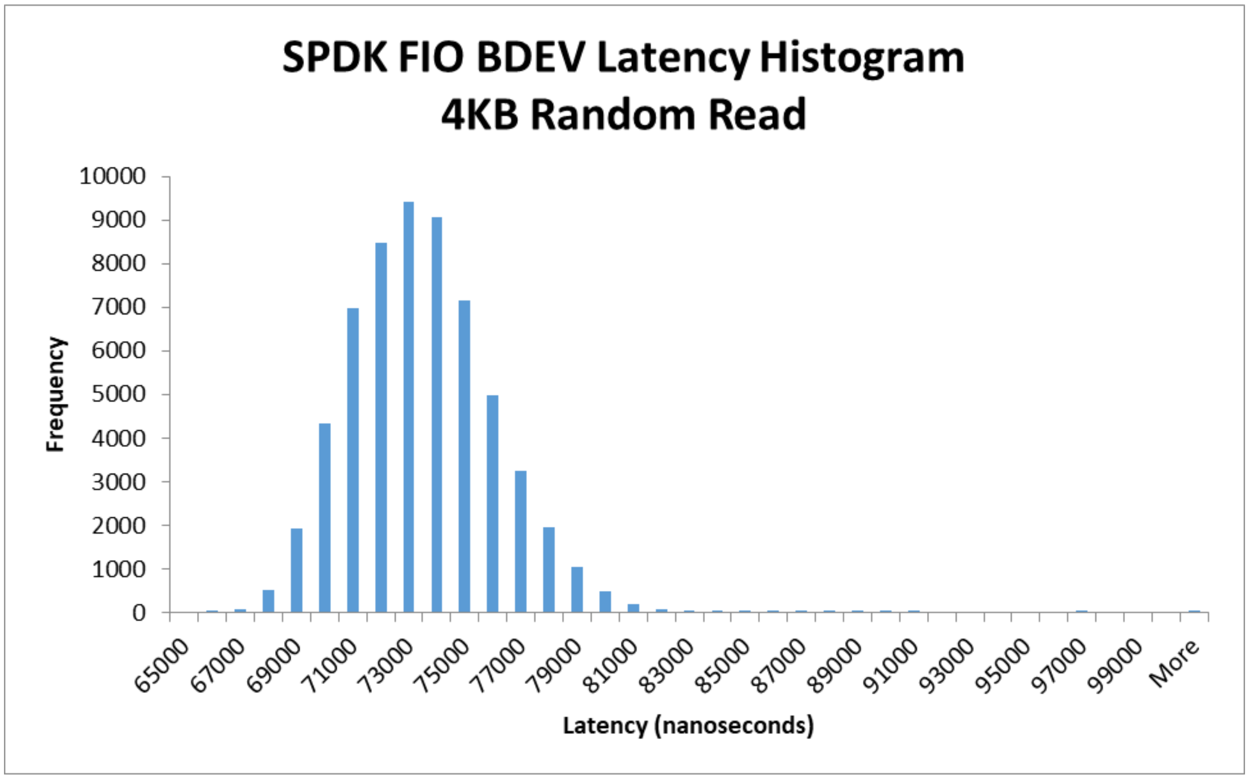

Figure 15: SPDK BDEV NVMe 4KB Random Read Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec)

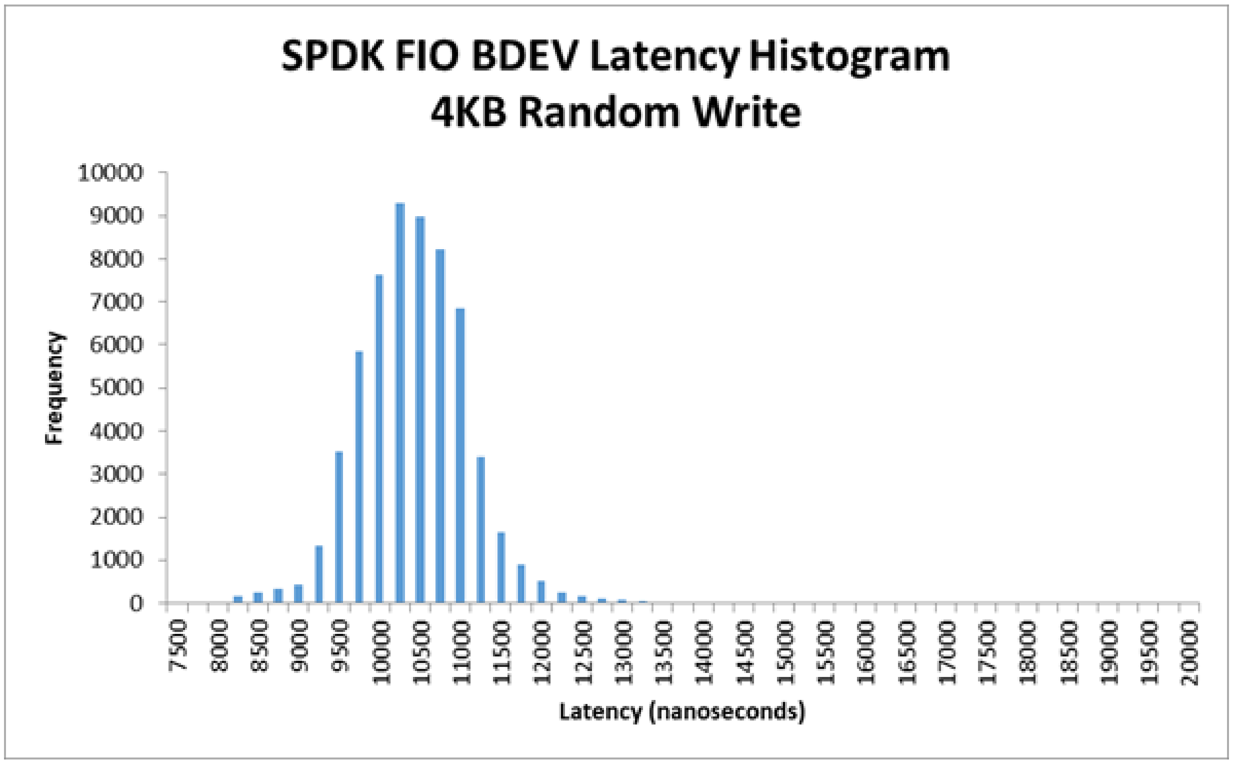

Figure 16: SPDK BDEV NVMe 4KB Random Write Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec)

Performance vs. increasing Queue Depth

Purpose: Understand the performance in IOPS and average latency of SPDK vs. the Linux io_uring polling and libaio block layer as the queue depth increases by powers of 2 from 1 to 512 for single NVMe SSD and single CPU Core.

Table 12: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io_uring polling (4KB Random Read , 1 NVMe SSD, 1 CPU Core, Numjobs=1)

	SPDK		Linux Kernel (Default libaio)		Linux Kernel (io_uring polling)	
QD	**IOPS**	**Avg. Lat. (usec)**	**IOPS**	**Avg. Lat. (usec)**	**IOPS**	**Avg. Lat. (usec)**
1	13665	73	10917	89	13233	75
2	27037	74	21548	91	26208	76
4	52903	75	43516	91	51498	77
8	101103	79	92208	86	98662	81
16	185392	86	169456	94	182100	88
32	315075	101	272832	117	308205	103
64	475523	134	434739	147	460466	139
128	610570	209	454580	281	602516	212
256	652667	392	451994	566	638331	401
512	652458	784	457528	1119	652116	785

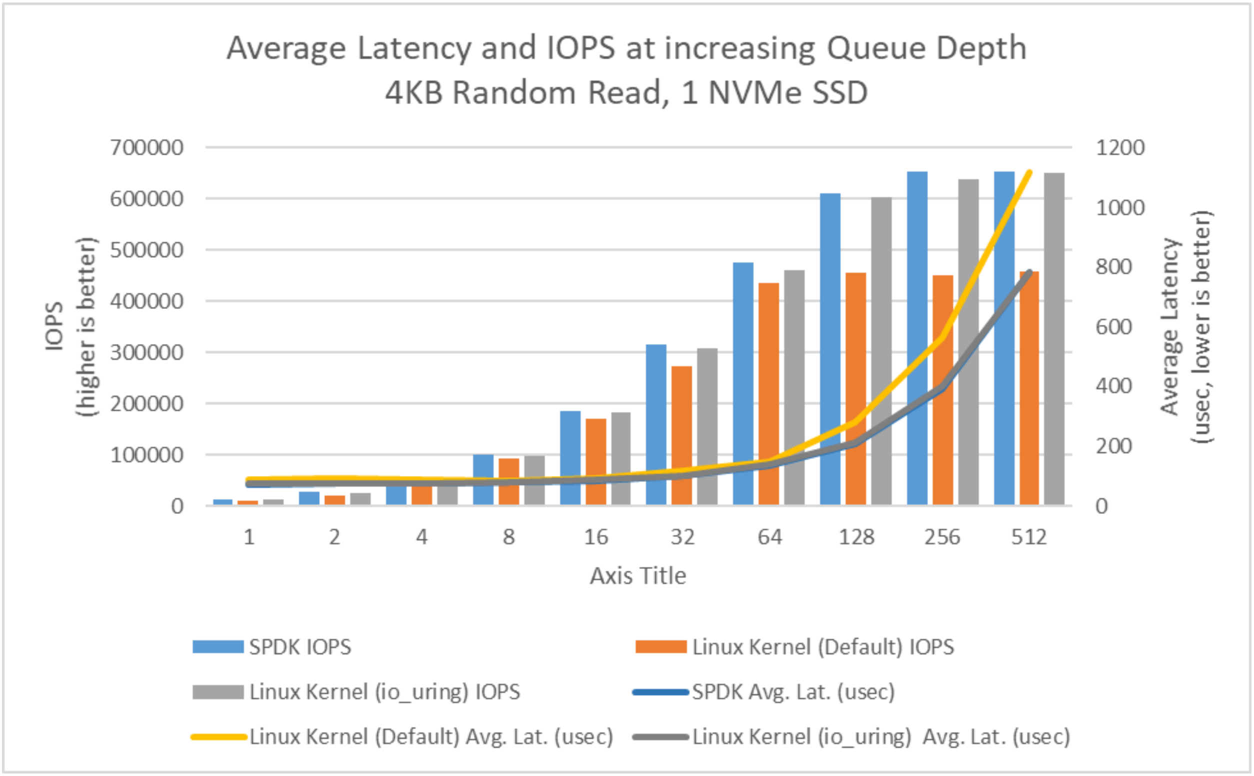

Figure 17: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io_uring polling (4KB Random Read, 1 NVMe SSD, 1 CPU Core, Numjobs=1)

	SPDK		Linux Kernel (Default libaio)		Linux Kernel (io_uring polling)	
QD	**IOPS**	**Avg. Lat. (usec)**	**IOPS**	**Avg. Lat. (usec)**	**IOPS**	**Avg. Lat. (usec)**
1	147405	7	75461	12	116068	8
2	252384	8	174843	11	198724	10
4	376663	10	297566	13	313584	12
8	482868	16	390023	20	423848	19
16	482545	33	429256	37	474053	33
32	474412	67	438171	73	472449	67
64	475148	135	439068	146	473402	135
128	472990	271	440820	290	468268	273
256	457449	560	434798	589	454665	563
512	483392	1059	428037	1196	476167	1076

Table 13: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io_uring polling (4KB Random Write, 1 NVMe SSD, 1 CPU Core, Numjobs=1)

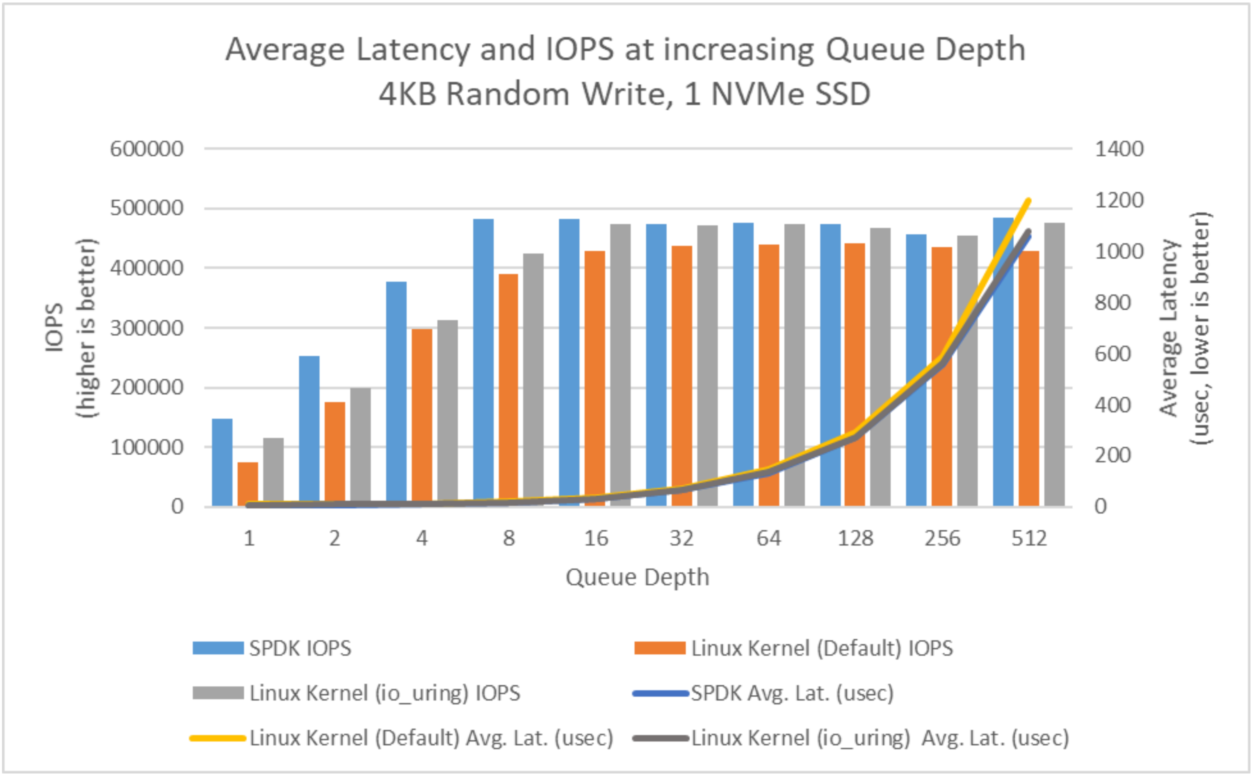

Figure 18: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io_uring polling (4KB Random Write, 1 NVMe SSD, 1 CPU Core, Numjobs=1)

Conclusions

1. Polling hardware for completion instead of relying on interrupts lowers both total latency and its variance.

2. SPDK NVMe Bdev average latency was up to 18% and 25% lower than Linux Kernel Libaio, for Random Read and Random Write workloads respectively.

3. SPDK NVMe Bdev average latency was up to 1.5% and 9.85% lower than Linux Kernel io_uring, for Random Read and Random Write workloads respectively.

4. SPDK NVMe Bdev IOPS throughput scaled almost linearly with increasing queue depth until the NVMe SSDs was saturated

5. Kernel io_uring IOPS throughput scaled almost linearly with increasing queue depth until the NVMe SSD was saturated.

6. Kernel libaio IOPS throughput scaling was not linear and the peak IOPS of approximately 440K IOPS was achieve at QD=64. We were unable to fully saturate NVMe SSD by increasing queue depth with just one CPU core. Both IOPS and latency results are worse than SPDK and io_uring.

Test Case 4: IOPS vs. Latency at different queue depths

Purpose: This test case was performed in order to understand throughput & latency trade-offs with varying queue depth while running SPDK NVMe driver vs. Kernel NVMe driver.

Results in the table represent performance in IOPS and average latency for the SPDK NVMe driver and Linux Kernel NVMe driver. We limited both the SPDK and Linux NVMe driver to use the same number of CPU Cores.

Test Workloads: We use the following Random Read/Write mixes

- 4KB 100% Random Read
- 4KB 100% Random Write
- 4KB Random 70% Read 30% Write

Table 14: SPDK NVMe BDEV Latency Test at different Queue Depths configuration

Item	Description
Test case	Test SPDK NVMe BDEV Latency Test at different Queue Depths
Test configuration	**FIO Version:** fio-3.19 **Number of CPU Cores:** 1 **Number of NVMe SSDs:** 1

Linux Kernel io_uring NVMe block device configuration	`echo 0 > /sys/block/nvme0n1/queue`
	`echo 0 > /sys/block/nvme0n1/rq_affinity`
	`echo 2 > /sys/block/nvme0n1/nomerges`
	`echo -1 > /sys/block/nvme0n1/io_poll_delay`

FIO configuration (common part)	`[global]`
	`direct=1`
	`thread=1`
	`time_based=1`
	`norandommap=1`
	`group_reporting=1`
	
	`rw={randread | randwrite | randrw}`
	`rwmixread={100 | 0 | 70}`
	`bs=4096`
	`runtime=240`
	`ramp_time=60`
	`numjobs=1`

FIO configuration (SPDK specific)	`[global]`
	`ioengine=spdk_bdev`
	`spdk_conf=/tmp/bdev.conf`

	[filename0]
	iodepth={6, 12, 24, 48, 96, 192, 512, 768, 1536, 3072, 4608}*
	cpus_allowed=0
	filename=Nvme0n1
	…
	filename=Nvme5n1
	
	[filename1]
	iodepth={6, 12, 24, 48, 96, 192, 512, 768, 1536, 3072, 4608}*
	cpus_allowed=21
	filename=Nvme6n1
	…
	filename=Nvme11n1
	
	[filename2]
	iodepth={6, 12, 24, 48, 96, 192, 512, 768, 1536, 3072, 4608}*
	cpus_allowed=22
	filename=Nvme12n1
	…
	filename=Nvme17n1
	
	[filename3]
	iodepth={6, 12, 24, 48, 96, 192, 512, 768, 1536, 3072, 4608}*
	cpus_allowed=23
	filename=Nvme18n1
	…
	filename=Nvme23n1
	
	* - actual iodepth parameter value used in test; this was multiplied by the number of "filename" objects in job section to achieve desired iodepth value per NVMe SSD (e.g. iodepth=3072 in this case is iodepth=512 per SSD)

FIO configuration (Linux Kernel common)	[global]
	ioengine={libaio | io_uring}
	
	[filename0]
	iodepth={6, 12, 24, 48, 96, 192, 512, 768, 1536, 3072, 4608}*
	cpus_allowed=21
	filename=/dev/nvme0n1
	…
	filename=/dev/nvme5n1
	
	[filename1]
	iodepth={6, 12, 24, 48, 96, 192, 512, 768, 1536, 3072, 4608}*
	cpus_allowed=22
	filename=/dev/nvme6n1
	…
	filename=/dev/nvme11n1
	
	[filename2]
	iodepth={6, 12, 24, 48, 96, 192, 512, 768, 1536, 3072,

	4608}*
	cpus_allowed=0
	filename=/dev/nvme12n1
	…
	filename=/dev/nvme17n1
	
	[filename3]
	iodepth={6, 12, 24, 48, 96, 192, 512, 768, 1536, 3072, 4608}*
	cpus_allowed=23
	filename=/dev/nvme18n1
	…
	filename=/dev/nvme23n1
	
	* - actual iodepth parameter value used in test; this was multiplied by the number of "filename" objects in job section to achieve desired iodepth value **per SSD** (e.g. iodepth=3072 in this case is iodepth=512 per SSD)

FIO configuration (Linux Kernel io_uring specific)	[global]
	fixedbufs=1
	hipri=1
	registerfiles=1
	sqthread_poll=1

4KB Random Read Results

Table 15: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io_uring polling (4KB Random Read, 24 NVMe SSDs, 4 CPU Cores)

	SPDK		Linux Kernel (Default libaio)		Linux Kernel (io_uring polling)	
QD	**IOPS (millions)**	**Avg. Lat. (usec)**	**IOPS (millions)**	**Avg. Lat. (usec)**	**IOPS (millions)**	**Avg. Lat. (usec)**
1	0.32	74	0.29	84	0.39	76
2	0.64	75	0.60	80	0.74	78
4	1.26	76	1.14	84	1.37	83
8	2.40	80	1.69	113	2.36	97
16	4.35	88	1.68	229	3.19	146
32	7.20	105	1.68	457	3.47	264
64	9.89	149	1.70	903	3.92	474
128	10.01	292	1.69	1815	3.40	1074
256	8.17	586	1.67	3681	-	-
512	5.91	1856	1.59	7711	-	-

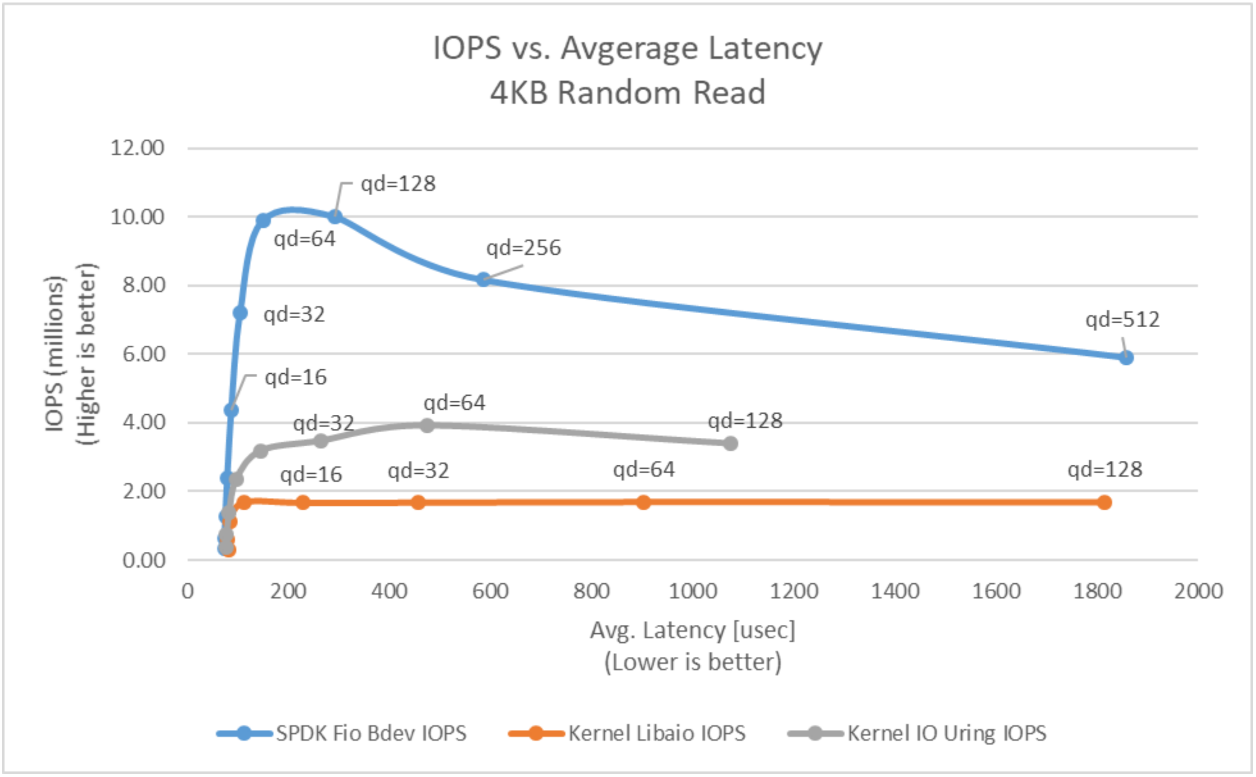

Figure 19: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io_uring polling (4KB Random Read, 24 NVMe SSDs, 4 CPU Cores)

4KB Random Write Results

Table 16: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io_uring polling (4KB Random Write, 24 NVMe SSDs, 4 CPU Cores)

	SPDK		Linux Kernel (Default libaio)		Linux Kernel (io_uring polling)	
QD	**IOPS (millions)**	**Avg. Lat. (usec)**	**IOPS (millions)**	**Avg. Lat. (usec)**	**IOPS (millions)**	**Avg. Lat. (usec)**
1	2.92	8	1.57	15	2.21	4
2	4.54	10	1.64	29	2.79	7
4	6.05	14	1.51	65	2.86	13
8	7.45	22	1.59	121	3.33	23
16	8.54	36	1.63	235	3.55	42
32	8.65	70	1.58	486	4.04	73
64	5.63	202	1.45	1071	3.98	148
128	6.56	361	1.47	2083	3.40	345
256	5.00	977	1.43	4292	-	-
512	2.60	4251	1.41	8698	-	-

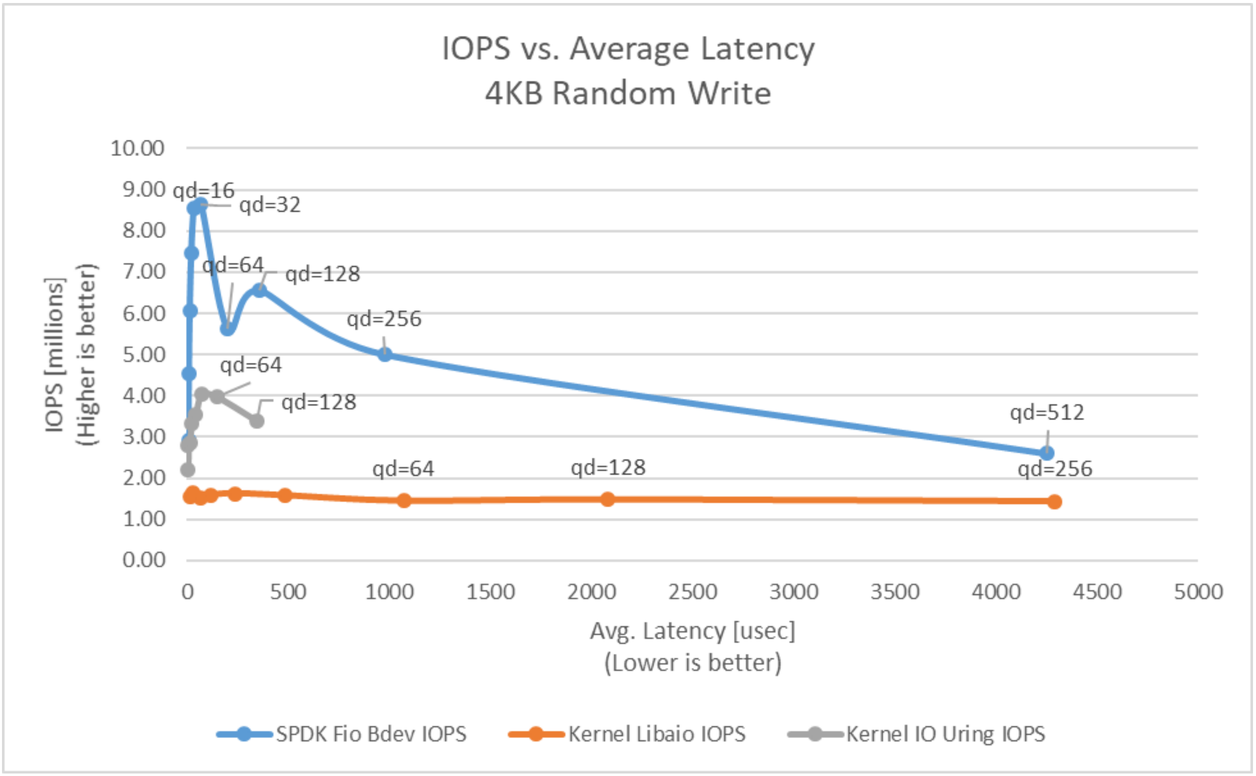

Figure 20: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io_uring polling (4KB Random Write, 24 NVMe SSDs, 4 CPU Cores)

4KB Random 70%/30% Read/Write Results

Table 17: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io_uring polling (4KB 70/30 Random Read/Write, 24 NVMe SSDs, 4 CPU Cores)

	SPDK		Linux Kernel (Default libaio)		Linux Kernel (io_uring polling)	
QD	**IOPS IOPS (millions)**	**Avg. Lat. (usec)**	**IOPS (millions)**	**Avg. Lat. (usec)**	**IOPS (millions)**	**Avg. Lat. (usec)**
1	0.43	55	0.39	62	0.51	19
2	0.82	58	0.75	63	0.94	20
4	1.44	66	1.31	73	1.51	25
8	2.41	79	1.65	116	2.17	35
16	3.60	106	1.65	233	2.82	54
32	5.47	139	1.64	468	3.58	87
64	7.57	199	1.62	951	3.71	166
128	8.60	332	1.58	1948	3.04	401
256	7.14	736	1.53	4026	-	-
512	5.20	2106	1.49	8249	-	-

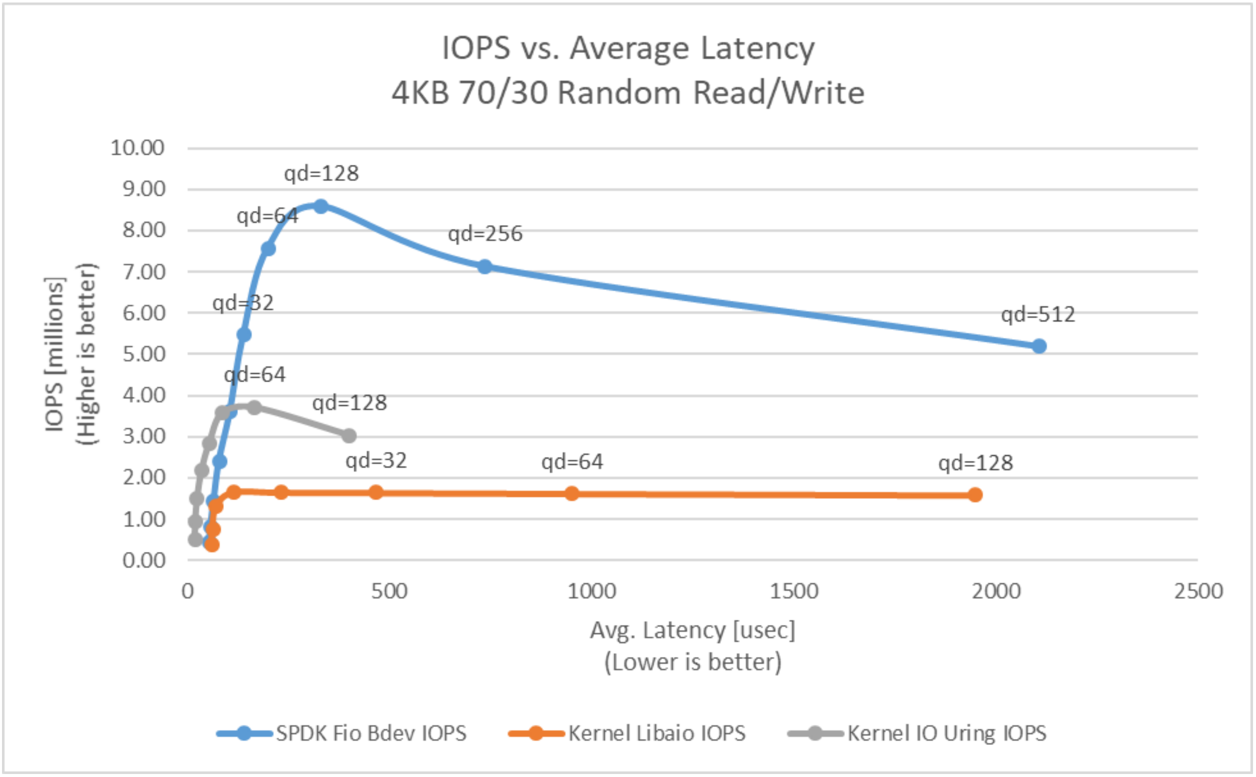

Figure 21: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io_uring polling (4KB 70/30 Random Read/Write, 24 NVMe SSDs, 4 CPU Cores)

Conclusions

1. In all workloads Kernel libaio ioengine achieved maximum performance of up to 1.6M IOPS with 4 CPU cores and was unable to saturate platforms NVMe disks or PCIe switches throughput. Peak performance was reached at QD=8 for Random Read and Random Read/Write workloads and at QD=1 for Random Write workload. Beyond these queue depth values there was no IOPS improvement, but the latency increase.

2. SPDK NVMe BDEV fio plugin reached up to around 10 million IOPS for Random Read workload at Queue Depth = 64, 128. This is similar to result measured in Test Case 2 - I/O Cores Scaling using Bdevperf.

3. SPDK NVMe BDEV fio plugin reached up to around 8.5 million IOPS for Random Read/Write workload at Queue Depth = 128, which is similar to result measured in Test Case 2 - I/O Cores Scaling using Bdevperf.

4. The results for the Random Write workload exceeded what the platforms NVMe SSDs are capable of (around 4.8M IOPS). This is probably due to a not perfect preconditioning process, which wears off over time. However, these results were repeatable and still show SPDK's high scalability with increase in the I/O requests.

5. For all workloads (when running SPDK NVMe BDEV fio plugin) with increasing queue depth, after reaching peak performance there is a noticeable performance drop. The reason for this degradation is still under investigation.

6. The Kernel io_uring engine reached a performance peaks of 3.7-4.0 million IOPS at Queue Depth = 64 for all workloads when we configured 6 NVMe SSDs per fio job, and similarly to SPDK NVMe BDEV fio plugin starts to drop beyond this Queue Depth value. However, when we looked at htop we noticed that io_uring was using 8 CPU cores, because when we configured the sqthread_poll parameter to eliminate system calls io_uring starts a special kernel thread that polls the shared submission queue for I/O added by the fio thread.

Therefore, in terms of CPU efficiency we measured up to 500K IOPS/Core for io_uring vs up to 2.5M IOPS/Core for the SPDK NVMe bdev. The Submission Queue Polling blog provides more information about how to eliminate system calls with io_uring. .

7. We were unable to run tests using Kernel io_uring ioengine with Queue Depth = 256 and 512 when we configured 6 NVMe SSDs per fio job. The reason for that was fio job configuration paired with limiting system settings. In this test 4 CPU cores were used, which in fio job configuration translates to 4 job sections, each with multiple "filename" arguments for target NVMe devices and an upscaled iodepth argument to match the number of devices. For example: a single job section was limited to a single cpu core using cpus_allowed argument; 6 NVMe devices were attached to this section using 6 "filename" arguments, and iodepth was set to iodepth=1536 (6*256). Queue depth of this value makes the test impossible to run because of fio "registerfiles" option (which is required to enable polling). When "registerfiles" is used the test fails because of the default UIO_MAXIOV limitation in sys/uio.h header file.

Summary

1. SPDK NVMe BDEV Block Layer using SPDK Bdevperf benchmarking tool can deliver up to 5.7 million IOPS on a single Intel® Xeon® Gold 6230N with Turbo Boost enabled.

2. The SPDK NVMe BDEV IOPS scale linearly with addition of CPU cores. We demonstrated up to 10.4 million IOPS on just 3 CPU cores (Intel® Xeon® Gold 6230N with Turbo Boost enabled).

3. The SPDK NVMe BDEV has lower QD=1 latency than the Linux Kernel NVMe block driver for small (4KB) blocks.

    a. SPDK BDEV latency was 14% lower than Linux Kernel Libaio latency for Random Read workload.

    b. SPDK BDEV latency was 30% lower than Linux Kernel Libaio latency for Random Write workload.

    c. SPDK BDEV latency was 12% lower than Linux Kernel io_uring latency for Random Read workload.

    d. SPDK BDEV latency was 29% lower than Linux Kernel io_uring latency for Random Write workload.

4. SPDK NVMe Bdev Fio reaches up to 10 million IOPS and keeping average latency less than 300usec while using 4 CPU cores. With the same fio workloads Kenrel io_uring and Kernel libaio reach up to 4 million and 1.7 million IOPS respectively.

List of tables

List of figures

References

[1] Damien Le Moal, "I/O Latency Optimization with Polling", Vault – Linux Storage and Filesystem Conference – 2017, May 22nd, 2017.

Notices & Disclaimers

§