SPDK NVMe BDEV Performance Report Release 22.01

Testing Date: March 2022

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 (5.15.7-200.fc35.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: Hardware setup configuration

Item	Description
Server Platform	Supermicro SYS-220U-TNR 
Motherboard	Server board **X12DPU-6**
CPU	2 CPU sockets, Intel(R) Xeon(R) Gold 6348 CPU @ 2.60GHz Number of cores 28 per socket, number of threads 56 per socket Both sockets populated Microcode: 0xd0002e0
Memory	16 x 32GB SK Hynix DDR4 HMA84GR7DJR4N-XN Total 512 GBs Memory channel population:
Operating System	Fedora 35
BIOS	1.1a
Linux kernel version	5.15.7-200.fc35.x86_64
SPDK version	SPDK 22.01
Fio version	3.28
Storage	**OS:** 1x 250GB Crucial CT250MX500SSD1 **Storage**: 22x Kioxia® KCM61VUL3T20 3.2TBs (FW: 0105) (10 on CPU NUMA Node 0, 12 on CPU NUMA Node 1)

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
CPU1_DIMM_F1	CPU2_DIMM_F1
CPU1_DIMM_G1	CPU2_DIMM_G1
CPU1_DIMM_H1	CPU2_DIMM_H1

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 24 NVMe drives. Therefore, PCIe switches were used to fan out PCIe lanes to NVMe SSDs on the riser cards. For more information on PCIe capabilities of the platform please refer to its technical specification.

Table 3: Test platform NVMe storage setup

<table>
<tr><th>Item</th><th colspan="2">Description</th></tr>
<tr><td>PCIe Riser cards</td><td colspan="2">“Ultra” Riser Card: AOC-2UR68G4-i2XT
• PCIe Slot 1 – x16, CPU2
• PCIe Slot 2 – x8, CPU2
• PCIe Slot 3 – x8, CPU2
Right-facing riser card: RSC-WR-6
• PCIe Slot 4 – x16, CPU1
Left-facing riser card: RSC-W2-66G4
• PCIe Slot 5 – x16, CPU2
• PCIe Slot 7 – x16, CPU1
More information can be found in SYS-220U-TNR manual document.</td></tr>
<tr><td>PCIe Retimer cards</td><td colspan="2">3 x AOC-SLG4-4E4T
Installed in:
○ PCIe Retimer 1: RSC-WR-6, PCIe Slot 4 (using CPU1 PCIe Lanes)
○ PCIe Retimer 2: AOC-2UR68G4-i2XT, PCIe Slot 1 (using CPU2 PCIe Lanes)
○ PCIe Retimer 3: RSC-W2-66G4, PCIe Slot 5 (using CPU2 PCIe Lanes)</td></tr>
<tr><td rowspan="5">NVMe Drives distribution across the system</td><td>Nvme0 – 5</td><td>Motherboard ports (CPU1 PCIe Lanes)</td></tr>
<tr><td>Nvme6 – 9</td><td>Motherboard ports (CPU2 PCIe Lanes)</td></tr>
<tr><td>Nvme9 – 13</td><td>PCIe Retimer 1 (CPU1 PCIe Lanes)</td></tr>
<tr><td>Nvme14 - 17</td><td>PCIe Retimer 2 (CPU2 PCIe Lanes)</td></tr>
<tr><td>Nvme18 - 21</td><td>PCIe Retimer 3 (CPU2 PCIe Lanes)</td></tr>
</table>

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 actual read operation on the device. Therefore, prior to running each performance test case we preconditioned the SSDs by writing 128K blocks sequentially across the namespace’s full LBA range twice to ensure the controller accesses the NAND media for each subsequent I/O. 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 60 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 5.15.7-200 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 NVMe driver and binds them to a userspace NVMe driver instead. This allows a userspace application to directly access the device and its queues 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 something else, for example Ceph RBD or malloc ramdisk block device. 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_set_options",
            "params": {
              "bdev_io_pool_size": 65535,
              "bdev_io_cache_size": 2048,
              "bdev_auto_examine": true
            }
        },
        {
          "method": "bdev_nvme_attach_controller",
          "params": {
            "trtype": "PCIe",
            "name": "Nvme0",
            "traddr": "0000:1a:00.0",
            "bdev_io_cache_size": 2048
          }
        },

        [...]

        {
          "method": "bdev_nvme_attach_controller",
          "params": {
            "trtype": "PCIe",
            "name": "Nvme23",
            "traddr": "0000:df:00.0",
            "bdev_io_cache_size": 2048
          }
        }
      ]
    }
  ]
```

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_json_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_json_conf=/tmp/bdev.conf

rw=randread
rwmixread=70
bs=4096
numjobs=1
runtime=300
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/Core 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:

- 4KiB 100% Random Read
- 4KiB 100% Random Write
- 4KiB 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 22 Kioxia KCM61VUL3T20 NVMe devices and decrease the number of SSDs on each subsequent run.
    - This shows us the IOPS scaling as we add SSDs till the maximum IOPS/Core is reached.
    - Starting with 22 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-3.28 **Number of NVMe SSDs**: {1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 14, 16, 18, 20, 22} **SPDK_BDEV_IO_CACHE_SIZE** changed from 256 to 2048 (using bdev_set_options RPC call).
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_json_conf=bdev.conf`

	gtod_reduce=1 direct=1 thread=1 norandommap=1 time_based=1 ramp_time=60s runtime=300s bs=4k numjobs=1
(Random read and mixed workloads)	rw={randread, randrw} rwmixread={100,70,0} iodepth={128, 192, 256}
(Random write workload)	rw=randwrite rwmixread=0 iodepth={32,64}

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.

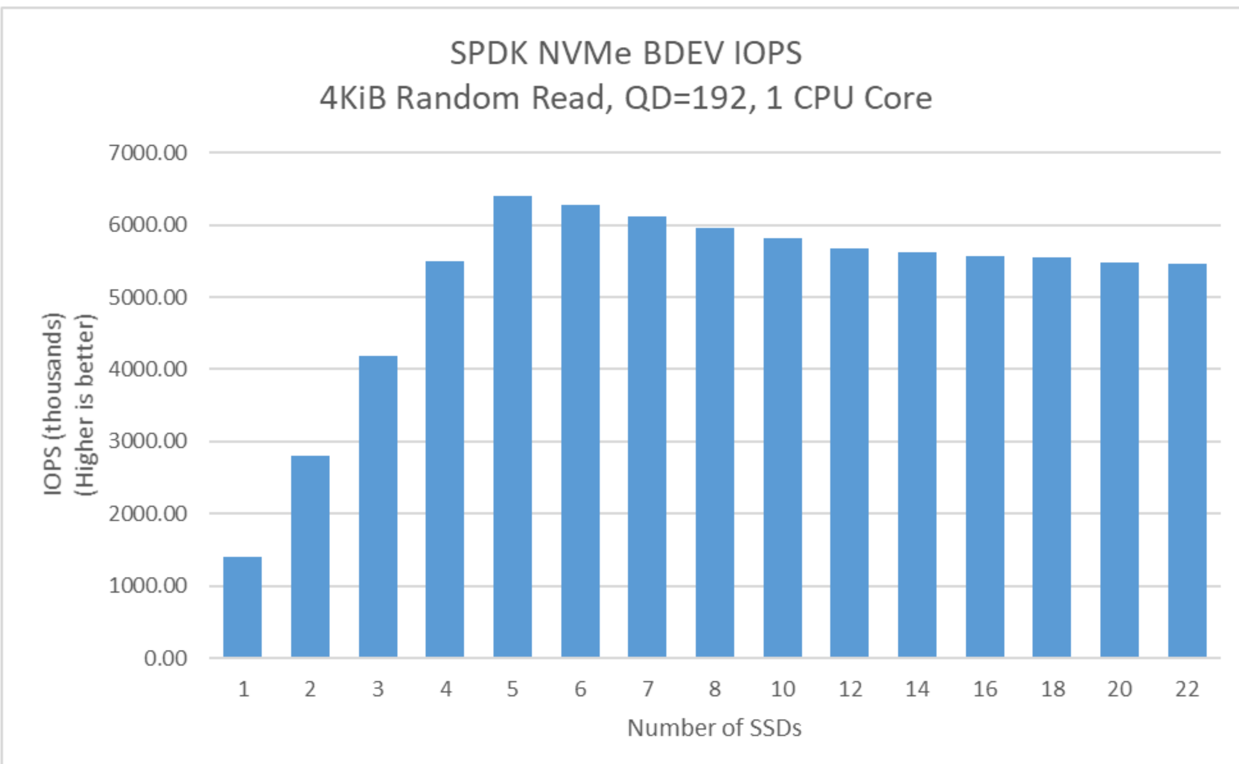

Figure 3: SPDK NVMe BDEV IOPS scalability with addition of SSDs (4KiB Random Read, 1CPU Core, QD=192, using bdevperf tool)

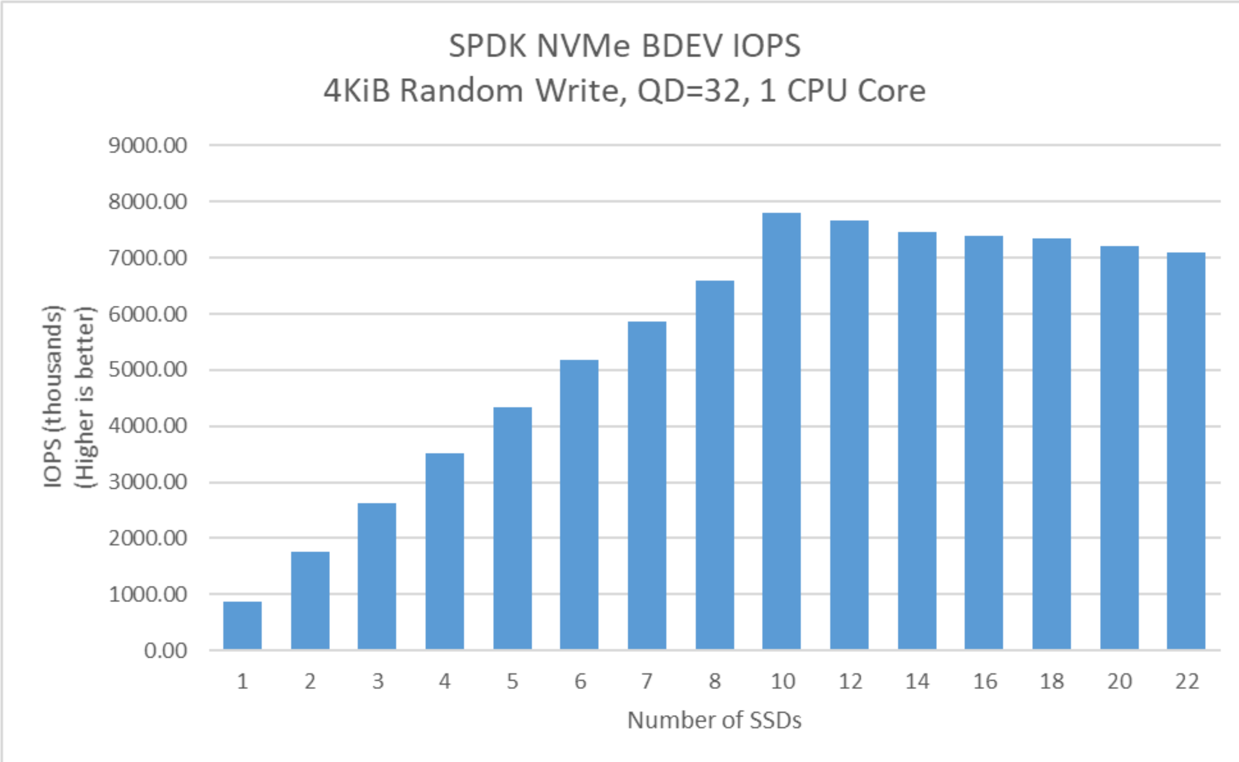

Figure 4: SPDK NVMe BDEV IOPS scalability with addition of SSDs (4KiB Random Write, 1CPU Core, QD=32, using bdevperf tool)

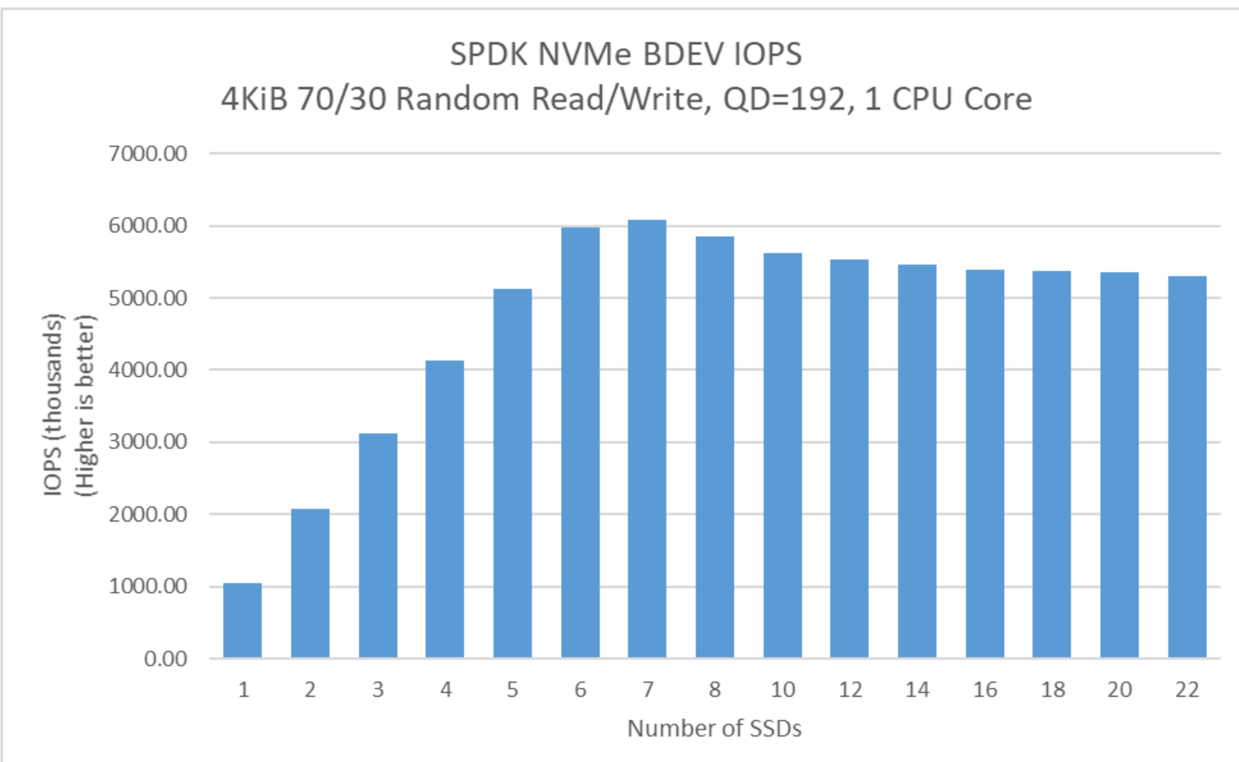

Figure 5: SPDK NVMe BDEV IOPS scalability with addition of SSDs (4KiB 70/30 Random Read/Write, 1CPU Core, QD=192, using bdevperf tool)

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=4KiB, 1 CPU Core)

Workload	SDPK Fio BDEV Plugin (IOPS, thousands)	SPDK Bdevperf (IOPS, thousands)	Performance gain
4KiB Random Read, QD=192, 5 SSDs	3615.91	6401.72	77.0%
4KiB Random Write, QD=32, 10 SSDs	2471.55	7805.63	215.8%
4KiB 70/30 Random Read/Write, QD=192, 6 SSDs	3176.01	5983.65	88.4%

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 110.7% 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 approximately 8.7% and 6.3% more CPU utilization for Random Read and Random Write workloads. 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 12 I/O cores for Random Read workload and 1 CPU core for every 15 I/O cores for Random Write workload. Comparison was done using SPDK Bdevperf and Nvmeperf test tools.

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

Workload	SPDK Bdevperf (IOPS, thousands)	SPDK Nvmeperf (IOPS, thousands)	Performance gain
4KiB Random Read, QD=192, 5 SSDs	6401.72	6958.93	8.7%
4KiB Random Write, QD=32, 10 SSDs	7805.63	8303.38	6.3%

Conclusions

1. The SPDK NVMe block device module adds approximately 8.7% and 6.3% overhead compared to using only the SPDK NVMe Polled-Mode Driver without the block device module for Random Read and Random Write workloads respectively.

2. Performance scales linearly with addition of NVMe SSDs up to 5 NVMe SSDs for Random Read workload, reaching around 6.4 million IOPS.

3. Performance scaling is linear for Random Write workload up to 10 NVMe SSDs, reaching around 7.8 million IOPS.

4. Performance scales linearly with addition of NVMe SSDs up 6 SSDs for Random Read/Write workloads, reaching around 5.98 million IOPS.

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

6. The IOPS for the 4 KiB Random Write workload exceeded the expected NVMe SSDs maximum throughput. We suspect this is due to a not perfect preconditioning process, which wears off over time. The results, however, were repeatable for a number of test runs.

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, 5 and 6.

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

- 4KiB 100% Random Read
- 4KiB 100% Random Write
- 4KiB 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, 6 **Number of NVMe SSDs:** 5 per each CPU Core used in test, up to maximum of 22 NVMe SSDs **NUMA optimization:** CPUs for test were selected in a way to match NVMe drives distribution across platform NUMA nodes.

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 (4KiB 100% Random Read IOPS at QD=192; 4KiB 100% Random Write IOPS at QD=32; 4KiB 70/30 Random Read/Write IOPS at QD=192)

		IOPS (thousands)		
CPU Cores	**NVMe SSDs**	**Random Read QD=192**	**Random Write QD=32**	**70/30 Random Read/Write QD=192**
1	**5**	6419.58	2184.05	4775.30
2	**10**	12995.87	5635.65	9899.66
3	**15**	18970.86	9616.15	14944.27
4	**20**	25648.86	13950.19	20046.09
5	**22**	28900.08	16766.80	22401.81
6	**22**	30065.16	18301.09	22697.11

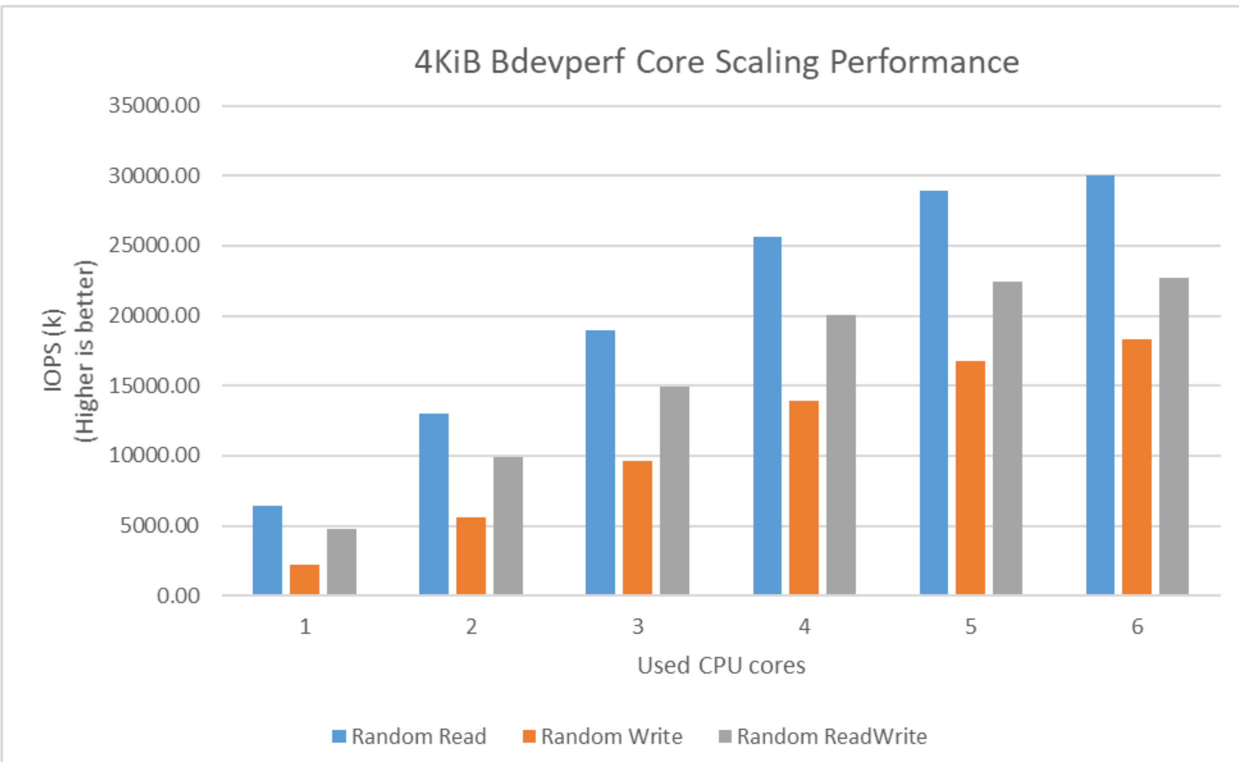

Figure 6: SPDK NVMe BDEV I/O Cores Scalability (4KiB 100% Random Read IOPS at QD=192; 4KiB 100% Random Write IOPS at QD=32; 4KiB 70/30 Random Read/Write IOPS at QD=192)

Conclusions

1. The IOPS for 4 KiB Random Read workload scales up linearly with addition of I/O cores until NVMe drives used for test are saturated.

2. The IOPS for 4 KiB Random Read/Write workloads scale up linearly with the addition of I/O cores up to 5 I/O cores. Increasing the number of cores to 6 does not result in performance improvement.

3. The IOPS for the 4 KiB Random Write workload scale up linearly. The IOPS exceeded the expected NVMe SSDs throughput for this workload which is about 7.7M 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.

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:

- 4KiB 100% Random Read
- 4KiB 100% Random Write

Important note: For 21.01 benchmark tests we have been unable to successfully run tests for Kernel io_uring engine with sqthread_pool option enabled when workload was using write path and Queue Depth was set to 1. Because of this, Random Write QD=1 workload was run without sqthread_poll option enabled. This is caused by some unidentified bug, probably in Fio or Kernel itself. For more information, please see the issue for this problem on Github.

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].

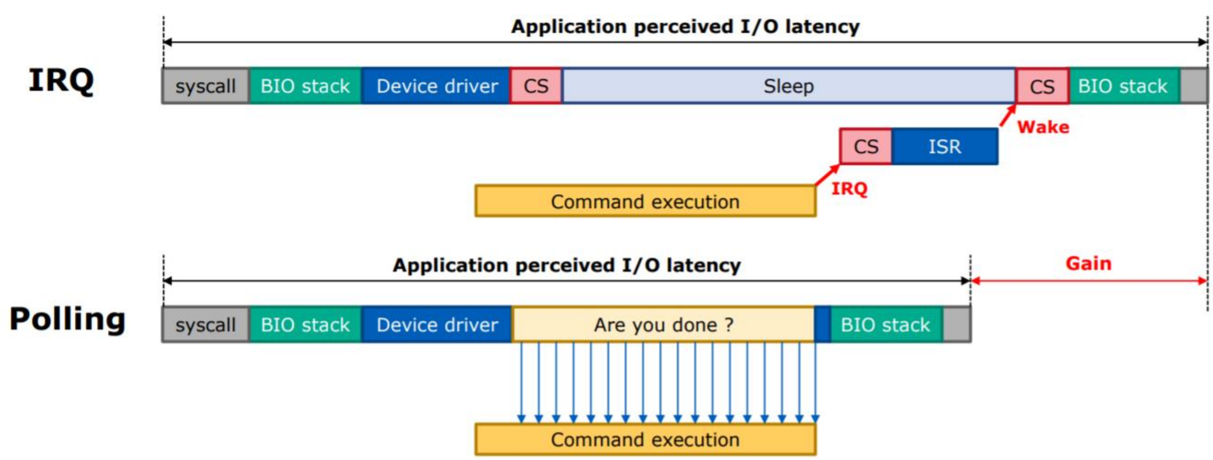

Figure 7: Linux Block Layer I/O Optimization with Polling. Source [1]

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 polling mode was not used for testing in this document.

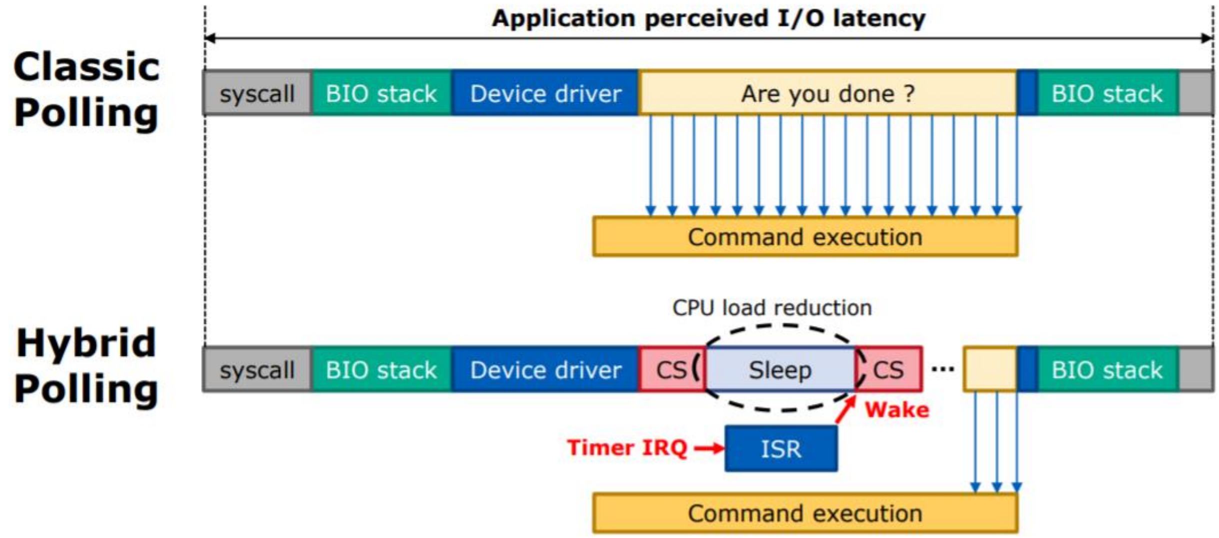

Figure 8: Linux Block I/O Classic and Hybrid Polling latency breakdown. Source [1]

The data in tables and charts compares the I/O latency for a various 4KiB 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 (4KiB Random Read, QD=1, runtime=900s)

Latency metrics (usec)	SDPK Fio BDEV Plugin	Linux Kernel (libaio)	Linux Kernel (io_uring)
Average	73.119	76.321	74.113
P90	82.432	84.48	83.456
P99	83.456	85.504	84.48
P99.99	148.48	150.528	148.48
Stdev	6.992	6.995	10.045
Average submission latency	0.161	0.933	0
Average completion latency	72.957	75.306	74.073

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

Latency metrics (usec)	SDPK Fio BDEV Plugin	Linux Kernel (Default libaio)	Linux Kernel (io_uring)
Average	5.131	8.129	6.435
P90	6.048	7.2	6.56
P99	6.112	7.264	6.752
P99.99	8.64	10.048	8.896
Stdev	1.205	1.562	2.249
Average submission latency	0.177	0.924	0
Average completion latency	4.953	7.129	6.376

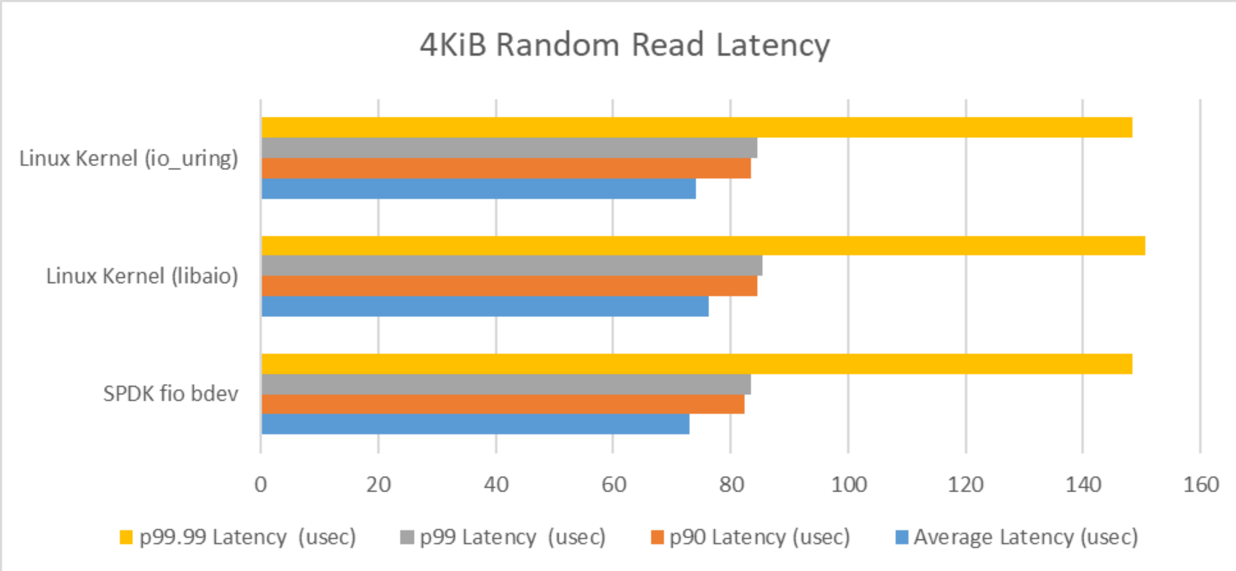

Figure 9: SPDK bdev vs Linux Kernel Latency comparison (4KiB Random Read)

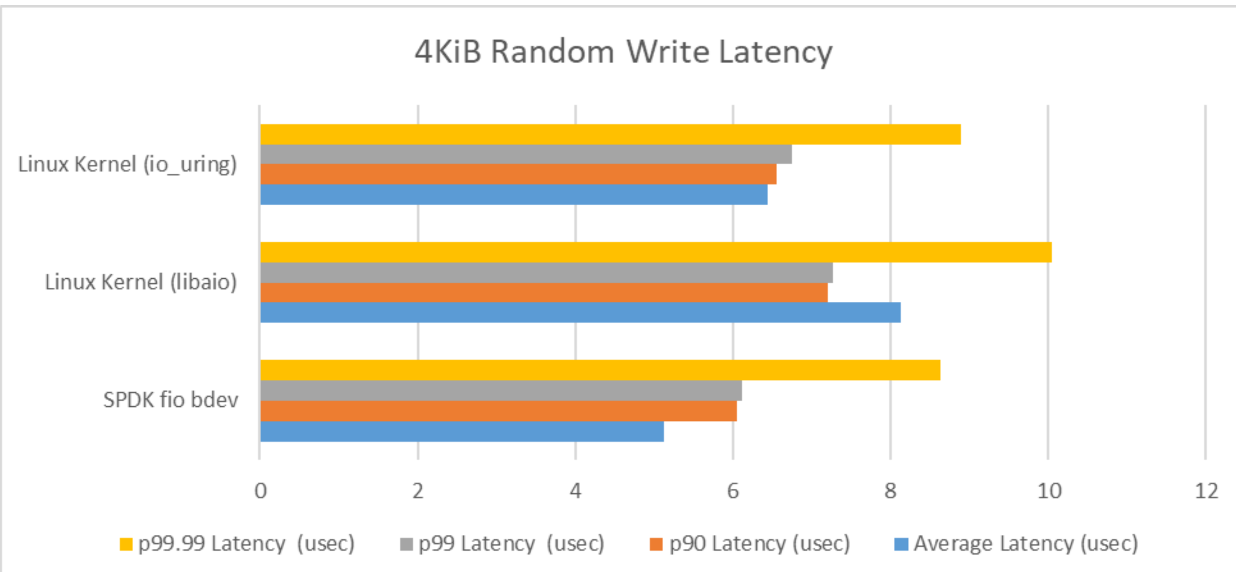

Figure 10: SPDK bdev vs Linux Kernel Latency comparison (4KiB Random Write)

Linux Kernel libaio Histograms

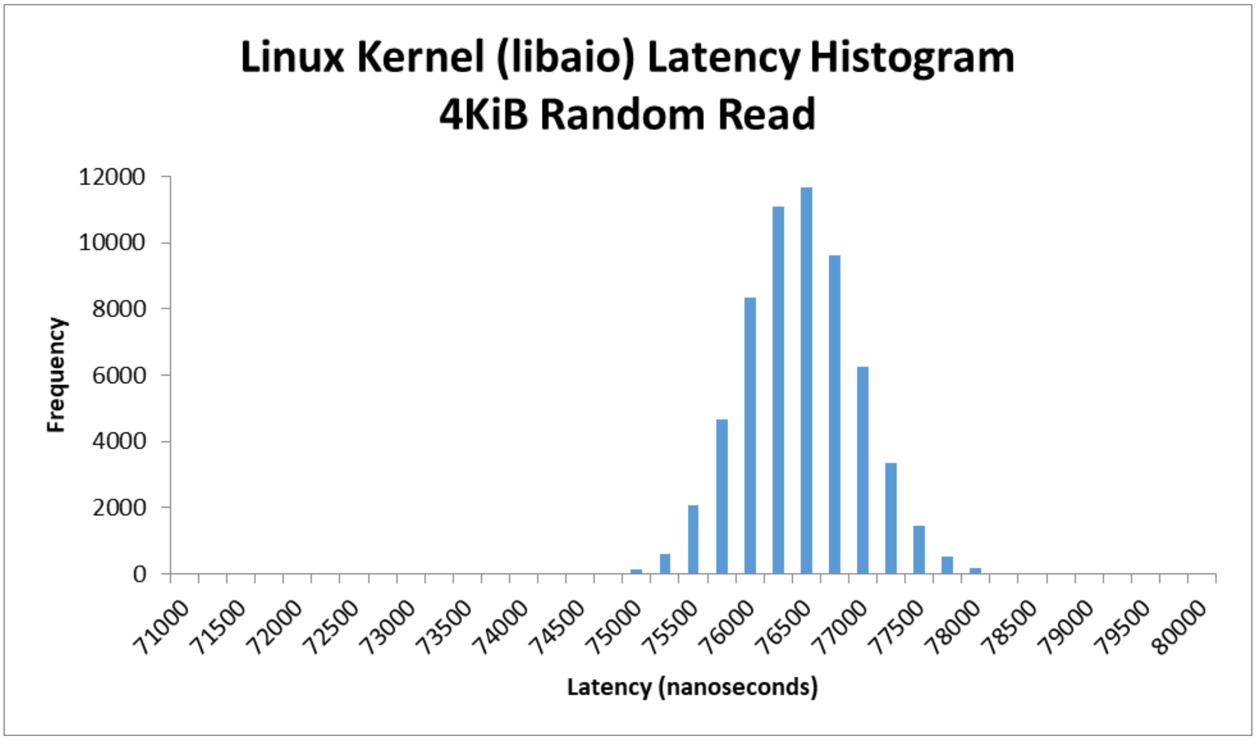

Figure 11: Linux Kernel (Default libaio) 4KiB Random Read Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec)

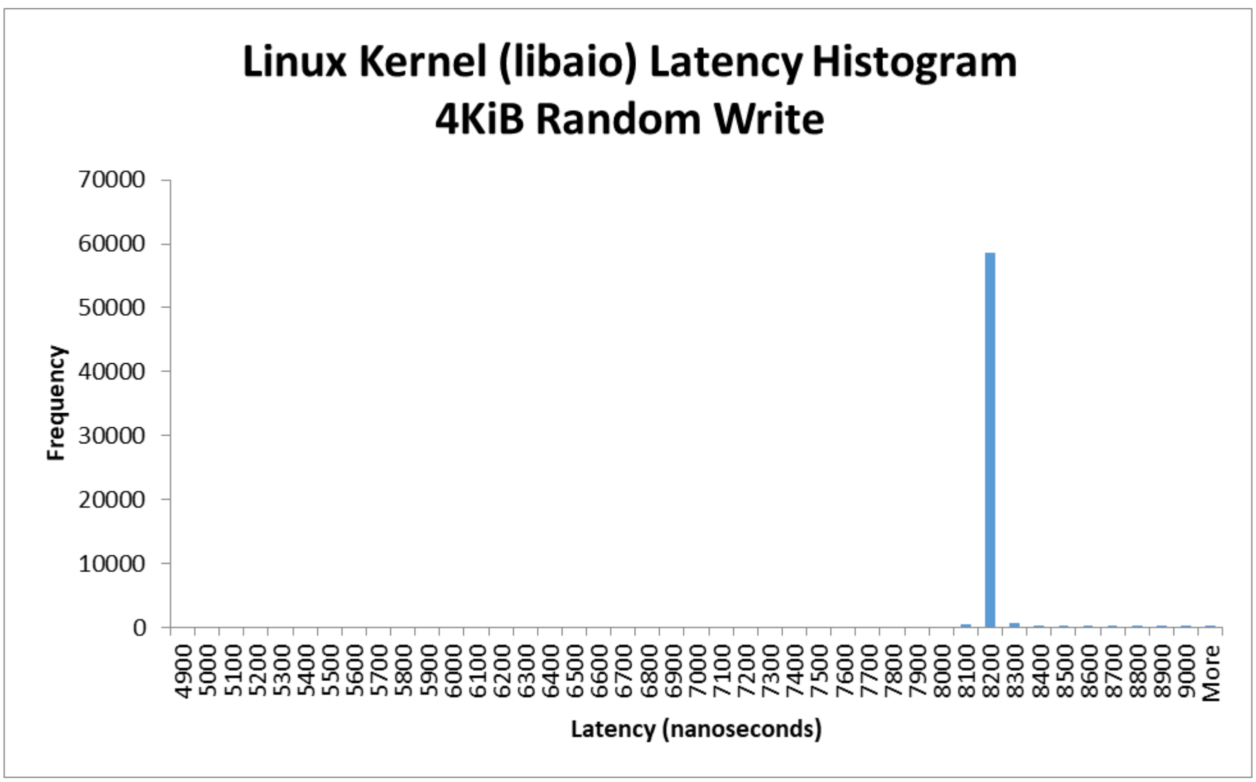

Figure 12: Linux Kernel (Default libaio) 4KiB Random Write Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec)

Linux Kernel io_uring Histograms

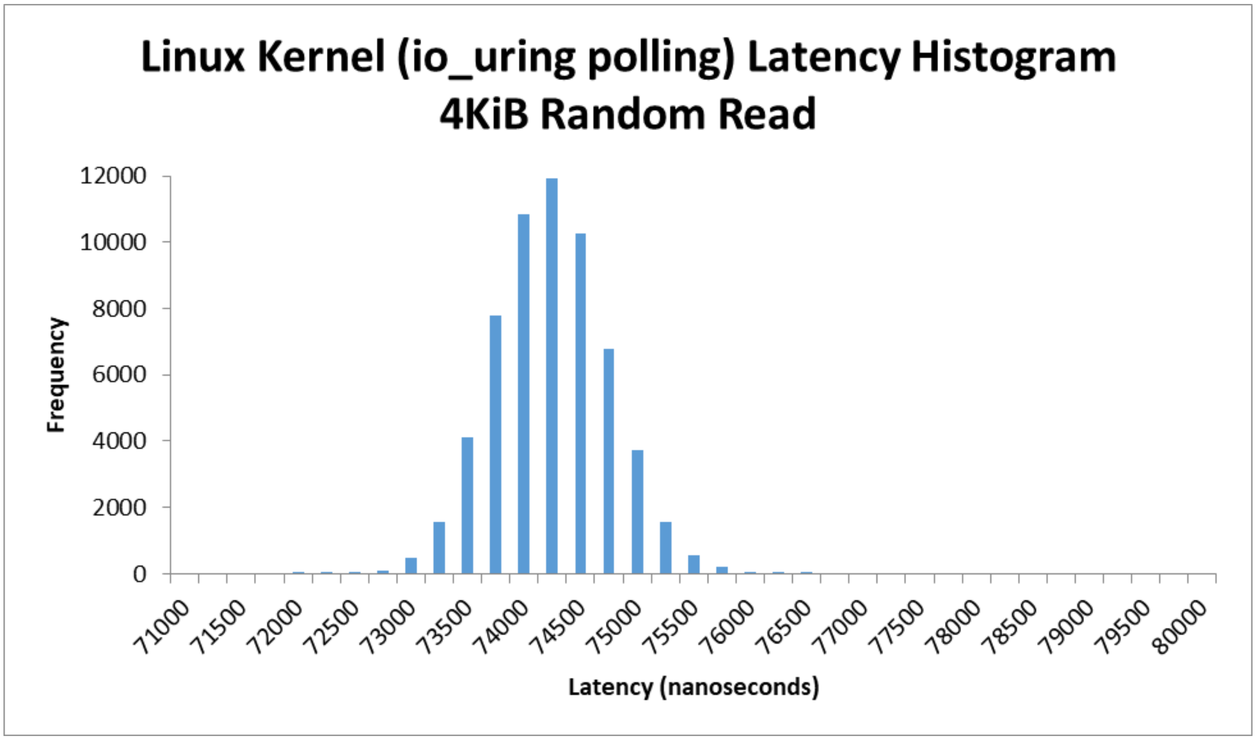

Figure 13: Linux Kernel (io_uring polling) 4KiB Random Read Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec)

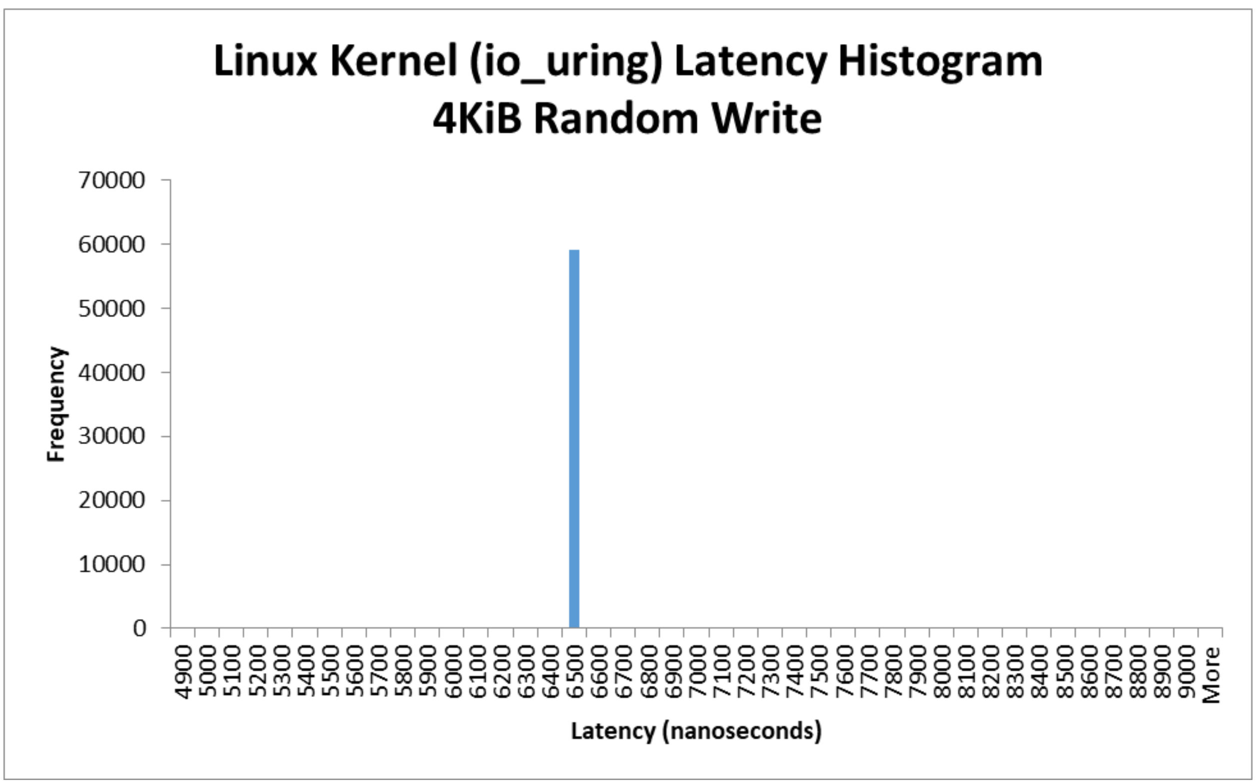

Figure 14: Linux Kernel (io_uring polling) 4KiB Random Write Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec)

SPDK FIO Bdev Histograms

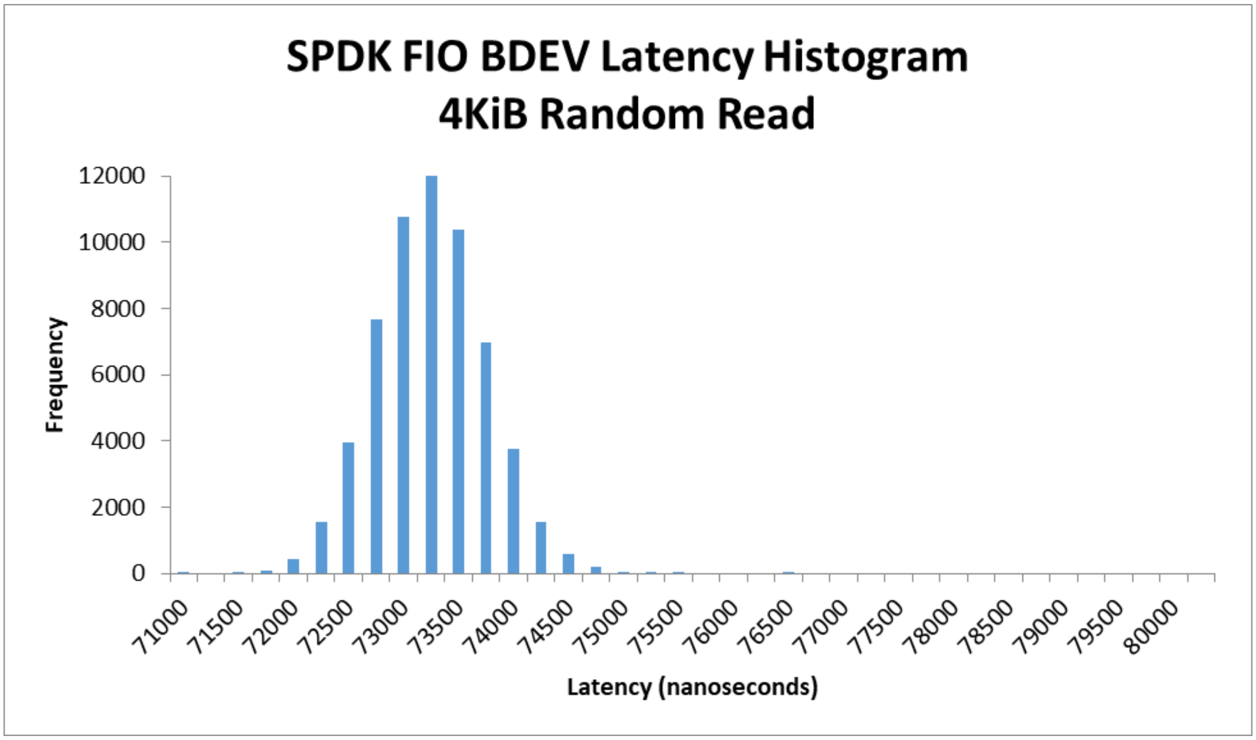

Figure 15: SPDK BDEV NVMe 4KiB Random Read Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec)

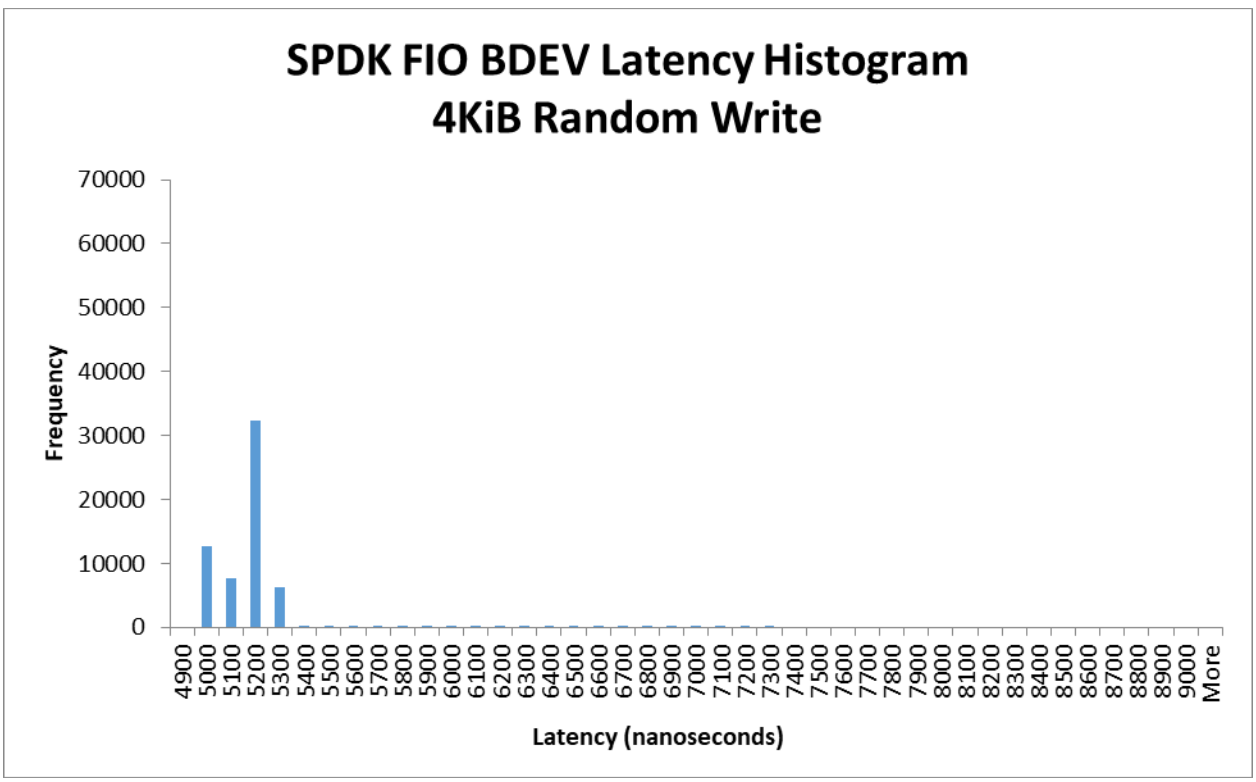

Figure 16: SPDK BDEV NVMe 4KiB Random Write Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec)

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 (4KiB 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	13636	73	13067	76	13448	74
2	27062	74	25924	77	26700	75
4	53838	74	51592	77	53090	75
8	106531	75	102014	78	104999	76
16	208566	76	199084	80	205111	78
32	398695	80	375210	85	390140	82
64	724825	88	499787	128	706757	90
128	1171975	109	500896	255	1108317	115
256	1400458	183	498937	513	1369478	187
512	1400572	365	499245	1025	1458476	351

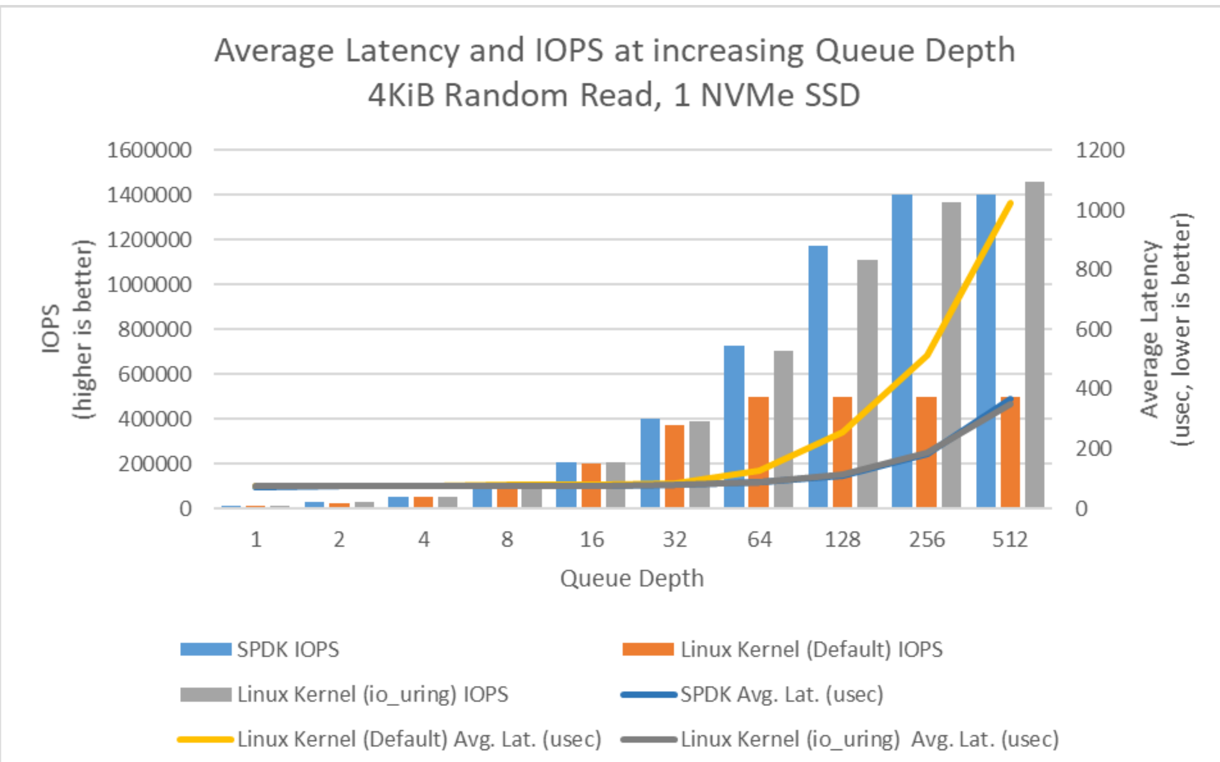

Figure 17: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io_uring polling (4KiB Random Read, 1 NVMe SSD, 1 CPU Core, Numjobs=1)

Table 13: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io_uring polling (4KiB Random Write, 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	193042	5	119221	8	154333	6
2	353539	5	222836	9	299880	6
4	677135	6	481814	8	540184	7
8	857840	9	483394	16	816246	10
16	785819	20	484533	33	804858	20
32	811626	39	483388	66	802629	40
64	801006	80	485423	132	808337	79
128	792944	161	483717	264	783661	163
256	758330	338	482649	530	762003	336
512	697820	736	481613	1063	713214	720

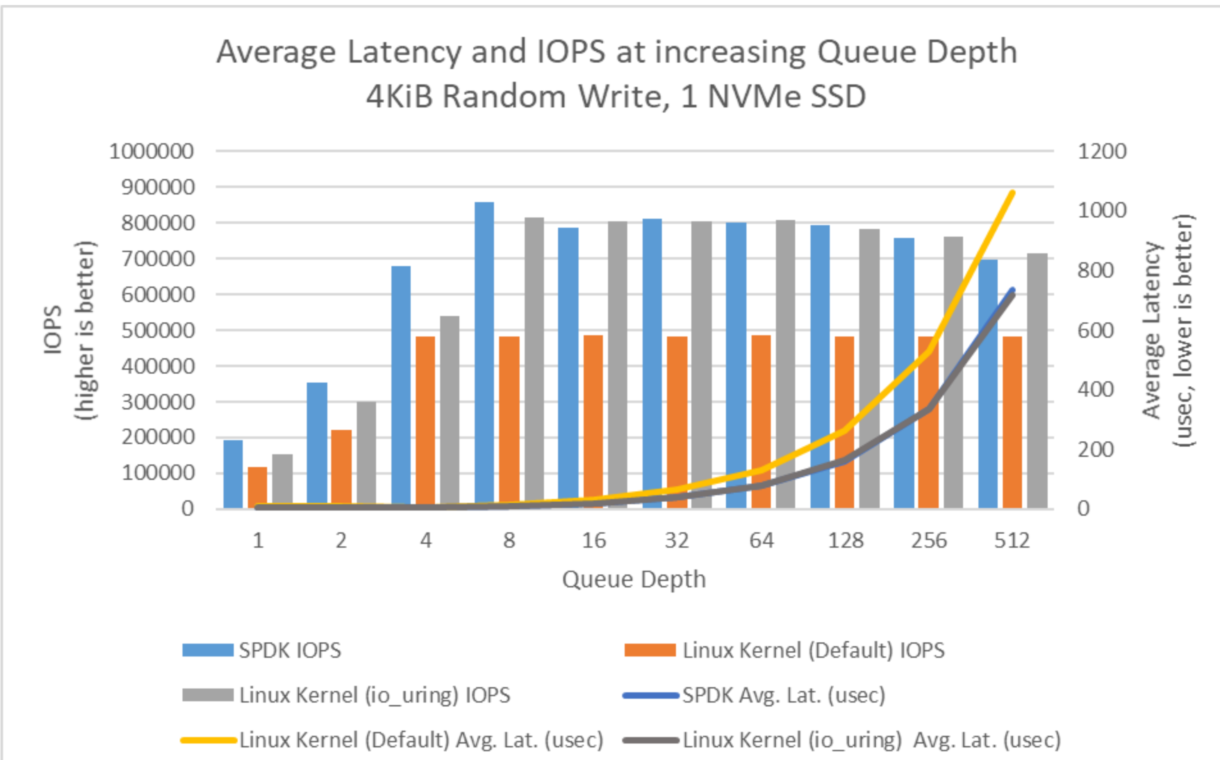

Figure 18: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io_uring polling (4KiB Random Write, 1 NVMe SSD, 1 CPU Core, Numjobs=1)

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 4.2% and 36.88% lower than Linux Kernel Libaio, for Random Read and Random Write workloads respectively.
3. SPDK NVMe Bdev average latency was up to approximately 1.3% and 20.3% lower than Linux Kernel io_uring for Random Read and Random Write workloads respectively.
4. Frequency buckets for 4KiB Random Write at QD=1 workload were so narrow that it was decided to present the results using 100ns as an interval unit for x-axis.
5. For 4KiB Random Read workload all test engines scaled linearly up to QD=32 queue depth. Beyond this value:
    a. SPDK NVMe Bdev scaling became non-linear and peaked at QD=256, reaching 1.4 million IOPS and saturating NVMe drive
    b. Kernel io_uring scaling became non-linear and peaked ad QD=512, reaching 1.4 million IOPS and saturating NVMe drive
    c. Kernel libaio peaked at QD=64 reaching approximately 500k IOPS. Increasing queue depth did not improve throughput.
6. For 4KiB Random Write workload all test engines scaled linearly up to QD=4 queue depth. Beyond this value performance improvement was none or negligible.
    a. SPDK NVMe Bdev and Kernel io_uring peaked at about 800k IOPS. Further increasing the queue depth resulted in minor performance degradation.
    b. Kernel libaio peaked at about 480k IOPS.

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 vs. Kernel NVMe block layers.

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

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

- 4KiB 100% Random Read
- 4KiB 100% Random Write
- 4KiB 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-3.28 **Number of CPU Cores:** 12 **Number of NVMe SSDs:** 22

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`

<table>
<tr><td></td><td><pre>[filename0]
iodepth={2, 4, 8, 16, 32, 64, 128, 256, 512, 1024}*
cpus_allowed=0
filename=Nvme0n1
filename=Nvme1n1

[filename1]
iodepth={2, 4, 8, 16, 32, 64, 128, 256, 512, 1024}*
cpus_allowed=1
filename=Nvme2n1
filename=Nvme3n1

[…]

[filename11]
iodepth={2, 4, 8, 16, 32, 64, 128, 256, 512, 1024}*
cpus_allowed=11
filename=Nvme20n1
filename=Nvme21n1

* - - actual iodepth parameter value used in test; this
was multiplied by the number of "filename" objects in
job section to achieve desired queue depth value per
NVMe SSD (e.g. QD=256 in this case is QD=128 per SSD)</pre></td></tr>
<tr><td>FIO configuration (Linux Kernel common)</td><td><pre>[global]
ioengine={libaio | io_uring}

[filename0]
iodepth={2, 4, 8, 16, 32, 64, 128, 256, 512, 1024}*
cpus_allowed=0
filename=/dev/nvme0n1
filename=/dev/nvme1n1

[filename1]
iodepth={2, 4, 8, 16, 32, 64, 128, 256, 512, 1024}*
cpus_allowed=1
filename=/dev/nvme2n1
filename=/dev/nvme3n1

[…]

[filename11]
iodepth={2, 4, 8, 16, 32, 64, 128, 256, 512, 1024}*
cpus_allowed=11
filename=/dev/nvme20n1
filename=/dev/nvme21n1

* - - actual iodepth parameter value used in test; this
was multiplied by the number of "filename" objects in
job section to achieve desired queue depth value per
NVMe SSD (e.g. QD=256 in this case is QD=128 per SSD)</pre></td></tr>
<tr><td>FIO configuration (Linux Kernel io_uring specific)</td><td><pre>[global]
fixedbufs=1
hipri=1
registerfiles=1
sqthread_poll=1</pre></td></tr>
</table>

4KiB Random Read Results

Table 15: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io_uring polling (4KiB Random Read, 22 NVMe SSDs, 12 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	0.30	73	0.29	77	0.30	74
2	0.59	74	0.57	77	0.59	75
4	1.18	74	1.13	78	1.17	75
8	2.34	75	2.21	79	2.31	76
16	4.56	77	4.21	83	4.51	78
32	8.69	80	5.49	128	8.52	82
64	15.68	89	5.78	243	14.29	98
128	24.70	111	5.77	488	18.65	151
256	29.25	188	5.77	975	20.97	268
512	28.27	391	5.76	1954	20.78	542

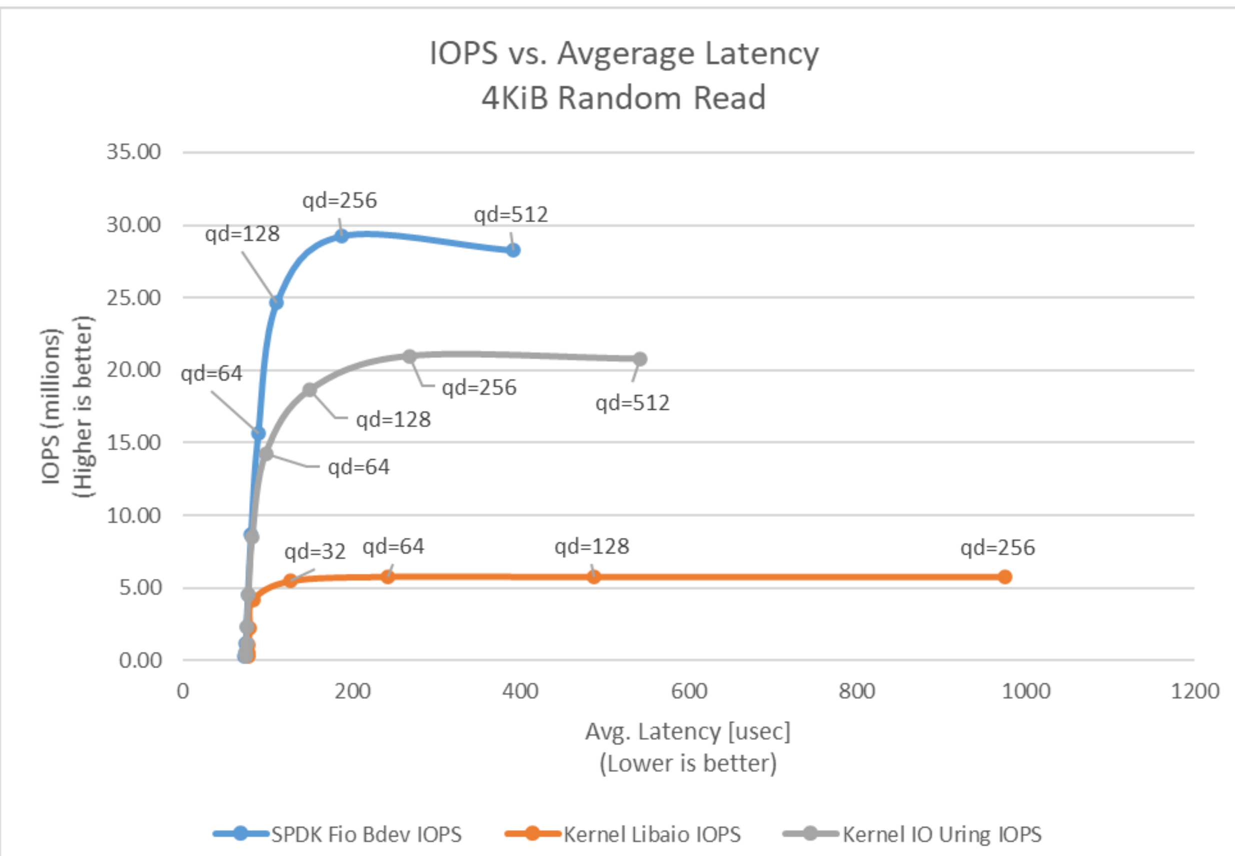

Figure 19: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io_uring polling (4KiB Random Read, 22 NVMe SSDs, 12 CPU Cores)

4KiB Random Write Results

Table 16: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io_uring polling (4KiB Random Write, 22 NVMe SSDs, 12 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	3.85	5	2.34	9	3.46	2
2	7.25	6	4.97	8	6.88	2
4	12.47	7	5.61	15	11.19	3
8	16.18	10	5.70	31	14.55	4
16	16.37	20	5.68	62	15.97	7
32	16.37	41	5.68	124	16.30	14
64	16.22	84	5.67	248	15.83	30
128	16.14	172	5.66	497	15.76	59
256	15.76	355	5.63	1000	15.78	357
512	15.34	731	5.55	2030	15.14	744

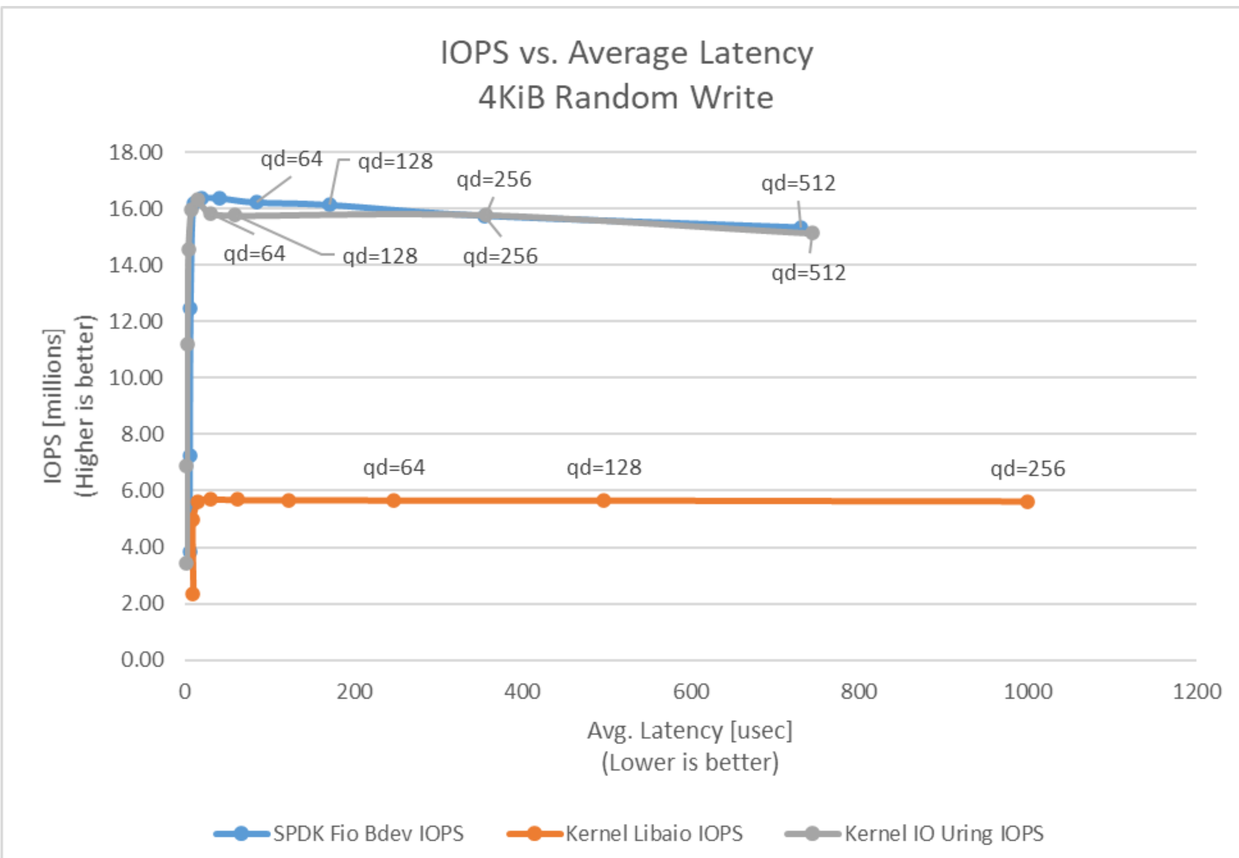

Figure 20: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io_uring polling (4KiB Random Write, 22 NVMe SSDs, 12 CPU Cores)

4KiB 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 (4KiB 70/30 Random Read/Write, 22 NVMe SSDs, 12 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	0.45	48	0.42	52	0.44	17
2	0.89	49	0.84	52	0.84	17
4	1.72	51	1.66	53	1.76	17
8	3.29	53	3.17	55	3.30	18
16	6.04	58	4.98	70	5.94	20
32	10.13	69	5.57	126	9.27	25
64	14.84	94	5.74	245	12.05	39
128	18.15	152	5.77	488	14.20	66
256	19.60	283	5.74	980	15.93	353
512	18.77	596	5.73	1966	17.20	655

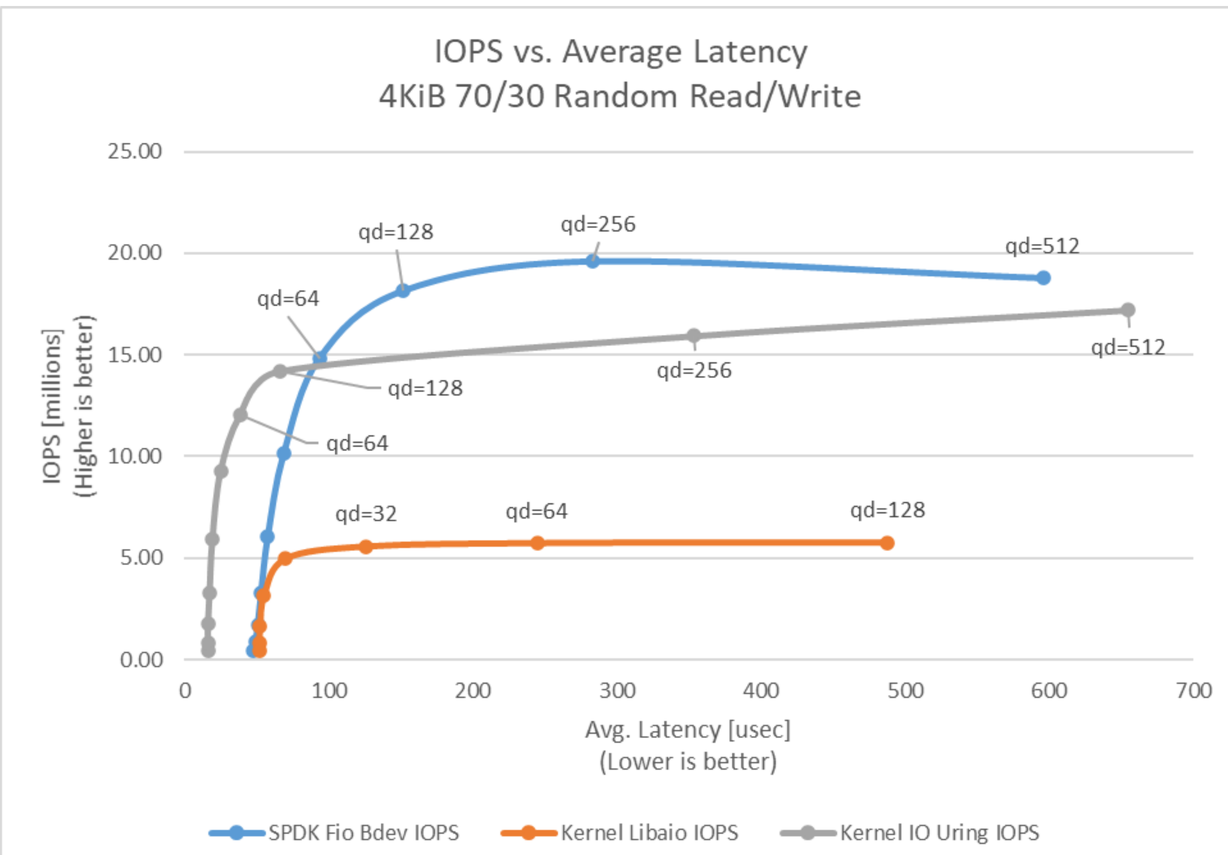

Figure 21: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io_uring polling (4KiB 70/30 Random Read/Write, 22 NVMe SSDs, 12 CPU Cores)

Conclusions

1. SPDK NVMe BDEV fio plugin reached up to around 29.3 million IOPS for Random Read workload at Queue Depth = 256. This is close to 30 million IOPS result measured in Test Case 2 - I/O Cores Scaling using Bdevperf.

2. For the 4KiB Random Write workload SPDK NVMe BDEV Fio plugin and Kernel io_uring had similar performance. We observed noticeable performance drop for both I/O engines when increasing queue depth, after reaching peak performance.

3. SPDK NVMe BDEV fio plugin reached up to 19.6 million IOPS for Random Read/Write workload at Queue Depth = 256, which is lower than the 22.6 million IOPS we measured in Test Case 2 - I/O Cores Scaling using Bdevperf.

4. The results for the Random Write workload exceeded what the 22NVMe SSDs are capable of (around 7.7M 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. The Kernel libaio ioengine achieved maximum performance of up to 5.7M IOPS with 12 CPU cores and was unable to saturate platforms NVMe disks or PCIe switches throughput. Peak performance was reached at QD=64 for Random Read and Random Read/Write workloads and at QD=8 for Random Write workload. Beyond these queue depth values there was no IOPS improvement, but the latency increased.

6. The Kernel io_uring engine reached a peak performance of 20.97 million IOPS at Queue Depth = 256 for Random Read workload, 16.30 million at QD = 32 for Random Write and 17.20 million at QD = 512 for Random Read/Write workload. However, when we looked at htop we noticed that io_uring was using 24 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 873K IOPS/Core for io_uring vs up to about 2.44M IOPS/Core for the SPDK NVMe bdev. The Submission Queue Polling blog provides more information about how to eliminate system calls with io_uring.

Summary

1. SPDK NVMe BDEV Block Layer using SPDK Bdevperf benchmarking tool can deliver up to 7.7 million IOPS on a single Intel® Xeon® Gold 6348 with Turbo Boost enabled.

2. The SPDK NVMe BDEV IOPS scale linearly with addition of CPU cores. We demonstrated up to 30 million IOPS on just 5 CPU cores (Intel® Xeon® Gold 6348 with Turbo Boost enabled).

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

    a. SPDK BDEV latency was 4.38% and 58.4% lower than Linux Kernel Libaio latency for Random Read and Random Write workloads.

    b. SPDK BDEV latency was about 1.36% lower than Linux Kernel io_uring latency for Random Read workload and 25.4% lower for Random Write workload.

4. SPDK NVMe Bdev Fio reaches up to 29.25 million IOPS with an average latency of around 188 while using 12 CPU cores an queue depth of 256. With the same fio workloads Kernel io_uring and Kernel libaio reach up to 20.97 million (using 24 cores: 12 for fio and 12 for submission queue polling) and 5.75 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

Performance varies by use, configuration and other factors. Learn more at www.Intel.com/PerformanceIndex.

Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates.

Your costs and results may vary.

No product or component can be absolutely secure.

Intel technologies may require enabled hardware, software or service activation.

§