

# SPDK NVMe BDEV Performance Report Release 24.01

---

**Testing Date:** January 2024

**Performed by:**

Karol Latecki ([karol.latecki@intel.com](mailto:karol.latecki@intel.com))

Jaroslav Chachulski ([jaroslawx.chachulski@intel.com](mailto:jaroslawx.chachulski@intel.com))

**Acknowledgments:**

Krzysztof Karas ([krzysztof.karas@intel.com](mailto:krzysztof.karas@intel.com))

# Contents

---

Contents .....	2
Audience and Purpose.....	3
Test setup .....	4
Hardware configuration .....	4
BIOS Settings .....	5
SSD Preconditioning .....	5
Introduction to SPDK Block Device Layer.....	6
Test Case 1: SPDK NVMe BDEV IOPS/Core Test .....	9
SPDK NVMe BDEV Single Core Throughput .....	10
Bdevperf vs. fio IOPS/Core results.....	12
NVMe BDEV vs. Polled-Mode Driver IOPS/Core.....	12
Conclusions .....	13
Test Case 2: SPDK NVMe BDEV I/O Cores Scaling .....	14
Results .....	15
Conclusions .....	16
Test Case 3: SPDK NVMe BDEV Latency .....	17
Average and tail latency comparison.....	19
Linux Kernel libaio Histograms .....	21
Linux Kernel io_uring Histograms .....	22
SPDK fio Bdev Histograms .....	23
Performance vs. increasing Queue Depth .....	24
Conclusions .....	26
Test Case 4: IOPS vs. Latency at different queue depths.....	27
4KiB Random Read Results.....	29
4KiB Random Write Results .....	30
4KiB Random 70%/30% Read/Write Results.....	31
Conclusions .....	32
Summary .....	33
List of tables.....	34
List of figures.....	35
References .....	36

## ***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 (6.1.6-200.fc37.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	<a href="#">Ultra SuperServer SYS-220U-TNR</a> 																		
Motherboard	Server board <a href="#">X12DPU-6</a>																		
CPU	2 CPU sockets, <a href="#">Intel(R) Xeon(R) Gold 6348 CPU @ 2.60GHz</a> Number of cores 28 per socket, number of threads 56 per socket Both sockets populated Microcode: 0xd000389																		
Memory	16 x 32GB SK Hynix DDR4 HMA84GR7DJR4N-XN; Total 512 GBs. Memory channel population: <table border="1"> <thead> <tr> <th>P1</th><th>P2</th></tr> </thead> <tbody> <tr> <td>CPU1_DIMM_A1</td><td>CPU2_DIMM_A1</td></tr> <tr> <td>CPU1_DIMM_B1</td><td>CPU2_DIMM_B1</td></tr> <tr> <td>CPU1_DIMM_C1</td><td>CPU2_DIMM_C1</td></tr> <tr> <td>CPU1_DIMM_D1</td><td>CPU2_DIMM_D1</td></tr> <tr> <td>CPU1_DIMM_E1</td><td>CPU2_DIMM_E1</td></tr> <tr> <td>CPU1_DIMM_F1</td><td>CPU2_DIMM_F1</td></tr> <tr> <td>CPU1_DIMM_G1</td><td>CPU2_DIMM_G1</td></tr> <tr> <td>CPU1_DIMM_H1</td><td>CPU2_DIMM_H1</td></tr> </tbody> </table>	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
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																		
Operating System	Fedora 37																		
BIOS	1.4b																		
Linux kernel version	6.1.6-200.fc37.x86_64 Spectre-meltdown mitigations enabled																		
SPDK version	SPDK 24.01																		
Fio version	3.28																		
Storage	<b>OS:</b> 1x 250GB Crucial CT250MX500SSD1 <b>Storage:</b> 22x Kioxia® KCM61VUL3T20 3.2TBs (FW: 0105) (10 on CPU NUMA Node 0, 12 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

Table 3: Test System NVMe storage setup

Table 37: Test system NVMe Storage Setup		
Item	Description	
PCIe Riser cards	<b>“Ultra” Riser Card:</b> AOC-2UR68G4-i2XT <ul style="list-style-type: none"><li>• PCIe Slot 1 – x16, CPU2</li><li>• PCIe Slot 2 – x8, CPU2</li><li>• PCIe Slot 3 – x8, CPU2</li></ul> <b>Right-facing riser card:</b> RSC-WR-6 <ul style="list-style-type: none"><li>• PCIe Slot 4 – x16, CPU1</li></ul> <b>Left-facing riser card:</b> RSC-W2-66G4 <ul style="list-style-type: none"><li>• PCIe Slot 5 – x16, CPU2</li><li>• PCIe Slot 7 – x16, CPU1</li></ul> More information can be found in <a href="#">SYS-220U-TNR manual document</a> .	
	PCIe Retimer cards	
NVMe Drives distribution across the system	3 x <a href="#">AOC-SLG4-4E4T</a> Installed in: <ul style="list-style-type: none"><li>○ PCIe Retimer 1: RSC-WR-6, PCIe Slot 4 (using CPU1 PCIe Lanes)</li><li>○ PCIe Retimer 2: AOC-2UR68G4-i2XT, PCIe Slot 1 (using CPU2 PCIe Lanes)</li><li>○ PCIe Retimer 3: RSC-W2-66G4, PCIe Slot 5 (using CPU2 PCIe Lanes)</li></ul>	
	Nvme0 – 5	Motherboard ports (CPU1 PCIe Lanes)
	Nvme6 – 9	Motherboard ports (CPU2 PCIe Lanes)
	Nvme9 – 13	PCIe Retimer 1 (CPU1 PCIe Lanes)
	Nvme14 - 17	PCIe Retimer 2 (CPU2 PCIe Lanes)
	Nvme18 - 21	PCIe Retimer 3 (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 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](#).

# 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 userspace 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 with 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"  
          }  
        }  
      ],  
      [...]  
    },  
    {  
      "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\_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
```

```
[filename3]
iodepth=192
cpus_allowed=23
filename=Nvme17n1
filename=Nvme18n1
filename=Nvme19n1
filename=Nvme20n1
filename=Nvme21n1
```

*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 the 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 runs eliminates having to precondition between runs because all SSDs were used in the previous run, so they still are 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	<b>fio version:</b> fio-3.28  <b>Number of NVMe SSDs:</b> {1, 2, 3, 4, 5, 6, 7, 8, 10, 12, 14, 16, 18, 20, 22}  <b>SPDK_BDEV_IO_CACHE_SIZE</b> changed from 256 to 2048 (using bdev_set_options RPC call).
Bdevperf configuration	spdk/test/bdev/bdevperf/bdevperf -c bdev.conf -q \${iodepth} -o \${block_size} -w \${rw} -M \${rwmixread} -t 300 -m 0 -p 0
fio configuration	[global] ioengine=spdk_bdev spdk_json_conf=bdev.conf

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

## SPDK NVMe BDEV Single Core Throughput

The first test was performed using SPDK bdevperf, which is a 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.

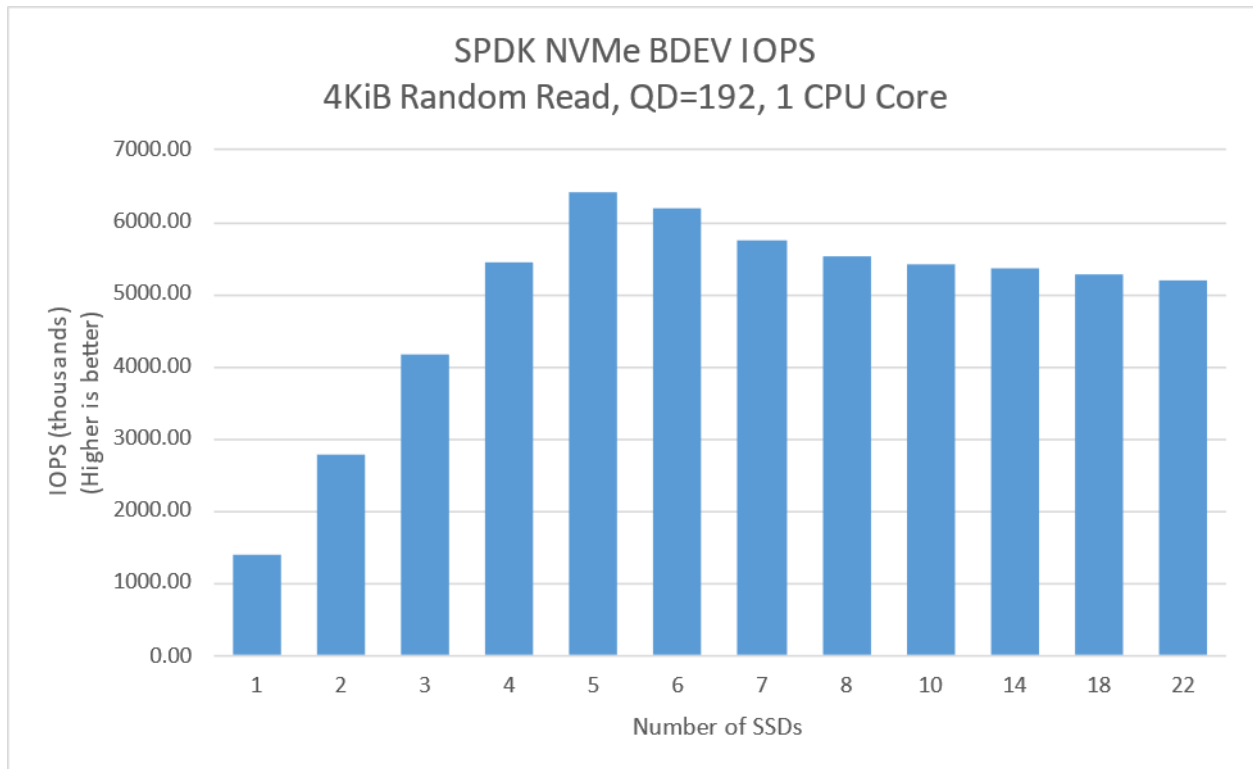


Figure 3: SPDK NVMe BDEV IOPS scalability with addition of SSDs (4KiB Random Read, 1CPU Core, QD=192, using bdevperf tool)

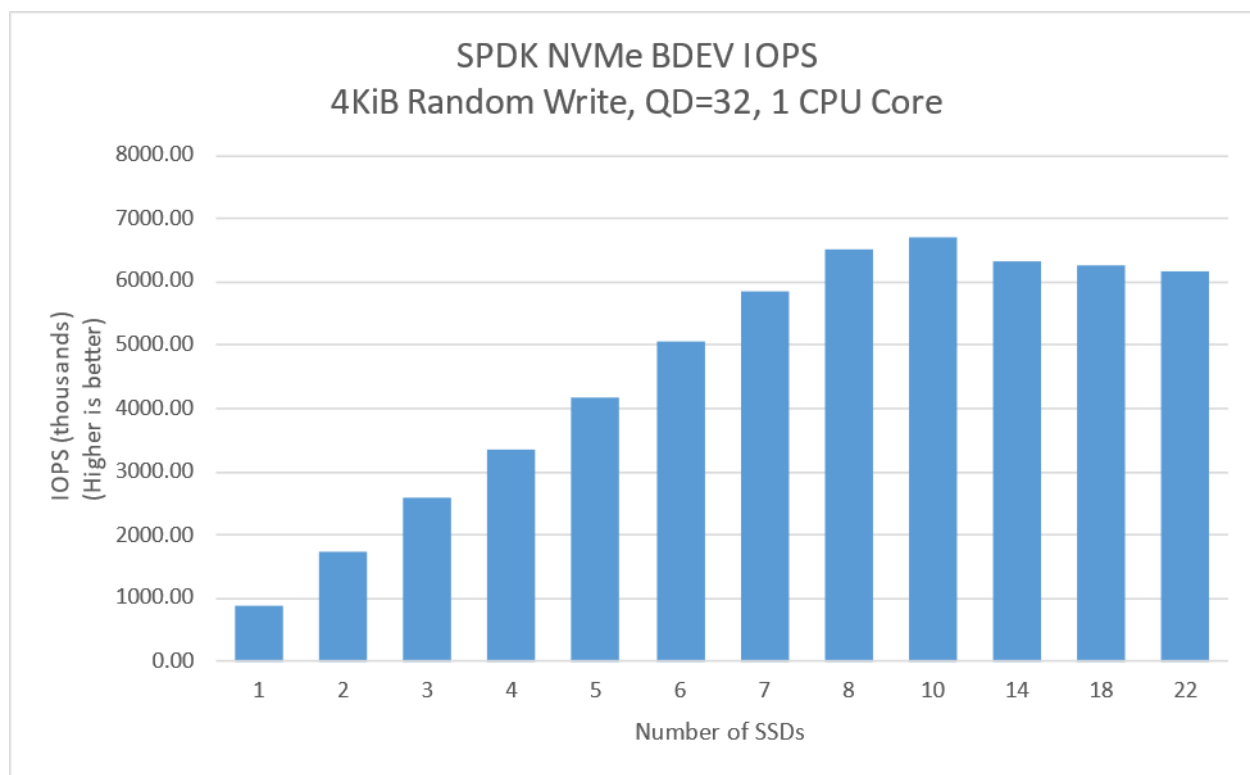


Figure 4: SPDK NVMe BDEV IOPS scalability with addition of SSDs (4KiB Random Write, 1CPU Core, QD=32, using bdevperf tool)

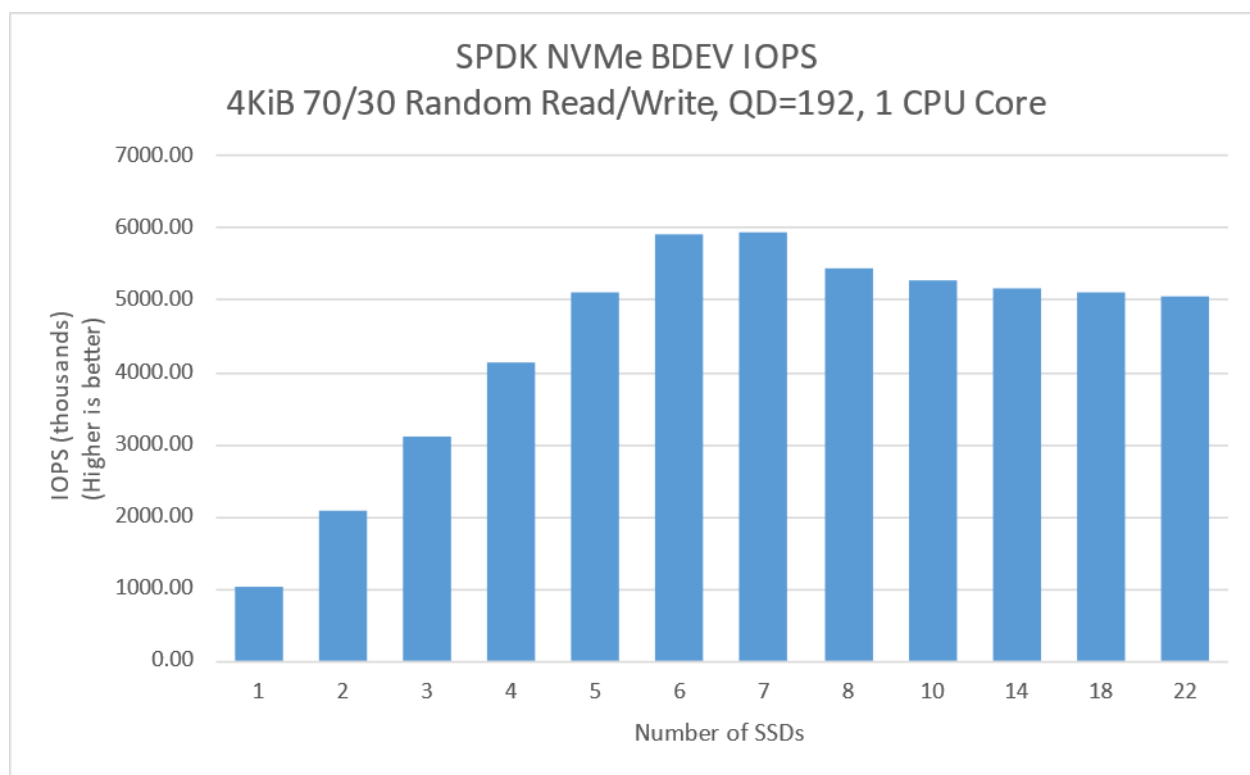


Figure 5: SPDK NVMe BDEV IOPS scalability with addition of SSDs (4KiB 70/30 Random Read/Write, 1CPU Core, QD=192, using bdevperf tool)

## 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 bdevperf vs. the fio benchmarking tool with the SPDK bdev plugin.

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

Workload	SPDK fio bdev Plugin (IOPS, thousands)	SPDK bdevperf (IOPS, thousands)	Performance gain
4KiB Random Read, QD=192, 5 SSDs	3381.56	6409.81	89.6%
4KiB Random Write, QD=32, 10 SSDs	2652.09	6702.06	152,7%
4KiB 70/30 Random Read/Write, QD=192, 7 SSDs	2974.93	5933.43	99%

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 177% 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 8.5% and 26.7% 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 additional CPU core for every 4 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	6409.81	6956.54	8.5%
4KiB Random Write, QD=32, 10 SSDs	6702.06	8491.02	26.7%

## Conclusions

1. The SPDK NVMe block device module adds approximately 8.5% and 26.7% 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 6.7 million IOPS.
4. Performance scales linearly with addition of NVMe SSDs up 7 SSDs for Random Read/Write workload, reaching around 6.0 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 4KiB Random Write workload exceeded the expected NVMe SSDs maximum throughput. We suspect this is due to imperfect preconditioning process, which wears off over time. The results, however, were repeatable for several 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:**

- 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	<b>Number of CPU Cores:</b> 1, 2, 3, 4, 5, 6  <b>Number of NVMe SSDs:</b> 5 per each CPU Core used in test, up to maximum of 22 NVMe SSDs  <b>NUMA optimization:</b> CPUs for test were selected in a way to match NVMe drives distribution across platform NUMA nodes.
Bdev perf configuration	<code>spdk/test/bdev/bdevperf/bdevperf --json bdev.conf \ -q 128 -o 4096 -w randrw -M \${MIXREAD} \ -t 300 -m \${CORE_MASK} -p \${PRIMARY_CORE}</code>

## 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)

CPU Cores	NVMe SSDs	IOPS (thousands)		
		Random Read QD=192	Random Write QD=32	70/30 Random Read/Write QD=192
1	5	6396.46	2698.18	4692.37
2	10	12894.94	6566.63	9800.18
3	15	18994.18	10601.70	14849.33
4	20	25211.06	15069.18	19927.24
5	22	28401.12	17412.86	22310.85
6	22	29603.57	18121.11	22656.35

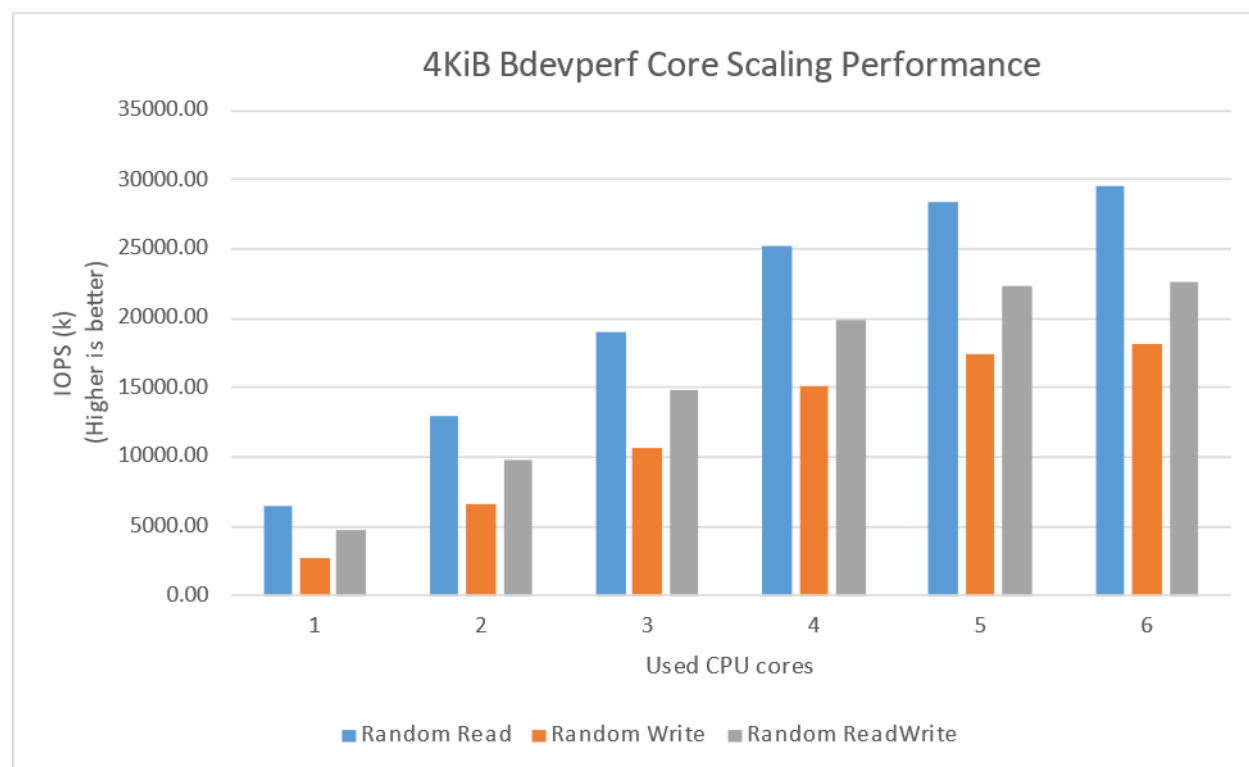


Figure 6: SPDK NVMe BDEV I/O Cores Scalability (4KiB 100% Random Read IOPS at QD=192; 4KiB 100% Random Write IOPS at QD=32; 4KiB 70/30 Random Read/Write IOPS at QD=192)

## Conclusions

1. The IOPS for 4KiB Random Read workload scales linearly with addition of I/O cores until NVMe drives used for test are saturated, reaching around 29 million IOPS.
2. The IOPS for 4KiB 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 4KiB Random Write workload scale linearly. The IOPS exceeded the expected NVMe SSDs throughput for this workload which is about 7.7M IOPS. We suspect this is due to imperfect 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 (Interrupt Request) delivery delay and IRQ handler scheduling. This test case includes a comparison of the I/O latency for the Kernel vs. SPDK.

### Test Workloads:

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

Table 9: SPDK NVMe BDEV Latency Test

Item	Description
<b>Test case</b>	Test SPDK NVMe BDEV Latency Test
<b>Test configuration</b>	<b>Fio version:</b> fio-3.28 <b>Number of CPU cores:</b> 1 <b>Number of NVMe SSDs:</b> 1
<b>SPDK NVMe Driver Configuration</b>	ioengine=spdk_bdev
<b>Linux Kernel Default (libaio) Configuration</b>	ioengine=libaio
<b>Linux Kernel io_uring</b>	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
<b>fio configuration (common part)</b>	[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
<b>fio configuration (SPDK specific)</b>	<pre>[global] ioengine=spdk_bdev spdk_conf=/tmp/bdev.conf  [filename0] iodepth=1 cpus_allowed=0 filename=Nvme0n1</pre>
<b>fio configuration (Linux Kernel common)</b>	<pre>[global] ioengine={libaio   io_uring}  [filename0] iodepth=1 cpus_allowed=0 filename=/dev/nvme0n1</pre>
<b>fio configuration (Linux Kernel io_uring specific)</b>	<pre>[global] fixedbufs=1 hipri=1 registerfiles=1 sqthread_poll=1</pre>

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

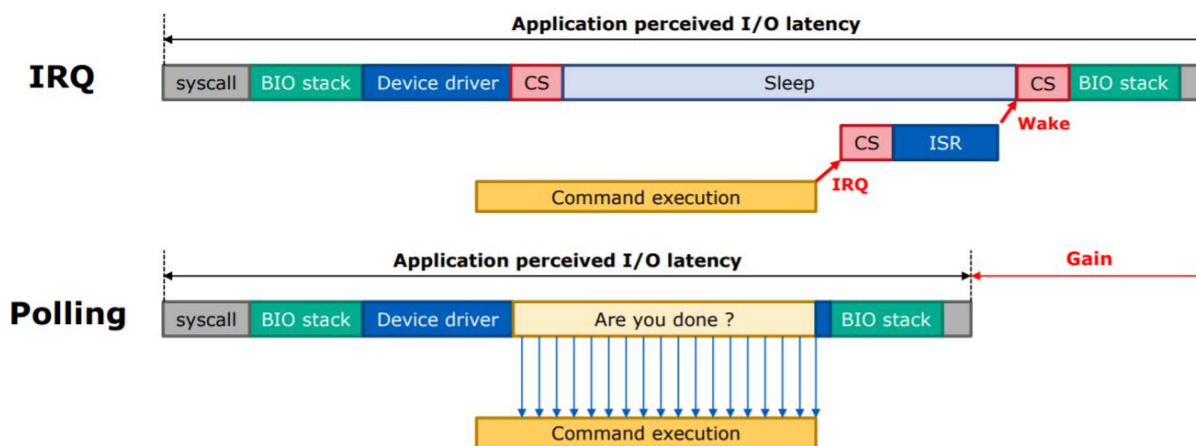


Figure 7: Linux Block Layer I/O Optimization with Polling. Source [1]

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.

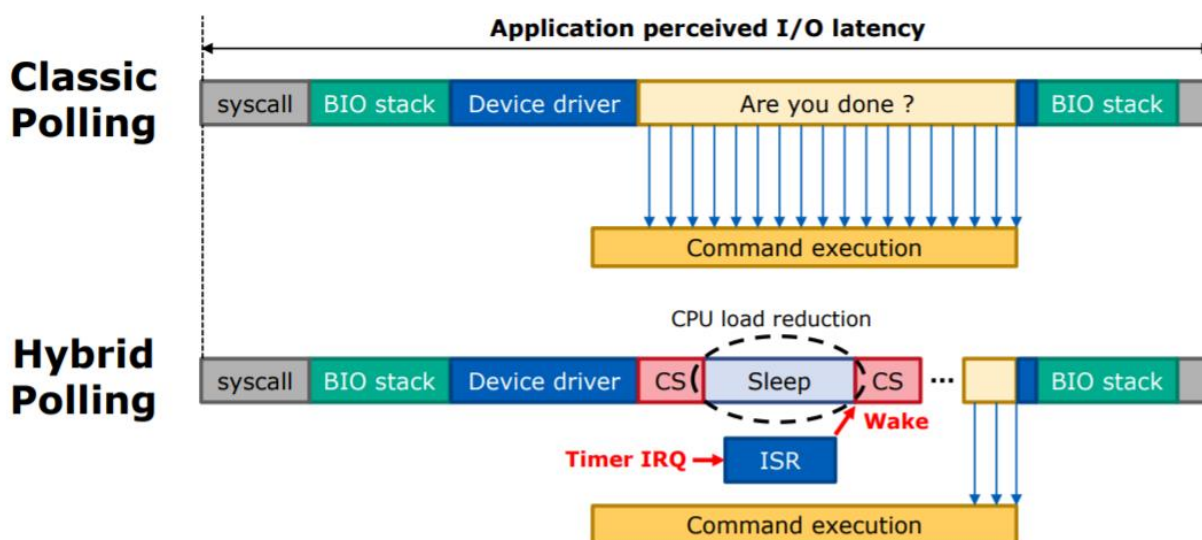


Figure 8: Linux Block I/O Classic and Hybrid Polling latency breakdown. Source [1]

The data in tables and charts compares the I/O latency for a various 4KiB workloads performed using the SPDK bdev vs. Linux block layer I/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)	SPDK Fio BDEV Plugin	Linux Kernel (libaio)	Linux Kernel (io_uring)
Average	73.34	76.97	74.89
P90	82.43	85.50	84.48
P99	83.46	86.53	85.50
P99.99	148.48	154.62	150.53
Stdev	7.01	10.28	10.05
Average submission latency	0.16	0.98	0.00
Average completion latency	73.18	75.91	74.85

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

Latency metrics (usec)	SPDK Fio BDEV Plugin	Linux Kernel (Default libaio)	Linux Kernel (io_uring)
Average	5.48	9.02	6.80
P90	5.34	8.03	6.94
P99	6.05	8.10	7.14
P99.99	8.77	11.46	9.41
Stdev	1.44	0.43	2.77
Average submission latency	0.18	0.99	0.00
Average completion latency	5.30	7.95	6.73

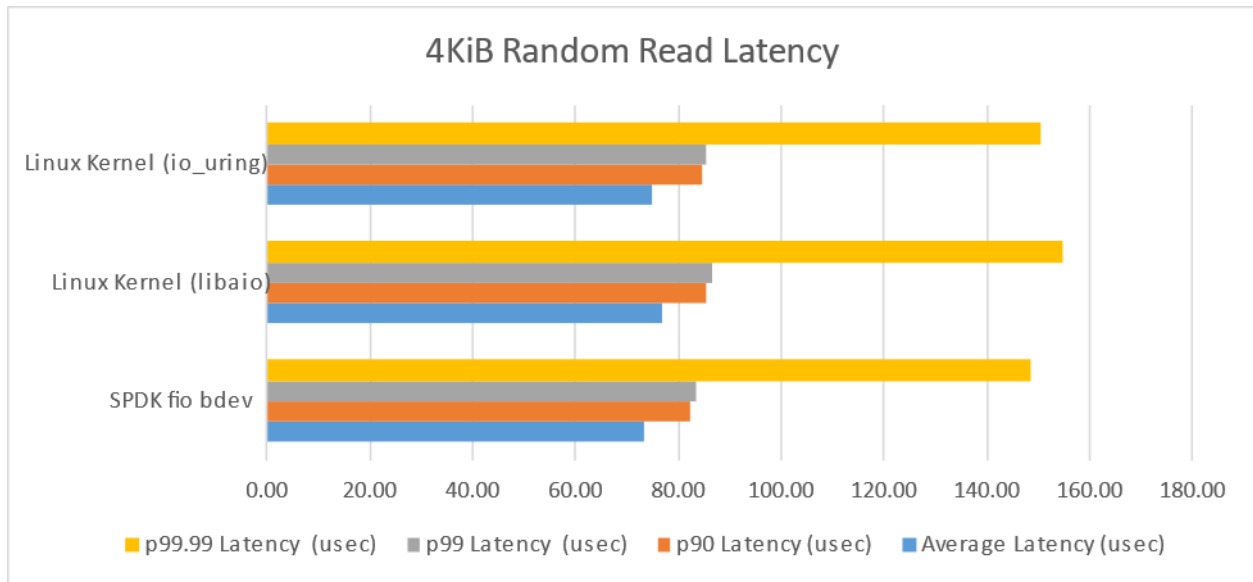


Figure 9: SPDK bdev vs Linux Kernel Latency comparison (4KiB Random Read)

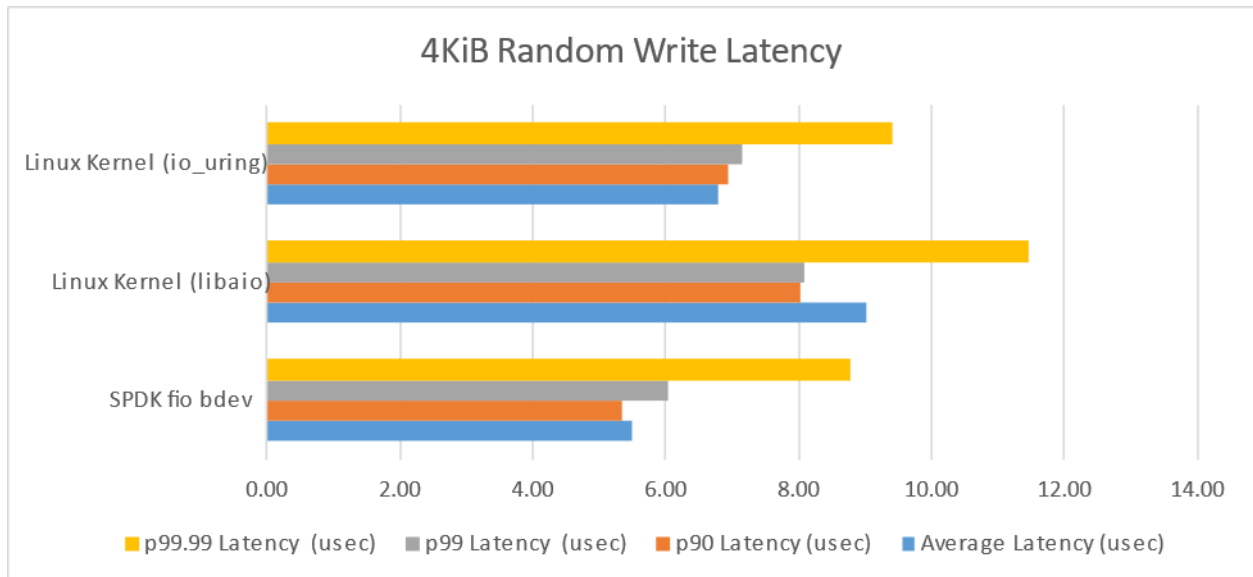


Figure 10: SPDK bdev vs Linux Kernel Latency comparison (4KiB Random Write)

## Linux Kernel libaio Histograms

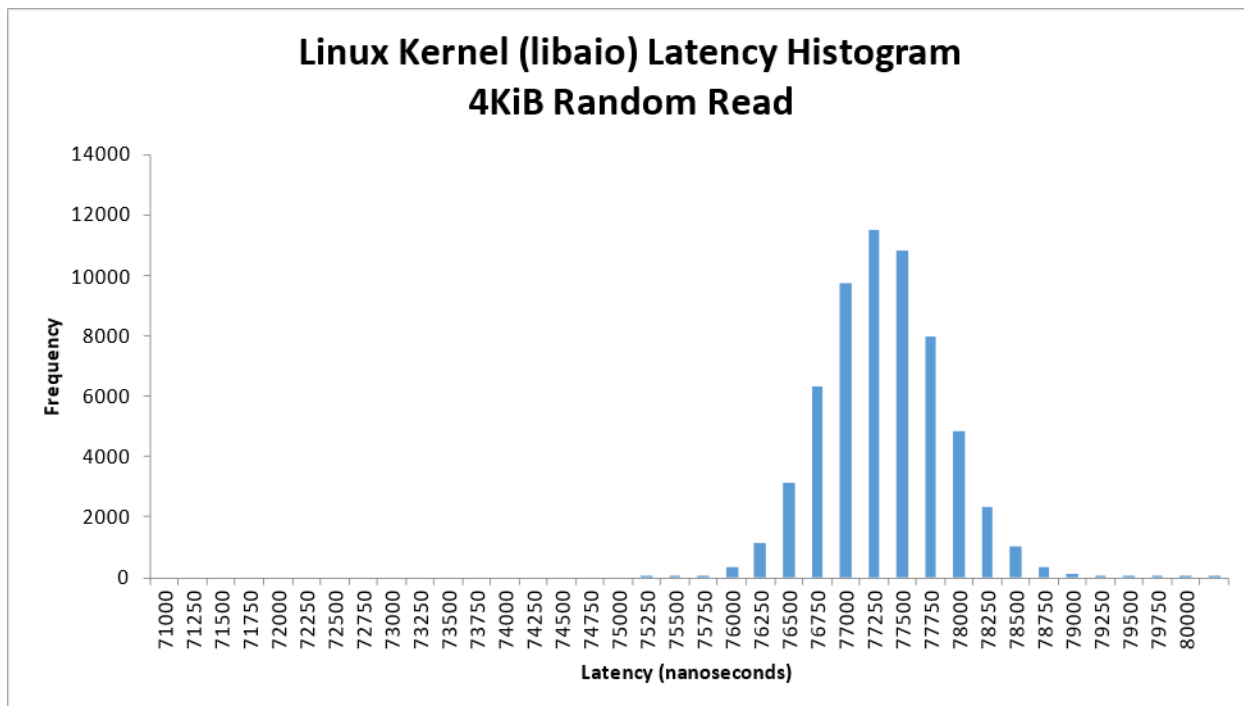


Figure 11: Linux Kernel (Default libaio) 4KiB Random Read Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec)

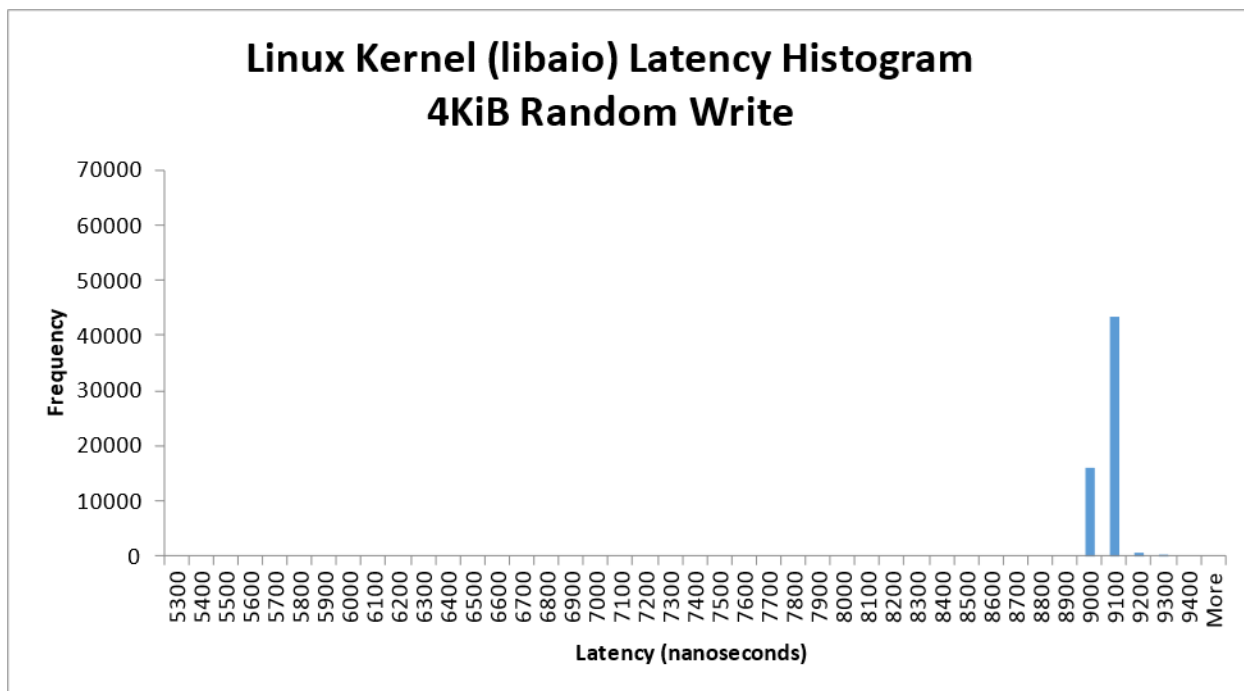


Figure 12: Linux Kernel (Default libaio) 4KiB Random Write Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec)

## Linux Kernel io\_uring Histograms

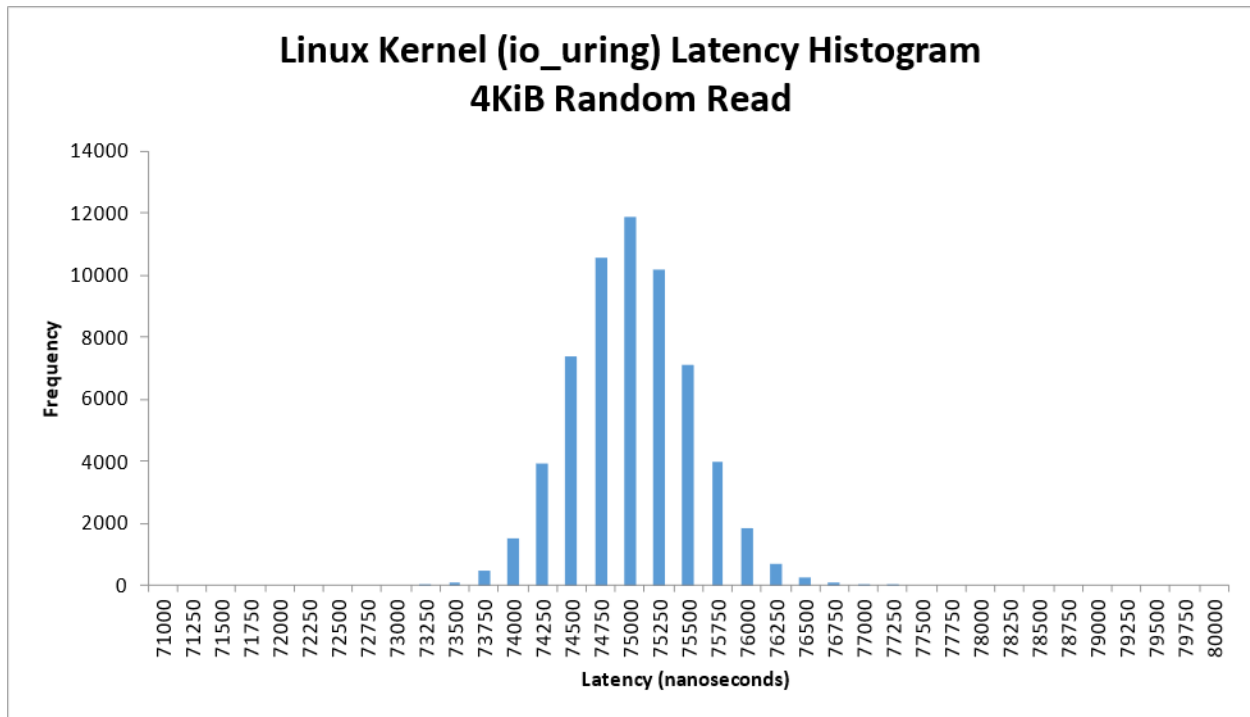


Figure 13: Linux Kernel (io\_uring polling) 4KiB Random Read Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec)

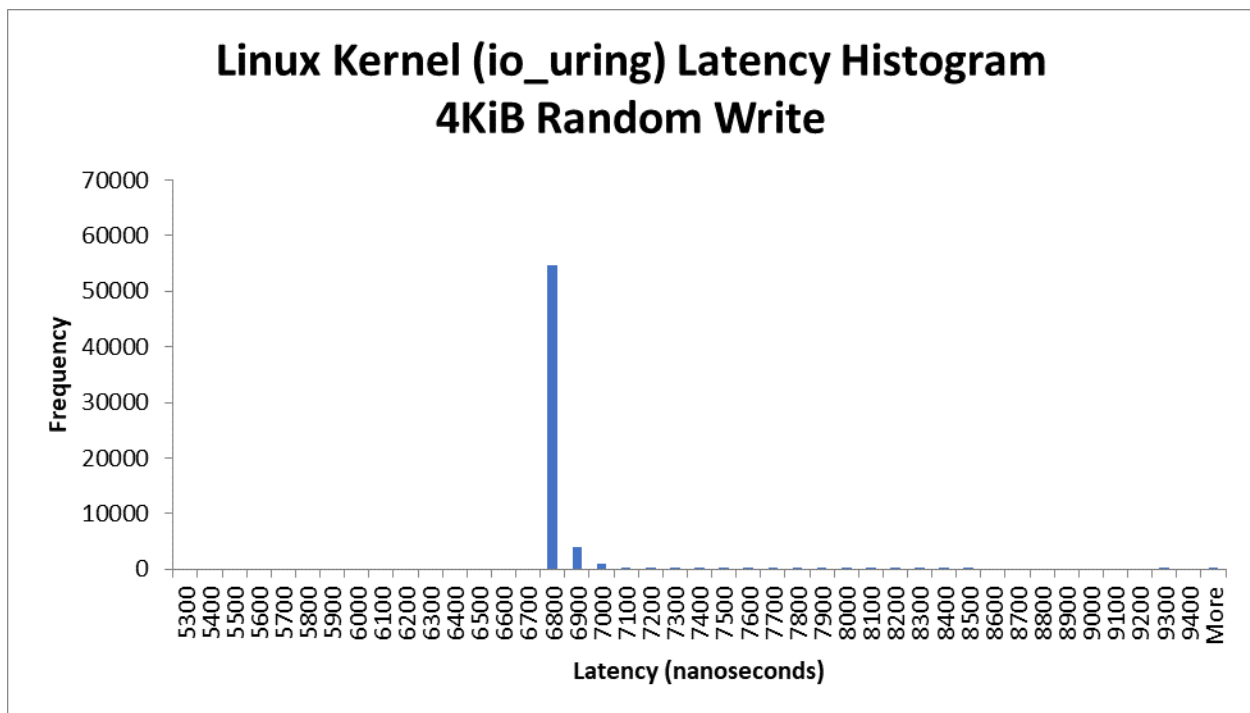


Figure 14: Linux Kernel (io\_uring polling) 4KiB Random Write Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec)

## SPDK fio Bdev Histograms

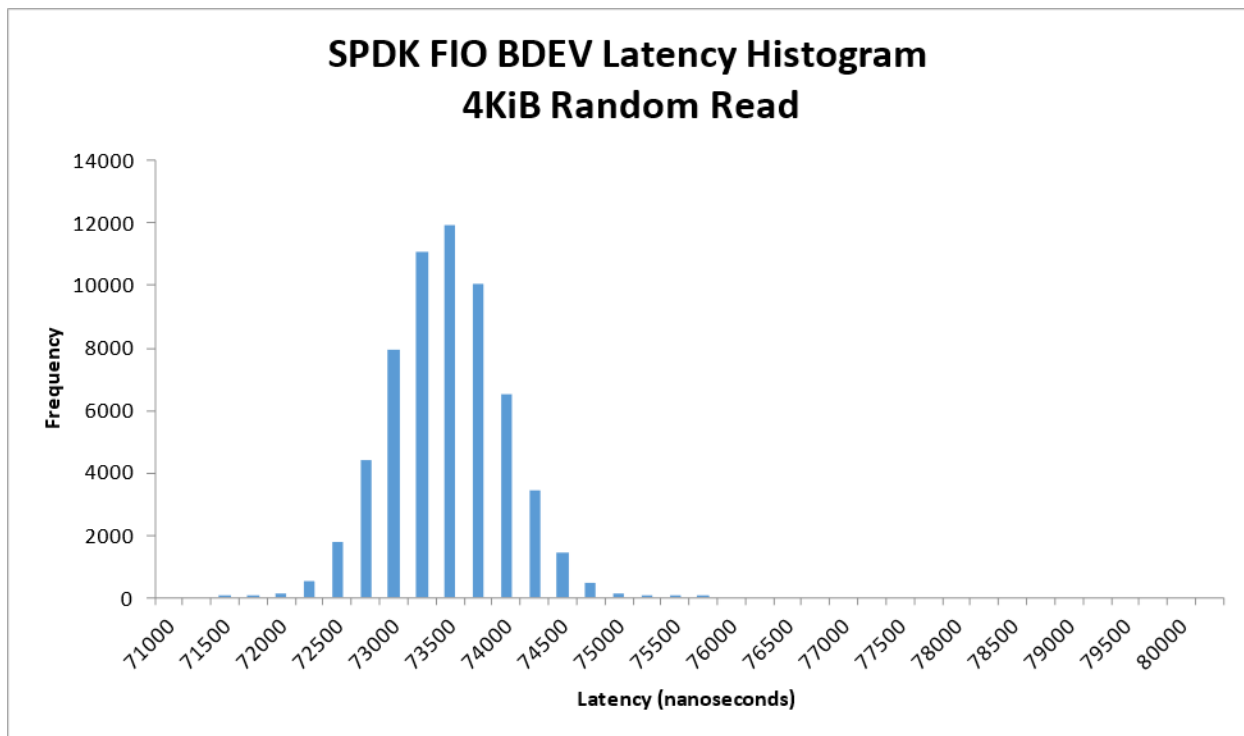


Figure 15: SPDK BDEV NVMe 4KiB Random Read Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec)

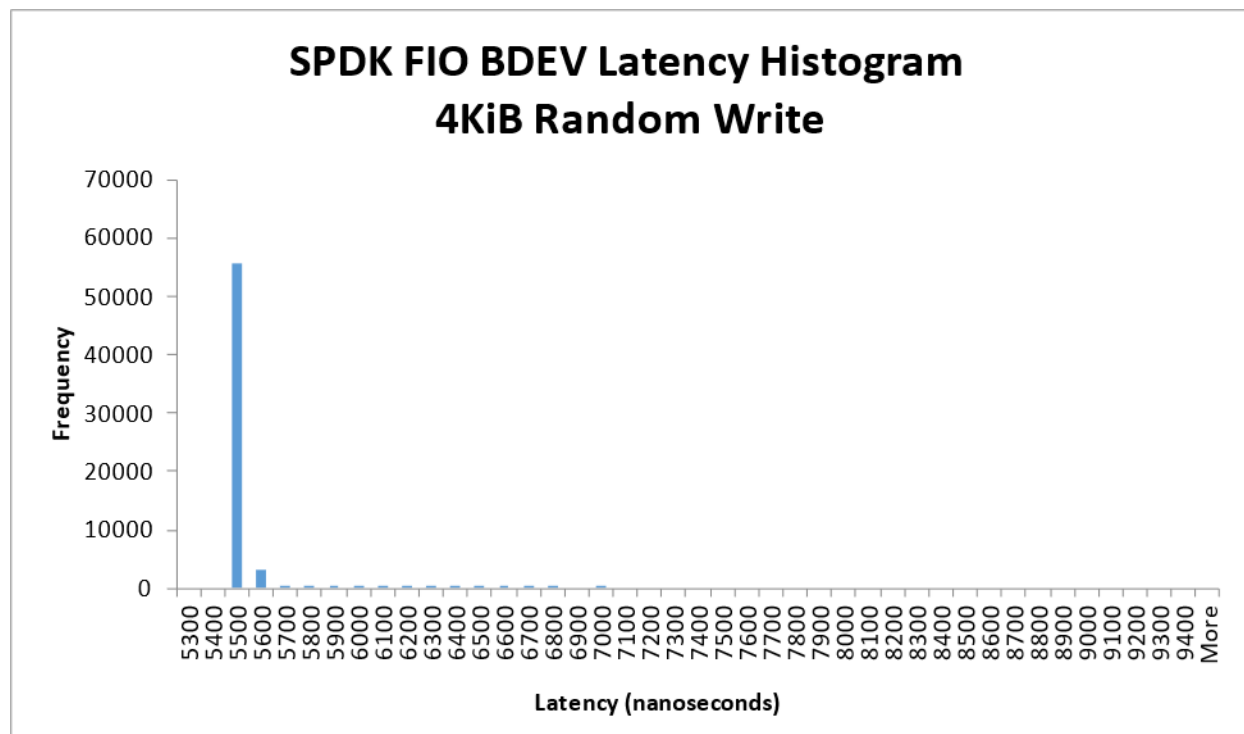


Figure 16: SPDK BDEV NVMe 4KiB Random Write Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec)

## 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	13590	73	12953	77	13368	75
2	27115	74	25818	77	26643	75
4	53952	74	51437	78	52964	75
8	106770	75	101655	78	104485	76
16	208921	76	198328	80	204049	78
32	399249	80	373963	85	387492	82
64	725220	88	493325	129	696265	92
128	1171557	109	489629	261	1065702	120
256	1399551	183	490289	522	1333316	192
512	1399512	366	490453	1044	1480304	346

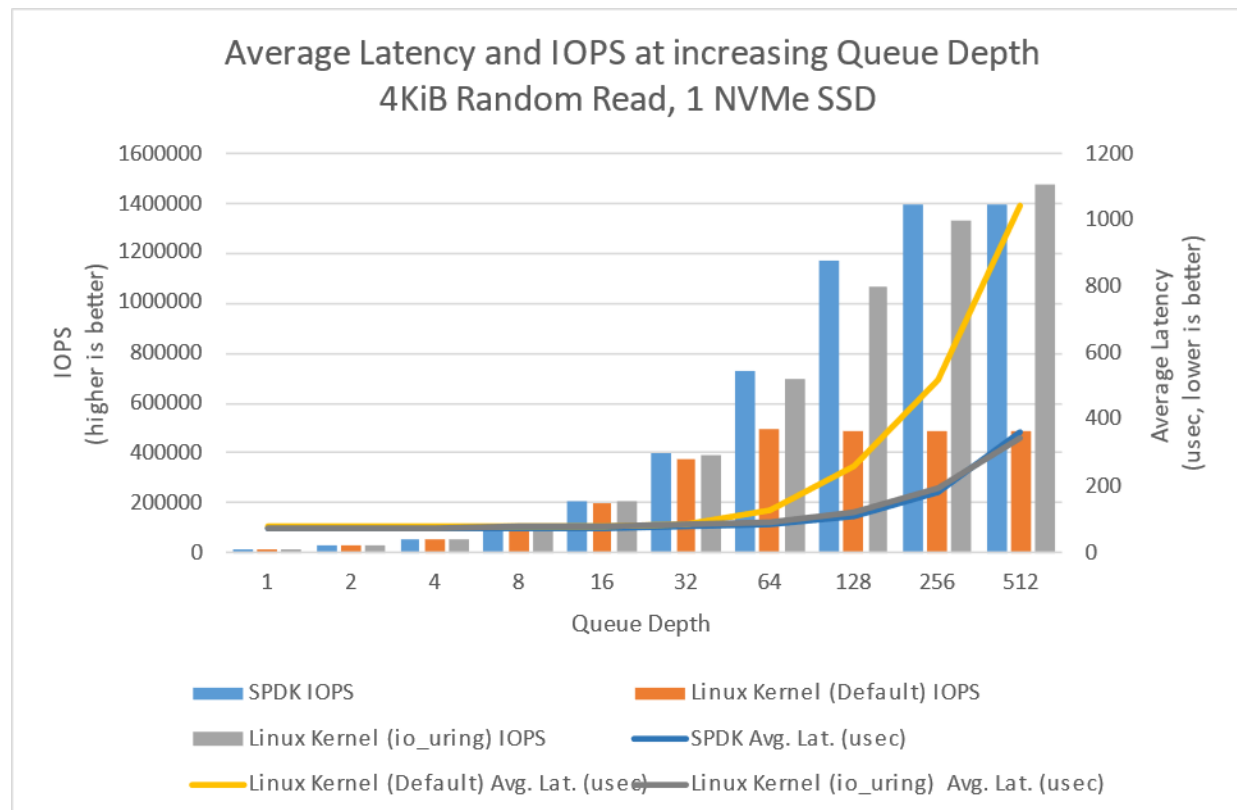


Figure 17: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io\_uring polling (4KiB Random Read, 1 NVMe SSD, 1 CPU Core, numjobs=1)



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)

QD	SPDK		Linux Kernel (Default libaio)		Linux Kernel (io_uring polling)	
	IOPS	Avg. Lat. (usec)	IOPS	Avg. Lat. (usec)	IOPS	Avg. Lat. (usec)
1	176526	5	107922	9	145435	7
2	347722	6	200648	10	281266	7
4	665749	6	455650	9	522824	7
8	861838	9	468345	17	844226	9
16	869300	18	467545	34	845590	19
32	806526	40	465244	69	857455	37
64	842971	76	465843	137	828599	77
128	794914	161	464816	275	804492	159
256	795837	322	466488	549	816535	314
512	728925	703	465346	1100	721546	710

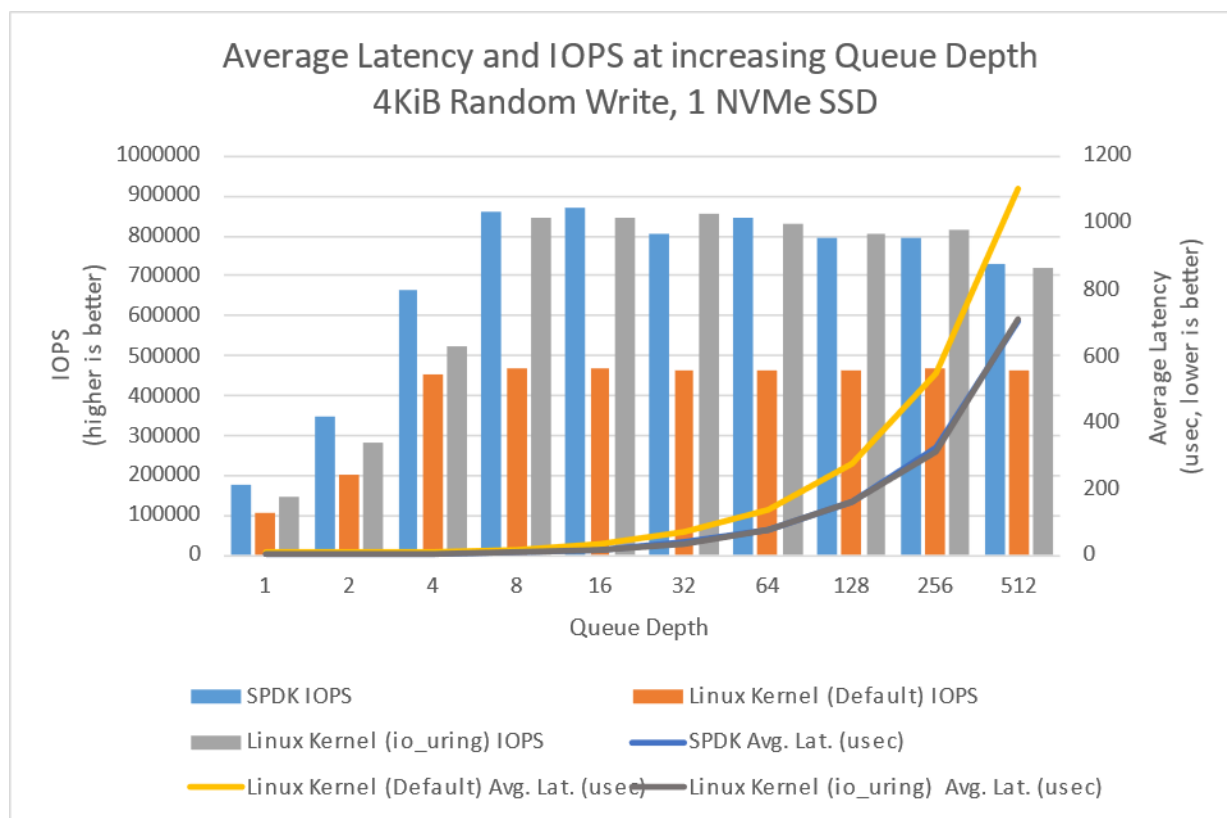


Figure 18: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io\_uring polling (4KiB Random Write, 1 NVMe SSD, 1 CPU Core, numjobs=1)

## 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 4.7% and 39.2% lower than Linux Kernel libaio, for Random Read and Random Write workloads respectively.
3. SPDK NVMe Bdev average latency was up to approximately 2.1% and 19.4% 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 at QD=512, reaching 1.48 million IOPS and saturating NVMe drive.
  - c. Kernel libaio peaked at QD=64 reaching approximately 485k IOPS. Increasing queue depth did not improve throughput.
6. For 4KiB Random Write workload, the IOPS for the libaio scaled linearly up to QD=4 and the IOPS for the io\_uring and SPDK engines scaled linearly up to QD=8. Further increasing the queue depth resulted in minor performance degradation.

## 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:

- 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
<b>Test case</b>	Test SPDK NVMe BDEV Latency Test at different Queue Depths
<b>Test configuration</b>	<b>fio version:</b> fio-3.28 <b>Number of CPU cores:</b> 12 <b>Number of NVMe SSDs:</b> 22
<b>Linux Kernel io_uring NVMe block device configuration</b>	<pre>echo 0 &gt; /sys/block/nvme0n1/queue echo 0 &gt; /sys/block/nvme0n1/rq_affinity echo 2 &gt; /sys/block/nvme0n1/nomerges echo -1 &gt; /sys/block/nvme0n1/io_poll_delay</pre>
<b>fio configuration (common part)</b>	<pre>[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</pre>
<b>fio configuration (SPDK specific)</b>	<pre>[global] ioengine=spdk_bdev spdk_conf=/tmp/bdev.conf</pre>

	<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>
<b>fio configuration (Linux Kernel common)</b>	<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>
<b>fio configuration (Linux Kernel io_uring specific)</b>	<pre> [global] fixedbufs=1 hipri=1 registerfiles=1 sqthread_poll=1 </pre>

## 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	74	0.28	77	0.30	74
2	0.60	74	0.57	78	0.59	74
4	1.19	74	1.12	78	1.17	75
8	2.35	75	2.21	80	2.31	76
16	4.59	76	4.19	84	4.49	78
32	8.77	80	5.42	130	8.44	83
64	15.84	88	5.67	248	13.99	100
128	25.15	110	5.67	496	18.94	149
256	29.65	186	5.66	995	23.15	243
512	28.92	384	5.67	1986	24.65	457

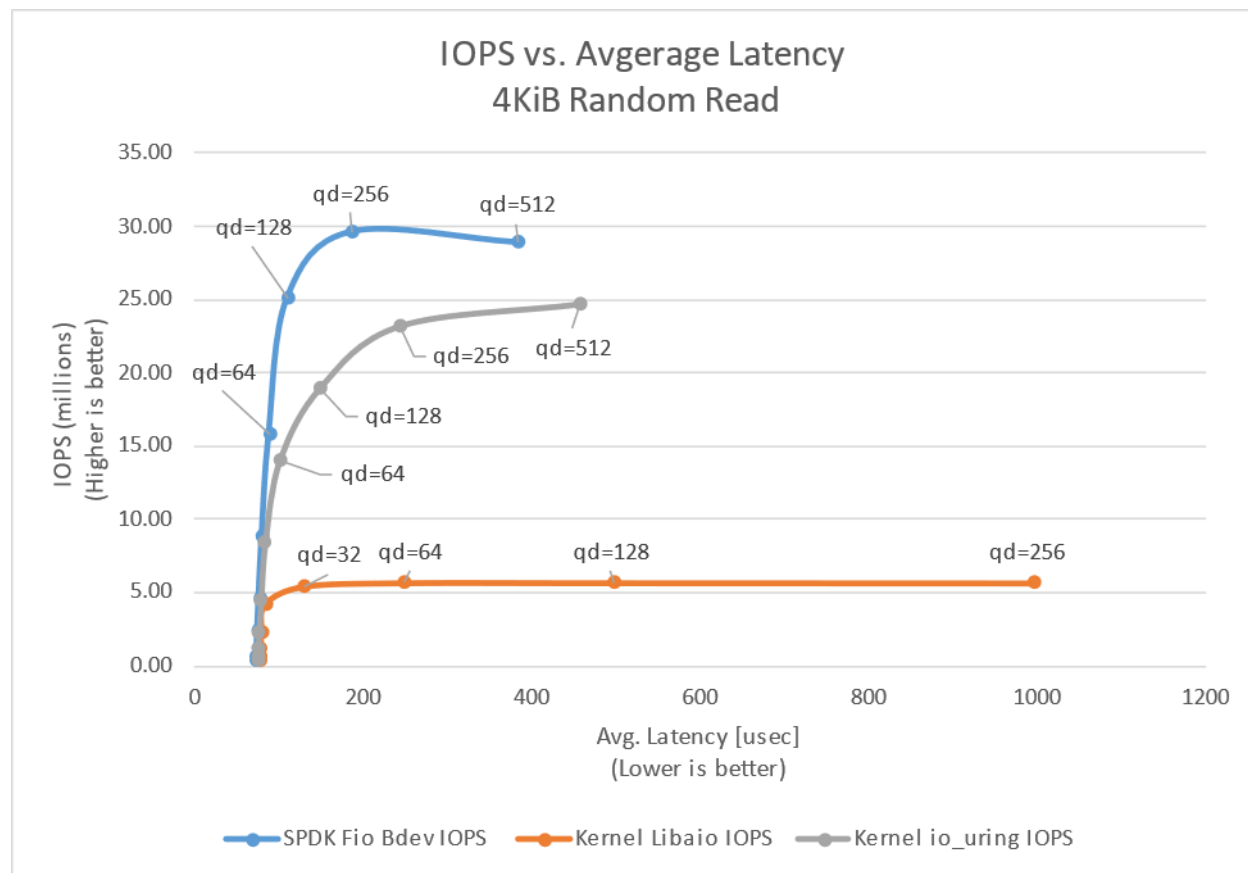


Figure 19: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io\_uring polling (4KiB Random Read, 22 NVMe SSDs, 12 CPU Cores). In the chart, we do not show results for Kernel Libaio higher than 256QD as it reduces the readability of the graph.

## 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	3.56	6	2.11	10	3.30	6
2	7.27	6	4.36	10	6.46	7
4	13.52	6	5.43	16	11.43	8
8	17.10	10	5.46	32	14.44	12
16	17.08	20	5.44	64	16.80	21
32	17.03	40	5.44	129	16.66	42
64	16.83	83	5.45	258	16.66	84
128	16.28	172	5.44	517	16.52	170
256	16.21	346	5.42	1038	16.26	346
512	15.54	723	5.35	2104	15.94	707

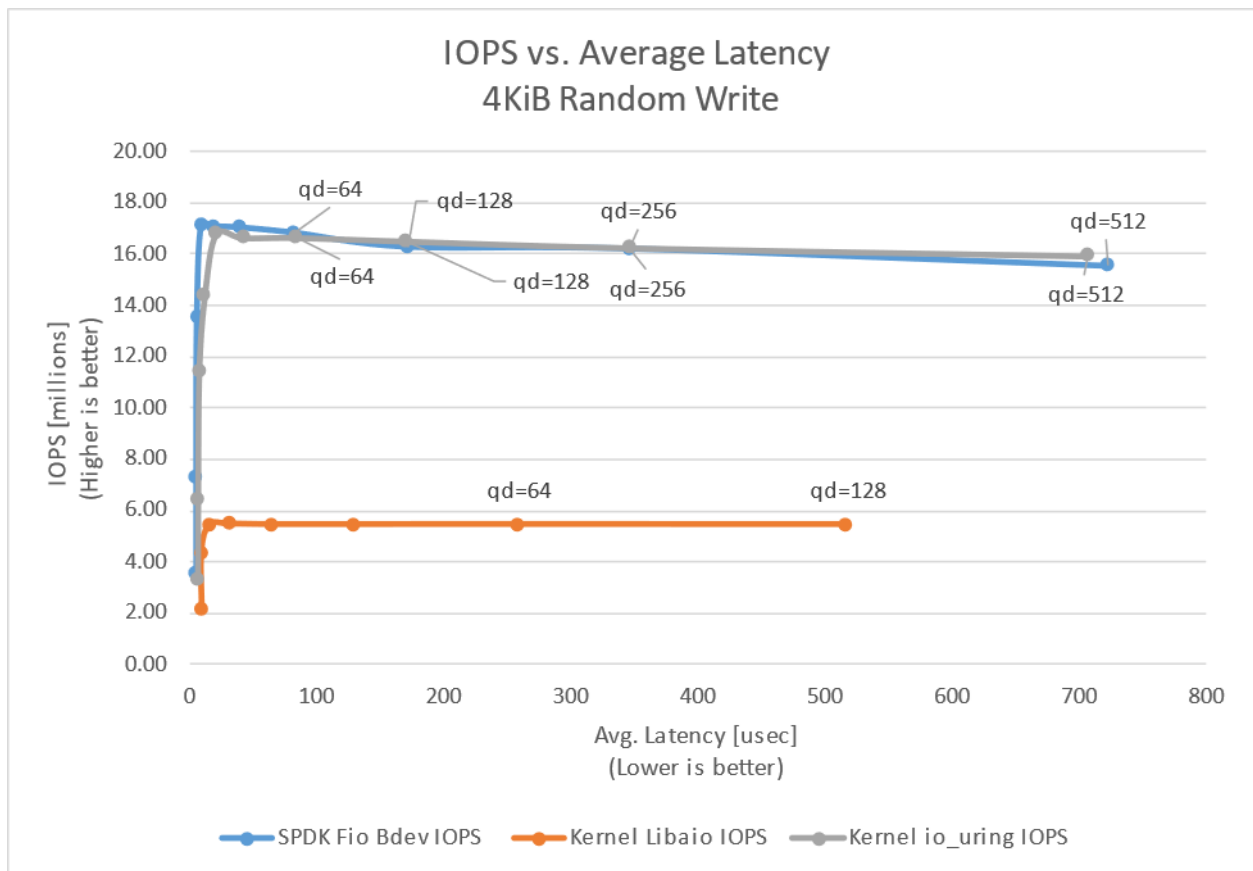


Figure 20: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io\_uring polling (4KiB Random Write, 22 NVMe SSDs, 12 CPU Cores). In the chart, we do not show results for Kernel Libaio higher than 128QD as it reduces the readability of the graph.

## 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)

QD	SPDK		Linux Kernel (Default libaio)		Linux Kernel (io_uring polling)	
	IOPS (millions)	Avg. Lat. (usec)	IOPS (millions)	Avg. Lat. (usec)	IOPS (millions)	Avg. Lat. (usec)
1	0.47	16	0.44	17	0.47	16
2	0.90	16	0.85	17	0.92	16
4	1.77	16	1.67	17	1.79	16
8	3.40	17	3.21	18	3.30	18
16	6.12	19	4.91	24	5.97	20
32	9.99	23	5.44	43	8.58	27
64	14.43	32	5.52	85	11.34	41
128	18.12	51	5.57	169	13.84	68
256	19.47	287	5.56	1012	15.75	357
512	20.13	556	5.54	2034	17.68	637

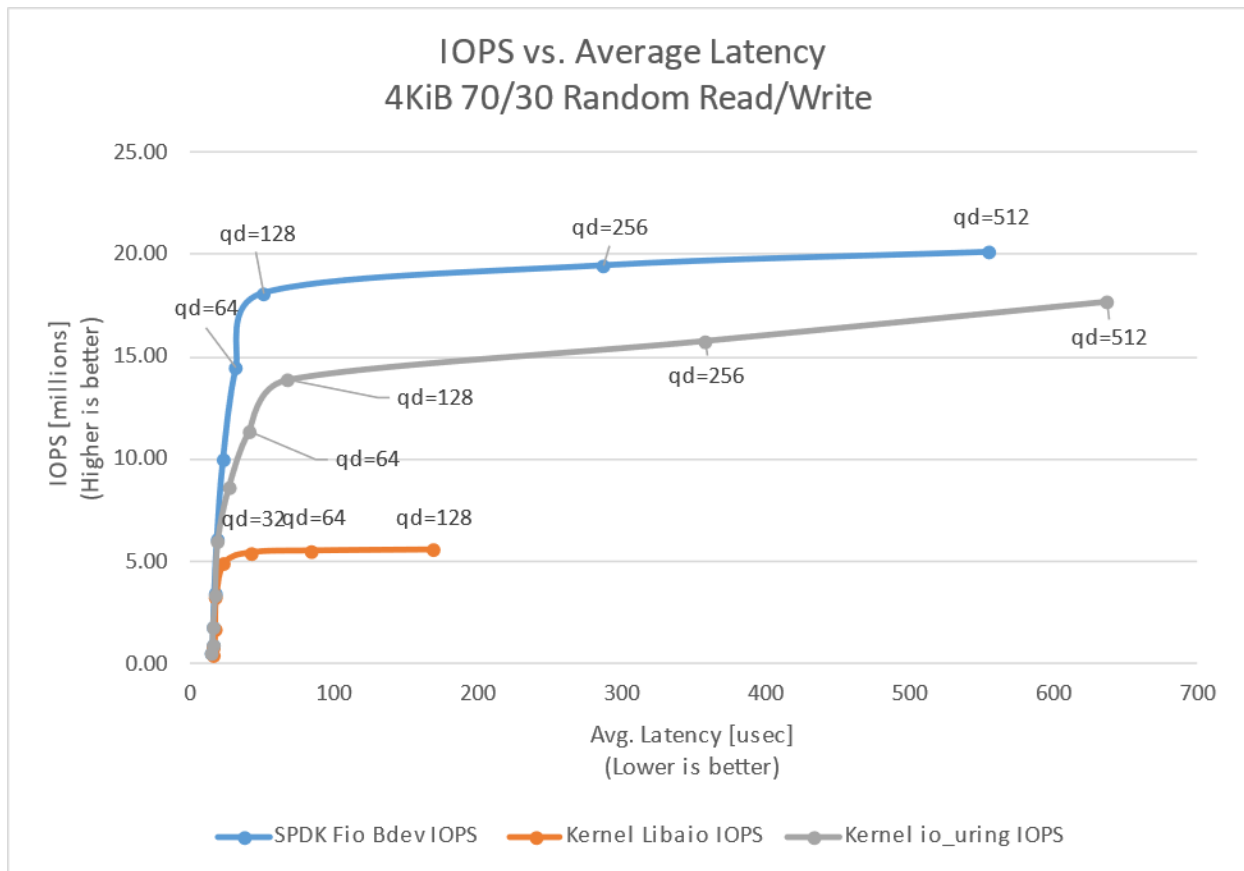


Figure 21: Performance at increasing Queue Depth; SPDK NVMe BDEV vs Linux Default libaio vs Linux io\_uring polling (4KiB 70/30 Random Read/Write, 22 NVMe SSDs, 12 CPU Cores). In the chart, we do not show results for Kernel Libaio higher than 128QD as it reduces the readability of the graph.

## Conclusions

1. SPDK NVMe bdev fio plugin reached up to around 29.65 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 20.13 million IOPS for Random Read/Write workload at Queue Depth = 512, 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 imperfect 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 IO engine achieved maximum performance of up to 5.67M IOPS with 12 CPU cores and was unable to saturate platform's NVMe disks 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 24.65 million IOPS at Queue Depth = 512 for Random Read workload, 16.8 million at QD = 16 for Random Write and 17.68 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; 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 submitted by the fio thread. Therefore, in terms of CPU efficiency we measured up to 1.03M IOPS/Core for io\_uring vs up to about 2.41M IOPS/Core for the SPDK NVMe bdev fio plugin. 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 6.7 million IOPS on a single Intel® Xeon® Gold 6348 CPU Core with Turbo Boost enabled.
2. The SPDK NVMe BDEV IOPS scale linearly with addition of CPU cores. We demonstrated up to 29.6 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 4.70% and 39.2% lower than Linux Kernel libaio latency for Random Read and Random Write workloads.
  - b. SPDK BDEV latency was about 2.1% lower than Linux Kernel io\_uring latency for Random Read workload and 19.4% lower for Random Write workload.
4. SPDK NVMe bdev Fio reaches up to 29.65 million IOPS with an average latency of around 186 usec while using 12 CPU cores at queue depth of 256. With the same fio workloads Kernel io\_uring and Kernel libaio reach up to 23.15 million (using 24 cores: 12 for fio and 12 for submission queue polling) and 5.66 million IOPS respectively.

## List of tables

---

Table 1: Hardware setup configuration .....	4
Table 2: Test setup BIOS settings .....	5
Table 3: Test System NVMe storage setup .....	5
Table 4: SPDK NVMe BDEV IOPS Test configuration.....	9
Table 5: IOPS/Core performance; SPDK fio bdev plugin vs SPDK bdevperf (Blocksize=4KiB, 1 CPU Core).....	12
Table 6: SPDK NVMe Bdev vs SPDK NVMe PMD IOPS/Core (Blocksize=4KiB, 1 CPU Core) .....	12
Table 7: SPDK NVMe BDEV I/O Cores Scalability Test .....	14
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) .....	15
Table 9: SPDK NVMe BDEV Latency Test .....	17
Table 10: SPDK bdev vs. Linux Kernel latency comparison (4KiB Random Read, QD=1, runtime=900s) .....	19
Table 11: SPDK bdev vs. Linux Kernel latency comparison (4KiB Random Write, QD=1, runtime=900s) .....	19
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) .....	24
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) .....	25
Table 14: SPDK NVMe BDEV Latency Test at different Queue Depths configuration.....	27
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).....	29
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) .....	30
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).....	31

---

## List of figures

---

Figure 1 : Example NVMe bdev configuration file.....	7
Figure 2: Example SPDK Fio BDEV configuration file .....	8
Figure 3: SPDK NVMe BDEV IOPS scalability with addition of SSDs (4KiB Random Read, 1CPU Core, QD=192, using bdevperf tool) .....	10
Figure 4: SPDK NVMe BDEV IOPS scalability with addition of SSDs (4KiB Random Write, 1CPU Core, QD=32, using bdevperf tool) .....	11
Figure 5: SPDK NVMe BDEV IOPS scalability with addition of SSDs (4KiB 70/30 Random Read/Write, 1CPU Core, QD=192, using bdevperf tool) .....	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).....	15
Figure 7: Linux Block Layer I/O Optimization with Polling. Source [1] .....	18
Figure 8: Linux Block I/O Classic and Hybrid Polling latency breakdown. Source [1] .....	19
Figure 9: SPDK bdev vs Linux Kernel Latency comparison (4KiB Random Read).....	20
Figure 10: SPDK bdev vs Linux Kernel Latency comparison (4KiB Random Write) .....	20
Figure 11: Linux Kernel (Default libaio) 4KiB Random Read Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec) .....	21
Figure 12: Linux Kernel (Default libaio) 4KiB Random Write Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec) .....	21
Figure 13: Linux Kernel (io_uring polling) 4KiB Random Read Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec) .....	22
Figure 14: Linux Kernel (io_uring polling) 4KiB Random Write Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec) .....	22
Figure 15: SPDK BDEV NVMe 4KiB Random Read Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec).....	23
Figure 16: SPDK BDEV NVMe 4KiB Random Write Average Latency Histogram (QD=1, Runtime=900s, fio, sampling interval = 15msec).....	23
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) .....	24
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) .....	25
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).....	29
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) .....	30
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).....	31

## 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](https://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.

© Intel Corporation. Intel, the Intel logo, and other Intel marks are trademarks of Intel Corporation or its subsidiaries. Other names and brands may be claimed as the property of others.