Virtual BDEVs: The Secret to Customizing SPDK
Paul Luse, Intel Corporation
Fiona Trahe, Intel Corporation
Agenda

- What Is SPDK?
- Block Device Layer
- Virtual Block Devices
- The PassThru Vbev Module
- The Crypo Vbdev Module via DPDK
- Future Work
What Is SPDK?

Storage Performance Development Kit

Open Source Software
- Optimized for latest generation CPUs and SSDs
- Software building blocks (BSD licensed)
- Designed to extract maximum performance from non-volatile media

Scalable and Efficient Software Ingredients
- User space, lockless, polled-mode components
- Up to millions of IOPS per core
- Minimize average and tail latencies

Available via spdk.io
Bdev Layer Terminology

• **Bdev Layer**: The entire block device abstraction layer in the code. The public interface is in include/spdk/bdev.h and the implementation is in lib/bdev.

• **Bdev Module**: Block devices have types (NVMe, Malloc, AIO, etc.). The code to implement a specific type of block device is called a module.

• **Bdev**: An individual block device that may be sent I/O requests.

• **Base bdevs**: A bdev that handles I/O requests directly, as opposed to a virtual bdev.

• **Virtual bdevs (aka vbdevs)**: A bdev that handles I/O requests by routing them to other bdevs. *Note: This is only a distinction in terminology - all bdevs are represented in the code using the same structure and interface.*
Block device layer: 50K Foot View

- Automatic queueing of I/O requests in response to queue full or out-of-memory conditions
- Hot remove support, even while I/O traffic is occurring.
- I/O statistics such as bandwidth and latency
- Device reset support and I/O timeout tracking
- Quality of Service Features
Block device layer: 1K Foot View

Block Device Layer

Bdev Manager

I/O pool  small buffs  large buffs  zero buff

Channel

Channel

Channel

Channel

Bdev

name  descriptors  bdev/vbdev []  Flags/counters  I/O func *

Channel

Channel

Channel

Channel

Drivers

<Driver Specific API>
Virtual block devices: 50K Foot View

- Can represent multiple base bdevs and surface a single vbdev
- Can intercept I/O on the way down and the way back
- Same struct as Base bdevs, `struct spdk_bdev`
- Well defined API enables isolated IP for value added services
Virtual block devices: Examples

- **Logical Volumes**: Virtual bdevs carved out of non-contiguous regions on a larger backing bdev implemented using SPDK's Blobstore.
- **Error**: Enables the ability to inject errors at the block device layer API level.
- **GPT**: Surfaces GPT partitions as separate BDEVs.
- **PassThru**: An example/template for creating new VBDEV modules. Lots more on this in tomorrow’s lab.
- **Crypto**: At rest data encryption via the DPDK Cryptodev Framework.
The passthru vbdev module - Initialization

```c
static struct spdk_bdev_module passthru_if = {
    .name = "passthru",
    .module_init = vbdev_passthru_init,
    .config_text = vbdev_passthru_get_spdk_running_config,
    .get_ctx_size = vbdev_passthru_get_ctx_size,
    .examine = vbdev_passthru_examine,
    .module_fini = vbdev_passthru_finish
};
```

```c
/* When we register our bdev this is how we specify our entry points. */
static const struct spdk_bdev_fn_table vbdev_passthru_fn_table = {
    .destruct = vbdev_passthru_destruct,
    .submit_request = vbdev_passthru_submit_request,
    .io_type_supported = vbdev_passthru_io_type_supported,
    .get_io_channel = vbdev_passthru_get_io_channel,
    .dump_info_json = vbdev_passthru_info_config_json,
    .write_config_json = vbdev_passthru_write_json_config,
};
```
The passthru vbdev module - initialization

/* On init, just parse config file and build *list of pt vbdevs and bdev name pairs. */

static int
vbdev_passthru_init(void)

The bdev layer calls this entry point where early setup stuff can be done, in the template the conf file is parsed.

Anytime a new bdev shows up, each vbdev module gets a chance to take action in its examine() callback.

/* Because we specified this function in our pt bdev function table when we * registered our pt bdev, we'll get this call anytime a new bdev shows up. *
* Here we need to decide if we care about it and if so what to do. We *
* parsed the config file at init so we check the new bdev against the list *
* we built up at that time and if the user configured us to attach to this *
* bdev, here's where we do it. */

static void
vbdev_passthru_examine(struct spdk_bdev *bdev)
The passthru vbdev module - examine

```c
rc = spdk_bdev_module_claim_bdev(bdev, pt_node->base_desc, pt_node->pt_bdev.module);
if (rc) {
    SPDK_ERRLOG("could not claim bdev %s\n", spdk_bdev_get_name(bdev));
    spdk_bdev_close(pt_node->base_desc);
    TAILQ_REMOVE(&g_pt_nodes, pt_node, link);
    free(pt_node->pt_bdev.name);
    free(pt_node);
    break;
}
SPDK_NOTICELOG("bdev claimed\n");

rc = spdk_vbdev_register(&pt_node->pt_bdev, &bdev, 1);
if (rc) {
    SPDK_ERRLOG("could not register pt_bdev\n");
    spdk_bdev_close(pt_node->base_desc);
    TAILQ_REMOVE(&g_pt_nodes, pt_node, link);
    free(pt_node->pt_bdev.name);
    free(pt_node);
    break;
}
```

Too many steps to show all of them here, but as part of the examine() call, the vbdev module claims a base bdev and registers a virtual bdev.
The passthru vbdev module - Submission

/* Called when someone above submits IO to this pt vbdev. We're simply passing it on here */
* via SPDK IO calls which in turn allocate another bdev IO and call our cpl callback provided */
* below along with the original bdiv_io so that we can complete it once this IO completes. */

static void
vbdev_passthru_submit_request(struct spdio_channel *ch, struct spdio_bdev_io *bdev_io)
{
    struct vbdev_passthru *pt_node = SPDK_CONTAINEROF(bdev_io->bdev, struct vbdev_passthru, pt_bdev);
    struct ptio_channel *pt_ch = spdio_channel_get_ctx(ch);
    struct passthru_bdev_io *io_ctx = (struct passthru_bdev_io *)bdev_io->driver_ctx;
    int rc = 1;

    /* Setup a per IO context value; we don't do anything with it in the vbdev other */
    /* than confirm we get the same thing back in the completion callback just to */
    /* demonstrate. */
    io_ctx->test = 0x5a;

    switch (bdev_io->type) {
    case SPDK_BDEV_IO_TYPE_READ:
        rc = spdk_bdev_readv_blocks(pt_node->base_desc, pt_ch->base_ch, bdev_io->u.bdev.iovs,
                                   bdev_io->u.bdev.iovcnt, bdev_io->u.bdev.offset_blocks,
                                   bdev_io->u.bdev.num_blocks, _pt_complete_io,
                                   bdev_io);

        break;
    case SPDK_BDEV_IO_TYPE_WRITE:
        break;
    }

    return rc;
}
The passthru vbdev module - completion

```c
/* Completion callback for IO that were issued from this bdev. The original bdev_io
 * is passed in as an arg so we'll complete that one with the appropriate status
 * and then free the one that this module issued.
 */
static void
_pt_complete_io(struct spdk_bdev_io *bdev_io, bool success, void *cb_arg)
{
    struct spdk_bdev_io *orig_io = cb_arg;
    int status = success ? SPDK_BDEV_IO_STATUS_SUCCESS : SPDK_BDEV_IO_STATUS_FAILED;
    struct passthru_bdev_io *io_ctx = (struct passthru_bdev_io *)orig_io->driver_ctx;

    /* We setup this value in the submission routine, just showing here that it is
     * passed back to us.
     */
    if (io_ctx->test != 0x5a) {
        SPDK_ERRLOG("Error, original IO device_ctx is wrong! 0x%x\n",
                     io_ctx->test);
    }

    /* Complete the original IO and then free the one that we created here
     * as a result of issuing an IO via submit_request.
     */
    spdk_bdev_io_complete(orig_io, status);
    spdk_bdev_free_io(bdev_io);
}
```
The Crypto Vbdev Module

- Relies on DPDK CryptoDev
- Initially supports software encryption AESNI multi-buffer CBC
- Also supports hardware offload with Intel® QuickAssist Technology (in validation still)
- Can be layered on any bdev or vbdev
The Crypto Vbdev Module

Application

<SPDK Defined Block API>

Block Device Layer

<Driver Specific API>

Drivers

I/O

CRYPTO OPERATIONS

DPDK

CryptoDev API

AESNI PMD

QAT PMD

QAT HW

VBEDEV

BDEV

Crypto

VBEDEV

BDEV

BDEV

BDEV

2018 Storage Developer Conference. © Insert Your Company Name. All Rights Reserved.
Future Work: Compression from DPDK
### dpdk/compressdev Key Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous burst API</td>
<td>To support HW &amp; SW acceleration.</td>
</tr>
<tr>
<td>Chained Mbufs</td>
<td>To allow compression for data greater than 64K.</td>
</tr>
<tr>
<td>Compression Algorithms</td>
<td>Deflate</td>
</tr>
<tr>
<td>Compression Levels</td>
<td>-1: PMD Default&lt;br&gt;1: Fastest&lt;br&gt;9: Best Ratio</td>
</tr>
<tr>
<td>Checksum</td>
<td>#1 CRC32&lt;br&gt;#2 Adler32&lt;br&gt;#3 Combined - Adler32_CRC32</td>
</tr>
<tr>
<td>Hash Generation</td>
<td>#1 SHA1&lt;br&gt;#2 SHA256</td>
</tr>
</tbody>
</table>
compressdev components

DPDK

DPDK Compression Framework API Components
- Device Mgmnt
- Stats & Capabilities
- Operation Mgmnt
- Stream Mgmnt
- Burst

compressdev Poll Mode Drivers

- ISA-L
  - libisal.a
- ZLIB
  - libzlib.a
- QAT
  - QATH/W
- Octeontx
  - Octeontx

Hardware Accelerators
Typical compressdev API flow

Application Start

Allocate src/dst memory → Op Pool Creation → Device Config → Queue Pair setup → Device Start → Create Private Xform

Main Loop

Op allocate → Build the Ops → Enqueue Burst → Dequeue Burst → Free ops

Free Private Xform → Stop Device → Close Device → Application End
Compression - stateless

Raw data

Compression operation

Compressed data

Decompression operation

Decompressed data

Compression operation

Decompression operation