Understanding Applications Through NVMe Driver Tracing Using BPF

John Mazzie
Member of Technical Staff, Systems Performance Engineer
Micron Technology, Inc.
Agenda

- BPF and the NVMe Driver
- Application Analysis: MLPerf™ Storage
BPF and the NVMe Driver
BPF Overview

- Originally “Berkeley Packet Filter”
  - Developed to analyze network traffic

- Integrated with kernel
  - Executes sandbox programs in kernel
    - Used to trace, profile and monitor
  - Utilizes a just-in-time compiler
  - Verification Engine to protect kernel space
  - Various features supported in different kernel versions
    - Kernel 3.18 – eBPF VM
    - Kernel 5.5 – BPF Trampoline support

- BPF stack (Kernel) is limited to 512 bytes
  - Use maps to increase memory availability
Methods of Tracing Kernel/Drivers

- **Tracepoints**
  - Stable interface
    - Managed by developers over multiple kernel versions
    - Limited to the data provided by tracepoint.
- **Kprobes (Kernel Probes – kprobe/kretprobe)**
  - Can attach/register probe to virtually any instruction.
    - Attachment to none kernel methods/functions requires debug kernel.
  - Can access data not directly provided.
  - Unstable interface
    - Kernel Functions are not stable across versions
- **BPF Trampoline (kfunc/kretfunc and fentry/fexit)**
  - Interface is similar to kprobes
  - Reduced overhead from kprobes
  - Doesn’t cause events to be missed due to interruption
  - Requires kernel support (Added in mainline kernel 5.5)
Need

- **Original Multiple Tools**
  - **Blktrace**
    - Used to analyze read/write pattern that was going to the device at the block layer
    - Requires post processing to get necessary output
  - **Nvmelat**
    - Bpftrace based tool, to give latency histogram of transactions at the driver layer
    - Could miss some transactions

- **New Tool**
  - Data processing done in line
  - Collects data for every transaction
Linux Storage Stack

Applications

Page cache

VFS/File System

Write/read data

Direct I/O

BIOs

Requests

Block driver (nvme)

Block (Request Q, I/O schedule, plug/unplug)

Requests

Enqueue Tasks

Complete Tasks

HW IRQ handling

Storage Device

BIOs

nvmetrace

blktrace

Dma mapping

Host Bus Driver

Dma unmapping

Host Bus Driver

IOMMU

Host RAM

Data Transfer
NVMeTrace

- Collections information on every transaction in the nvme driver.
  - Starting LBA
  - Transaction Size/Length
  - Start Time/Completion Time/Latency
  - Process ID/Name
  - Device
  - Queue ID
  - Transaction Type
    - Read, write, flush, admin…
- Developed using libbpf
- Kernel version specific (sometimes)
Why Libbpf?

- **Bpftrace**
  - High level scripting language
  - Helpful to build tools quickly
  - Built on bcc and libbpf
  - Limited control over implementation

- **Libbpf**
  - More difficult entry point
  - More detailed control over implementation
    - Kernel space handlers
    - User space processing and output
  - CO-RE (Compile Once – Run Everywhere)
    - Can be done, might be difficult to implement depending on tool requirements
Code Flow

- **Kernel Space**
  - Memory Maps
    - Store data in program while it's being processed.
    - Use Per CPU memory maps to avoid locking of map.
  - Ring Buffer
    - Used to transfer processed data to user space.
  - Three handlers tracing functions in the NVMe driver
    - `nvme_setup_discard`
      - Handles parsing multiple discards sent as single DSM command
    - `nvme_submit_cmd`
      - Handles submission of transactions to the NVMe device queue
      - Collect information about the transaction and store in a memory map
    - `nvme_complete_rq`
      - Handles completion of transactions, called when interrupt is activated.
      - Get completion time of transaction
      - Calculate latency
      - Put processed data on ring buffer

- **User Space**
  - Loads BPF application
  - Verification is done by the JIT compiler/BPF VM
  - Handles data passed through from kernel space
Request/Command Structure

- **Request**
  - Structure containing data from block layer provided to NVMe Driver

- **nvme_iod**
  - Structure containg Nvme I/O data.
  - Exists immediately after request in memory
  - Contains nvme_request, nvme_command, nvme_queue

- **Pointers for all structures are not passed into each traced function**
  - Limits direct access and reusability of code across kernel versions
  - Tool needs access to request and nvme_command in all functions

- **Getting data from nvme_iod and request requires moving around memory**
  - Jumping between structures in memory requires knowledge of the specific structures
    - Size, members, relative memory locations
  - Function interfaces and structures are not stable across kernel version
    - Kernel versions could require recompile, or even rewrite of handler logic
Loops in BPF are hard
- Must have a defined end
- JIT compiler does a basic check
- Loop helper exists in newer kernel versions – bpf_loop

Discards are sent through Data Set Management (DSM) command
- Up to 256 discards per DSM command
- Need to loop through individual

```c
SEC("fentry/nvme_setup_discard")
int BPF_PROG(do_nvme_setup_discard, struct nvme_ns *ns, struct request *req, struct nvme_command *cmnd)
{
    int temp_index;
    struct bio *_bio = BPF_CORE_READ(req, bio);

    // max ranges = 256 for discard DSM command.
    for (int index = 0; index < 256; index++) {
        // Can't use index directly because verifier thinks it can be changed when used in bpf_map_lookup_elem
        temp_index = index;
        struct discard_data *temp_discard_data = bpf_map_lookup_elem(&discard_heap, &temp_index);
        if (temp_discard_data) {
            if (_bio == NULL) {
                temp_discard_data->slba = 0;
                temp_discard_data->length_bytes = 0;
                temp_discard_data->length_lbas = 0;
                break;
            } else {
                temp_discard_data->slba = BPF_CORE_READ(_bio, bi_iter.bi_sector) >> (BPF_CORE_READ(ns, lba_shift) - 9);
                temp_discard_data->length_bytes = BPF_CORE_READ(_bio, bi_iter.bi_size);
                temp_discard_data->length_lbas = temp_discard_data->length_bytes >> BPF_CORE_READ(ns, lba_shift);
                _bio = BPF_CORE_READ(_bio, bi_next);
            }
        } else {
            break;
        }
    }
    return 0;
}
```
nvme_submit_cmd Handler

- Generate pointers to necessary memory locations for structures
- Check if memory is available on the heap
- Start collecting available data for the event
- Check if it’s a non-admin command
  - Length = 1 (No device name)
- Stores collected information in event_map for use in nvme_complete_rq handler
nvme_complete_rq Handler

- Gets matching information from request in event_map
- Reserves space on the ring buffer
- Calculates latency
- Writes all collected data to ring buffer for user space processing.

```c
SEC("fentry/nvme_complete_rq")
int BPF_PROG(do_nvme_complete_rq, struct request *req)
{
    __u64 req_address = (__u64)req;
    struct event *info = bpf_map_lookup_elem(&event_map, &req_address);
    if (info) {
        struct event *e;
        e = bpf_ringbuf_reserve(&ring_buffer, sizeof(*e), 0); //This is allocating too slow
        if (!e) {
            bpf_printk("BUFFER FULL!
");
            return 0;
        }
        e->start_time_ns = info->start_time_ns;
        e->end_time_ns = bpf_ktime_get_ns();
        e->latency_ns = e->end_time_ns - e->start_time_ns;
        e->qid = info->qid;
        e->pid = info->pid;
        bpf_probe_read_str(e->process_name, sizeof(e->process_name), info->process_name);
        bpf_probe_read_str(e->device_name, sizeof(e->device_name), info->device_name);
        e->opcode = info->opcode;
        e->slba = info->slba;
        e->length = info->length;
        bpf_map_delete_elem(&event_map, &req_address);
        bpf_ringbuf_submit(e, 0);
    }
    return 0;
}
```
## Example Output

<table>
<thead>
<tr>
<th>start_time_ns</th>
<th>end_time_ns</th>
<th>latency_ns</th>
<th>process_name</th>
<th>pid</th>
<th>device</th>
<th>qid</th>
<th>slba</th>
<th>length_bytes</th>
<th>length_lbas</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>945661828630244</td>
<td>945661828679823</td>
<td>49579</td>
<td>systemd-udevd</td>
<td>823</td>
<td>nvme2n1</td>
<td>18,0</td>
<td>4096,8,2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>945661828720722</td>
<td>94566182874932</td>
<td>24210</td>
<td>systemd-udevd</td>
<td>823</td>
<td>nvme2n1</td>
<td>18,8</td>
<td>4096,8,2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>945661828762102</td>
<td>945661828780561</td>
<td>18459</td>
<td>systemd-udevd</td>
<td>823</td>
<td>nvme2n1</td>
<td>18,24</td>
<td>4096,8,2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>945661833805074</td>
<td>945661833822884</td>
<td>17810</td>
<td>systemd-udevd</td>
<td>823</td>
<td>nvme2n1</td>
<td>18,0</td>
<td>4096,8,2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>945661833841224</td>
<td>945661833856614</td>
<td>15390</td>
<td>systemd-udevd</td>
<td>823</td>
<td>nvme2n1</td>
<td>18,8</td>
<td>4096,8,2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>945661833869263</td>
<td>945661833884423</td>
<td>15160</td>
<td>systemd-udevd</td>
<td>823</td>
<td>nvme2n1</td>
<td>18,24</td>
<td>4096,8,2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9456618338342307</td>
<td>9456618338359766</td>
<td>17459</td>
<td>systemd-udevd</td>
<td>823</td>
<td>nvme2n1</td>
<td>18,0</td>
<td>4096,8,2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9456618338394956</td>
<td>9456618338431165</td>
<td>36209</td>
<td>systemd-udevd</td>
<td>823</td>
<td>nvme2n1</td>
<td>14,8</td>
<td>4096,8,2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9456618338451645</td>
<td>9456618338466984</td>
<td>15339</td>
<td>systemd-udevd</td>
<td>823</td>
<td>nvme2n1</td>
<td>14,24</td>
<td>4096,8,2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>945661839510777</td>
<td>945661839552986</td>
<td>42209</td>
<td>systemd-udevd</td>
<td>55562</td>
<td>nvme2n1</td>
<td>31,0,4096,8,2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>945661839579855</td>
<td>945661839596465</td>
<td>16610</td>
<td>systemd-udevd</td>
<td>55562</td>
<td>nvme2n1</td>
<td>31,0,4096,8,2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>945661839609995</td>
<td>945661839625125</td>
<td>15130</td>
<td>systemd-udevd</td>
<td>55562</td>
<td>nvme2n1</td>
<td>31,0,4096,8,2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
BPF Helpers

- **bpf_ktime_get_ns()**
  - Get current kernel timestamp

- **bpf_get_current_comm()**
  - Gets process name of process that triggered event being traced

- **bpf_get_current_pid_tgid()**
  - Gets PID of process that triggered event being traced

- **BPF_CORE_READ()**
  - Reads memory space of structures
  - Can read arbitrarily deep through structures with pointers.

- **bpf_probe_read_kernel()**
  - bpf_core_read
  - Read arbitrary memory location

- **bpf_probe_read_str()**
  - bpf_core_read_str
  - Reads string value and stores it in another point in memory
BPF CO-RE

- CO-RE
  - Compile Once – Run Everywhere
    - Compile once and execute on multiple kernel versions

- Helper functions and methodology that help develop portable applications

- BTF
  - BPF Type Format
  - Debug information to describe all kernel/driver type information
  - Used by BPF Verifier
    - Finds matching structures and gets offsets for structure members
    - Enables ability to not have to fully define a structure to access a member of that structure
  - Build Kernel with CONFIG_DEBUG_INFO_BTF=y

Application Analysis

MLPerf™ Storage
How do we size storage for AI training?

- How do we size storage for AI training?
  - MLCommons produces many AI workload benchmarks
    - Training, Inference, Tiny, HPC, etc
  - Training benchmark has been scaled to nearly 4 thousand accelerators
  - The performance of storage has been optimized out of the Training benchmark
  - Can’t be used for measuring storage workload

Options:

- De-optimizing
  - Limit memory to the system to prevent filesystem caching
  - Some datasets are very small, and it is impossible to find a memory capacity that allows the models to train properly without caching the entire dataset

- Develop a new process
  - Must do IO in the same way as the real AI training process
  - Must reduce hardware requirements for testing
    - (few storage vendors have hundreds of GPU systems for load testing)
  - Must provide larger datasets to limit effect of filesystem caching
### MLPerf™ Storage Benchmark

- **Using the tool DLIO from Argonne Leadership Computing Facility (ALCF)**
  - Uses the same data loaders as the real workload (pytorch, tensorflow, etc) to move data from storage to CPU memory
  - Implements a sleep in the execution loop for each batch
    - Sleep time is computed from running the real workload
  - A sleep time and batch size effectively defines an accelerator
    - How much data per batch and how long to spend on forward & backward passes
  - Scale up/out testing performed by adding clients running DLIO and using MPIO for multiple emulated accelerators per client

- **MLPerf™ Storage**
  - Defines a set of configurations to represent results submitted to MLPerf™ Training
  - Version 0.5:
    - BERT & Unet3D (NLP and 3D medial imaging)
  - Allows scale out and scale up testing without requiring GPUs
  - Reported metrics are:
    - Samples per Second
    - Number of supported accelerators
  - Requires maintaining a minimum Accelerator Utilization for a passed test
  - Still in early development
  - Get involved!
    - [https://mlcommons.org/en/groups/research-storage/](https://mlcommons.org/en/groups/research-storage/)
Unet3D
I/O throughput versus time

- For a single Accelerator (top plot)
  - Data transferred in 1 second intervals ranges from 0 to 600 MB with peaks to 1,600 MB
  - The peaks correspond to the start of an epoch where the prefetch buffer is filled before starting compute
- For 15 accelerators (bottom plot)
  - Near the drive’s limit (17 accelerators)
  - Workload continues to have bursty behavior with some 1 second intervals showing 0 MB transferred
- While the system does hit the maximum throughput of the device, the low QD and idle times result in an average throughput that is 15 – 20% less than max supported
  - Traditional tools will not show the peak throughput as measured here
Unet3D
Queue depth versus time

- **1 accelerator (top plot):**
  - Over time, queue depth remains low (less than 10)
  - After initial ramp, QD remains constant even during epoch starts which showed higher MB per second

- **15 accelerators (bottom plot):**
  - Queue depth peaks at 145 early then stabilized at 120 and below
  - This heavily loaded system still has low Queue Depth operations
Unet3D
Percent of I/Os by queue depth for 1 accelerator

- For 1 accelerator:
  - Less than 1% of IOs are at Queue Depths 2-5
  - Nearly 50% of IOs were inserted as the only IO in the queue
  - Nearly 50% were inserted as the second IO in the queue (QD1)

- Note: The specific transfer size is dependent on the device, block settings, and filesystem settings but we consistently see the max available size (512KB – 1280KB)
For 15 accelerators:
- We see a distribution of Queue Depths
- The bump at low QDs is important
- A not-insignificant number of IOs are inserted at very low Queue Depths (less than 5)
- This behavior introduces idle time in workloads that were expected to be constantly high throughput
How device settings can affect I/O pattern

- **Maximum Data Transfer Size – MDTS**
  - Controller Setting
  - Sets maximum transfer size drive will accept
    - `/sys/block/nvmeXnY/queue/max_hw_sectors_kb` (Value in KiB)
  - Can be adjusted down
    - "echo <value_kb> > /sys/block/nvmeXnY/queue/max_sectors_kb"
    - `max_sectors_kb` – Working limit on OS

- **Namespace Optimal Write Size – NOWS**
  - Namespace setting – Cannot be adjust in OS
  - Hint for applications & file systems – not enforced by drive
Unet3D
I/O Blocksize Pattern 16 Accelerators – XFS Filesystem

MDTS: 4MiB / NOWS: 4KiB

MDTS: 4MiB / NOWS: 256KiB

MDTS: 512KiB / NOWS: 256KiB
Future Improvements

- Trace of files accessed
- Trace application processes
- Analysis Improvements
Please take a moment to rate this session.

Your feedback is important to us.
Reference Links

- libbpf - [https://github.com/libbpf/libbpf](https://github.com/libbpf/libbpf)
- libbpf-bootstrap - [https://github.com/libbpf/libbpf-bootstrap](https://github.com/libbpf/libbpf-bootstrap)
- MLPerf™ Storage - [https://mlcommons.org/en/groups/research-storage/](https://mlcommons.org/en/groups/research-storage/)