Radian®

MEMORY SYSTEMS

Porting Radian's Block Translation Layer for Zoned Flash to SPDK

Bob Varney

Radian Memory Systems

July 2020

Legal

Porting Radian's Block Translation Layer for Zoned Flash to SPDK

Abstract

Radian has developed a Block Translation Layer (BTL) that enables turning sequential Flash zones into conventional zones. This presentation will provide an overview of the application use cases, software architecture, and analysis of what was involved in porting it to spdk. It will conclude with a performance comparison between the BTL in kernel mode versus the BTL in spdk using the same Zoned Flash SSDs with the fio tester.

Agenda

- Why a Block Translation Layer (BTL)?
- Radian BTL Overview
- BTL Architecture
- Porting BTL to spdk
- Performance Comparison

Why a Block Translation Layer (BTL)?

Overwriting vs. Non-Overwriting

- NAND Flash is non-overwriting
- Flash in data centers is typically shared, and hence accessed by applications through a storage management layer
- Most modern storage management layers are non-overwriting

NAS

NetApp
DELL EMC

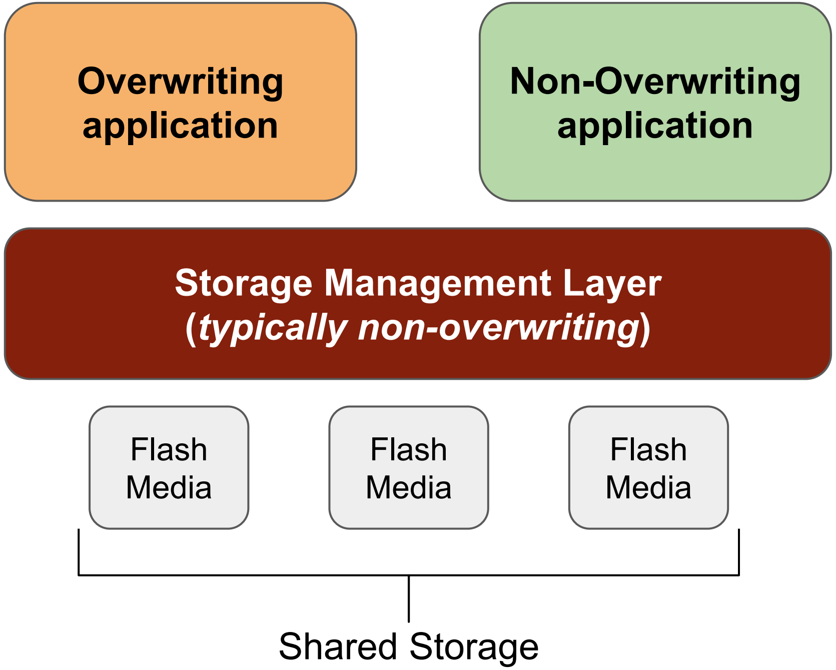
SAN (AFA)

Hewlett Packard Enterprise
PURE STORAGE
kaminario
INFINIDAT

SDS & HCI

vmware VSAN
Microsoft ReFS
Hewlett Packard Enterprise simplivity
cisco springpath

→ Software-Defined Flash (e.g., ZNS)



Overwriting application

Non-Overwriting application

Storage Management Layer (*typically non-overwriting*)

Flash Media

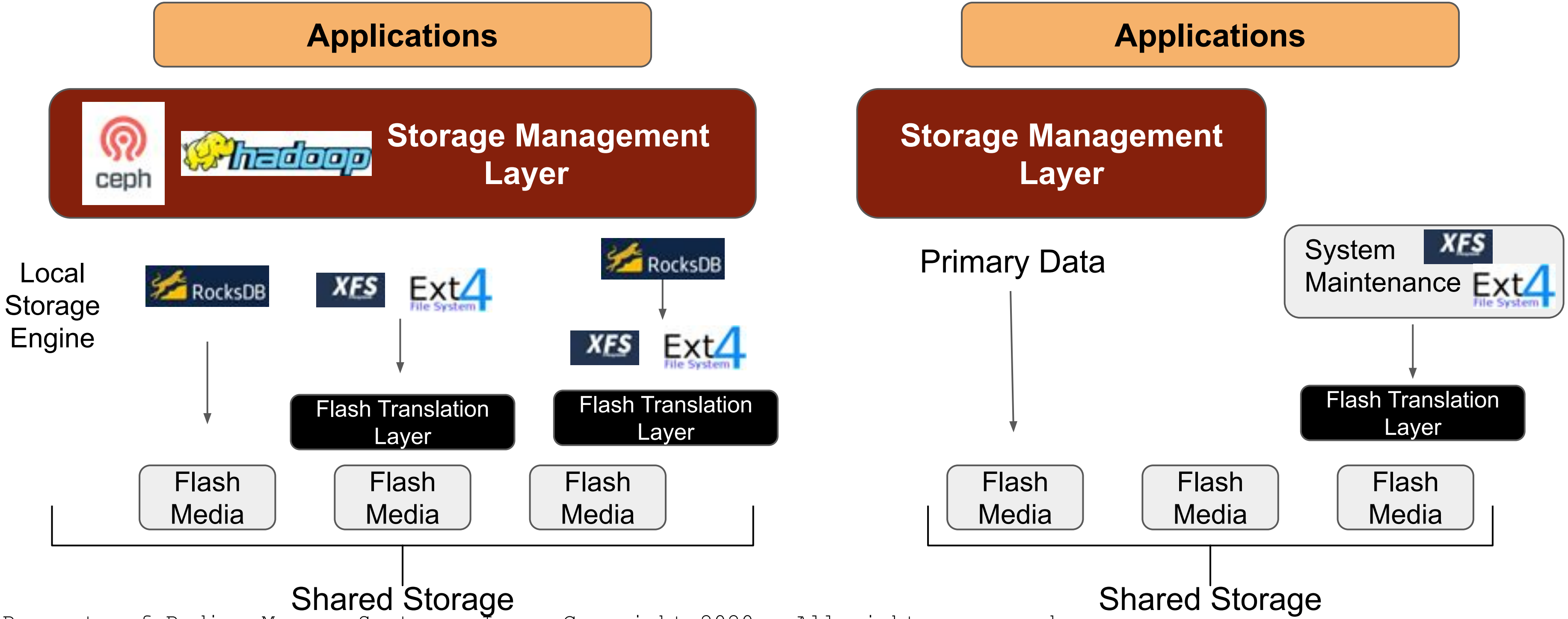
Flash Media

Flash Media

Shared Storage

Not Always

- Distributed Storage Management Layers dependent upon local storage engines or filesystems that could be overwriting
- Using in-kernel filesystems for system maintenance/backup



Applications

ceph hadoop **Storage Management Layer**

Local Storage Engine

RocksDB

XFS Ext4 File System

Flash Translation Layer

RocksDB

XFS Ext4 File System

Flash Translation Layer

Flash Media

Flash Media

Flash Media

Shared Storage

Applications

Storage Management Layer

Primary Data

System Maintenance XFS Ext4 File System

Flash Translation Layer

Flash Media

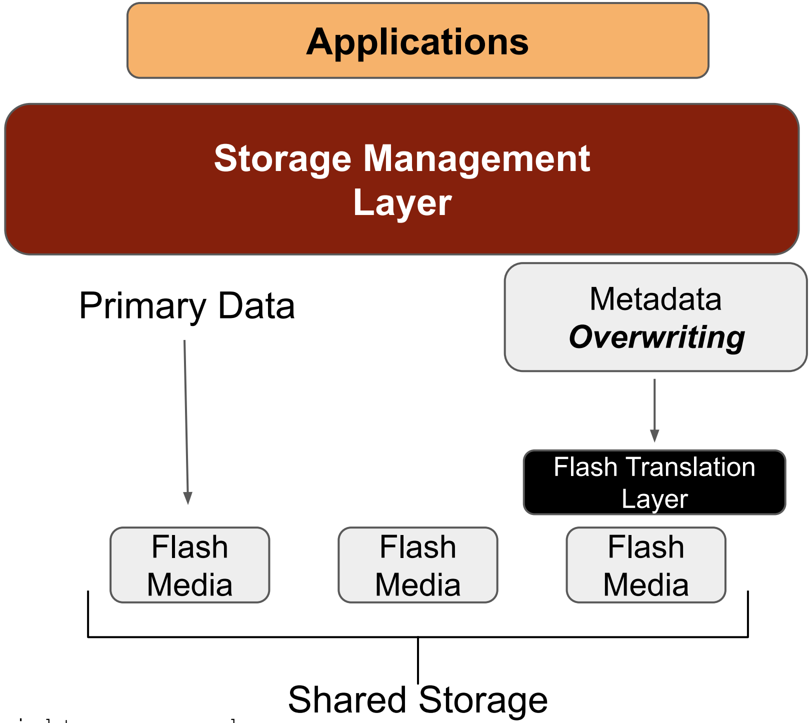
Flash Media

Flash Media

Shared Storage

Metadata Regions

- Non-overwriting Storage Management Layers may have small regions of metadata that do overwrite in-place
- Similar to using in-kernel filesystems for local system management
- High performance and high churn



JESD218 & Zoned Flash

- Endurance and Data Retention
- JESD219 workload pattern requires overwriting
- RMS-350 first Zoned Flash SSD to pass JESD218 Qual
- Targeting compliance with the NVM Express^{TM} specification for Zoned Namespaces (ZNS)

RMS-350

Block Translation Layer

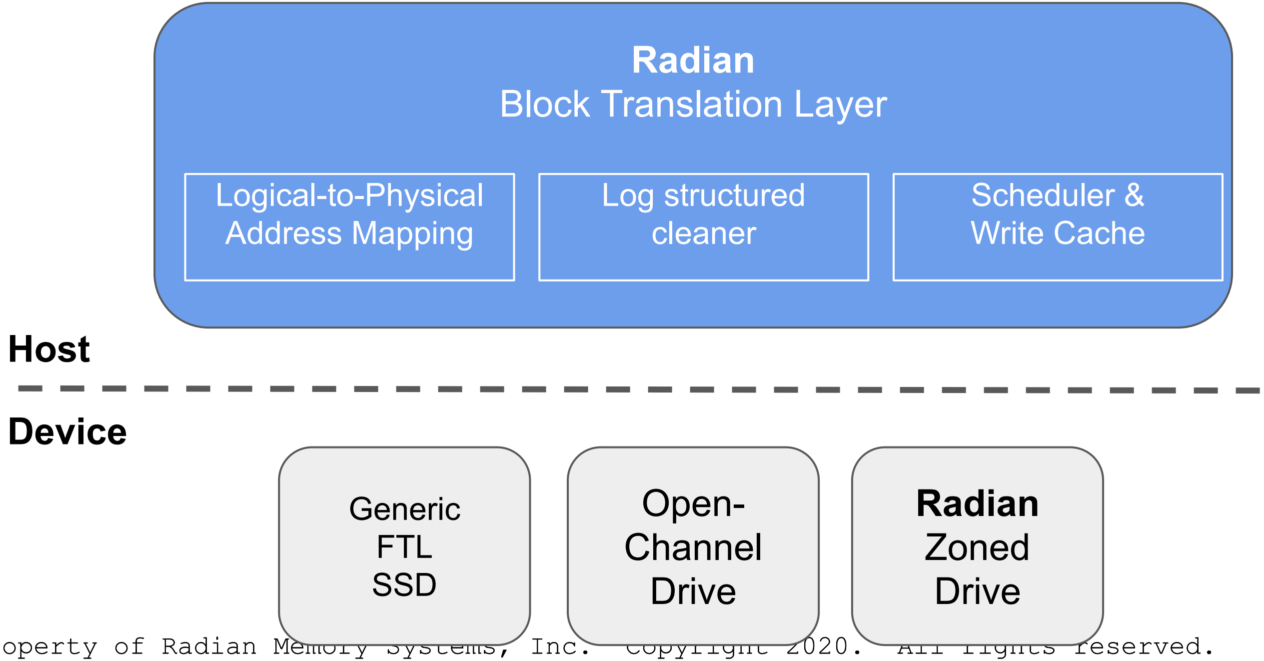
- Direct Attached Storage (DAS)
- Overwriting local filesystem under a Storage Management Layer
- Overwriting local filesystem for system maintenance
- Non-overwriting Storage Management Layer that updates metadata in-place
- Emulation of Storage Management Layers with system testing and SSD qualification testing

Radian BTL Overview

Block Translation Layer

Supports overwriting requirements

*Provides '**Conventional**' overwriting zones*



Radian
Block Translation Layer

Logical-to-Physical Address Mapping

Log structured cleaner

Scheduler & Write Cache

Host

Device

Generic FTL SSD

Open-Channel Drive

Radian Zoned Drive

General BTL Features:

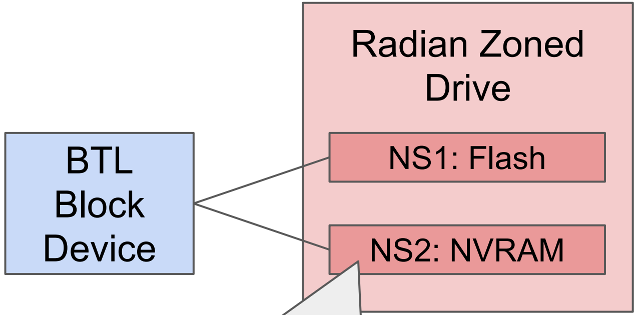
- Log structured design analogous to purpose-built Storage Management Layers
- Supports random overwriting by serializing writes
- Clean & write at variable granularities
- Runs in user-space or kernel-space host environments

With Radian Drives:

- Lower overall write-amp
- Delegated copy move
- Create multiple performance isolated devices on a single drive, or a single device spanning multiple drives

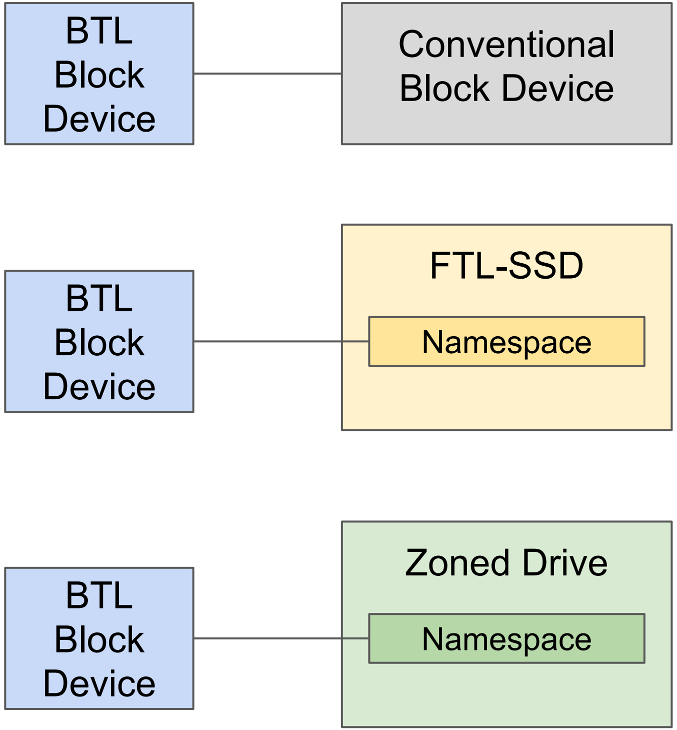
BTL Block Device Targets

(Single Drive, Single Namespace)



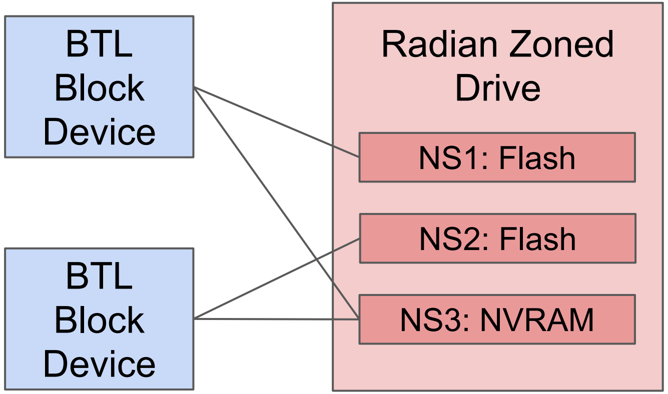
Analogous to the internal NVRAM of an FTL used, Radian's NVRAM namespace can be used here:

- as a write-cache
- as a write-ahead log



BTL Block Device Targets

(Multi-namespace)



BTL Block Device

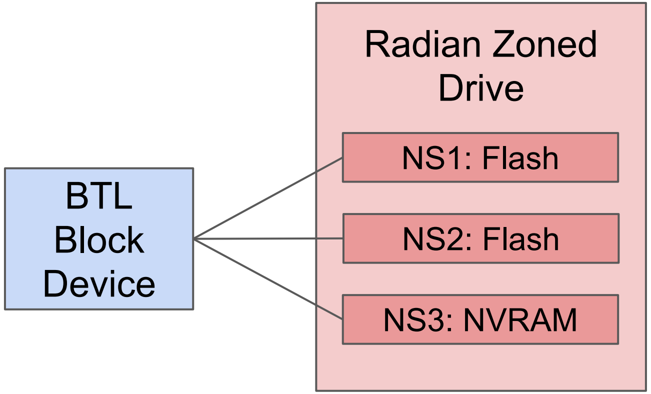
BTL Block Device

Radian Zoned Drive

NS1: Flash

NS2: Flash

NS3: NVRAM



BTL Block Device

Radian Zoned Drive

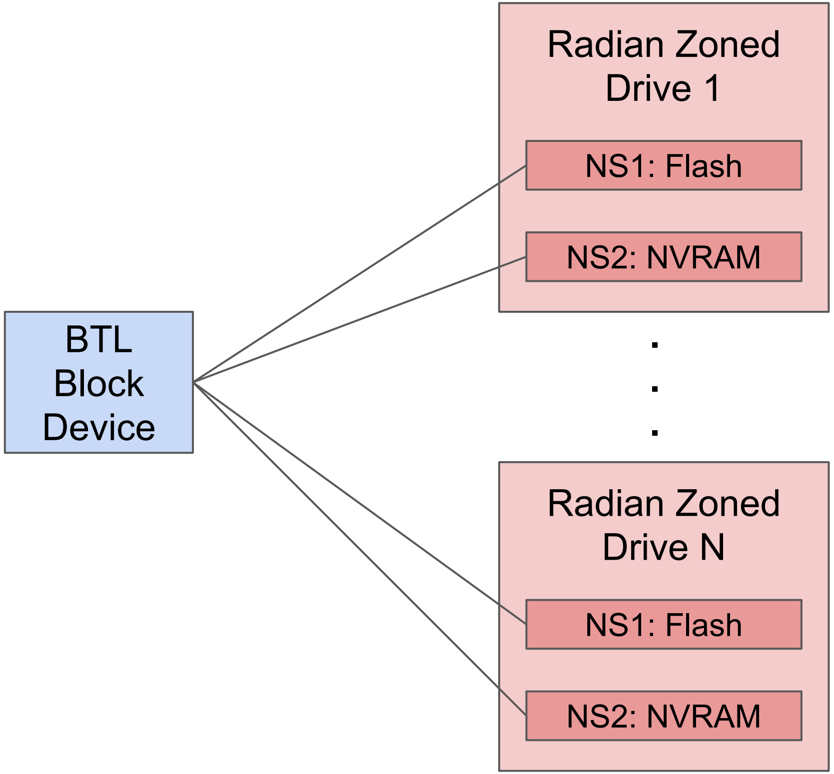
NS1: Flash

NS2: Flash

NS3: NVRAM

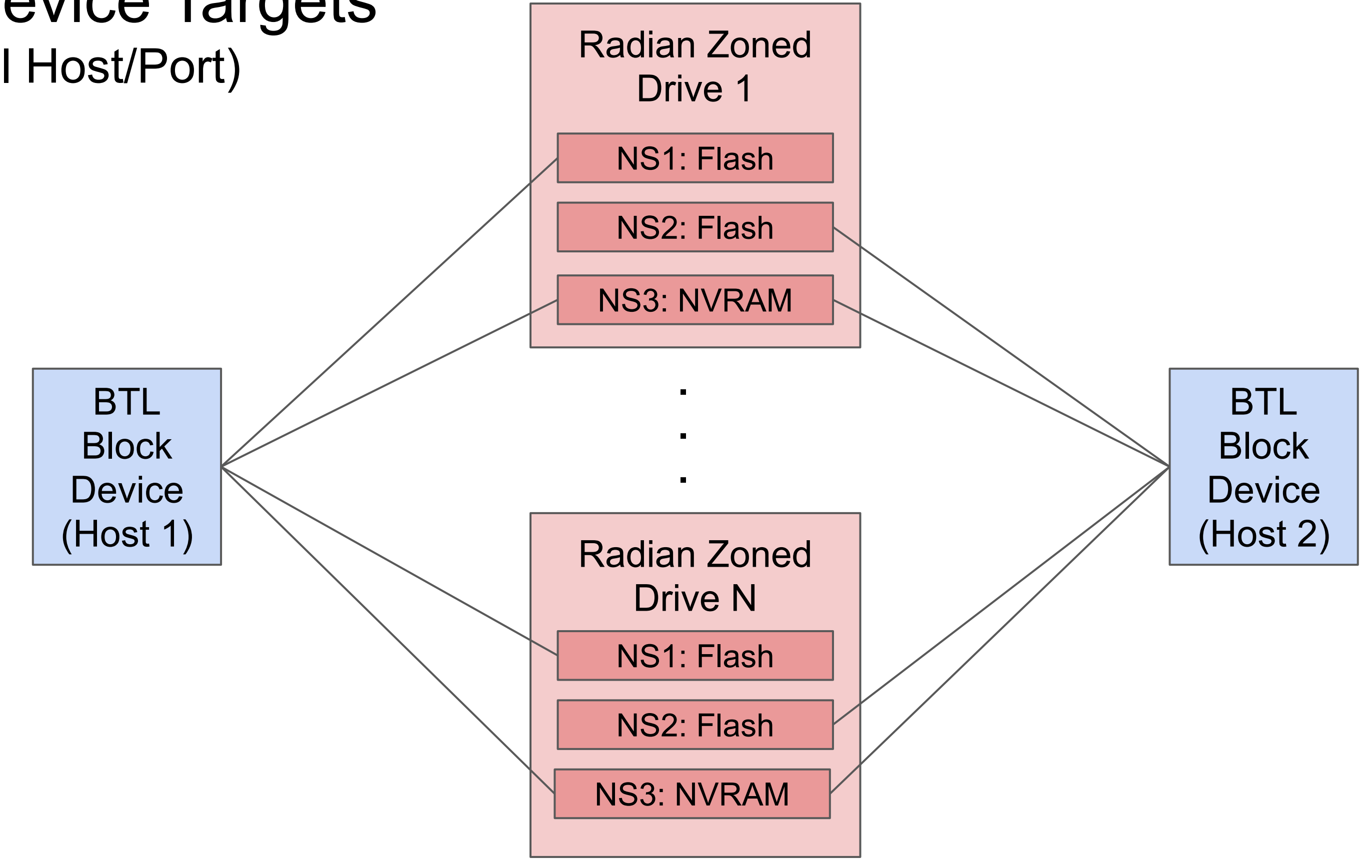
BTL Block Device Targets

(Multi-drive)



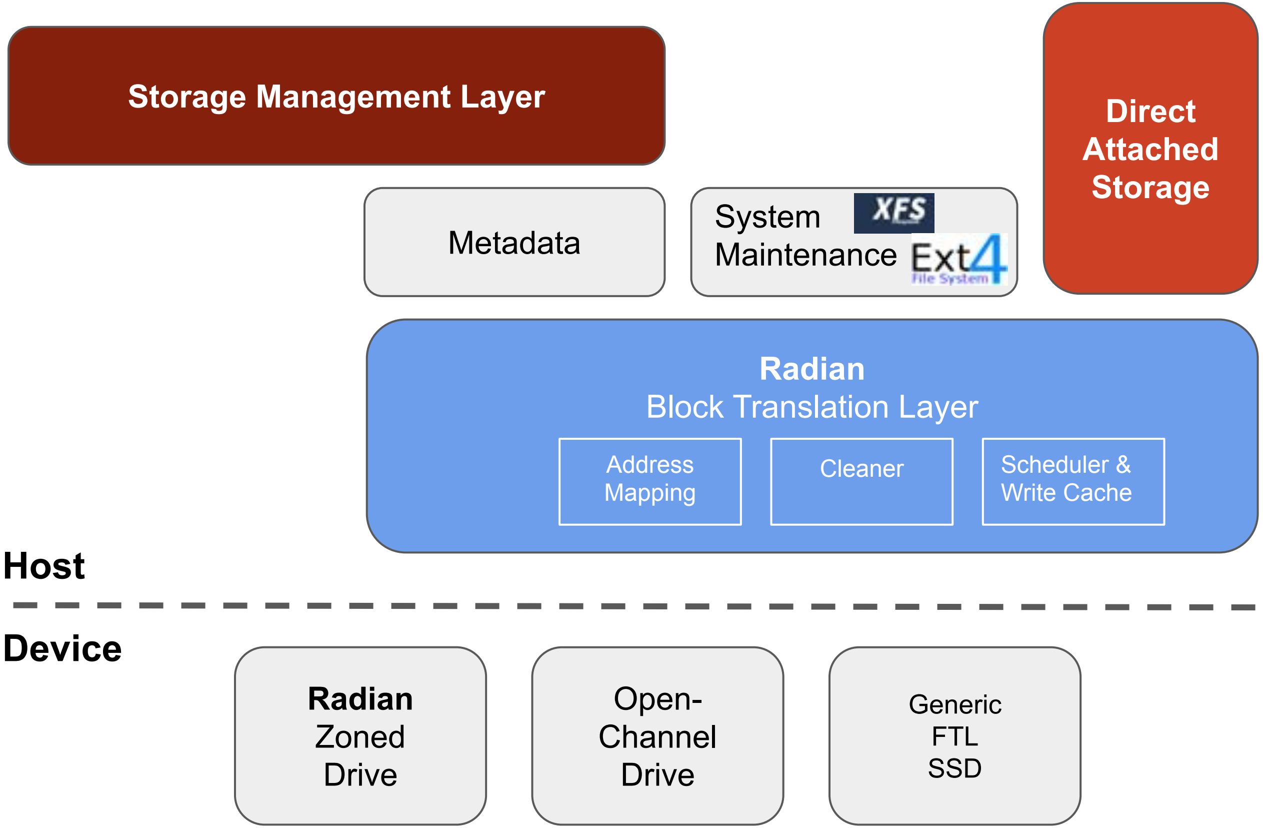
BTL Block Device Targets

(Multi-drive, Dual Host/Port)



Radian BTL Architecture

Block Translation Layer



Storage Management Layer

Metadata

System Maintenance

Direct Attached Storage

Radian
Block Translation Layer

Address Mapping

Cleaner

Scheduler & Write Cache

Host

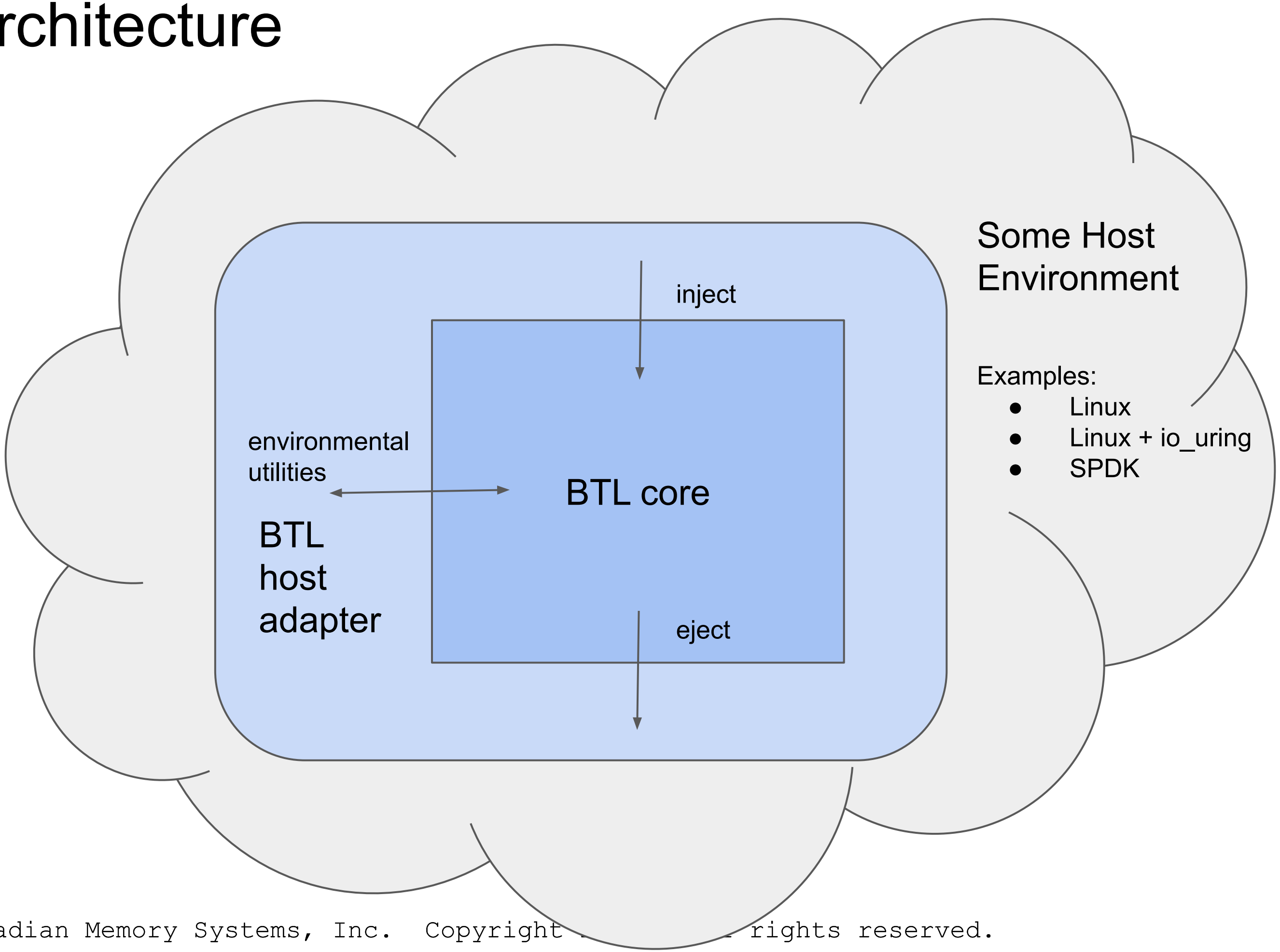
Device

Radian Zoned Drive

Open-Channel Drive

Generic FTL SSD

BTL Architecture



inject

environmental
utilities

BTL core

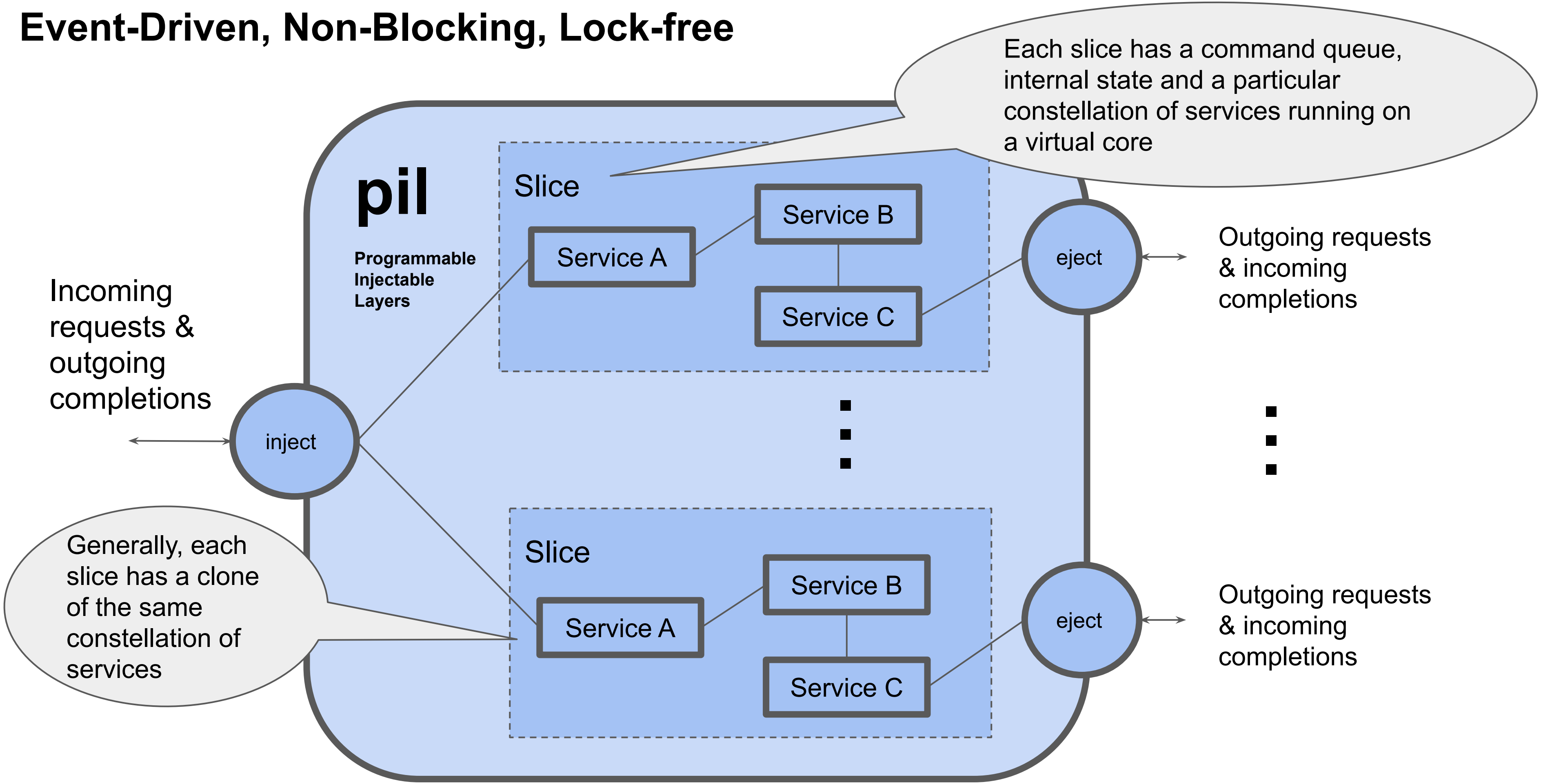
BTL
host
adapter

eject

Some Host
Environment

Examples:
- Linux
- Linux + io_uring
- SPDK

Event-Driven, Non-Blocking, Lock-free



Each slice has a command queue, internal state and a particular constellation of services running on a virtual core

Generally, each slice has a clone of the same constellation of services

Incoming requests & outgoing completions

Outgoing requests & incoming completions

Outgoing requests & incoming completions

BTL Services

inject = front door from the host environment to the hfm core, assigns requests to slices

overwrite = overwriting service (forward address map, top-level of cleaning)

zonewrite = non-overwriting service (cache mapping, address translation, reverse mapping, and the bottom-level of cleaning)

pnv = write cache/buffer for NVRAM

sfmx = ASL translation and eject command marshalling

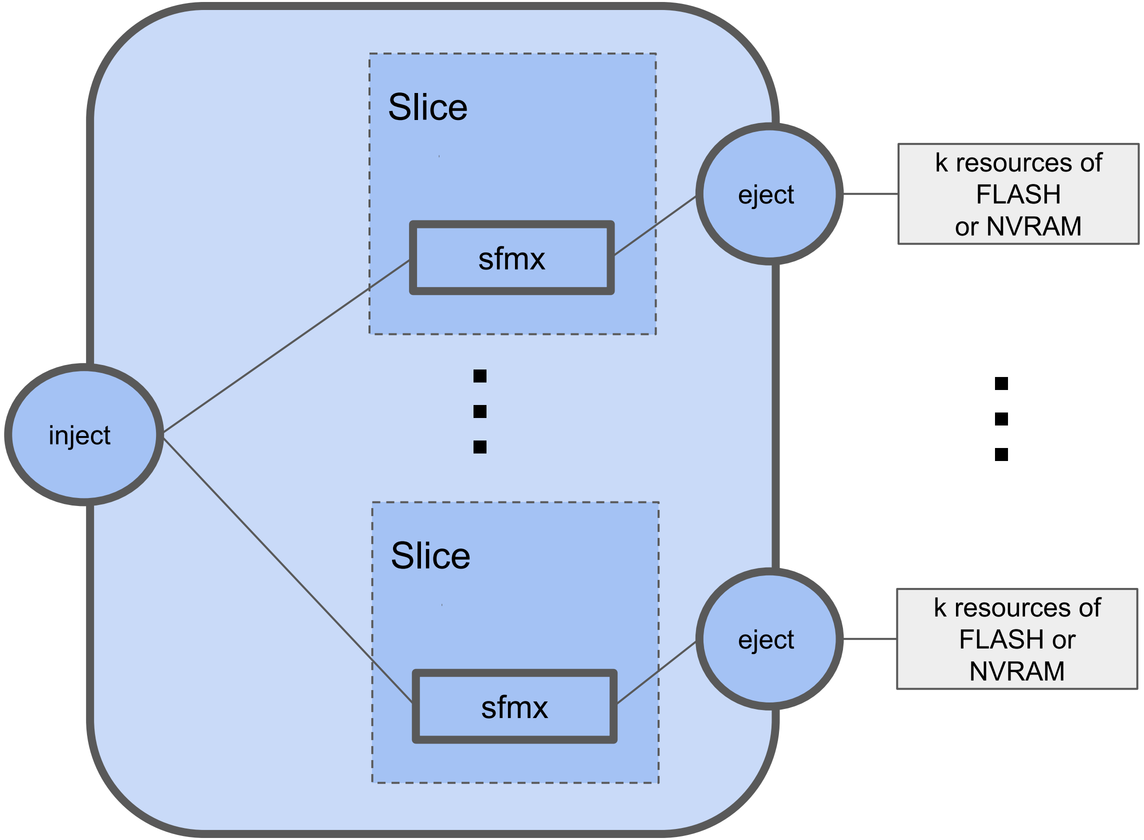
eject = back door, ushers bottom level requests from hfm back to the host environment, and receives their completions

Services and Asynchronous Command Chains

```
static void
xxx_exec_mycmd(struct pil_service *svc,struct pil_command *cmd,struct pil_dispatch *d)
{
    ...
    pil_dispatch( CMD1(svc->subsvc1, subarg,
                    CMD2(svc->subsvc2, subarg, cmd->next)), d );
    ...
}
```

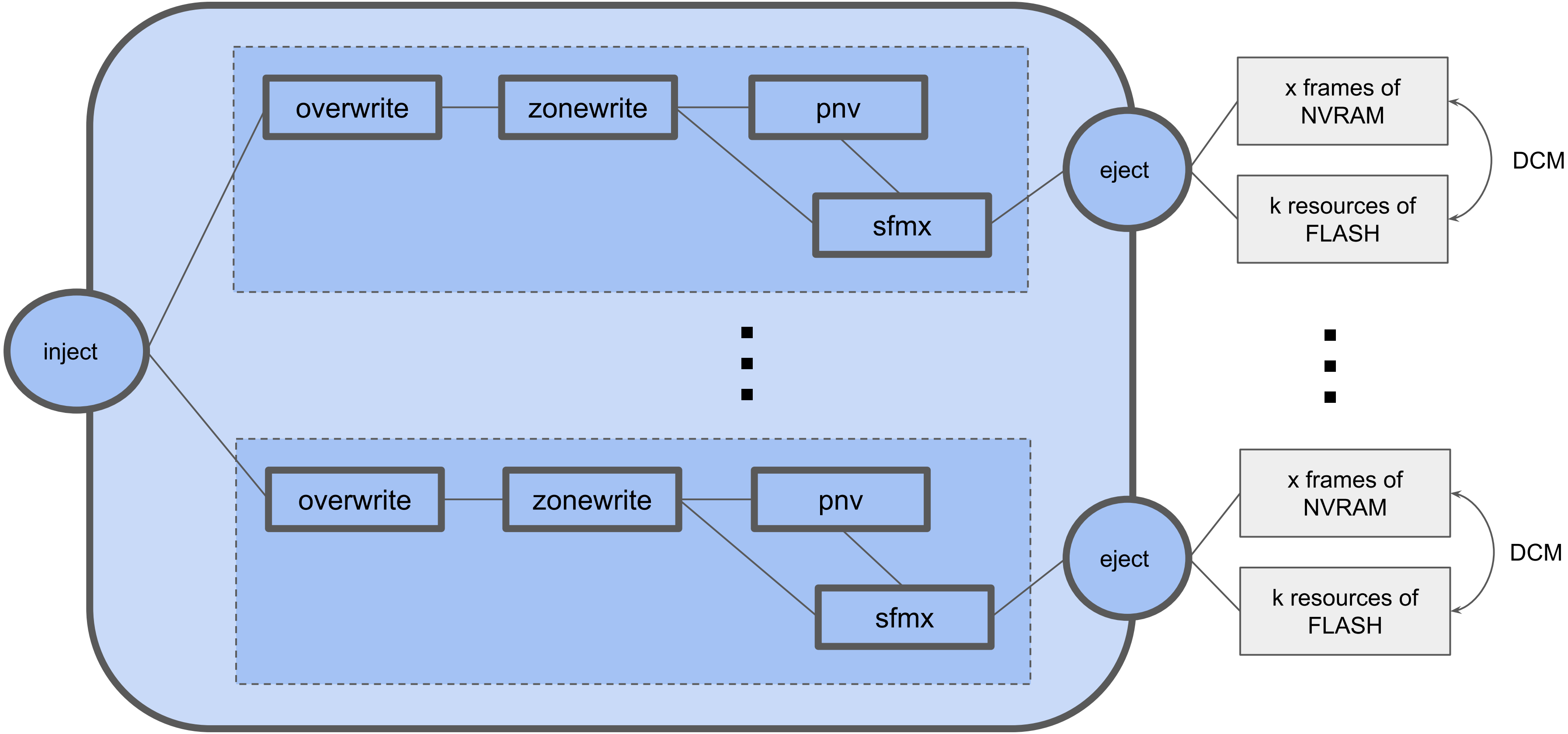
“SFMX” Mode

(Address-mapping, Non-overwriting, no caching, with ASL, no DCM)



“Overwriting” Mode

(Overwriting (MAPPING & CLEANING), CACHING, ASL, and DCMs up and down)



Major Components of BTL’s Core

inject

overwrite

zonewrite

pnv

sfmx

eject

Services

gc = garbage collection

resource = parallel resources, streams, address management

schedule = scheduling

space = store dimensions, frame types, forward and reverse mappings

btlstd = common argument types used by BTL services

pil = asynchronous execution core (defines devices, stores, targets, services, commands, and the host environment interface)

config = configuration recipes

Utilities

Porting BTL to SPDK

What is SPDK*

- Storage Performance Development Kit (SPDK):
  - Set of tools and libraries for high performance, userspace storage applications
  - Leverages DPDK (Data Plane Development Kit)
  - All drivers run in userspace, avoiding syscalls and enabling zero-copy access
  - Uses polling instead of interrupts to reduce latency and latency variation
  - Lockless I/O path based on message-passing
  - Based on a polled-mode, asynchronous, lockless NVMe driver that passively runs in userspace (spawns no threads of its own)
  - Started at Intel in 2013, open sourced in 2015 (BSD license)

* See https://spdk.io/doc/about.htm
* See Jim Harris, "*Storage Performance Development Kit: Using DPDK to accelerate storage services*," Jim Harris (Intel), DPDK Summit, San Jose, 2017, https://www.youtube.com/watch?v=4GOfsPDX_Bs

Porting BTL Abstractions

Implemented by Each Host Environment:

- Command Injection/Ejection
- Threads
- Cores
- Targets
- Memory Allocation
- Time

Implemented Once:

- Devices
- Slices
- Services
- Stores
- Address Spaces
- Forward and Reverse Address Maps
- Storage Frames
- Resources
- Segments
- Asynchronous Command Chain Execution
- Other internal queuing/scheduling

- Configuration tools

- Configuration types

Porting Device Interfaces

KERNEL

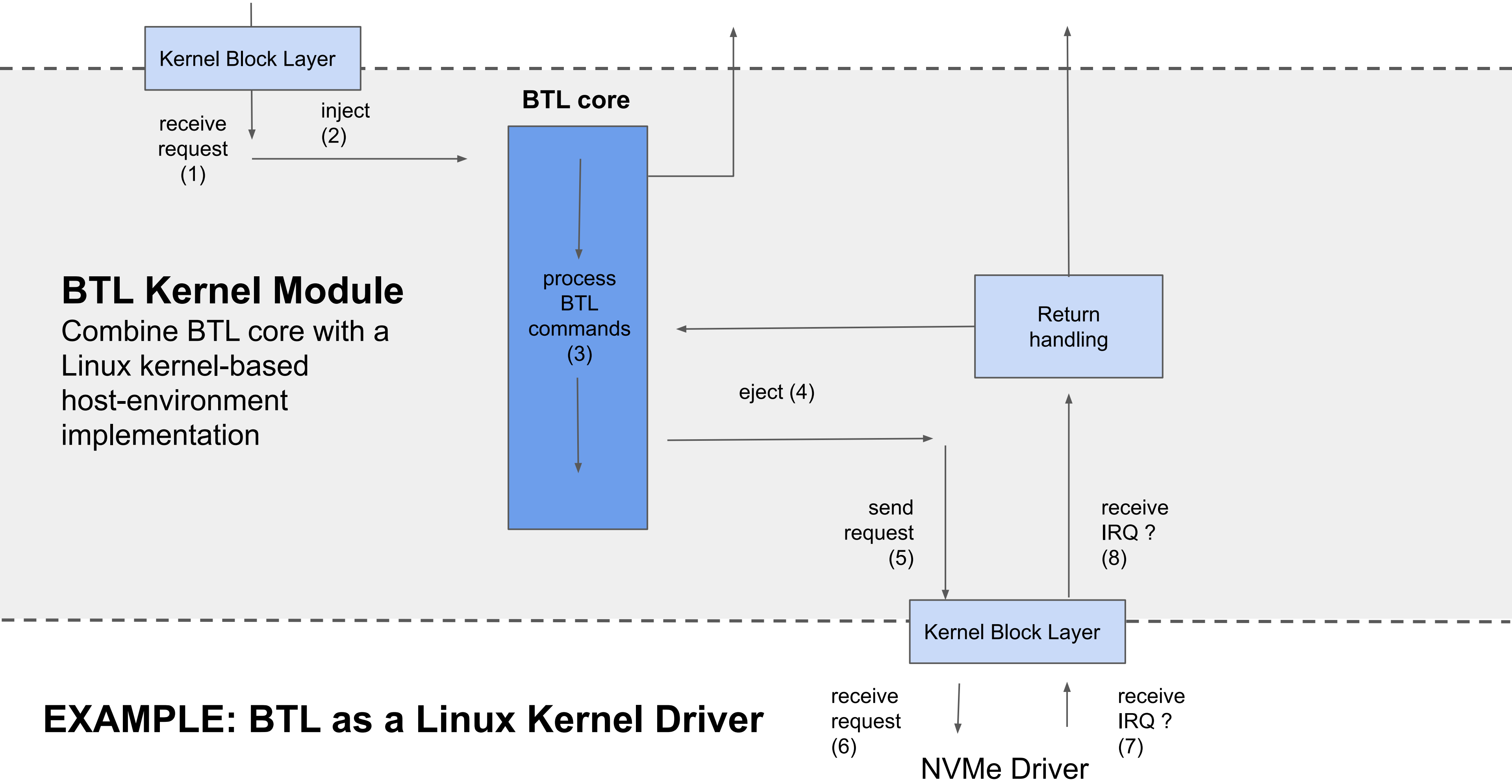
```
static const struct file_operations btl_fops = {
    .open = ...,
    ...
};

static const misdevice btl_ctrl = {
    .fops = &btl_fops,
    ...
};

...

misc_register(&btl_ctrl);

...

blk_alloc_queue(GFP_KERNEL);

...

blk_queue_make_request(dev->rq, request_func);
```

SPDK

```
static const struct spdk_bdev_fn_table btl_bdev_fn = {
    .submit_request = request_func,
};

...

spdk_io_device_register( btl_bdev,
    chan_create_cb, chan_destroy_cb, size, name );

...

spdk_bdev_register( btl_bdev->bdev );
```

An SPDK Block Device Module is the SPDK equivalent of a device driver in SPDK.

EXAMPLE: BTL as a Linux Kernel Driver



Kernel Block Layer

receive request (1)

inject (2)

BTL core

process BTL commands (3)

eject (4)

Return handling

send request (5)

receive IRQ ? (8)

Kernel Block Layer

receive request (6)

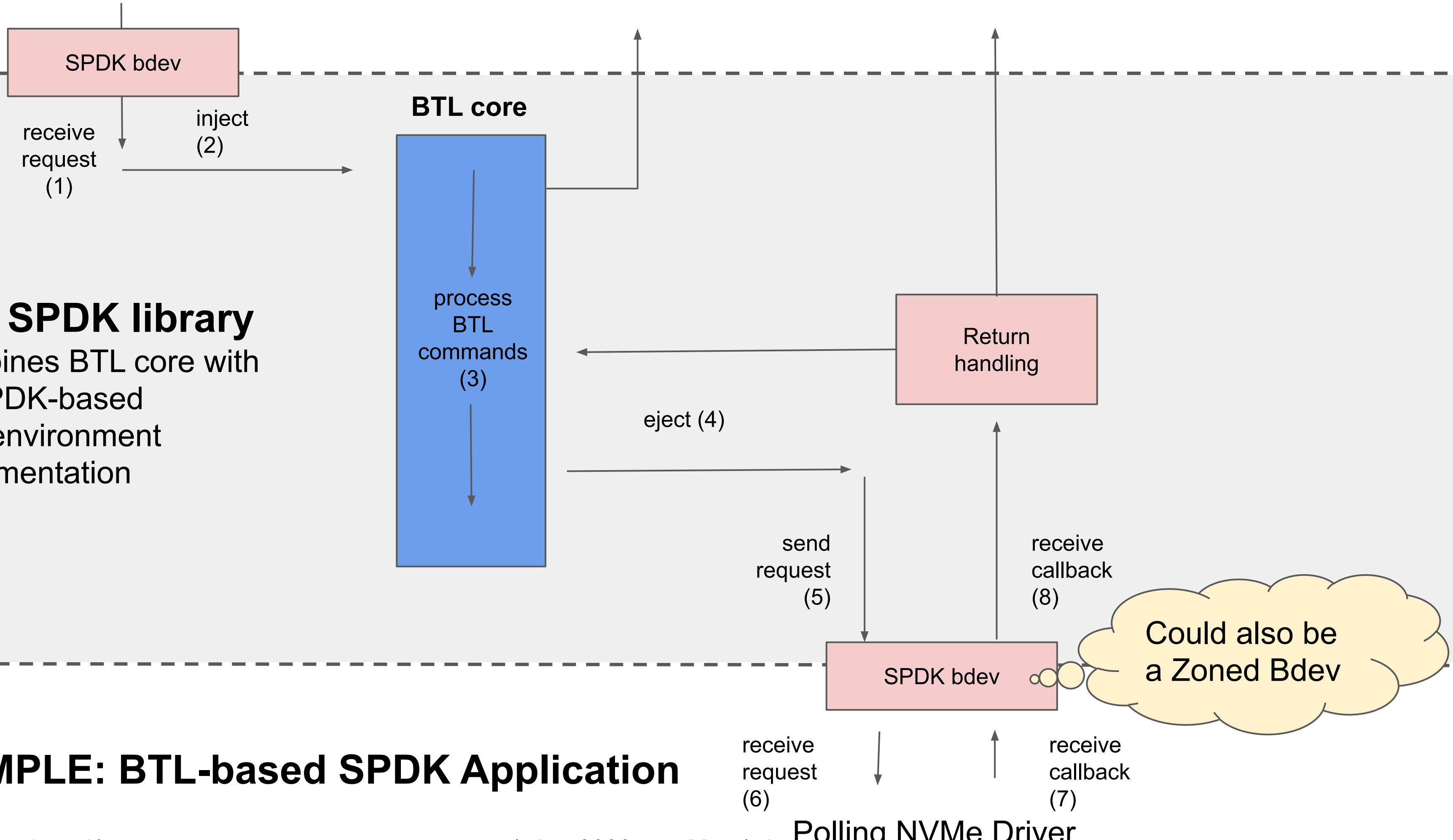
receive IRQ ? (7)

NVMe Driver

BTL Kernel Module

Combine BTL core with a Linux kernel-based host-environment implementation

EXAMPLE: BTL-based SPDK Application



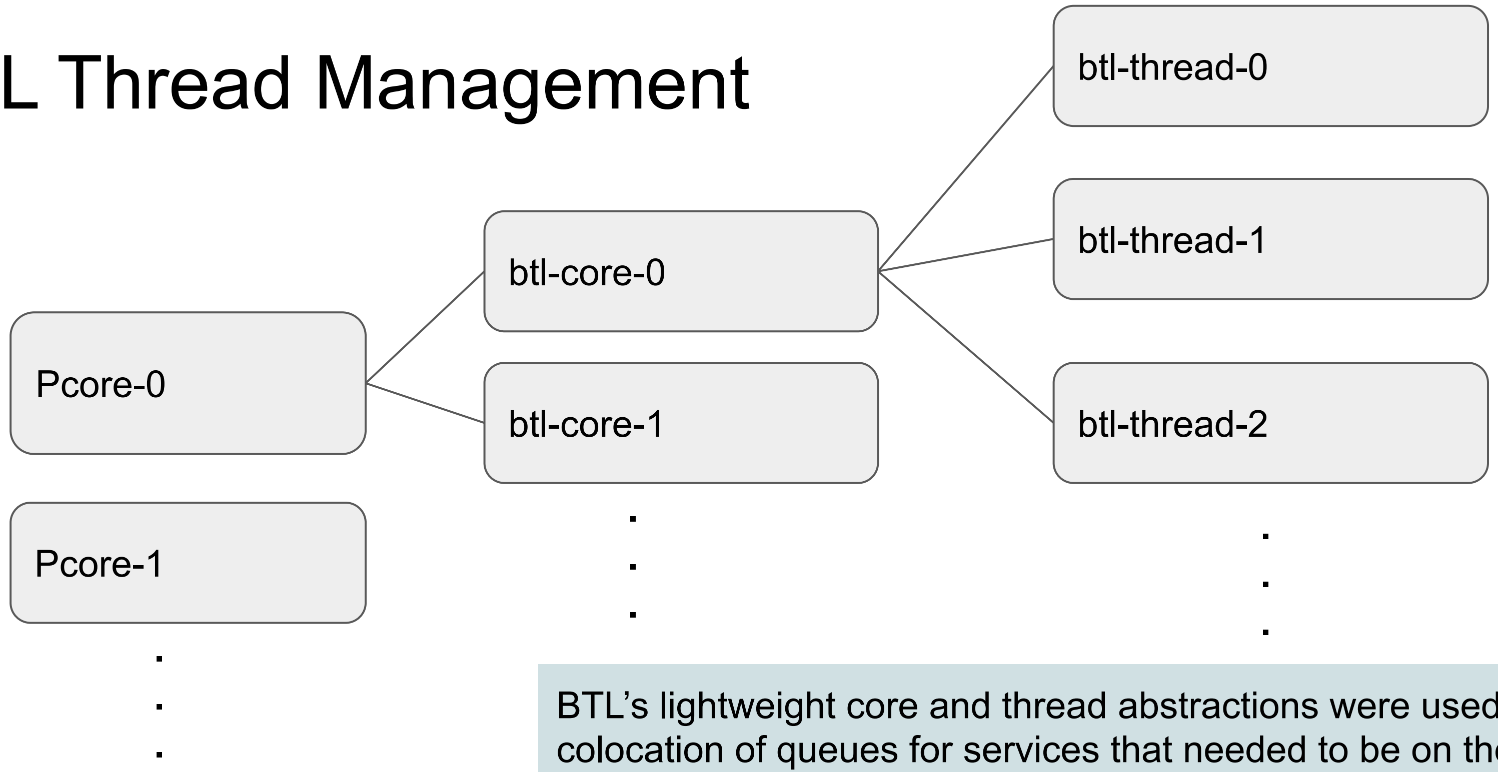
BTL SPDK library

Combines BTL core with an SPDK-based host-environment implementation

Challenges Encountered Porting to SPDK

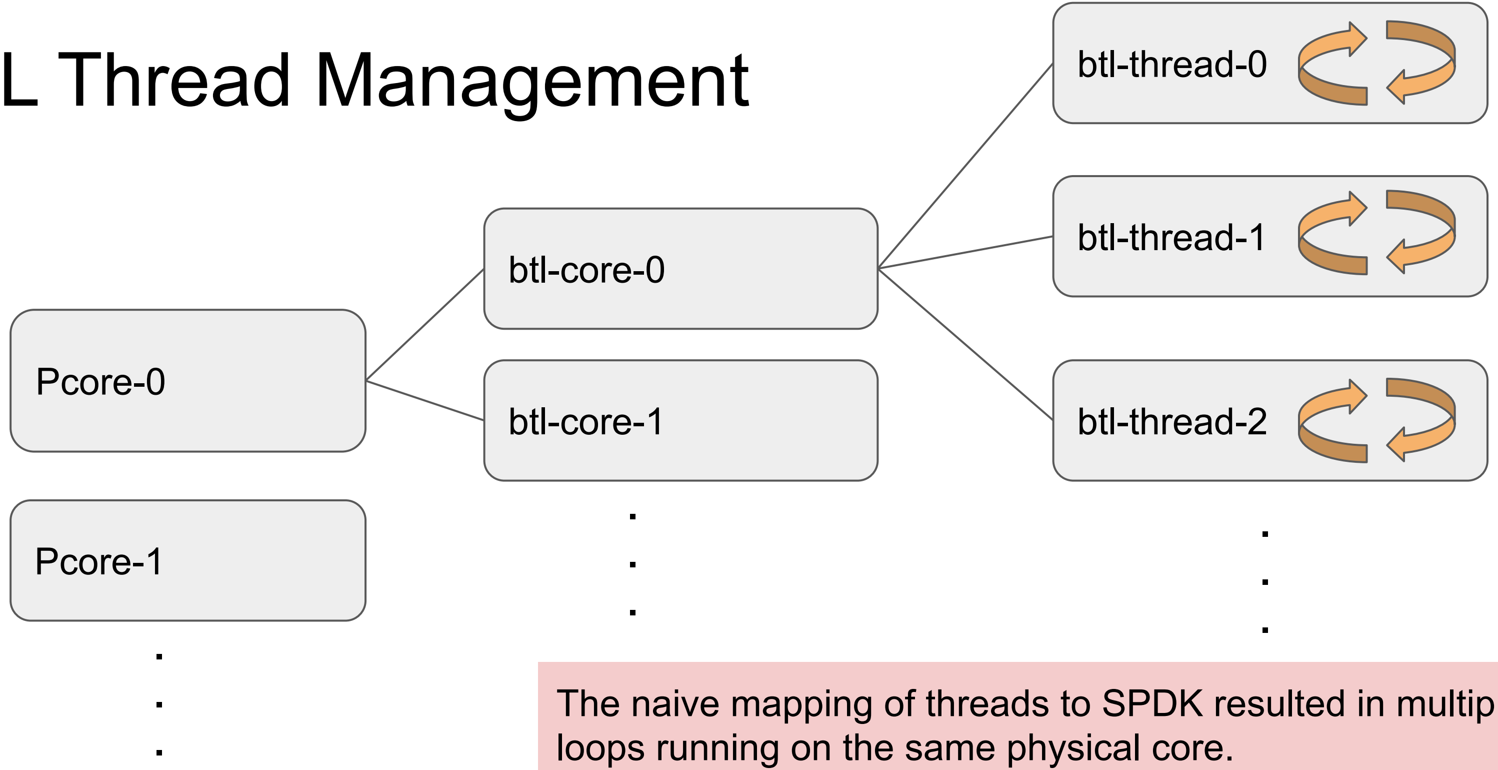
- A few latent bugs with BTL that we didn't detect in the kernel environment
- Configuration and initialization idiosyncrasies:
  - Had to refactor and make asynchronous some configuration code that had grown up in the kernel host environment
- Core/thread management issues:
  - Not all SPDK apps handle threads the same way
  - Needed to ensure a single event loop per core
  - Needed to add a configurable thread mode to determine event loop mapping

BTL Thread Management



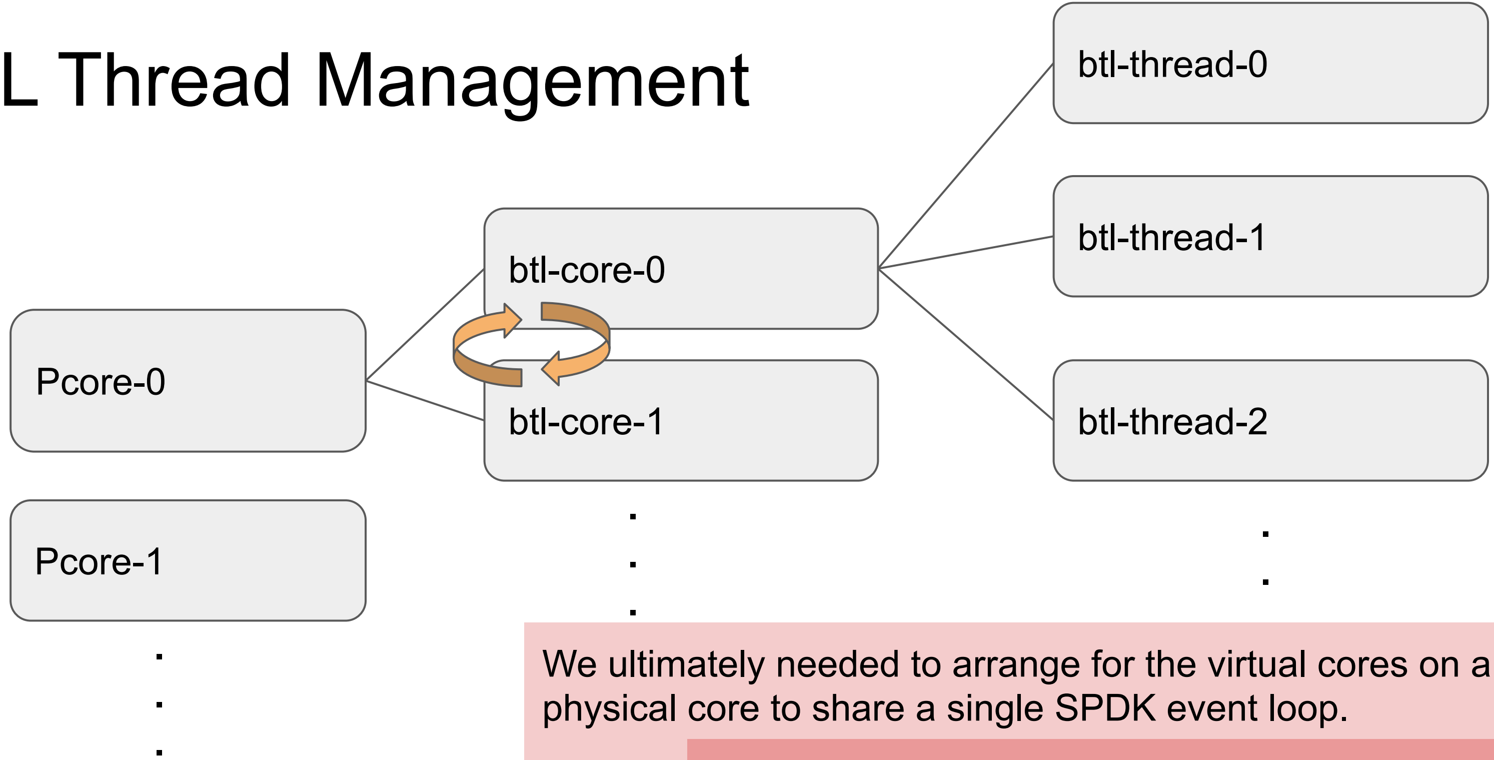
BTL's lightweight core and thread abstractions were used to colocation of queues for services that needed to be on the same thread, but there was no per-thread or per-core processing associated with these abstractions when realized in the kernel.

BTL Thread Management



The naive mapping of threads to SPDK resulted in multiple event loops running on the same physical core.

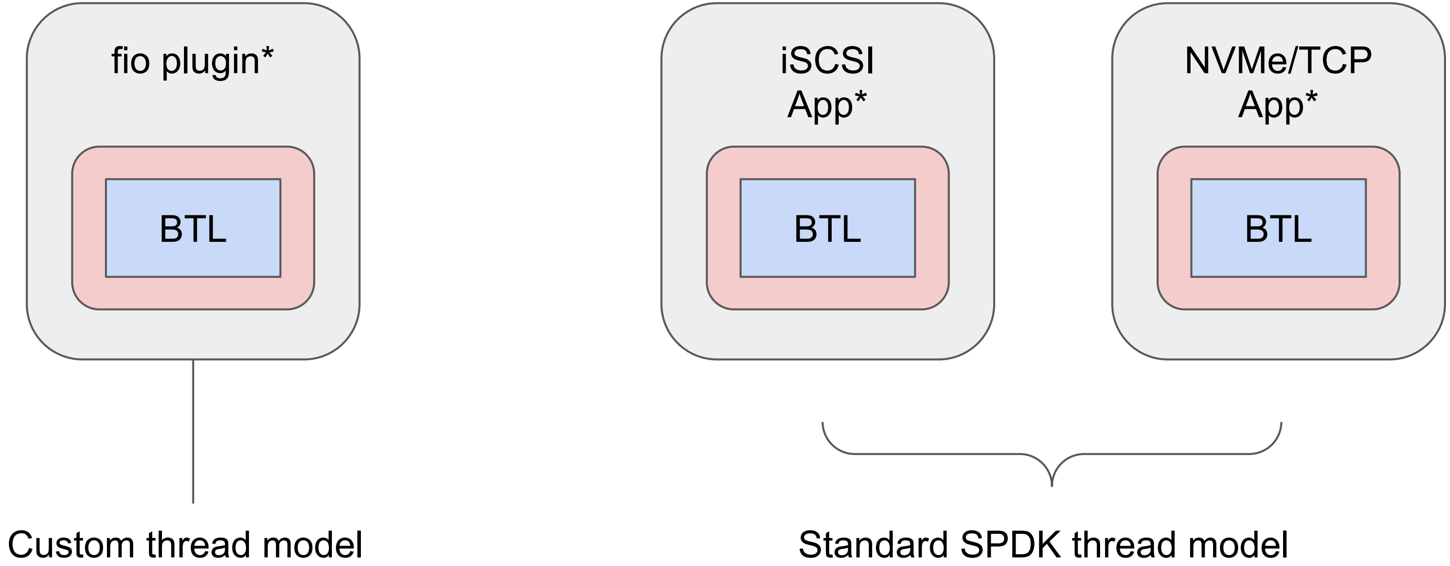
BTL Thread Management



We ultimately needed to arrange for the virtual cores on a single physical core to share a single SPDK event loop.

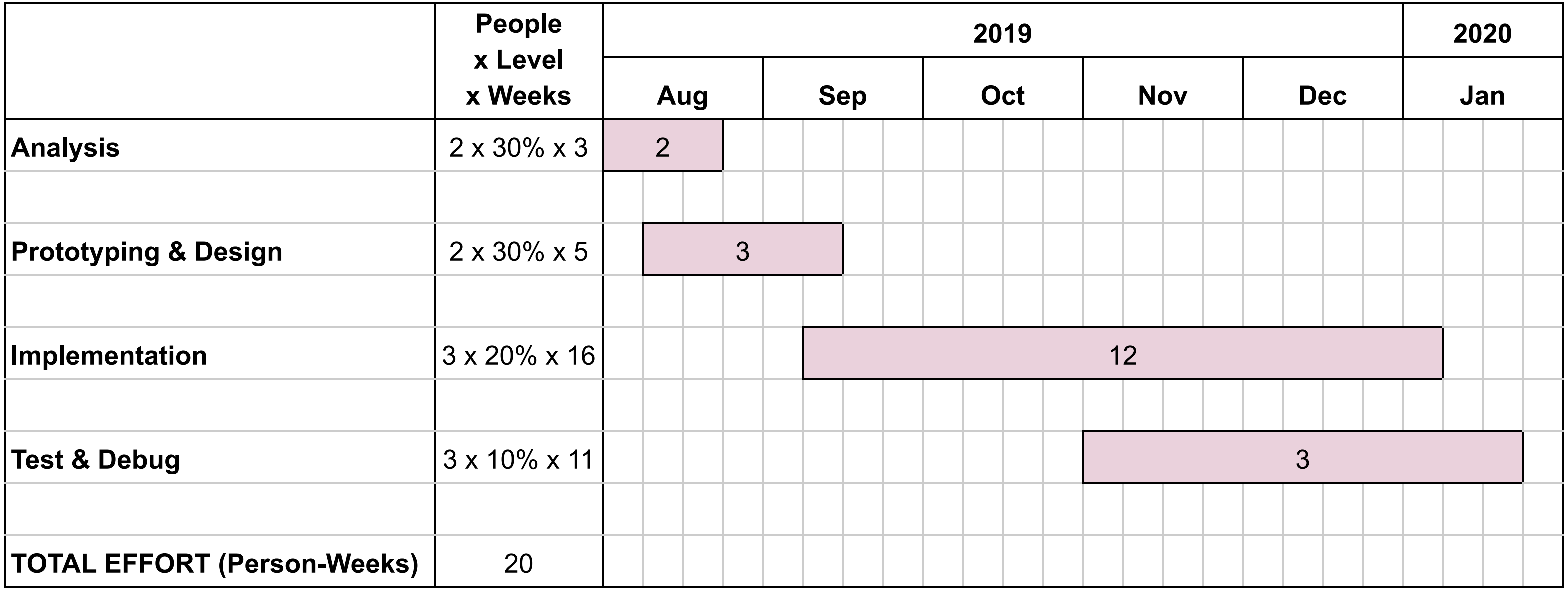
In addition, the fio plugin does NOT use the standard SPDK reactor mechanism, which led to additional changes to the thread model.

SPDK Applications



* Based on SPDK source code examples

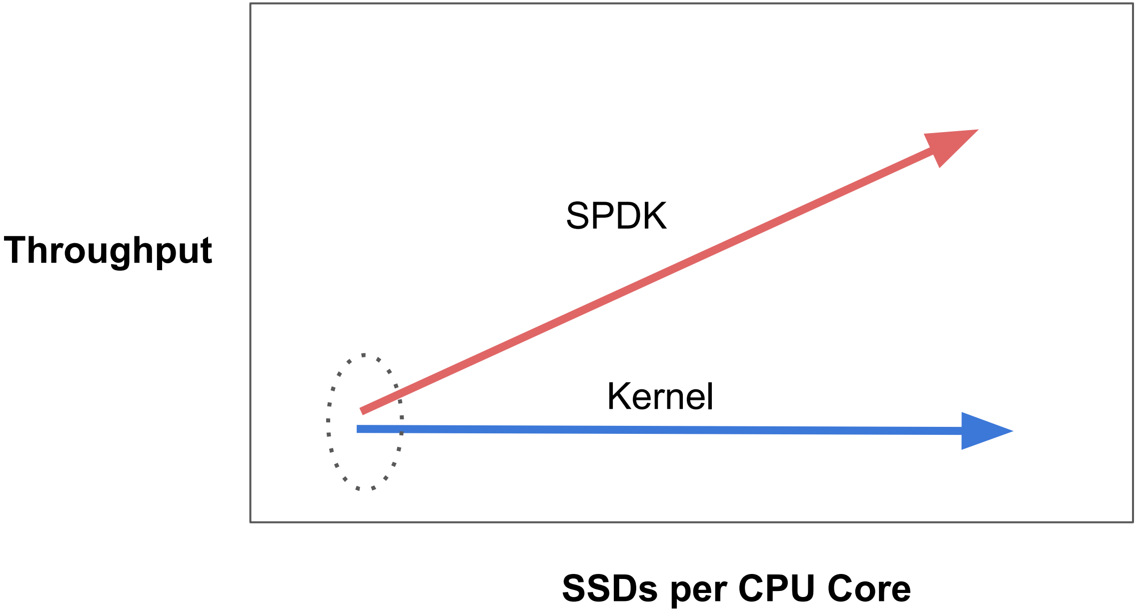
SPDK Porting Effort



	People x Level x Weeks	2019					2020
		Aug	Sep	Oct	Nov	Dec	Jan
Analysis	2 x 30% x 3	2					
Prototyping & Design	2 x 30% x 5	3					
Implementation	3 x 20% x 16				12		
Test & Debug	3 x 10% x 11					3	
TOTAL EFFORT (Person-Weeks)	20						

Performance Comparison

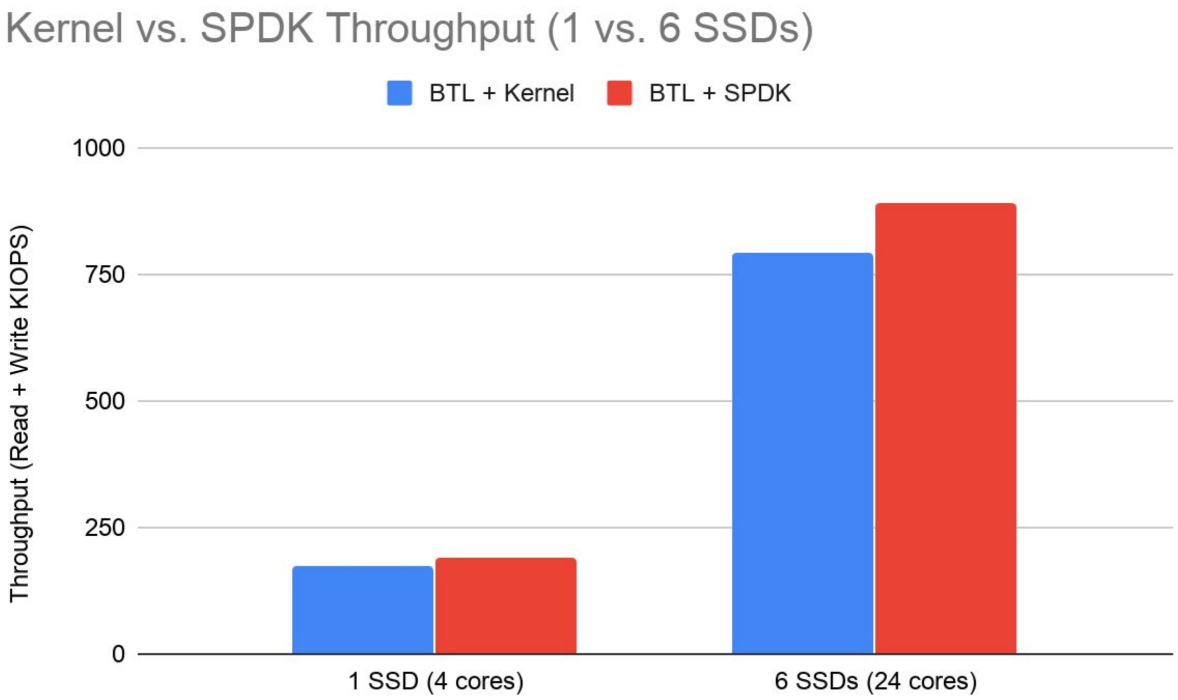
SPDK Value Proposition (Intel)



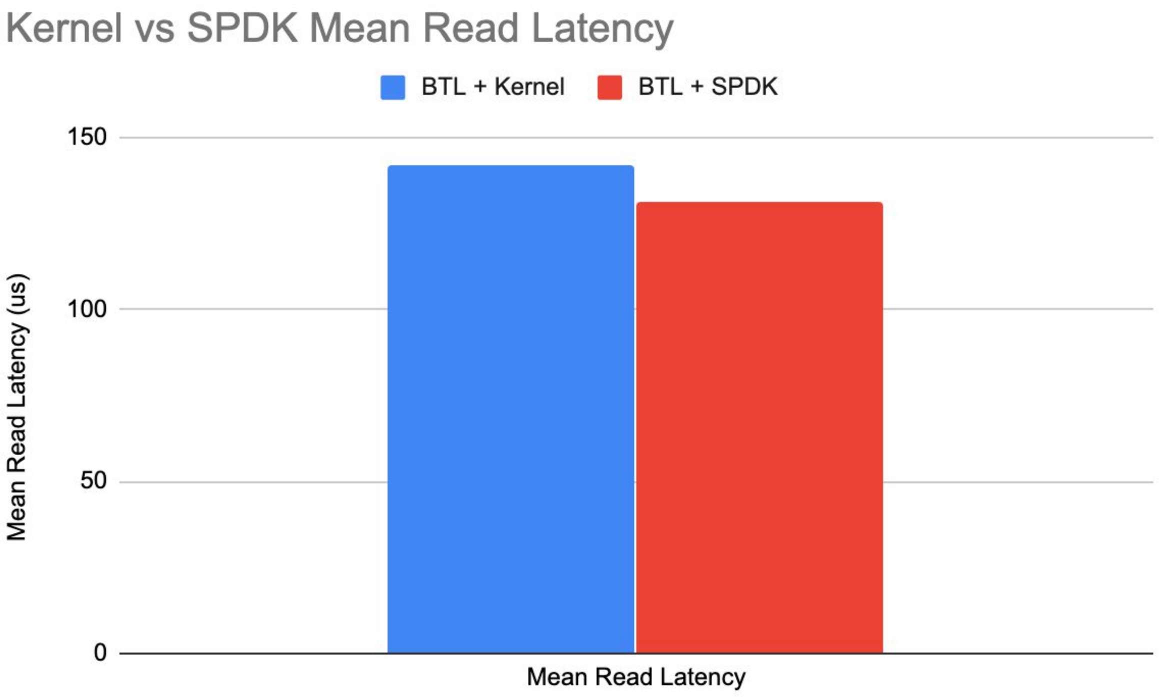
"SPDK enables (*):

- more CPU cycles for storage services
- lower I/O latency"

(*) Jim Harris, "*Storage Performance Development Kit: Using DPDK to accelerate storage services*," Jim Harris (Intel), DPDK Summit, San Jose, 2017, https://www.youtube.com/watch?v=4GOfsPDX_Bs



- RMS 350D 4TB SSDs
- 4 cores per SSD
- AIC Chassis (24 cores)
- Linux Kernel = 4.18?
- SPDK = 19.07.1
- Aggregated Dual host / Dual port
- Total K-IOPS (Read + Write)
- 70% Read / 30% Write
- Random 4K
- Queue depth 32
- Throughput as perceived by fio workload generator
- Measured after hours of preconditioning to stabilize write amplification



- RMS 350D 4TB SSDs
- 4 cores per SSD
- AIC Chassis (24 cores)
- Linux Kernel = 4.18?
- SPDK = 19.07.1
- Aggregated Dual host / Dual port
- Total K-IOPS (Read + Write)
- 70% Read / 30% Write
- Random 4K
- Queue depth 32
- Throughput as perceived by fio workload generator
- Measured after hours of preconditioning to stabilize write amplification

Performance Test Considerations

Device? CPU? Device-limited or CPU-limited?

Device preconditioned or fresh?

Workload generator (e.g., perf, bdevperf, fio)?

What software stacks are being compared?

References:

- Kariuka & Verma, “SPDK Performance Report, Release 18.04”, Intel, July 2018, https://ci.spdk.io/download/performance-reports/SPDK_nvme_bdev_perf_report_18.04.pdf
- Jim Harris, “*Storage Performance Development Kit: Using DPDK to accelerate storage services*,” Jim Harris (Intel), DPDK Summit, San Jose, 2017, https://www.youtube.com/watch?v=4GOfsPDX_Bs

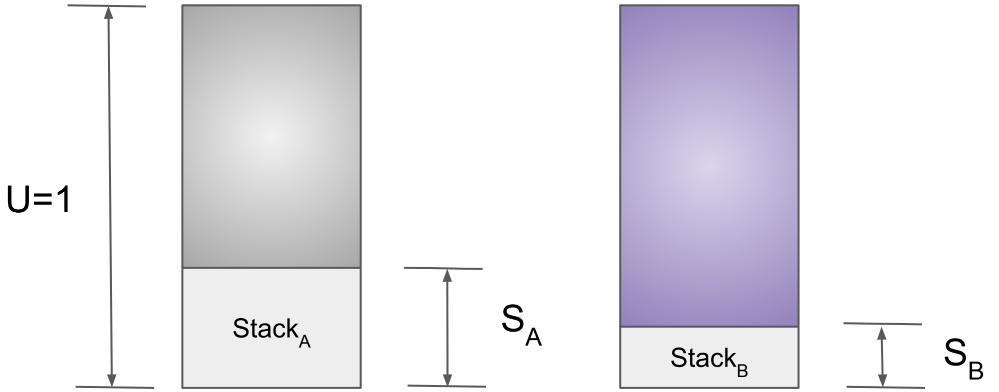
Comparing Throughput of Software Stacks

Consider CPU cost for a given, fixed amount of throughput $X = X_A = X_B$
Performance limited only by CPU (not network or device limited)

$X = X_A = S_A * M_A$

$M_A = X_A/S_A = X/S_A$

M_A = Maximum CPU-limited throughput of stack A



$X = X_B = S_B * M_B$

$M_B = X_B/S_B = X/S_B$

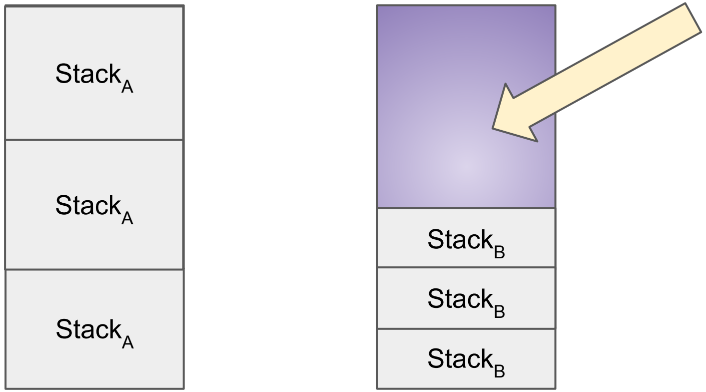
M_B = Maximum CPU-limited throughput of stack B

Maximum Throughput Ratio of Two Stacks:

$M_B / M_A = (X/S_B)/(X/S_A) = S_A/S_B$

At Maximum Throughput of Weakest Stack

There is excess CPU capacity on the stronger stack, room to add functionality

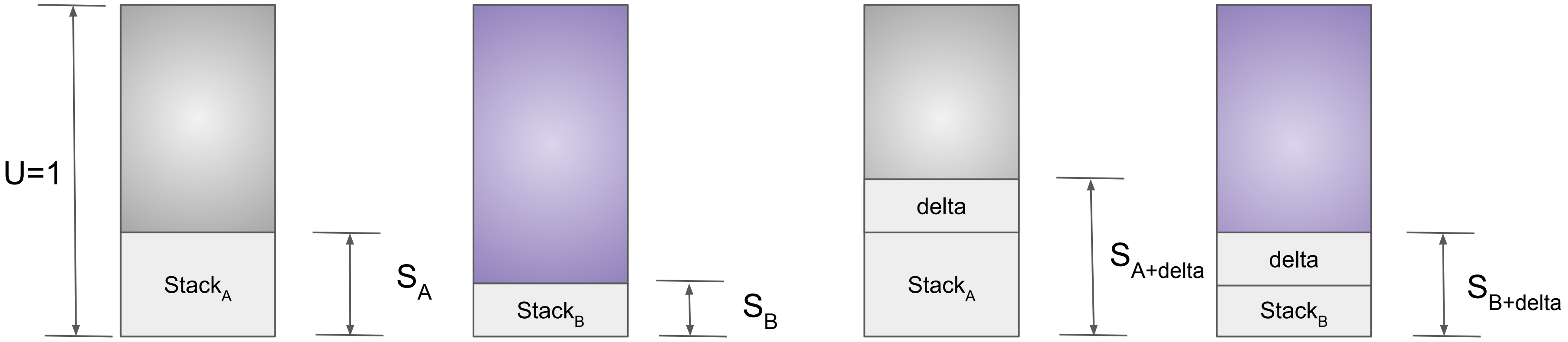


At maximum throughput M_A on $Stack_B$

Excess capacity available on $Stack_B$

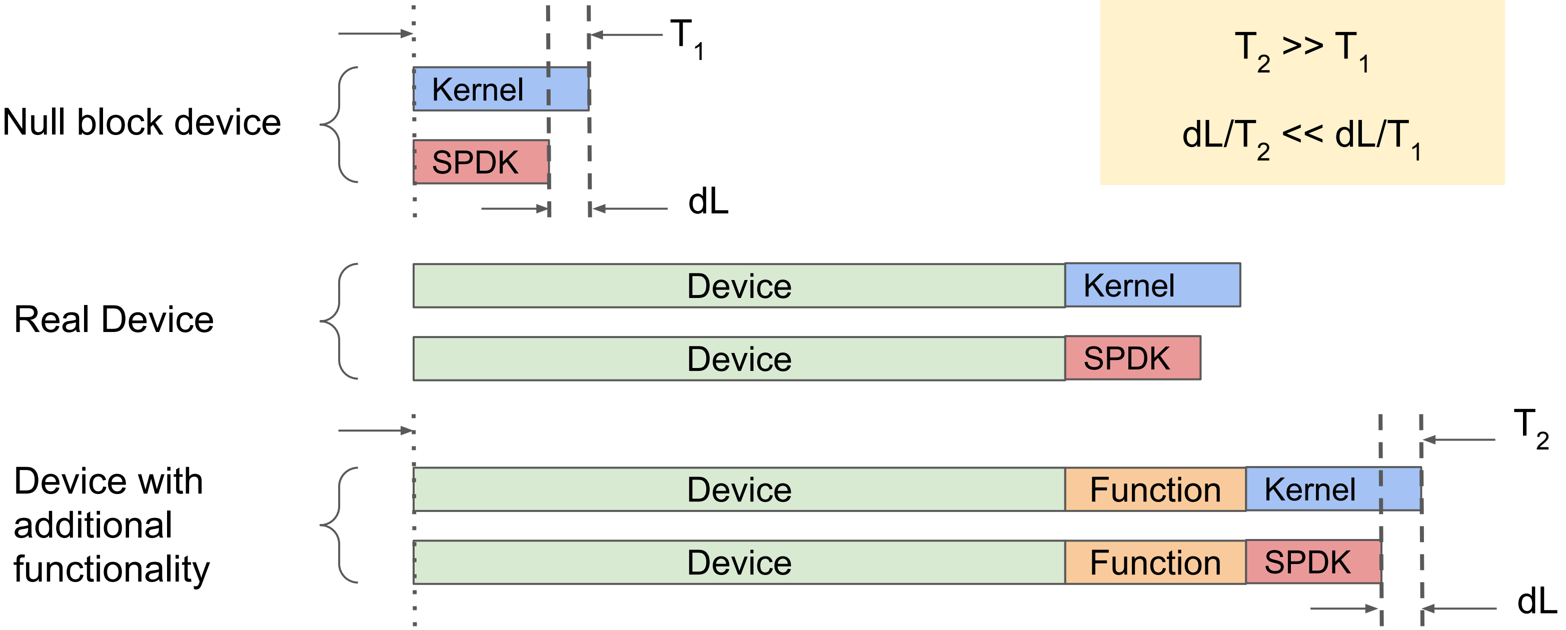
Adding Functionality

Reduces the Maximum Throughput Ratio between the two functionally equivalent stacks



$$\mathbf{M_B / M_A = S_A/S_B} \quad \mathbf{>} \quad \mathbf{M_{B+delta} / M_{A+delta} = S_{A+delta}/S_{B+delta}}$$

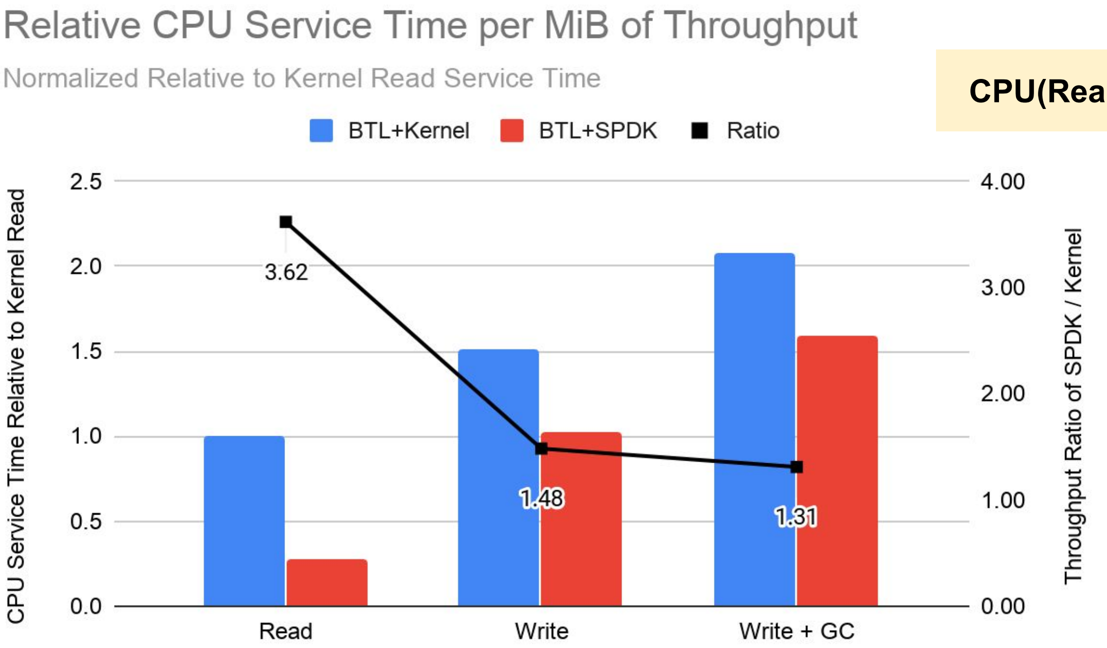
Similar Effect on Latency Improvements



$T_2 >> T_1$

$dL/T_2 << dL/T_1$

The Greater the CPU Cost of Additional Functionality,
The Smaller the System Performance Difference



CPU(Read) < CPU(Write) < CPU(Write+GC)

Conclusion

- BTL’s port to SPDK was straightforward:
    - Add a new host-specific environment for SPDK
    - Fix bugs in the BTL core that went unnoticed until a host environment change

- SPDK
    - CPU efficiency enables more functionality per core
    - Performance metrics observed at the SPDK/Kernel level do not translate to the same relative performance at the system

Potential Future Work

- SPDK
  - Zoned bdev in SPDK
  - ZNS NVMe driver (async support)
  - Further performance characterization:
    - Possible advantages of SPDK around control/determinism for user space implementations compared to going through kernel (context switching)
    - kernel vs userspace cost of memory usage

- io_uring
  - Support for ZNS
  - Could reduce impact of kernel/spdk performance

Radian®

MEMORY SYSTEMS