软硬件存储栈
及针对高性能设备的优化

张佳辰　2019. 9. 29

jczhang@nbjl.nankai.edu.cn

Parallel and Distributed Software Technology Lab

Nankai - Baidu Joint Laboratory

目录

- NVMe 硬件存储栈
- 软件存储栈
- 高性能软件存储栈优化
- 存储栈发展趋势

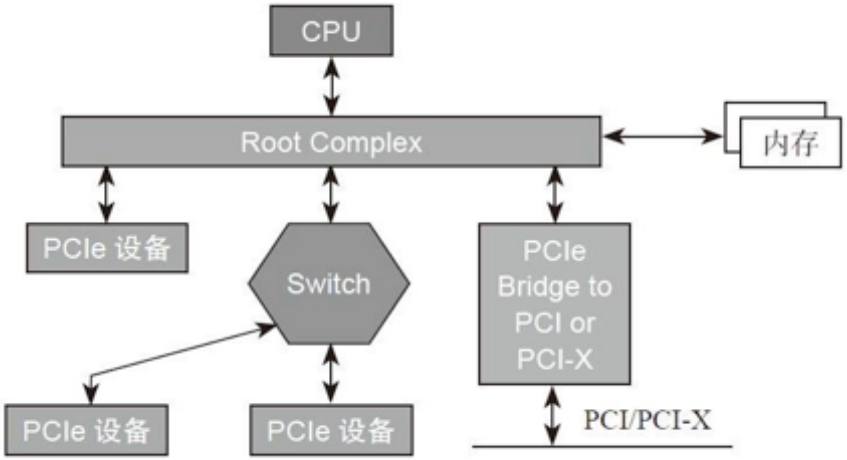
目录

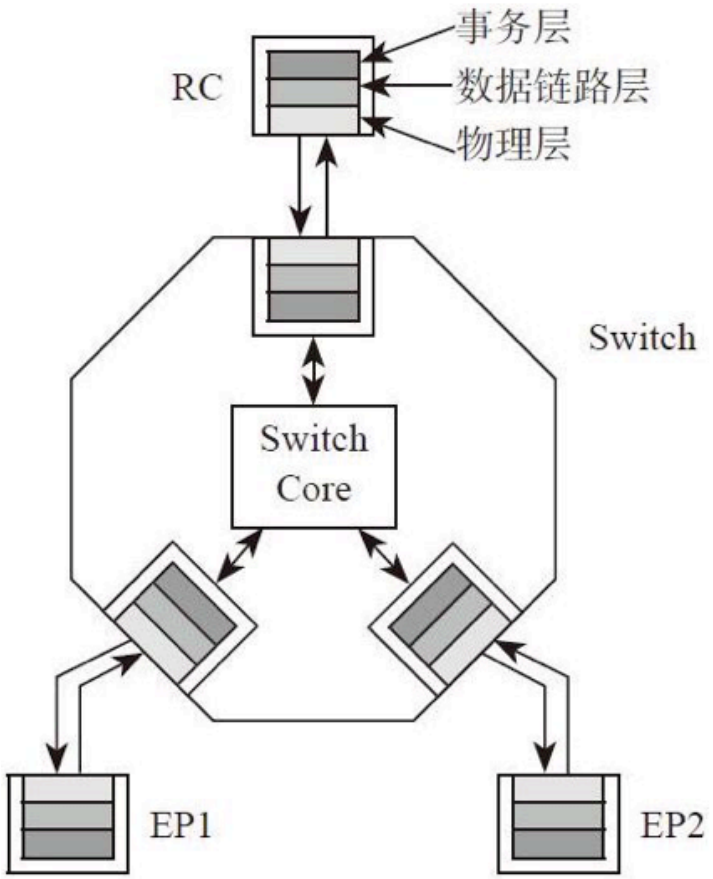
- NVMe 硬件存储栈
  - PCIe 传输
  - NVMe 协议
- 软件存储栈
- 高性能软件存储栈优化
- 存储栈发展趋势

NVMe 硬件存储栈

图6-3　NVMe处于协议栈的最高层

硬件存储栈简介 – PCIe层





硬件存储栈简介 – PCIe层

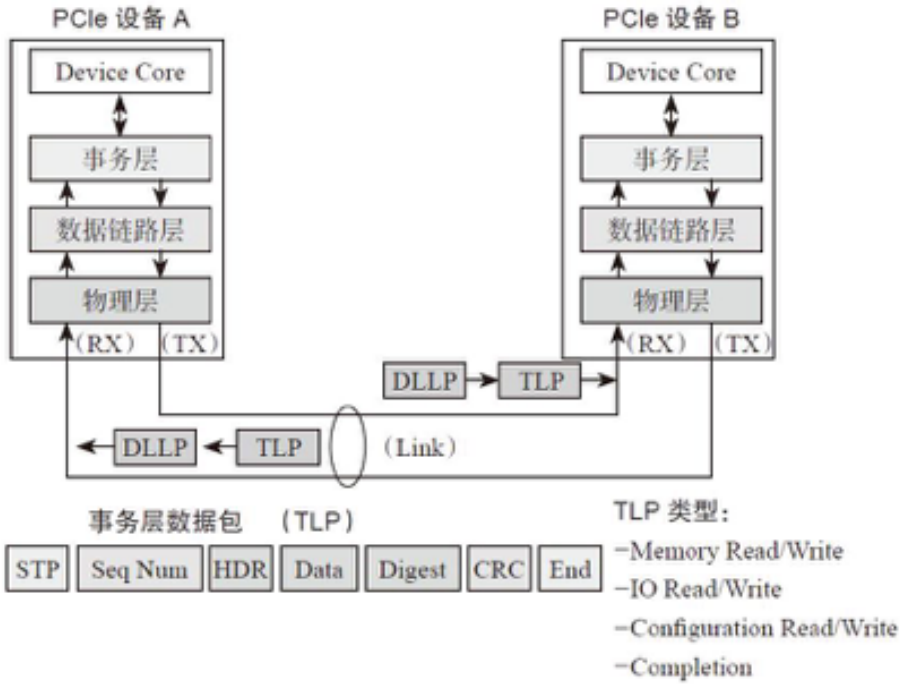
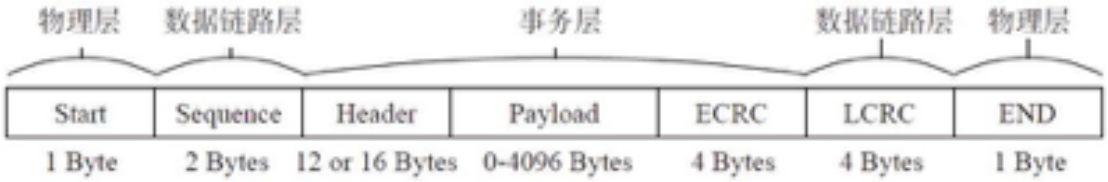
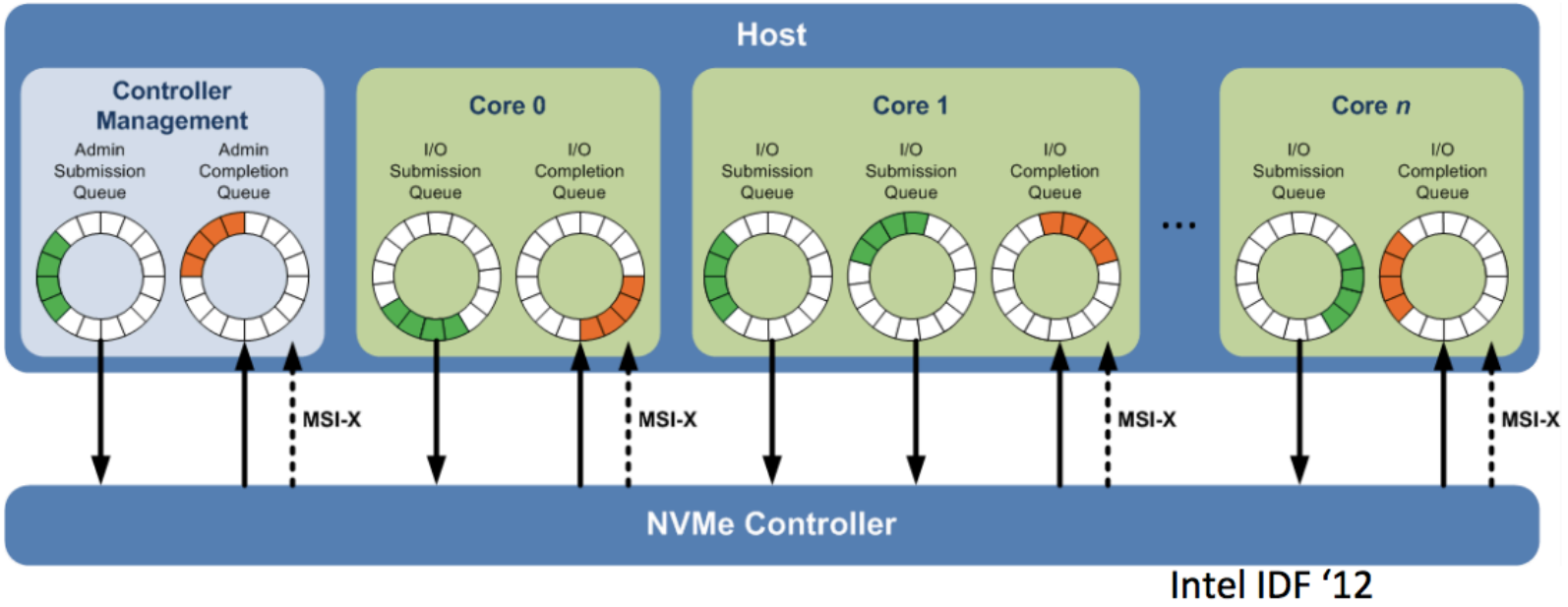


图6-30 PCIe两设备通信示意图



数据传输格式

硬件存储栈简介 – NVMe协议



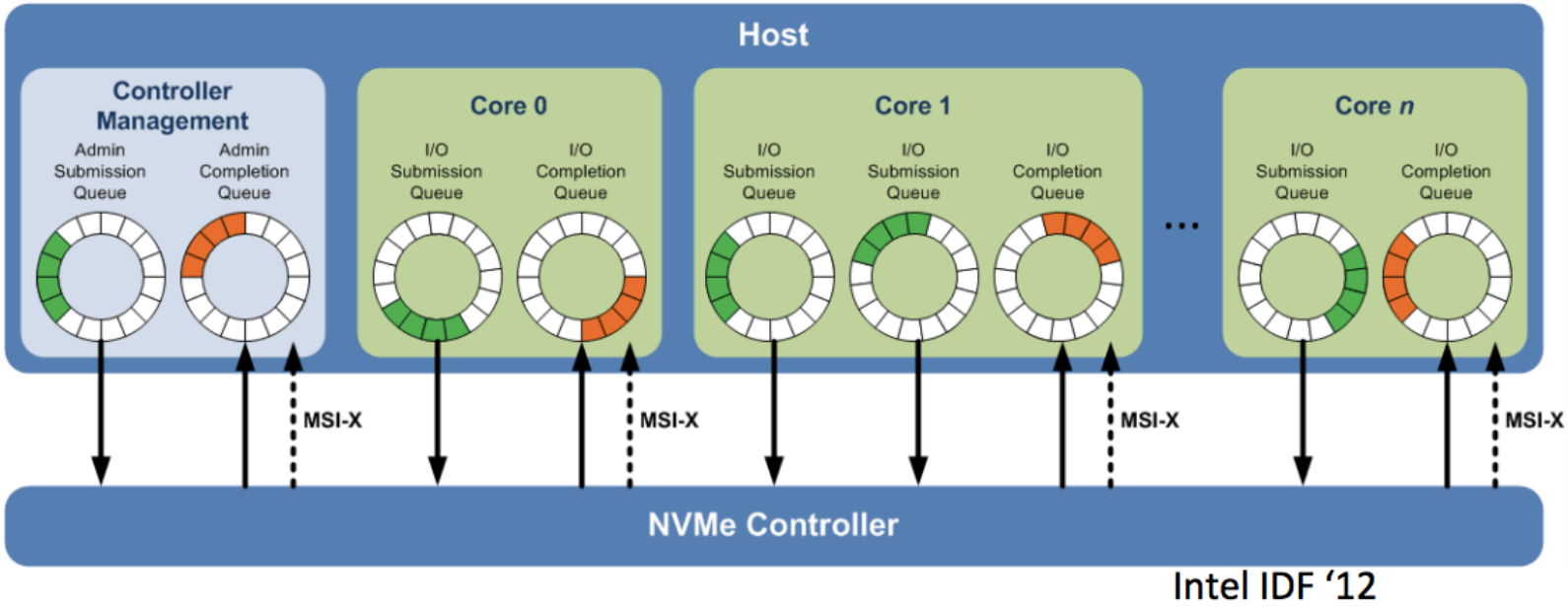
Intel IDF '12

- Admin SQ/CQ 每个系统只有一对。
- I/O SQ/CQ可以最多有65535对：

  每个物理核最多1个CQ；

  但每个核可以有多个SQ，因此可以实现线程粒度的不同优先级。
- SSD中还有称为Doorbell的寄存器，负责记录SQ/CQ的完成情况。

硬件存储栈简介 – NVMe协议



Intel IDF '12

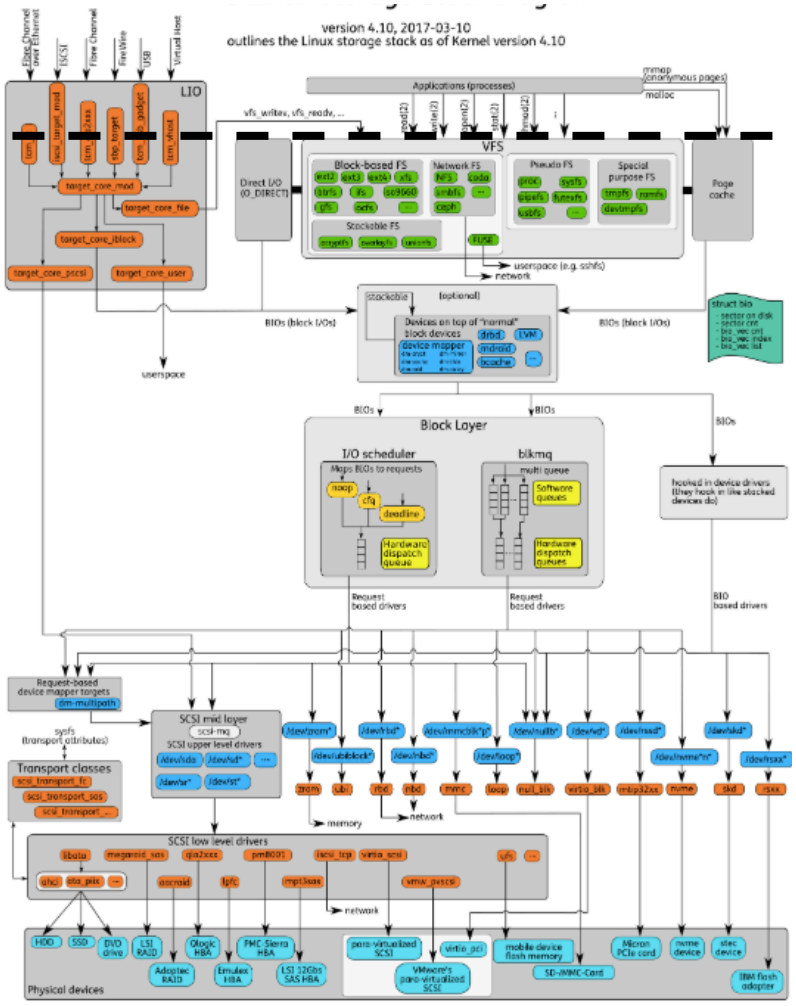
1. 主机写命令到SQ； (Host写SQ)
2. 主机写SQ的DB，通知SSD取指； (Host通知SSD)
3. SSD到SQ中取指执行，并写执行结果到CQ； (SSD执行并写CQ)
4. 然后SSD发(MSI-X)中断通知主机指令完成； (SSD通知Host)
5. 收到中断，主机处理CQ，查看指令完成状态； (Host处理完成)
6. 通过DB回复SSD：指令执行已完！ (Host通知SSD完成)

摘自《深入浅出SSD：固态存储核心技术、原理与实战》

目录

- NVMe 硬件存储栈
- 软件存储栈
  - Linux 内核存储I/O栈
  - 用户态存储栈
  - Persistent Memory 存储栈
- 高性能软件存储栈优化
- 存储栈发展趋势

Linux 内核存储栈 [5]



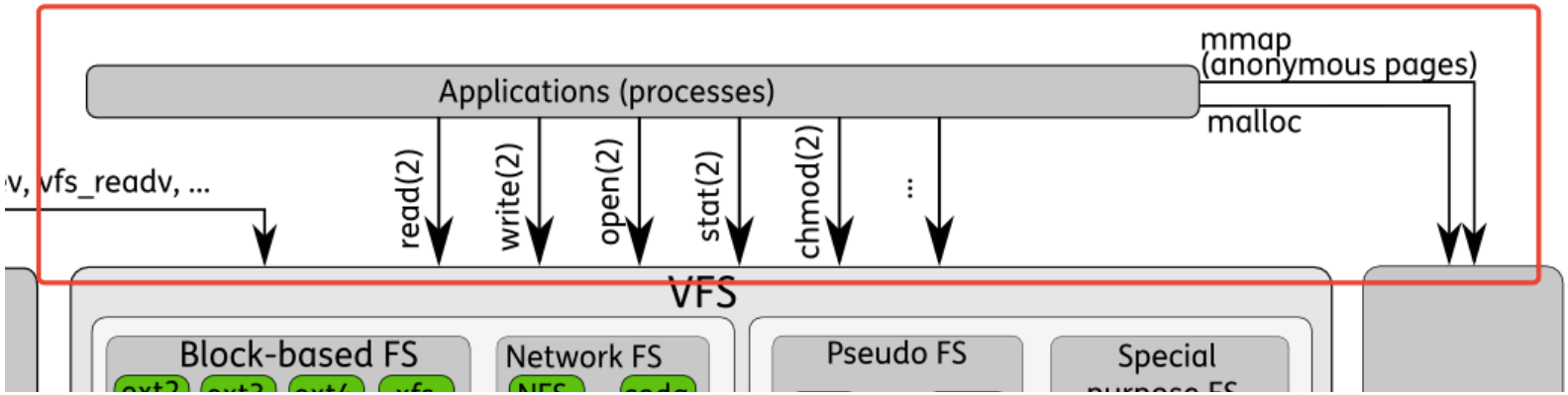
应用层 (用户态)

文件系统层 (内核态)

块IO层

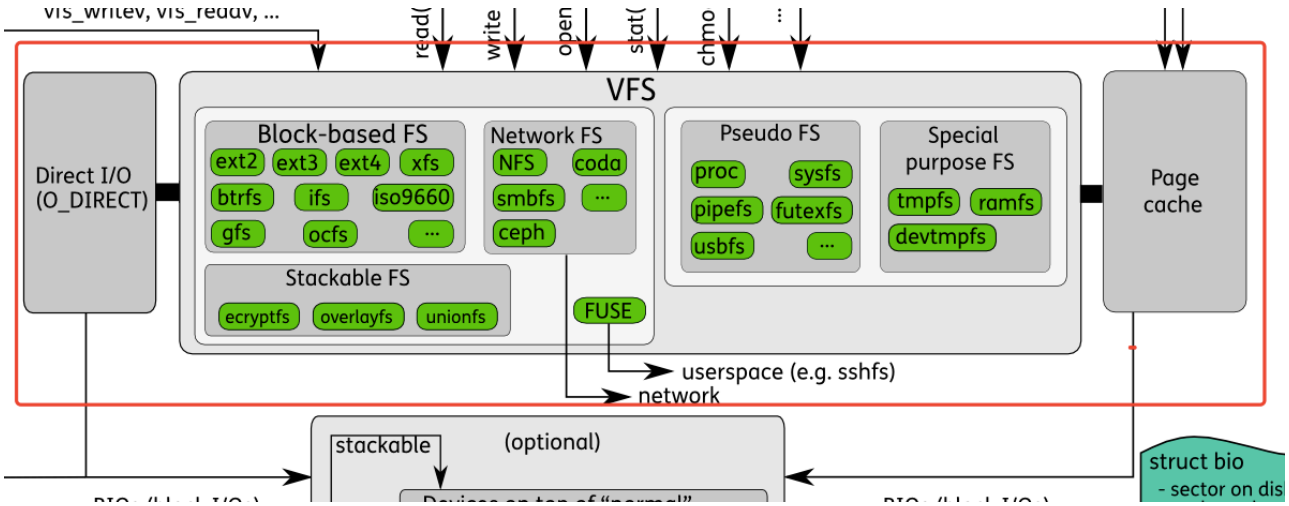
设备驱动层

Linux 内核存储栈 -- 应用层



最不固定(灵活)，但也是存储栈重要一环，比如数据库存储引擎一般实现于应用层，最终通过系统调用进入文件系统层。

Linux 内核存储栈 --文件系统层



- VFS为不同文件系统的实现提供统一的接口，调用对应的文件系统实现。
- 主要分两个路径Buffered I/O (使用Page Cache)和Direct I/O (open文件时加O_DIRECT标志) 。

文件系统层 -- 作用和功能

文件和文件夹(目录)形式的管理是最为广泛使用的数据存储抽象。文件系统负责管理文件数据的存取。

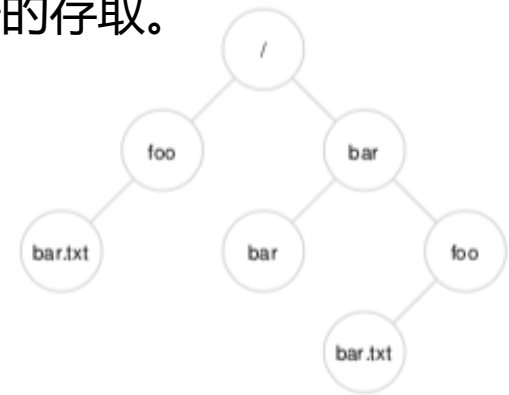


Figure 39.1: **An Example Directory Tree**

一些文件系统的常用接口：

创建删除文件：	creat, unlink
打开文件：	open
读写文件：	read, write
映射文件：	mmap
与磁盘同步(刷盘)：	fsync, msync
修改权限：	chmod
获取文件信息：	stat

文件系统层 -- page cache

Linux 系统默认的buffered IO读写会“吃掉”暂时空闲的内存空间用于缓存文件数据，这部分缓存称为 page cache。

Page cache 对提升磁盘I/O性能，充分利用内存资源。

但有些应用在用户态实现自己的文件缓存，因此可以用direct IO来绕过page cache。

文件系统层 -- page cache

Buffered I/O (Page Cache): 内核以4K页为单位进行IO请求和文件数据缓存。

Direct I/O (O_DIRECT): 不使用page cache，但偏移/地址/大小等要和扇区大小对齐。

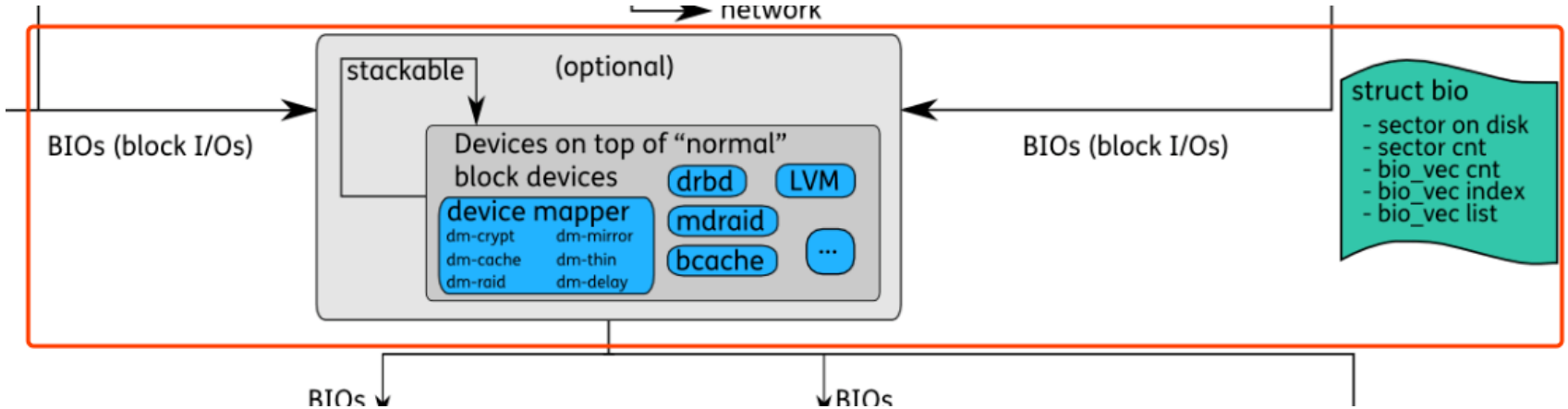
```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

#define FILENAME "test_file"

int main()
{
    int fd = open(FILENAME, O_CREAT|O_RDWR, 0755);
    int size = 123;
    int offset = 456;
    void *buf = malloc(size);
    int n = pwrite(fd, buf, size, offset);
    if (n < 0)
        return -1;
    return 0;
}
```

Buffered I/O 写示例

```c
// 需要加入以下宏定义来支持direct_IO
#define _GNU_SOURCE

#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

#define FILENAME "test_file"
#define SECTOR_SIZE 512

int main()
{
    int fd = open(FILENAME, O_CREAT|O_RDWR|O_DIRECT, 0755);
    int n_sector = 5;
    int off_sector = 3;
    void *buf;
    // 申请的buffer的起始地址需要对齐扇区
    posix_memalign(&buf, SECTOR_SIZE, SECTOR_SIZE * n_sector);
    // 写文件时，写数据的大小、偏移都需要对齐扇区。
    int n = pwrite(fd, buf, SECTOR_SIZE * n_sector, SECTOR_SIZE * off_sector);
    if (n < 0)
        return -1;
    return 0;
}
```

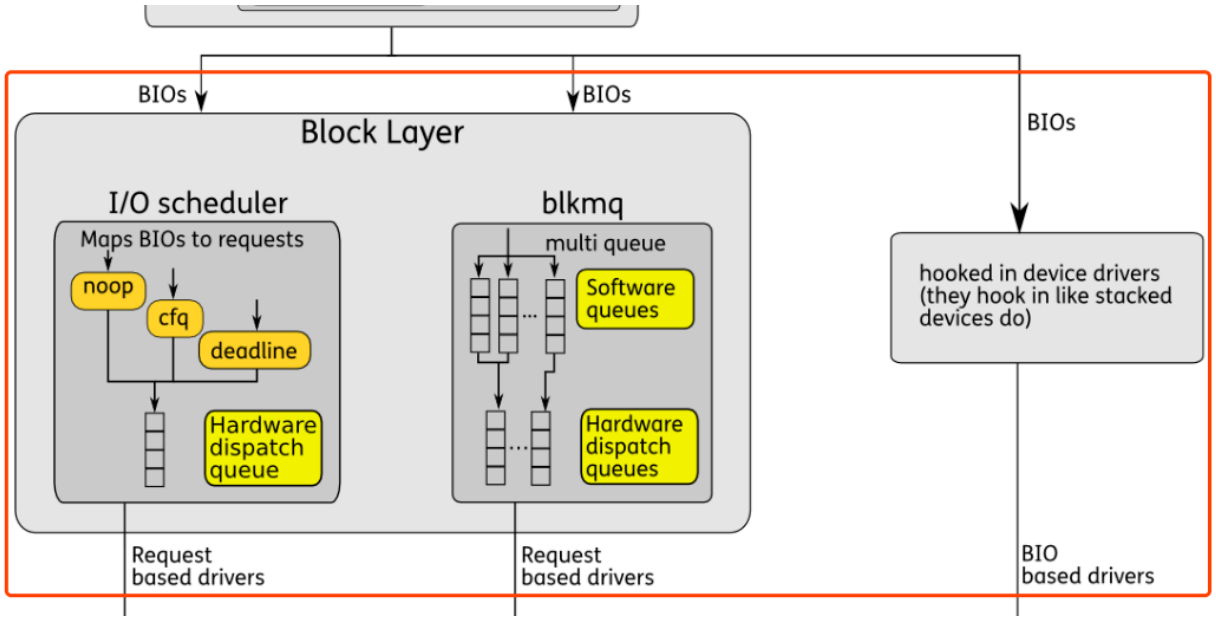
Direct I/O 写示例

Linux 内核存储栈 -- 块层 Stackable Layers



Stackable block layer (bio layer) 管理逻辑块到物理资源的映射，实现一些存储虚拟化的特性。支持md (RAID), LVM卷, 块层cache(Bcache)等功能。数据重复删除、远程备份等功能也适合在这一层实现。

Linux 内核存储栈 -- 块层 I/O Scheduler



块调度层主要包括single-queue和multi-queue两种请求队列，每种队列又对应自己的调度器。

块层 – I/O Scheduler

I/O调度器：

- noop (no operation): 简单的FIFO，直接插入到调度队列的末尾。
- deadline: 有4个请求队列。其中有一对分别是放读写请求队列(排序队列)；另一对是按最后期限排列的读写请求队列(deadline队列)。
- CFQ(complete fairness queue): 多个请求队列，用hash将一个进程号的请求发到一个请求队列（因此，一个进程常发到一个队列）。
- Anticipatory("预期", 默认算法[2]): 类似deadline，但加入了一些启发式准则。

块层 – I/O Scheduler

I/O调度器：

- noop (no operation): 简单的FIFO，直接插入到调度队列的末尾。
- deadline: 有4个请求队列。其中有一对分别是放读写请求队列(排序队列)；另一对是按最后期限排列的读写请求队列(deadline队列)。
- CFQ(complete fairness queue): 多个请求队列，用hash将一个进程号的请求发到一个请求队列（因此，一个进程常发到一个队列）。
- Anticipatory("预期", 默认算法[2]): 类似deadline，但加入了一些启发式准则。

Multi-queue 调度器：

- none: 类似于single-queue的noop调度器。
- mq-deadline
- bfq
- kyber

块层 – I/O Scheduler

为什么会有multi-queue [14]

(1) 利用了多核的特性提升IO调度层的可扩展性

(2) 可以更好地利用NVMe等高性能存储设备的并性能

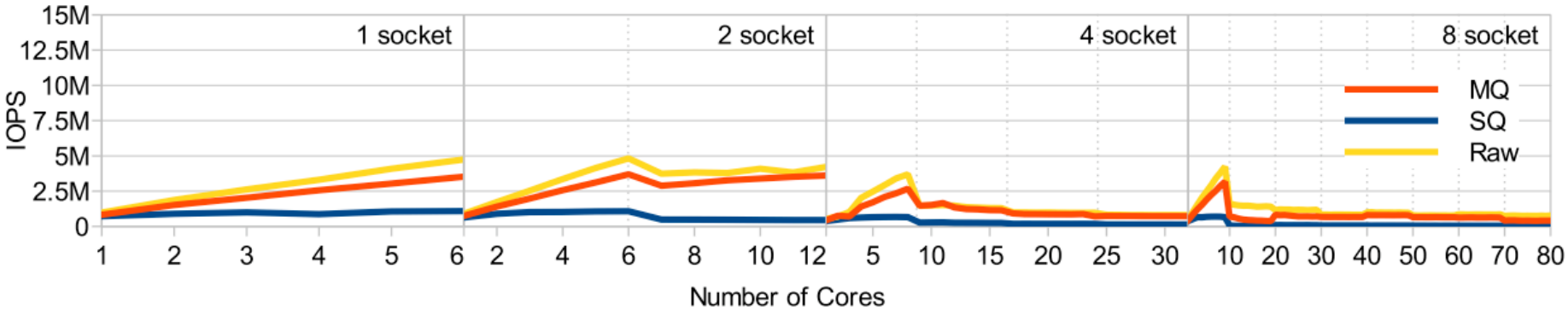


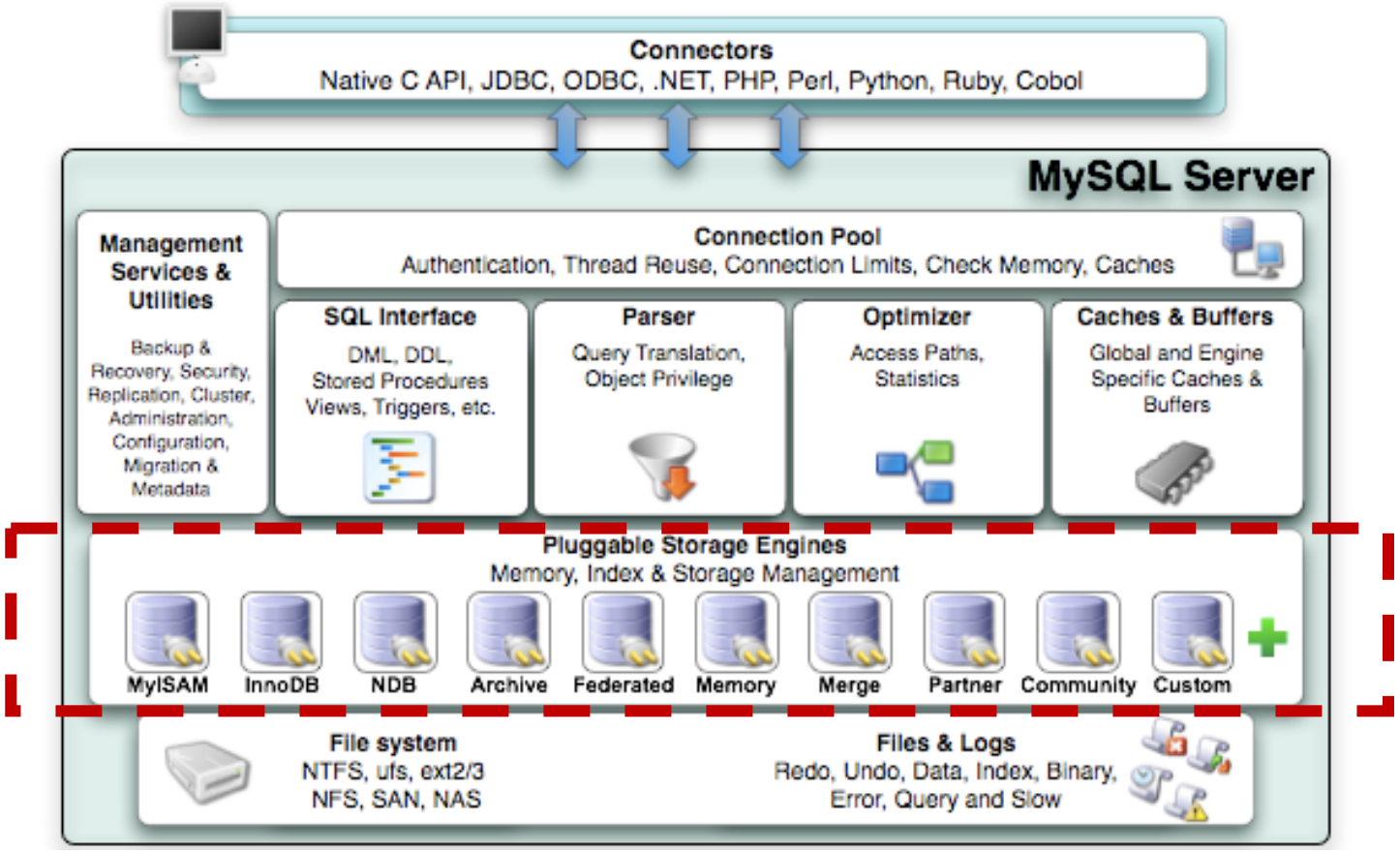
Figure 6: IOPS for single/multi-queue and raw on the 1, 2, 4 and 8-nodes systems.

用户态(应用层)存储栈

数据库、大数据存储引擎、虚拟机等场景都有自己的文件格式，相当于用户态也有自己的存储栈：

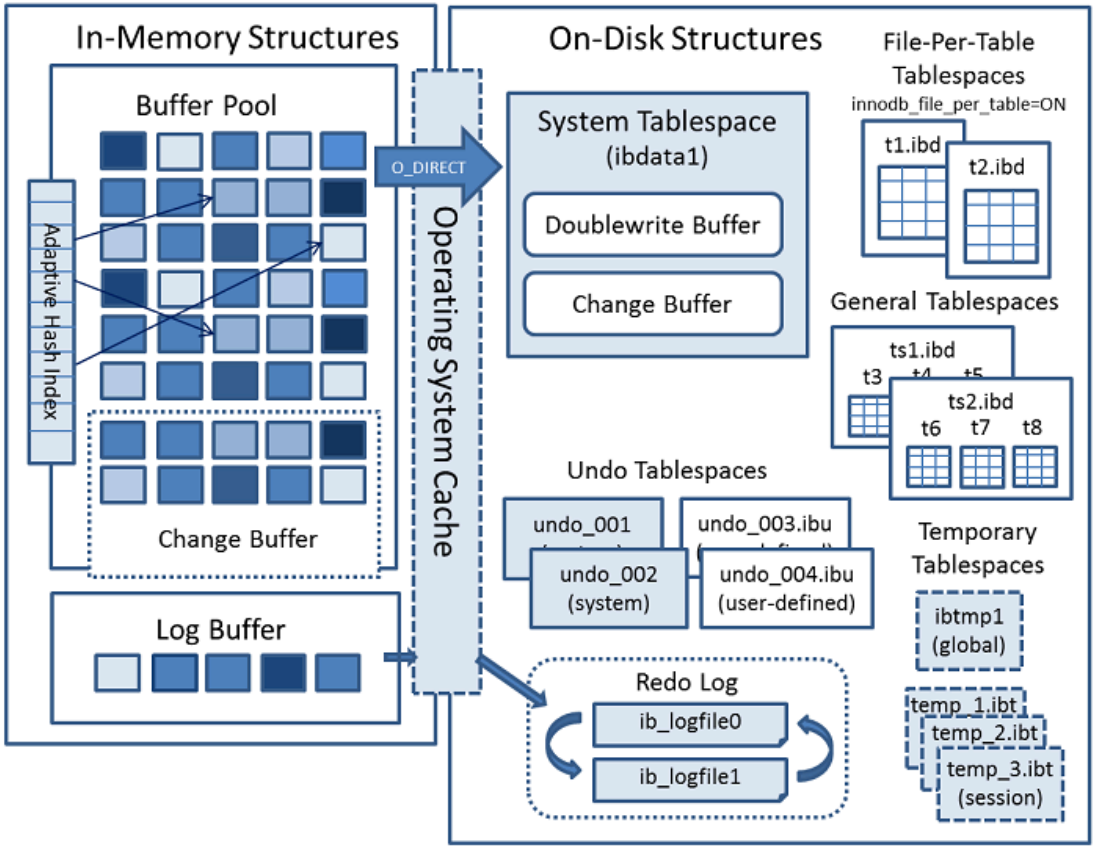
- MySQL 数据库支持多种存储引擎。
- QEMU 虚拟机支持多种虚拟磁盘格式。
- Docker 支持多种“存储driver”，用于管理容器镜像的分层。

用户态(应用层)存储栈 -- MySQL



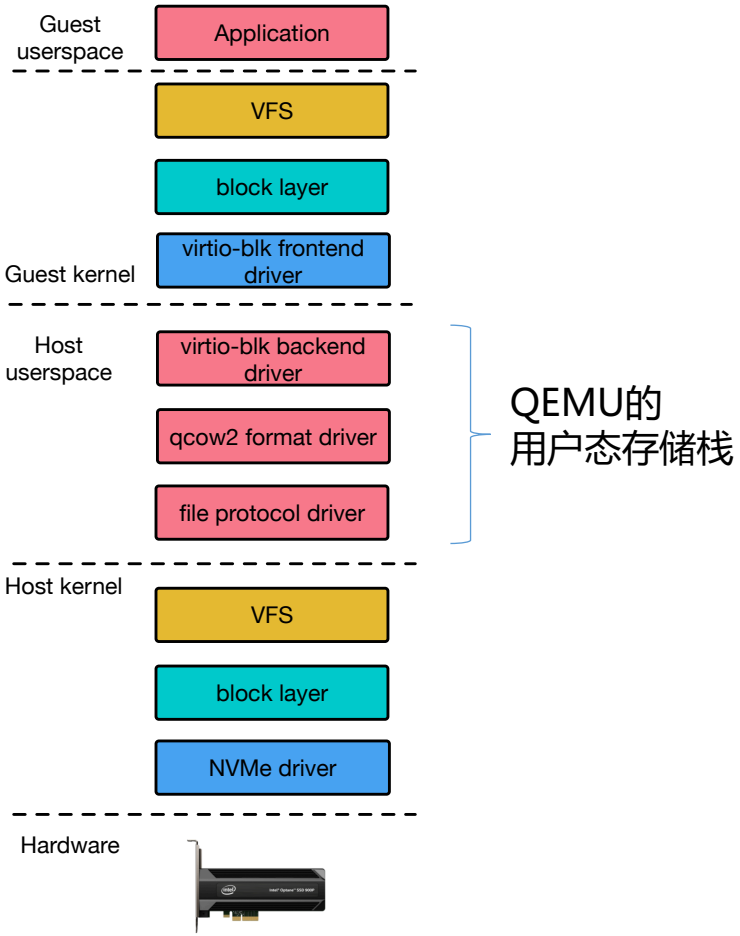
MySQL 支持多种存储引擎（InnoDB, MyISAM, Memory, CSV, Merge, Archive, Federated, Blackhole, Example …）

用户态(应用层)存储栈 -- MySQL



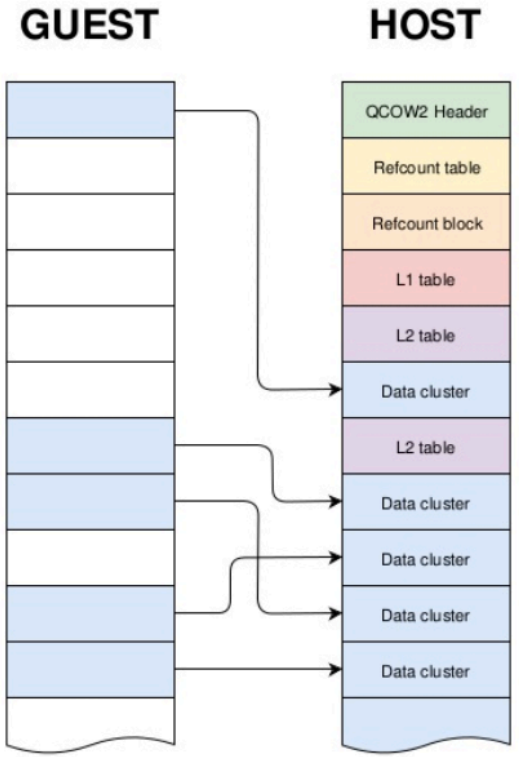
MySQL InnoDB 存储格式 [9]

用户态(应用层)存储栈 -- QEMU



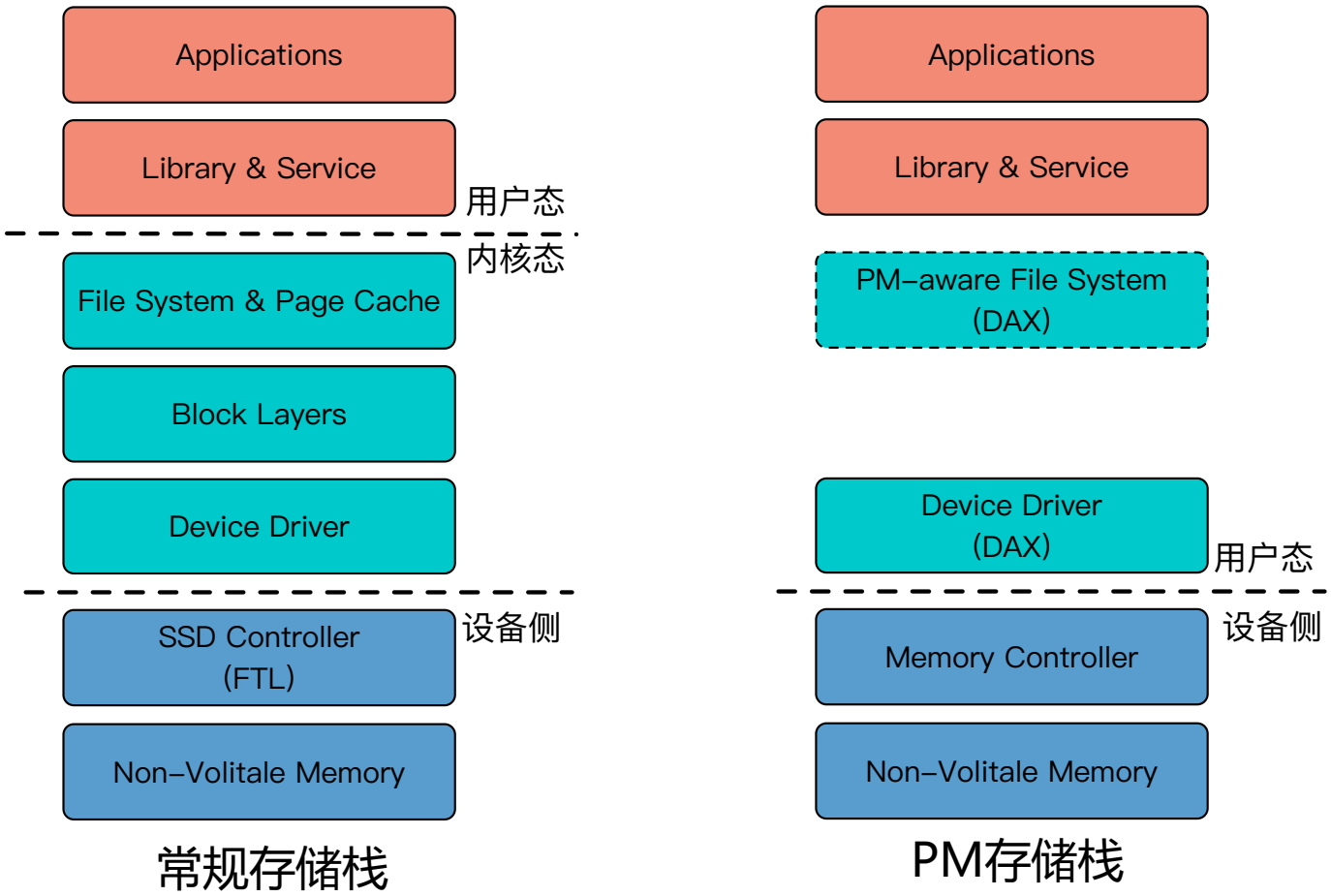
QEMU的
用户态存储栈

QEMU 虚拟机存储栈



QEMU的qcow2 镜像格式 [10]

Persistent Memory 存储栈



目录

- NVMe 硬件存储栈
- 软件存储栈
- 高性能软件存储栈优化
  - 存储性能指标 (延迟-吞吐曲线)
  - 延迟优化 (轮询)
  - 可扩展性优化 (多队列)
- 存储栈发展趋势

存储性能指标

- **延迟 (Latency)** 完成一次操作的时间。
  测试方法： 小块(4KB), 单线程
- **带宽 (Bandwidth)** 读/写速度。
  测试方法： 大块(64KB~8MB), 单/多线程
- **IOPS (I/O operations per second)** 每秒的操作数。
  测试方法：小块(4KB), 多线程

	DRAM 内存	持久型内存	HDD/SSD 存储
			
延迟	**~100 ns**	**~250 ns**	**10us ~ 10ms**
单条读带宽	~20 GB/s	~ 7 GB/s	120 MB ~ 2.5 GB/s
单条写带宽	~13.5 GB/s	~ 2.3 GB/s	80 MB ~ 2.2 GB.s

存储性能指标 – 延迟-吞吐曲线

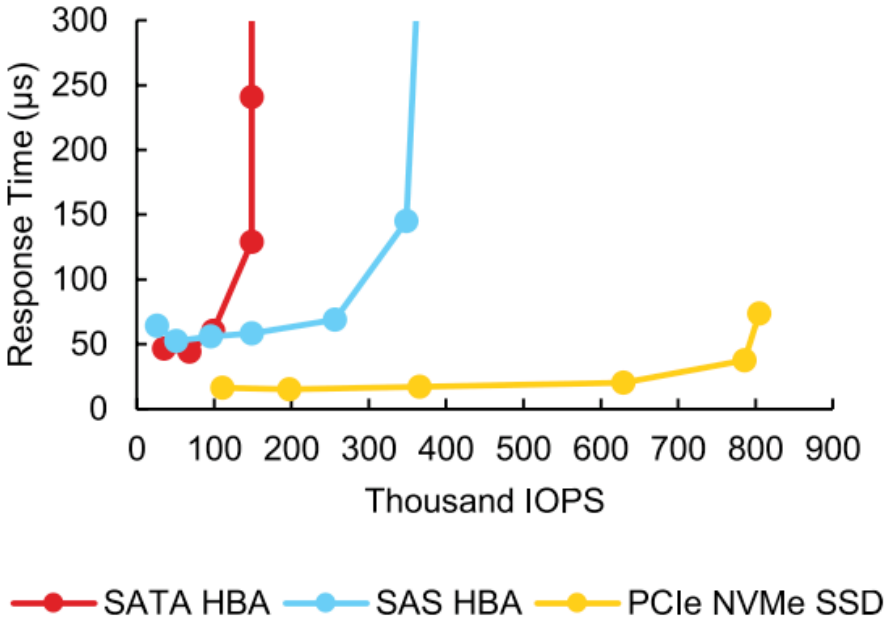


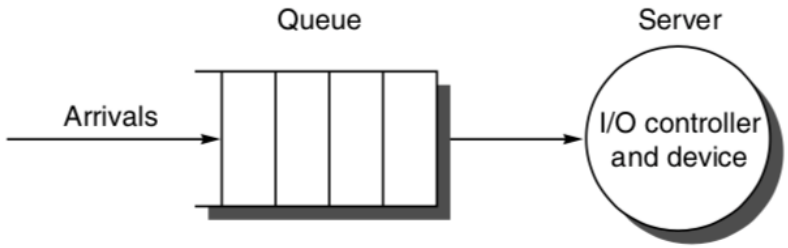
Fig. 3. ***99.99th percentile response times (excluding media) plotted against IOPS for 4-kB reads for three SSD interfaces.***

三种设备的延迟-吞吐曲线 [8]， 可以发现在接近吞吐极限时延迟也急剧增加。

所以对于延迟敏感的上层的在线应用，一定要考虑IO负载，尽量不要打满。

存储性能指标 – 延迟-吞吐曲线

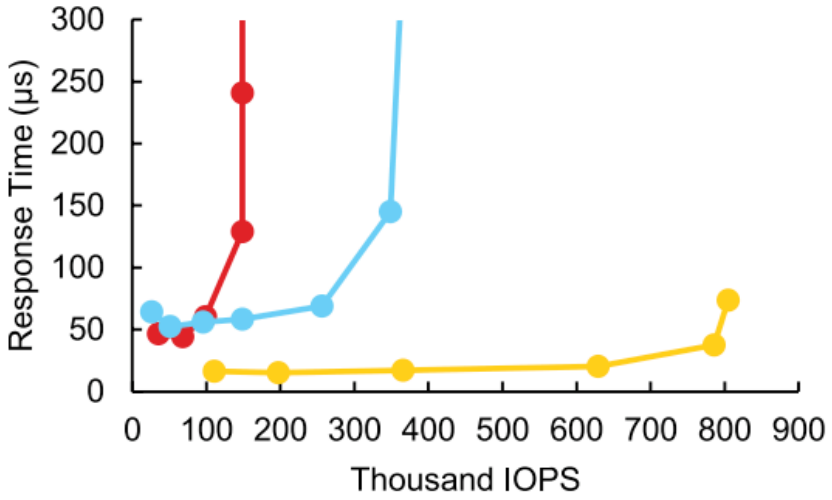
一点排队论[12]



$$\text{Time}_{\text{queue}} = \text{Time}_{\text{server}} \times \frac{\text{Server utilization}}{(1 - \text{Server utilization})}$$

其中Server utilization 数值为到达速率除以服务速率，可以理解为IOPS除以最大IOPS值。$\text{Time}_{\text{server}}$ 可以理解为latency的最小值。

$$\text{Latency} = \text{Time}_{\text{server}} + \text{Time}_{\text{queue}}$$
$$= \text{Time}_{\text{server}} \times [1 / (1 - \text{Server Utilization})]$$



存储性能指标 – 延迟-吞吐曲线

吞吐(IOPS) – 延迟(Latency)曲线

- 对Optane等超高性能SSD，单线程无法提高设备queue depth，需要用多个IO线程才能测到SSD的最大IOPS。

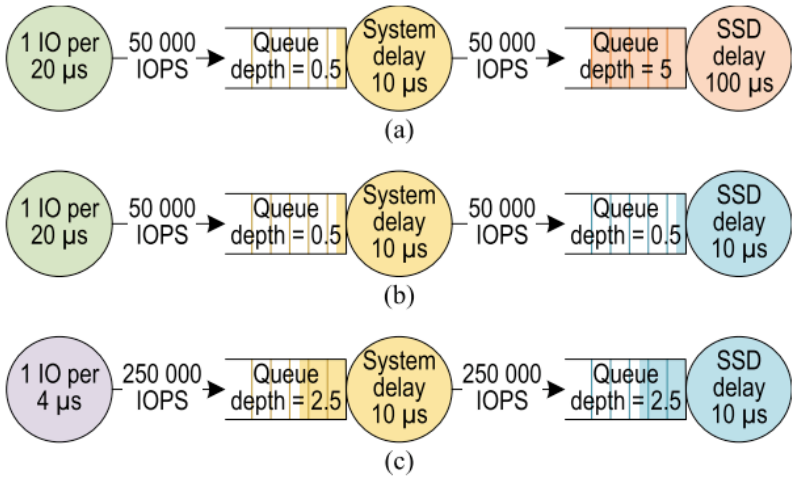
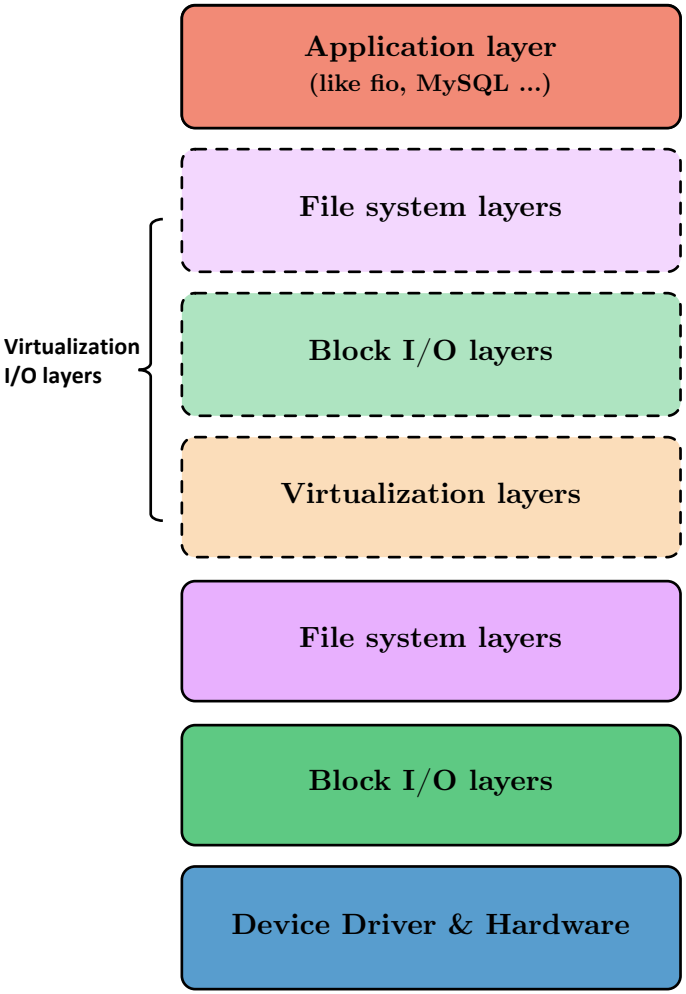


Fig. 6. ***A comparison of average queue depths for systems with (a) a higher delay SSD; (b) a lower delay SSD; and (c) a lower delay SSD under a higher request rate.***

存储栈的性能优化



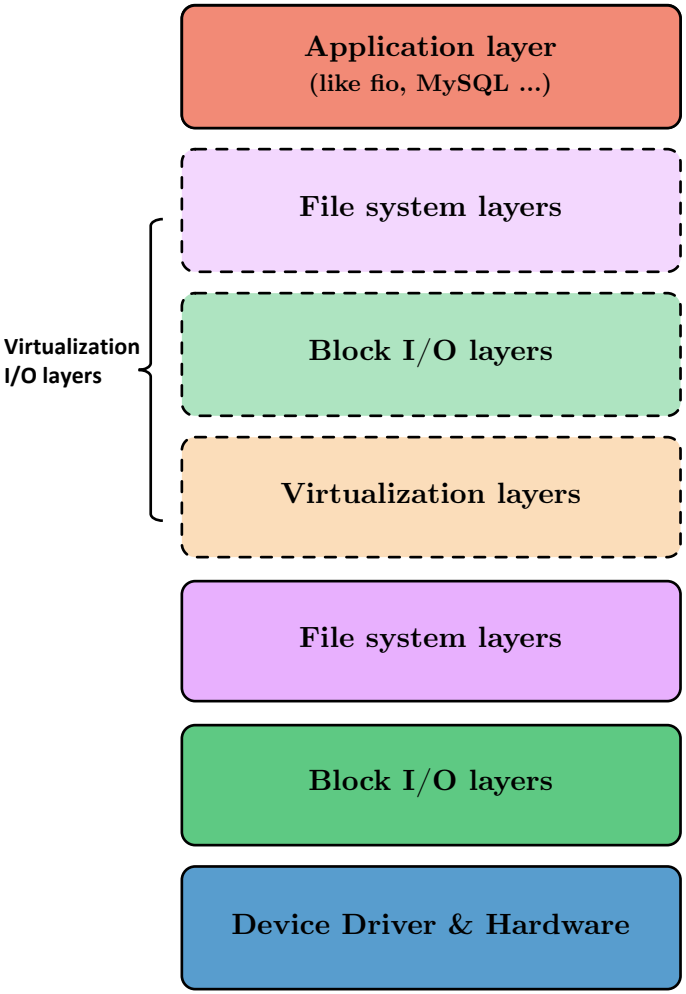
多队列技术 (multi-queue)

利用多核、多队列提升IOPS

轮询完成 (polling)

占用CPU进行轮询代替
中断或者消息通知

延迟优化 – 轮询



思路：

用轮询代替中断(或事件)，减小软件栈延迟，缺点是增加了CPU占用率。

轮询完成 (polling)

占用CPU进行轮询代替
中断或者消息通知

延迟优化 – 轮询

内核中的polling

(1) irq_poll 可以减少中断数。
原理是在第一次中断后开始处理所有可用的事件，当所有事件处理完后才开中断。
(2) 当blk_mq_poll在可以在块层polling同步I/O。

用户态的polling

(1) QEMU中轮询 vring 减少 virtqueue kick 造成的vm-exit次数

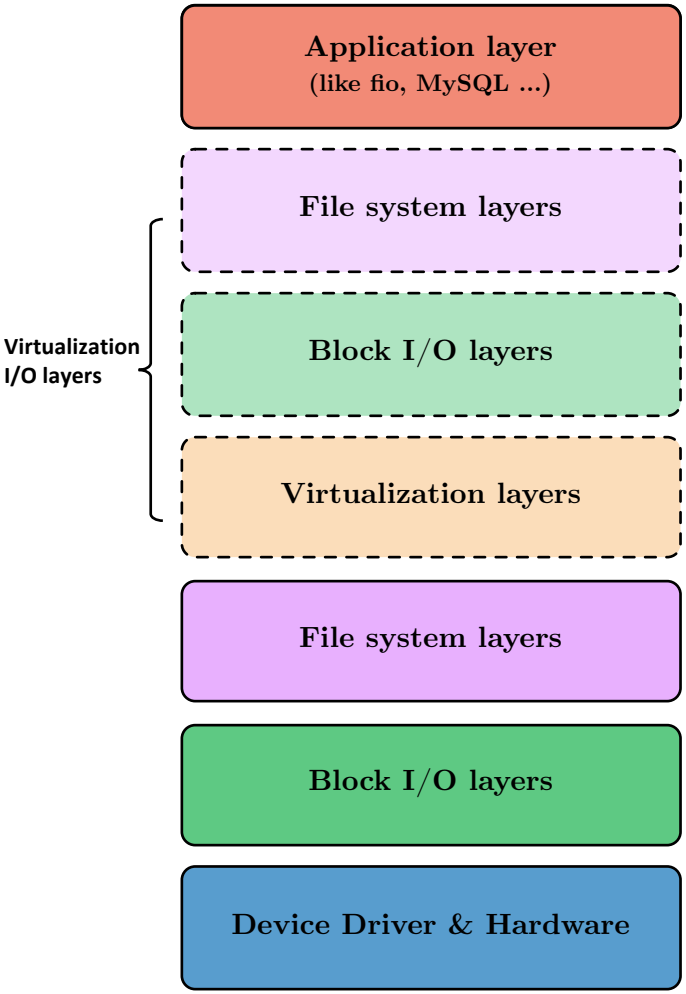
(2) 主动poll libaio的完成

(3) 内核提供最新的io_uring系列系统调用提供polling模式

(4) SPDK实现了基于polling完成状态的用户态NVMe驱动

用polling代替中断/事件通知，Optane SSD从延迟13us 降到到 7us，QEMU虚拟机存储栈的延迟可以降低10us左右的延迟[11] 。

可扩展性优化 – 多队列



多队列技术 (multi-queue)

利用多核、多队列提升IOPS

思路:

用多队列提升多核系统具有IO请求的扩展性，也利用了多通道存储设备的硬件并进性，提升IOPS。

可扩展性优化 -- 多队列

Linux 块层

- Linux块层的blk-mq模式是多队列的

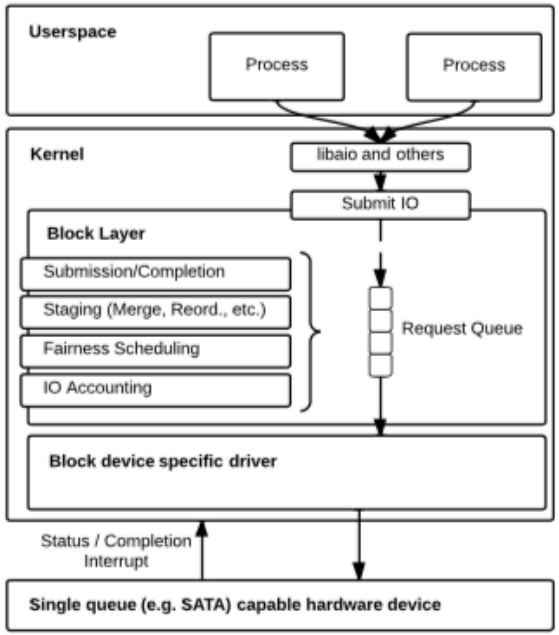


Figure 2: Current single queue Linux block layer

(a) 原块层

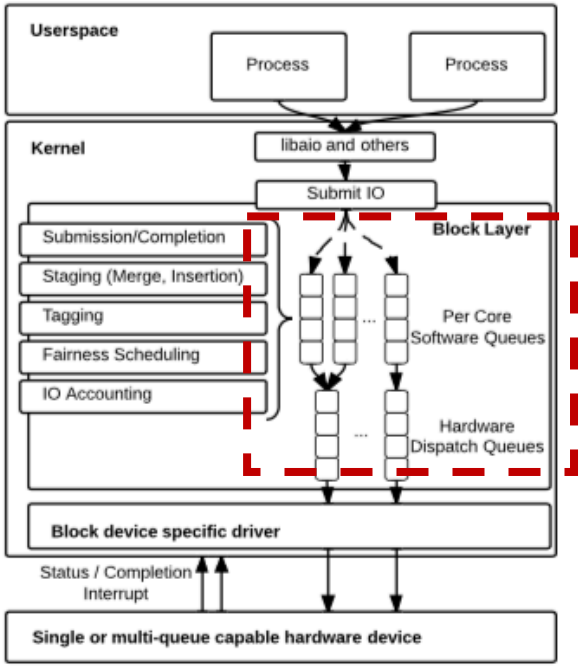


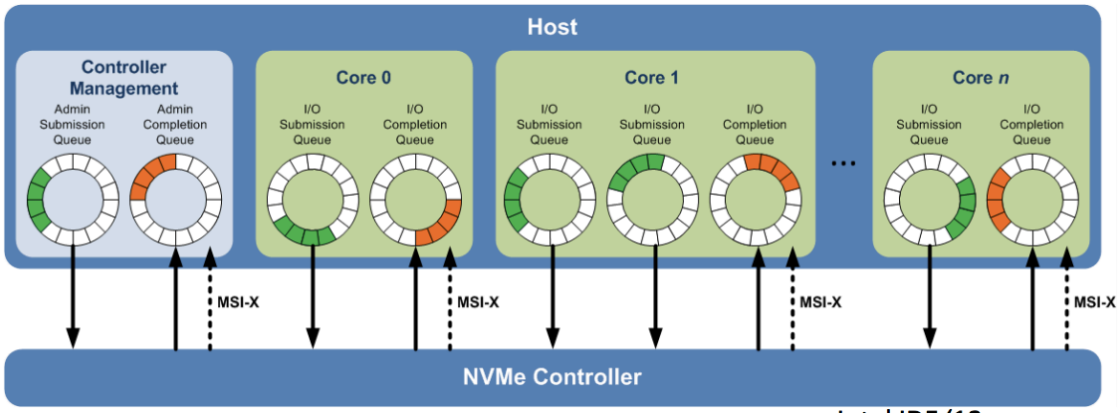
Figure 5: Proposed two level Linux block layer design.

(b) blk-mq 块层

可扩展性优化 -- 多队列

NVMe驱动层

- NVMe协议驱动是支持多队列的：



Intel IDF '12

1.	主机写命令到SQ；	(Host写SQ)
2.	主机写SQ的DB，通知SSD取指；	(Host通知SSD)
3.	SSD到SQ中取指执行，并写执行结果到CQ；	(SSD执行并写CQ)
4.	然后SSD发中断通知主机指令完成；	(SSD通知Host)
5.	收到中断，主机处理CQ，查看指令完成状态；	(Host处理完成)
6.	通过DB回复SSD：指令执行已完！	(Host通知SSD完成)

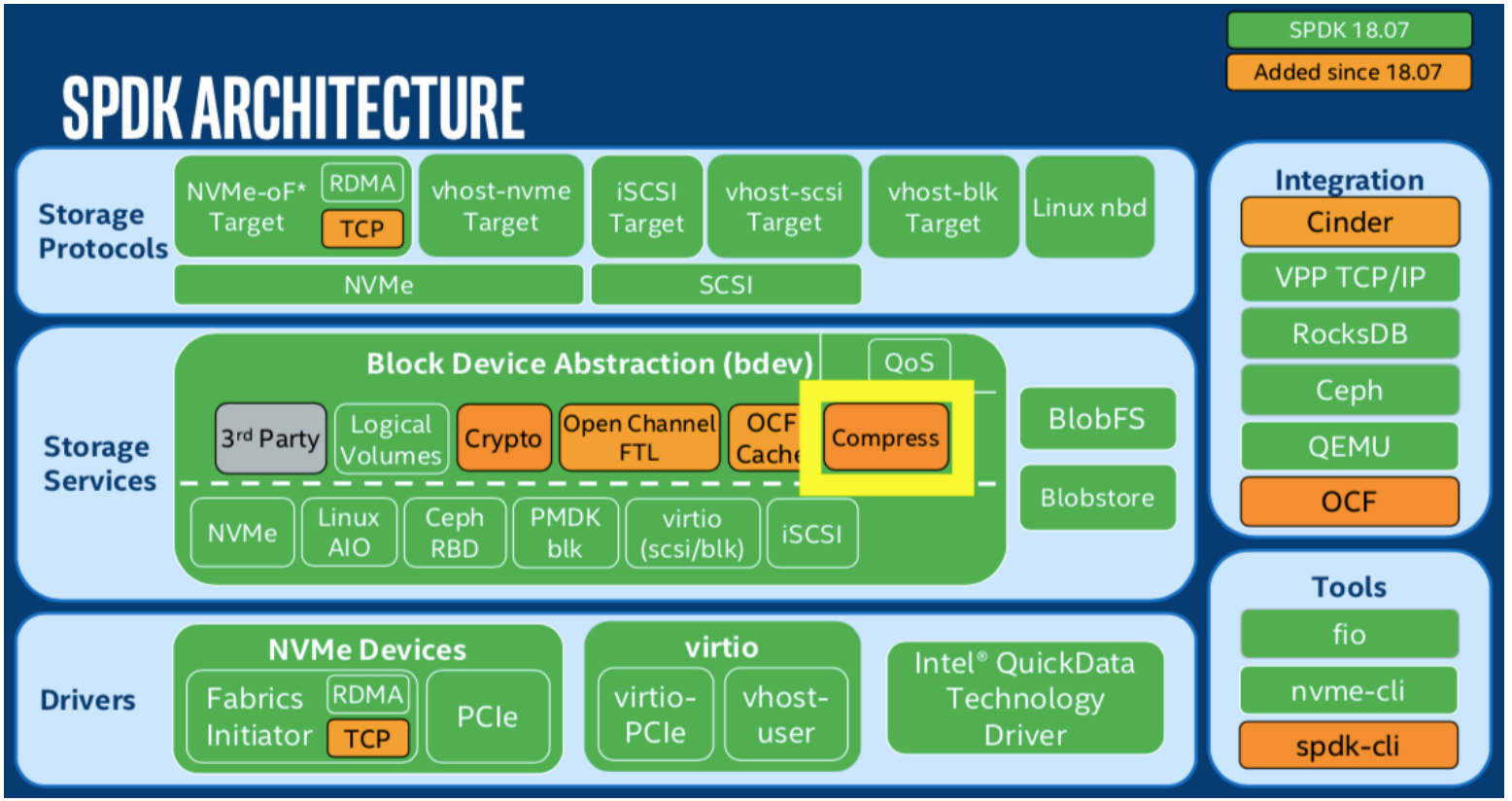
NVMe协议流程

目录

- NVMe 硬件存储栈
- 软件存储栈
- 高性能软件存储栈优化
- **存储栈发展趋势**
  - 用户态存储栈 (内核态 ->用户态)
  - 存储栈的offload (软件 -> 硬件)
  - 软件定义存储 (硬件 -> 软件)

存储栈发展趋势

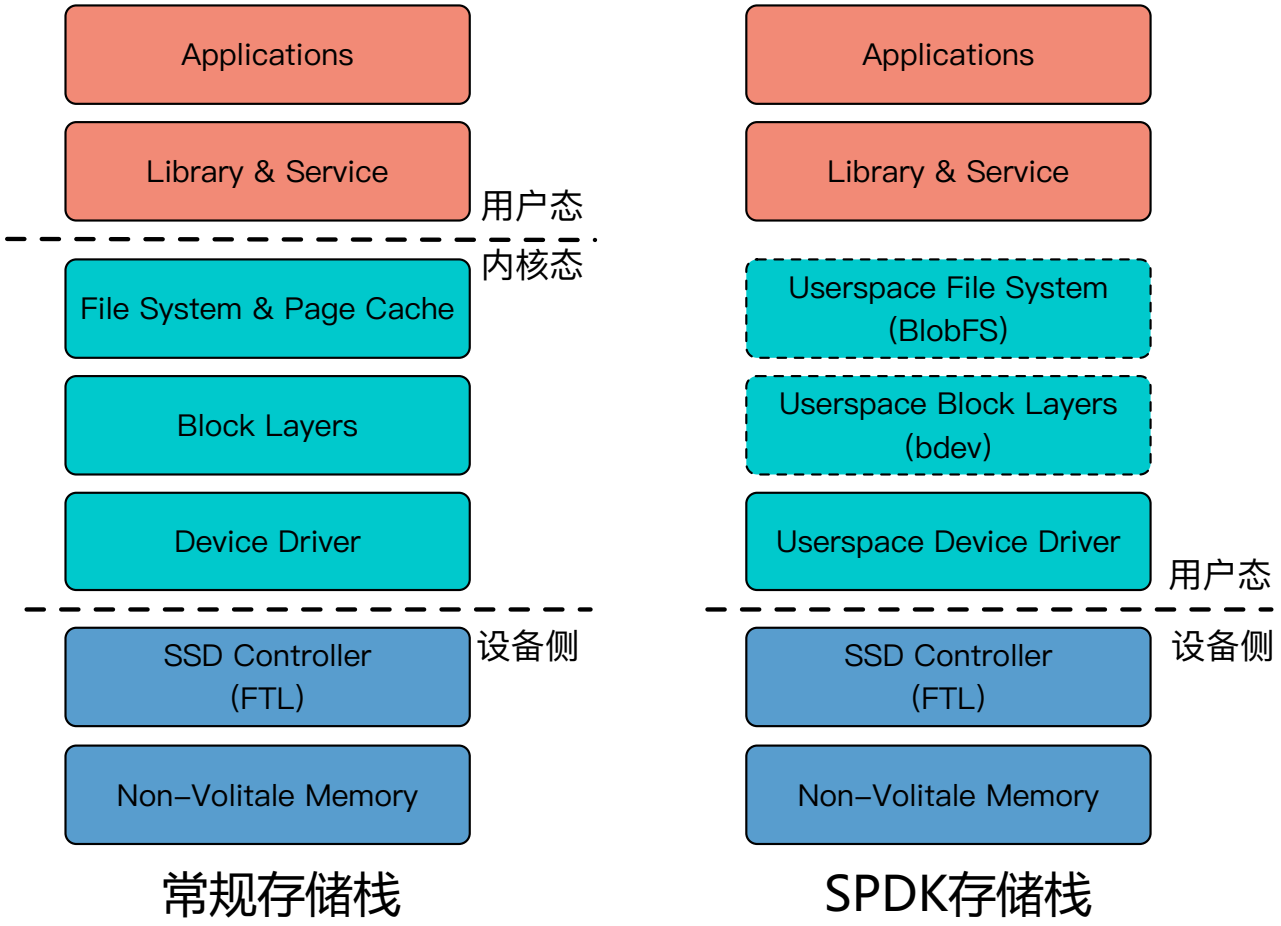
用户态存储栈 (内核态 ->用户态) -- SPDK为例



SPDK 架构图

存储栈发展趋势

用户态存储栈 (内核态 ->用户态) -- SPDK为例



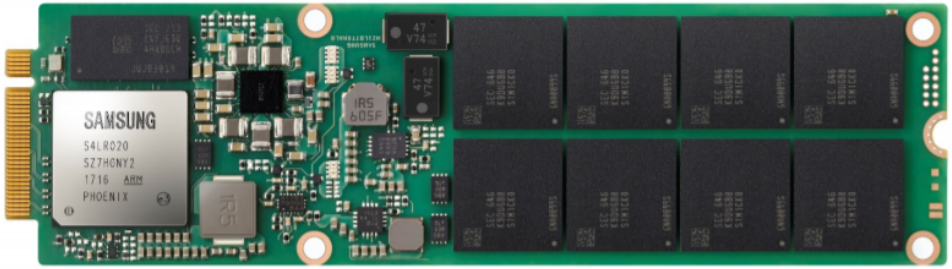
存储栈发展趋势

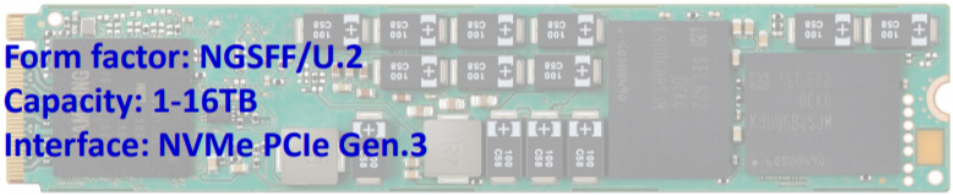
存储栈的offload (软件 -> 硬件) -- KVSSD为例

KV SSD相当于在普通SSD上强化了Flash Translate Layer的功能，这包括：

- 原本garbage collection在SSD和database软件中各占一层，KV SSD将其纳入FTL中为一层。
- 原本KV->LBA的转换运行在host，KV SSD将其纳入FTL，向上层提供KV接口。

NGSFF KV SSD

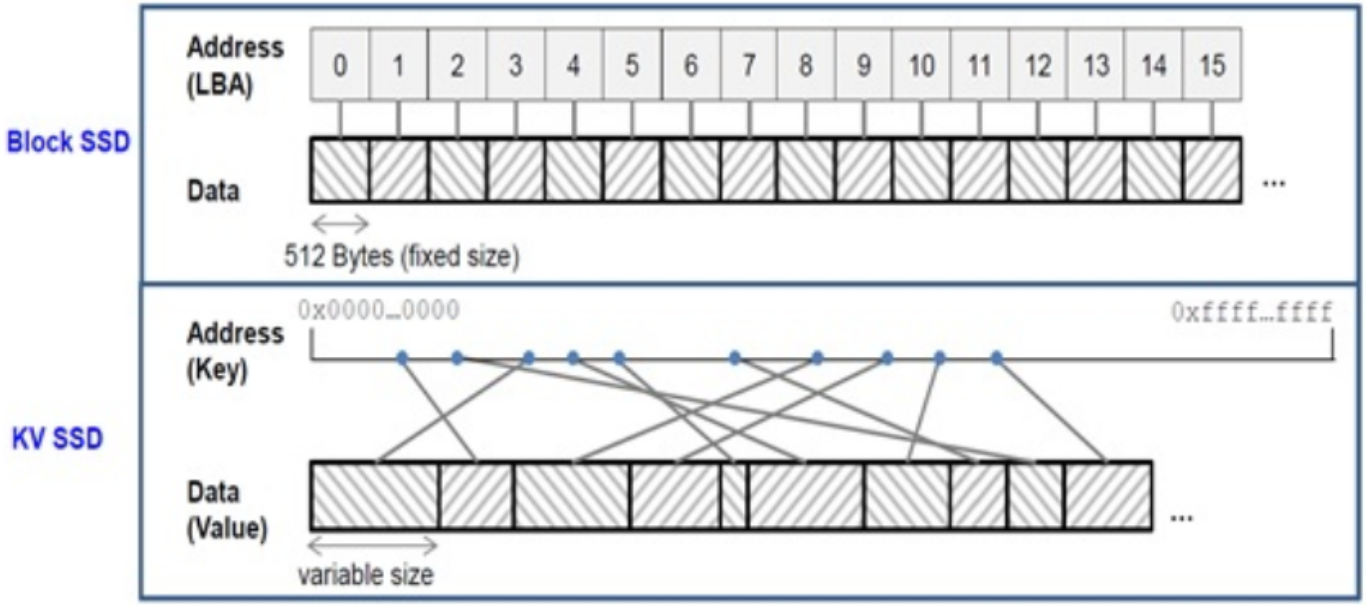




KV SSD的原理

KV SSD向上层提供的是“可变”的KV接口，与普通SSD的块存储接口相比：

- 支持可变大小的Keys，而非固定大小的Logical Block Addresses（通常为48或64bit）
- 支持可变大小的Values，而非固定大小的Blocks（通常为512bytes或4KB）

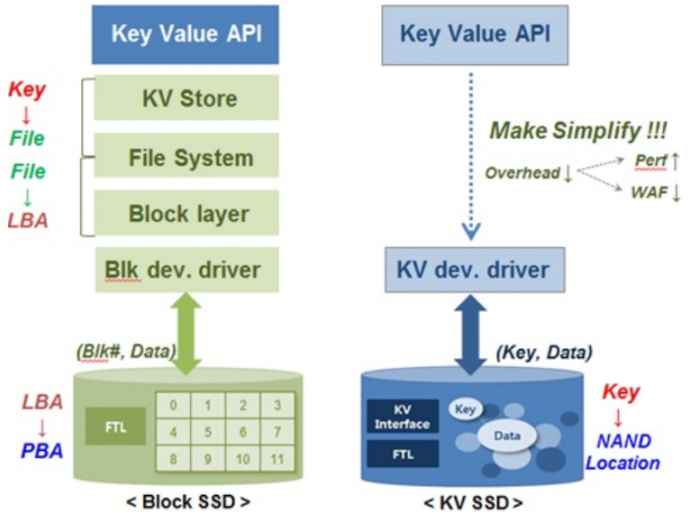


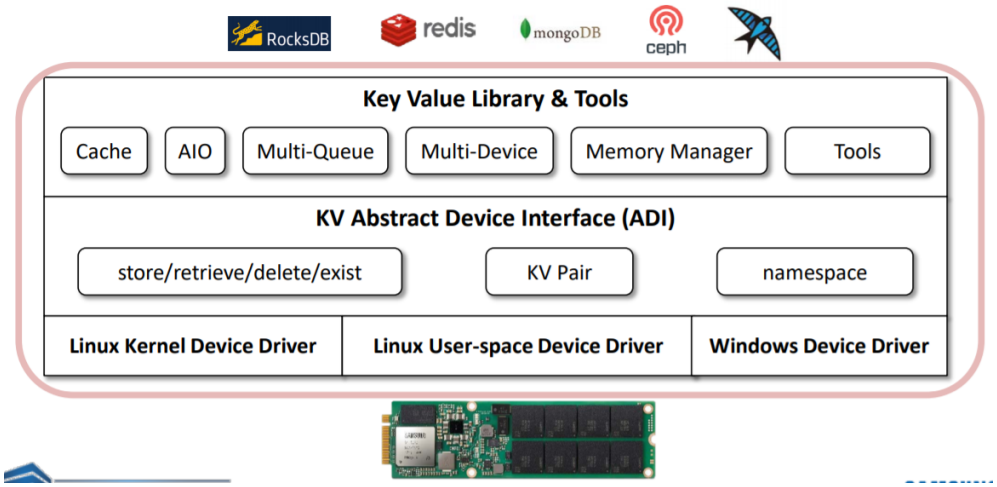
KV SSD的存储栈

KV SSD 相比于传统SSD存储栈

- 传统KV存储栈需要多个复杂软件层实现从KV数据到逻辑块的转换。此外，一些KV应用也定义了自己的数据结构（LSM，B-tree等）来管理KV数据，但这些软件层往往需要较高的内存，同时也导致较高的写入放大。

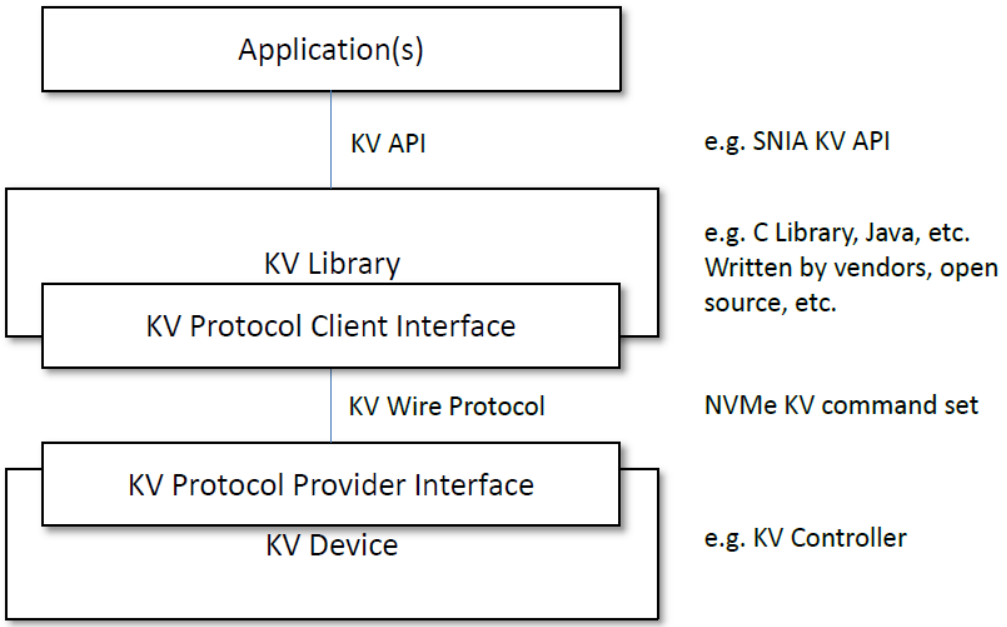
- KV SSD的KV存储栈较thin，通过降低cpu的内存利用来减少主机系统的负担。





KV SSD的标准化

- NVMe在设备驱动和KV SSD之间定义了接口标准，包括Store，Retrieve，Delete，Exist和Iterate。
- SNIA在上层应用和设备驱动之间定义了API规范。



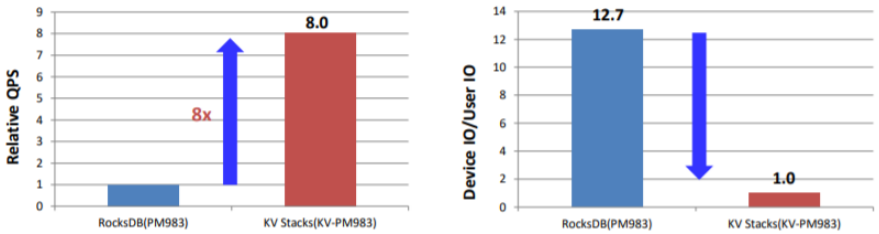
KV SSD的性能

KV SSD相对于在普通SSD上运行的Rocksdb，具有：

- 更好的性能
- 更好的IO效率
- 更好的扩展性

Performance: Random PUT

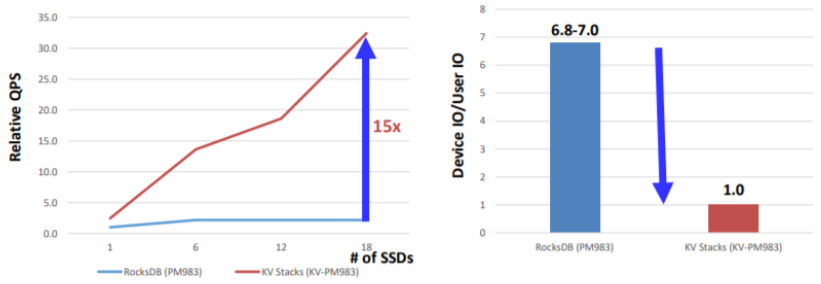
- 8x more QPS (Query Per Second) with KV Stacks than RocksDB on block SSD
- 90+% less traffic goes from host to device with KV SSD than RocksDB on block device



* Workload: 100% random put, 16 byte keys of random uniform distribution, 4KB-fixed values on single PM983 and KV-PM983 in a clean state

Scale-up Performance: Random Key PUT

- 15x IO performance over S/W key value store on block devices



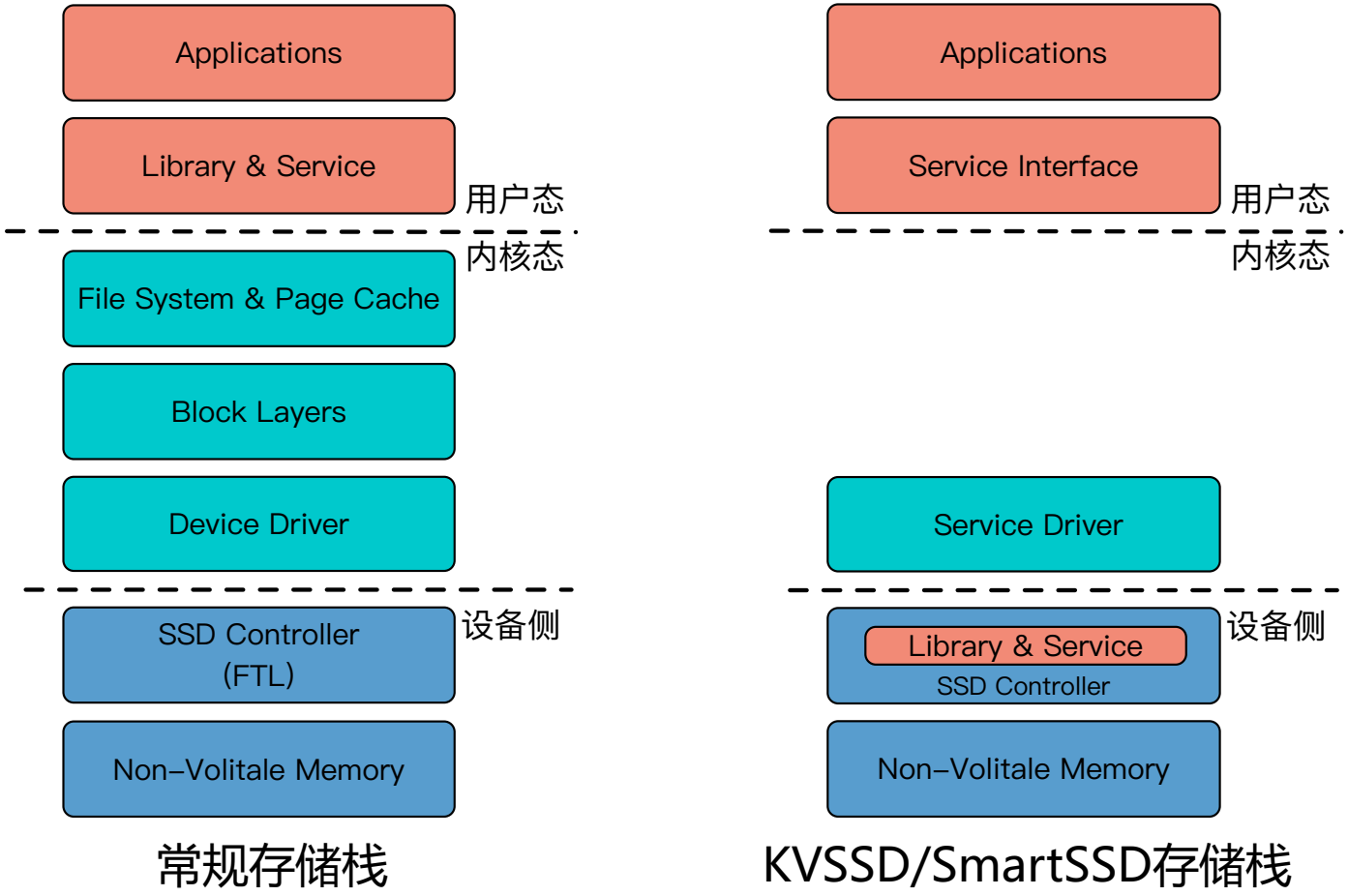
Relative performance to the maximum aggregate RocksDB random Put QPS for 1 SSD with a default configuration for 1 PM983 SSD in a clean state.
System: Ubuntu 16.04.2 LTS, , Ext4, RAID0 for block SSDs, Actual CPU utilization could be 70-90% at CPU saturation point.
Workload: 100% puts, 16 byte keys of random uniform distribution for RocksDB v. 5.0.2, 4KB-fixed values, 24 RocksDB instances with 4 client threads, 50GB/Instance or
Data is used

COLLABORATE. INNOVATE. GROW.

SAMSUNG

存储栈发展趋势

存储栈的offload (软件 -> 硬件) -- KVSSD为例



存储栈发展趋势

软件定义存储 (硬件 -> 软件) -- Open-channel SSD为例

12TB Flash
12GB User NVRAM
NVMe PCIe x8 Gen3 Edge Card

将部分的FTL功能移到Host server，少部分留在SSD controller。相比传统SSD更灵活。

存储栈发展趋势

软件定义存储 (硬件 -> 软件) -- Open-channel SSD为例

原FTL

data placement
I/O scheduling
Garbage collection
Media-centric metadata
error handling
Scrubbing
wear-leveling

Open-channel

data placement
I/O scheduling
Garbage collection

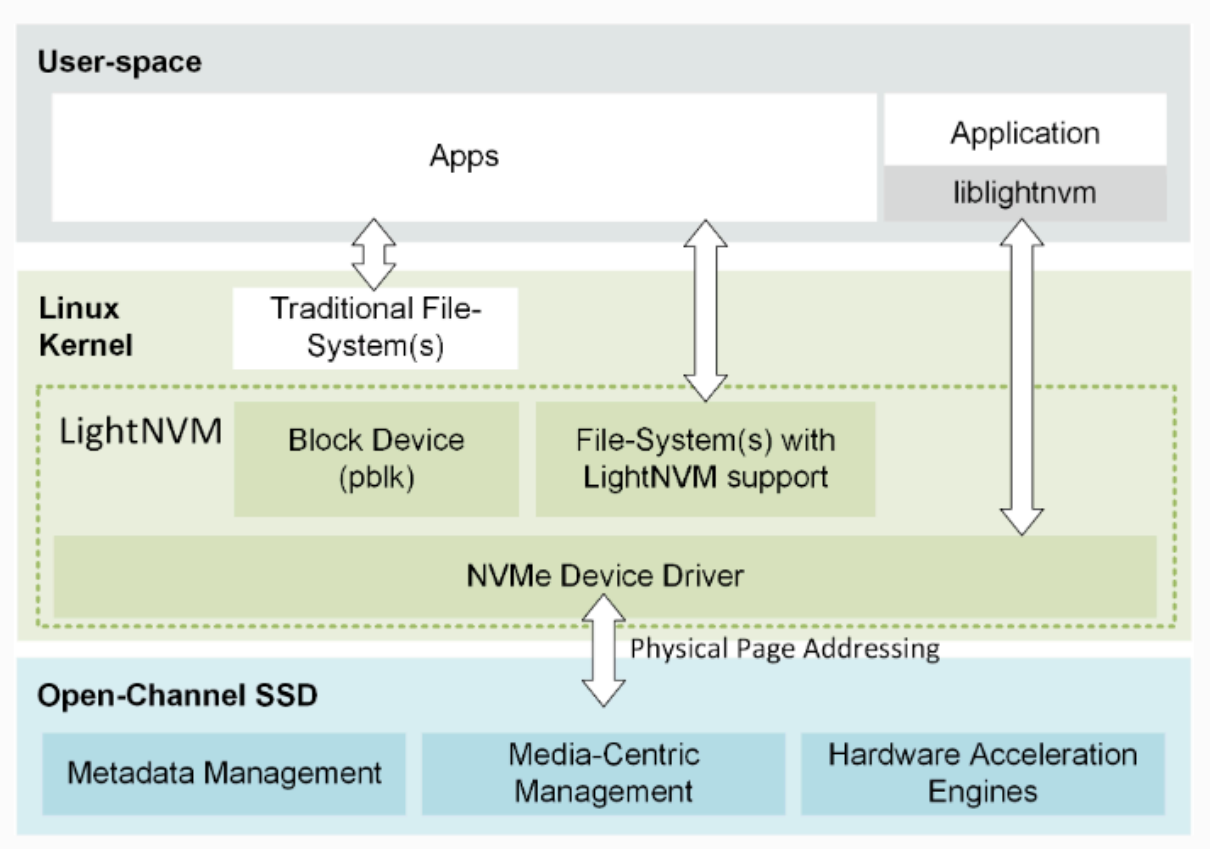
Host server-side

Media-centric metadata
error handling
Scrubbing
wear-leveling

Device-side

存储栈发展趋势

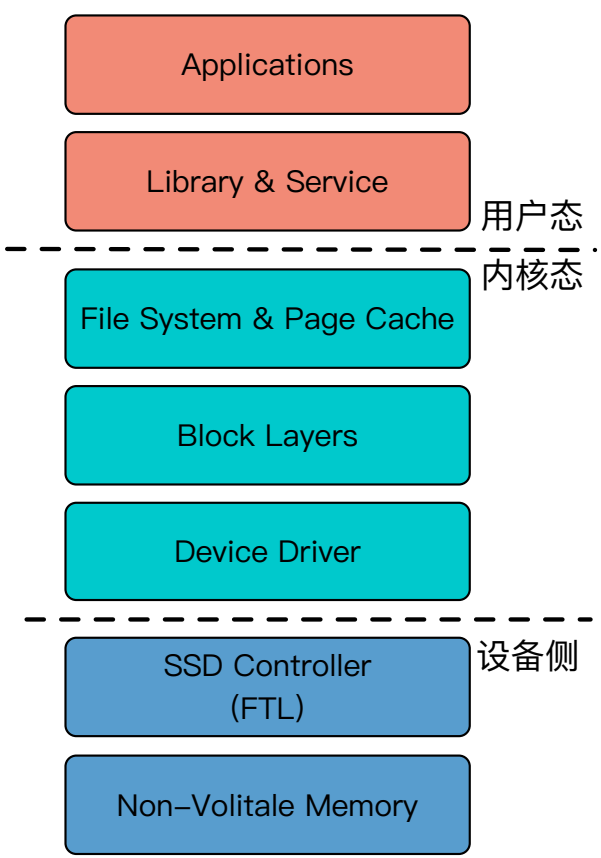
软件定义存储 (硬件 -> 软件) -- Open-channel SSD为例



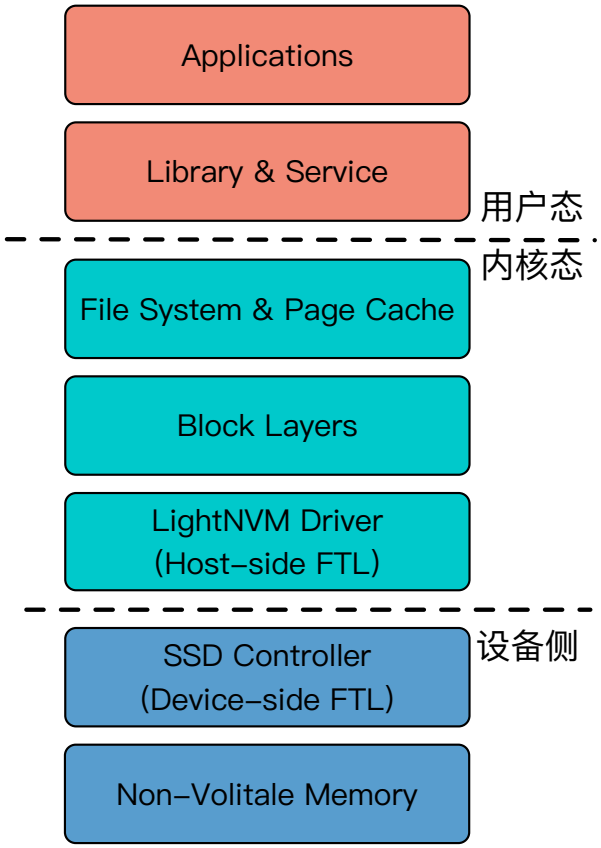
LightNVM 架构图

存储栈发展趋势

软件定义存储 (硬件 -> 软件) -- Open-channel SSD为例



常规存储栈



Open-channel SSD存储栈

Thanks! Questions?

参考:

[1] 《大话存储2:存储系统架构与底层原理极限剖析》
[2] 《深入浅出SSD：固态存储核心技术、原理与实战》
[3] 《Operating Systems: Three Easy Pieces》, Remzi H. Arpaci-Dusseau and Andrea C., Arpaci-Dusseau, Arpaci-Dusseau Books , March, 2015 (Version 0.80)
[4] https://unix.stackexchange.com/questions/144561/in-what-sense-does-sata-talk-scsi-how-much-is-shared-between-scsi-and-ata
[5] https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram
[6] https://www.slideshare.net/kerneltlv/high-performance-storage-devices-in-the-linux-kernel
[7] A block layer introduction, https://lwn.net/Articles/736534/
[8] A. Foong and F. Hady, "Platform Storage Performance With 3D XPoint Technology," vol. 105, no. 9, pp. 1822–1833, 2017.
[9] InnoDB Architecture, https://dev.mysql.com/doc/refman/8.0/en/innodb-architecture.html
[10] https://www.slideshare.net/igalia/improving-the-performance-of-the-qcow2-format-kvm-forum-2017
[11] L. S. Hajnoczi and K. V. M. Forum, "Applying Polling Techniques to QEMU," 2017.
[12] Hennessy, John L., and David A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.
[13] Open-Channel Solid State Drives, https://events.static.linuxfound.org/sites/events/files/slides/LightNVM-Vault2015.pdf
[14] M. Bjørling, J. Axboe, D. Nellans, and P. Bonnet, "Linux block IO: Introducing Multi-queue SSD Access on Multi-core Systems," *Systor '13*, p. 1, 2013.