E8 STORAGE

Come join us at careers@e8storage.com
https://goo.gl/FOuczd

HIGH PERFORMANCE STORAGE DEVICES IN KERNEL

E8 Storage

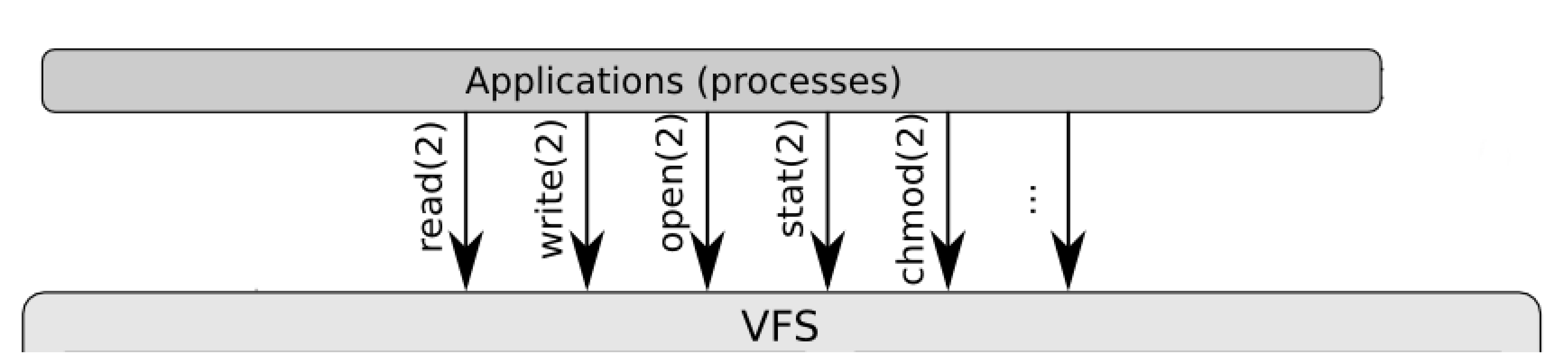
By Evgeny Budilovsky / @budevg

12 Jan 2016

STORAGE STACK 101

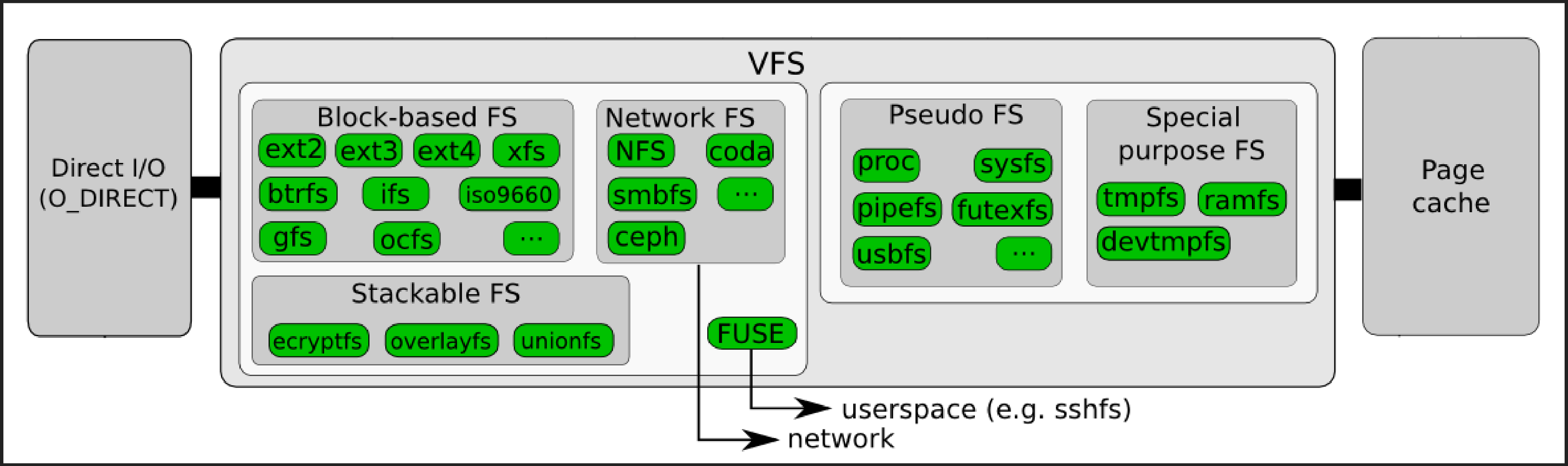
APPLICATION

- Application invokes system calls (read, write, mmap) on files
    - Block device files - direct access to block device
    - Regular files - access through specific file system (ext4, btrfs, etc.)



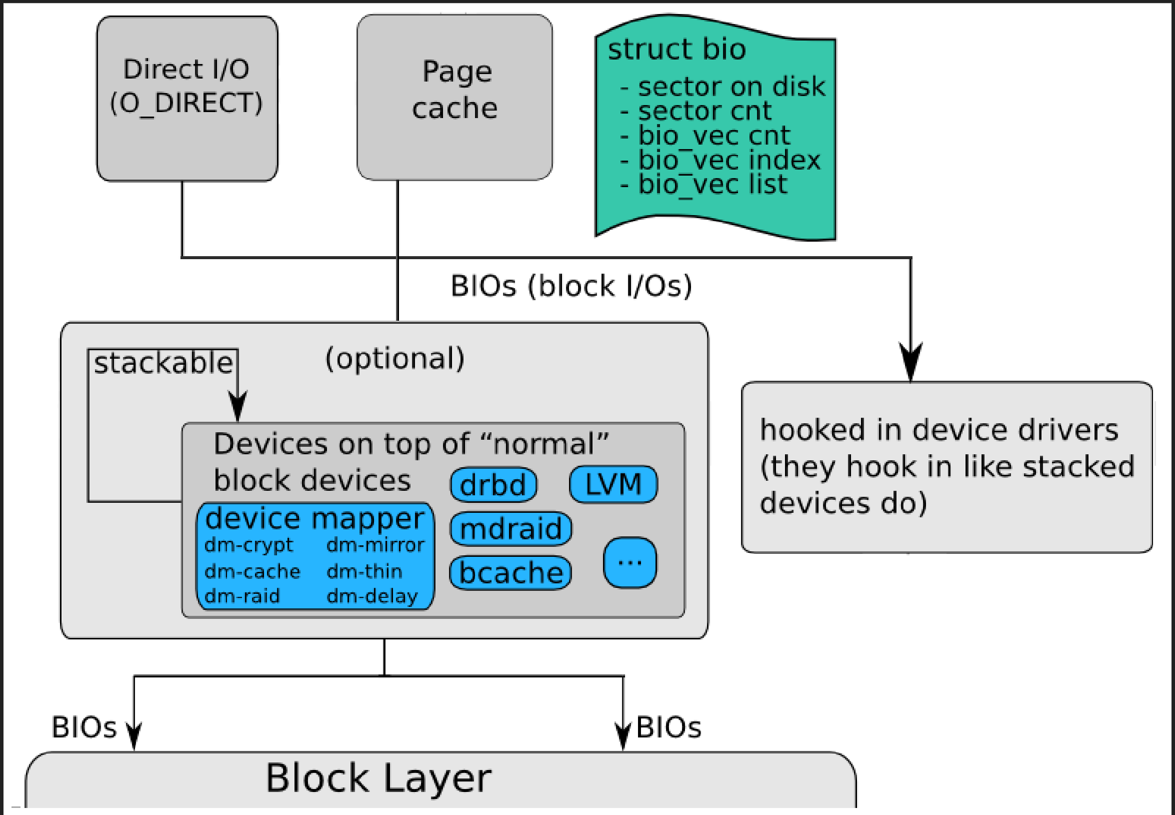
FILE SYSTEM

- system calls go through **VFS** layer
- specific file system logic applied
- read/write operations translated into read/write of memory pages
- **Page cache** involved to reduce disk access



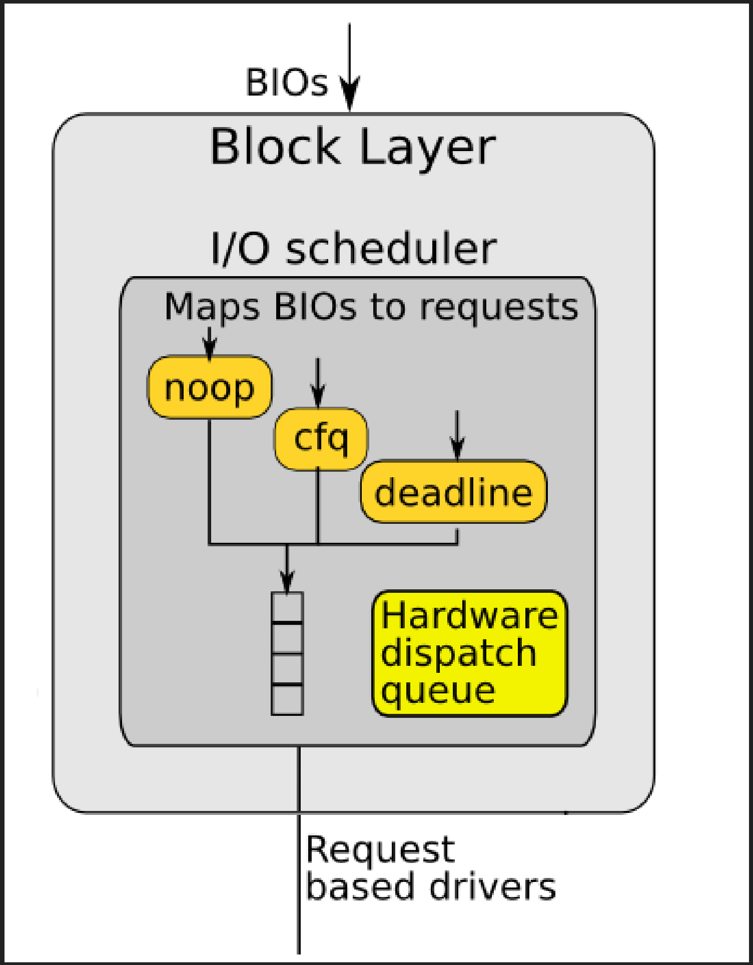
BIO LAYER (STRUCT BIO)

- File system constructs BIO unit which is the main unit of IO
- Dispatch bio to block layer / bypass driver (*make_request_fn driver*)



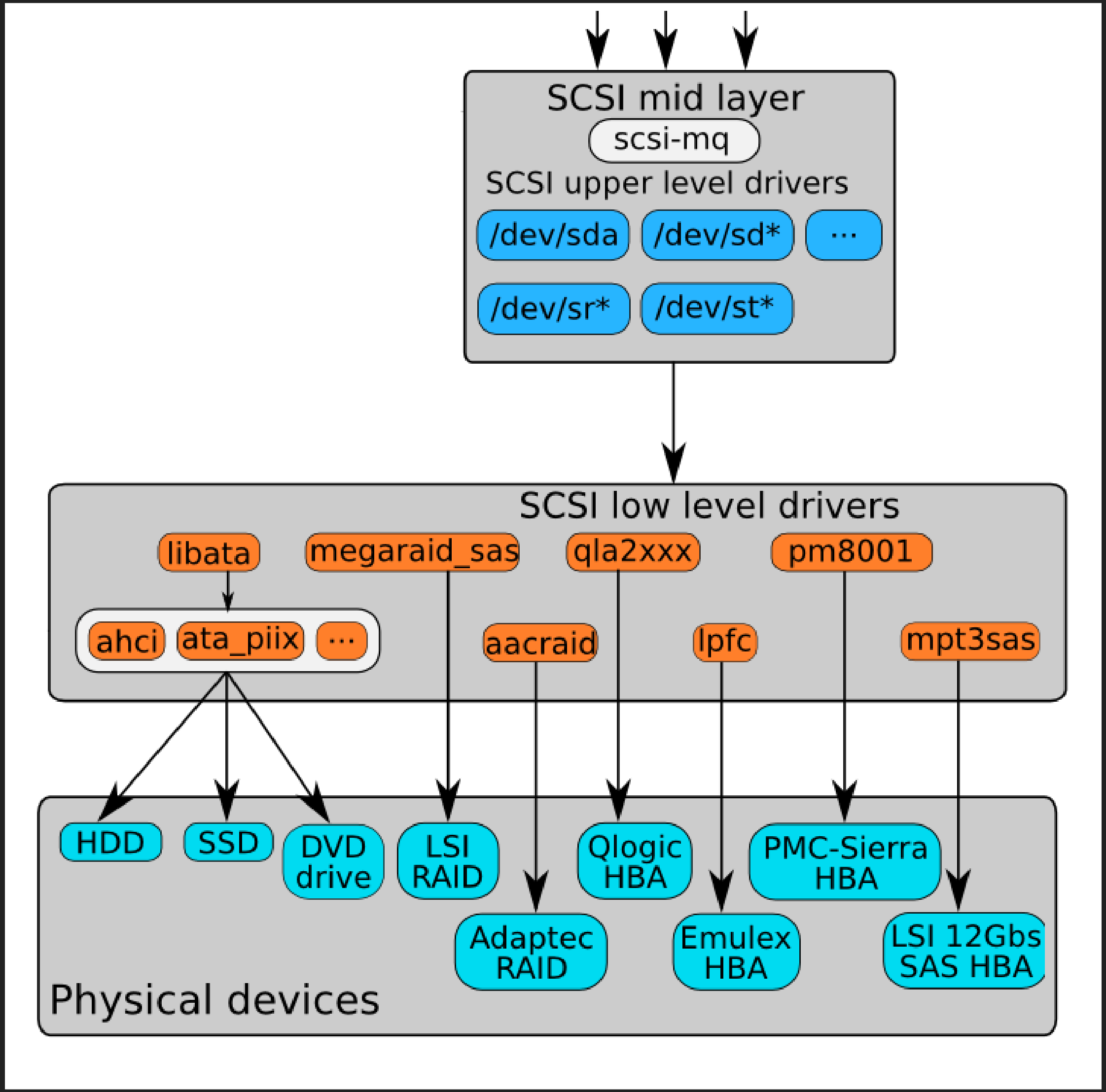
BLOCK LAYER (STRUCT REQUEST)

- Setup *struct request* and move it to request queue (single queue per device)
- IO scheduler can delay/merge requests
- Dispatch requests
  - To hardware drivers (*request_fn* driver)
  - To scsi layer (scsi devices)



SCSI LAYER (STRUCT SCSI_CMND)

- translate request into scsi command
- dispatch command to low level scsi drivers



WRITE (O_DIRECT)

```
# Application
perf record -g dd if=/dev/zero of=1.bin bs=4k \
     count=1000000 oflag=direct
...
perf report --call-graph --stdio
```

```
# Submit into block queue (bypass page cache)
write
  system_call_fastpath
    sys_write
      vfs_write
        new_sync_write
          ext4_file_write_iter
            __generic_file_write_iter
              generic_file_direct_write
                ext4_direct_IO
                  ext4_ind_direct_IO
                    __blockdev_direct_IO
                      do_blockdev_direct_IO
                        submit_bio
                          generic_make_request
```

```
# after io scheduling submit to scsi layer and to the hardware
     ...
io_schedule
  blk_flush_plug_list
   queue_unplugged
    __blk_run_queue
      scsi_request_fn
        scsi_dispatch_cmd
          ata_scsi_queuecmd
```

WRITE (WITH PAGE CACHE)

```
# Application
perf record -g dd if=/dev/zero of=1.bin bs=4k \
     count=1000000
...
perf report --call-graph --stdio
```

```
# write data into page cache
write
  system_call_fastpath
    sys_write
      vfs_write
        new_sync_write
          ext4_file_write_iter
            __generic_file_write_iter
              generic_perform_write
                ext4_da_write_begin
                  grab_cache_page_write_begin
                    pagecache_get_page
                      __page_cache_alloc
                        alloc_pages_current
```

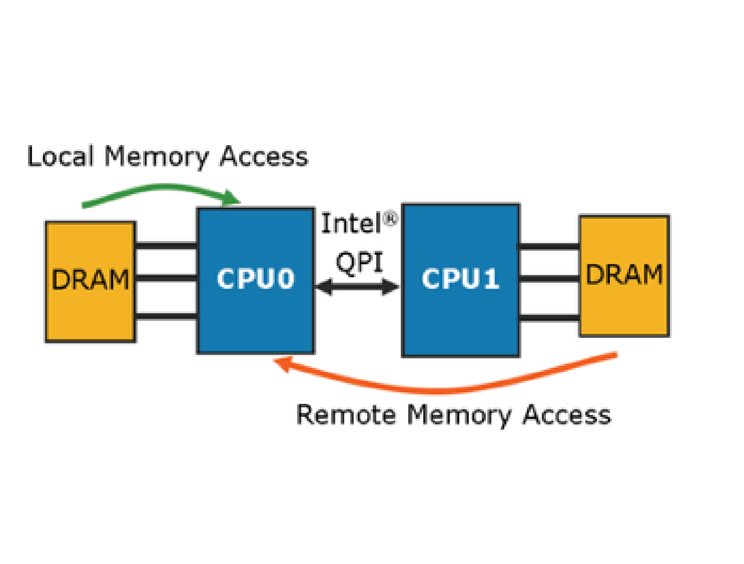
```
# asynchronously flush dirty pages to disk

kthread
  worker_thread
    process_one_work
      bdi_writeback_workfn
        wb_writeback
          __writeback_inodes_wb
            writeback_sb_inodes
              __writeback_single_inode
                do_writepages
                  ext4_writepages
                    mpage_map_and_submit_buffers
                      mpage_submit_page
                        ext4_bio_write_page
                          ext4_io_submit
                            submit_bio
                              generic_make_request
                                blk_queue_bio
```

HIGH PERFORMANCE BLOCK DEVICES

CHANGES IN STORAGE WORLD

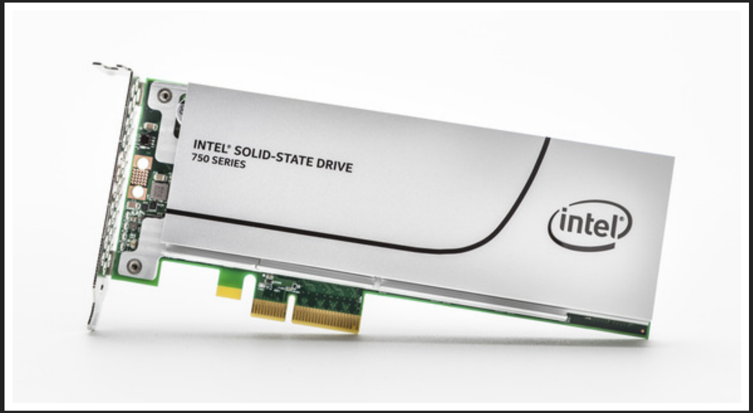
- Rotational devices (HDD) "hundreds" of IOPS , "tens" of milliseconds latency
- Today, flash based devices (SSD) "hundreds of thousands" of IOPS, "tens" of microseconds latency
- Large internal data parallelism
- Increase in cores and NUMA architecture
- New standardized storage interfaces (NVME)



NVM EXPRESS

HIGH PERFORMANCE STANDARD

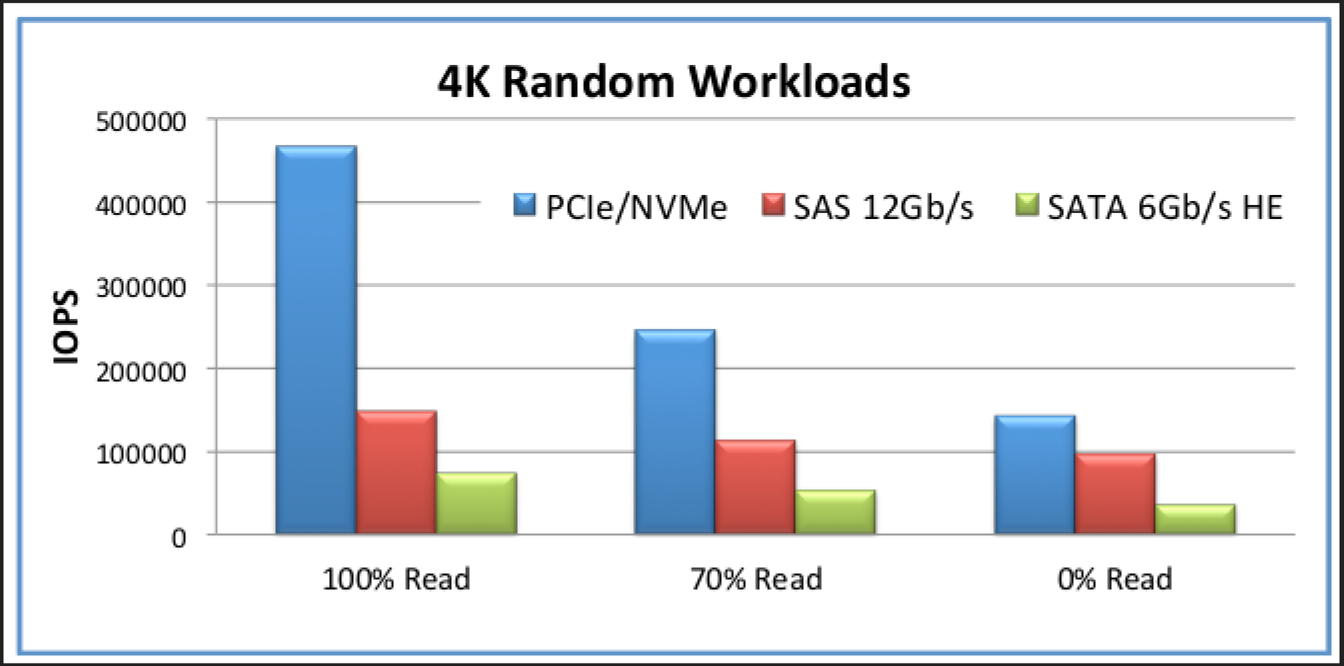
- Standardized interface for PCIe SSDs
- Designed from the ground up to exploit
  - Low latency of today's PCIe-based SSD's
  - Parallelism of today's CPU's



BEATS AHCI STANDARD FOR SATA HOSTS

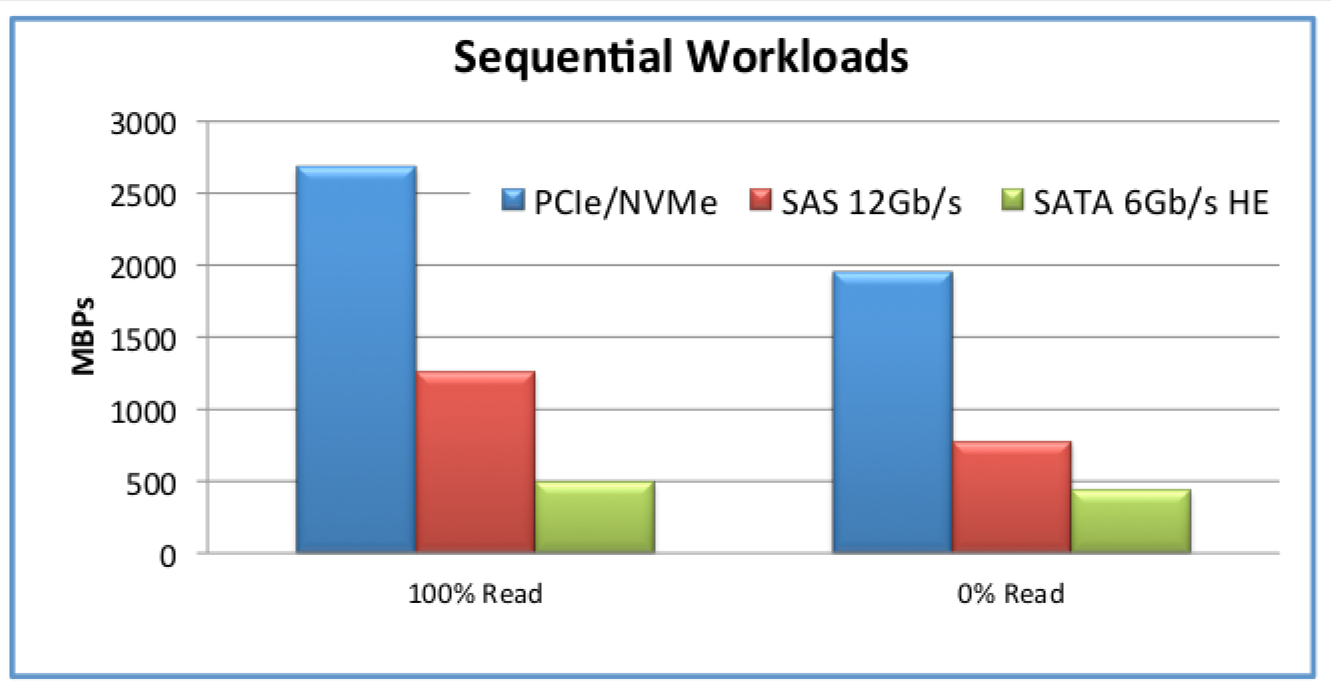
	AHCI	NVME
Maximum Queue Depth	1 command queue 32 commands per Q	64K queues 64K Commands per Q
Un-cacheable register accesses (2K cycles	6 per non-queued command 9 per queued command	2 per command
MSI-X and Interrupt Steering	Single interrupt; no steering	2K MSI-X interrupts
Parallelism & Multiple Threads	Requires synchronization lock to issue command	No locking
Efficiency for 4KB Commands	Command parameters require two serialized host DRAM fetches	Command parameters in one 64B fetch

IOPS

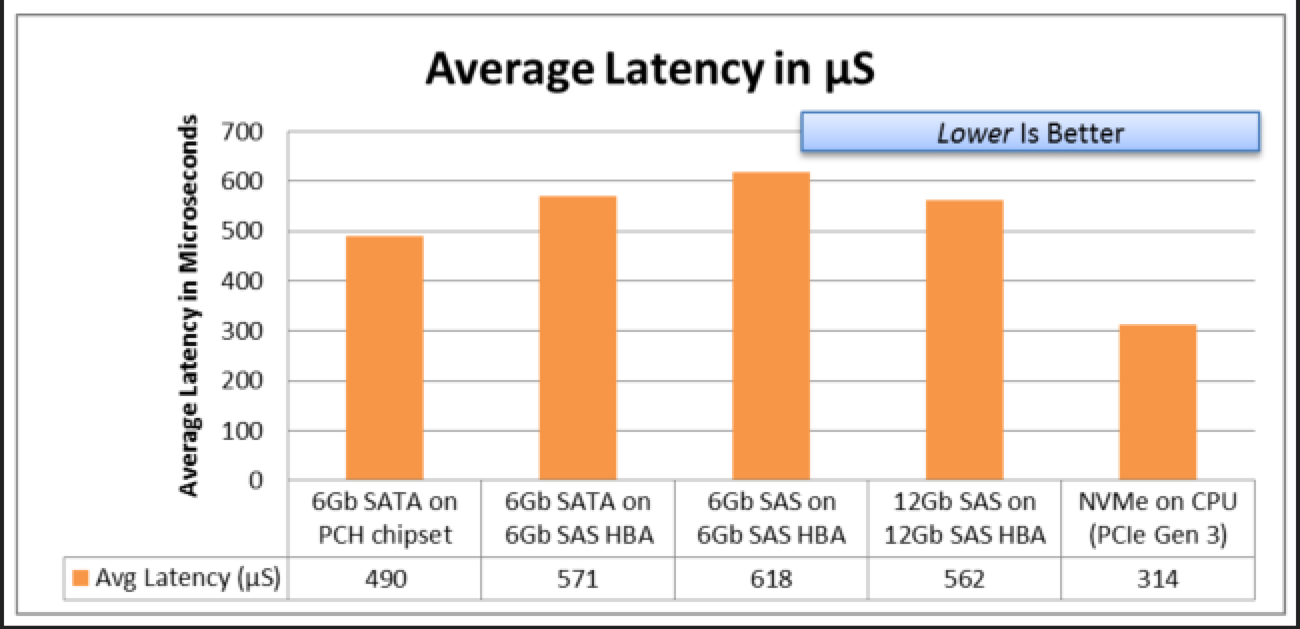


- consumer grade NVME SSDs (enterprise grade have much better performance)
- 100% writes less impressive due to NAND limitation

BANDWIDTH


Sequential Workloads
MBPs
3000
2500
2000
1500
1000
500
0
🟦 PCIe/NVMe
🟥 SAS 12Gb/s
🟩 SATA 6Gb/s HE
100% Read
0% Read

LATENCY



Average Latency in µS

Lower Is Better

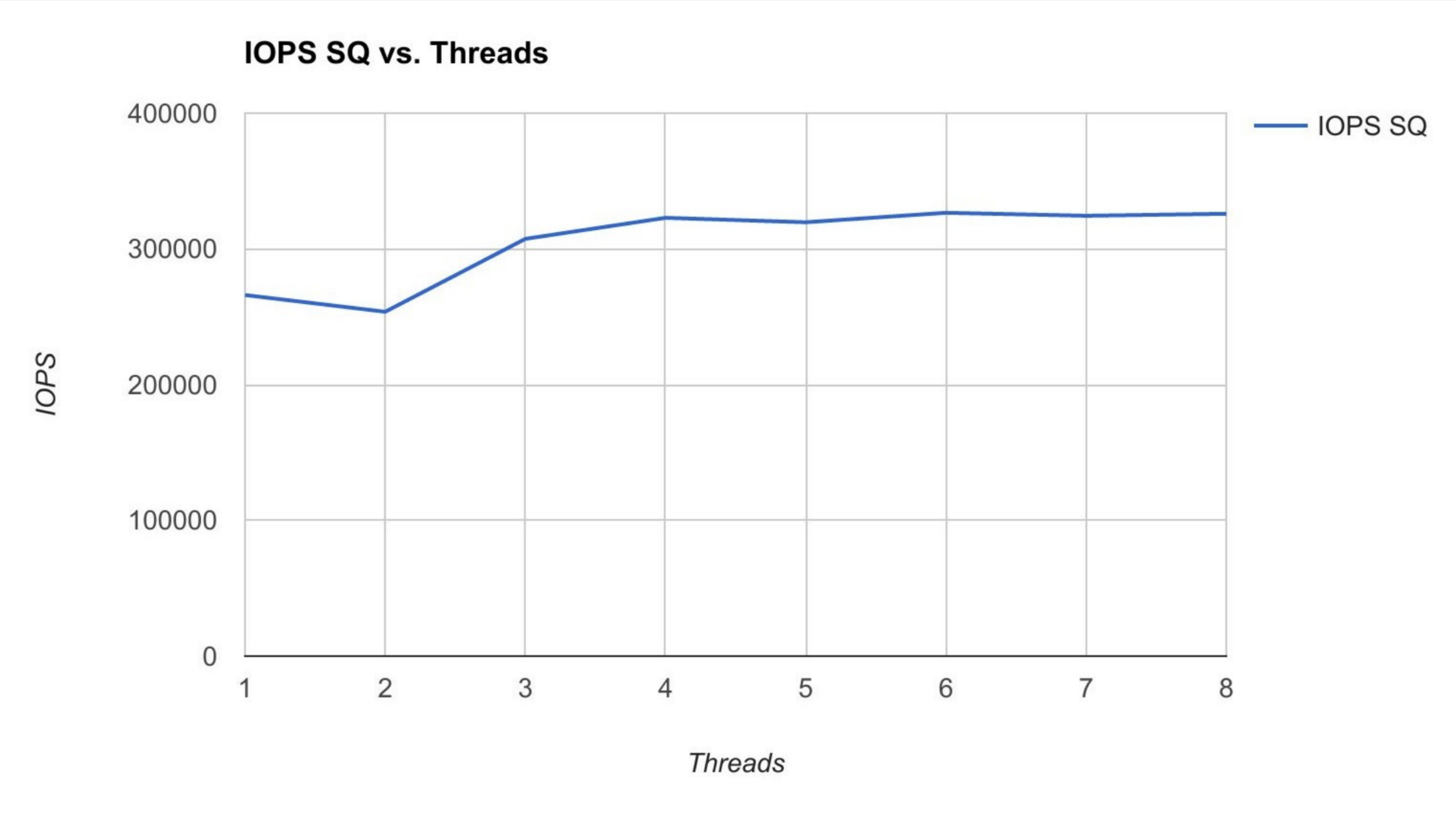
	6Gb SATA on PCH chipset	6Gb SATA on 6Gb SAS HBA	6Gb SAS on 6Gb SAS HBA	12Gb SAS on 12Gb SAS HBA	NVMe on CPU (PCIe Gen 3)
🟧 Avg Latency (µS)	490	571	618	562	314

THE OLD STACK DOES NOT SCALE

THE NULL_BLK EXPERIMENT

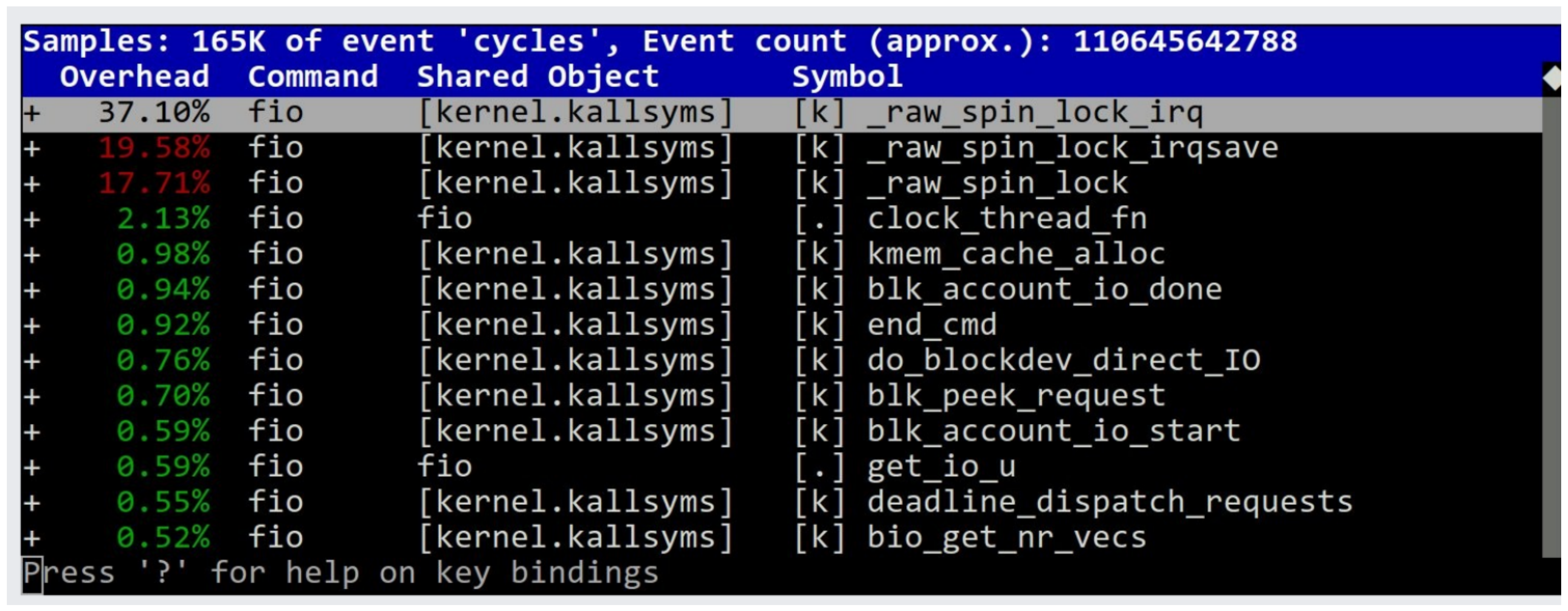
- Jens Axboe (Facebook)
- null_blk configuration
  - queue_mode=1(rq) completion_nsec=0 irqmode=0(none)
- fio
  - Each thread does pread(2), 4k, randomly, O_DIRECT
- Each added thread alternates between the two available NUMA nodes (2 socket system, 32 threads)

LIMITED PERFORMANCE



PERF

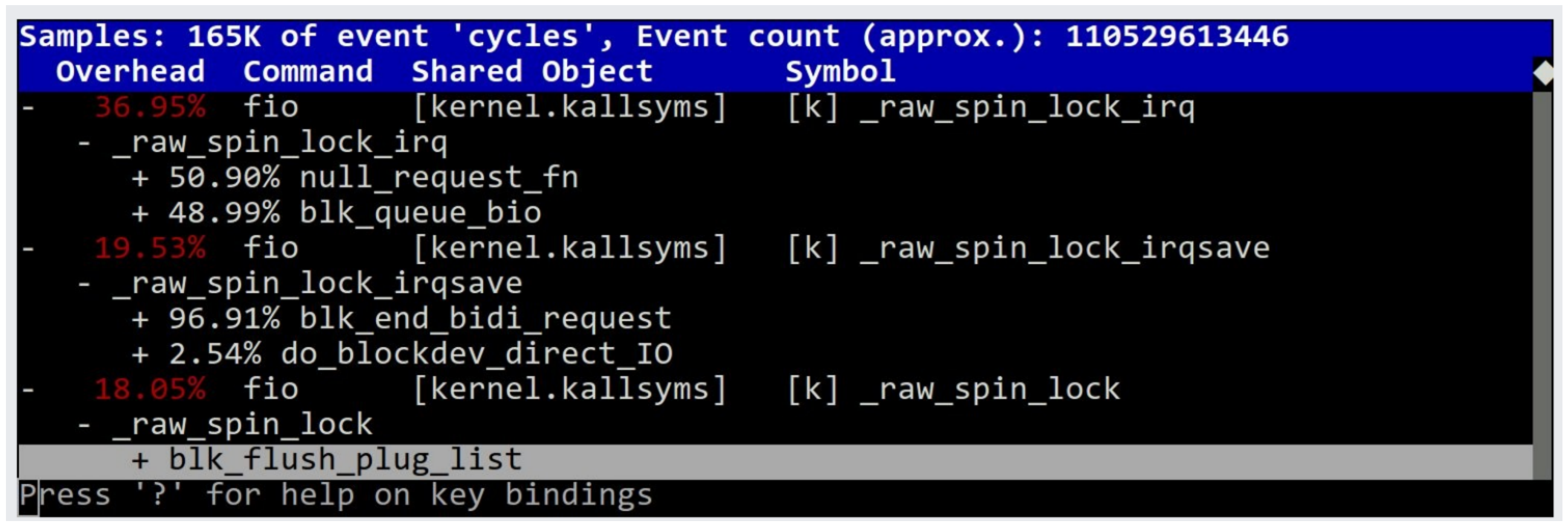
- Spinlock contention



```
Samples: 165K of event 'cycles', Event count (approx.): 110645642788
  Overhead  Command  Shared Object      Symbol
+   37.10%  fio      [kernel.kallsyms]  [k] _raw_spin_lock_irq
+   19.58%  fio      [kernel.kallsyms]  [k] _raw_spin_lock_irqsave
+   17.71%  fio      [kernel.kallsyms]  [k] _raw_spin_lock
+    2.13%  fio      fio                [.] clock_thread_fn
+    0.98%  fio      [kernel.kallsyms]  [k] kmem_cache_alloc
+    0.94%  fio      [kernel.kallsyms]  [k] blk_account_io_done
+    0.92%  fio      [kernel.kallsyms]  [k] end_cmd
+    0.76%  fio      [kernel.kallsyms]  [k] do_blockdev_direct_IO
+    0.70%  fio      [kernel.kallsyms]  [k] blk_peek_request
+    0.59%  fio      [kernel.kallsyms]  [k] blk_account_io_start
+    0.59%  fio      fio                [.] get_io_u
+    0.55%  fio      [kernel.kallsyms]  [k] deadline_dispatch_requests
+    0.52%  fio      [kernel.kallsyms]  [k] bio_get_nr_vecs
Press '?' for help on key bindings
```

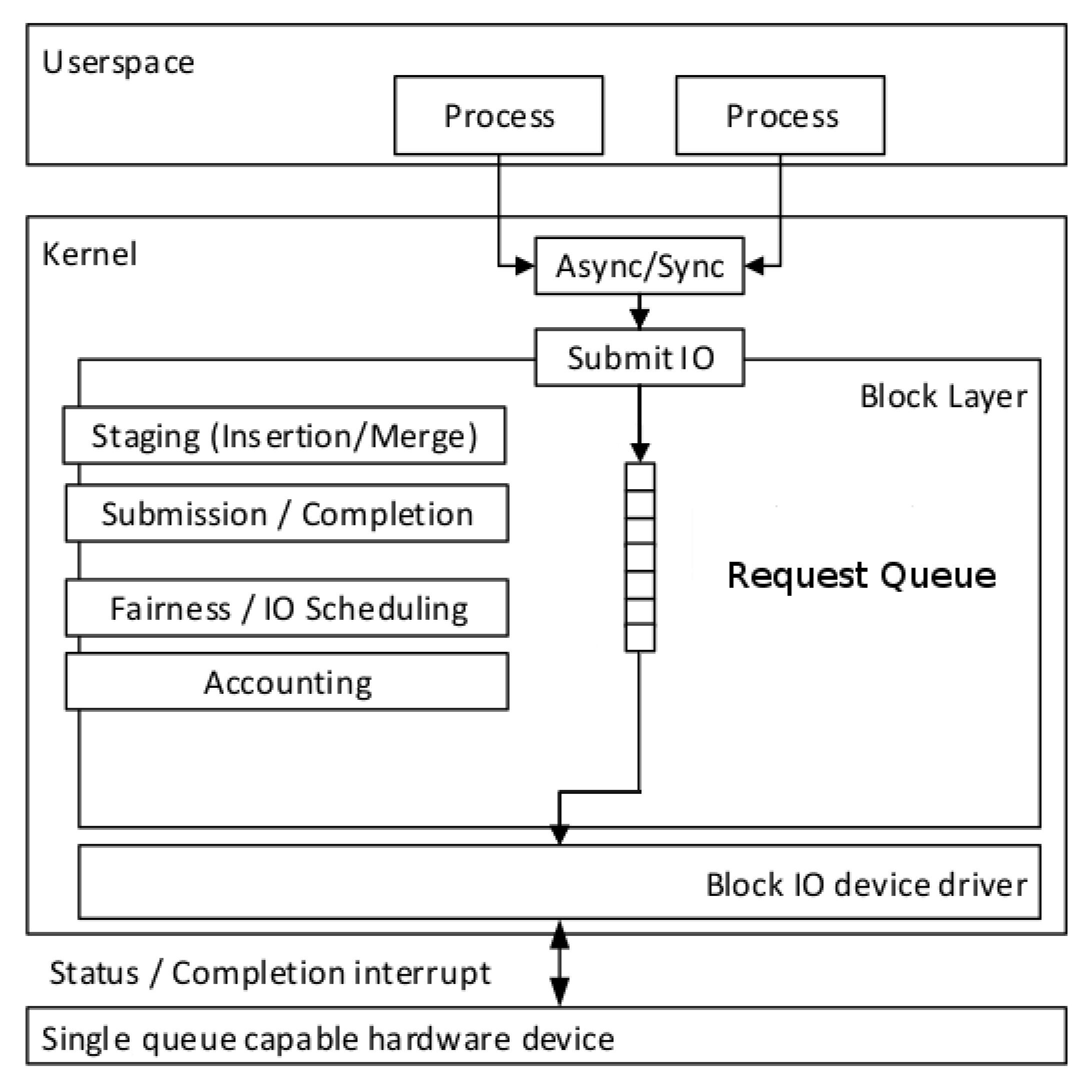
PERF

- 40% from sending request (blk_queue_bio)
- 20% from completing request (blk_end_bidi_request)
- 18% sending bio from the application to the bio layer (blk_flush_plug_list)



```
Samples: 165K of event 'cycles', Event count (approx.): 110529613446
  Overhead  Command  Shared Object      Symbol
-   36.95%  fio      [kernel.kallsyms]  [k] _raw_spin_lock_irq
   - _raw_spin_lock_irq
      + 50.90% null_request_fn
      + 48.99% blk_queue_bio
-   19.53%  fio      [kernel.kallsyms]  [k] _raw_spin_lock_irqsave
   - _raw_spin_lock_irqsave
      + 96.91% blk_end_bidi_request
      + 2.54% do_blockdev_direct_IO
-   18.05%  fio      [kernel.kallsyms]  [k] _raw_spin_lock
   - _raw_spin_lock
      + blk_flush_plug_list
Press '?' for help on key bindings
```

OLD STACK HAS SEVERE SCALING ISSUES



PROBLEMS

- Good scalability before block layer (file system, page cache, bio)
- Single shared queue is a problem
- We can use bypass mode driver which will work with bio's without getting into shared queue.
- Problem with bypass driver: code duplication

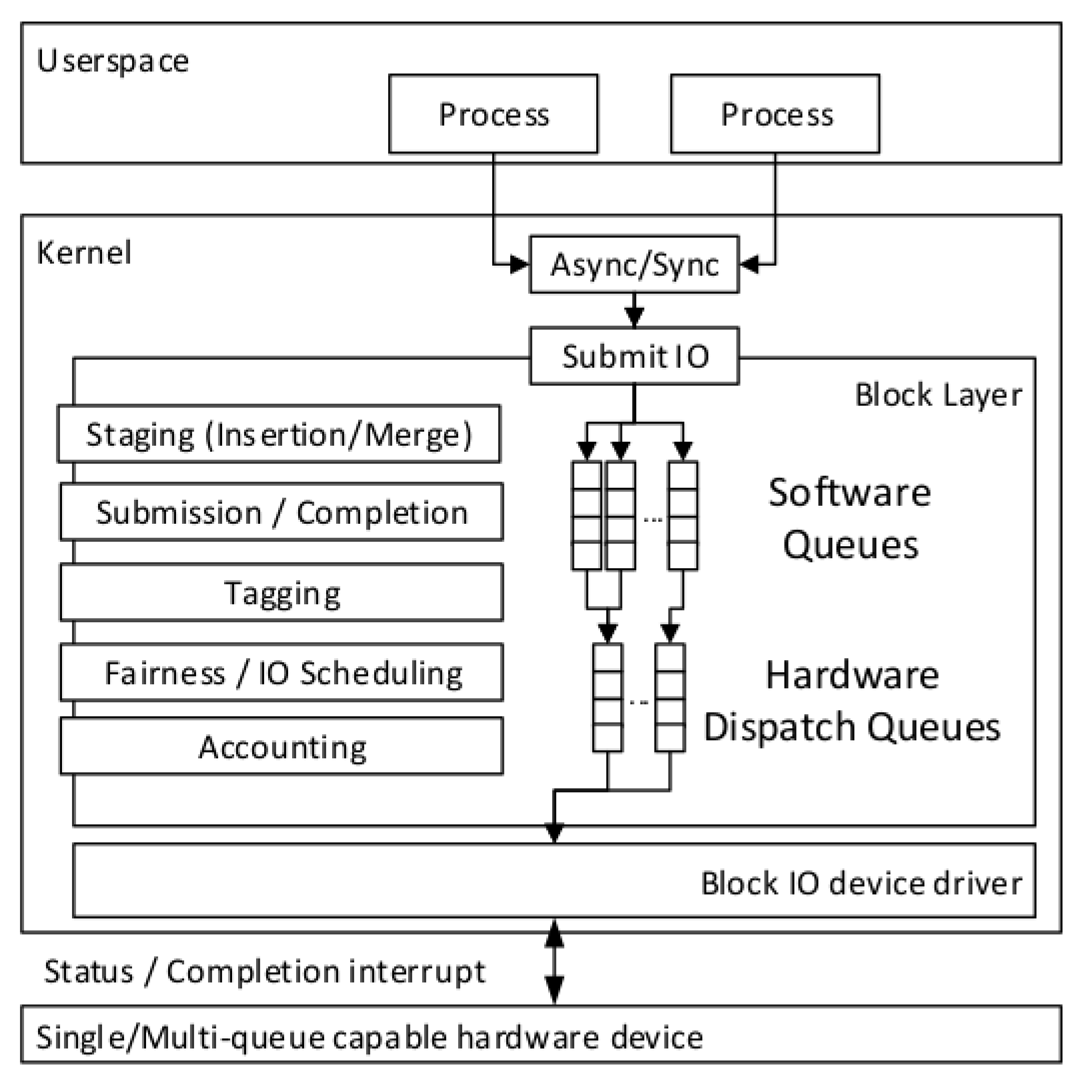
BLOCK MULTI-QUEUE TO THE RESCUE

HISTORY

- Prototyped in 2011
- Paper in SYSTOR 2013
- Merged into linux 3.13 (2014)
- A replacement for old block layer with different driver API
  - Drivers gradually converted to blk-mq (scsi-mq, nvme-core, virtio_blk)

ARCHITECTURE - 2 LAYERS OF QUEUES

- Application works with per-CPU software queue
- Multiple software queues map into hardware queues
- Number of HW queues is based on number of HW contexts supported by device
- Requests from HW queue submitted by low level driver to the device



ARCHITECTURE - ALLOCATION AND TAGGING

- IO tag
  - Is an integer value that uniquely identifies IO submitted to hardware
  - On completion we can use the tag to find out which IO was completed
  - Legacy drivers maintained their own implementation of tagging
- With block-mq, requests allocated at initialization time (based on queue depth)
- Tag and request allocations combined
- Avoids per request allocations in driver and tag maintenance

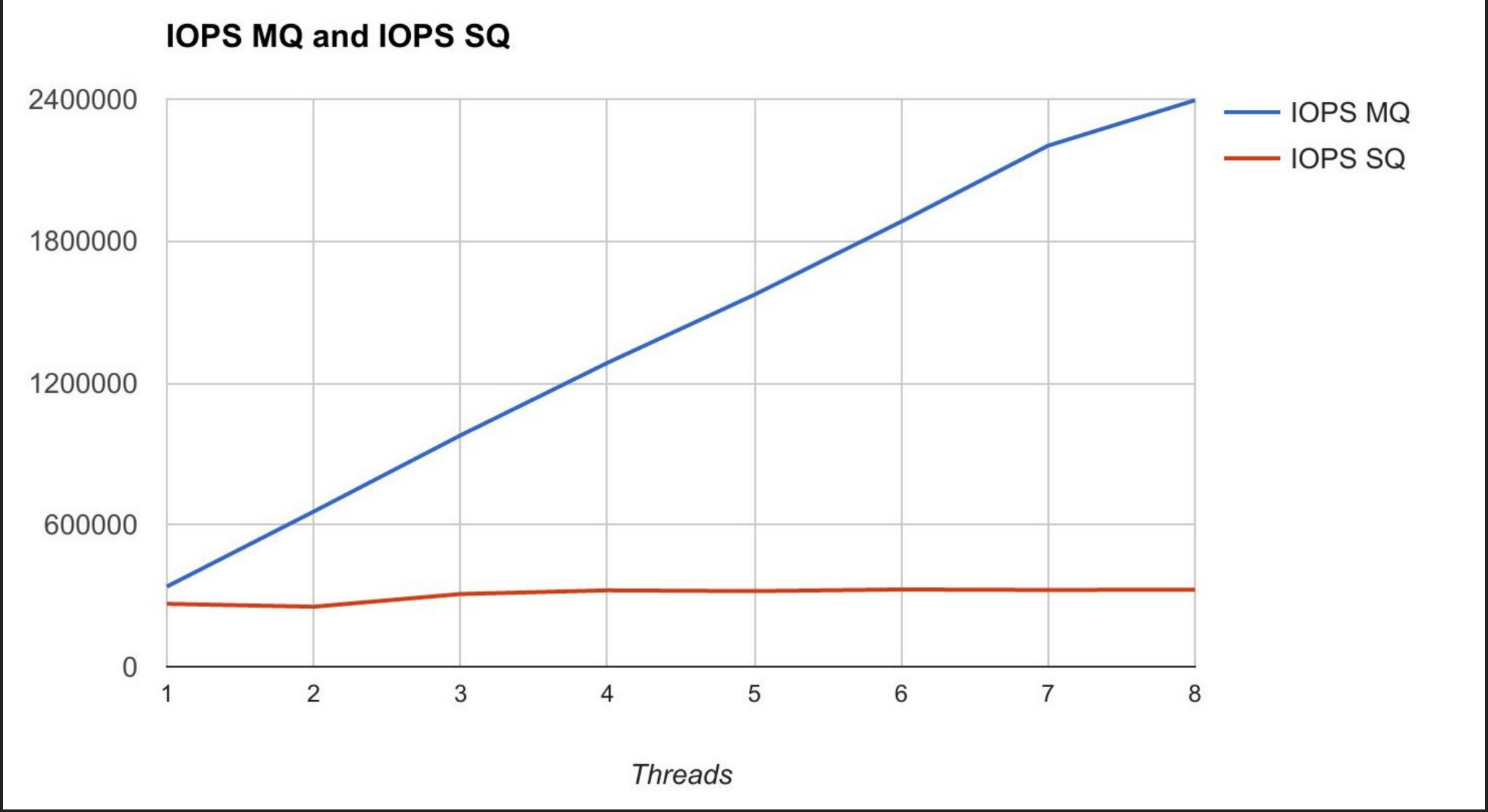
ARCHITECTURE - I/O COMPLETIONS

- Generally we want completions to be as local as possible
- Use IPIs to complete requests on submitting node
- Old block layer was using software interrupts instead of IPIs
- Best case there is an SQ/CQ pair for each core, with MSI-X interrupt setup for each CQ, steered to the relevant core
- IPIs used when there aren't enough interrupts/HW queues

NULL_BLK EXPERIMENT AGAIN

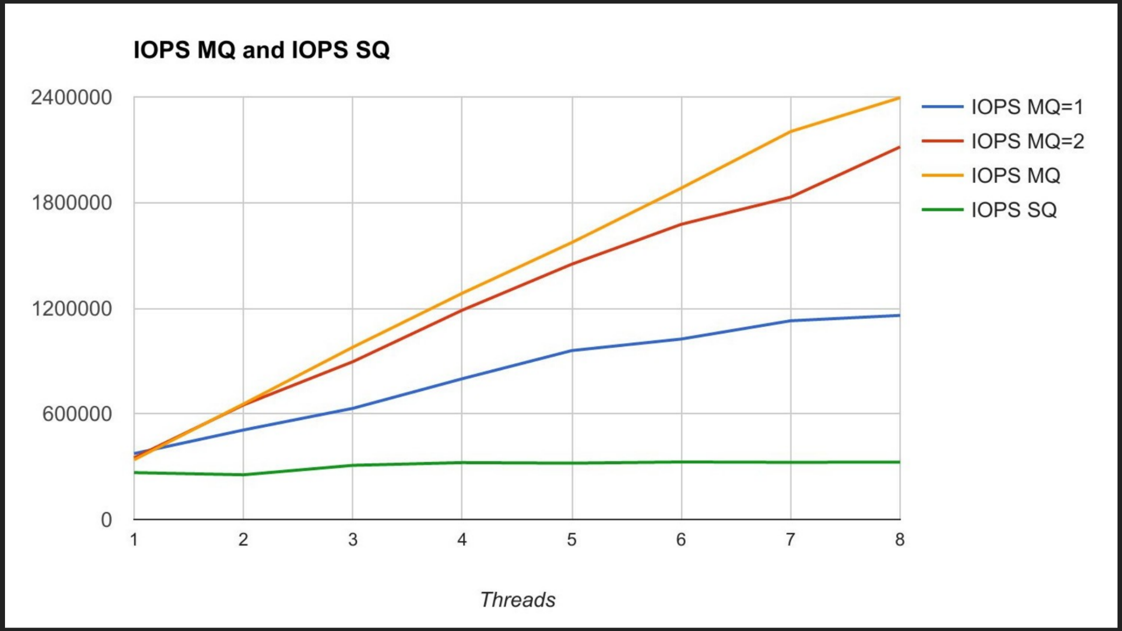
- null_blk configuration
  - queue_mode=2(multiqueue) completion_nsec=0 irqmode=0(none) submit_queues=32
- fio
  - Each thread does pread(2), 4k, randomly, O_DIRECT
- Each added thread alternates between the two available NUMA nodes (2 socket system, 32 threads)

SUCCESS


IOPS MQ and IOPS SQ
2400000
1800000
1200000
600000
0
1
2
3
4
5
6
7
8
Threads
IOPS MQ
IOPS SQ

WHAT ABOUT HARDWARE WITHOUT MULTI QUEUE SUPPORT

- Same null_blk setup
- 1/2/n hw queues in blk-mq
- mq-1 and mq-2 so close since we have 2 sockets system
- Numa issues eliminated once we have queue per numa node



CONVERSION PROGRESS

- mtip32xx (micron SSD)
- NVMe
- virtio_blk, xen block driver
- rbd (ceph block)
- loop
- ubi
- SCSI (scsi-mq)

SUMMARY

- Storage device performance has accelerated from hundreds of IOPS to hundreds thousands of IOPS
- Bottlenecks in software gradually eliminated by exploiting concurrency and introducing lock-less architecture
- blk-mq is one example

Questions ?

REFERENCES

1. Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems
2. The Performance Impact of NVMe and NVMe over Fabrics
3. Null block device driver
4. blk-mq: new multi-queue block IO queueing mechanism
5. fio
6. perf
7. Solving the Linux storage scalability bottlenecks - by Jens Axboe