facebook

Solving the Linux storage scalability bottlenecks

Jens Axboe
Software Engineer

Kernel Recipes 2015, Oct 2nd 2015

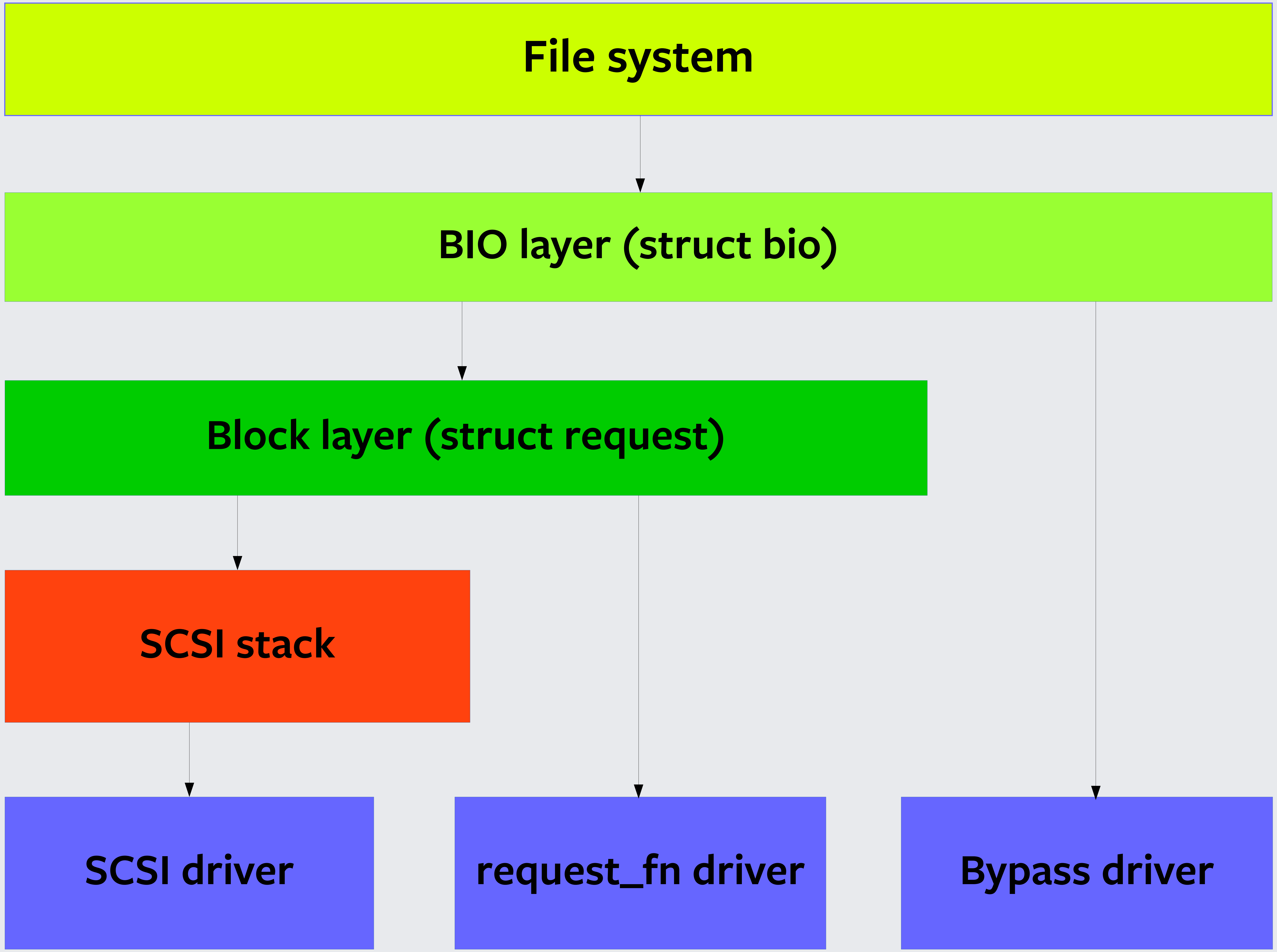
What are the issues?

- Devices went from “hundreds of IOPS” to “hundreds of thousands of IOPS”
- Increases in core count, and NUMA
- Existing IO stack has a lot of data sharing
  - For applications
  - And between submission and completion
- Existing heuristics and optimizations centered around slower storage

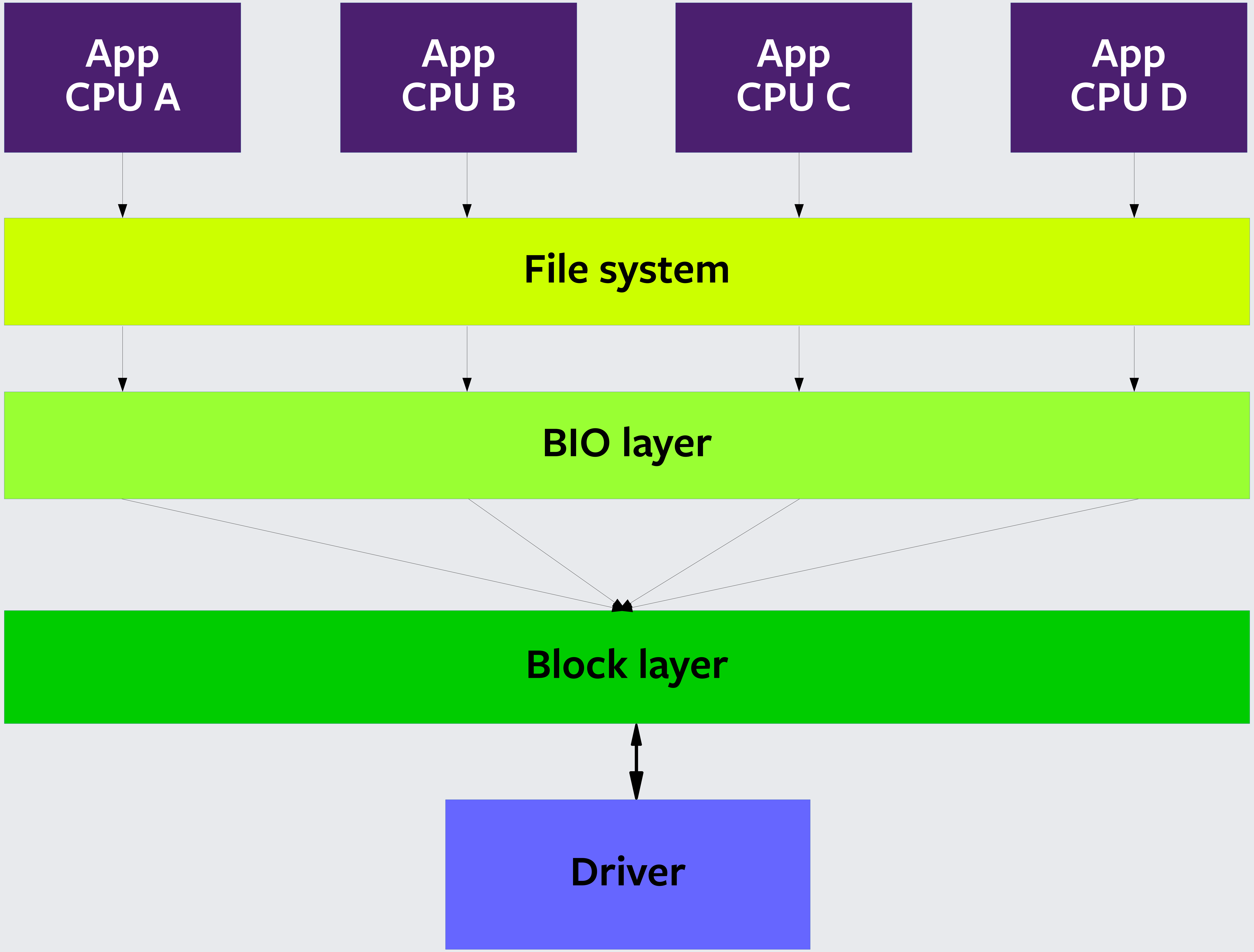
Observed problems

- The old stack had severe scaling issues
  - Even negative scaling
  - Wasting lots of CPU cycles
- This also lead to much higher latencies
- But where are the real scaling bottlenecks hidden?

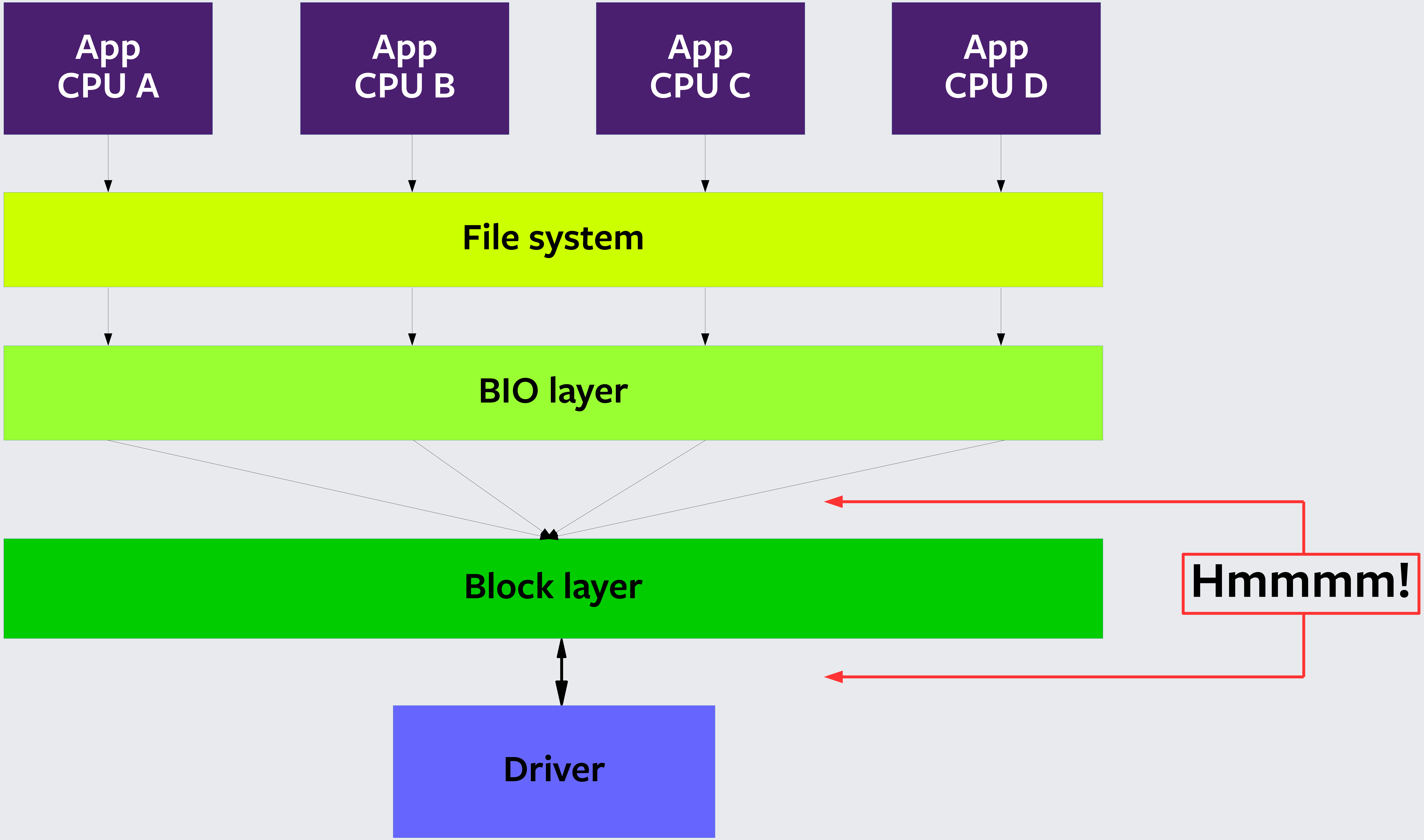
IO stack


File system
BIO layer (struct bio)
Block layer (struct request)
SCSI stack
SCSI driver
request_fn driver
Bypass driver

Seen from the application

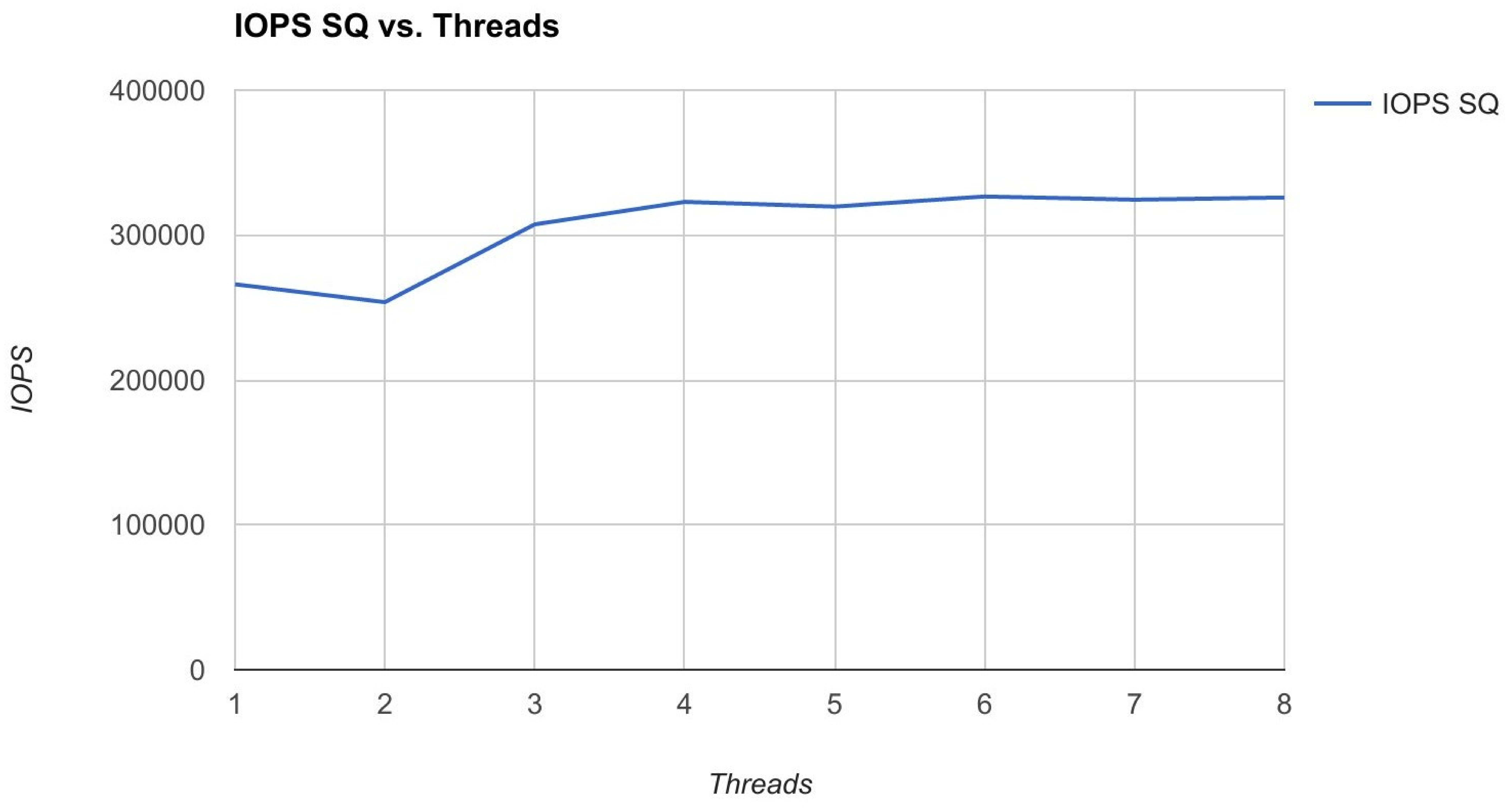


Seen from the application



Testing the theory

- At this point we may have a suspicion of where the bottleneck might be. Let's run a test and see if it backs up the theory.
- We use null_blk
  - queue_mode=1 completion_nsec=0 irqmode=0
- 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)

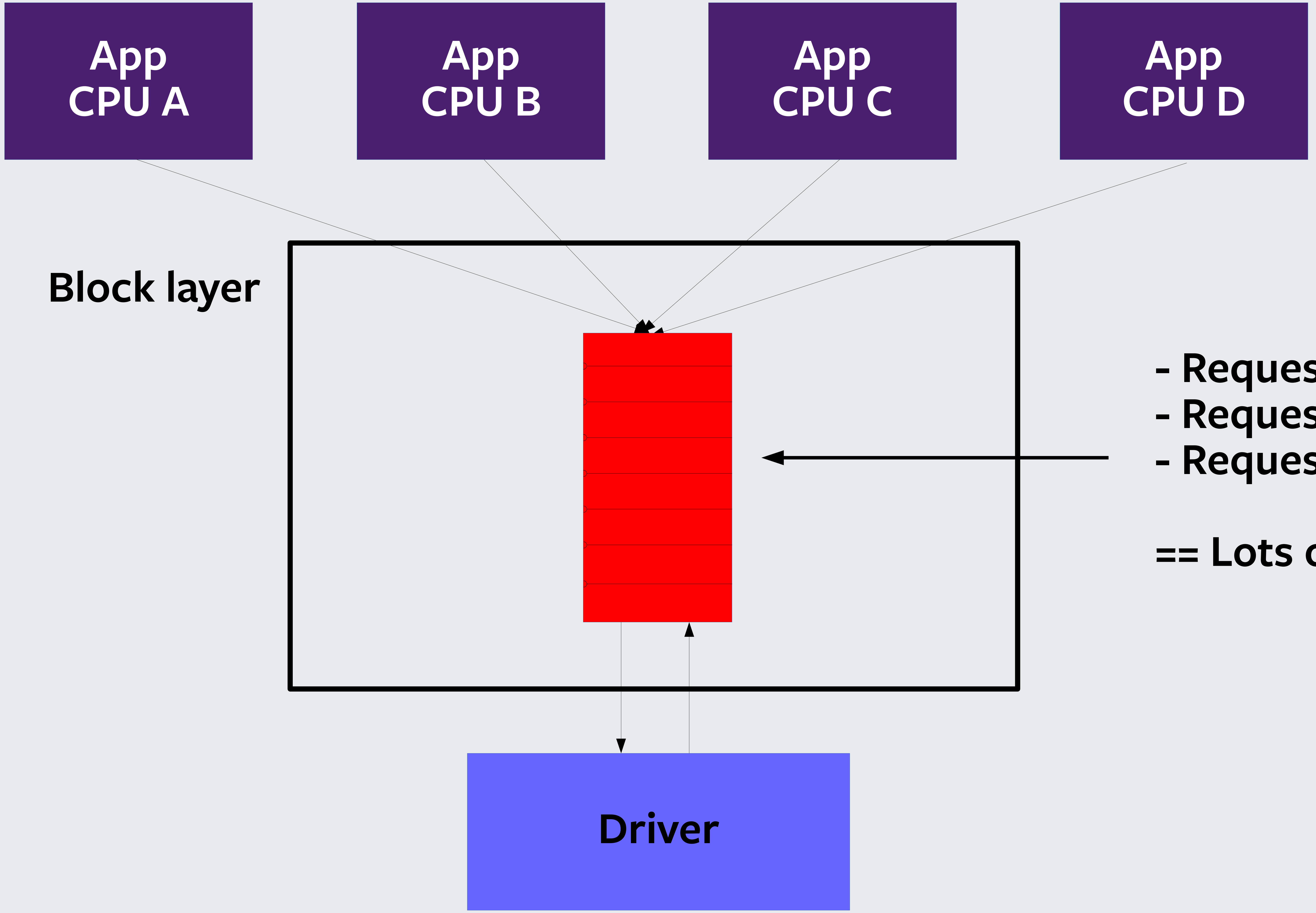

IOPS SQ vs. Threads
IOPS SQ
400000
300000
200000
100000
0
IOPS
1
2
3
4
5
6
7
8
Threads

```
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
```

That looks like a lot of lock contention… Fio reports spending 95% of the time in the kernel, looks like ~75% of that time is spinning on locks.

Looking at call graphs, it's a good mix of queue vs completion, and queue vs queue (and queue-to-block vs queue-to-driver).

```
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
```



- Requests placed for processing
- Requests retrieved by driver
- Requests completion signaled

== Lots of shared state!

Problem areas

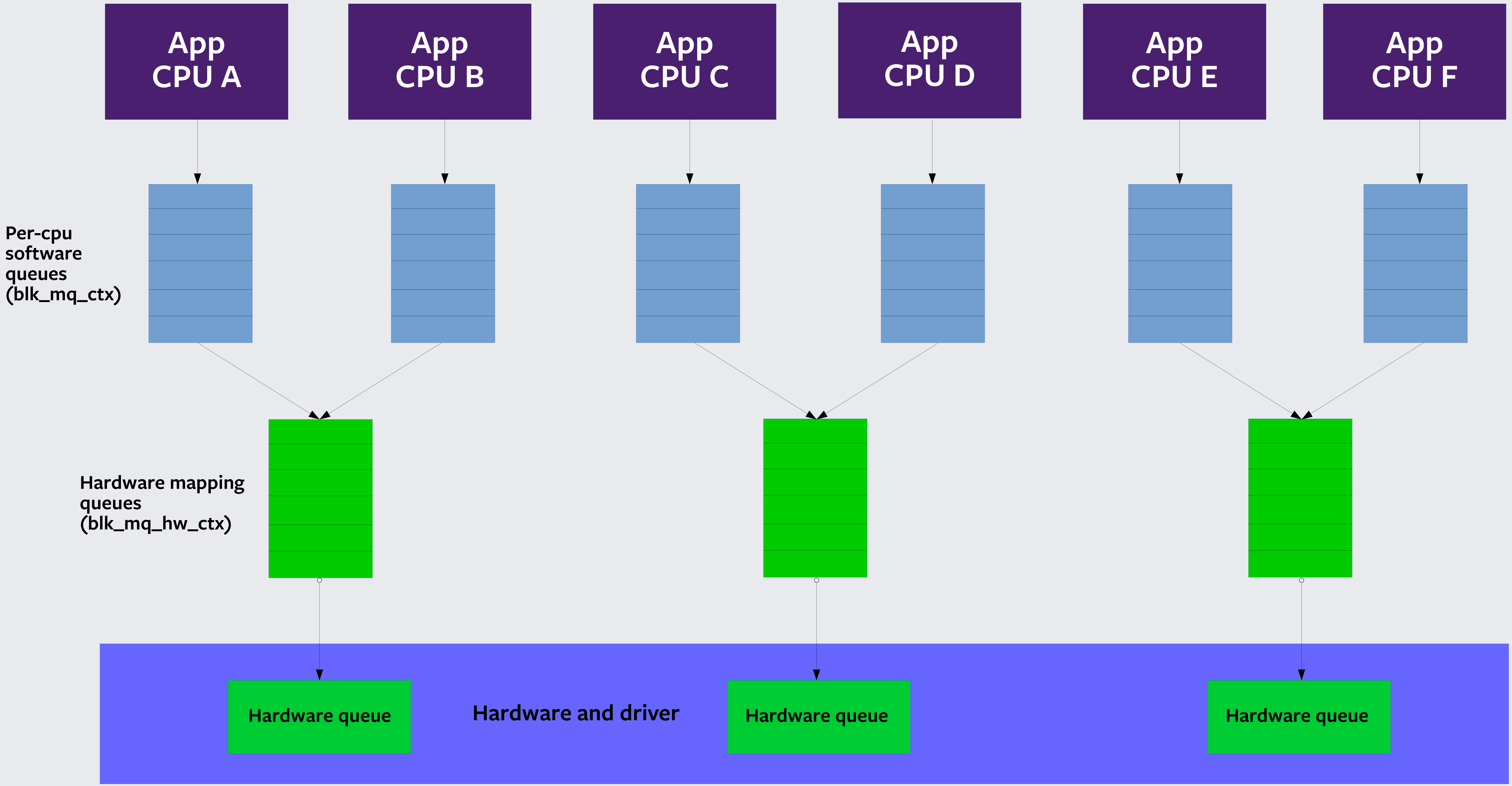
- We have good scalability until we reach the block layer
  - The shared state is a massive issue
- A bypass mode driver could work around the problem
- We need a real and future proof solution!

Enter block multiqueue

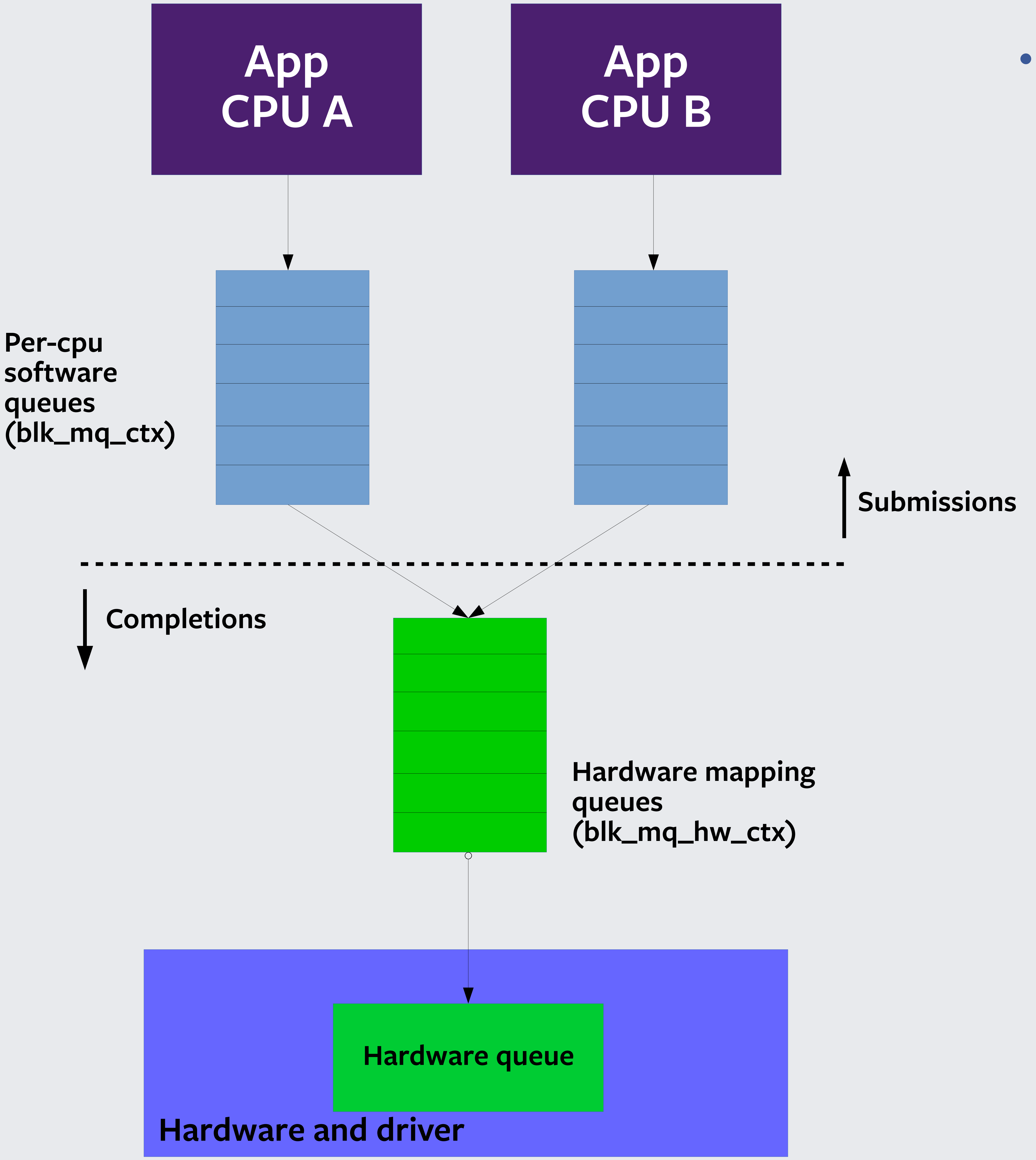
- Shares basic name with similar networking functionality, but was built from scratch
- Basic idea is to separate shared state
  - Between applications
  - Between completion and submission
- Improve scaling on non-mq hardware was a criteria
- Provide a full pool of helper functionality
  - Implement and debug once
- Become THE queuing model, not “the 3rd one”

History

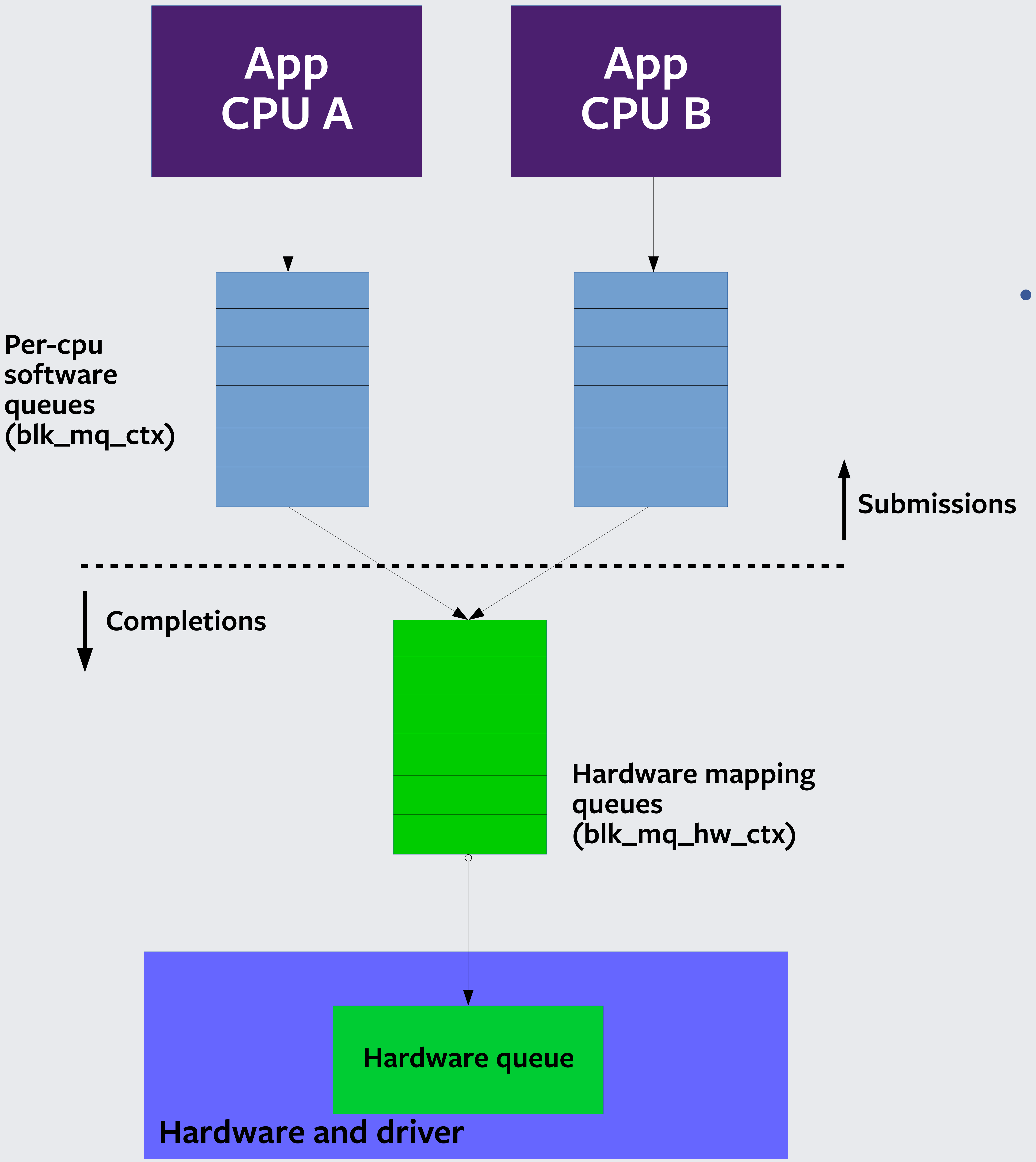
- Started in 2011
- Original design reworked, finalized around 2012
- Merged in 3.13



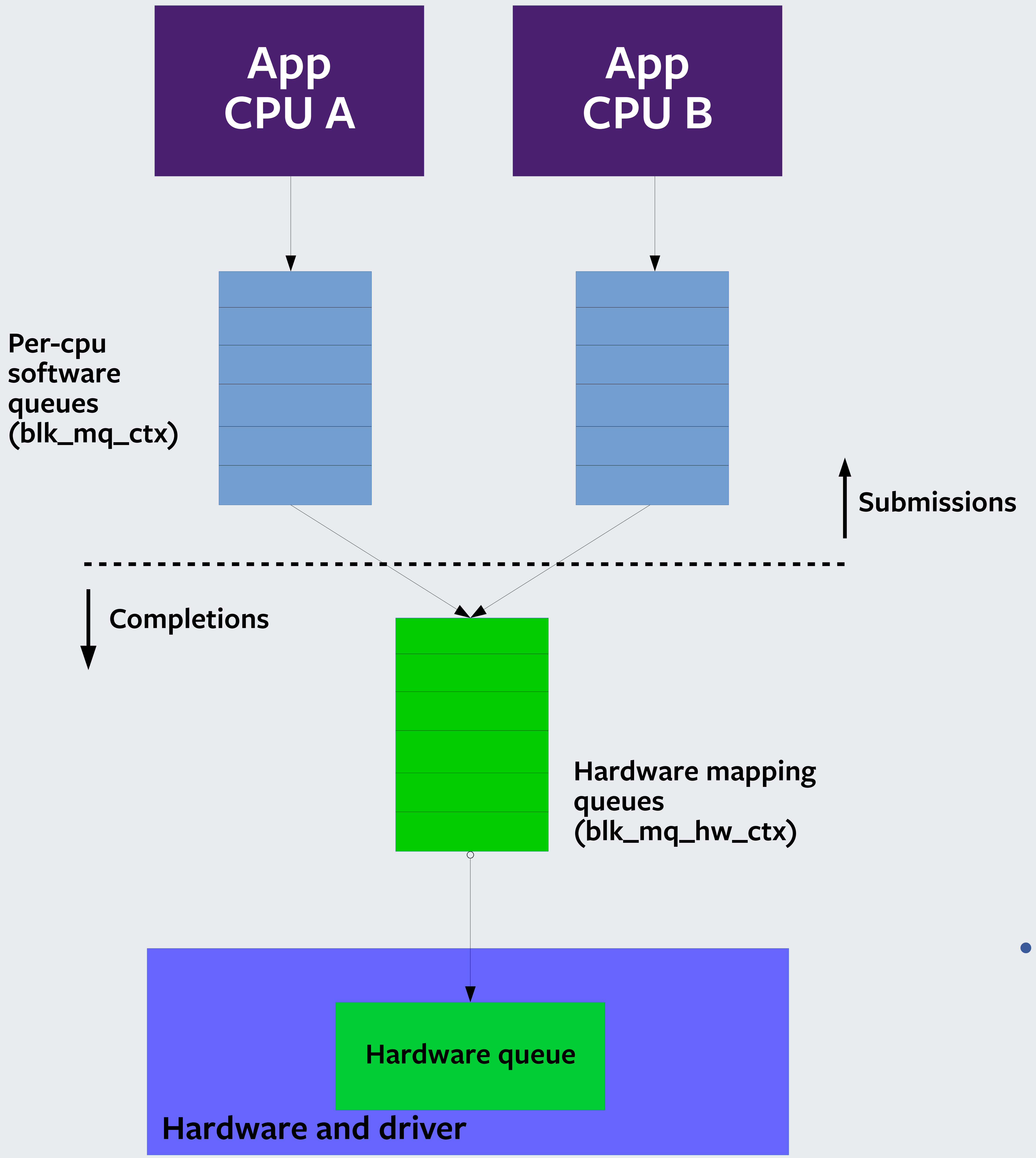
App
CPU A
App
CPU B
App
CPU C
App
CPU D
App
CPU E
App
CPU F
Per-cpu
software
queues
(blk_mq_ctx)
Hardware mapping
queues
(blk_mq_hw_ctx)
Hardware queue
Hardware and driver
Hardware queue
Hardware queue



- Application touches private per-cpu queue
  - Software queues
  - Submission is now almost fully privatized



- Software queues map M:N to hardware queues
  - There are always as many software queues as CPUs
  - With enough hardware queues, it's a 1:1 mapping
  - Fewer, and we map based on topology of the system

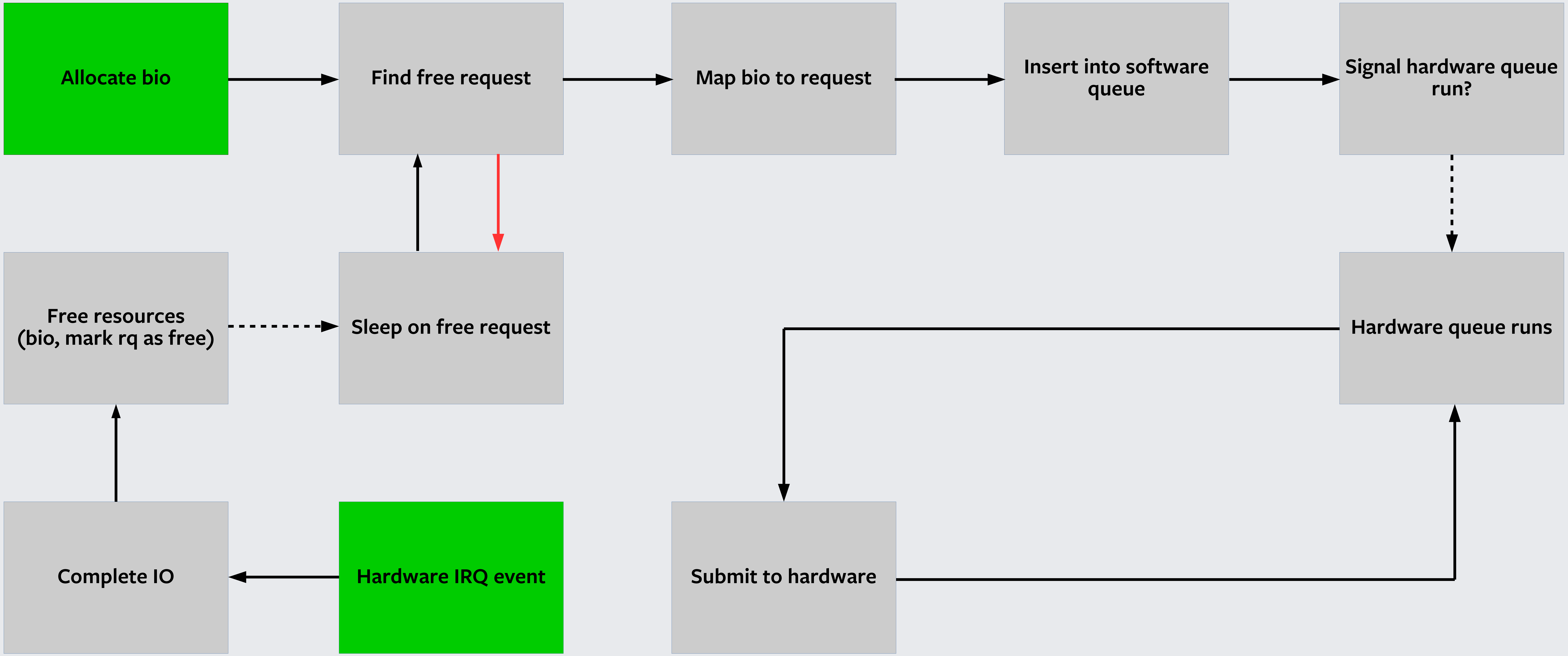


- Hardware queues handle dispatch to hardware and completions

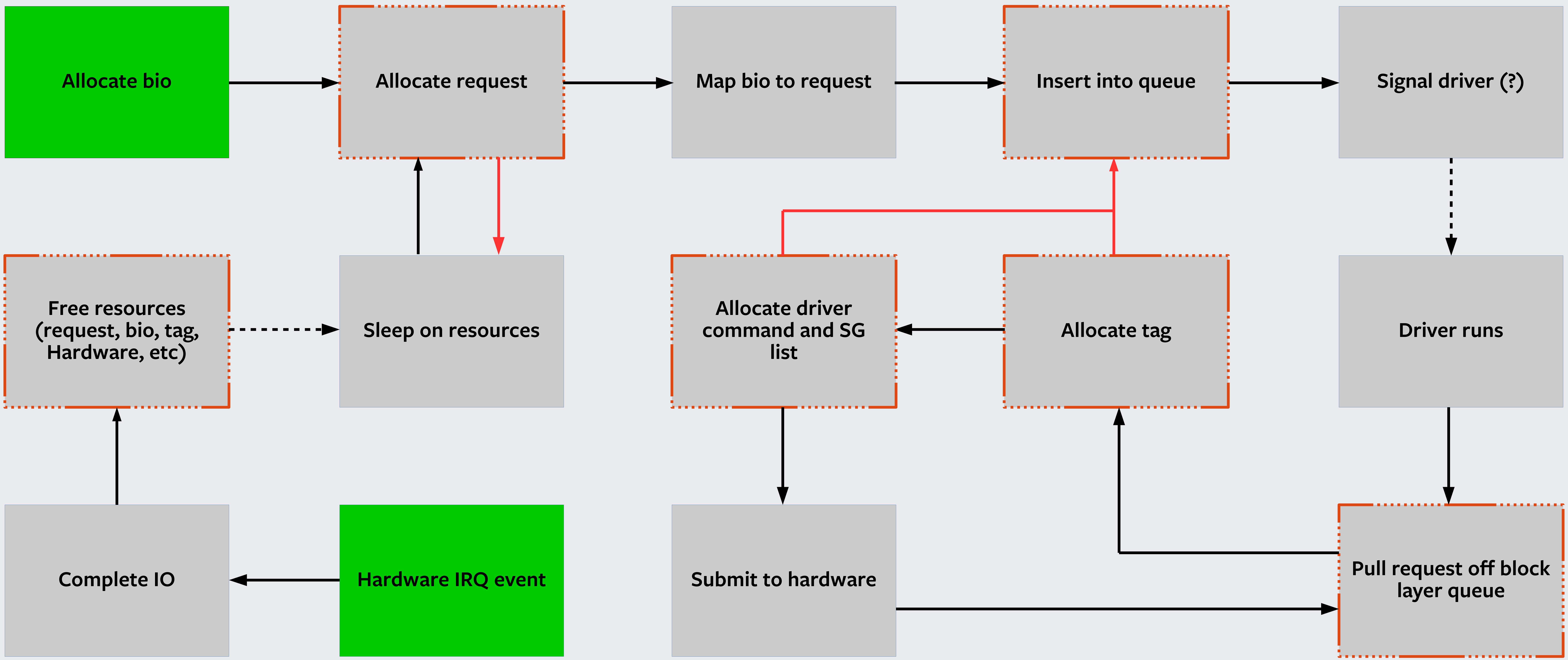
Features

- Efficient and fast versions of:
  - Tagging
  - Timeout handling
  - Allocation eliminations
  - Local completions
- Provides intelligent queue ↔ CPU mappings
  - Can be used for IRQ mappings as well
- Clean API
  - Driver conversions generally remove more code than they add

blk-mq IO flow

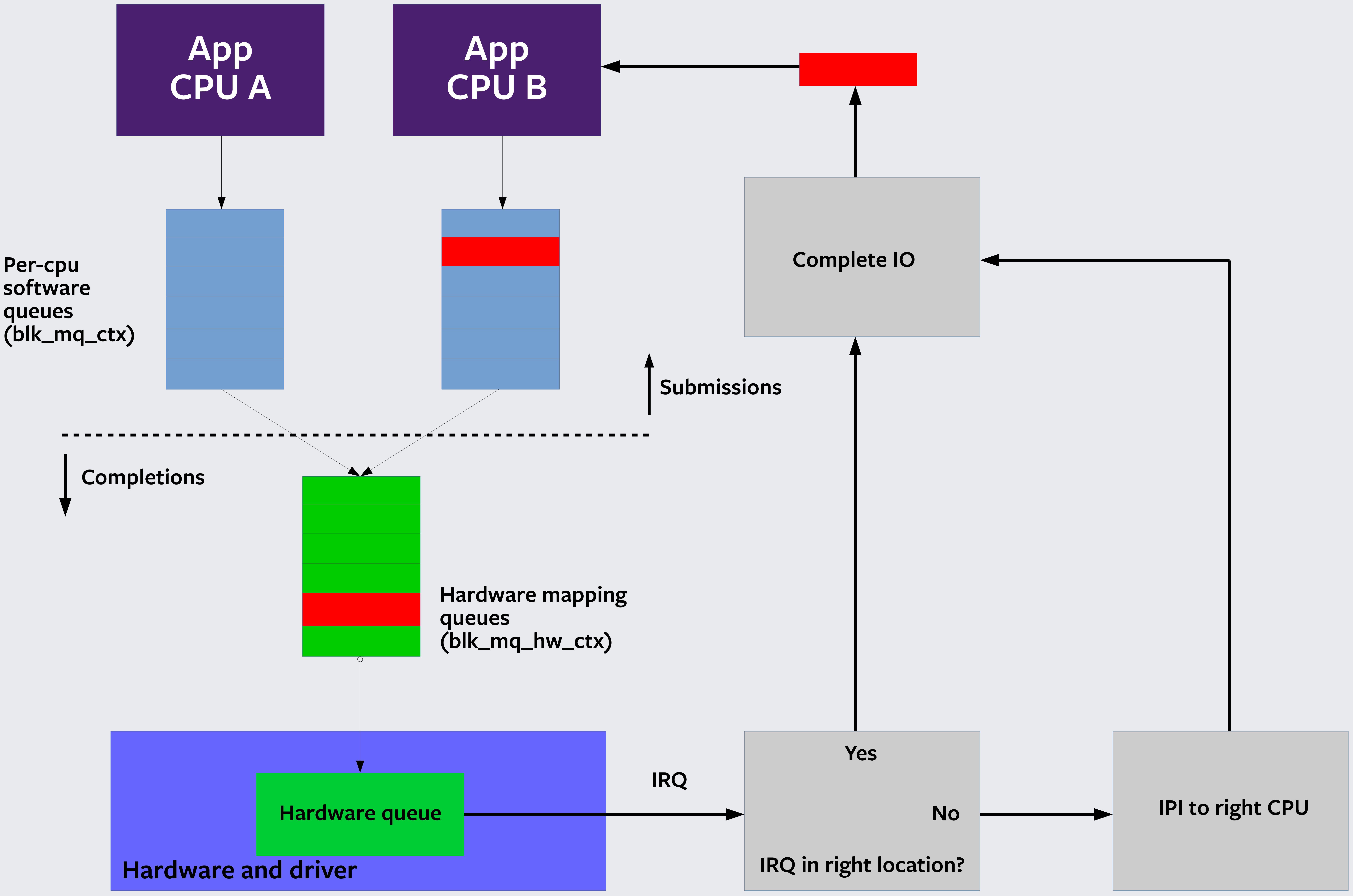


Block layer IO flow



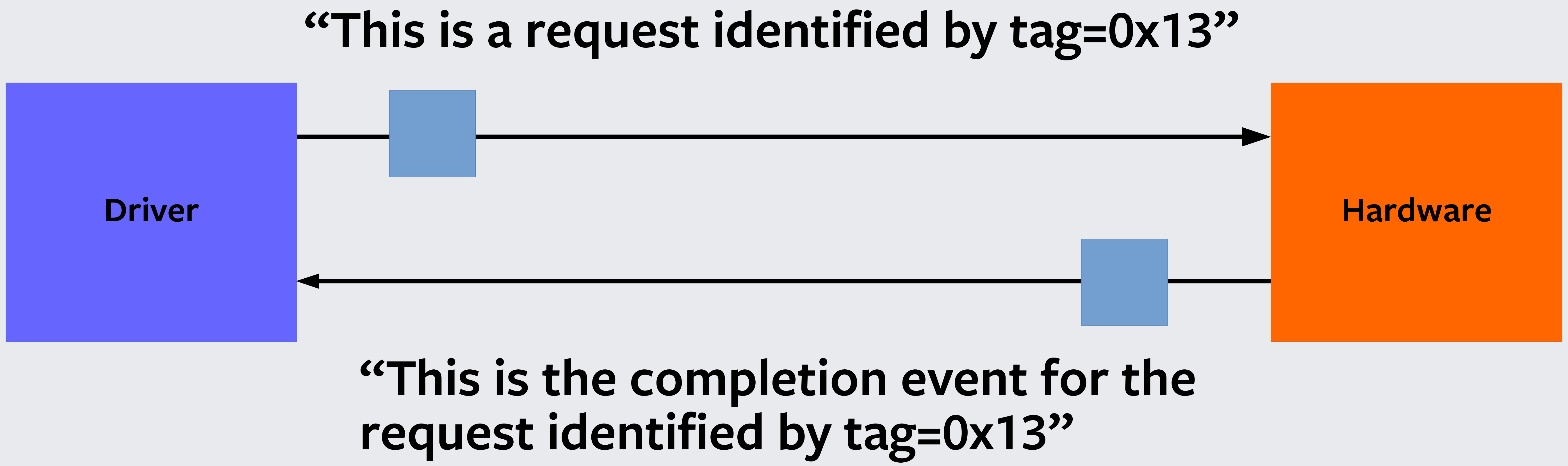
Completions

- Want completions as local as possible
  - Even without queue shared state, there's still the request
- Particularly for fewer/single hardware queue design, care must be taken to minimize sharing
- If completion queue can place event, we use that
  - If not, IPI


App
CPU A
App
CPU B
Per-cpu
software
queues
(blk_mq_ctx)
Complete IO
Submissions
Completions
Hardware mapping
queues
(blk_mq_hw_ctx)
Yes
IRQ
Hardware queue
No
IPI to right CPU
Hardware and driver
IRQ in right location?

Tagging

- Almost all hardware uses tags to identify IO requests
  - Must get a free tag on request issue
  - Must return tag to pool on completion



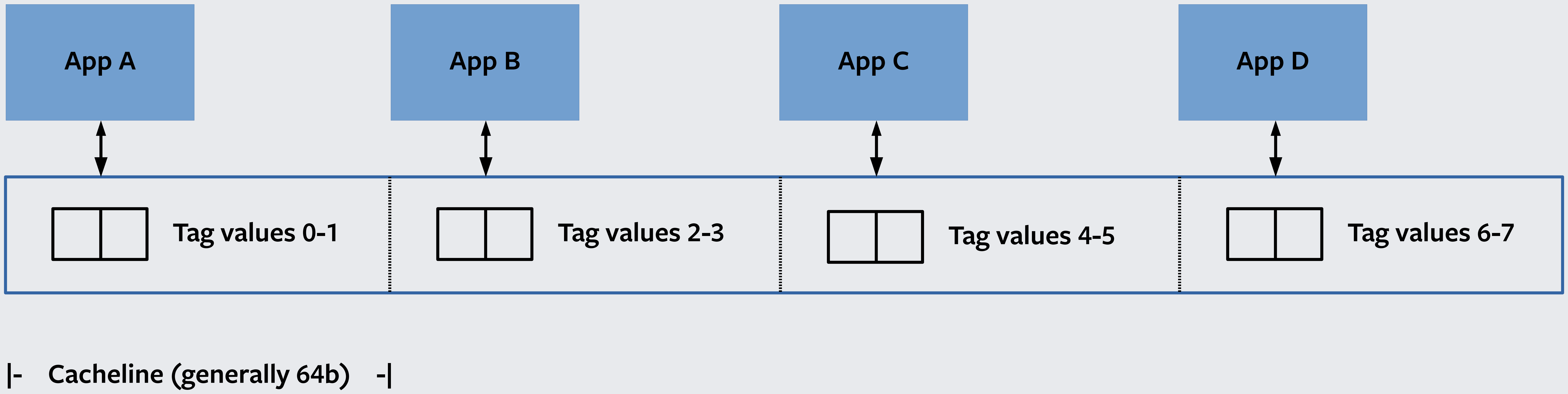
Tag support

- Must have features:
  - Efficient at or near tag exhaustion
  - Efficient for shared tag maps
- Blk-mq implements a novel bitmap tag approach
  - Software queue hinting (sticky)
  - Sparse layout
  - Rolling wakeups

Sparse tag maps

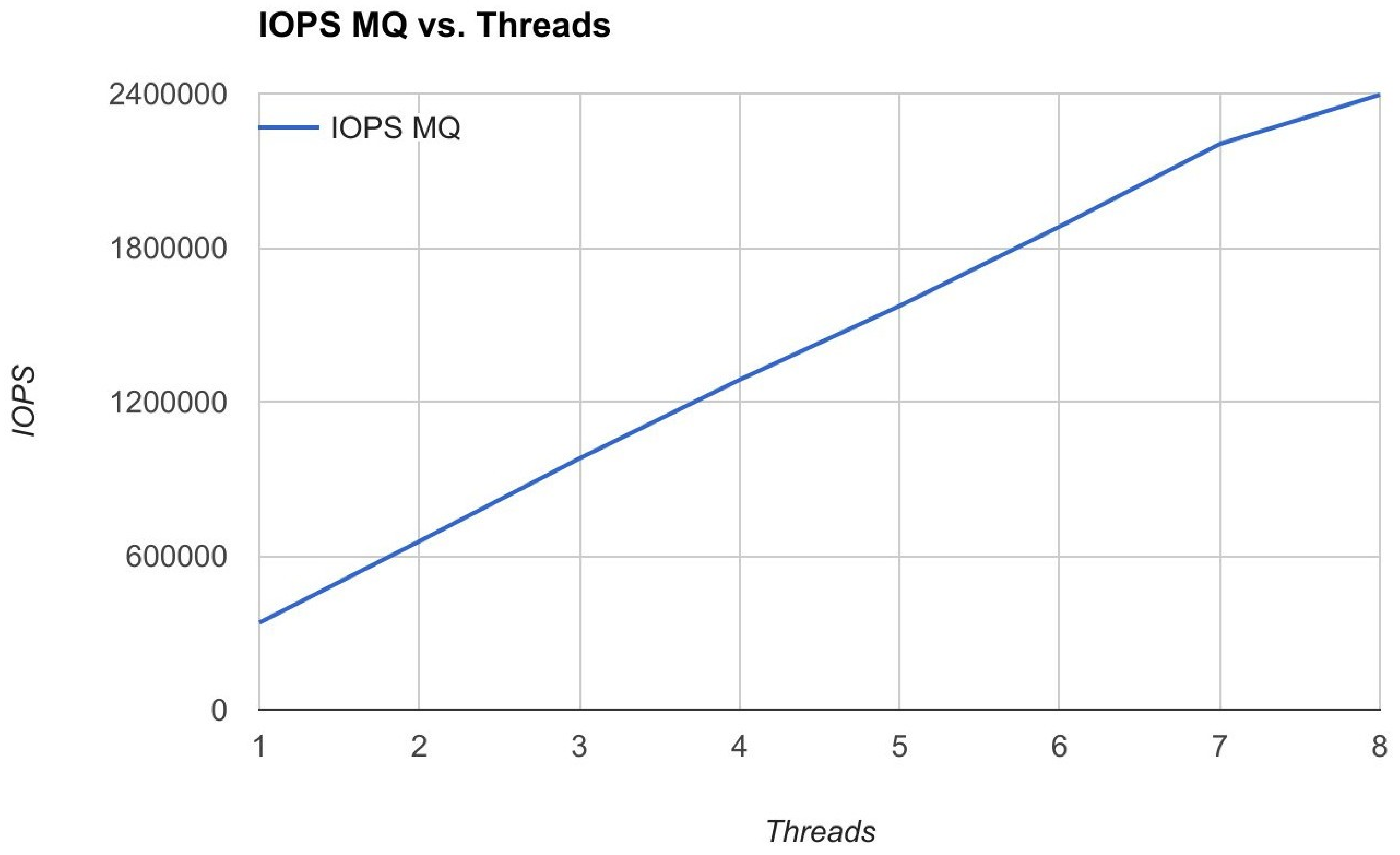
```
$ cat /sys/block/sda/mq/0/tags
nr_tags=31, reserved_tags=0, bits_per_word=2
nr_free=31, nr_reserved=0
```

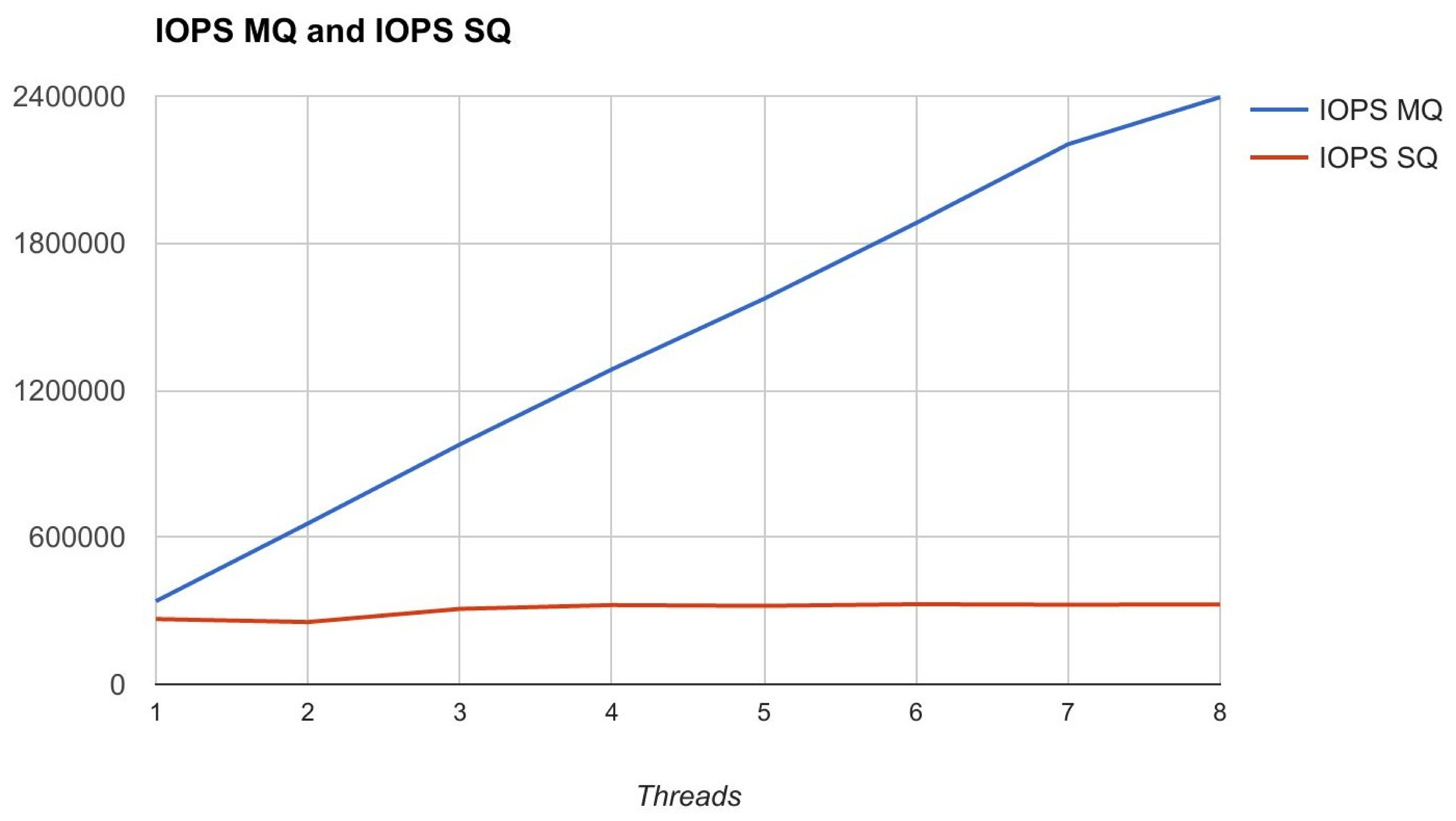
- Applications tend to stick to software queues
  - Utilize that concept to make them stick to tag cachelines
  - Cache last tag in software queue



Rerunning the test case

- We use null_blk
- Fio
  - Each thread does pread(2), 4k, randomly, O_DIRECT
- queue_mode=2 completion_nsec=0 irqmode=0 submit_queues=32
- Each added thread alternates between the two available NUMA nodes (2 socket system)


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


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

```
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
```

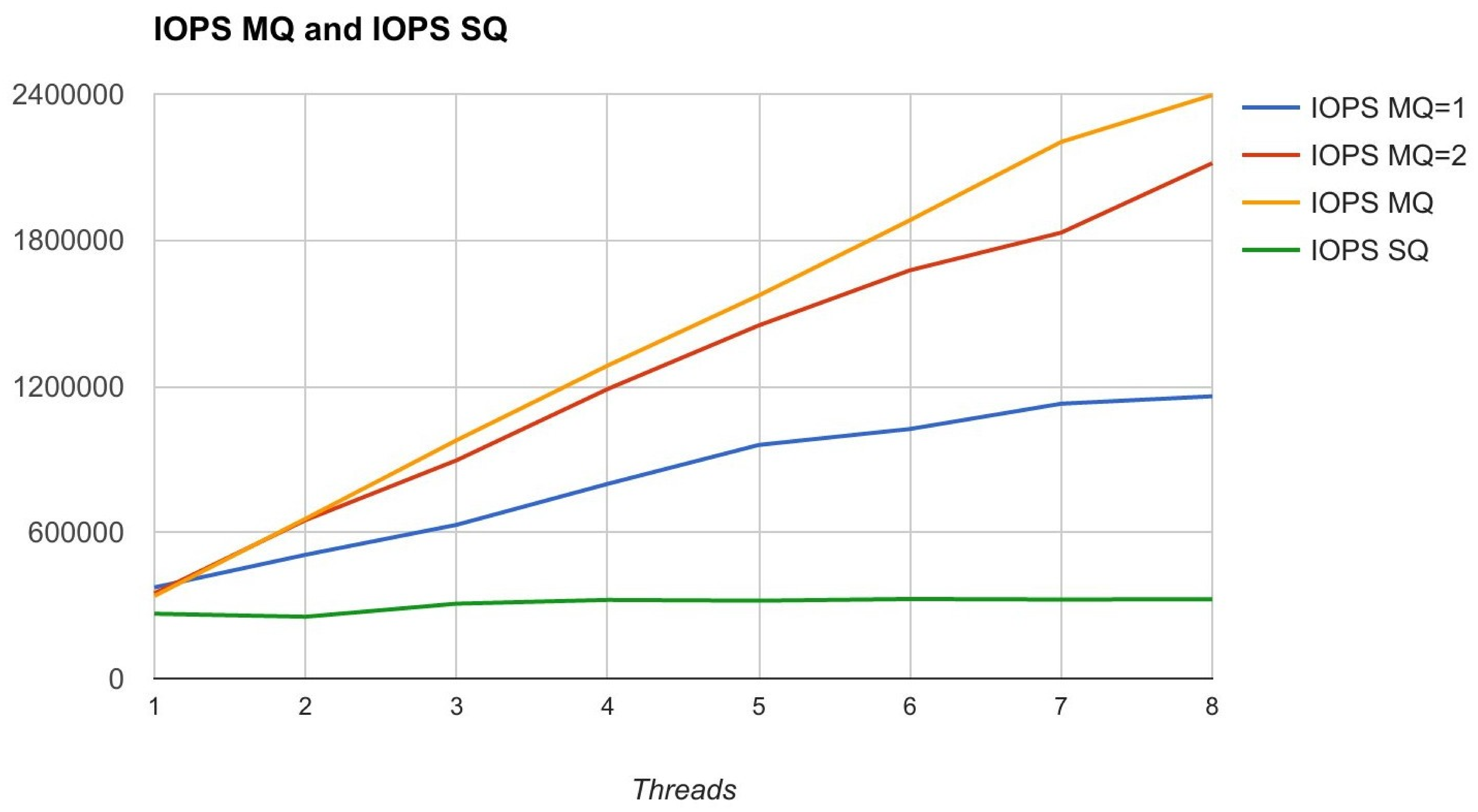
Single queue mode, basically all system time is spent banging on the device queue lock. Fio reports 95% of the time spent in the Kernel. Max completion time is 10x higher than blk-mq mode, 50th percentile is 24usec.

In blk-mq mode, locking time is drastically reduced and the profile Is much cleaner. Fio reports 74% of the time spent in the kernel. 50th percentile is 3 usec.

```
Samples: 165K of event 'cycles', Event count (approx.): 110637184263
  Overhead  Command  Shared Object        Symbol
+    7.25%  fio      [kernel.kallsyms]    [k] do_blockdev_direct_IO
+    4.39%  fio      [kernel.kallsyms]    [k] generic_make_request_checks
+    3.77%  fio      fio                  [.] get_io_u
+    3.30%  fio      [kernel.kallsyms]    [k] inode_dio_done
+    2.48%  fio      fio                  [.] __fio_gettime
+    2.36%  fio      [kernel.kallsyms]    [k] blkdev_read_iter
+    2.05%  fio      fio                  [.] thread_main
+    2.01%  fio      [kernel.kallsyms]    [k] _raw_spin_lock_irqsave
+    1.91%  fio      [kernel.kallsyms]    [k] __blk_mq_alloc_request
+    1.85%  fio      fio                  [.] io_completed
+    1.82%  fio      fio                  [.] clock_thread_fn
+    1.80%  fio      [kernel.kallsyms]    [k] blk_mq_map_queue
+    1.72%  fio      [kernel.kallsyms]    [k] bt_clear_tag
Press '?' for help on key bindings
```

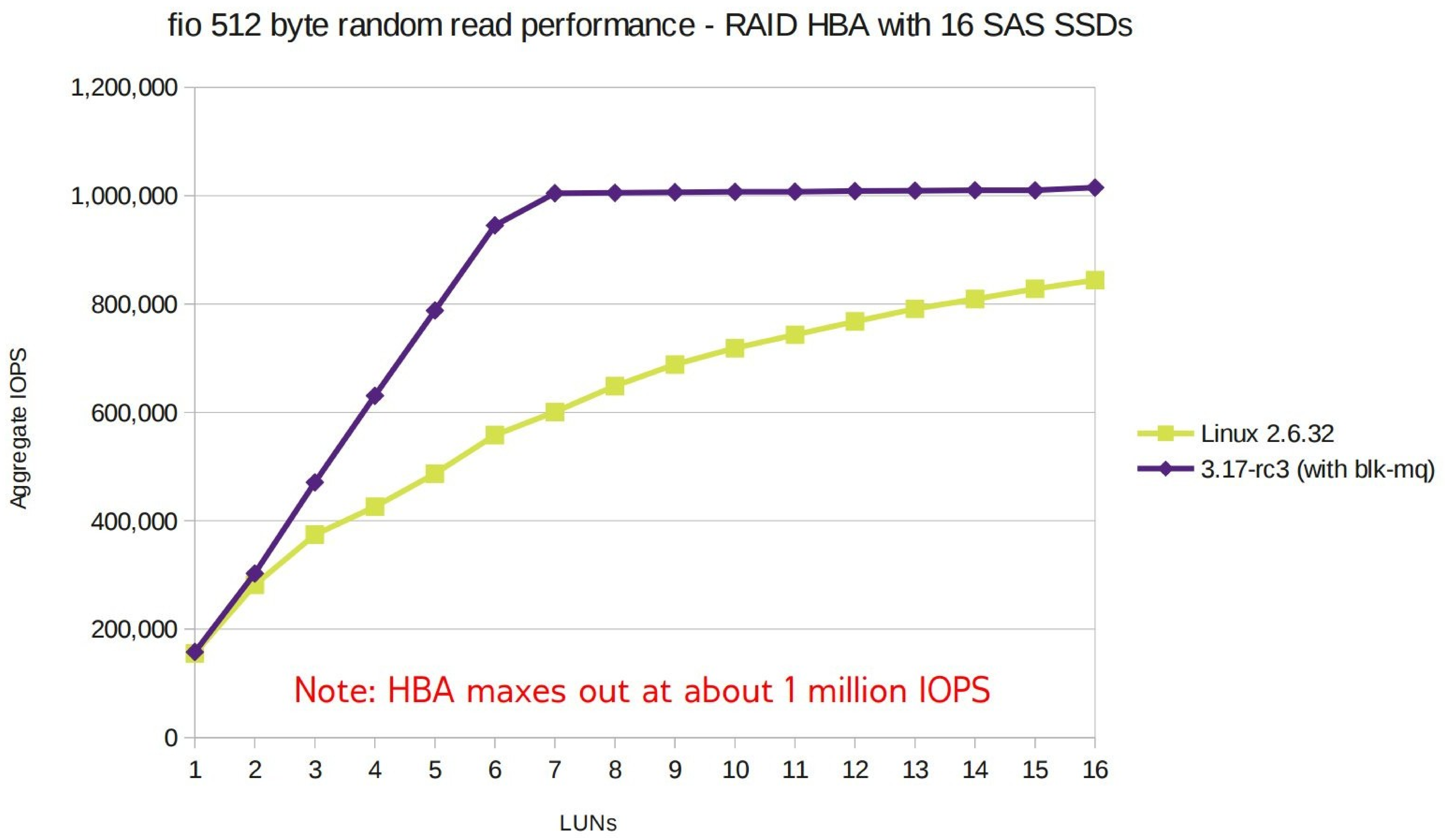
> “But Jens, isn't most storage hardware still single queue? What about single queue performance on blk-mq?”

— Astute audience member


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

Scsi-mq

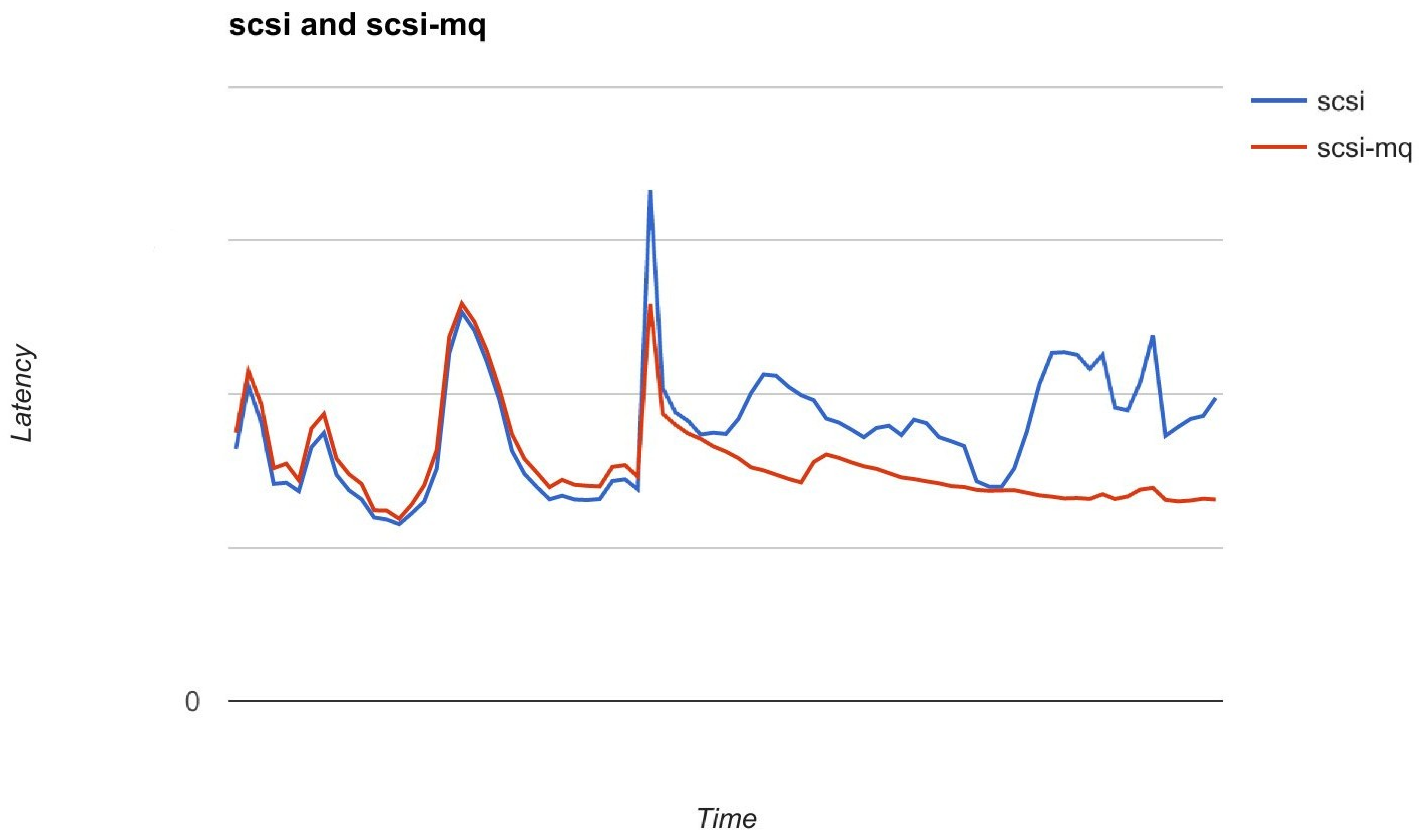
- SCSI had severe scaling issues
  - Per LUN performance limited to ~150K IOPS
- SCSI queuing layered on top of blk-mq
- Initially by Nic Bellinger (Datera), later continued by Christoph Hellwig
- Merged in 3.17
  - CONFIG_SCSI_MQ_DEFAULT=y
  - scsi_mod.use_blk_mq=1
- Helped drive some blk-mq features



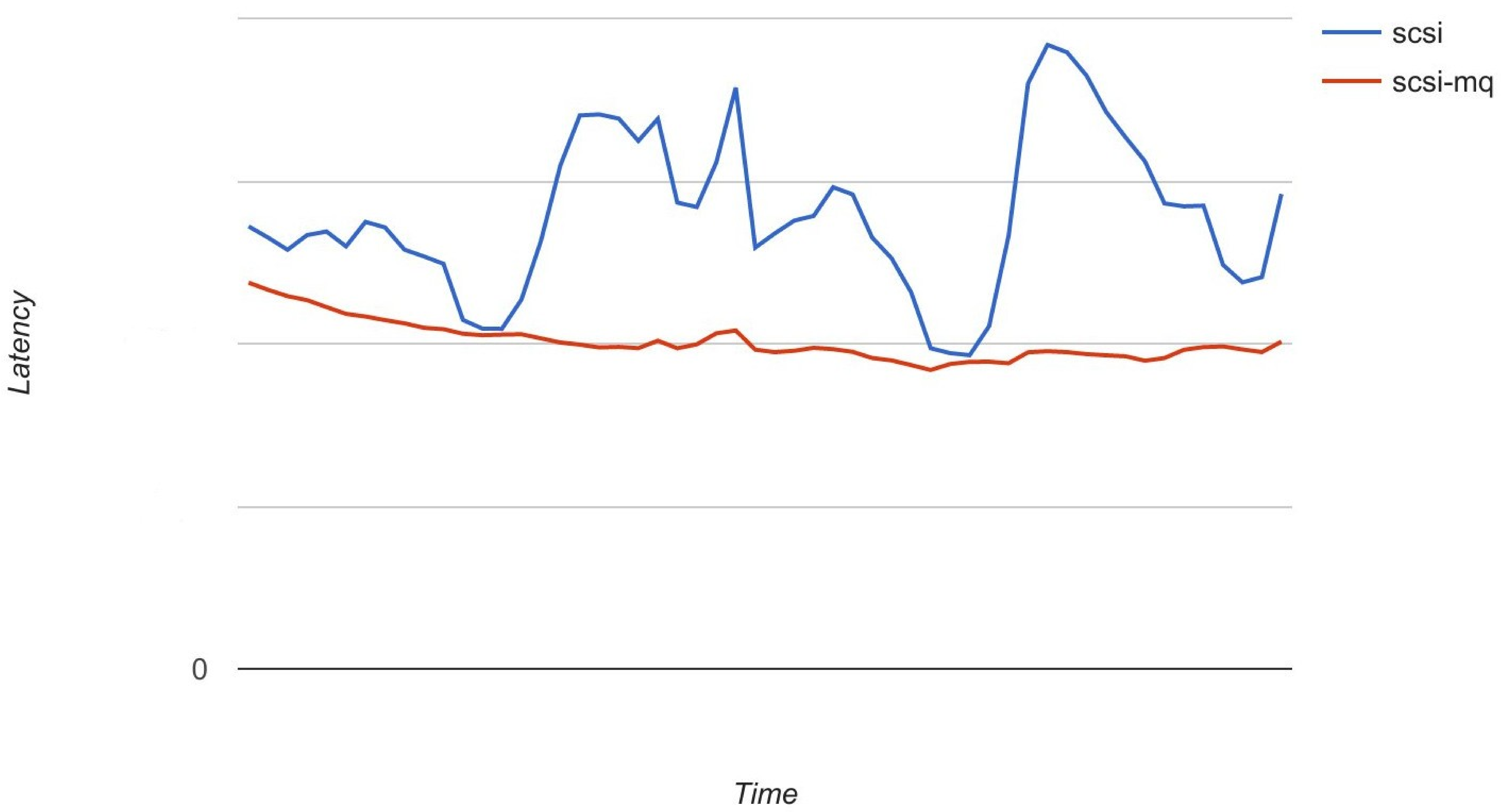
Graph from Christoph Hellwig

At Facebook

- Backport
- Ran a pilot last half, results were so good it was immediately put in production.
- Running in production at Facebook
  - TAO, cache
- Biggest win was in latency reductions
  - FB workloads not that IOPS intensive
  - But still saw sys % wins too


scsi and scsi-mq
scsi
scsi-mq
Latency
0
Time

scsi and scsi-mq



Conversion progress

- As of 4.3-rc2
  - mtip32xx (micron SSD)
  - NVMe
  - virtio_blk, xen block driver
  - rbd (ceph block)
  - loop
  - ubi
  - SCSI
- All over the map (which is good)

Future work

- An IO scheduler
- Better helpers for IRQ affinity mappings
- IO accounting
- IO polling
- More conversions
  - Long term goal remains killing off request_fn

facebook