BFQ, Multiqueue-Deadline, or Kyber? Performance Characterization of Linux Storage Schedulers in the NVMe Era

Zebin Ren[1], Krijn Doekemeijer[1], Nick Tehrany[2], and Animesh Trivedi[1]

[1]VU Amsterdam
[2]BlueOne Business Software LLC

This paper won the **best paper award** in ICPE'24.

@Large Research
Massivizing Computer Systems
https://atlarge-research.com/

Background

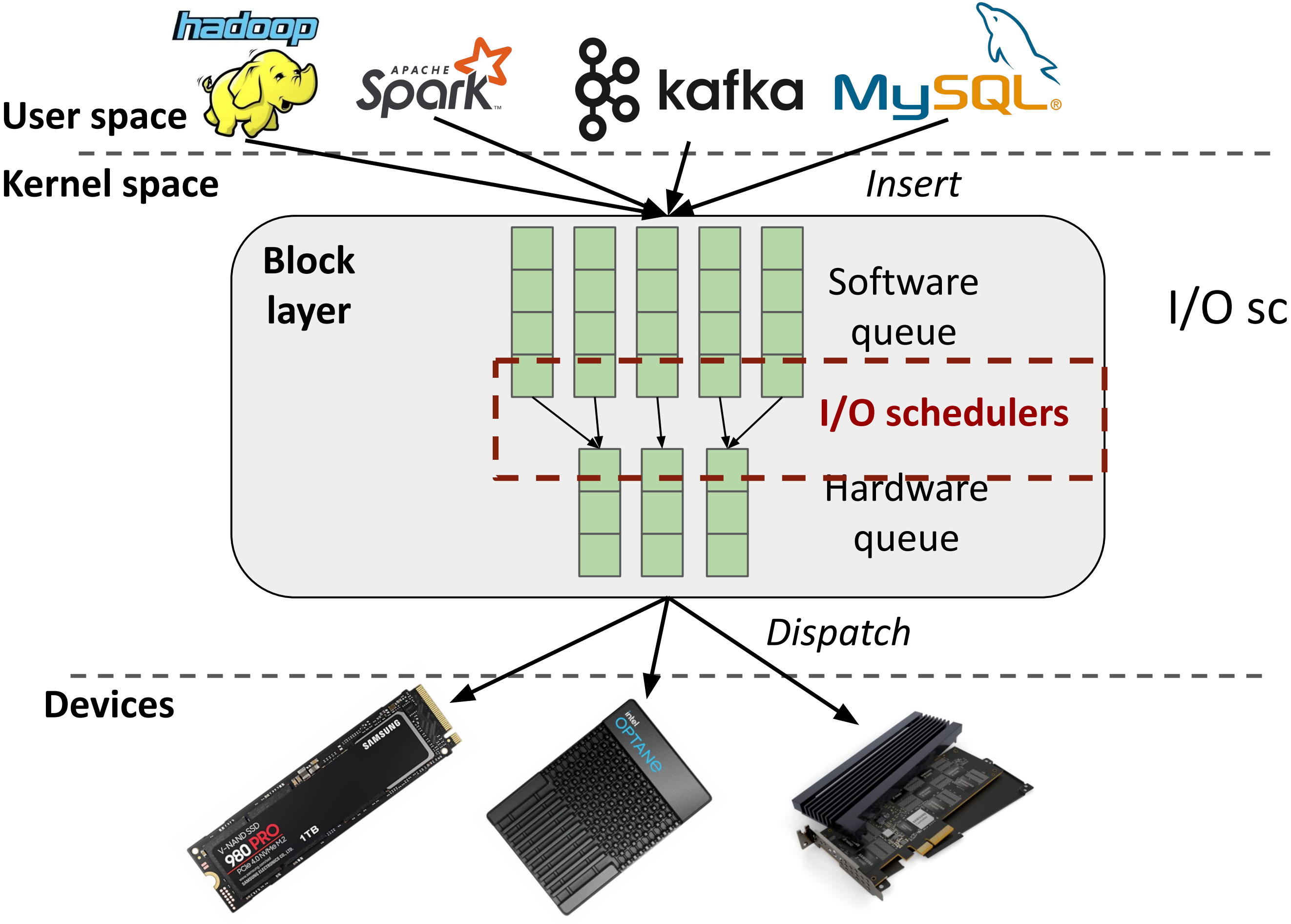
QoS guarantees

- Latency
- Throughput

Up to millions of IOPS

< 10 µs latency

Background: I/O Schedulers



User space

Kernel space

Insert

Block layer

Software queue

I/O schedulers

Hardware queue

Dispatch

Devices

I/O schedulers

- Staging
- Reordering
- Merging

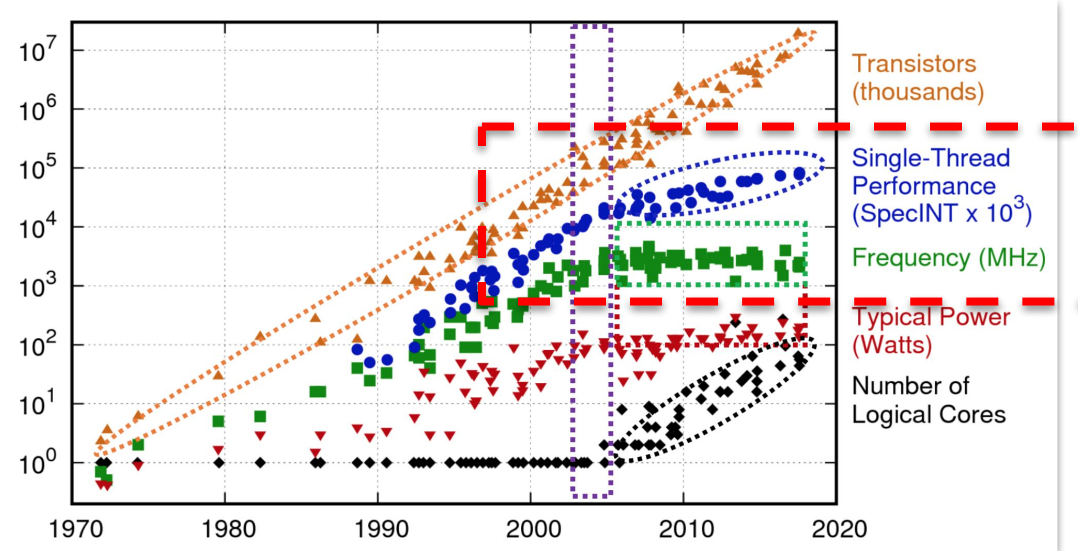
Background: What has Changed?

1. Huge improvement of storage performance.

~ 10 KIOPS →

> 1000 KIOPS

2. Improvement of CPU performance stalls.

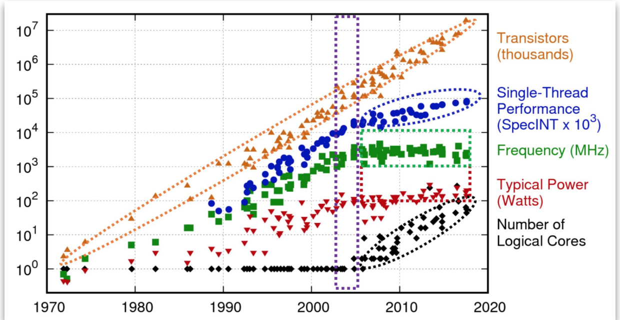


Background: What has Changed?

1. Huge improvement of storage performance.

~ 10 KIOPS →

2. Improvement of CPU performance stalls.



3. Research on I/O schedulers for these SSDs[1-4]

D2FQ: Device-Direct Fair Queueing for NVMe SSDs

Jiwon Woo, Minwoo Ahn, Gyusun Lee, Jinkyu Jeong
Sungkyunkwan University

Rearchitecting Linux Storage Stack for μs Latency and High Throughput

Jaehyun Hwang, *Cornell University*
Midhul Vuppalapati, *Cornell University*
Simon Peter, *UT Austin*
Rachit Agarwal, *Cornell University*

K2: Work-Constraining Scheduling of NVMe-Attached Storage

Till Miemietz, Hannes Weisbach
Operating Systems Group

Michael Roitzsch, Hermann Härtig
Barkhausen Institut

Multi-Queue Fair Queueing

Mohammad Hedayati, *University of Rochester*
Kai Shen, *Google*
Michael L. Scott, *University of Rochester*
Mike Marty, *Google*

Abstract

Modern high-speed devices (e.g., network adapters, storage, accelerators) use new host interfaces, which expose multiple software queues directly to the device. These *multi-queue* interfaces allow mutually distrusting applications to access the device without any cross-core interaction, enabling throughput in the order of millions of IOP/s on multicore systems. Unfortunately, while independent device access is scalable, it also introduces a new problem: unfairness. Mechanisms that were used to provide fairness for older devices are no longer tenable in the wake of multi-queue design, and straightforward attempts to re-introduce it would require cross-core synchronization that undermines the scalability for which multiple queues were designed.

To address these challenges, we present Multi-Queue Fair Queueing (MQFQ), the first fair, work-conserving scheduler suitable for multi-queue systems. Specifically, we (1) reformulate a classical fair queueing algorithm to accommodate multi-queue designs, and (2) describe a scalable implementation that bounds potential unfairness while minimizing synchronization overhead. Our implementation of MQFQ in Linux 4.15 demonstrates both fairness and high throughput. Evaluation with an NVMe over RDMA fabric (NVMf) device shows that MQFQ can reach up to 3.1 Million IOP/s on a single machine—20× higher than the state-of-the-art Linux Budget Fair Queueing. Compared to a system with no fairness, MQFQ reduces the slowdown caused by an antagonist from 3.78× to 1.33× for the FlashX workload and from 6.57× to 1.03× for the Aerospike workload (2× is considered "fair" slowdown).

1 Introduction

Recent years have seen the proliferation of very fast devices for I/O, networking, and computing acceleration. Commodity solid-state disks (e.g., Intel Optane DC P4800X [22] or Samsung PM1725a [38]) can perform at or near a million I/O operations per second. System-area networks (e.g., InfiniBand) can sustain several million remote operations per second over a single link [25]. RDMA delivers data across fabric within a few microseconds. GPUs and machine learning accelerators may offload computations that run just a few microseconds at a time [30]. At the same time, the proliferation of multicore processors has necessitated architectures tuned for independent I/O across multiple hardware threads [4, 36].

These technological changes have shifted performance bottlenecks from hardware resources to the software stacks that manage them. In response, it is now common to adopt a *multi-queue* architecture in which each hardware thread owns a dedicated I/O queue, directly exposed to the device, giving it an independent path over which to send and receive requests. Examples of this architecture include multi-queue SSDs [22, 38, 50] and NICs [42], and software like the Windows and Linux NVMe drivers, the Linux multi-queue block layer [5], SCSI multi-queue support [8], and data-plane OSes [4, 36]. A recent study [51] demonstrated up to 8× performance improvement for YCSB-on-Cassandra, using multi-queue NVMe instead of single-queue SATA.

To support the full bandwidth of modern devices, multi-queue I/O systems are designed to incur no cache-coherence traffic in the common case when sending and receiving requests. It's easy to see why: a device supporting millions of IOP/s sees each new request in a fraction of a microsecond—a time interval that allows for fewer than 10 cross-core cache coherence misses, and is comparable to the latency of a single inter-processor interrupt (IPI). Serializing requests at such high speeds is infeasible now, and will only become more so as device speeds continue to increase while single-core performance stays relatively flat. As a result, designers have concluded that conventional fair-share I/O schedulers, including fair queueing approaches [35, 40], which reorder requests in a single queue, are unsuited for modern fast devices.

Unfortunately, by cutting out the OS resource scheduler, direct multi-queue device access undermines the OS's traditional responsibility for fairness and performance isolation. While I/O devices (e.g., SSD firmware, NICs) may multiplex hardware queues, their support for fairness is hampered by their inability to reason in terms of system-level policies for resource principals (applications, virtual machines, or Linux

USENIX Association   2019 USENIX Annual Technical Conference 301

The Linux I/O Schedulers

D2FQ: Device-Direct Fair Queueing for NVMe SSDs

Jiwon Woo, Minwoo Ahn, Gyusun Lee, Jinkyu Jeong
Sungkyunkwan University

Rearchitecting Linux Storage Stack for μs Latency and High Throughput

Jaehyun Hwang *Cornell University*
Midhul Vuppalapati *Cornell University*
Simon Peter *UT Austin*
Rachit Agarwal *Cornell University*

K2: Work-Constraining Scheduling of NVMe-Attached Storage

Till Miemietz, Hannes Weisbach
Operating Systems Group

Michael Roitzsch, Hermann Härtig
Barkhausen Institut

Multi-Queue Fair Queueing

Mohammad Hedayati *University of Rochester*
Kai Shen *Google*
Michael L. Scott *University of Rochester*
Mike Marty *Google*

Abstract

Modern high-speed devices (e.g., network adapters, storage, accelerators) use new host interfaces, which expose multiple software queues directly to the device. These *multi-queue* interfaces allow mutually distrusting applications to access the device without any cross-core interaction, enabling throughput in the order of millions of IOP/s on multicore systems. Unfortunately, while independent device access is scalable, it also introduces a new problem: unfairness. Mechanisms that were used to provide fairness for older devices are no longer tenable in the wake of multi-queue design, and straightforward attempts to re-introduce it would require cross-core synchronization that undermines the scalability for which multiple queues were designed.

To address these challenges, we present Multi-Queue Fair Queueing (MQFQ), the first fair, work-conserving scheduler suitable for multi-queue systems. Specifically, we (1) reformulate a classical fair queueing algorithm to accommodate multi-queue designs, and (2) describe a scalable implementation that bounds potential unfairness while minimizing synchronization overhead. Our implementation of MQFQ in Linux 4.15 demonstrates both fairness and high throughput. Evaluation with an NVMe over RDMA fabric (NVMf) device shows that MQFQ can reach up to 3.1 Million IOP/s on a single machine—20× higher than the state-of-the-art Linux Budget Fair Queueing. Compared to a system with no fairness, MQFQ reduces the slowdown caused by an antagonist from 3.78× to 1.33× for the FlashX workload and from 6.57× to 1.03× for the Aerospike workload (2× is considered "fair" slowdown).

1 Introduction

Recent years have seen the proliferation of very fast devices for I/O, networking, and computing acceleration. Commodity solid-state disks (e.g., Intel Optane DC P4800X [22] or Samsung PM1725a [38]) can perform at or near a million I/O operations per second. System-area networks (e.g., InfiniBand) can sustain several million remote operations per second over a single link [25]. RDMA delivers data across fabric within a few microseconds. GPUs and machine learning accelerators may offload computations that run just a few microseconds at a time [30]. At the same time, the proliferation of multicore processors has necessitated architectures tuned for independent I/O across multiple hardware threads [4, 36].

These technological changes have shifted performance bottlenecks from hardware resources to the software stacks that manage them. In response, it is now common to adopt a *multi-queue* architecture in which each hardware thread owns a dedicated I/O queue, directly exposed to the device, giving it an independent path over which to send and receive requests. Examples of this architecture include multi-queue SSDs [22, 38, 50] and NICs [42], and software like the Windows and Linux NVMe drivers, the Linux multi-queue block layer [5], SCSI multi-queue support [8], and data-plane OSes [4, 36]. A recent study [51] demonstrated up to 8× performance improvement for YCSB-on-Cassandra, using multi-queue NVMe instead of single-queue SATA.

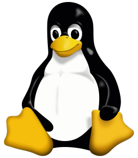
To support the full bandwidth of modern devices, multi-queue I/O systems are designed to incur no cache-coherence traffic in the common case when sending and receiving requests. It's easy to see why: a device supporting millions of IOP/s sees each new request in a fraction of a microsecond—a time interval that allows for fewer than 10 cross-core cache coherence misses, and is comparable to the latency of a single inter-processor interrupt (IPI). Serializing requests at such high speeds is infeasible now, and will only become more so as device speeds continue to increase while single-core performance stays relatively flat. As a result, designers have concluded that conventional fair-share I/O schedulers, including fair queueing approaches [35, 40], which reorder requests in a single queue, are unsuited for modern fast devices.

Unfortunately, by cutting out the OS resource scheduler, direct multi-queue device access undermines the OS's traditional responsibility for fairness and performance isolation. While I/O devices (e.g., SSD firmware, NICs) may multiplex hardware queues, their support for fairness is hampered by their inability to reason in terms of system-level policies for resource principals (applications, virtual machines, or Linux

USENIX Association 2019 USENIX Annual Technical Conference 301

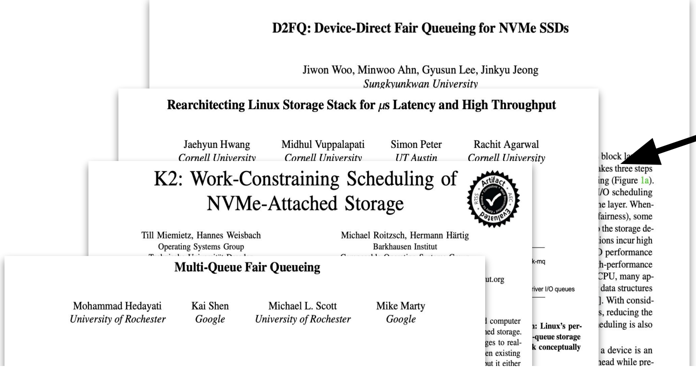
No plug-and-play implementations.

The most available I/O schedulers?



Linux I/O schedulers!

The Linux I/O Schedulers



D2FQ: Device-Direct Fair Queueing for NVMe SSDs

Jiwon Woo, Minwoo Ahn, Gyusun Lee, Jinkyu Jeong
Sungkyunkwan University

Rearchitecting Linux Storage Stack for µs Latency and High Throughput

Jaehyun Hwang, *Cornell University*
Midhul Vuppalapati, *Cornell University*
Simon Peter, *UT Austin*
Rachit Agarwal, *Cornell University*

K2: Work-Constraining Scheduling of NVMe-Attached Storage

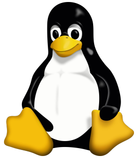
Till Miemietz, Hannes Weisbach
Operating Systems Group
Michael Roitzsch, Hermann Härtig
Barkhausen Institut

Multi-Queue Fair Queueing

Mohammad Hedayati, *University of Rochester*
Kai Shen, *Google*
Michael L. Scott, *University of Rochester*
Mike Marty, *Google*

No plug-and-play implementations.

The most available I/O schedulers?



Linux I/O schedulers!

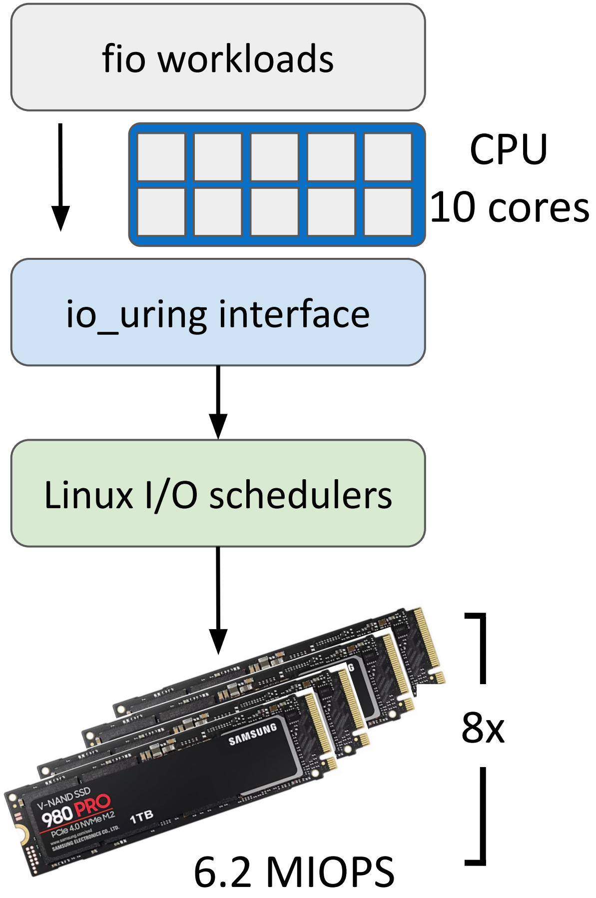
None
- Least overhead.
- No performance guarantees.

BFQ
- Fair-share between apps.
- Complex, high overhead.

Kyber
- Designed for fast SSDs.
- Balancing between read and write.

MQ-Deadline
- Issues request with increasing sector order.
- Soft latency deadlines.

Setup



fio workloads

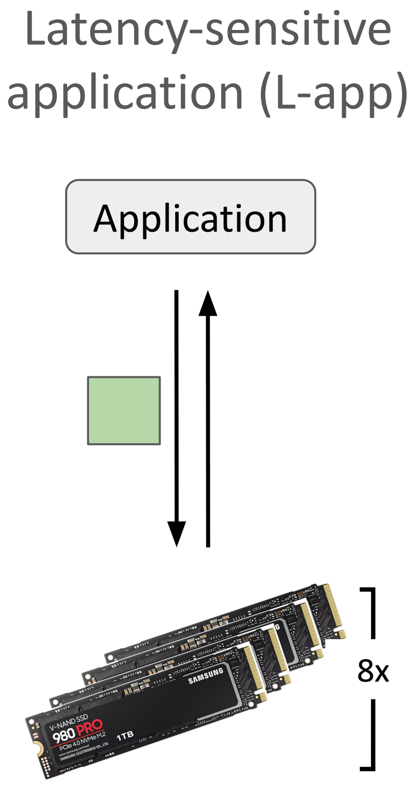
CPU
10 cores

io_uring interface

Linux I/O schedulers

8x

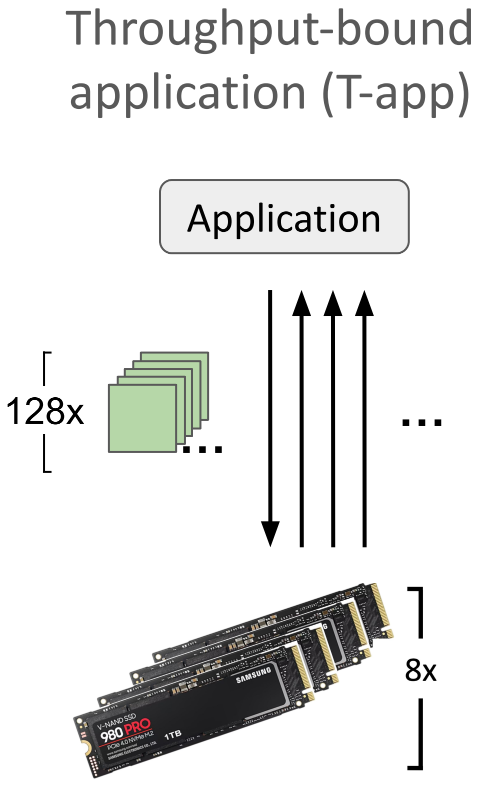
6.2 MIOPS



Latency-sensitive application (L-app)

Application

8x



Throughput-bound application (T-app)

Application

128x

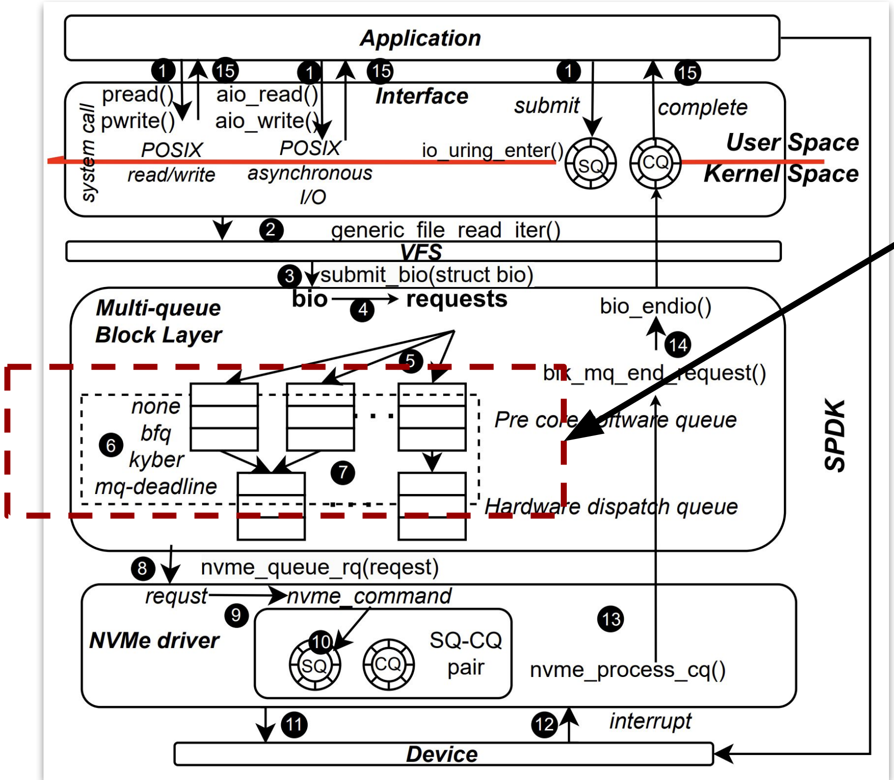
...

...

8x

Research Questions

RQ1: Overhead



I/O schedulers

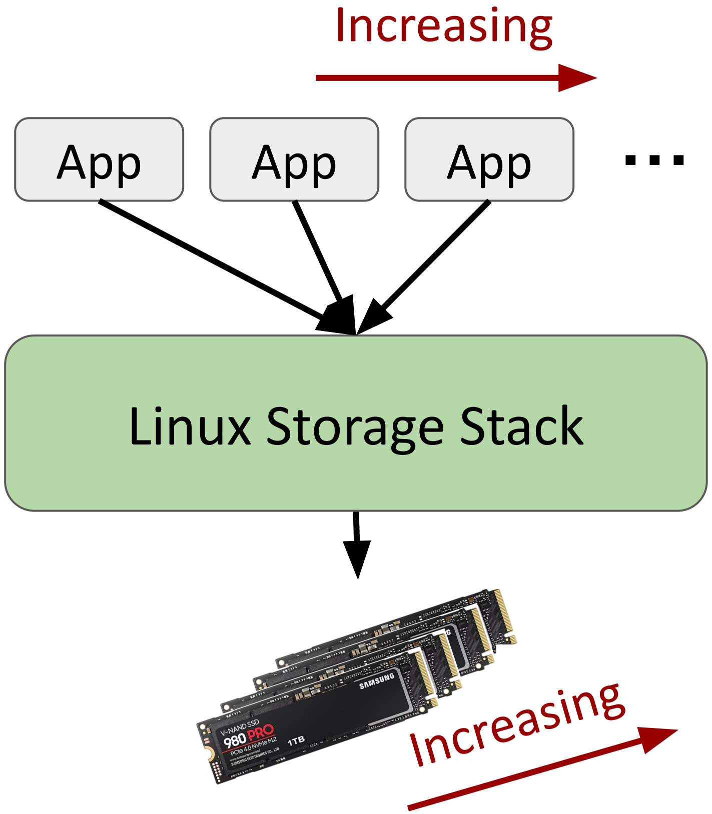
- Latency?
- Throughput?

Research Questions

RQ1: Overhead

- Latency?
- Throughput?

RQ2: Scalability



Increasing

App App App …

Linux Storage Stack

Increasing

Research Questions

RQ1: Overhead

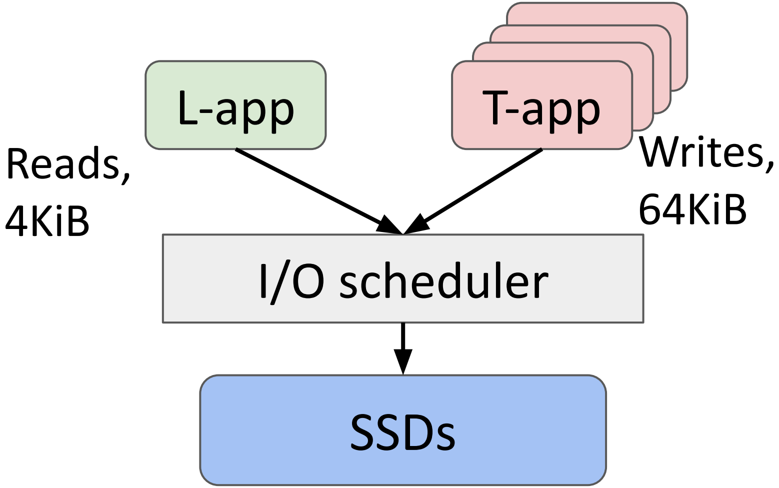
- Latency?
- Throughput?

RQ2: Scalability

- L-apps.
- T-apps.
- SSDs.

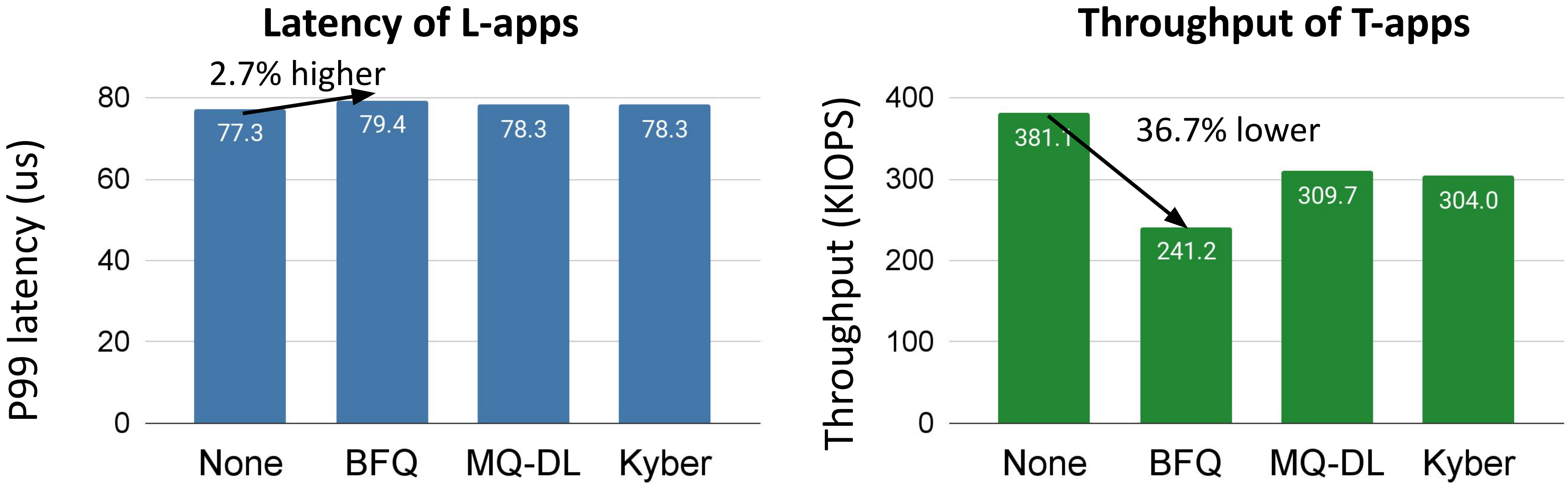
RQ3: Interference

- Read L-app + increasing write T-apps.



RQ1. Overheads

Overhead



Slightly higher latency, up to 2.7% higher latency.

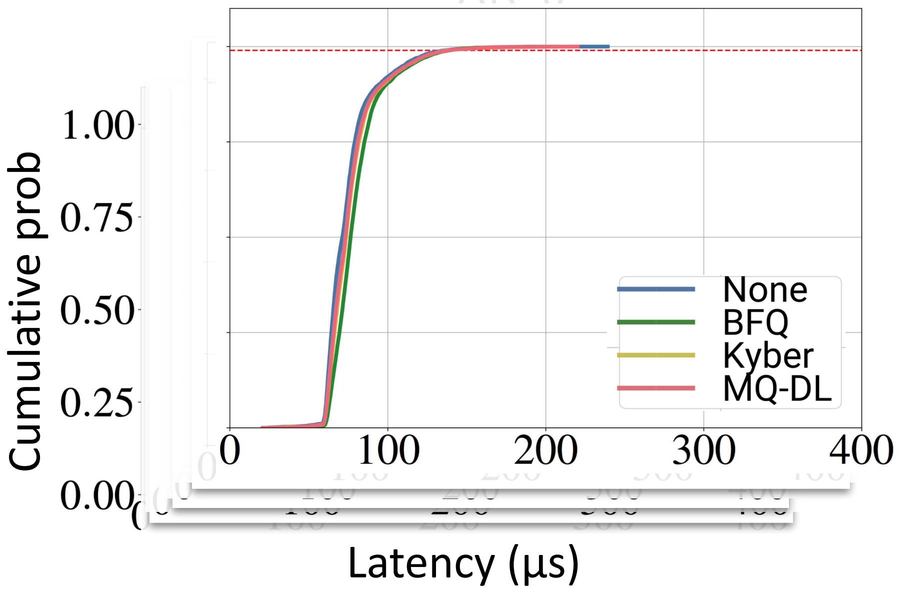
Significantly lower throughput, up to 36.7% lower.

I/O schedulers → Significantly higher throughput overhead.

RQ2. Scalability

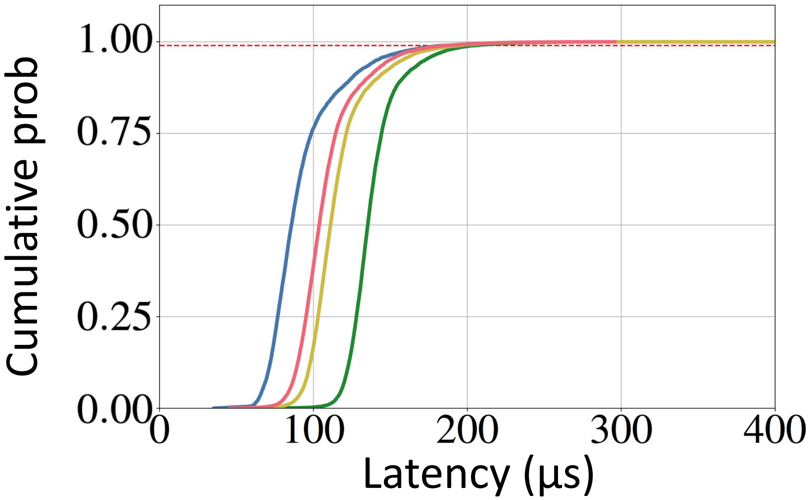
Scalability of L-apps

QD=16



Better

QD=32

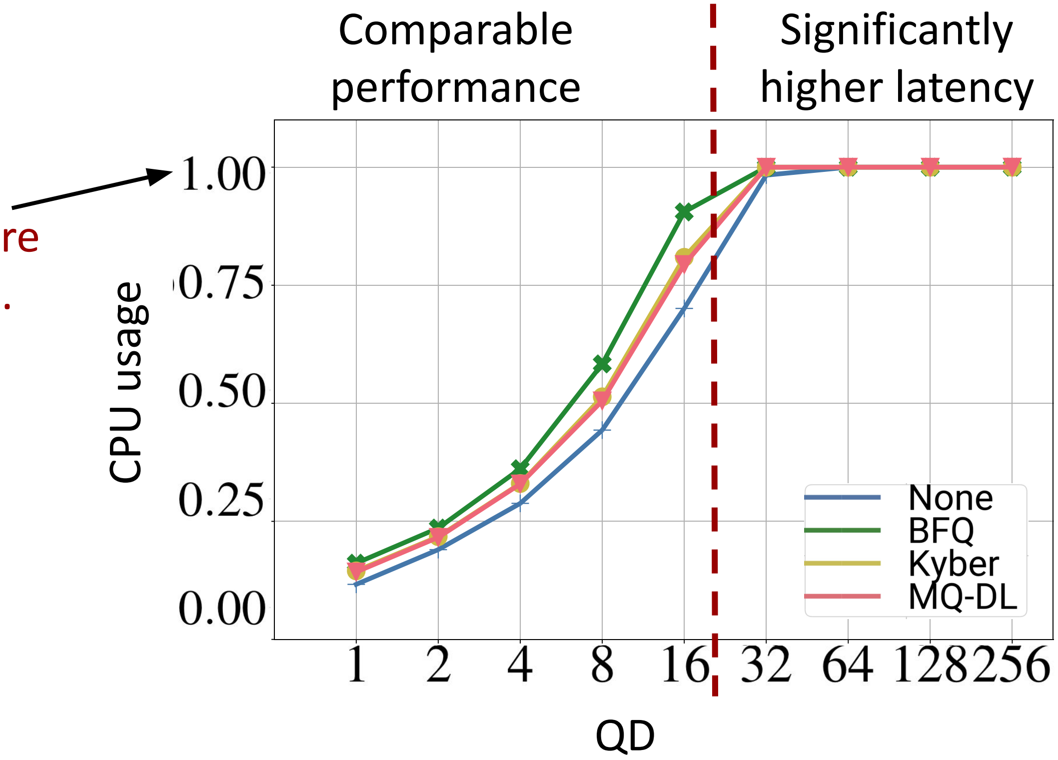


Better

Higher workload →higher overhead.

Why?

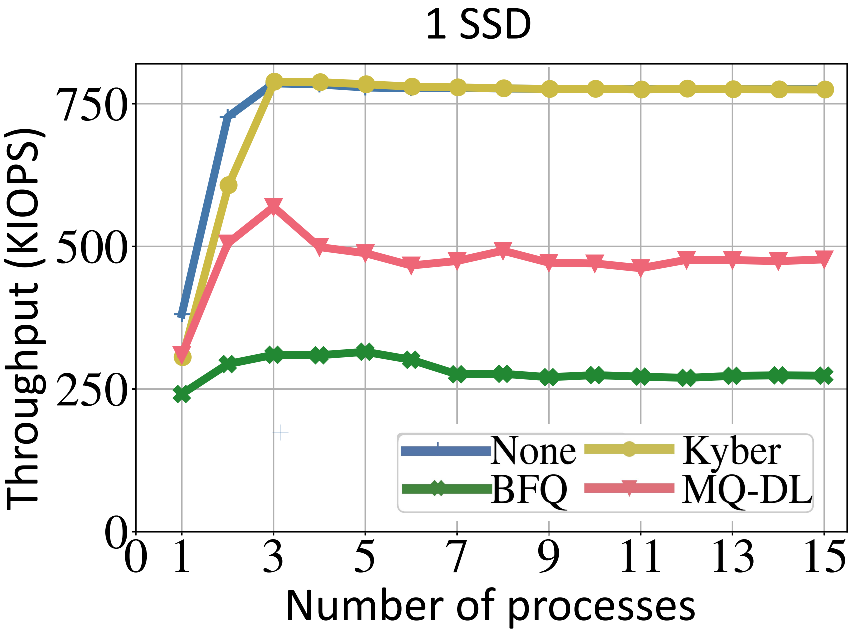
Scalability of L-apps: CPU Usage



A single CPU core is 100% utilized.

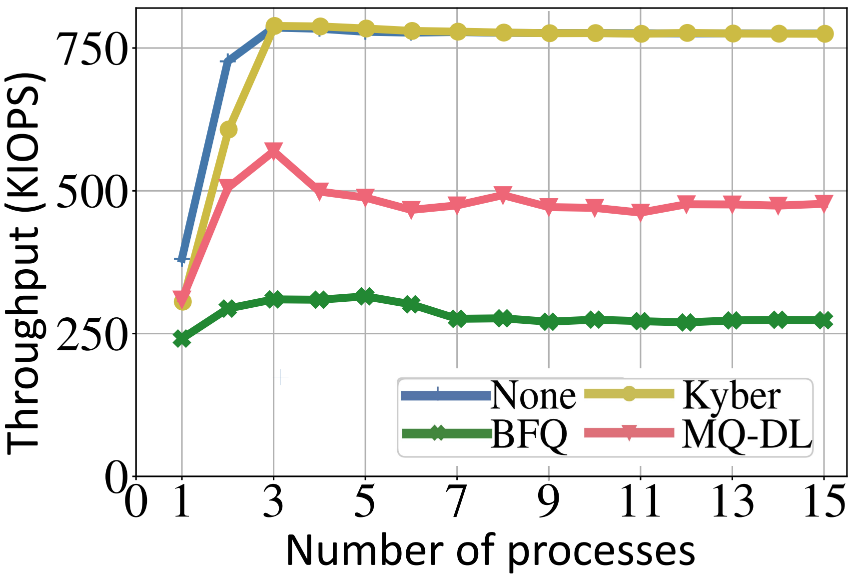
When CPU bottlenecked → higher latency overheads.

Scalability of T-apps: 1 SSD vs. 8 SSDs

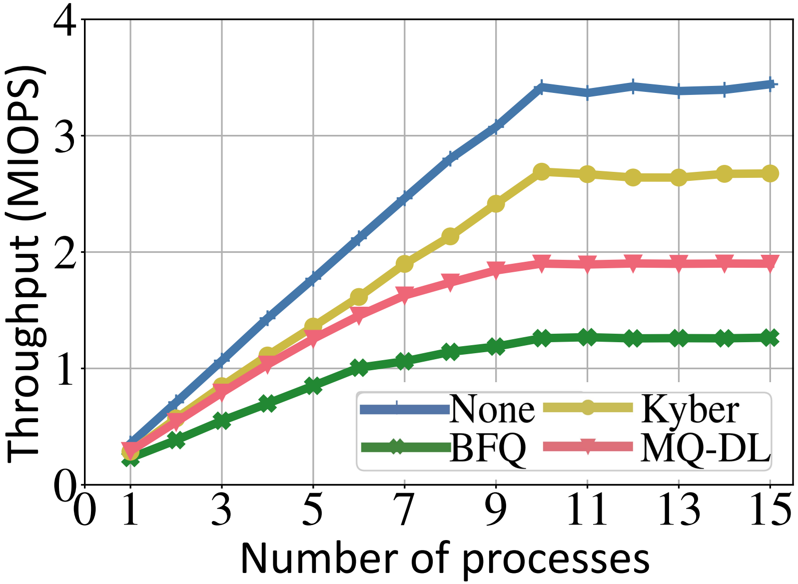


Big gap of scalability on throughput between different I/O schedulers.

Scalability of T-apps: 1 SSD vs. 8 SSDs



1 SSD

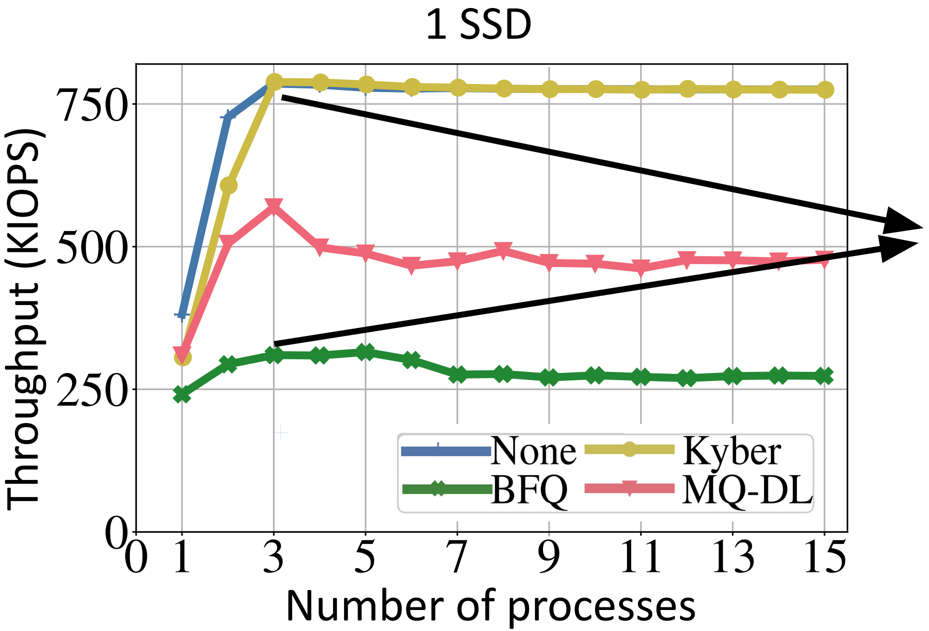


8 SSDs

Big gap of scalability on throughput between different I/O schedulers.

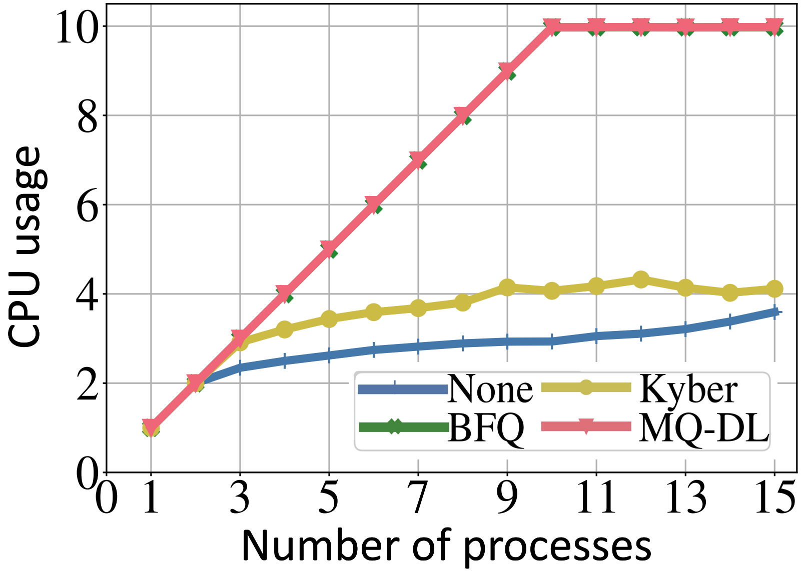
More devices → better scalability.

Scalability of T-apps: 1 SSD vs. 8 SSDs



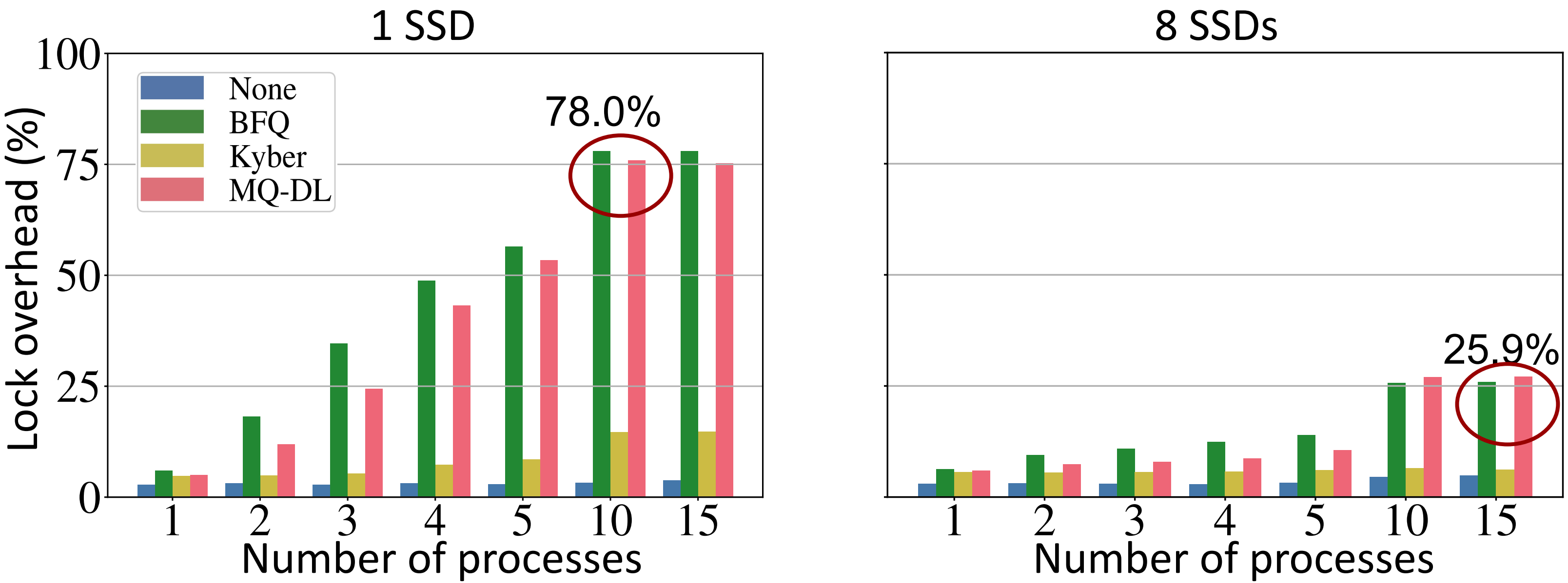
What cause this scalability issue?

Scalability of T-apps: CPU Usage



The scalability issues are caused by high CPU contention.

Lock Overhead of I/O Schedulers



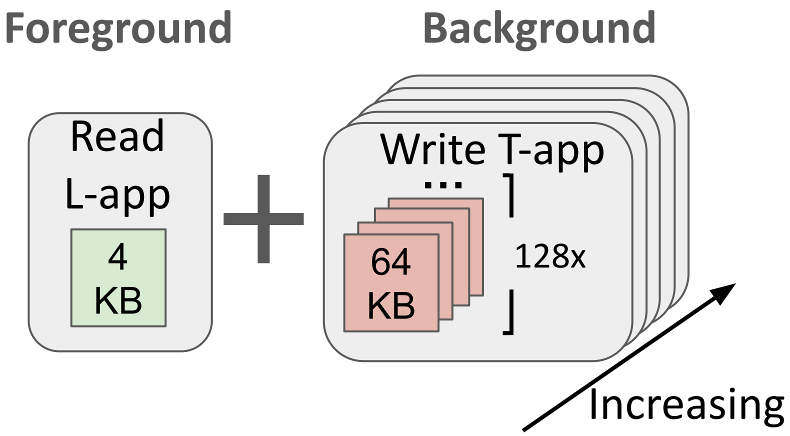
BFQ and MQ-Deadline → high CPU lock overhead.

Adding devices mitigates the lock overhead.

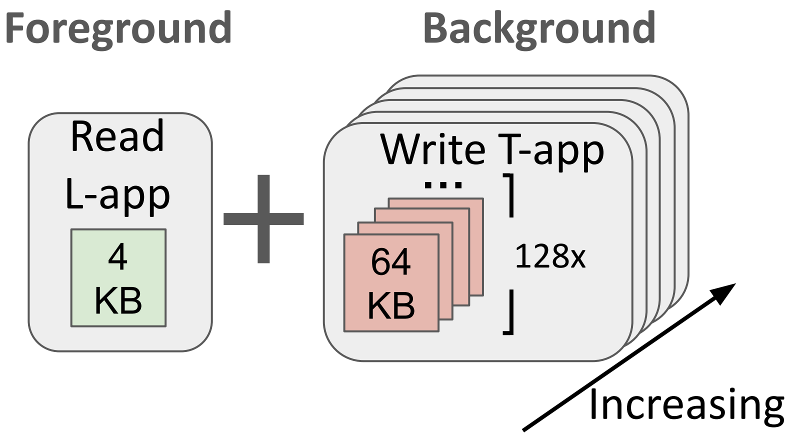
January, 2024: Identified by the Linux kernel developers[5][6].

RQ3. Taming I/O Interference

1 L-app (R) + Increasing T-apps (W)

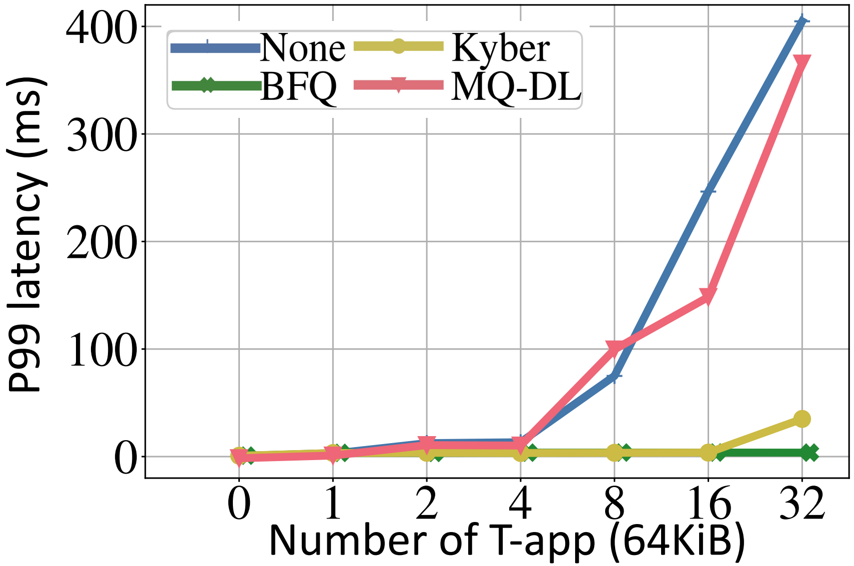


1 L-app (R) + Increasing T-apps (W)



Foreground

Background

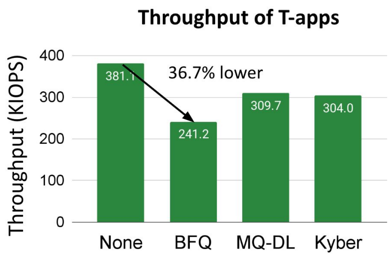


BFQ and Kyber → low latency for the foreground L-app.

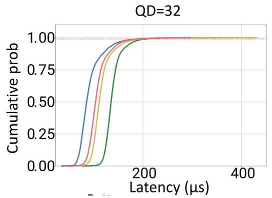
Conclusions

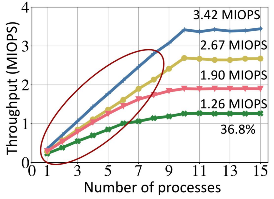
RQ1: What is the overhead of Linux I/O schedulers?

- Minor latency overhead.
- Significantly throughput overhead.



RQ2: What is the scalability of Linux I/O schedulers?

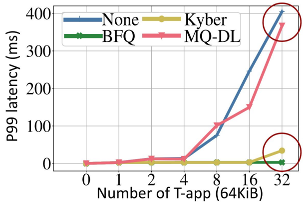




- Latency → depends on CPU.
- Throughput, BFQ and MQ-DL → high lock contention.
- Throughput, Kyber → good, similar to None.

RQ3: Can the Linux I/O schedulers tame I/O inference?

- Only BFQ and Kyber can provide bounded performance.



Take-Home Messages

1. *I/O Schedulers can influence the performance significantly*.
   None has the lowest overhead and highest scalability.
   BFQ has the highest overhead and lowest scalability.

2. *Different schedulers have different locking and scaling overheads.*
   BFQ = MQ-Deadline > Kyber > None.

3. Use ***Kyber*** to prioritize ***foreground reads with background writes.***
   HotCloudPerf'24 *A Systematic Configuration Space Exploration of the Linux Kyber I/O Scheduler*

Paper: https://atlarge-research.com/pdfs/2024-io-schedulers.pdf
Source code: https://github.com/ZebinRen/icpe24_io_scheduler_study_artifact

Thank you!
Questions?

Paper: https://atlarge-research.com/pdfs/2024-io-schedulers.pdf
Source code: https://github.com/ZebinRen/icpe24_io_scheduler_study_artifact

Resources

Images used:

https://www.samsung.com/nl/memory-storage/nvme-ssd/980-pro-pcle-4-0-nvme-m-2-ssd-1tb-mz-v8p1t0bw/
https://www.intel.com/content/www/us/en/products/details/memory-storage/data-center-ssds/optane-dc-ssd-series.html
https://www.anandtech.com/show/12376/samsung-launches-zssd-sz985-up-to-800gb-of-znand
https://www.storagereview.com/review/intel-x25-v-ssd-review-40gb

References

[1] Till Miemietz, Hannes Weisbach, Michael Roitzsch, Hermann Härtig: K2: Work-Constraining Scheduling of NVMe-Attached Storage. RTSS 2019: 56-68
[2] Mohammad Hedayati, Kai Shen, Michael L. Scott, Mike Marty: Multi-Queue Fair Queuing. USENIX Annual Technical Conference 2019: 301-314 2018
[3] Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, Rachit Agarwal: Rearchitecting Linux Storage Stack for µs Latency and High Throughput. OSDI 2021: 113-128
[4] Jiwon Woo, Minwoo Ahn, Gyusun Lee, Jinkyu Jeong: D2FQ: Device-Direct Fair Queueing for NVMe SSDs. FAST 2021: 403-415
[5] https://www.phoronix.com/news/BFQ-IO-Better-Scalability
[6] https://www.phoronix.com/news/MQ-Deadline-Scalability

Resources

Linux I/O schedulers

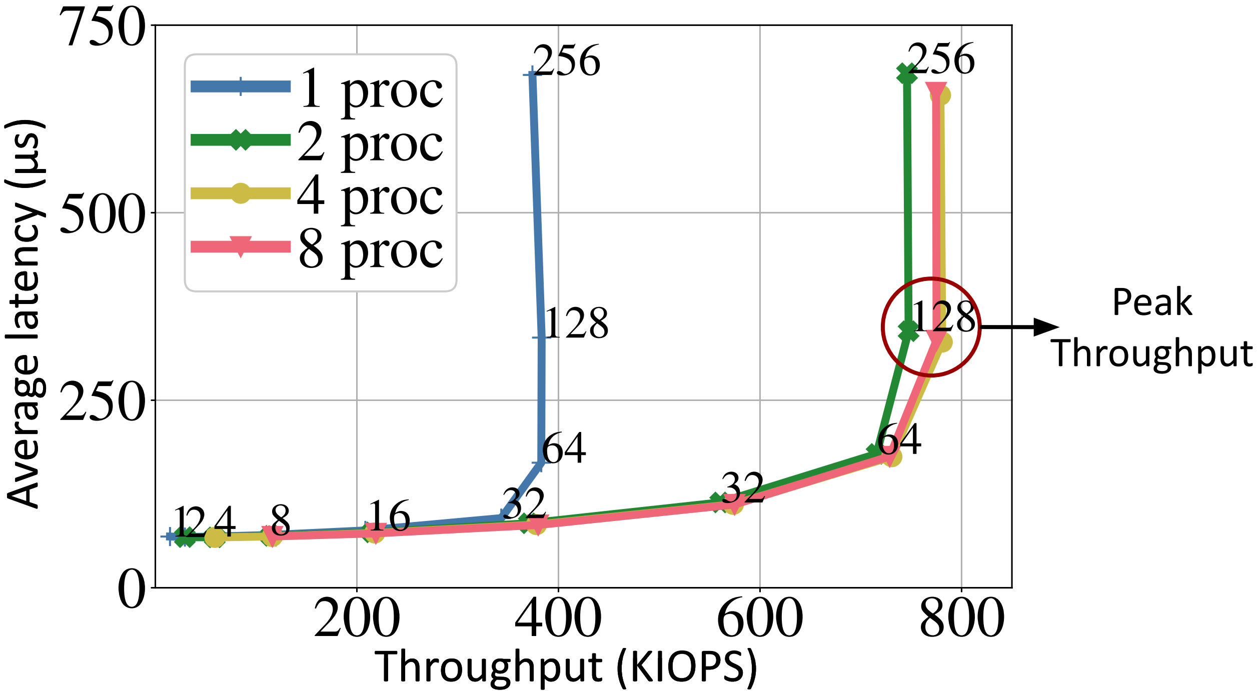
1. BFQ (Budget Fair Queueing) https://www.kernel.org/doc/html/latest/block/bfq-iosched.html
2. Two new block I/O schedulers for 4.12 https://lwn.net/Articles/720675/
3. Deadline IO scheduler tunables https://docs.kernel.org/block/deadline-iosched.html#:~:text=The%20goal%20of%20the%20deadline,value%20in%20units%20of%20milliseconds.
4. BFQ I/O Scheduler For Linux Sees Big Scalability Improvement https://www.phoronix.com/news/BFQ-IO-Better-Scalability
5. MQ-Deadline Scheduler Optimized For Much Better Scalability

New I/O schedulers

1. Myoungsoo Jung, Wonil Choi, Shekhar Srikantaiah, Joonhyuk Yoo, and Mahmut T. Kandemir. HIOS: A Host Interface I/O Scheduler for Solid State Disks. ISCA 2014.
2. Mingyang Wang and Yiming Hu. An I/O Scheduler Based on Fine-Grained Access Patterns to Improve SSD Performance and Lifespan. In Symposium on Applied Computing, SAC 2014.
3. Hui Lu, Brendan Saltaformaggio, Ramana Rao Kompella, and Dongyan Xu. vFair: Latency-Aware Fair Storage Scheduling via per-IO Cost-Based Differentiation. SoCC 2015.
4. Jiayang Guo, Yiming Hu, Bo Mao, and Suzhen Wu. Parallelism and Garbage Collection Aware I/O Scheduler with Improved SSD Performance. IPDPS 2017.
5. Minhoon Yi, Minho Lee, and Young Ik Eom. 2017. CFFQ: I/O Scheduler for Providing Fairness and High Performance in SSD Devices. IMCOM 2017.
6. Mohammad Hedayati, Kai Shen, Michael L. Scott, and Mike Marty. Multi- Queue Fair Queuing. In 2019 USENIX Annual Technical Conference, USENIX ATC 2019.
7. Till Miemietz, Hannes Weisbach, Michael Roitzsch, and Hermann Härtig. K2: Work-Constraining Scheduling of NVMe-Attached Storage. RTSS 2019.
8. Jaehyun Hwang, Midhul Vuppalapati, Simon Peter, and Rachit Agarwal. Rearchitecting Linux Storage Stack for μs Latency and High Throughput. OSDI 2021.
9. Jiwon Woo, Minwoo Ahn, Gyusun Lee, and Jinkyu Jeong. D2FQ: Device- Direct Fair Queueing for NVMe SSDs. FAST 2021.
10. Jieun Kim, Dohyun Kim, and Youjip Won Fair I/O Scheduler for Alleviating Read/Write Interference by Forced Unit Access in Flash Memory. HotStorage 2022.
11. Caeden Whitaker, Sidharth Sundar, Bryan Harris, and Nihat Altiparmak. Do We Still Need I/O Schedulers for Low-Latency Disks?. HotStorage 2023.

Backup Slides

CPU or NVMe SSD, What is the Bottleneck?

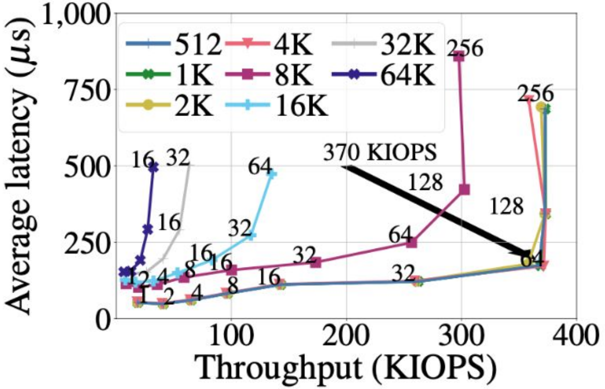


To saturate a SSD:

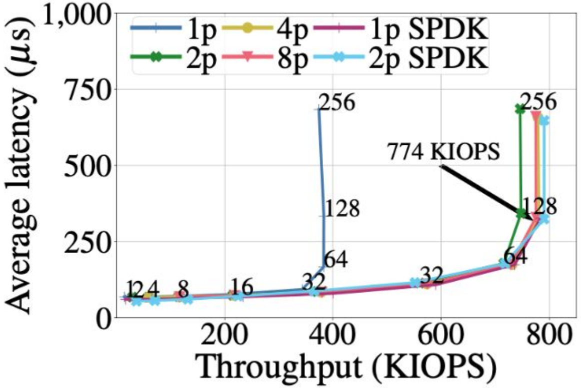
Multiple I/O requests

Enough CPU resources ⟶ 4 processes

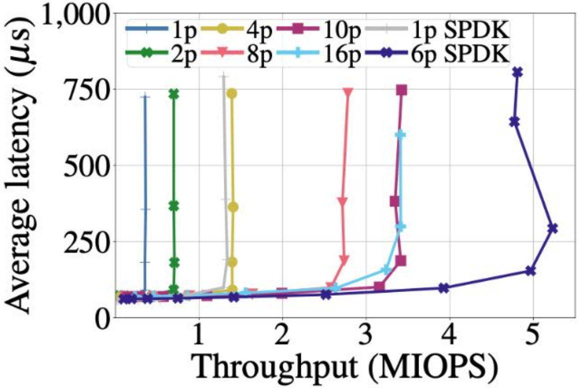
SSD Performance



(a) Vary request sizes, 1 SSD.

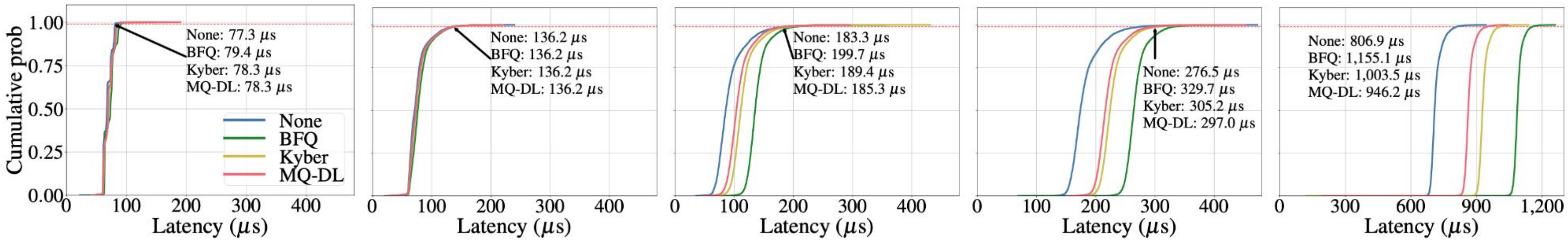


(b) Vary # processes with 1 SSD.



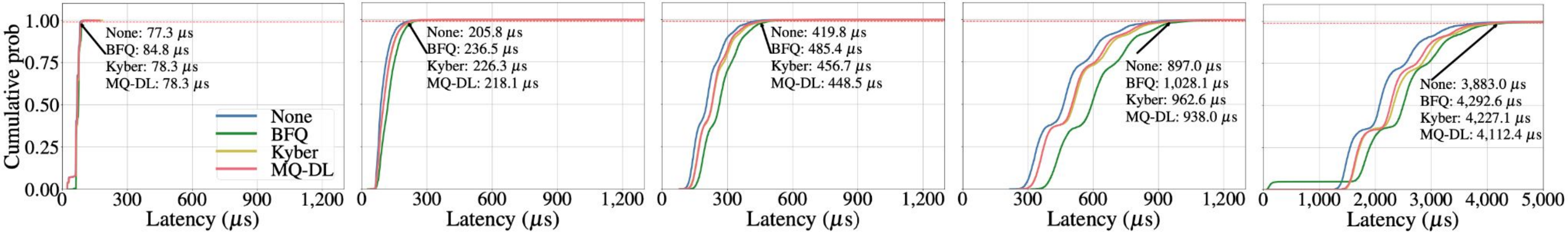
(c) Vary # processes with 8 SSDs.

L-app Scalability



(a) QD = 1 **(b) QD = 16** **(c) QD = 32** **(d) QD = 64** **(e) QD = 256**

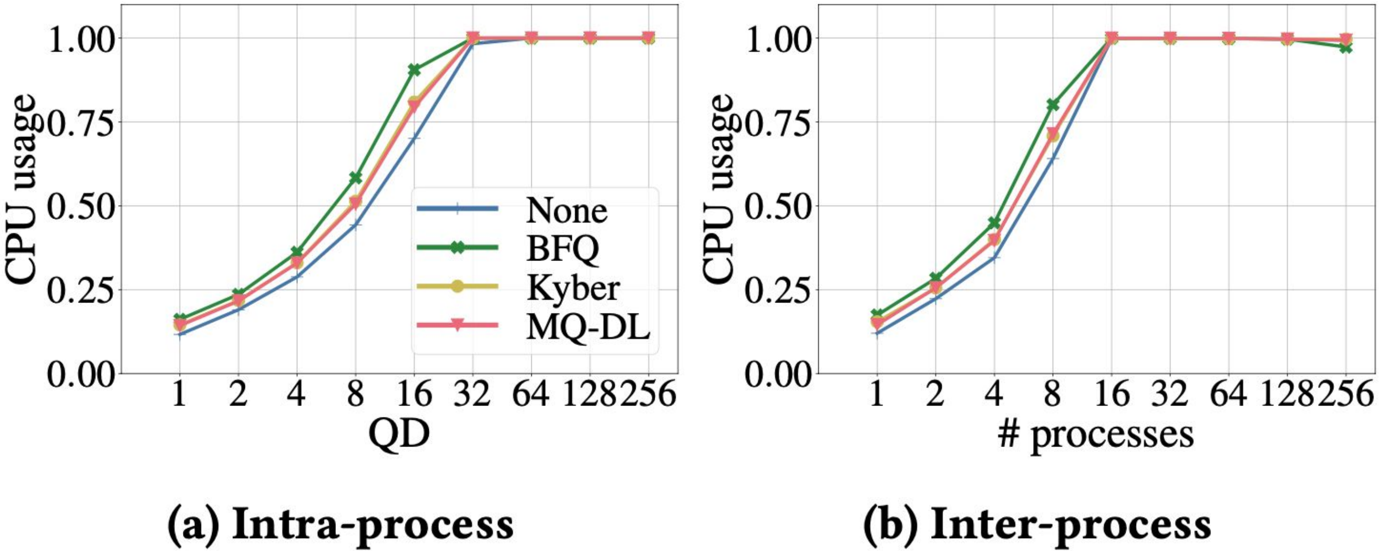
Figure 2: Intra-process scalability latency CDFs with increasing queue depth (QD); Note the different x-axis scale for (e).



(a) 1 L-app **(b) 16 L-apps** **(c) 32 L-apps** **(d) 64 L-apps** **(e) 256 L-apps**

Figure 3: Inter-process scalability latency CDFs with increasing number of L-apps; Note the different x-axis scale for (e).

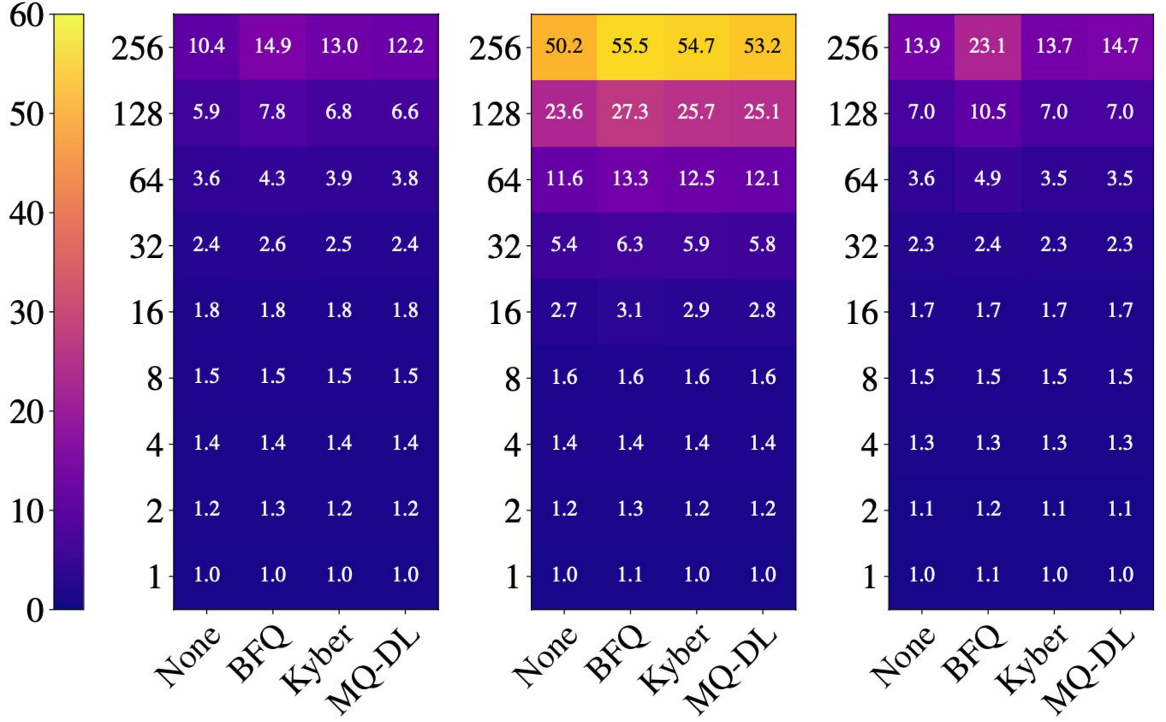
L-app CPU cost



(a) Intra-process **(b) Inter-process**

Figure 4: CPU usage for intra/inter-process concurrency.

L-app Scalability Heatmap

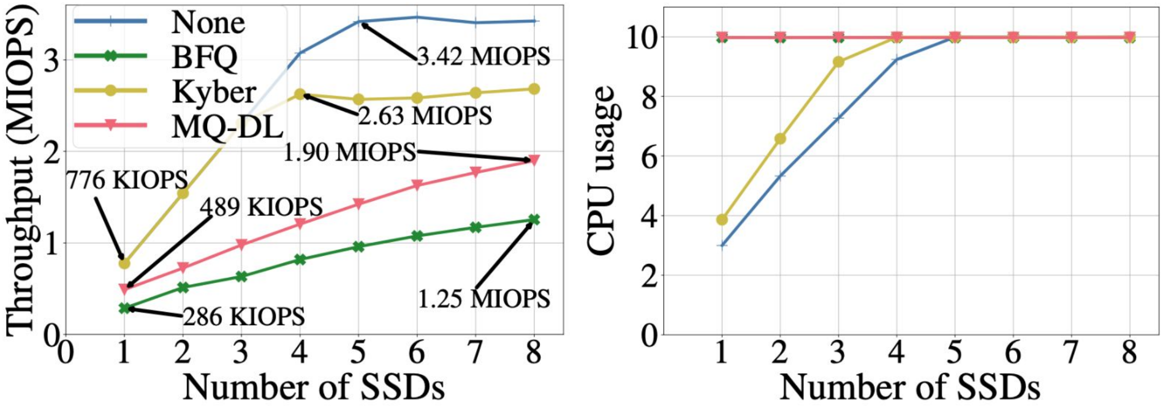


(a) 1 core, 1 process
y-axis: QD

(b) 1 core, QD 1
y-axis: # processes

(c) 10 cores, QD 1
y-axis: # processes

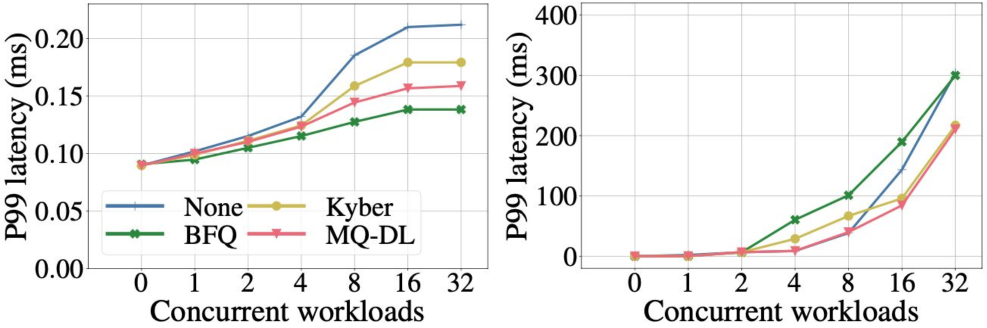
SSD Scalability



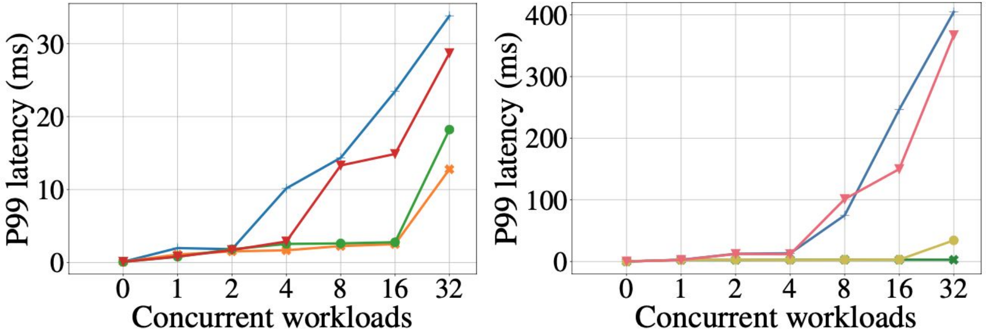
(a) Total throughput **(b) Total CPU usage**

Figure 9: T-app inter-process scalability (10 cores, 10 concurrent T-4KiB-apps) with an increasing number of SSDs.

L-app Interference



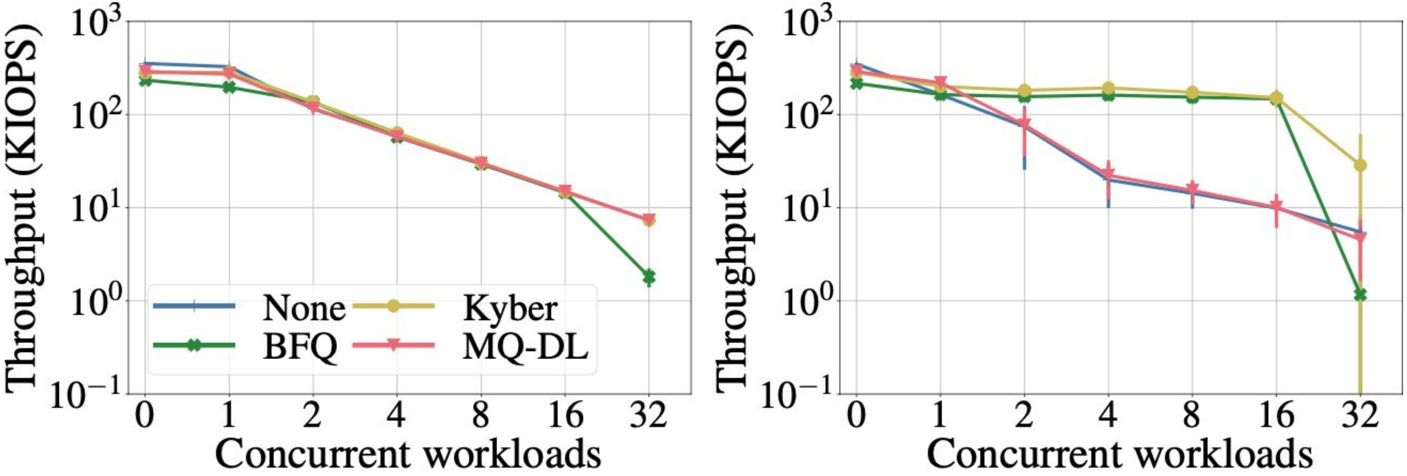
(a) T-4KiB-app (reads) (b) T-64KiB-app (reads)



(c) T-4KiB-app (writes) (d) T-64KiB-app (writes)

Figure 11: L-app tail latency with an increasing number of interfering background applications; ***Note: scales differ on the y-axis and they are in Milliseconds!***

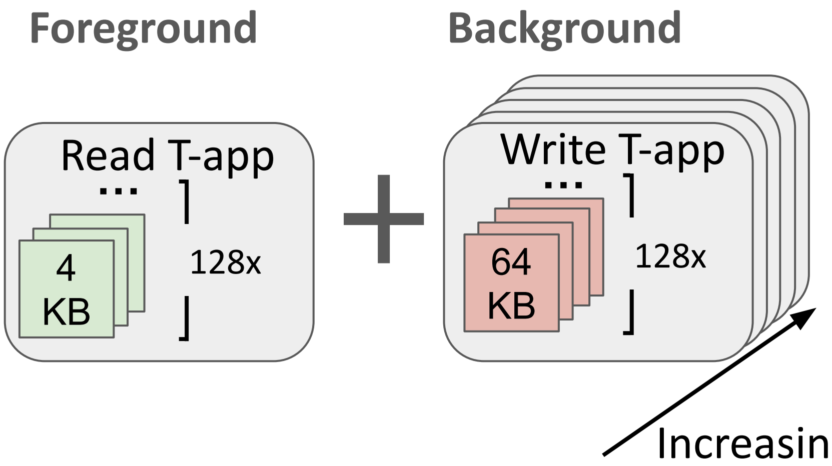
T-app Interference

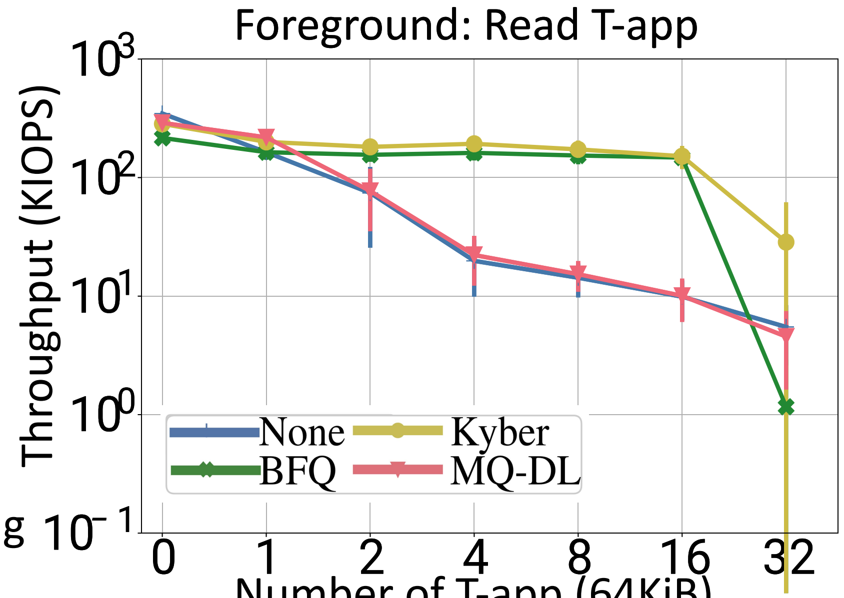


(a) T-64KiB-apps (reads) **(b) T-64KiB-apps (writes)**

Figure 12: Read throughput (IOPS) of a T-4KiB-app workload with an increasing number of interfering background T-64KiB-app workload. Note: The y-axis is log-scale.

1 T-app (R) + Increasing T-apps (W)

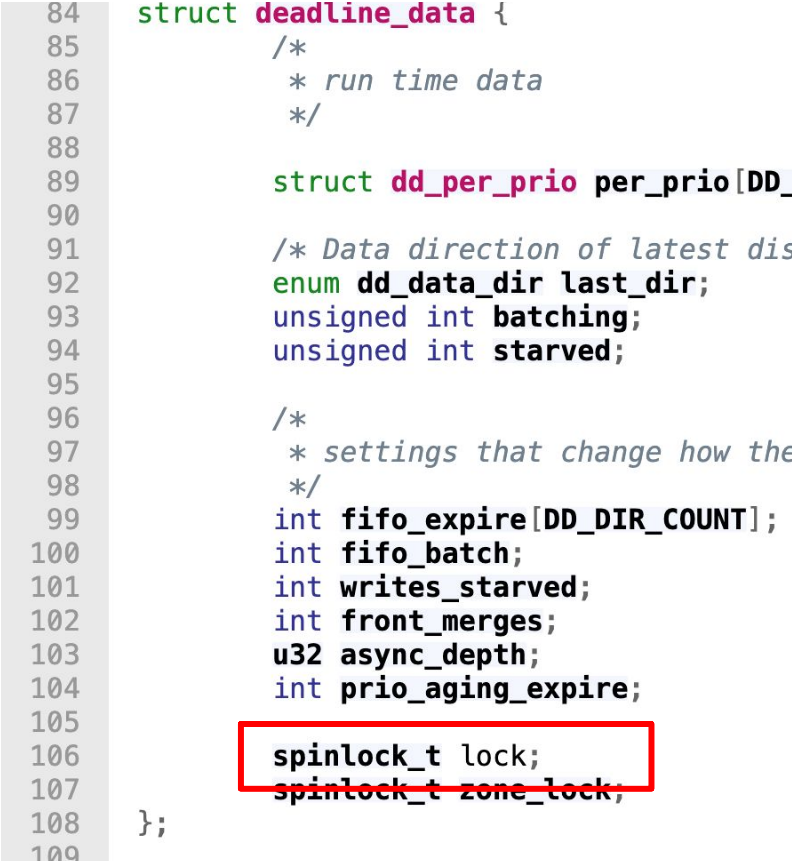




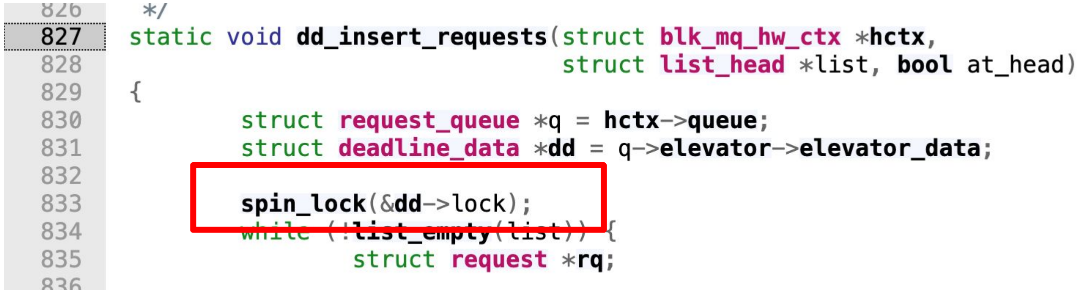
BFQ and Kyber → higher bandwidth for the foreground T-app.

Lock in the I/O Schedulers

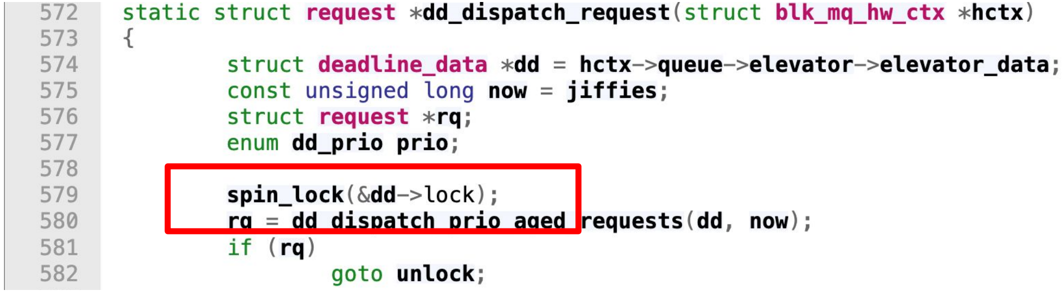
In block/mq-deadline.c



```
struct deadline_data {
        /*
         * run time data
         */

        struct dd_per_prio per_prio[DD_

        /* Data direction of latest dis
        enum dd_data_dir last_dir;
        unsigned int batching;
        unsigned int starved;

        /*
         * settings that change how the
         */
        int fifo_expire[DD_DIR_COUNT];
        int fifo_batch;
        int writes_starved;
        int front_merges;
        u32 async_depth;
        int prio_aging_expire;

        spinlock_t lock;
        spinlock_t zone_lock;
};
```



```
 */
static void dd_insert_requests(struct blk_mq_hw_ctx *hctx,
                               struct list_head *list, bool at_head)
{
        struct request_queue *q = hctx->queue;
        struct deadline_data *dd = q->elevator->elevator_data;

        spin_lock(&dd->lock);
        while (!list_empty(list)) {
                struct request *rq;
```

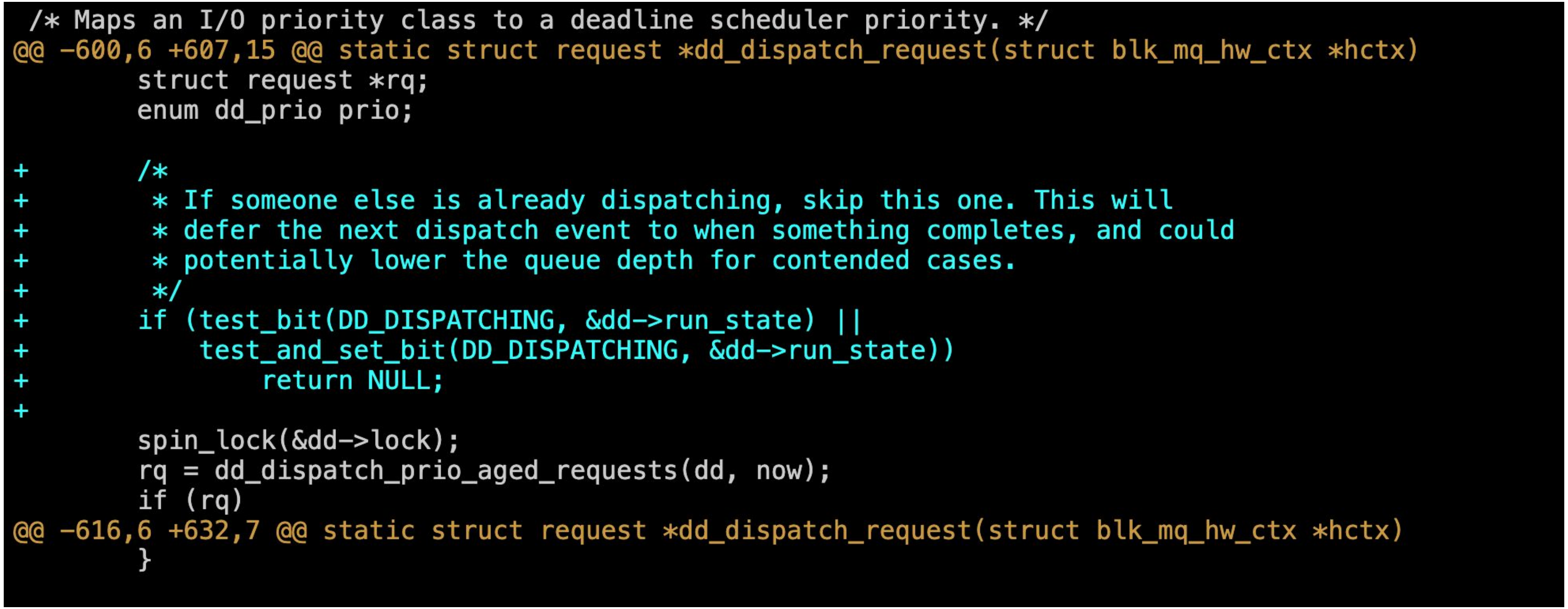


```
static struct request *dd_dispatch_request(struct blk_mq_hw_ctx *hctx)
{
        struct deadline_data *dd = hctx->queue->elevator->elevator_data;
        const unsigned long now = jiffies;
        struct request *rq;
        enum dd_prio prio;

        spin_lock(&dd->lock);
        rq = dd_dispatch_prio_aged_requests(dd, now);
        if (rq)
                goto unlock;
```

Lock in the I/O Schedulers

Reduced lock contention

Dispatch



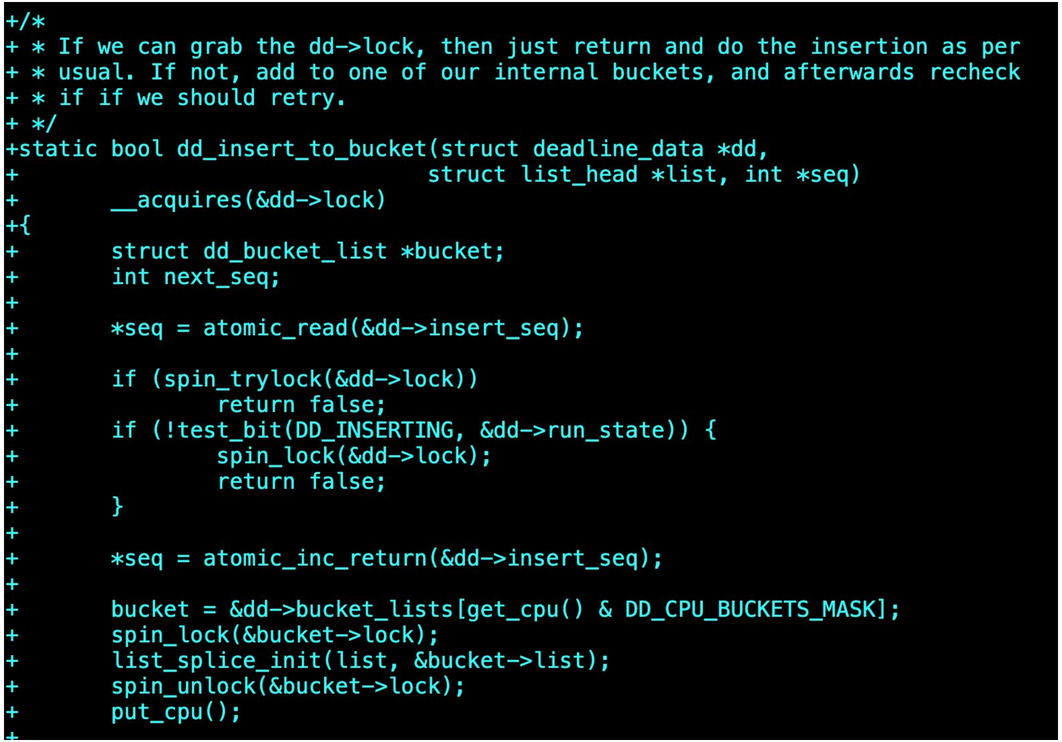
```
 /* Maps an I/O priority class to a deadline scheduler priority. */
@@ -600,6 +607,15 @@ static struct request *dd_dispatch_request(struct blk_mq_hw_ctx *hctx)
        struct request *rq;
        enum dd_prio prio;

+       /*
+        * If someone else is already dispatching, skip this one. This will
+        * defer the next dispatch event to when something completes, and could
+        * potentially lower the queue depth for contended cases.
+        */
+       if (test_bit(DD_DISPATCHING, &dd->run_state) ||
+           test_and_set_bit(DD_DISPATCHING, &dd->run_state))
+               return NULL;
+
        spin_lock(&dd->lock);
        rq = dd_dispatch_prio_aged_requests(dd, now);
        if (rq)
@@ -616,6 +632,7 @@ static struct request *dd_dispatch_request(struct blk_mq_hw_ctx *hctx)
        }
```

https://lore.kernel.org/linux-block/20240118180541.930783-2-axboe@kernel.dk/

Lock in the I/O Schedulers

Reduced lock contention

Insertion



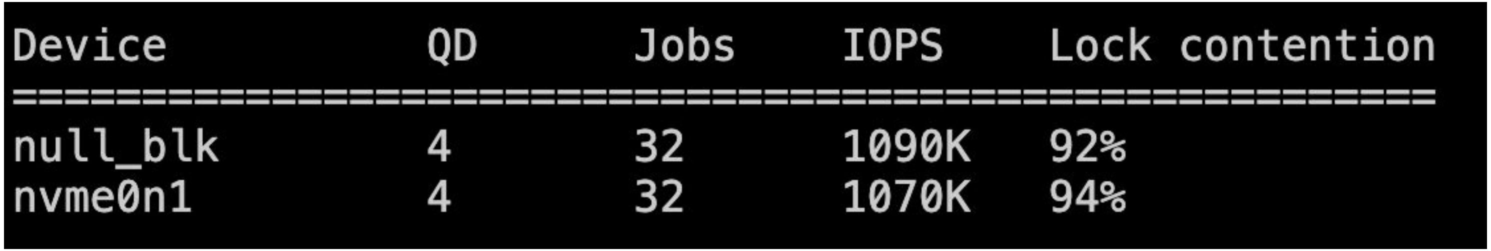
```
+/*
+ * If we can grab the dd->lock, then just return and do the insertion as per
+ * usual. If not, add to one of our internal buckets, and afterwards recheck
+ * if if we should retry.
+ */
+static bool dd_insert_to_bucket(struct deadline_data *dd,
+                               struct list_head *list, int *seq)
+       __acquires(&dd->lock)
+{
+       struct dd_bucket_list *bucket;
+       int next_seq;
+
+       *seq = atomic_read(&dd->insert_seq);
+
+       if (spin_trylock(&dd->lock))
+               return false;
+       if (!test_bit(DD_INSERTING, &dd->run_state)) {
+               spin_lock(&dd->lock);
+               return false;
+       }
+
+       *seq = atomic_inc_return(&dd->insert_seq);
+
+       bucket = &dd->bucket_lists[get_cpu() & DD_CPU_BUCKETS_MASK];
+       spin_lock(&bucket->lock);
+       list_splice_init(list, &bucket->list);
+       spin_unlock(&bucket->lock);
+       put_cpu();
+
```

https://lore.kernel.org/linux-block/ede4179c-8fa5-4496-ac21-4e3fda41df81@kernel.dk/

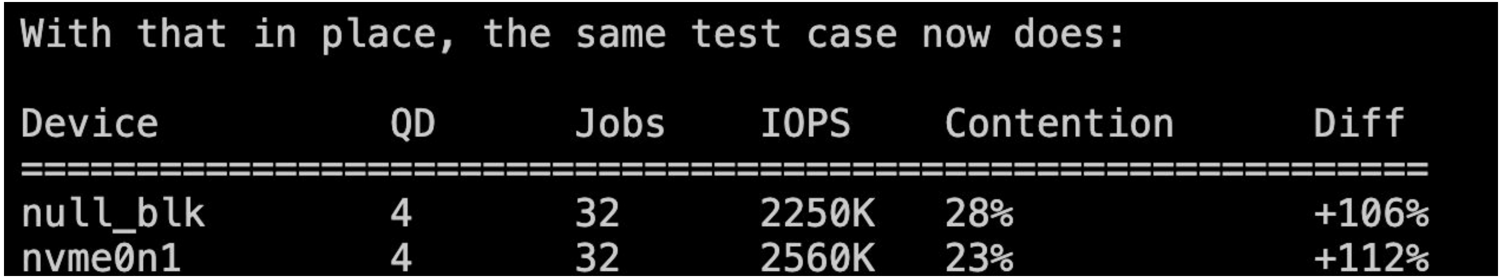
Lock in the I/O Schedulers

Reduced lock contention

Results



```
Device          QD      Jobs    IOPS    Lock contention
=======================================================
null_blk        4       32      1090K   92%
nvme0n1         4       32      1070K   94%
```



```
With that in place, the same test case now does:

Device          QD      Jobs    IOPS    Contention      Diff
=============================================================
null_blk        4       32      2250K   28%             +106%
nvme0n1         4       32      2560K   23%             +112%
```

https://lore.kernel.org/linux-block/20240118180541.930783-1-axboe@kernel.dk/?s=09

Who Are We/Am I?