NVMeTCP Offload

Implementation and Performance Gains

Shai Malin, Aurelien Aptel
Boris Pismenny, Yoray Zack, Ben Ben-Ishay and Or Gerlitz

Agenda

- The motivation for the NVMeTCP offload
- The challenge
- The offload design and implementation
- Performance
- Debug lessons

The Offload Opportunities

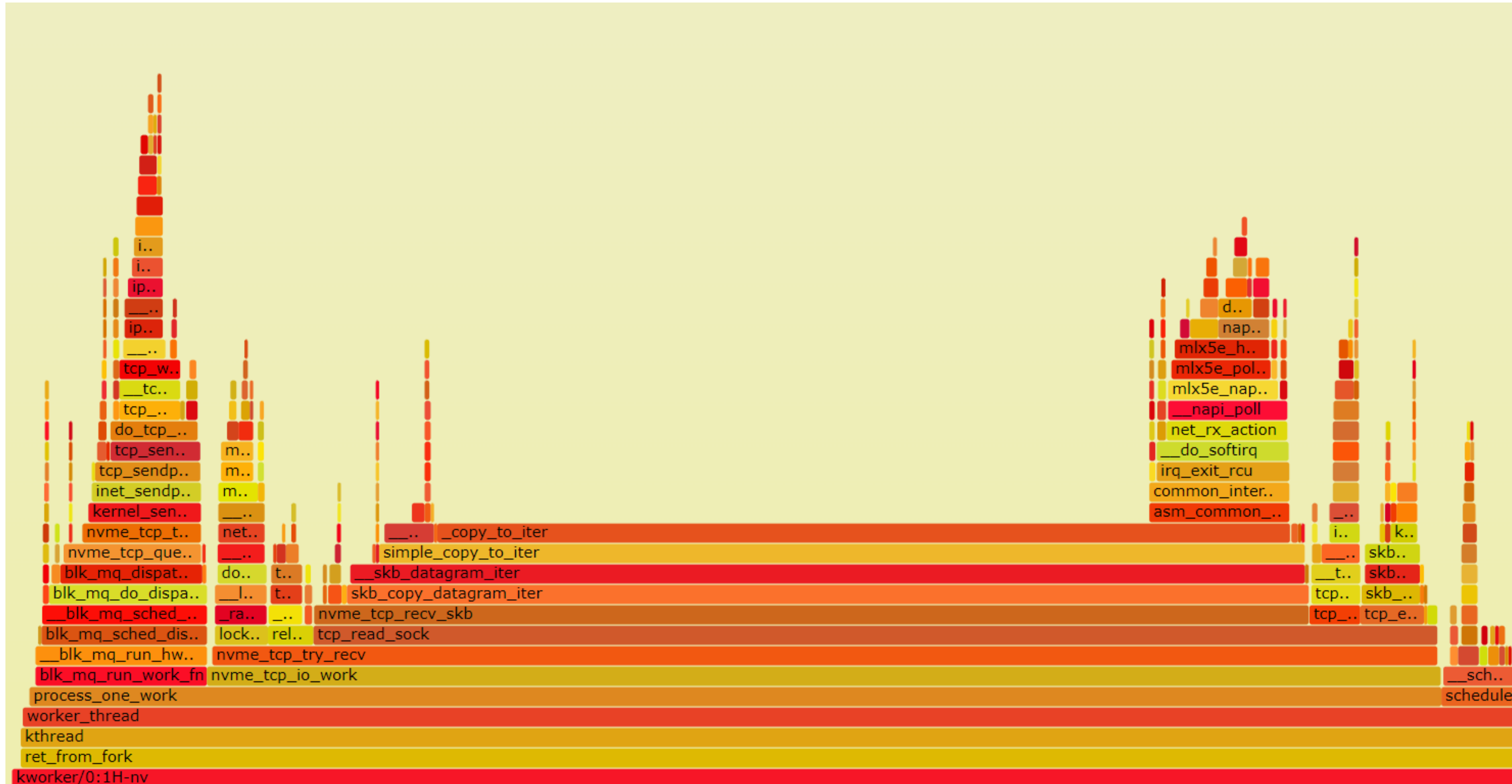
- Receive side zero-copy

  The data arrives as TCP stream and needs to be copied into the destination buffers.

- Receive side data CRC validation

- Transmit side data CRC calculation

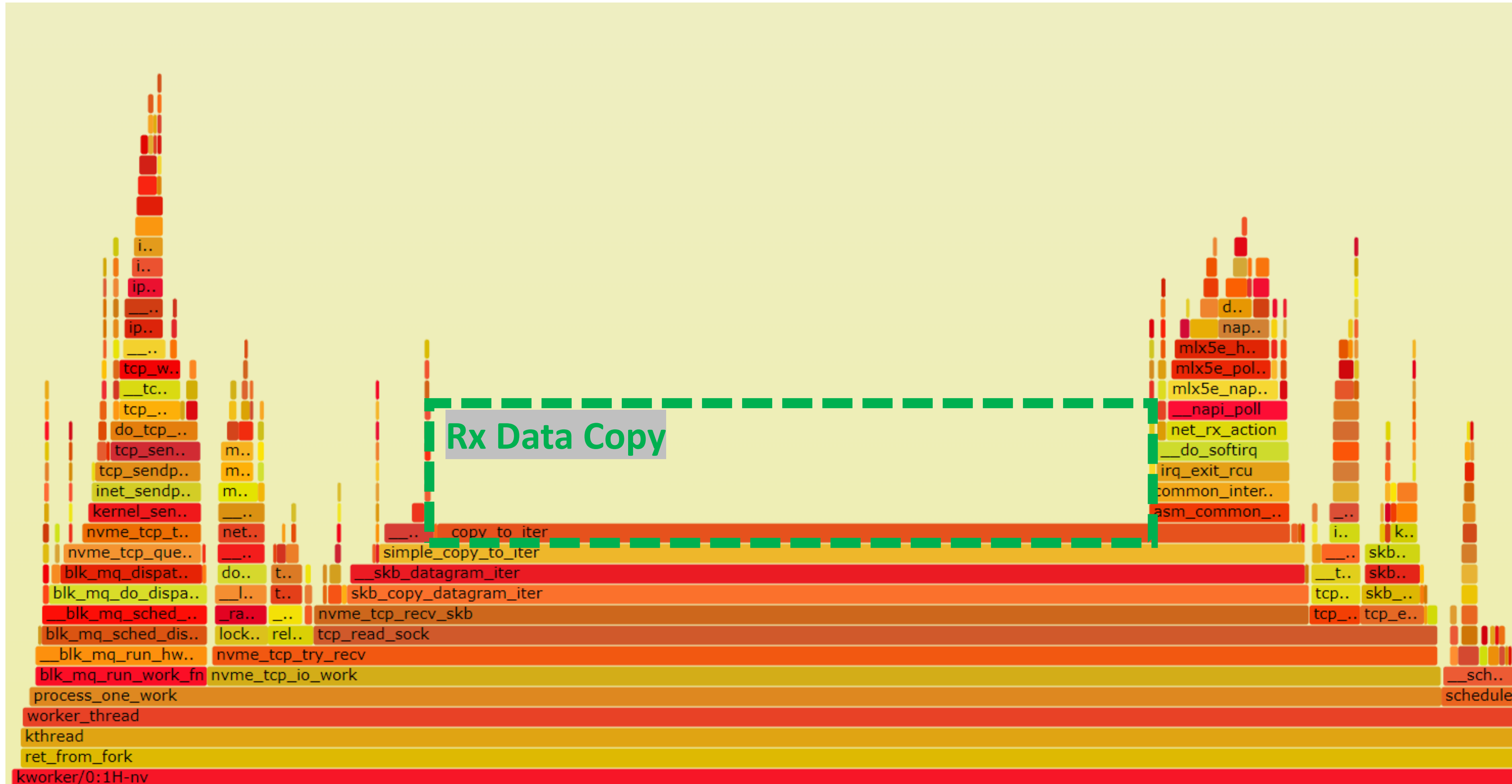
The Motivation



fio --bs=128k --rw=randread --ioengine=libaio --bs=[x] iodepth=128 --numjobs=1 | Without data digest

The Motivation

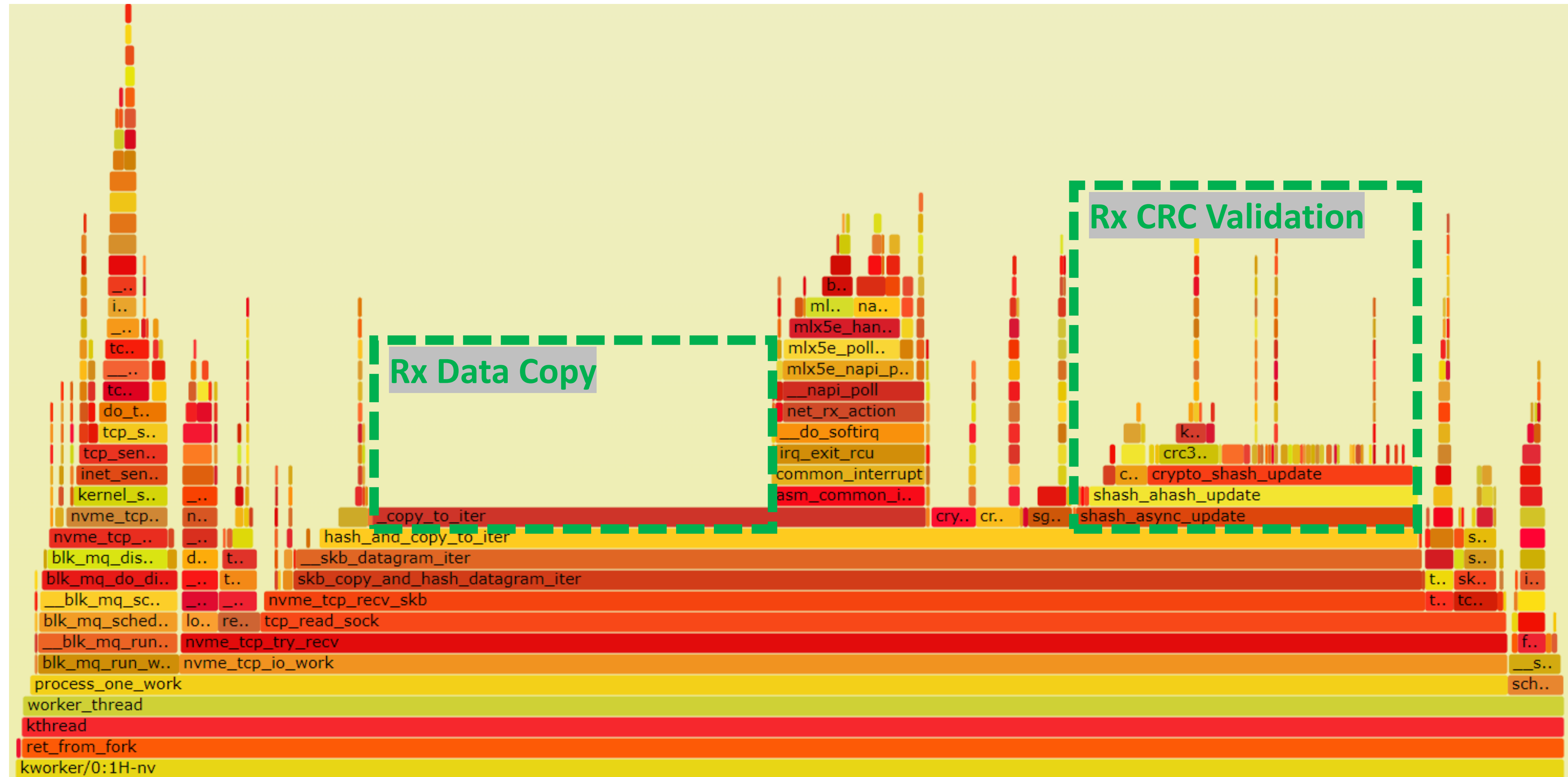
Saving the Data Copy:



fio --bs=128k --rw=randread --ioengine=libaio --bs=[x] iodepth=128 --numjobs=1 | Without data digest

The Motivation

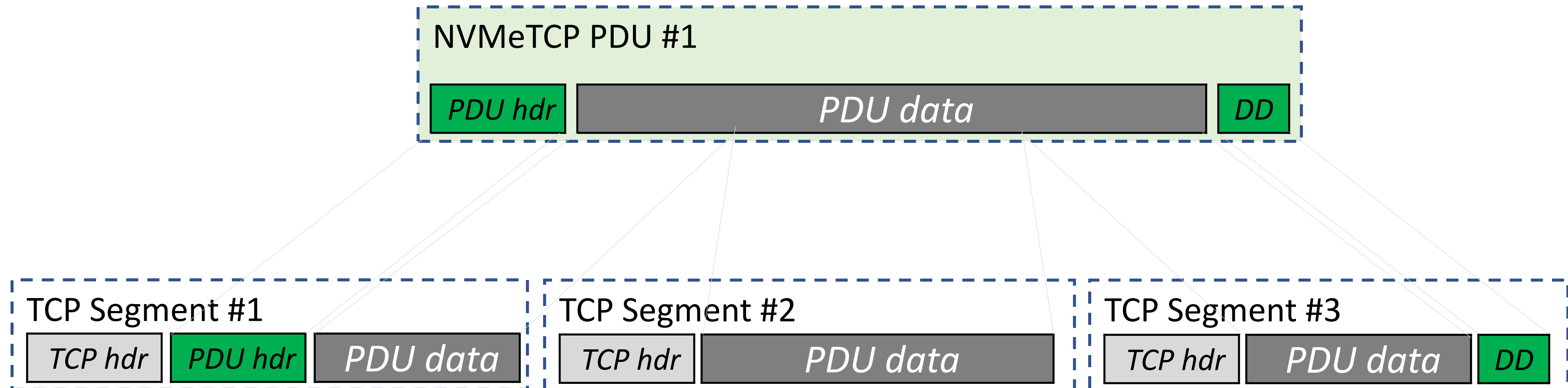
Saving the Data Copy and the CRC Calculation:



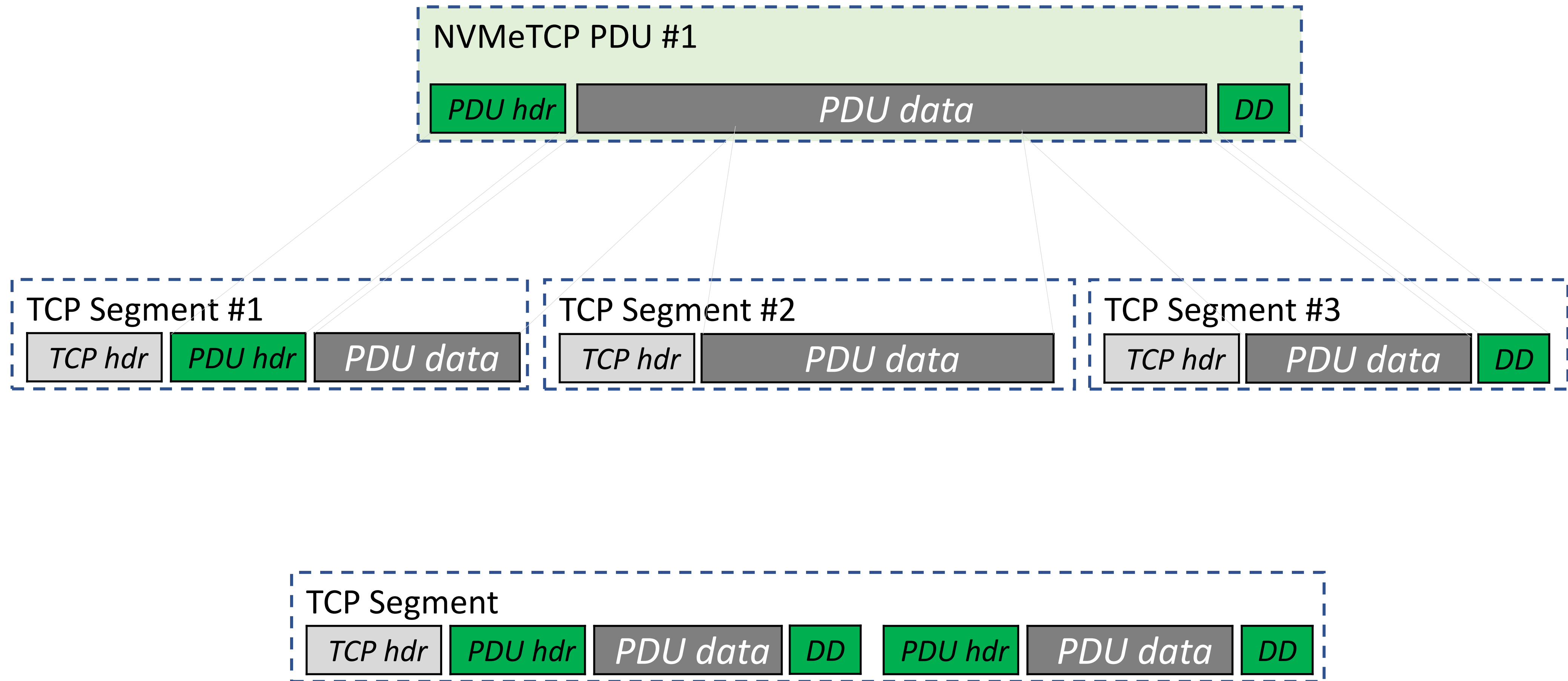
fio --bs=128k --rw=randread --ioengine=libaio --bs=[x] iodepth=128 --numjobs=1 | With data digest

The Challenge

NVMeTCP PDU vs TCP Segments



NVMeTCP PDU vs TCP Segments



The Challenge

- TCP receives data in anonymous unaligned buffers.
- PDU out of order is allowed – the remote side is allowed to reorder the transmitted PDUs.

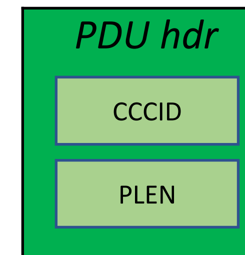
-> Generic receive zero-copy will not work.

This requires to track the TCP stream and distinguish the

NVMeTCP PDU based on the PDU header CCCID and PLEN.

CCCID – Command Capsule CID

PLEN – Total length of the PDU



The Offload Design

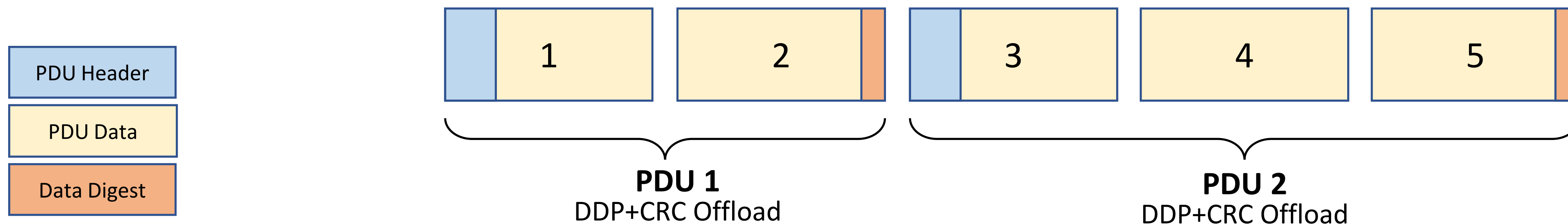
The Offload Rx Design (in-order)

1. Identify the TCP connection based on a 5-tuple steering rule.

2. Track the TCP stream and **identify the NVMeTCP PDUs** in the stream based on PDU headers which include command identifier (CCCID) and PDU length (PLEN).

3. In case of Data PDU:

   - **DDP (Direct Data Placement)** – Place the PDU data directly into the end-buffer in the relevant offset which is based on the TCP sequence.
   - **CRC Offload** - Calculate the start/continuation of the data digest and verify the result at the end of the PDU.



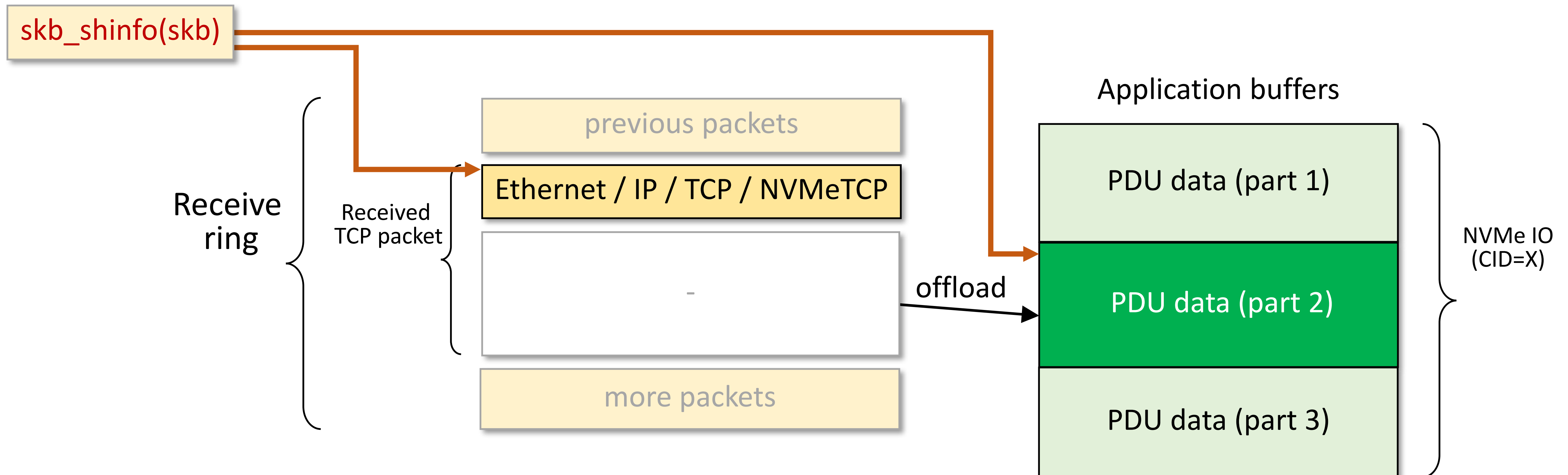
The Driver SKB Build

NIC driver builds SKBs of packets:

- Packet headers from receive ring
- Storage protocol header from receive ring
- Payload from destination buffers

Storage protocol skips copy

If (src addr == dst addr)

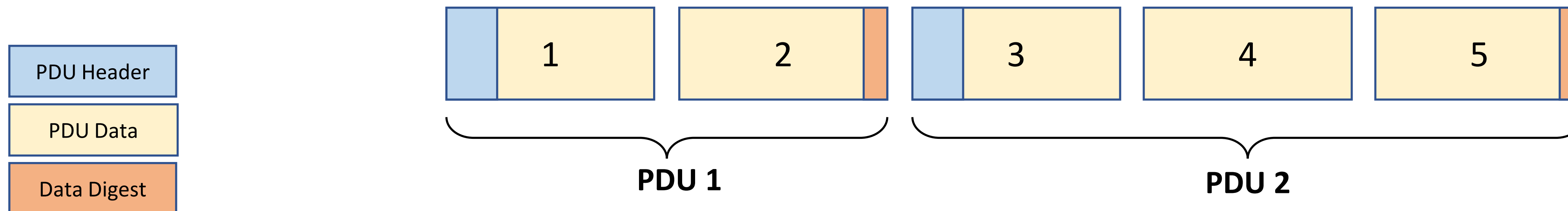


The Offload Rx Design (out-of-order)

1. Based on the PDU header DATAL, the HW can anticipate which TCP sequence range is within the current PDU.

2. If the missing TCP packet is in the middle of the PDU, the HW will continue the direct data placement of the following packets.

   In this case the PDU data digest will be calculated by software.



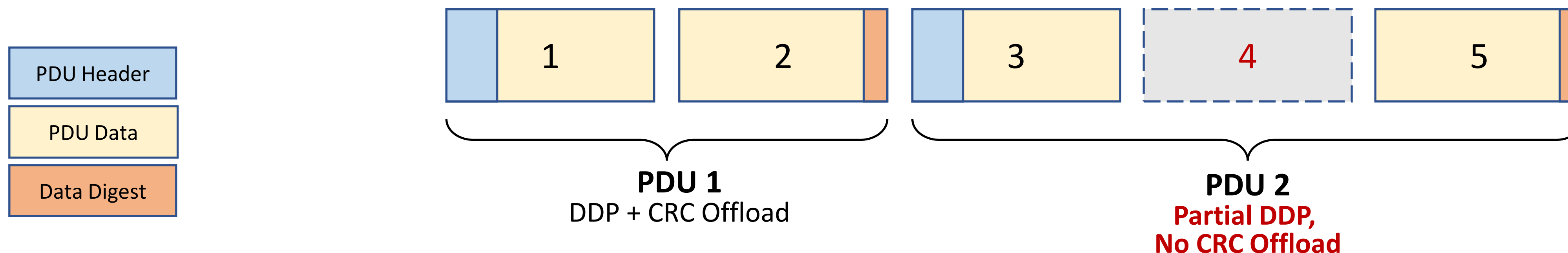
The Offload Rx Design (out-of-order)

1. Based on the PDU header DATAL, the HW can anticipate which TCP sequence range is within the current PDU.

2. If the missing TCP packet is in the middle of the PDU, the HW will continue the direct data placement of the following packets.

   In this case the PDU data digest will be calculated by software.

   When the missing packet will arrive (out-of-order), the HW will bypass the offload flow.



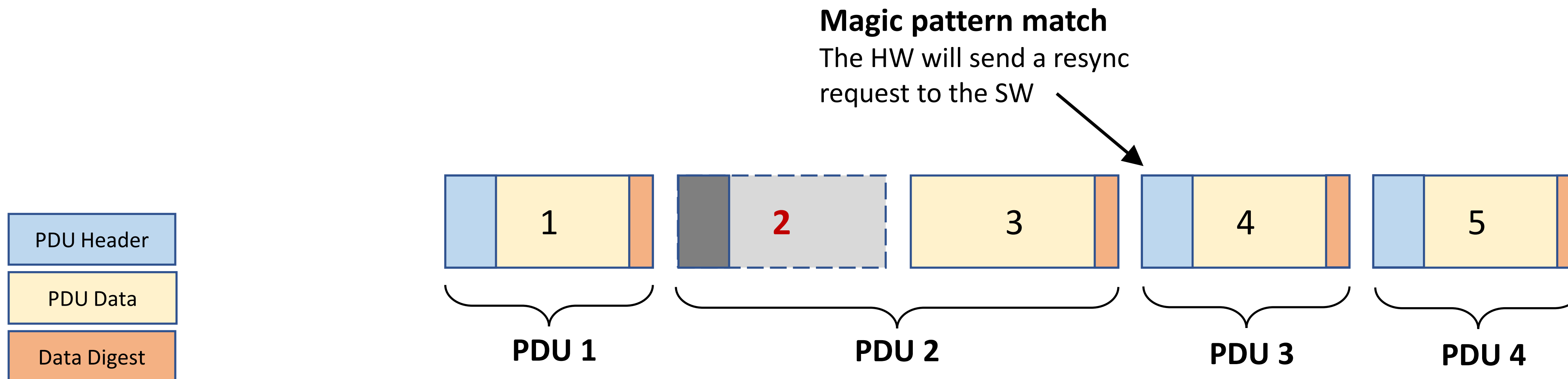
*For more detail -> Netdev 0x15 Autonomous NVMe-TCP Offload

The Offload Rx Design (out-of-order)

3. If the missing TCP packet includes a PDU header:
   - The HW will pause the offloading and in the next following packets, the HW will compare the start of the packet with the **magic pattern** as optimistic approach, and in case of a match, the HW will send a **resync request** to the software.

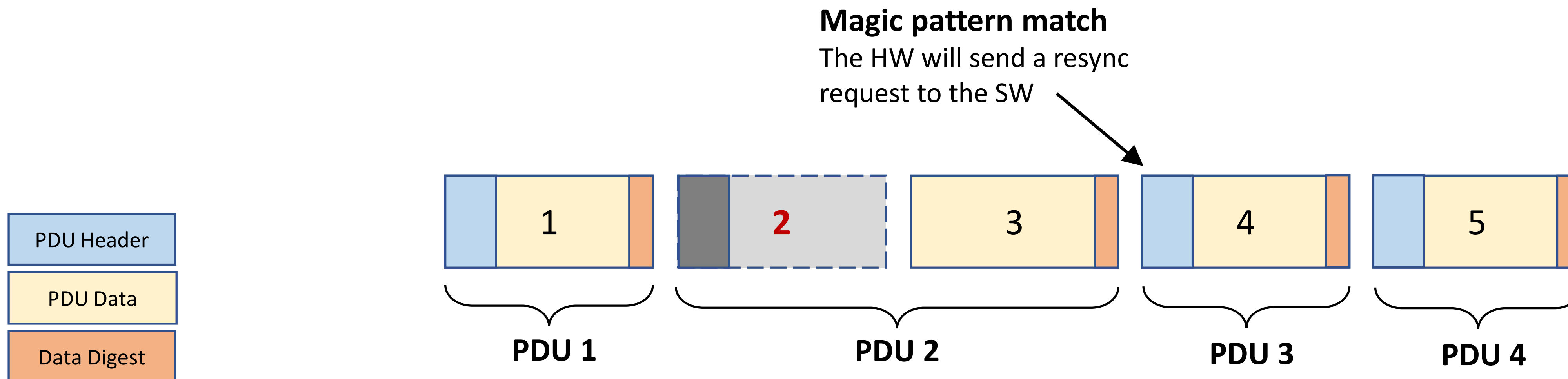
The Offload Rx Design (out-of-order)

3. If the missing TCP packet includes a PDU header:
    - The HW will pause the offloading and in the next following packets, the HW will compare the start of the packet with the **magic pattern** as optimistic approach, and in case of a match, the HW will send a **resync request** to the software.



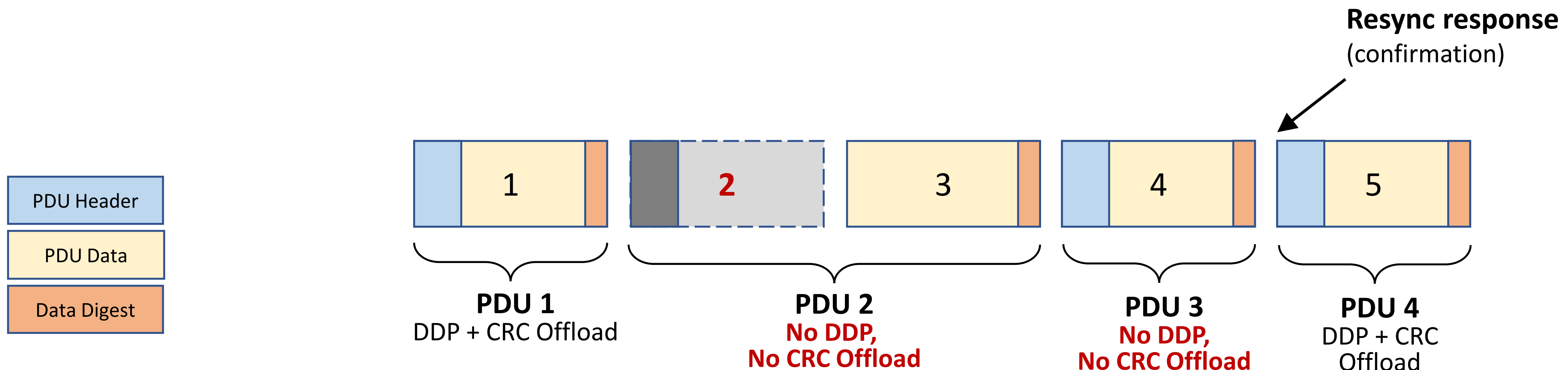
The Offload Rx Design (out-of-order)

3. If the missing TCP packet includes a PDU header:
   - The HW will pause the offloading and in the next following packets, the HW will compare the start of the packet with the **magic pattern** as optimistic approach, and in case of a match, the HW will send a **resync request** to the software.
   - The HW will continue to track the incoming stream, without performing the DDP, while it is waiting for the resync response.



The Offload Rx Design (out-of-order)

3. If the missing TCP packet includes a PDU header:
    - The HW will pause the offloading and in the next following packets, the HW will compare the start of the packet with the **magic pattern** as optimistic approach, and in case of a match, the HW will send a **resync request** to the software.
    - The HW will continue to track the incoming stream, without performing the DDP, while it is waiting for the resync response.
    - Once the software resyncs the HW with the new state (confirmation of the magic pattern), the offload will continue.



The Offload Rx Design (out-of-order)

3. If the missing TCP packet includes a PDU header:
    - The HW will pause the offloading and in the next following packets, the HW will compare the start of the packet with the **magic pattern** as optimistic approach, and in case of a match, the HW will send a **resync request** to the software.
    - The HW will continue to track the incoming stream, without performing the DDP, while it is waiting for the resync response.
    - Once the software resyncs the HW with the new state (confirmation of the magic pattern), the offload will continue.

The resync does not terminate the offload or stop the Rx from receiving the incoming packets.

*For more detail -> Netdev 0x15 Autonomous NVMe-TCP Offload

The Offload Tx Design

1. Instead of computing the NVMeTCP PDU data digest by the software layer, the driver marks packets for data digest offload based on the socket the packet is attached to.

2. The HW identifies the packet as requiring data digest offload handling and performs data digest calculation of the PDU data. It replaces the PDU data digest and TCP checksum with correct values.

3. Both the device and the driver maintain expected TCP sequence values in order to handle retransmissions.

4. Retransmission of a packet in the middle of the PDU will not require to be handled by the offload IO path.

5. If the retransmission includes the PDU data digest, the software will resend the entire PDU to the HW, which will calculate the data digest but will send to the wire only last segment which includes the data digest.

*For more detail -> Netdev 0x15 Autonomous NVMe-TCP Offload

The Offload Flows

1. The offload (DDP and CRC offload) is a net-device capability. When creating a new NVMeTCP queue, the offload will be enabled if the module param *ulp_offload* is set and if *netdev->features & NETIF_F_HW_ULP_DDP.*

2. The offload for IO queues is initialized after the handshake of the NVMe-TCP protocol is finished by calling *nvme_tcp_offload_socket()*. This operation sets all relevant hardware contexts in the hardware.

3. On the IO path, per IO:
    - The NVMeTCP layer will call *nvme_tcp_setup_ddp()* to map the IO buffer and to configure the HW for the specific IO (CCCID, buffer). This flow is opportunistic in order to avoid the waiting for the HW completions.
    - Once the IO has completed by the NVMeTCP layer, but before posting the completion to the upper layer, the *nvme_tcp_teardown_ddp()* will invalidate the HW buffer.

4. The resynchronization flow:
    - Resync request from the device HW to the SW, regarding a possible location of a PDU header.
    - Resync response from the NVMe-TCP driver to the device HW.

SKB changes

In order the allow the design, 2 New SKB bits **skb->ulp_ddp** and **skb->ulp_crc**

Used similarly to TLS's skb->decrypted

- On transmit skb->ulp_crc indicates to the HW that CRC offload is expected

- On receive skb->ulp_crc indicates to the driver that no CRC errors in the packets' payload

  skb->ulp_crc==0 triggers software PDU CRC calculation

- On receive skb->ulp_ddp indicates to avoid skb_condense which copies data from destination buffer back to SKB.

Enablement

In order to enable the NVMeTCP offload:

```
ethtool -K <device> ulp-ddp-offload on
modprobe nvme-tcp ulp_offload=1
```

Following the enablement, all the NVMeTCP queues/sockets which are running on the device are offloaded.

DDP and Rx CRC Offload Performance Results

on ConnectX 7

The Tests Config

Servers:

Host server: Intel(R) Xeon(R) Platinum 8380 CPU @ 2.30GHz| Kernel: 5.19.0 | HW: ConnectX-7 2x200

Target server Intel(R) Xeon(R) Platinum 8380 CPU @ 2.30GHz | Kernel: 5.19.0 | HW: ConnectX-7 2x200

Target backend: null-device

Topology and Networking:

Back-to-back connectivity, 1x200 Gbps

ipv4, 1500 MSS

Tuning:

Host server offload: not needed.

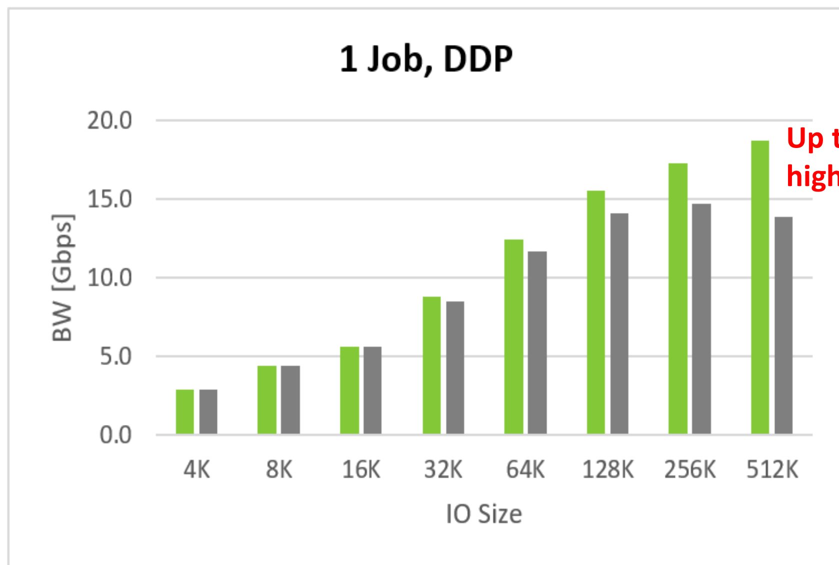
Host server non-offload (“SW”): aRFS, num combined queues = num fio jobs

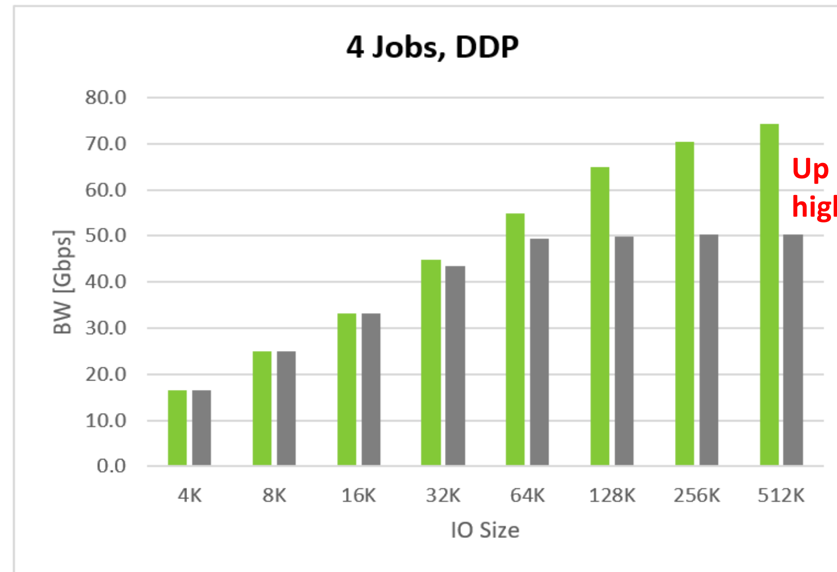
Target server (“SW”): aRFS, num combined queues = num fio jobs

Workload:

fio: num fio jobs = fio cpu allowed

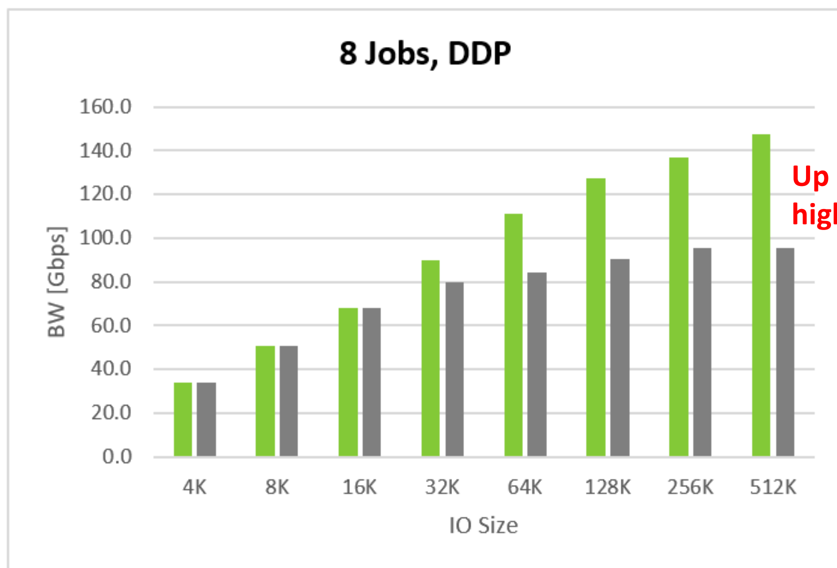
DDP vs SW – Bandwidth Comparison





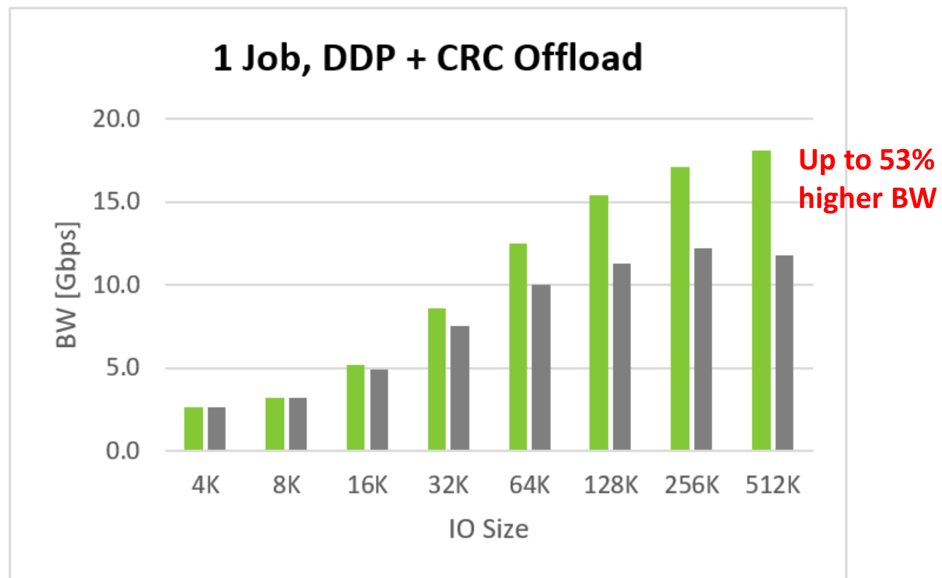
Offload (DDP)

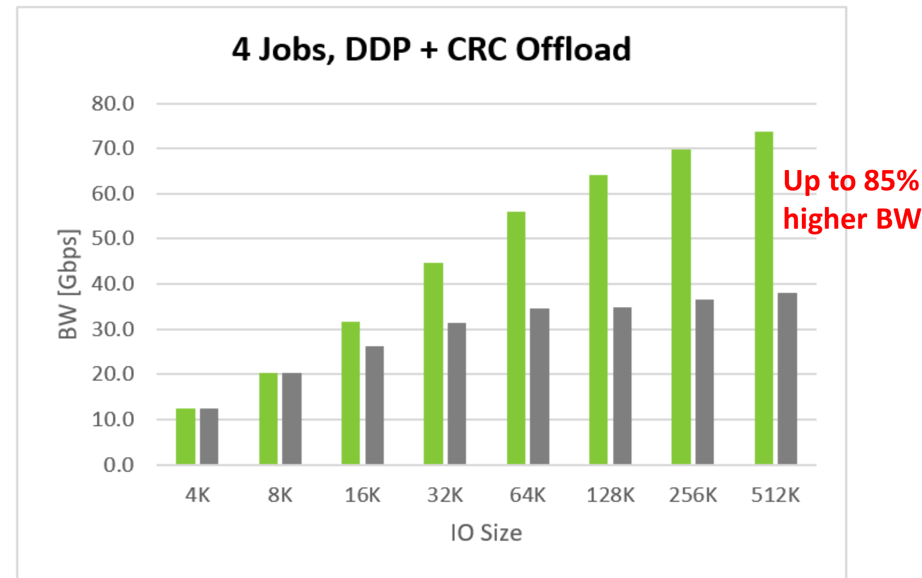
SW




```
fio
--rw=randread --ioengine=libaio
--bs=[x] iodepth=128 --numjobs=[x]
```

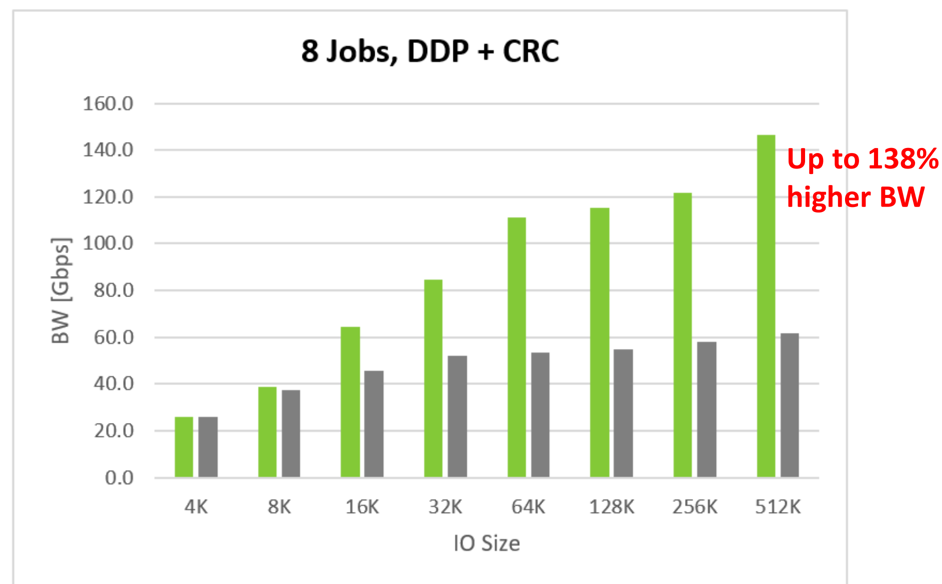
DDP (+ CRC Offload) vs SW (+ SW CRC) – Bandwidth Comparison





Offload (DDP+CRC)

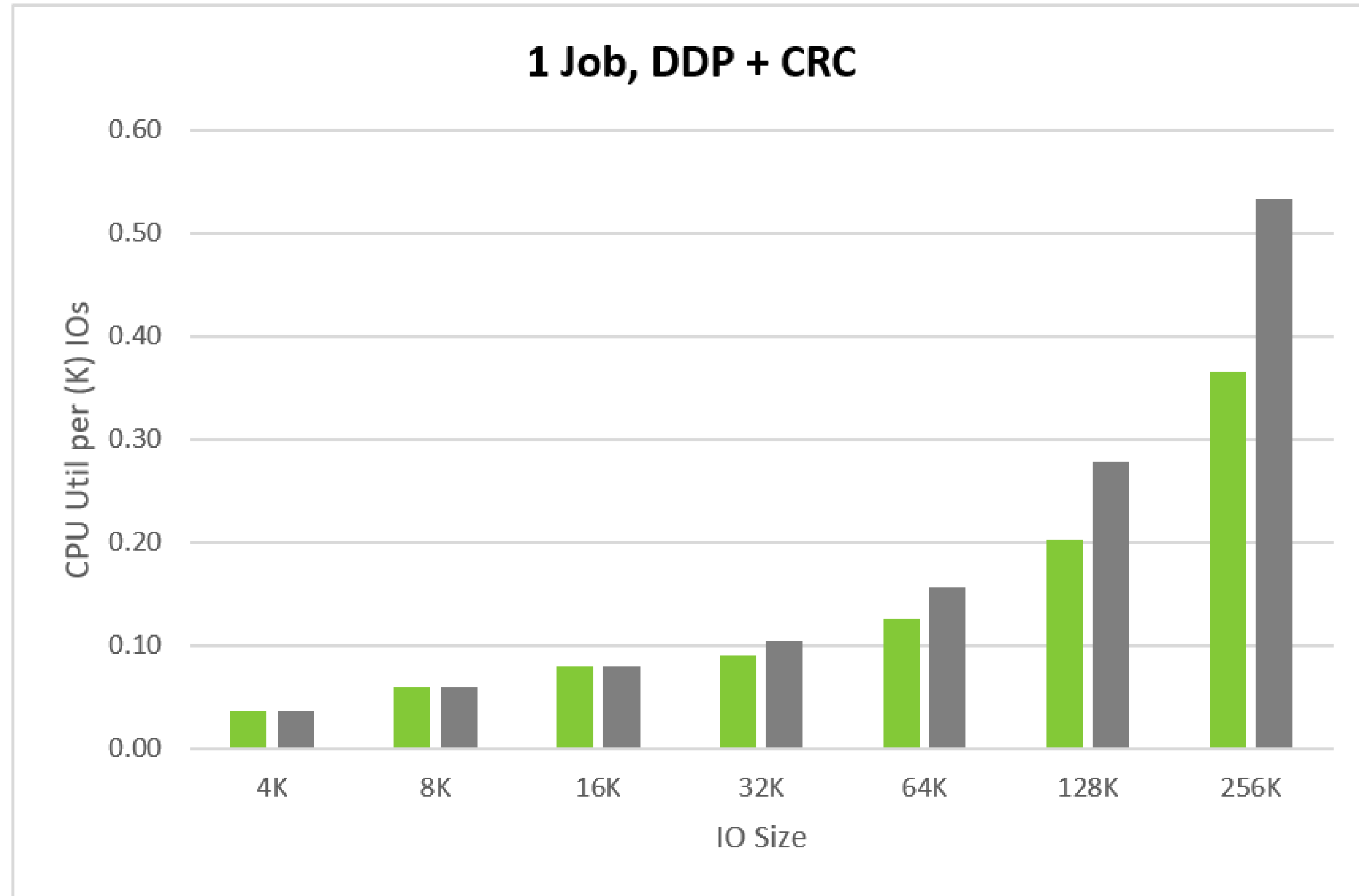
SW



fio
--rw=randread --ioengine=libaio
--bs=[x] iodepth=128 --numjobs=[x]

DDP (+ CRC Offload) vs SW (+ SW CRC) – The CPU Cost Comparison

The comparison of the CPU utilization, normalized per 1000 IOs



The offload shows consistent advantage

Offload (DDP+CRC)

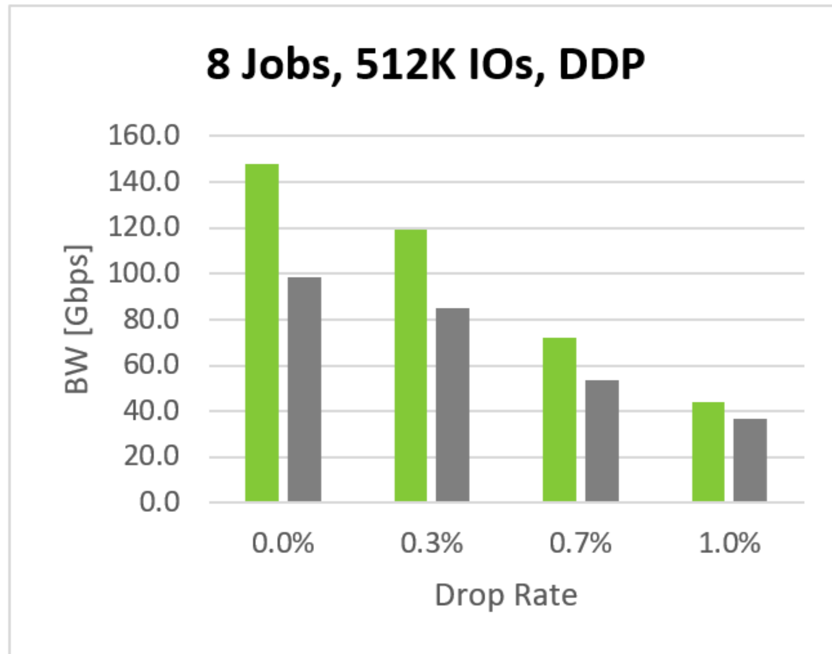
SW

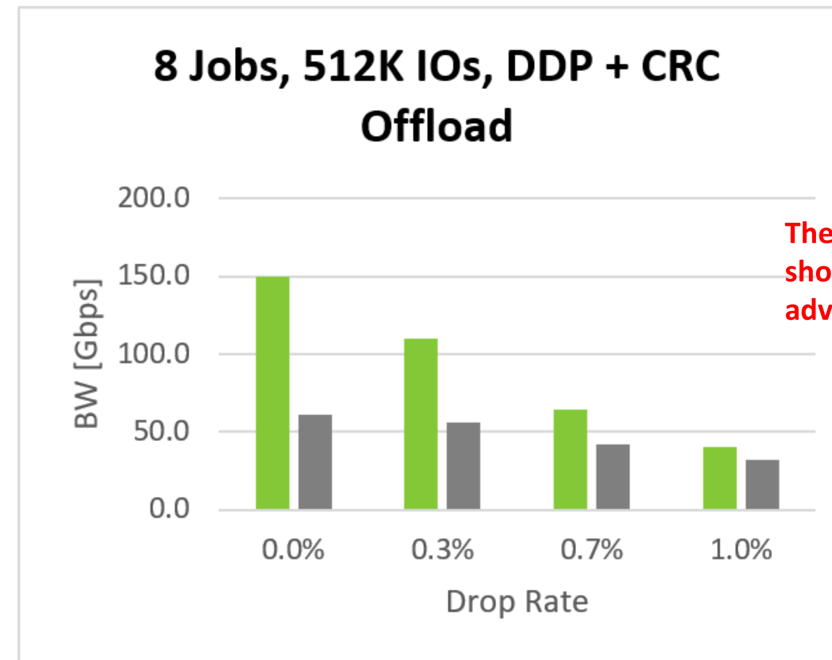
fio
--rw=randread --ioengine=libaio
--bs=[x] iodepth=128 --numjobs=1

Network Congestion

The network congestion was simulated using:

```
tc qdisc add dev eth0 root netem loss []%
```





The offload shows consistent advantage

Offload (DDP)

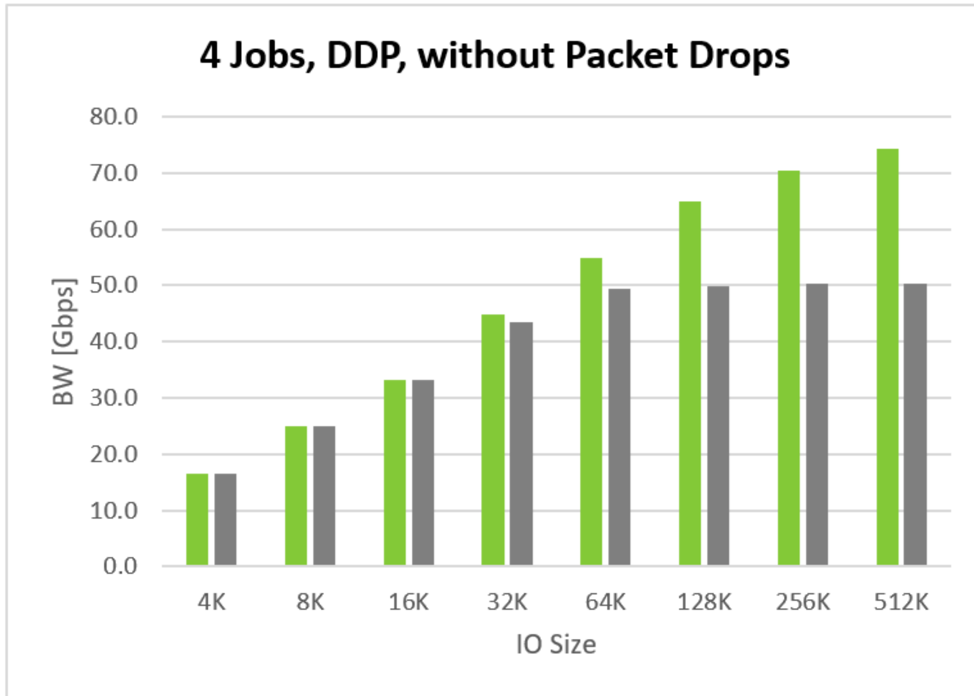
SW

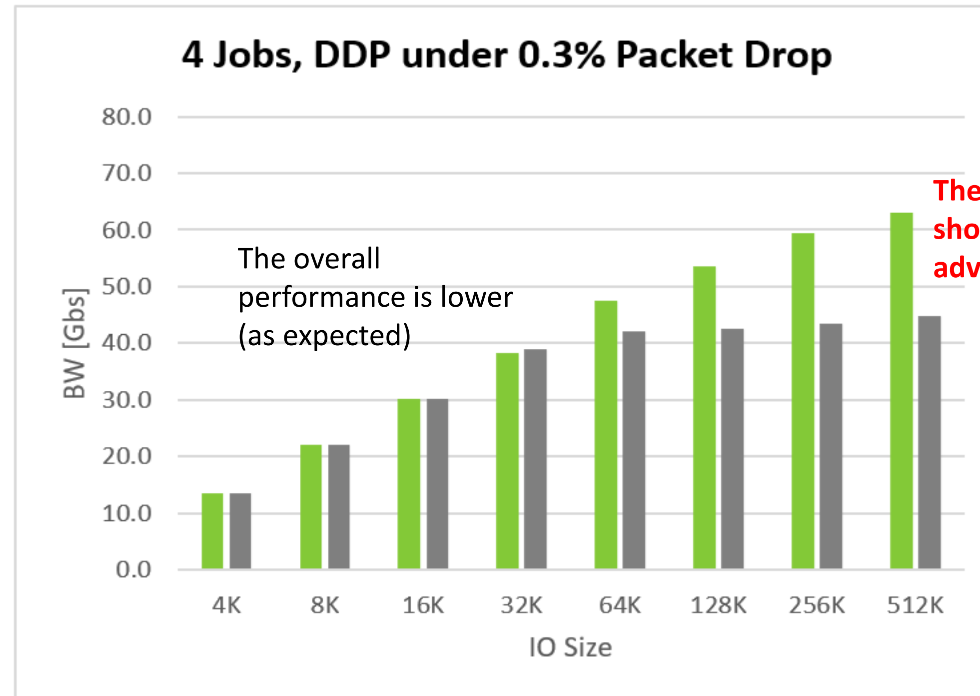
```
fio
--rw=randread --ioengine=libaio
--bs=512k iodepth=128 --numjobs=8
```

DDP vs SW
– Bandwidth Comparison Under Congestion

The network congestion was simulated using:

```
tc qdisc add dev eth0 root netem loss 0.3%
```





The offload shows consistent advantage

Offload (DDP)

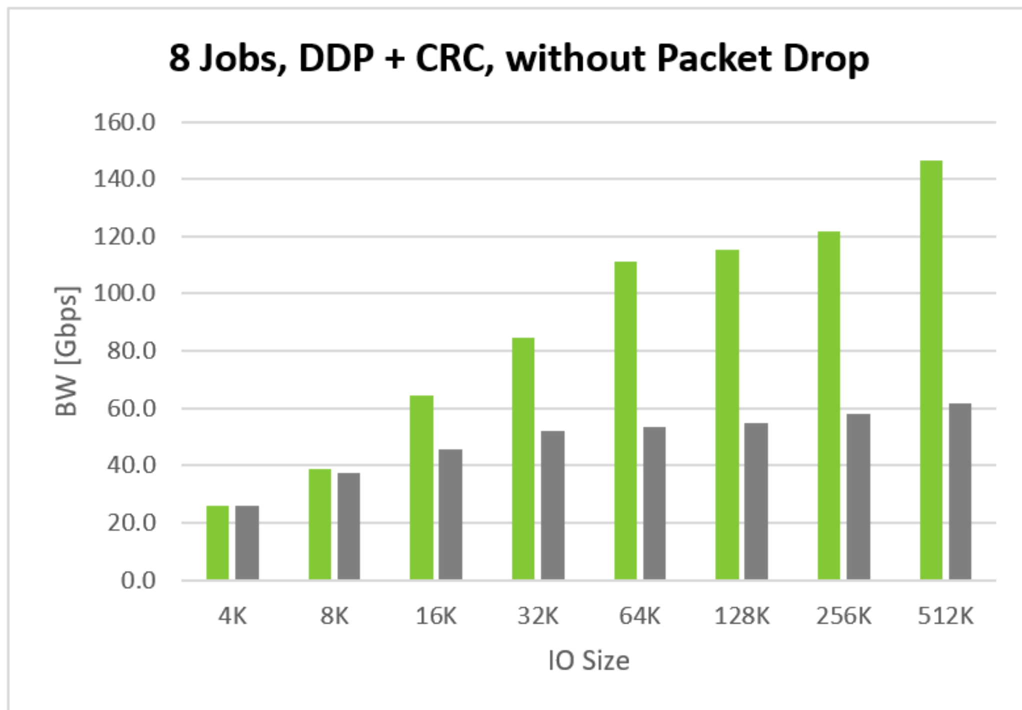
SW

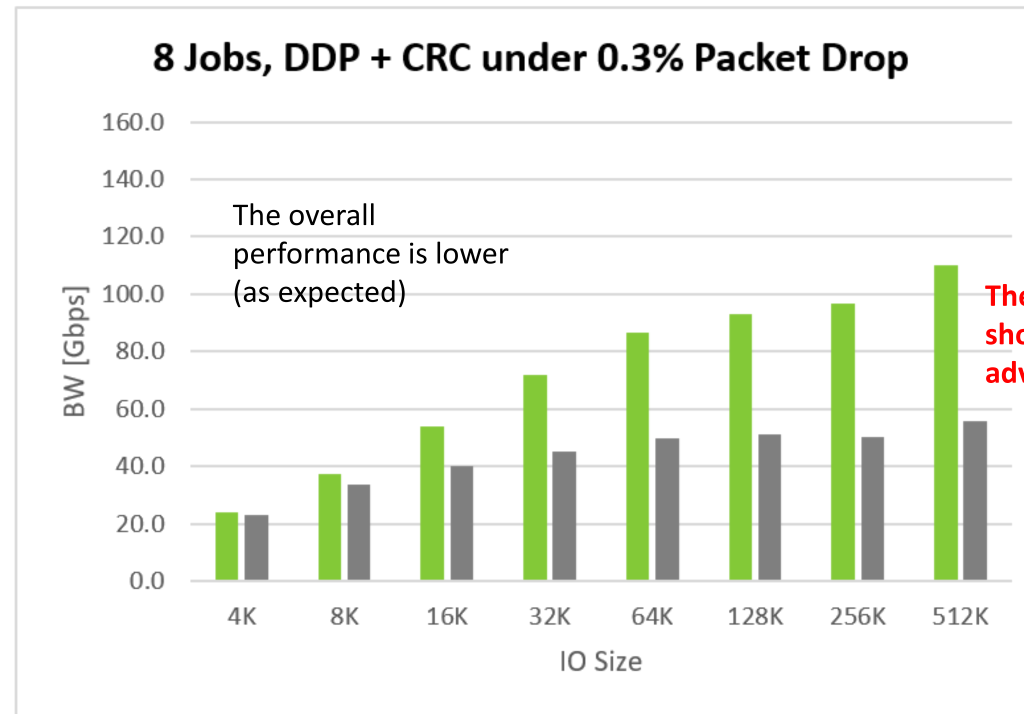
fio
--rw=randread --ioengine=libaio
--bs=[x] iodepth=128 --numjobs=4

DDP (+ CRC Offload) vs SW (+ SW CRC) – Bandwidth Comparison Under Congestion

The network congestion was simulated using:

```
tc qdisc add dev eth0 root netem loss 0.3%
```





Offload (DDP+CRC)

SW

```
fio
--rw=randread --ioengine=libaio
--bs=[x] iodepth=128 --numjobs=8
```

Lessons

Context Switches are Expensive

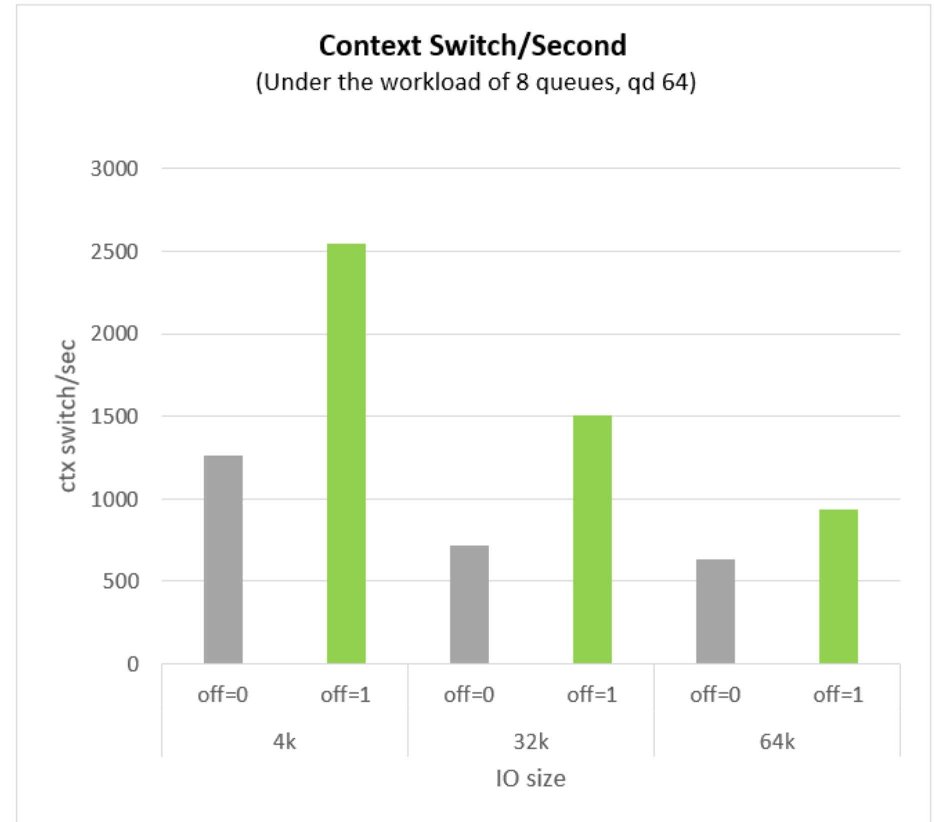
A lot of interruptions resulted in a lot of scheduling and context switching:

- This resulted in a significant overhead.
- Fixed by disabling the notification of UMR completion.
- Because the offload is opportunistic, it doesn’t have to wait for the completion.

Useful tools to for analysis and debugging:

- Perf(1)
- flamegraphs

```
perf record -a -g -F 99 -o data.perf -- fio –bs=64k …
perf script -i data.perf > data.txt
./stackcollapse-perf.pl data.txt > collapsed.txt
./flamegraph.pl --title "fio flamegraph" collapsed.txt > graph.svg
```



GRO Impacts Performances a Lot

- GRO is an optimization to coalesce multiple packets together while receiving.
- The initial method used to track the DDP and CRC offload in SKB affected GRO.
  - GRO should only group packets which have their CRC computed.
  - Initial design used the same bit to track CRC computation and Direct Data placement.
  - Packets group would be flushed early and resulted in more processing (worse performance than non-offloaded).

- Fixed by tracking CRC and DDP independently so as to not split packets early in GRO.

- Useful tools for analysis and debugging
  - Tracked average received packet size via tcpdump and wireshark:

```
tcpdump -i $iface -Q in -s 64 -c 2000 -w net.cap
tshark -r net.cap -q -z 'io,stat,0,MIN(frame.len),MAX(frame.len),AVG(frame.len)frame.len'
```

Upstream status

Upstream status

nvme-tcp-receive offload v6 sent last week to:

- net-next
- linux-nvme

Patch	Series
[v6,23/23] net/mlx5e: NVMEoTCP, statistics	nvme-tcp receive offloads
[v6,22/23] net/mlx5e: NVMEoTCP, data-path for DDP+DDGST offload	nvme-tcp receive offloads
[v6,21/23] net/mlx5e: NVMEoTCP, async ddp invalidation	nvme-tcp receive offloads
[v6,20/23] net/mlx5e: NVMEoTCP, ddp setup and resync	nvme-tcp receive offloads
[v6,19/23] net/mlx5e: NVMEoTCP, queue init/teardown	nvme-tcp receive offloads
[v6,18/23] net/mlx5e: NVMEoTCP, use KLM UMRs for buffer registration	nvme-tcp receive offloads
[v6,17/23] net/mlx5e: TCP flow steering for nvme-tcp acceleration	nvme-tcp receive offloads
[v6,16/23] net/mlx5e: NVMEoTCP, offload initialization	nvme-tcp receive offloads
[v6,15/23] net/mlx5: Add NVMEoTCP caps, HW bits, 128B CQE and enumerations	nvme-tcp receive offloads
[v6,14/23] net/mlx5e: Refactor doorbell function to allow avoiding a completion	nvme-tcp receive offloads
[v6,13/23] net/mlx5e: Have mdev pointer directly on the icosq structure	nvme-tcp receive offloads
[v6,12/23] net/mlx5e: Refactor ico sq polling to get budget	nvme-tcp receive offloads
[v6,11/23] net/mlx5e: Rename from tls to transport static params	nvme-tcp receive offloads
[v6,10/23] Documentation: add ULP DDP offload documentation	nvme-tcp receive offloads
[v6,09/23] nvme-tcp: Add ulp_offload modparam to control enablement of ULP offload	nvme-tcp receive offloads
[v6,08/23] nvme-tcp: Deal with netdevice DOWN events	nvme-tcp receive offloads
[v6,07/23] nvme-tcp: RX DDGST offload	nvme-tcp receive offloads
[v6,06/23] nvme-tcp: Add DDP data-path	nvme-tcp receive offloads
[v6,05/23] nvme-tcp: Add DDP offload control path	nvme-tcp receive offloads
[v6,04/23] Revert "nvme-tcp: remove the unused queue_size member in nvme_tcp_queue"	nvme-tcp receive offloads
[v6,03/23] net/tls: export get_netdev_for_sock	nvme-tcp receive offloads
[v6,02/23] iov_iter: DDP copy to iter/pages	nvme-tcp receive offloads
[v6,01/23] net: Introduce direct data placement tcp offload	nvme-tcp receive offloads

Questions