Accelerating Packet Processing with GRO and GSO in DPDK
A major part of packet processing “applications” are performed on a per-packet basis (e.g. Firewall, TCP/IPv4 stack)

Per-packet routines (e.g. header processing) dominate packet processing overheads

Reduce the packet number to be processed can mitigate the per-packet overhead
Methods to Reduce the Packet Number

- Use large Maximum Transmission Unit (MTU)
  - MTU size depends on physical links
- Use network interface card (NIC) features: Large Receive Offload (LRO), TCP Segmentation Offload (TSO) and UDP Fragmentation Offload (UFO)

**Drawbacks:**
- Only support TCP and UDP
- Not all NICs support LRO/TSO/UFO
GRO and GSO

Software Method:

• Generic Receive Offload (GRO) merges small packets into larger ones
• Generic Segmentation Offload (GSO) splits large packets into smaller ones

Advantages:

• Don’t reply on physical link support
• Don’t reply on NIC hardware support
• Able to support various kinds of protocols, like TCP, VxLAN and GRE
In DPDK, GRO and GSO are two standalone libraries.
GRO API
IPv4-GRE-TCP/IPv4 GRO
- Packet type: IPv4-GRE-IPv4-TCP
- One GRO type, one kind of packets
- Indicated by MBUF->packet_type

TCP/IPv4 GRO
- Packet type: IPv4-IPv4

GSO API
IPv4-GRE-TCP/IPv4 GSO
- ol_flags: PKT_TX_TCP_SEG | PKT_TX_IPV4 | PKT_TX_OUTER_IPV4 | PKT_TX_TUNNEL_GRE
- One GSO type, one kind of packets
- Indicated by MBUF->ol_flags

TCP/IPv4 GSO
- ol_flags: PKT_TX_TCP_SEG | PKT_TX_IPV4

rte_gro_ctx_create()
rte_gro_ctx_destroy()
rte_gro_reassemble...
rte_gro_timeout_flush()
rte_gro_get_pkt_count()
### API: How Applications Use it?

<table>
<thead>
<tr>
<th>GRO Sample</th>
<th>GSO Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>nb_pkts = rte_eth_rx_burst(…, pkts, …);</td>
<td>nb_segs = rte_gro_reassemble_burst(pkts, nb_pkts, …);</td>
</tr>
<tr>
<td></td>
<td>rte_eth_tx_burst(…, pkts, nb_segs);</td>
</tr>
<tr>
<td>struct rte_mbuf *gso_segs[N];</td>
<td>nb_segs = rte_gso_segment(…, pkt, gso_segs, …);</td>
</tr>
<tr>
<td></td>
<td>rte_eth_tx_burst(…, gso_segs, nb_segs);</td>
</tr>
</tbody>
</table>
API: Two Sets in GRO

Lightweight Mode API

- Support to merge a **small number** of packets **rapidly**

```
rte_gro_reassemble_burst()
```

Heavyweight Mode API

- Supports to merge a **large number** of packets with **fine-grained control**

```
ctx = rte_gro_ctx_create()
rte_gro_reassemble(pkts, ctx)
rte_gro_timeout_flush(ctx)
```
How to Merge and Split Packets?

- GRO Reassembly Algorithm merges packets
- GSO Segmentation Scheme splits packets
GRO Algorithm: Challenges

- A high cost algorithm/implementation would cause packet dropping in a high speed network

  **Algorithm is lightweight to scale fast networking speed**

- Packet reordering makes it hard to merge packets
  - Linux GRO fails to merge packets when encounters packet reordering

  **Algorithm is capable of handling packet reordering**
GRO Algorithm: Key-based Approach

TCP/IPv4 Packets

Header fields representing a “flow”:
- src/dst: mac, IP, port same
- ACK number value

Header fields deciding “neighbor”:
- IP id incremental
- Sequence number

Categorize into an existed “flow”

Search for a “neighbor” in the “flow”

Merge the packets

not find

• Insert a new “flow”
• Store the packet

not find

Store the packet in the “flow”
**Two Characters**

- **Lightweight**: classify packets into “flows” to accelerate packet aggregation is simple
- **More**: storing out-of-order packets makes it possible to merge later

**Address challenge 1 and 2**

1. **Categorize into an existed “flow”**
2. **Search for a “neighbor” in the “flow”**
3. **Merge the packets**

- **not find**
  - **Insert a new “flow”**
  - **Store the packet**

- **find a “flow”**
  - **not find**
  - **not find**
    - **Store the packet in the “flow”**
GRO Algorithm: TCP/IPv4 Example

Flow | MAC | IP | Port | ACK Number
--- | --- | --- | --- | ---
**F1** | 1:2:3:4:5:6 11:22:33:44:55:66 | 1.1.1.1 | 1.1.1.2 | 5041 5043 | 1
**F2** | 1:2:3:4:5:6 11:22:33:44:55:66 | 1.1.1.1 | 1.1.1.2 | 5001 5002 | 1

Packet | IP ID | Sequence number | Payload length
--- | --- | --- | ---
P1 | 7 | 701 | 1500
P0 | 1 | 1 | 100
P2 | 3 | 301 | 100

#DPDKSummit
### GRO Algorithm: TCP/IPv4 Example

#### Flow Table

<table>
<thead>
<tr>
<th>Flow</th>
<th>MAC</th>
<th>IP</th>
<th>Port</th>
<th>ACK Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>1:2:3:4:5:6</td>
<td>1.1.1.1</td>
<td>1.1.1.2</td>
<td>5041 5043 1</td>
</tr>
<tr>
<td></td>
<td>11:22:33:44:55:66</td>
<td>1.1.1.1</td>
<td>1.1.1.2</td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td>1:2:3:4:5:6</td>
<td>1.1.1.1</td>
<td>1.1.1.2</td>
<td>5001 5002 1</td>
</tr>
<tr>
<td></td>
<td>11:22:33:44:55:66</td>
<td>1.1.1.1</td>
<td>1.1.1.2</td>
<td></td>
</tr>
</tbody>
</table>

#### Payload Table

<table>
<thead>
<tr>
<th>Flow</th>
<th>IP ID</th>
<th>Sequence number</th>
<th>Payload length</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2</td>
<td>4</td>
<td>401</td>
<td>100</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Packets</th>
<th>IP ID</th>
<th>Sequence number</th>
<th>Payload length</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
<tr>
<td>P2</td>
<td>3</td>
<td>301</td>
<td>100</td>
</tr>
</tbody>
</table>

- IP ID and sequence number are incremental → Neighbors
### GRO Algorithm: TCP/IPv4 Example

<table>
<thead>
<tr>
<th>Flow</th>
<th>MAC</th>
<th>IP</th>
<th>Port</th>
<th>ACK Number</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>f0</td>
<td>1:2:3:4:5:6</td>
<td>1.1.1.1:2</td>
<td>5041:5043</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>f1</td>
<td>1:2:3:4:5:6</td>
<td>1.1.1.1:2</td>
<td>5001:5002</td>
<td>1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Packets</th>
<th>IP ID</th>
<th>Sequence number</th>
<th>Payload length</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>7</td>
<td>701</td>
<td>1500</td>
</tr>
<tr>
<td>P2</td>
<td>4</td>
<td>301</td>
<td>200</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Packets</th>
<th>IP ID</th>
<th>Sequence number</th>
<th>Payload length</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>1</td>
<td>1</td>
<td>100</td>
</tr>
</tbody>
</table>

GRO Algorithm: TCP/IPv4 Example
GSO Scheme: Overview

Segmentation Workflow

- **GSO Segments**
  - Split the payload into smaller parts
  - Add the packet header to each payload part
  - Update packet headers for all GSO segments

How to organize a GSO segment?
“Two-part” MBUF is used to organize a GSO segment

- Direct MBUF holds the packet header
- Indirect MBUF: "pointer", point to a payload part of the packet to segment

Forming a GSO segment just needs to copy the header!
GSO Scheme: Example

1. **Copy** HDR to direct MBUF
2. “Attach” indirect MBUF to PKT and make it point to the correct payload part
3. **Reduce** reference counter of PKT’s MBUF by 1
   - PKT will be freed automatically, when all indirect MBUFS are freed
Experiment: Physical Topology

Host Kernel

NIC 0

Switch (testpmd)

NIC 1

VM

virtio-net

vhost-user

Server 0

iperf

TCP/IPv4 packets

logically connected

Physically connected

#DPDKSummit
Experiment: Setup

**GRO**

- **iperf client**
  - NIC 0
  - Host Kernel
  - Small TCP/IPv4 packets
- **Testpmd**
  - NIC 1
- **virtio-net**
  - vhost-user
- **Linux GRO**
- **DPDK GRO**

**GSO**

- **iperf server**
  - NIC 0
  - VM
  - large TCP/IPv4 packets
- **Testpmd**
  - NIC 1
- **virtio-net**
  - vhost-user
- **Linux GSO**
- **DPDK GSO**

**Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz**

Ethernet Controller XL710 for 40GbE QSFP+
Experiment: GRO Performance

Figure 1. Iperf throughput Improvement of DPDK-GRO over No-GRO

Figure 2. Iperf throughput Improvement of DPDK-GRO over Linux-GRO
Figure 1. Iperf throughput Improvement of DPDK-GSO over No-GSO

Figure 2. Iperf throughput Improvement of DPDK-GSO over Linux-GSO
Thanks!

Jiyu Hu
jiayu.hu@intel.com