Revisiting checksum offloads in OVS

David Marchand

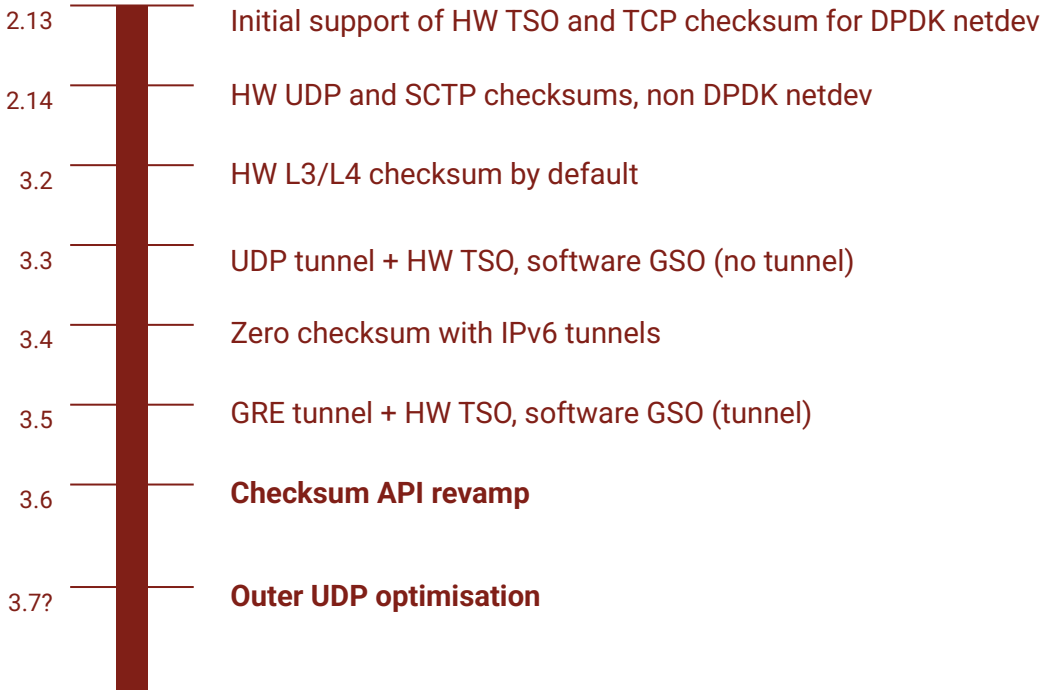
Network Services Team / Fast Datapath

Red Hat

Agenda

1. Introduction
2. OVS legacy API
3. Revamped OVS API
4. Outer UDP checksum
5. Next steps

Introduction: it was a long road



OVS legacy API

OVS processing until 3.2

OVS legacy API is mapped on DPDK offload API.
OVS tracks IP version during flow extraction miniflow_extract() and sets **Tx** flags:

```
void
miniflow_extract(struct dp_packet *packet, struct miniflow *dst)
{
  ...
  dp_packet_hwol_set_tx_ipv4(packet);
  if (dp_packet_ip_checksum_good(packet)) {
    dp_packet_hwol_set_tx_ip_csum(packet);
  }
  ...
  if (dp_packet_l4_checksum_good(packet)
      || dp_packet_ol_l4_csum_partial(packet)) {
    dp_packet_hwol_set_csum_tcp(packet);
  }
```

For DPDK drivers,
Rx → Tx

OVS processing 3.3

As DPDK API is centered around a “outer” Tx flags API, encapsulating meant marking packets with “outer” IP versions, and “outer” checksums.
Evaluating the **Tx** flags during packet processing became complex:

```
static inline bool
dp_packet_hwol_l3_ipv4(const struct dp_packet *b)
{
    if (dp_packet_hwol_is_outer_ipv4(b)) {
        return true;
    } else if (!dp_packet_hwol_is_outer_ipv6(b)) {
        return dp_packet_hwol_tx_ip_csum(b);
    }
    return false;
}
```

OVS processing 3.3 (cont)

On the other hand, OVS was tracking offsets in the packet with “current” offsets and “inner” offsets (after encapsulation).

This contradicted the logic on outer **Tx** flags and ended up with some crash during packet recirculation.
This bug was resolved by making GSO unavailable in this case.

1c1f173ce8a8 ("dpif-netdev: Fix crash due to tunnel offloading on recirculation.")

This also made some use cases hard to support, like if some vhost-user sent encapsulated traffic.

Revamped OVS API

Enhancing unit tests

Before touching the API, coverage was increased, starting to separate Rx and Tx concerns in the dummy netdev

```
userspace offload - tcp csum offload (simple match) ok
userspace offload - tcp csum offload            ok
userspace offload - udp csum offload            ok
userspace offload - IPv6 tcp csum offload       ok
userspace offload - IPv6 udp csum offload       ok
userspace offload - conntrack                   ok
```

Enhancing unit tests (cont)

Coverage counters were added.
Those can help understand checksum related issues.

- Conntrack validating checksums: **conntrack_l3/l4csum_checked** (in addition to existing _err counters)
- IP fragmentation module validating IP checksum: **ipf_l3csum_checked** (in addition to existing _err counter)
- Tunnel decapsulation: **native_tnl_l3/l4csum_checked** and **native_tnl_l3/l4csum_err**

Rx status flags

A first change for the new API was to isolate the Rx status into a new **dp_packet::offloads** field.
This new field is **32 bits** wide, with the lower 16 bits applying to the "**outermost**" headers.

Moving bits from DPDK mbuf to OVS dp_packet has been made easier by mapping OVS on DPDK mbuf bits.

```
DP_PACKET_OL_IP_CKSUM_BAD == RTE_MBUF_F_RX_IP_CKSUM_BAD
DP_PACKET_OL_IP_CKSUM_GOOD == RTE_MBUF_F_RX_IP_CKSUM_GOOD
DP_PACKET_OL_L4_CKSUM_BAD == RTE_MBUF_F_RX_L4_CKSUM_BAD
DP_PACKET_OL_L4_CKSUM_GOOD == RTE_MBUF_F_RX_L4_CKSUM_GOOD
```

Tunnel

Tunnel flags are now a status metadata, populated in dp_packet::offloads when encapsulating (and dropped when decapsulating, see later explanation with IP checksum).
It is netdev-* responsibility to convert, netdev-dpdk for example converts this to RTE_MBUF_F_TX_TUNNEL* bits.

```
static bool
netdev_dpdk_prep_hwol_packet(struct netdev_dpdk *dev, struct rte_mbuf *mbuf)
{
  ...
  if (dp_packet_tunnel_geneve(pkt)) {
    mbuf->ol_flags |= RTE_MBUF_F_TX_TUNNEL_GENEVE;
  } else if (dp_packet_tunnel_vxlan(pkt)) {
    mbuf->ol_flags |= RTE_MBUF_F_TX_TUNNEL_VXLAN;
  ...
```

IP checksums

A dp_packet IP checksum has now four status, like in DPDK mbuf API:

- "**unknown**": no information about the IP checksum
- "**bad**": the IP checksum set in the packet (when it was first received) is wrong. As stated in OVS documentation, OVS must *not* fix it
- "**good**": the IP checksum in the packet is valid
- "**partial**": the IP checksum is not correct in the packet data, but the integrity of the IP header was verified and OVS *must* fix it

This status is that of the "**outermost**" IP header when considering a dp_packet.

IP checksums (cont)

Flow extraction now does not care about which (outer?) IP checksum to request:

```
@@ -949,15 +949,6 @@ miniflow_extract(struct dp_packet *packet, struct miniflow *dst)
         nw_proto = nh->ip_proto;
         nw_frag = ipv4_get_nw_frag(nh);
         data_pull(&data, &size, ip_len);
-        if (tunneling) {
-            if (dp_packet_ip_checksum_good(packet)) {
-                dp_packet_hwol_set_tx_outer_ipv4_csum(packet);
-            }
-        } else {
-            if (dp_packet_ip_checksum_good(packet)) {
-                dp_packet_hwol_set_tx_ip_csum(packet);
-            }
-        }
```

IP checksums (cont)

During tunnel encapsulation, IP checksum status flags are pushed to the upper 16 bits in dp_packet::offloads.

```
#define DP_PACKET_OL_SHIFT_COUNT 16
    /* Inner offloads. */
    DP_PACKET_OL_INNER_IP_CKSUM_BAD =
        DP_PACKET_OL_IP_CKSUM_BAD << DP_PACKET_OL_SHIFT_COUNT,
    DP_PACKET_OL_INNER_IP_CKSUM_GOOD =
        DP_PACKET_OL_IP_CKSUM_GOOD << DP_PACKET_OL_SHIFT_COUNT,
```

IP checksums (cont)

And like for Tunnel flags, it is now netdev-* responsibility to convert from OVS IP checksum status to the right implementation.
Only the “partial” status matters.

L4 checksums

The approach is similar to IP, with the same four status:

- "**unknown**": no information about the L4 checksum
- "**bad**": the L4 checksum set in the packet (when it was first received) is wrong.
  As stated in OVS documentation, OVS must *not* fix it
- "**good**": the L4 checksum in the packet is valid
- "**partial**": the L4 checksum is not correct in the packet data, but the integrity of the L4 header was verified and OVS *must* fix it

This status is that of the "**outermost**" L4 header when considering a dp_packet.

L4 checksums (cont)

The implications on flow extraction, checksum status and update or encapsulation of the packet are the same than for IP checksum. Yet L4 protocol must be tracked.

```
@@ -1063,10 +1061,7 @@ miniflow_extract(struct dp_packet *packet, struct miniflow *dst)
                     } else if (dl_type == htons(ETH_TYPE_IPV6)) {
                         dp_packet_update_rss_hash_ipv6_tcp_udp(packet);
                     }
-                    if (dp_packet_l4_checksum_good(packet)
-                        || dp_packet_ol_l4_csum_partial(packet)) {
-                        dp_packet_hwol_set_csum_tcp(packet);
-                    }
+                    dp_packet_l4_proto_set_tcp(packet);
                 }
```

TCP segmentation

TCP segmentation is now only a matter of setting the **dp_packet::tso_segsz** field to a non 0 value.

Having a Tx flag was not helping in taking a decision in OVS.

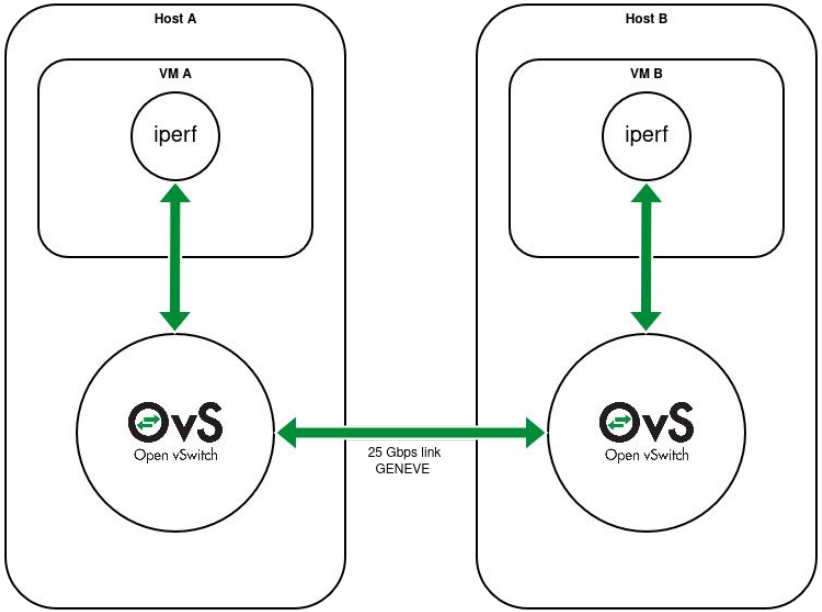
Impact of the series

This rework has moved netdev-* specific logics out of OVS common processing flow.

There was a slight cost increase identified in netdev-dpdk.c helper that sets DPDK **Tx** flags, which has been addressed by handling IP checksum updates in OVS itself.

```
ab062d3cb4b6 ("netdev-dpdk: Adjust IPv4 checksum capability for vhost-user.")
```

Tunnel performance



Tunnel performance

CX6-Dx (no outer csum):

```
IPv4/IPv4 [  5]   0.00-1.00   sec  1.45 GBytes  12.5 Gbits/sec  625    600 KBytes
IPv4/IPv6 [  5]   0.00-1.00   sec  1.27 GBytes  10.9 Gbits/sec  210   1.16 MBytes
```

X710 (no outer csum):

```
IPv4/IPv4 [  5]   0.00-1.00   sec  1.45 GBytes  12.4 Gbits/sec 1031    459 KBytes
IPv4/IPv6 [  5]   0.00-1.00   sec  1.19 GBytes  10.2 Gbits/sec  140   1.51 MBytes
```

E810 (no outer csum):

```
IPv4/IPv4 [  5]   0.00-1.00   sec  1.23 GBytes  10.6 Gbits/sec   28   1.57 MBytes
IPv4/IPv6 [  5]   0.00-1.00   sec  1.16 GBytes  9.98 Gbits/sec  101   2.29 MBytes
```

CX6-Dx:

```
IPv4/IPv4 [  5]   0.00-1.00   sec   768 MBytes  6.44 Gbits/sec    0   3.11 MBytes
IPv4/IPv6 [  5]   0.00-1.00   sec   747 MBytes  6.27 Gbits/sec    0   3.07 MBytes
```

X710:

```
IPv4/IPv4 [  5]   0.00-1.00   sec   622 MBytes  5.22 Gbits/sec    0   2.24 MBytes
IPv4/IPv6 [  5]   0.00-1.00   sec   637 MBytes  5.34 Gbits/sec    0   2.44 MBytes
```

E810:

```
IPv4/IPv4 [  5]   0.00-1.00   sec  1.16 GBytes  9.94 Gbits/sec   86   2.32 MBytes
IPv4/IPv6 [  5]   0.00-1.00   sec  1.16 GBytes  9.93 Gbits/sec   93   2.29 MBytes
```

Outer UDP checksum

More unit tests!

There still exists a gap in tunneling + checksum coverage.

```
1104: dpif-netdev - IPv4 vxlan tunnel (no csum) - no offload ok
1105: dpif-netdev - IPv4 vxlan tunnel (no csum) - ip csum offload ok
1106: dpif-netdev - IPv4 vxlan tunnel (no csum) - tcp csum offload ok
1107: dpif-netdev - IPv4 vxlan tunnel (no csum) - udp csum offload ok
1108: dpif-netdev - IPv4 vxlan tunnel (no csum) - IPv6 tcp csum offload ok
1109: dpif-netdev - IPv4 vxlan tunnel (no csum) - IPv6 udp csum offload ok
1110: dpif-netdev - IPv4 vxlan tunnel - no offload ok
1111: dpif-netdev - IPv4 vxlan tunnel - ip csum offload ok
1112: dpif-netdev - IPv4 vxlan tunnel - tcp csum offload ok
1113: dpif-netdev - IPv4 vxlan tunnel - udp csum offload ok
1114: dpif-netdev - IPv4 vxlan tunnel - IPv6 tcp csum offload ok
1115: dpif-netdev - IPv4 vxlan tunnel - IPv6 udp csum offload ok
...
```

https://patchwork.ozlabs.org/project/openvswitch/patch/20251112170420.3155127-7-david.marchand@redhat.com/

We don’t need HW outer UDP checksum…

For inner UDP traffic with no checksum, or for unsupported inner L3/L4 protocols (like ARP), we don’t need HW outer UDP checksum. The inner payload can be treated as opaque.

With the revamped API, it is as simple as:

```
p->offloads &= ~DP_PACKET_OL_TUNNEL_MASK;
```

https://patchwork.ozlabs.org/project/openvswitch/patch/20251112170420.3155127-9-david.marchand@redhat.com/

… but sometimes we do … ?

Like for HW TSO:

```
    0x0000:  043f 72b2 c090 043f 72b2 c084 0800 4500  .?r....?r.....E.
    0x0010:  0076 0000 4000 4011 8af7 c0a8 1718 c0a8  .v..@.@.........
    0x0020:  1717 c65b 17c1 0062 3f4a 0240 6558 0000  ...[...b?J.@eX..
--
    0x0000:  043f 72b2 c090 043f 72b2 c084 0800 4500  .?r....?r.....E.
    0x0010:  006e 0000 4000 4011 8aff c0a8 1718 c0a8  .n..@.@.........
    0x0020:  1717 c65b 17c1 005a 3f52 0240 6558 0000  ...[...Z?R.@eX..
--
    0x0000:  043f 72b2 c090 043f 72b2 c084 0800 4500  .?r....?r.....E.
    0x0010:  05b2 0000 4000 4011 85bb c0a8 1718 c0a8  ....@.@.........
    0x0020:  1717 c65b 17c1 059e 3a0e 0240 6558 0000  ...[....:..@eX..
--
    0x0000:  043f 72b2 c090 043f 72b2 c084 0800 4500  .?r....?r.....E.
    0x0010:  05b2 0001 4000 4011 85ba c0a8 1718 c0a8  ....@.@.........
    0x0020:  1717 c65b 17c1 059e 3a0e 0240 6558 0000  ...[....:..@eX..
```

0x0076 + 0x3f4a == **0x3fc0**, 0x006e + 0x3f52 == **0x3fc0**, 0x05b2 + 0x3a0e == **0x3fc0** 😳

Outer UDP checksum optimisation

A TCP encapsulated packet looks like:

Ether	IPv4	UDP	Geneve	Ether	IPv4	TCP	Data

Outer UDP checksum:

csum | UDP | Geneve | Ether | IPv4 | TCP | Data |

For valid inner IPv4, csum | IPv4 | == 0.

And for valid inner TCP, csum | TCP | Data | == 0.

Summary: outer UDP checksum is csum | UDP | Geneve | Ether | 

https://patchwork.ozlabs.org/project/openvswitch/patch/20251112170420.3155127-10-david.marchand@redhat.com/

Outer UDP checksum optimisation (cont)

Performance enhancement, with a mlx5 CX-6 Dx nic:

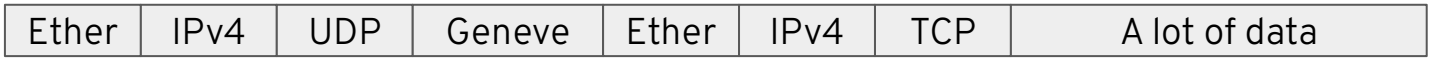
- before: **6.08** Gbits/sec, 100% cpu
- after: **7.31** Gbits/sec, 100% cpu

We are still relying on software GSO:

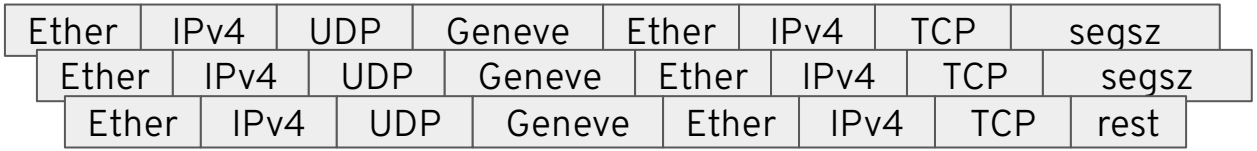
```
# ovs-appctl coverage/show | grep -E 'seg|csum'
netdev_soft_seg_good     14302.8/sec  1675.500/sec       27.9250/sec   total: 125485
```

Hardware segmentation: TSO

What does TCP segmentation mean? For a packet like



Once segmentation happened, the transmitted packets are:



Hardware must update inner TCP sequence number, inner TCP checksum, inner IPv4 length, inner IP id, inner IPv4 checksum, and outer IPv4 length, outer IP id, outer IPv4 checksum.

Outer UDP checksum is an additional problem.

Hardware segmentation: TSO (cont)

Applying the outer UDP optimisation requires to predict the lengths of the final transmitted segments

Ether	IPv4	UDP	Geneve	Ether	IPv4	TCP	A lot of data

achieved by splitting the TSO packet in one big and one small packet like:

Ether	IPv4	UDP	Geneve	Ether	IPv4	TCP	multiple of segsz

Ether	IPv4	UDP	Geneve	Ether	IPv4	TCP	rest

In the current series, this is done with a new helper around the GSO software code, and called “partial” segmentation.
The outer UDP optimisation is then applied on both of them, and the big packet can be HW TSO’d.

https://patchwork.ozlabs.org/project/openvswitch/patch/20251112170420.3155127-12-david.marchand@redhat.com/

Hardware segmentation: TSO (cont)

Performance enhancement, with a mlx5 CX-6 Dx nic:

- before: **7.3** Gbits/sec, **100%** cpu
- after: **10.4** Gbits/sec, **37%** cpu

Some “partial” segmentation happens (to be investigated):

```
# ovs-appctl coverage/show | grep -E 'seg|csum'
netdev_partial_seg_good    0.0/sec    53.800/sec        0.8967/sec   total: 3228
```

Next steps?

- Better debugging, especially at the netdev-dpdk “border” with DPDK
- Are there use cases missing in OVS?
- Time to mark this feature non experimental?

Thank you!

Questions?


linkedin.com/company/red-hat


youtube.com/user/RedHatVideos


facebook.com/redhatinc


twitter.com/RedHat

Red Hat

Annex

Vhost-user library packet reception (legacy mode)

```
if (hdr->flags == VIRTIO_NET_HDR_F_NEEDS_CSUM) {
switch (hdr->csum_offset) {
  case (offsetof(struct rte_tcp_hdr, cksum)):
    m->ol_flags |= RTE_MBUF_F_TX_TCP_CKSUM;
    break;
}
if (hdr->gso_type != VIRTIO_NET_HDR_GSO_NONE) {
  switch (hdr->gso_type & ~VIRTIO_NET_HDR_GSO_ECN) {
    case VIRTIO_NET_HDR_GSO_TCPV4:
    case VIRTIO_NET_HDR_GSO_TCPV6:
      m->ol_flags |= RTE_MBUF_F_TX_TCP_SEG;
      m->tso_segsz = hdr->gso_size;
      m->l4_len = tcp_len;
      break;
  }
}
```

No Rx flag

“Nice” for applications that don’t want to set Tx flags, but applications still need to configure HW accordingly

DPDK driver reception

Example from net/ice rx handler:

```
if (unlikely(stat_err0 & (1 << ICE_RX_FLEX_DESC_STATUS0_XSUM_IPE_S)))
  flags |= RTE_MBUF_F_RX_IP_CKSUM_BAD;
else
  flags |= RTE_MBUF_F_RX_IP_CKSUM_GOOD;

if (unlikely(stat_err0 & (1 << ICE_RX_FLEX_DESC_STATUS0_XSUM_L4E_S)))
  flags |= RTE_MBUF_F_RX_L4_CKSUM_BAD;
else
  flags |= RTE_MBUF_F_RX_L4_CKSUM_GOOD;
```

Doing the minimum, only providing Rx status

No Tx flag

OVS processing until 3.2 (cont)

During processing, a modification on the packet data meant conditional handling based on the **Tx** flags and updates of **Rx** flags, like in packet_set_tcp_port():

```
void
packet_set_tcp_port(struct dp_packet *packet, ovs_be16 src, ovs_be16 dst)
{
  ...
  if (dp_packet_hwol_l4_is_tcp(packet)) {
    dp_packet_ol_reset_l4_csum_good(packet);
  } else {
    csum = &th->tcp_csum;
  }
  ...
}
```

OVS processing until 3.2 (cont)

And finally, preparing the packet before calling netdev transmit: **Rx** and **Tx** flags are considered again for a final decision on **Tx** flags.

```
void
dp_packet_ol_send_prepare(struct dp_packet *p, uint64_t flags)
{
  ...
  if (dp_packet_l4_checksum_good(p)) {
    dp_packet_hwol_reset_tx_l4_csum(p);
    return;
  }
  if (dp_packet_hwol_l4_is_tcp(p)
      && !(flags & NETDEV_TX_OFFLOAD_TCP_CKSUM)) {
      packet_tcp_complete_csum(p);
      dp_packet_ol_set_l4_csum_good(p);
      dp_packet_hwol_reset_tx_l4_csum(p);
  }
  ...
```

Enhancing unit tests (cont)

The new unit tests caught some issues:

- flow: Fix checksum offloads with simple match.
- netdev-linux: Fix offloads for IPv6 UDP packets.
- netdev-native-tnl: Do not validate already checked checksum.
- conntrack: Fix embedded checksums in ICMP errors.
- conntrack: Do not validate already checked checksum.
- ipf: Consider checksum only for fragmented packets.

IP checksums (cont)

This also simplifies packet processing, OVS code does not need to check if some inner, or outer Tx checksum has been requested, a simple status check on the outermost header is enough.
Example for the conntrack code:

```
@@ -2246,8 +2249,7 @@ conn_key_extract(struct conntrack *ct, struct dp_packet *pkt, ovs_be16 dl_type,
             /* Validate the checksum only when hwol is not supported and the
              * packet's checksum status is not known. */
             ok = extract_l3_ipv4(&ctx->key, l3, dp_packet_l3_size(pkt), NULL,
-                                 !dp_packet_hwol_l3_csum_ipv4_ol(pkt)
-                                 && !dp_packet_ip_checksum_good(pkt));
+                                 dp_packet_ip_checksum_unknown(pkt));
         }
```

IP checksums (cont)

And when OVS touches data, it only needs to update the status for known to be valid packet:

```
@@ -3677,8 +3679,8 @@ handle_ftp_ctl(struct conntrack *ct, const struct conn_lookup_ctx *ctx,
                 }
                 if (seq_skew) {
                     ip_len = ntohs(l3_hdr->ip_tot_len) + seq_skew;
-                    if (dp_packet_hwol_tx_ip_csum(pkt)) {
-                        dp_packet_ol_reset_ip_csum_good(pkt);
+                    if (dp_packet_ip_checksum_valid(pkt)) {
+                        dp_packet_ip_checksum_set_partial(pkt);
                     } else {
```

IP checksums (cont)

When decapsulating, inner status is pushed (back?) to the lower 16 bits.
No question about which Tx flags to set like with the previous API.

As a bonus, by introducing the “partial” status, only “unknown” status implies a full checksum (which can be noticed via the **native_tnl_l3csum_checked** coverage counter).

More simplifications in processing…

As checksums are now tracked as a packet status, parts of OVS validating the checksum for the “unknown” case now update the packet checksums status as “good” or “bad” and avoiding revalidating the checksum later.

This is the case when using conntrack, tunnel decapsulation or ip fragmentation.

Hardware segmentation: TSO (cont)

Limiting factor seems to be on the receiving end:

```
# ovs-appctl dpif-netdev/pmd-rxq-show -pmd 4 -secs 5
Displaying last 5 seconds pmd usage %

pmd thread numa_id 0 core_id 4:
  isolated : false
  port: dpdk0             queue-id:  0 (enabled)   pmd usage: 87 %
  overhead:  1 %


# ovs-vsctl get interface vhost0 statistics | tr ',' '\n' | grep -v '=0$' | grep -E 'ovs_tx|notifications'
 ovs_tx_failure_drops=10865
 ovs_tx_retries=115
 rx_q0_guest_notifications=1686214
 tx_q0_guest_notifications=1010470
```