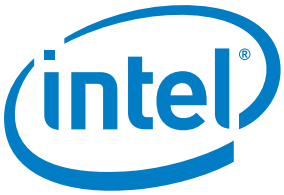


Intel® Ethernet Controller 700 Series GTPv1 – Dynamic Device Personalization

Application Note

April 2019

Document Number: 338930-001

Contents

Figures

Tables

Revision History

Date	Revision	Description
April 2019	001	Initial release

1.0 Introduction

To address the ever changing requirements for both Cloud and Network Functions Virtualization, the Intel® Ethernet Controller 700 Series was designed from the ground up to provide increased flexibility and agility. One of the design goals was to take parts of the fixed pipeline used in Intel® Ethernet Controller 500 Series, 82599 and X540, and move to a programmable pipeline allowing the Intel® Ethernet Controller 700 Series to be customized to meet a wide variety of customer requirements. This programmability has enabled over 60 unique configurations all based on the same core silicon.

With so many configurations being delivered to the market, the expanding role of Intel® Architecture in the Telecommunication market requires even more custom functionality. A common functionality request is for new packet classification types that are not currently supported, are customer-specific, or maybe not even fully defined yet. To address this, a new capability has been enabled on the Intel® Ethernet Controller 700 Series of NICs, called Dynamic Device Personalization (DDP). This capability allows dynamic reconfiguration of the packet processing pipeline to meet specific use case needs. Reconfiguration is achieved dynamically via application of firmware patches, herein labelled as *profiles*, which are specific to a use case.

The ability to classify new packet types inline, and distribute these packets to specified queues on the device's host interface, delivers the following performance and core utilization optimizations:

- Removes requirement for CPU cores on the host to perform classification and load balancing of packet types for the specified use case.
- Increases packet throughput, reduces packet latency for the use case.

In the case that multiple network controllers are present on the server, each controller can have its own pipeline profile, which can be applied without affecting other controllers and software applications using other controllers.

This application note describes the use of a GPRS tunneling protocol (GTPv1) profile to enhance performance and optimize core utilization for virtualized enhanced packet core (vEPC) and multi-access edge computing (MEC) use cases.

Section 2 describes the Dynamic Device Personalization capability, Section 3 describes the application of the GTPv1 profile to meet vEPC and MEC requirements, and Section 4 describes the enablement of this capability through DPDK software.

This document is part of the Network Transformation Experience Kit, which is available at: https://networkbuilders.intel.com/

1.1 Terminology

Table 1. Terminology

Term	Description
API	Application program interface
DPDK	Data Plane Development Kit
eNB	Evolved base station, also called eNodeB
FIN	Finish
GTP	GPRS Tunneling Protocol
IPSec	Internet Protocol Security
MAC	Media Access Control
MEC	Multi-access Edge Computing
NFV	Network Functions Virtualization
NIC	Network Interface Controller
NVM	Non Volatile Memory
PDN	Packet Data Network
PF	Physical Function
SCTP	Stream Control Transmission Protocol
SYN	Synchronized
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UE	User Equipment
vBNG	Virtualized Broadband Network Gateway
vEPC	Virtualized Enhanced Packet Core
VF	Virtual Function
VIM	Virtual Infrastructure Manager
VM	Virtual Machine
VNF	Virtual Network Function

1.2 Reference documents

Table 2. Reference documents

Document	Document No./Location
Intel® Ethernet Controller X710/XXV710/XL710 datasheet	https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf
Intel® Ethernet Controller 700 series dynamic device personalization presentation at DPDK summit, video	https://www.youtube.com/watch?v=X8aMDdAnnBI
Intel® Ethernet Controller 700 series dynamic device personalization presentation at DPDK summit, slides	https://www.slideshare.net/LF_DPDK/lfdpdk17flexible-and-extensible-support-for-new-protocol-processing-with-dpdk-using-dynamic-device-personalization
Intel® Ethernet Controller 700 series firmware version 6.01 or newer	https://downloadcenter.intel.com/product/75021/Intel-Ethernet-Controller-XL710-Series
Dynamic Device Personalization for Intel® Ethernet 700 Series user guide	https://software.intel.com/en-us/articles/dynamic-device-personalization-for-intel-ethernet-700-series
Intel® Ethernet Controller X710/XXV710/XL710 Adapters Dynamic Device Personalization GTPv1 Package	https://downloadcenter.intel.com/download/27587
Intel® Ethernet Controller X710/XXV710/XL710 Adapters Dynamic Device Personalization PPPoE Package	https://downloadcenter.intel.com/download/28040
Intel® Network Adapter Driver for PCIe* 40 Gigabit Ethernet Network Connections Under Linux*	https://downloadcenter.intel.com/download/24411/Intel-Network-Adapter-Driver-for-PCIe-40-Gigabit-Ethernet-Network-Connections-Under-Linux-

2.0 Dynamic Device Personalization

2.1 Overview

Most existing network controllers do not provide flexible reconfiguration of packet processing engines. In the best case, processing of new packet types or network protocols can be added to the controller by upgrading firmware. Normally, the firmware upgrade process includes total reset of the network controller and could include cold restart of the server in which the controller is installed. In this case, all Virtual Machines (VMs) running on the server must be detached from the Network Interface Controller (NIC) and migrated to another server during firmware update.

The ability to reconfigure network controllers for different Network Functions on-demand, without the need for migrating all VMs from the server, avoids unnecessary loss of compute for VMs during server cold restart. It also improves packet processing performance for applications/VMs by adding the capability to process new protocols in the network controller at runtime.

This kind of on-demand reconfiguration is offered by Intel® Ethernet Controller 700 Series' Dynamic Device Personalization capability. This section describes the instantiation of this device reconfiguration capability. At time of publication, several profiles are available from Intel download center:

- GTPv1 profile for vEPC: https://downloadcenter.intel.com/download/27587
- PPPoE profile for vBNG: https://downloadcenter.intel.com/download/28040

Additional profiles that target cable, 5G and switching/routing use cases are in development. For more information on accessing the library of profiles or to request a new profile, contact your local Intel representative.

2.2 Demystifying dynamic device personalization

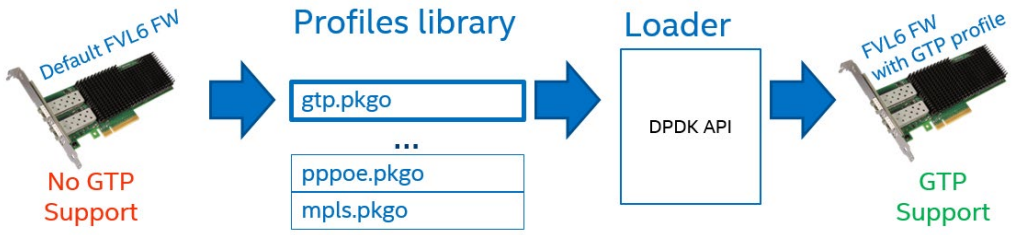
Dynamic Device Personalization describes the capability in the Intel® Ethernet Controller 700 Series devices to load an additional firmware profile on top of the device's default firmware image. This enables parsing and classification of additional specified packet types so these packet types can be distributed to specific queues on the NIC's host interface using standard filters. Software applies these custom profiles in a non-permanent, transaction-like mode, so that the original network controller's configuration is restored after NIC reset or by rolling back profile changes by software. Using APIs provided by drivers, personality profiles can be applied by the Linux* kernel and Data Plane Development Kit (DPDK). Integration with higher level management/orchestration tools is in progress.

Dynamic Device Personalization can be used to optimize packet processing performance for different network functions, native or running in a virtual environment. By applying a Dynamic Device Personalization profile to the network controller, the following use cases could be addressed:

- New Packet Classification types (flow types) for offloading packet classification to network controller:
  - New IP Protocols in addition to TCP/UDP/SCTP, for example, IP ESP, IP AH
  - New UDP Protocols, for example, MPLSoUDP or QUIC
  - New TCP subtypes, like TCP SYN-no-ACK
  - New tunnelled protocols like PPPoE, GTP-C/GTP-U
- New Packet Types for packets identification, reported on packet's RX descriptor:
  - IP6, GTP-U, IP4, UDP, PAY4
  - IP4, GTP-U, IP6, UDP, PAY4
  - IP4, GTP-U, PAY4
  - IP6, GTP-C, PAY4
  - MPLS, MPLS, IP6, TCP, PAY4

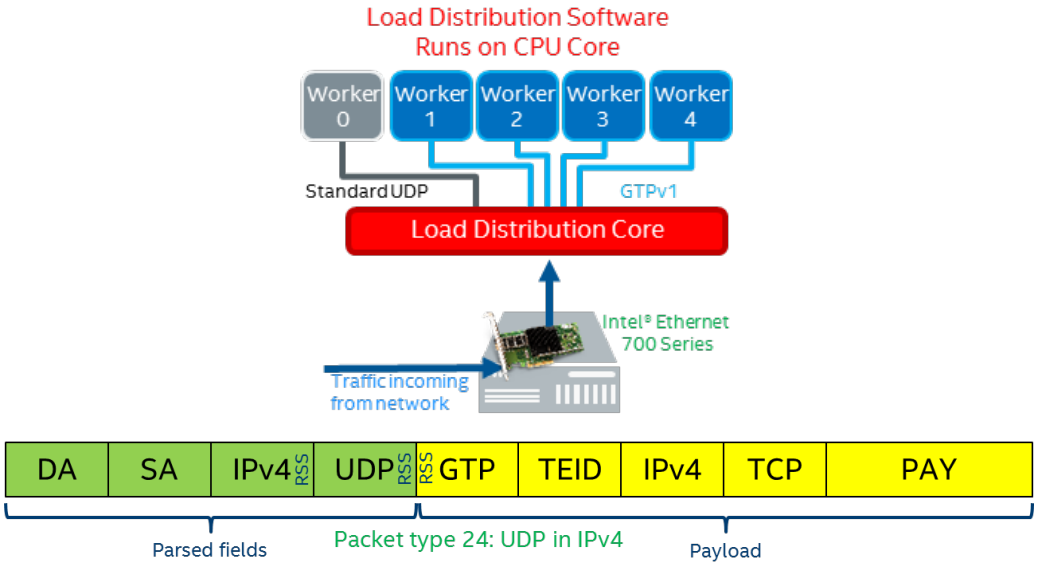
The profile application process is illustrated in Figure 1 below. In this case, the Intel® Ethernet Controller 700 Series device begins with the default firmware configuration. In this configuration, the NIC supports classification of some default packet types (UDP, TCP, VxLAN, GRE, etc), allowing these default packets to be identified and distributed to queues in the NIC. Classification of other packet types, such as those listed above, is not supported by default. To enable classification of GTP packets, the firmware profile enabling GTP packet classification is selected and loaded to the device via a DPDK API or ethtool. This profile is loaded in runtime. With this additional firmware profile loaded, the NIC now supports classification of GTP packets inline.

Figure 1. Dynamic device personalization firmware profile application



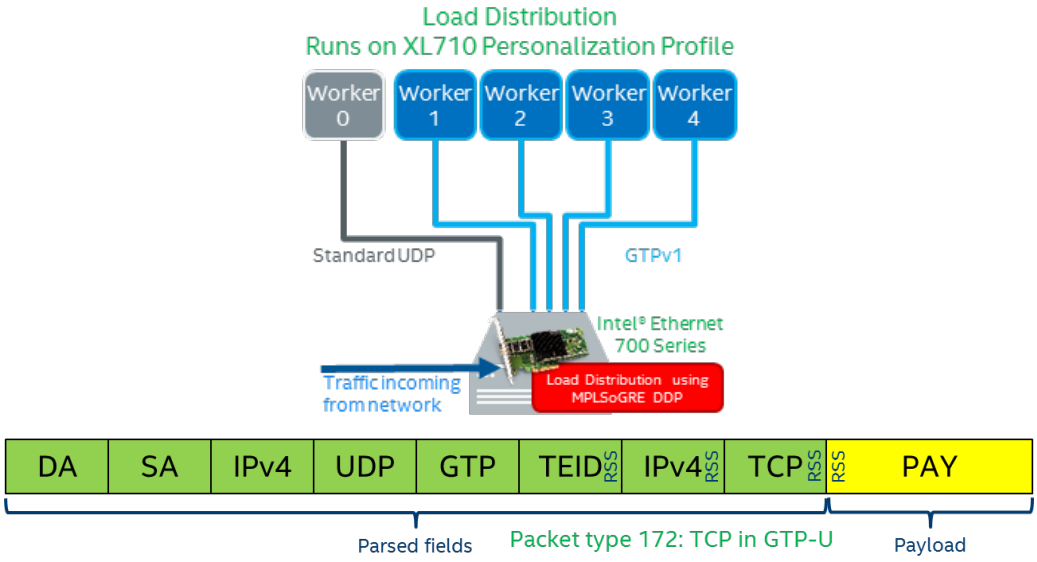
The NIC's visibility of the packet header fields before and after the application of the GTPv1 profile is indicated in Figure 2 and Figure 3 below. With the default firmware image, the GTP-encapsulated frame within the UDP outer header cannot be identified by the device, and the GTP-encapsulated frame is effectively the payload in the outer UDP packet. GTP is an unknown flow type here and so no receive side scaling (RSS) or flow director (FDIR) capabilities are possible on the encapsulated frame. In this case, classification and distribution of GTP packets must be performed by one or more cores on the CPU.

Figure 2. Packet header identification before application of GTPv1 profile



After the GTPv1 profile is loaded to the Intel® Ethernet Controller, the GTP flow type is defined and encapsulated frame fields (including GTP TEID) can be used for RSS, Flow Director, or Cloud Filters. The NIC has full visibility of all header fields and can perform load distribution to queues based on this improved classification capability.

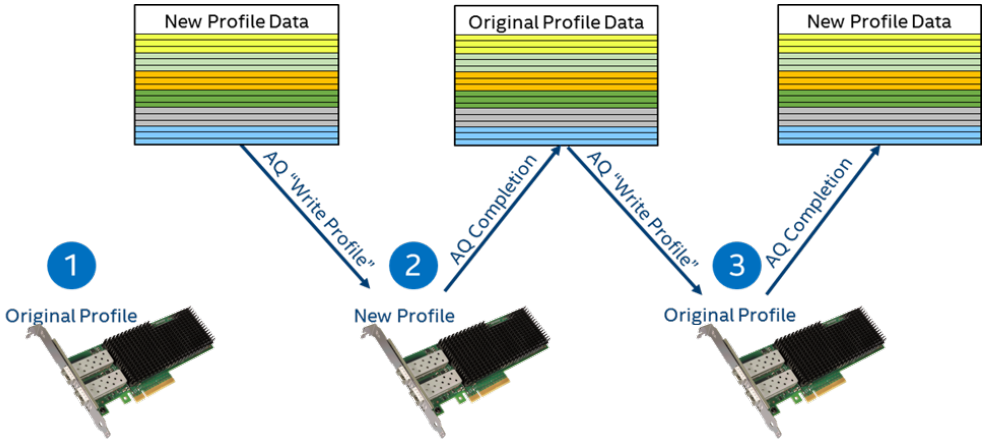
Figure 3. Packet header identification after application of GTPv1 profile



With this firmware profile applied, the Intel® Ethernet Controller 700 Series is performing classification of GTP packets and carrying out load distribution inline, removing the need for a load distribution core to perform the same function.

The profile is applied to the Intel® Ethernet Controller 700 Series device in a transaction-like mode, as illustrated in Figure 4, showing application and removal of a profile.

Figure 4. Dynamic reconfiguration of the Intel® Ethernet Controller 700 Series

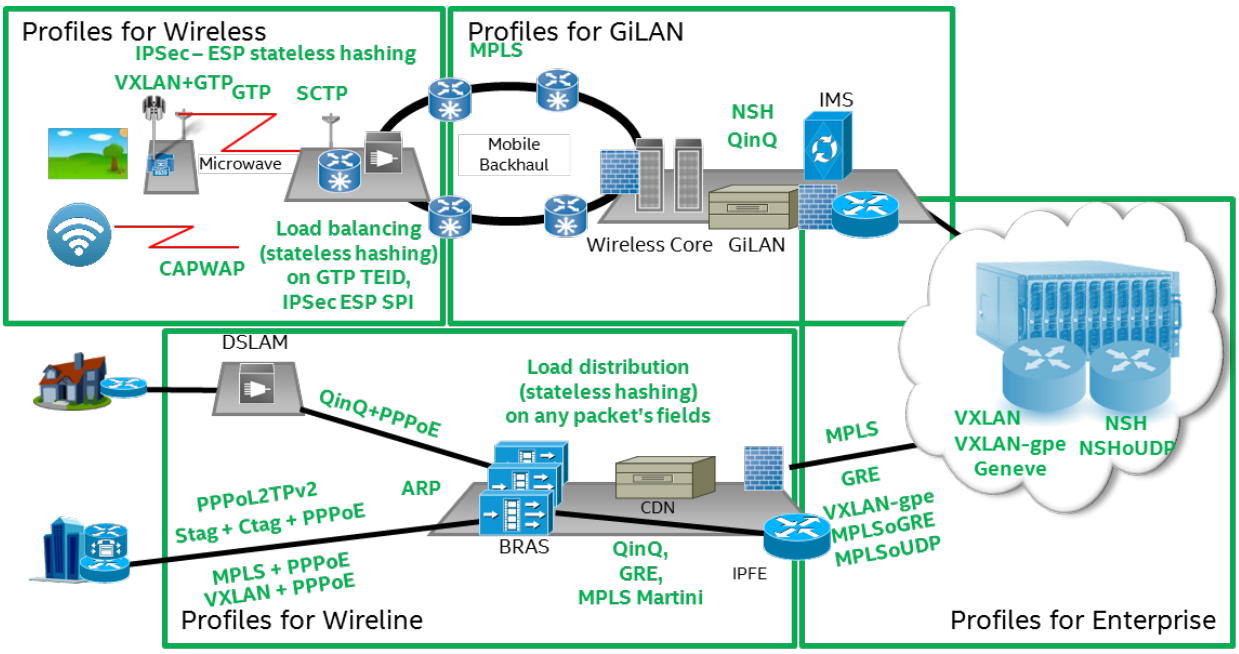


The NIC does not need to be reset to restore the original configuration and the profile can be applied/removed while traffic is present.

3.0 Utility / Use Case

The Dynamic Device Personalization capability can be used to optimize packet processing performance for different network functions, native or running in a virtual environment. Figure 5 below indicates typical packet types used at various locations in the network, separated by network segment. GTP encapsulation is primarily used in the wireless segment, with vEPC and MEC representing the dominant use cases. This section describes these use cases and the application of the Dynamic Device Personalization GTPv1 profile to improve performance.

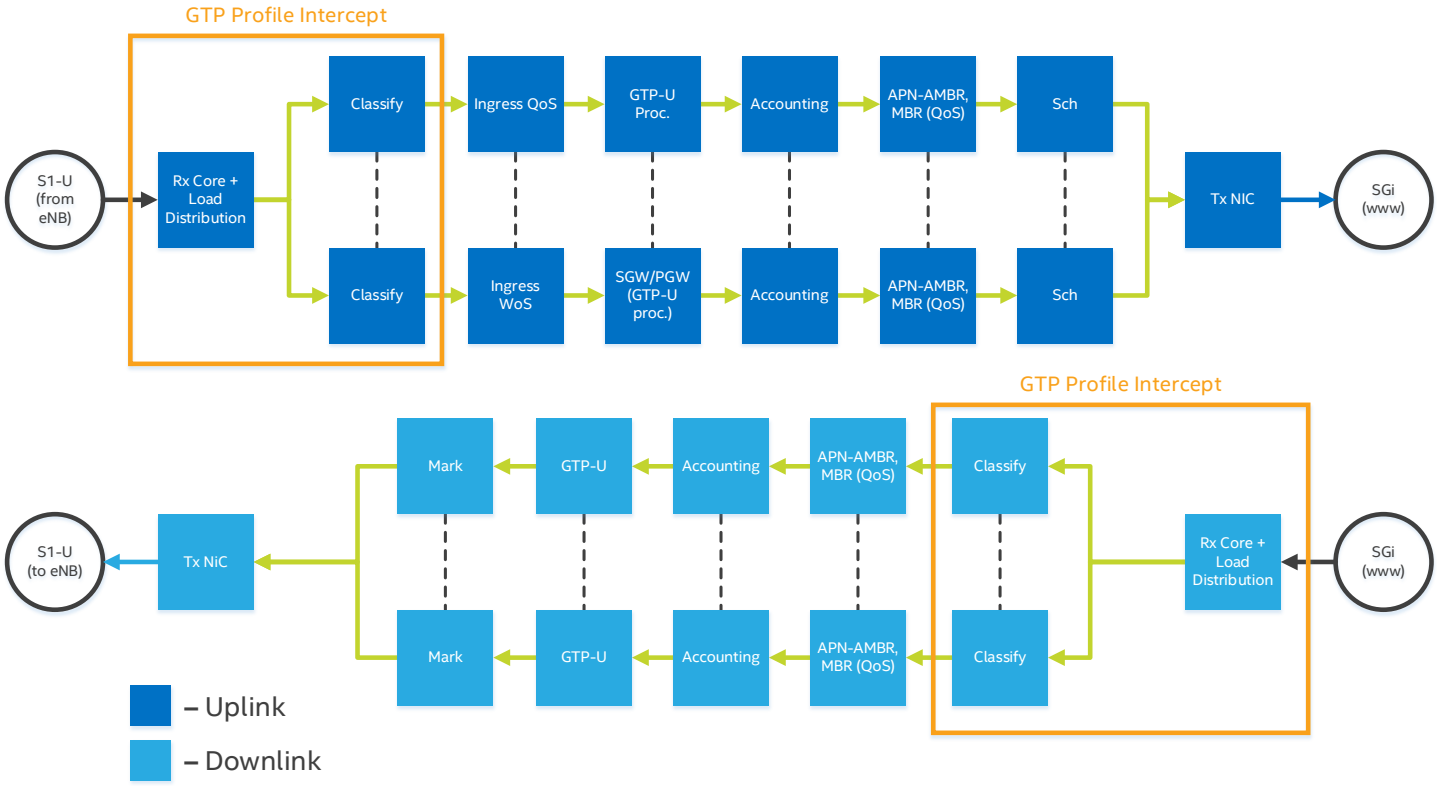
Figure 5. Packet encapsulations in the wider network



3.1 Application of Technology – vEPC

Figure 6 illustrates a typical packet pipeline for the server node in a virtualized EPC. Packets are classified and distributed across multiple parallel queues for further processing (QoS, GTP processing, egress scheduling) before transmission. In a virtualized implementation, typically the load distribution and classification functions are performed by CPU cores. This is the case when an Intel® Ethernet Controller 700 Series with default firmware configuration is used as the server's NIC.

Figure 6. Typical vEPC server node packet pipeline



3.1.1 DDP GTPv1 profile use case for LTE vEPC User Plane

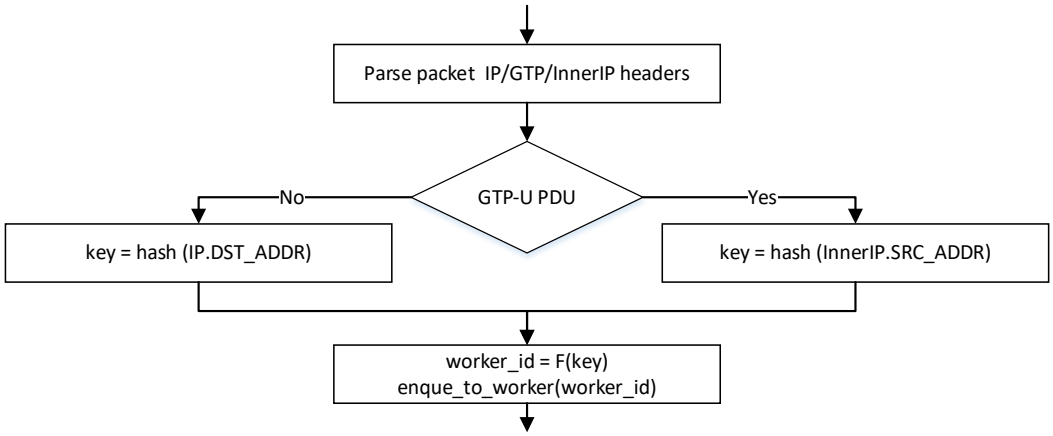
vEPC implements the concept of control and user plane separation. The vEPC User Plane (also called the Data Plane) consists of multiple instances with each instance running on several CPU cores inside the VM. CPU cores in vEPC may act in one of two main roles: receive (RX) core or worker core.

- Receive cores are responsible for fetching packets from the NIC RX rings, packet classification and packet distribution to a particular worker core, including workload balancing between worker cores.
- Worker cores implement LTE EPC user plane stack functionality and handle both uplink (UL, from UE/eNB to the PDN) and downlink (DL, from the PDN to eNB/UE) traffic. Worker cores process packets in a run-to-completion mode.

The vEPC User Plane classifies each received packet to identify the worker core for processing. In order to get better cache utilization and improve performance, vEPC binds all data traffic to and from the same UE IP to one of the worker cores, so control structures related to a particular UE IP are updated from a single worker core.

To pin UE IP to a worker core, vEPC uses the UE IP address as a key for the core identification. UL traffic is received on S1-U interface as GTP-U encapsulated IP packets, so the UE IP address is extracted as a source address from the encapsulated IP packet. DL traffic is received on the SGi interface as regular IP packets, so the UE IP address is extracted as a destination IP address of the packet. Figure 7 shows the flow diagram of a worker core identification process (a case where both UL and DL packets are received from the same NIC RX port).

Figure 7. Worker core identification inside vEPC



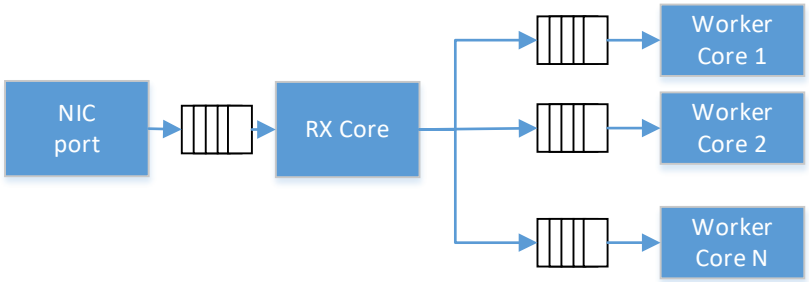
3.1.1.1 vEPC default configuration

In the default configuration of vEPC, RX cores:

- Fetch packets from S1U and SGi interfaces.
- Parse packets' headers, classify packets, and calculate worker_id assignment.
- Dispatch packets to the identified worker cores over dedicated software queues.

Worker cores fetch packets from the software queues and process them up to the moment the packet goes into the TX queue of the NIC.

Figure 8. Data flow in vEPC configuration with RX core

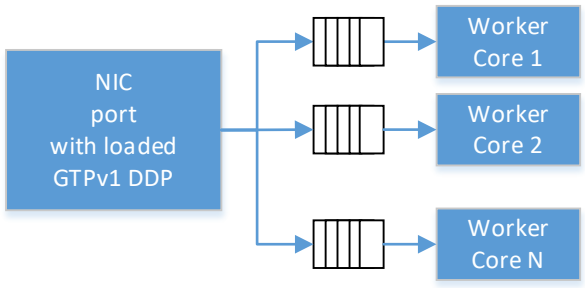


3.1.1.2 vEPC configuration with GTPv1 DDP profile enabled

With the Dynamic Device Personalization GTPv1 profile applied to the Intel® Ethernet Controller 700 Series in the server, the classification and load distribution functions can be performed inline in the NIC, thus preserving CPU core processing resources for higher value tasks.

Using DDP GTPv1 and modifying RSS inset configuration for selected packet classification types (PCTYPEs), vEPC implements a work mode where the functionality initially executed by the RX core is moved to the NIC. Here, worker cores fetch packets directly from the NIC RX rings.

Figure 9. Data flow in vEPC configuration with GTPv1 DDP profile



This implementation removes the processing overhead for one core (hyper-thread) for each instance of vEPC running. Taking into consideration that multiple instances of vEPC User Plane are often running on one server, **the DDP GTPv1 profile can create significant efficiency for multiple cores**. Table 3 shows profiling results for one instance of vEPC User Plane running two configurations described above.

Table 3. Example of core utilization for vEPC DP instance

	Configuration	
	Dedicated RX core	**GTPv1 DDP profile**
Packet rate	3.9 MPPS	
Bitrate	20.5 Gbps	
RX cores (hyper-threads)	1	0
Worker cores (hyper-threads)	7	7
Total number cores used for packet processing in vEPC User Plane	8	7
CPU utilization, receive core	40%	-
CPU utilization, worker cores	71%	71%

Utilization of the DDP GTPv1 profile also **reduces vEPC User Plane stack packet processing time (latency)** by removing the extra stage of packet processing in software (packet classification on RX core) and eliminating time packets spend in the software queue between RX core and Worker core.

3.2 Application of Technology – MEC

Multi-access Edge Computing (MEC) brings IT and cloud-computing capabilities into the Access Network in close proximity to mobile subscribers. MEC enables deployment of intelligent devices at the edge of the network.

MEC incorporates the benefits of virtualization and cloud-computing in order to place high-powered computing capabilities as close as possible to subscribers. Edge computing offers a service environment with ultra-low latency and high-bandwidth. This physical proximity could, as an example, reduce video stalling by storing video content closer to the edge (35% backhaul capacity reduction), or reduce webpage download time by 20 percent.) (Source: ETSI Industry Specification Group for Mobile Edge Computing presentation at SDN World Congress.)

Application developers and content providers can use direct access to real-time network information (such as subscriber location, cell load, etc.) to offer context-related services that are capable of differentiating the mobile broadband experience. MEC allows content, services and applications to be accelerated, increasing responsiveness from the edge. The mobile subscriber's experience can be enriched through efficient network and service operations, based on insight into the radio and network conditions.

MEC can be deployed on the S1 interface or the SGI interface. With S1 deployment, the MEC platform handles S1-U and S1-AP traffic, which is GTP and SCTP traffic, as shown in Figure 10. With SGi deployment, the MEC platform handles IP traffic, as shown in Figure 11.

Figure 10. Typical MEC deployment on S1 interface

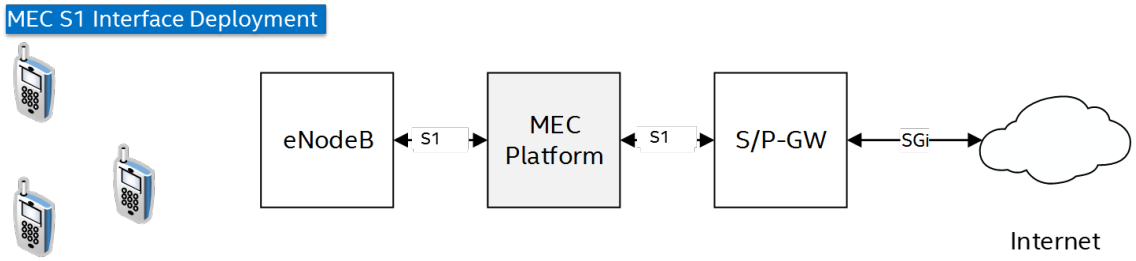
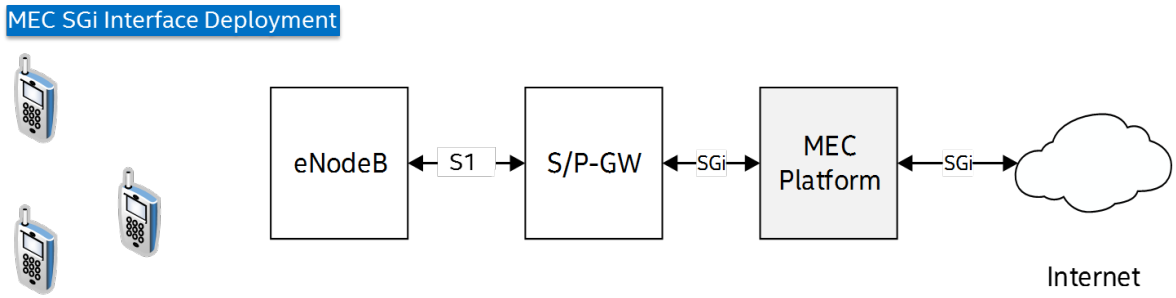


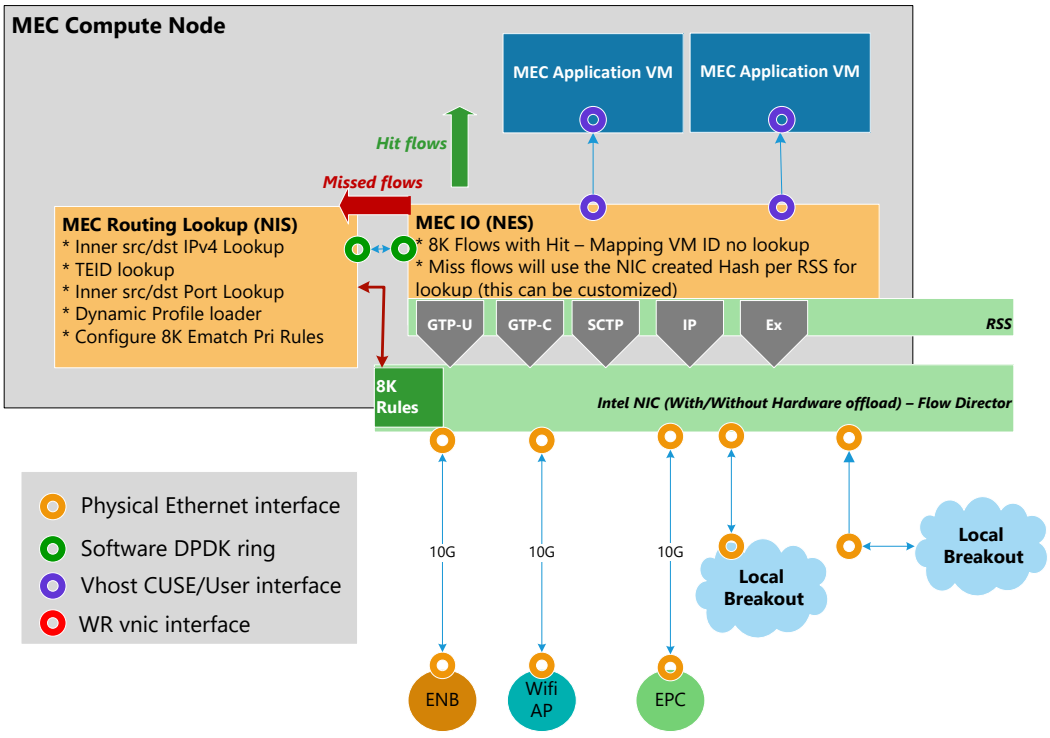
Figure 11. Typical MEC deployment on SGi interface



The DDP GTPv1 profile is applicable for S1 deployment. With the GTP DDP profile configured on the MEC platform, the entire MEC routing process can effectively be bypassed, as demonstrated in Figure 12.

The diagram shows the reference implementation on Open Network Edge Services Software (OpenNESS). For details, refer to: https://www.open-ness.org/

Figure 12. DDP offload on OpenNESS for MEC S1 interface



This approach of performing packet classification and load distribution in the NIC can lead to reduction in latency. It should be noted that the DDP profile is also applicable for SGi deployment, which can be used to implement advanced filtering on the IP address and ports.

4.0 Enablement

This section details how to install, configure, and use the Dynamic Device Personalization GTPv1 profile with Linux and DPDK.

Dynamic Device Personalization requires the Intel® Ethernet Controller XL710 based Ethernet NIC with the latest firmware 6.01, available here: https://downloadcenter.intel.com/product/75021/Intel-Ethernet-Controller-XL710-Series

Basic support for applying DDP profiles to Intel Ethernet 700 Series network adapters was added to DPDK 17.05. Later, DPDK 17.08 and 17.11 introduced more advanced DDP APIs, including the ability to report a profile's information without loading a profile to an Intel Ethernet 700 Series network adapter first. These APIs can be used to try out new DDP profiles with DPDK without implementing full support for the protocols in the DPDK rte_flow API.

The Intel® Network Adapter Driver for PCIe* 40 Gigabit Ethernet Network Connections under Linux supports loading and rolling back DDP profiles using ethtool, in version 2.7.26 and later.

The GTPv1 profile for vEPC is available on Intel® download center: https://downloadcenter.intel.com/download/27587

The GTPv1 protocol is supported in DPDK rte_flow(). Linux and DPDK mechanisms to apply a profile are described in the following sections.

4.1 DPDK APIs

The following three i40e private calls are part of DPDK 17.08.

rte_pmd_i40e_process_ddp_package(): This function is used to download a DDP profile and register it or rollback a DDP profile and un-register it.

```
int rte_pmd_i40e_process_ddp_package(
   uint8_t port,  /* DPDK port index to download DDP package to */
   uint8_t *buff, /* buffer with the package in the memory */
   uint32_t size, /* size of the buffer */
   rte_pmd_i40e_package_op op /* operation: add, remove, write profile
*/
);
```

rte_pmd_i40e_get_ddp_info(): This function is used to request information about a profile without downloading it to a network adapter.

```
int rte_pmd_i40e_get_ddp_info(
    uint8_t *pkg_buff,  /* buffer with the package in the memory */
    uint32_t pkg_size,  /* size of the package buffer */
    uint8_t *info_buff, /* buffer to store information to */
    uint32_t info_size, /* size of the information buffer */
    enum rte_pmd_i40e_package_info type /* type of required information
*/
);
```

rte_pmd_i40e_get_ddp_list(): This function is used to get the list of registered profiles.

```
int rte_pmd_i40e_get_ddp_list (
    uint8_t port,  /* DPDK port index to get list from */
    uint8_t *buff, /* buffer to store list of registered profiles */
    uint32_t size  /* size of the buffer */
);
```

DPDK 17.11 adds some extra DDP-related functionality with the following function.

rte_pmd_i40e_get_ddp_info(): Updated to retrieve more information about the profile.

New APIs were added to handle flow type, created by DDP profiles:

rte_pmd_i40e_flow_type_mapping_update(): Used to map hardware-specific packet classification type to DPDK flow types.

```
int rte_pmd_i40e_flow_type_mapping_update(
    uint8_t port, /* DPDK port index to update map on */
    /* array of the mapping items */
    struct rte_pmd_i40e_flow_type_mapping *mapping_items,
    uint16_t count, /* number of PCTYPEs to map */
    uint8_t exclusive /* 0 to overwrite only referred PCTYPEs */
);
```

rte_pmd_i40e_flow_type_mapping_get(): Used to retrieve current mapping of hardware-specific packet classification types to DPDK flow types.

```
int rte_pmd_i40e_flow_type_mapping_get(
    uint8_t port, /* DPDK port index to get mapping from */
    /* pointer to the array of RTE_PMD_I40E_FLOW_TYPE_MAX mapping
items*/
    struct rte_pmd_i40e_flow_type_mapping *mapping_items
);
```

rte_pmd_i40e_flow_type_mapping_reset(): Resets flow type mapping table.

```
int rte_pmd_i40e_flow_type_mapping_reset(
       uint8_t port /* DPDK port index to reset mapping on */
);
```

4.2 Ethtool commands

Prerequisites:

- Red Hat* Enterprise Linux* (RHEL*) 7.5 or later
- Linux* Kernel 4.0.1 or newer

To apply a profile, first copy it to the `intel/i40e/ddp` directory relative to your firmware root (usually `/lib/firmware`).

For example:

```
/lib/firmware/intel/i40e/ddp
```

Then use the ethtool -f|--flash flag with region 100 in this format:
`ethtool -f <interface name> <profile name> 100`

For example: `ethtool -f eth0 gtp.pkgo 100`

You can roll back to a previously loaded profile using - instead of the profile name in this format: `ethtool -f <interface name> - 100`

For example:

```
ethtool -f eth0 - 100
```

For every rollback request, one profile will be removed, from last to first (LIFO) order.

For more details, see the driver `readme.txt` file.

Note: Using ethtool, DDP profiles can be loaded only on the interface corresponding to the first physical function of the device (PF0), but the configuration is applied to all ports of the adapter.

4.3 Using DDP profiles with test-pmd

To demonstrate DDP functionality of Intel Ethernet 700 Series network adapters and explain DDP APIs, the GTPv1 profile is used along with the testpmd application from DPDK. Using this example and API explanations, it is possible to integrate DDP with any DPDK-based application.

Although DPDK 17.11 adds GTPv1 with IPv4 payload support at the rte_flow API level, lower-level APIs are used here to demonstrate how to work with the Intel® Ethernet 700 Series network adapter directly for any new protocols added by DDP and not yet enabled in rte_flow.

For demonstration, GTPv1-U packets with the following configuration are used.

Figure 13. GTPv1 GTP-U packets configuration

```
Source IP                1.1.1.1
Destination IP           2.2.2.2
IP Protocol              17 (UDP)
GTP Source Port          45050
GTP Destination Port     2152
GTP Message type         0xFF
GTP Tunnel id            0x11111111-0xFFFFFFFF random
GTP Sequence number      0x000001
-- Inner IPv4 Configuration --------------
Source IP                3.3.3.1-255 random
Destination IP           4.4.4.1-255 random
IP Protocol              17 (UDP)
UDP Source Port          53244
UDP Destination Port     57069
```

Clearly, the outer IPv4 header does not have any entropy for RSS because IP addresses and UDP ports defined statically. However, the GTPv1 header has random tunnel endpoint identifier (TEID) values in the range of 0x11111111 to 0xFFFFFFFF, and the inner IPv4 packet has IP addresses randomly host-generated in the range of 1 to 255.

The pcap file with synthetic GTPv1-U traffic using the configuration above can be downloaded here: https://software.intel.com/en-us/articles/dynamic-device-personalization-for-intel-ethernet-700-series

First, testpmd is started in receive only mode with four queues, and verbose mode and RSS are enabled.

Figure 14. testpmd startup configuration

```
testpmd -w 02:00.0 -- -i --rxq=4 --txq=4 --forward-mode=rxonly
testpmd> port config all rss all
testpmd> set verbose 1
testpmd> start
```

Using any GTP-U capable traffic generator, four GTP-U packets are sent. A provided pcap file with synthetic GTPv1-U traffic can be used as well.

All packets have the same outer IP header, therefore they are received on queue 1 and reported as IPv4 UDP packets.

Figure 15. Distribution of GTP-U packets without GTPv1 profile

```
testpmd> port 0/queue 1: received 4 packets
src=3C:FD:FE:A6:21:24 - dst=00:10:20:30:40:50 - type=0x0800 - length=178 -
nb_segs=1 - RSS hash=0xd9a562 - RSS queue=0x1 - hw ptype: L2_ETHER
L3_IPV4_EXT_UNKNOWN L4_UDP  - sw ptype: L2_ETHER L3_IPV4 L4_UDP  - l2_len=14
- l3_len=20 - l4_len=8 - Receive queue=0x1
  ol_flags: PKT_RX_RSS_HASH PKT_RX_L4_CKSUM_GOOD PKT_RX_IP_CKSUM_GOOD

src=3C:FD:FE:A6:21:24 - dst=00:10:20:30:40:50 - type=0x0800 - length=178 -
nb_segs=1 - RSS hash=0xd9a562 - RSS queue=0x1 - hw ptype: L2_ETHER
L3_IPV4_EXT_UNKNOWN L4_UDP  - sw ptype: L2_ETHER L3_IPV4 L4_UDP  - l2_len=14
- l3_len=20 - l4_len=8 - Receive queue=0x1
  ol_flags: PKT_RX_RSS_HASH PKT_RX_L4_CKSUM_GOOD PKT_RX_IP_CKSUM_GOOD

src=3C:FD:FE:A6:21:24 - dst=00:10:20:30:40:50 - type=0x0800 - length=178 -
nb_segs=1 - RSS hash=0xd9a562 - RSS queue=0x1 - hw ptype: L2_ETHER
L3_IPV4_EXT_UNKNOWN L4_UDP  - sw ptype: L2_ETHER L3_IPV4 L4_UDP  - l2_len=14
- l3_len=20 - l4_len=8 - Receive queue=0x1
  ol_flags: PKT_RX_RSS_HASH PKT_RX_L4_CKSUM_GOOD PKT_RX_IP_CKSUM_GOOD

src=3C:FD:FE:A6:21:24 - dst=00:10:20:30:40:50 - type=0x0800 - length=178 -
nb_segs=1 - RSS hash=0xd9a562 - RSS queue=0x1 - hw ptype: L2_ETHER
L3_IPV4_EXT_UNKNOWN L4_UDP  - sw ptype: L2_ETHER L3_IPV4 L4_UDP  - l2_len=14
- l3_len=20 - l4_len=8 - Receive queue=0x1
  ol_flags: PKT_RX_RSS_HASH PKT_RX_L4_CKSUM_GOOD PKT_RX_IP_CKSUM_GOOD
```

Here, hash values for all four packets are the same: 0xD9A562. This happens because IP source/destination addresses and UDP source/destination ports in the outer (tunnel end point) IP header are statically defined and do not change from packet to packet.

Now the DDP GTPv1 profile is applied to the network adapter port. For the purpose of the demonstration, it is assumed that the profile package file was downloaded and extracted to the `/lib/firmware/intel/i40e/ddp` folder. The profile will load from the `gtp.pkgo` file and the original configuration will be stored to the `gtp.bak` file:

Figure 16. Applying GTPv1 profile to device

```
testpmd> stop
testpmd> port stop 0
testpmd> ddp add 0 /lib/firmware/intel/i40e/ddp/gtp.pkgo,/home/pkg/gtp.bak
```

The `ddp add 0 /lib/firmware/intel/i40e/ddp/gtp.pkgo,/home/pkg/gtp.bak` command first loads the **`gtp.pkgo`** file to the memory buffer, then passes it to **rte_pmd_i40e_process_ddp_package()** with the **RTE_PMD_I40E_PKG_OP_WR_ADD** operation, and then saves the original configuration, returned in the same buffer, to the **`gtp.bak`** file.

For a scenario where the supported Linux driver is loaded on the device's physical function 0 (PF0) and testpmd uses any other physical function of the device, the profile can be loaded with the following ethtool command:

```
ethtool -f eth0 gtp.pkgo 100
```

Confirm that the profile was loaded successfully.

Figure 17. Checking whether the device has any profiles loaded

```
testpmd> ddp get list 0
Profile number is: 1

Profile 0:
Track id:     0x80000008
Version:      1.0.2.0
Profile name: GTPv1-C/U IPv4/IPv6 payload
```

The `ddp get list 0` command calls ***rte_pmd_i40e_get_ddp_list()*** and prints the returned information.

Track ID is the unique identification number of the profile that distinguishes it from any other profiles.

Get information about new packet classification types and packet types created by profile.

Figure 18. Getting information about the DDP profile

```
testpmd> ddp get info /lib/firmware/intel/i40e/ddp/gtp.pkgo
Global Track id:       0x80000008
Global Version:        1.0.2.0
Global Package name:   GTPv1-C/U IPv4/IPv6 payload

i40e Profile Track id: 0x80000008
i40e Profile Version:  1.0.2.0
i40e Profile name:     GTPv1-C/U IPv4/IPv6 payload

Package Notes:
This profile enables GTPv1-C/GTPv1-U classification
with IPv4/IPV6 payload
Hash input set for GTPC is TEID
Hash input set for GTPU is TEID and inner IP addresses (no ports)
Flow director input set is TEID

List of supported devices:
  8086:1572 FFFF:FFFF
  8086:1574 FFFF:FFFF
  8086:1580 FFFF:FFFF
  8086:1581 FFFF:FFFF
  8086:1583 FFFF:FFFF
  8086:1584 FFFF:FFFF
  8086:1585 FFFF:FFFF
  8086:1586 FFFF:FFFF
  8086:1587 FFFF:FFFF
  8086:1588 FFFF:FFFF
  8086:1589 FFFF:FFFF
  8086:158A FFFF:FFFF
  8086:158B FFFF:FFFF

List of used protocols:
  12: IPV4
```

```
  13: IPV6
  17: TCP
  18: UDP
  19: SCTP
  20: ICMP
  21: GTPU
  22: GTPC
  23: ICMPV6
  34: PAY3
  35: PAY4
  44: IPV4FRAG
  48: IPV6FRAG

List of defined packet classification types:
  22: GTPU IPV4
  23: GTPU IPV6
  24: GTPU
  25: GTPC

List of defined packet types:
  167: IPV4 GTPC PAY4
  168: IPV6 GTPC PAY4
  169: IPV4 GTPU IPV4 PAY3
  170: IPV4 GTPU IPV4FRAG PAY3
  171: IPV4 GTPU IPV4 UDP PAY4
  172: IPV4 GTPU IPV4 TCP PAY4
  173: IPV4 GTPU IPV4 SCTP PAY4
  174: IPV4 GTPU IPV4 ICMP PAY4
  175: IPV6 GTPU IPV4 PAY3
  176: IPV6 GTPU IPV4FRAG PAY3
  177: IPV6 GTPU IPV4 UDP PAY4
  178: IPV6 GTPU IPV4 TCP PAY4
  179: IPV6 GTPU IPV4 SCTP PAY4
  180: IPV6 GTPU IPV4 ICMP PAY4
  181: IPV4 GTPU PAY4
  182: IPV6 GTPU PAY4
  183: IPV4 GTPU IPV6FRAG PAY3
  184: IPV4 GTPU IPV6 PAY3
  185: IPV4 GTPU IPV6 UDP PAY4
  186: IPV4 GTPU IPV6 TCP PAY4
  187: IPV4 GTPU IPV6 SCTP PAY4
  188: IPV4 GTPU IPV6 ICMPV6 PAY4
  189: IPV6 GTPU IPV6 PAY3
  190: IPV6 GTPU IPV6FRAG PAY3
  191: IPV6 GTPU IPV6 UDP PAY4
  113: IPV6 GTPU IPV6 TCP PAY4
  120: IPV6 GTPU IPV6 SCTP PAY4
  128: IPV6 GTPU IPV6 ICMPV6 PAY4
```

The `ddp get info gtp.pkgo` command makes multiple calls of ***rte_pmd_i40e_get_ddp_info()*** to get different information about the profile, and prints it.

There is a lot of information, including the new packet classifier types.

Figure 19. New PCTYPEs defined by GTPv1 profile

```
List of defined packet classification types:
  22: GTPU IPV4
  23: GTPU IPV6
  24: GTPU
  25: GTPC
```

There are four new packet classification types created in addition to all default PCTYPEs available. (For details, refer to *Table 7-5. Packet classifier types and its input sets* in the Intel® Ethernet Controller X710/XXV710/XL710 datasheet: https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xl710-10-40-controller-datasheet.pdf.)

To enable RSS for GTPv1-U with the IPv4 payload, packet classifier type 22 is mapped to the DPDK flow type. Flow types are defined in `rte_eth_ctrl.h`; the first 21 are in use in DPDK 17.11 and so can map to flows 22 and up. After mapping to a flow type, the port is restarted and RSS enabled for flow type 22.

Figure 20. Mapping new PCTYPEs to DPDK flow types

```
testpmd> port config 0 pctype mapping update 22 22
testpmd> port start 0
testpmd> start
testpmd> port config all rss 22
```

The `port config 0 pctype mapping update 22 22` command calls **rte_pmd_i40e_flow_type_mapping_update()** to map new packet classifier type 22 to DPDK flow type 22 so that the `port config all rss 22` command can enable RSS for this flow type.

When GTP traffic is resent, the packets are classified as GTP in the NIC device and distributed to multiple queues.

Figure 21. Distribution of GTP-U packets with GTPv1 profiles applied to the device

```
port 0/queue 1: received 1 packets
  src=00:01:02:03:04:05 - dst=00:10:20:30:40:50 - type=0x0800 - length=178 -
nb_segs=1 - RSS hash=0x342ff376 - RSS queue=0x1 - hw ptype:
L3_IPV4_EXT_UNKNOWN TUNNEL_GTPU INNER_L3_IPV4_EXT_UNKNOWN INNER_L4_UDP  - sw
ptype: L2_ETHER L3_IPV4 L4_UDP  - l2_len=14 - l3_len=20 - l4_len=8 - VXLAN
packet: packet type =32912, Destination UDP port =2152, VNI = 3272871 -
Receive queue=0x1
  ol_flags: PKT_RX_RSS_HASH PKT_RX_L4_CKSUM_GOOD PKT_RX_IP_CKSUM_GOOD

port 0/queue 2: received 1 packets
  src=00:01:02:03:04:05 - dst=00:10:20:30:40:50 - type=0x0800 - length=178 -
nb_segs=1 - RSS hash=0xe3402ba5 - RSS queue=0x2 - hw ptype:
L3_IPV4_EXT_UNKNOWN TUNNEL_GTPU INNER_L3_IPV4_EXT_UNKNOWN INNER_L4_UDP  - sw
ptype: L2_ETHER L3_IPV4 L4_UDP  - l2_len=14 - l3_len=20 - l4_len=8 - VXLAN
packet: packet type =32912, Destination UDP port =2152, VNI = 9072104 -
Receive queue=0x2
  ol_flags: PKT_RX_RSS_HASH PKT_RX_L4_CKSUM_GOOD PKT_RX_IP_CKSUM_GOOD
```

```
port 0/queue 0: received 1 packets
  src=00:01:02:03:04:05 - dst=00:10:20:30:40:50 - type=0x0800 - length=178 -
nb_segs=1 - RSS hash=0x6a97ed3 - RSS queue=0x0 - hw ptype:
L3_IPV4_EXT_UNKNOWN TUNNEL_GTPU INNER_L3_IPV4_EXT_UNKNOWN INNER_L4_UDP  - sw
ptype: L2_ETHER L3_IPV4 L4_UDP  - l2_len=14 - l3_len=20 - l4_len=8 - VXLAN
packet: packet type =32912, Destination UDP port =2152, VNI = 5877304 -
Receive queue=0x0
  ol_flags: PKT_RX_RSS_HASH PKT_RX_L4_CKSUM_GOOD PKT_RX_IP_CKSUM_GOOD

port 0/queue 3: received 1 packets
  src=00:01:02:03:04:05 - dst=00:10:20:30:40:50 - type=0x0800 - length=178 -
nb_segs=1 - RSS hash=0x7d729284 - RSS queue=0x3 - hw ptype:
L3_IPV4_EXT_UNKNOWN TUNNEL_GTPU INNER_L3_IPV4_EXT_UNKNOWN INNER_L4_UDP  - sw
ptype: L2_ETHER L3_IPV4 L4_UDP  - l2_len=14 - l3_len=20 - l4_len=8 - VXLAN
packet: packet type =32912, Destination UDP port =2152, VNI = 1459946 -
Receive queue=0x3
  ol_flags: PKT_RX_RSS_HASH PKT_RX_L4_CKSUM_GOOD PKT_RX_IP_CKSUM_GOOD
```

Now, the parser knows that packets with UDP destination port 2152 should be parsed as GTP-U tunnel, and extra fields should be extracted from GTP and inner IP headers.

If the profile is no longer needed, it can be removed from the network adapter and the original configuration is restored.

Figure 22. Removing GTPv1 profile from the device

```
testpmd> port stop 0
testpmd> ddp del 0 /home/pkg/gtp.bak
testpmd> ddp get list 0
Profile number is: 0

testpmd>
```

The `ddp del 0 gtp.bak` command first loads the **`gtp.bak`** file to the memory buffer, then passes it to **rte_pmd_i40e_process_ddp_package()** but with the **RTE_PMD_I40E_PKG_OP_WR_DEL** operation, restoring the original configuration.

4.4 Using GTP protocol with rte_flow API

Generic rte_flow API can be used to steer GTP traffic to different Virtual Functions or queues.

For example, to direct GTP packets with TEID 4 to VF 1, queue 2, an application should use the following rte_flow pattern and action.

Figure 23. rte_flow pattern and actions for directing GTP packets to a VF

```
const struct rte_flow_item pattern [] = {
    {RTE_FLOW_ITEM_TYPE_ETH, NULL, NULL, NULL},
    {RTE_FLOW_ITEM_TYPE_IPV4, NULL, NULL, NULL},
    {RTE_FLOW_ITEM_TYPE_UDP, NULL, NULL, NULL},
    {RTE_FLOW_ITEM_TYPE_GTP, {.teid = 4}, NULL, {. teid = UINT32_MAX}},
};

const struct rte_flow_action actions [] = {
    {RTE_FLOW_ACTION_TYPE_VF, {.id = 1}},
    {RTE_FLOW_ACTION_TYPE_QUEUE, {.id = 2}},
    {RTE_FLOW_ACTION_TYPE_END, NULL},
};
```

Pattern and actions will be parsed by i40e PMD and corresponding tunnel filter entry will be added to direct GTP packets to the VF.

For a case where only queue action is defined, the Flow Director rule will be added.

Figure 24. rte_flow patter and actions to direct GTP packets to a queue

```
const struct rte_flow_item pattern [] = {
    {RTE_FLOW_ITEM_TYPE_ETH, NULL, NULL, NULL},
    {RTE_FLOW_ITEM_TYPE_IPV4, NULL, NULL, NULL},
    {RTE_FLOW_ITEM_TYPE_UDP, NULL, NULL, NULL},
    {RTE_FLOW_ITEM_TYPE_GTP, {.teid = 4}, NULL, {. teid = UINT32_MAX}},
};

const struct rte_flow_action actions [] = {
    {RTE_FLOW_ACTION_TYPE_QUEUE, {.id = 2}},
    {RTE_FLOW_ACTION_TYPE_END, NULL},
};
```

5.0 Summary

This new capability provides the means to accelerate packet processing for different network segments providing needed functionality of the network controller on-demand by applying Pipeline Personalization Profiles. The same underlying infrastructure (servers with already installed standard NICs) can be used for optimized processing of traffic of different network segments (wireline, wireless, enterprise) without the need of resetting NICs/restarting the server.

This capability delivers the following performance and core utilization optimizations:

- Removes requirement for CPU cores on the host to perform classification and load balancing of packet types for the specified use case
- Increases packet throughput, reduces packet latency for the use case

This application note described the use of a GPRS tunneling protocol (GTPv1) profile to enhance performance and optimize core utilization for virtualized enhanced packet core (vEPC) and multi-access edge computing (MEC) use cases.

The application of the GTPv1 Dynamic Device Personalization (DDP) profile in EPC and MEC use cases has been shown to reduce the processing overhead by at least 1 core per application instance, and also to reduce packet latency.