Introduction

Cloud Native is fast becoming a first-class deployment model for next generation network operations. Performance tuning can be difficult to manage in these set ups, however, as gaps in hardware awareness exist at the Orchestration level.

Dynamic Device Personalization (DDP), a feature available in some Intel® Ethernet Controllers, including Intel® Ethernet Controller X710, XXV710 and XL710 network adapters, offers a straightforward way to customize packet processing on a network card, improving throughput and performance.

This document describes how to orchestrate container workloads on a Kubernetes* (K8s*) cluster that can take advantages of these performance improvements by automating the application of DDP profiles and provisioning them to specific containerized applications.

DDP enables certain Intel® Ethernet Controller 700 series network adapters to reprogram its packet processing pipeline to identify new protocols. Once the specific packet type is detected, protocol specific flow rules can be automatically added into its control plane to improve key performance characteristics.

Several different DDP profiles are available to download and to be loaded into an adapter to suit specialized use cases across a broad swathe of network functions.

This document is part of the Network Transformation Experience Kit, which is available at https://networkbuilders.intel.com/network-technologies/network-transformation-exp-kits.
Table of Contents

1 Introduction ................................................................................................................................................. 1
  1.1 Terminology .......................................................................................................................................... 3
  1.2 Reference Documentation ...................................................................................................................... 3
2 Overview ...................................................................................................................................................... 3
  2.1 Challenges Addressed ............................................................................................................................. 4
  2.2 Use Cases ............................................................................................................................................... 4
    2.2.1 GTPv1 for Virtualized Enhanced Packet Core ..................................................................................... 4
    2.2.2 PPPoE for Virtual Broadband Network Gateway .............................................................................. 4
  2.3 Technology Description .......................................................................................................................... 5
    2.3.1 Bare Metal Reference Architecture Container Experience Kit Script ................................................. 5
    2.3.2 SR-IOV Network Device Plugin ........................................................................................................ 5
  2.4 Architecture ........................................................................................................................................... 5
3 Deployment .................................................................................................................................................. 6
  3.1.1 Prerequisites ..................................................................................................................................... 6
  3.1.2 DDP Provisioning in Kubernetes Cluster .......................................................................................... 6
4 Kubernetes Workload Scheduling with DDP .............................................................................................. 10
5 Summary .................................................................................................................................................... 11

Figures
Figure 1. DDP – Packet Classification ............................................................................................................. 4
Figure 2. Architecture Diagram ..................................................................................................................... 5
Figure 3. Workloads Deployment in K8s Cluster with SR-IOV Networking and DDP Profile Request .................. 6

Tables
Table 1. Terminology ....................................................................................................................................... 3
Table 2. Reference Documents ....................................................................................................................... 3
Table 3. Bare Metal Reference Architecture Ansible Script group_vars Configuration Parameters .................. 9
Table 4. Bare Metal Reference Architecture Ansible Script host_vars Configuration Parameters .................. 9
1.1 Terminology

Table 1. Terminology

<table>
<thead>
<tr>
<th>ABBREVIATION</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNFs</td>
<td>Cloud-Native Network Functions</td>
</tr>
<tr>
<td>DDP</td>
<td>Dynamic Device Personalization</td>
</tr>
<tr>
<td>DPDK</td>
<td>Data Plane Development Kit</td>
</tr>
<tr>
<td>GPRS</td>
<td>General Packet Radio Service</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processor Unit</td>
</tr>
<tr>
<td>K8s*</td>
<td>Kubernetes*</td>
</tr>
<tr>
<td>NFV</td>
<td>Network Function Virtualization</td>
</tr>
<tr>
<td>NIC</td>
<td>Network Interface Cards</td>
</tr>
<tr>
<td>OOM</td>
<td>Out of Memory</td>
</tr>
<tr>
<td>PF</td>
<td>Physical Function</td>
</tr>
<tr>
<td>PPPoE</td>
<td>Point-to-Point Protocol over Ethernet</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>SR-IOV</td>
<td>Single-Root Input / Output Virtualization</td>
</tr>
<tr>
<td>vBNG</td>
<td>virtual Broadband Network Gateway</td>
</tr>
<tr>
<td>vEPC</td>
<td>virtual Evolved Packet Core</td>
</tr>
<tr>
<td>VF</td>
<td>Virtual Function</td>
</tr>
</tbody>
</table>

1.2 Reference Documentation

Table 2. Reference Documents

<table>
<thead>
<tr>
<th>REFERENCE</th>
<th>SOURCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Container Experience Kit GitHub* repository</td>
<td><a href="https://github.com/intel/container-experience-kits">https://github.com/intel/container-experience-kits</a></td>
</tr>
<tr>
<td>Container Bare Metal for 2\textsuperscript{nd} Generation Intel\textsuperscript{®} Xeon\textsuperscript{®} Scalable Processor Reference Architecture</td>
<td><a href="https://builders.intel.com/docs/networkbuilders/container-bare-metal-for-2nd-generation-intel-xeon-scalable-processor.pdf">https://builders.intel.com/docs/networkbuilders/container-bare-metal-for-2nd-generation-intel-xeon-scalable-processor.pdf</a></td>
</tr>
<tr>
<td>SR-IOV Network Device Plugin GitHub repository</td>
<td><a href="https://github.com/intel/sriov-network-device-plugin">https://github.com/intel/sriov-network-device-plugin</a></td>
</tr>
</tbody>
</table>

2 Overview

DDP is a customizable packet filtering feature in certain Intel\textsuperscript{®} Ethernet 700 services Network Interface Cards (NICs). DDP, along with the enhanced Data Plane Development Kit (DPDK), support advanced packet forwarding and highly efficient packet processing for cloud and Network Function Virtualization (NFV) workloads. It allows workload specific optimization on the NIC, using a programmable packet-processing pipeline. \(^1\)

Note: For more information on enabling DDP profiles and which profiles are available, refer to: https://www.intel.com/content/www/us/en/architecture-and-technology/ethernet/dynamic-device-personalization-brief.html

Out of the box, these NICs can classify packets for well-known protocols. As the complexity of networking more broadly has grown, so has the number of protocols that underpin network traffic. Identifying combinations of the possible protocols a network is carrying at the correct point is a hard problem, and the lack of information about the structure and nature of these packets can result in reduced performance.

\(^1\) Refer to http://software.intel.com/en-us/articles/optimization-notice for more information regarding performance and optimization choices in Intel software products.
Different protocols will require different actions to be performed—but network packets that are not identified at the NIC hardware level will only have these required actions revealed after being processed by software on the host. Identifying protocols in the network card allows for processing even before reaching the host, potentially increasing performance across several dimensions.

By looking deeper inside a packet, DDP allows a NIC to run with profiles that empower it to classify new protocols. Analysis based on pre-determined profiles allows the NIC to take an early decision on the specific path to be followed by the packet. It does this without needing to travel through the software stack—saving CPU cycles and improving overall throughput. Several profiles are available and network administrators can load custom DDP profiles based on a specific use case.

Figure 1. DDP – Packet Classification

Kubernetes* offers the ability to manage an entire cluster of machines using a single programming interface. With DDP profile enablement, cluster operators can choose what DDP profile to load onto the individual network cards in a cluster. A single DDP profile can program each network card, and every VF using that card as its root will use the same DDP profile when piped through into a container. Workloads can then be declared as users of individual DDP profiles, causing them to be linked to devices with those profiles enabled. This allows an automated, declarative workflow that makes DDP usage scalable and performant in a Kubernetes cluster.

2.1 Challenges Addressed

There are two major facets to orchestrating DDP profiles in K8s:
1. Manage and apply DDP profile packages in K8s running on bare metal deployments.
2. Schedule K8s workloads that request a network interface with a specific DDP profile.

Our solution, which is implemented in publicly available open source projects, addresses both issues and allows:
- DDP profile package management and loading of profiles into NICs on demand
- DDP capability discovery and making K8s aware of available DDP empowered resources
- Dynamic workload placement based on device location

2.2 Use Cases

The following use cases are more generally applicable to profiles on DDP-enabled Intel® Ethernet 700 series network adapters. These features can be enabled to fully Cloud-Native Network Functions (CNFs) using the process described in subsequent sections.

2.2.1 GTPv1 for Virtualized Enhanced Packet Core

GTPv1 is the General Packet Radio Service (GPRS) Tunneling Protocol used for encapsulation of GPRS packets in wireless communications. A key domain for processing this type of packet is in the virtualized Enhanced Packet Core (vEPC), though it is also a common protocol in the Multi-access Edge Computing (MEC) to identify packets using GTPv1 before they hit the machine. The technology allows a reduction in packet processing time in a vEPC User Plane process, by removing the time spent in software queues.

For more on the use of DDP to process GTPv1 packets, refer to Intel® Ethernet Controller 700 Series GTPv1 - Dynamic Device Personalization Technology Guide (refer to Table 2).

2.2.2 PPPoE for Virtual Broadband Network Gateway

Point-to-Point Protocol over Ethernet (PPPoE) is a protocol for encapsulating Point-to-Point frames inside Ethernet Frames. This allows network providers to leverage several Ethernet specific technologies. PPPoE is one of the dominant protocols operating in virtual Broadband Network Gateway (vBNG) use cases at the network core. As with vEPC and GTP workloads, utilizing the PPPoE DDP profile allows for processing to be done in the card itself, keeping packets out of software queues on the target machine.

For information on packet processing using the PPPoE profile on DDP enabled Intel® Ethernet Network 700 Series Adapter, refer to (Table 2).
2.3 Technology Description

This technology guide describes the orchestration of DDP profiles inside a K8s cluster.

Two projects are used to achieve this:
- Bare Metal Reference Architecture script to apply DDP profiles on a cluster.
- Single-Root Input / Output Virtualization (SR-IOV) Device Plugin to manage devices after they have been created.

2.3.1 Bare Metal Reference Architecture Container Experience Kit Script

The Bare Metal Reference Architecture Container Experience Kit script is a set of Ansible* playbooks that provision a bare metal K8s cluster. For high performance networking workloads, it installs and configures some of the required advanced features and components. The script is used to automate the provisioning of DDP profiles and DDP enabled network cards at cluster creation time.

The Bare Metal Reference Architecture Experience Kit is an open source project available at: https://github.com/intel/container-experience-kits

2.3.2 SR-IOV Network Device Plugin

Kubernetes provides a device plugin framework so that devices can be added to manage extended hardware resources such as GPUs, Accelerators, FPGAs or any other PCIe add-ons in a platform.

The SR-IOV Network device plugin is a K8s device plugin that discovers and registers SR-IOV networking resources. This gives Kubernetes a global view of all available SR-IOV Virtual Functions (VFs) available in the cluster.

A workload can request for extended resources in their manifest. The Kubernetes Scheduler can then identify a node that has the required resources available and execute the new workload to that node. The Device Plugin is used to manage DDP enabled virtual functions at run-time in a Kubernetes Cluster.

The SR-IOV Network Device Plugin is an open source project available at: https://github.com/intel/sriov-network-device-plugin

2.4 Architecture

Below is an overall view of a Kubernetes system with DDP enabled SR-IOV virtual functions managed by Kubernetes Device Plugins and CNIs. The key components of the system are labeled and explained below.

Figure 2. Architecture Diagram
• **Intel XXV710 NIC (DDP Enabled)** is an Intel® Ethernet Adapter 700 series network card with a DDP profile applied. The DDP profile is applied at the card level and all VFs from the same card share the same DDP profile.

• **SR-IOV Network Device Plugin** detects which DDP profile the VFs are running on and advertises the VFs with associated DDP profile in the cluster.

• **Multus** handles the creation of pods with multiple network interfaces. In the above example one default flannel interface and one SR-IOV VF interface.

• **SR-IOV CNI** configures individual SR-IOV VFs to be used as interfaces in a pod. The other components shown in the diagram above are standard parts of the K8s cluster, like the K8s API-Server, or common default addons, such as the CNI plugin and Flannel.

### 3 Deployment

This section describes DDP specific configuration and deployment.

This deployment uses the Bare Metal Reference Architecture Container Experience Kit script for provisioning the SR-IOV networking with DDP. Refer to the following link for more details on how to provision your Kubernetes cluster: [https://github.com/intel/container-experience-kits](https://github.com/intel/container-experience-kits).

![Figure 3. Workloads Deployment in K8s Cluster with SR-IOV Networking and DDP Profile Request](image)

#### 3.1.1 Prerequisites

Several software packages are required before running the Bare Metal Reference Architecture script:

- Python v2 present on the target servers.
- Ansible v2.7.16 installed on the Ansible machine (the one you run these playbooks from).
- pip==9.0.3 installed on the Ansible machine.
- SSH keys copied to all K8s cluster nodes (`ssh-copy-id <user>@<host>` command can be used for that).
- Internet access on all target servers is mandatory. Proxy is supported.
- At least 8 GB of RAM on the target servers/VMs for minimal number of functions (some Docker image builds are memory-hungry and may cause Out of Memory (OOM) kills of Docker registry - observed with 4 GB of RAM). You will need more if you plan to run heavy workloads such as NFV applications.

#### 3.1.2 DDP Provisioning in Kubernetes Cluster

1. Clone the Ansible playbook from the git repository:
   ```bash
   # git clone https://github.com/intel/container-experience-kits.git
   # cd container-experience-kits
   ```

2. Copy the example inventory file to the playbook home location:
   ```bash
   # cp examples/inventory.ini .
   ```
3. Edit inventory.ini to reflect the requirements. Here is the sample file. Node names and IP addresses need to be added to provision a new cluster.

```
[all]
node1 ansible_host=10.250.250.161 ip=10.250.250.161
node2 ansible_host=10.250.250.162 ip=10.250.250.162
node3 ansible_host=10.250.250.163 ip=10.250.250.163
node4 ansible_host=10.250.250.166 ip=10.250.250.166
node5 ansible_host=10.250.250.167 ip=10.250.250.167

[kube-master]
node1
node2
node3

[etcd]
node1
node2
node3

[kube-node]
ode4
node5

[k8s-cluster:children]
kube-master
kube-node

[calico-rr]
```

4. Copy group_vars and host_vars directories to the playbook home location:

   ```
   # cp -r examples/group_vars examples/host_vars .
   ```

5. Update group-vars to enable SR-IOV networking related features as follows:

   ```
   ---
   ## BMRA master playbook variables ##
   
   # Intel SRIOV Network Device Plugin
   sriov_net_dp_enabled: true
   sriov_net_dp_namespace: kube-system
   
   # whether to build and store image locally or use one from public external registry
   sriov_net_dp_build_image_locally: false
   ```

6. Update host_vars for all hosts. The SR-IOV NIC configuration are given for each node. In the following example, we've shown a sample configuration from node1 and node2 in their corresponding files host_vars/node1.yml and host_vars/node2.yml.

**Configuration for host_vars/node1.yml:**

```
$ cat host_vars/node1.yml
---
# Kubernetes node configuration

# Enable SR-IOV networking related setup
sriov_enabled: true

# sriov_nics: SR-IOV PF specific configuration list
sriov_nics:
- name: enp24s0f0
  sriov_numvfs: 4
  vf_driver: vfio-pci
  ddp_profile: "gtp.pkgo"
- name: enp24s0f1
  sriov_numvfs: 4
  vf_driver: iavf

sriov_cni_enabled: true

# Set to 'true' to update i40e and i40evf kernel modules
force_nic_drivers_update: true

# install Intel x700 & x800 series NICs DDP packages
```
install_ddp_packages: true

Configuration for host_vars/node2.yml:

```yaml
$ cat host_vars/node2.yml
---
# Kubernetes node configuration

# Enable SR-IOV networking related setup
sriov_enabled: false

# sriov_nics: SR-IOV PF specific configuration list
sriov_nics:
- name: enp35s0f0
  sriov_numvfs: 4
  ddp_profile: "mplsogreudp.pkg"
  vf_driver: vfio-pci
- name: enp35s0f1
  sriov_numvfs: 4
  vf_driver: iavf

sriov_cni_enabled: true

# Set to 'true' to update i40e and i40evf kernel modules
force_nic_drivers_update: true

# install Intel x700 & x800 series NICs DDP packages
install_ddp_packages: true
```

7. **Create a SR-IOV netdp resource config with DDP profile selector in group_vars/all.yml.** This configuration allows the SR-IOV Network Device Plugin to detect which virtual functions are set with what DDP profiles and advertise them as independent resources in the cluster.

```yaml
# Intel SRIOV Network Device Plugin
sriov_net_dp_enabled: true
sriov_net_dp_namespace: kube-system
# whether to build and store image locally or use one from public external registry
sriov_net_dp_build_image_locally: false
# SR-IOV network device plugin configuration.
# For more information on supported configuration refer to: https://github.com/intel/sriov-network-device-plugin#configurations
sriovdp_config_data: |
  |
  "resourceList": [{
    "resourceName": "x700_gtp",
    "selectors": {
      "vendors": ["8086"],
      "devices": ["154c"],
      "ddpProfiles": ["GTPv1-C/U IPv4/IPv6 payload"]
    }
  },
  |
  "resourceName": "x700_pppoe",
  "selectors": {
    "vendors": ["8086"],
    "devices": ["154c"],
    "ddpProfiles": ["E710 PPoE and PPPoL2TPv2"]
  }
  ]
```

8. **Deploy Bare Metal Reference Architecture cluster with Ansible playbook:**

```bash
$ ansible-playbook -i inventory.ini playbooks/cluster.yml
```

9. **Once the playbook has completed successfully,** we can check to confirm the SR-IOV device plugin is running and seeing the correct DDP Profiles on the machine. This can be done by running:

```bash
$ kubectl describe node node1 | egrep -i "Capacity|Allocatable" -A 20
```
The result should look something like the following with `intel.com/x700_gtp` and `intel.com/x700_pppoe` advertised individually on nodes where they are available:

<table>
<thead>
<tr>
<th>Capacity:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu:</td>
<td>56</td>
</tr>
<tr>
<td>ephemeral-storage:</td>
<td>238165816Ki</td>
</tr>
<tr>
<td>hugepages-1Gi:</td>
<td>0</td>
</tr>
<tr>
<td>hugepages-2Mi:</td>
<td>8Gi</td>
</tr>
<tr>
<td>intel.com/x700_gtp:</td>
<td>10</td>
</tr>
<tr>
<td>intel.com/x700_pppoe:</td>
<td>0</td>
</tr>
<tr>
<td>memory:</td>
<td>72897180Ki</td>
</tr>
<tr>
<td>pods:</td>
<td>110</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Allocatable:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu:</td>
<td>54</td>
</tr>
<tr>
<td>ephemeral-storage:</td>
<td>219493615663</td>
</tr>
<tr>
<td>hugepages-1Gi:</td>
<td>0</td>
</tr>
<tr>
<td>hugepages-2Mi:</td>
<td>8Gi</td>
</tr>
<tr>
<td>intel.com/x700_gtp:</td>
<td>10</td>
</tr>
<tr>
<td>intel.com/x700_pppoe:</td>
<td>0</td>
</tr>
<tr>
<td>memory:</td>
<td>63357596Ki</td>
</tr>
<tr>
<td>pods:</td>
<td>110</td>
</tr>
</tbody>
</table>

### Table 3. Bare Metal Reference Architecture Ansible Script group_vars Configuration Parameters

<table>
<thead>
<tr>
<th>VARIABLE</th>
<th>OPTIONS/DEFAULTS</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>sriov_net_dp_enabled</td>
<td>Boolean, default “true”</td>
<td>Deploy SR-IOV network device plugin daemonset in the cluster.</td>
</tr>
<tr>
<td>sriov_net_dp_namespace</td>
<td>String, default “kube-system”</td>
<td>Kubernetes namespace to be used for SR-IOV Network Device Plugin deployment.</td>
</tr>
<tr>
<td>sriov_net_dp_build_image_locally</td>
<td>Boolean, default “true”</td>
<td>Build and host SRIOV Network Device Plugin image in the cluster local Docker registry, set to false to use SRIOV Network Device Plugin image from Docker Hub.</td>
</tr>
<tr>
<td>sriovdp_config_data</td>
<td>String in valid json format. Default values are located in: “roles/sriov-dp-install/defaults/main.yml”</td>
<td>Resource configuration for SR-IOV network device plugin. For configuration details, refer to: <a href="https://github.com/intel/sriov-network-device-plugin#configurations">https://github.com/intel/sriov-network-device-plugin#configurations</a></td>
</tr>
</tbody>
</table>

### Table 4. Bare Metal Reference Architecture Ansible Script host_vars Configuration Parameters

<table>
<thead>
<tr>
<th>VARIABLE</th>
<th>OPTIONS/DEFAULTS</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>sriov_nics</td>
<td>Array of composite values in the following format. name: string – PF interface name sriov_numvfs: int - Number of VFs vf_driver: string - VF driver to be attached for all VFs under this PF ddp_profile: string – the DDP package name</td>
<td>Host SR-IOV NIC configuration. Each configuration describes the targeted Physical Function (PF), the number of Virtual Functions to be created for that PF, the VF driver to be attached for all VFs under this PF, and optional DDP package name to be loaded into the NIC. Note: A DDP package can be loaded into a NIC with first PF only. If the PF is not the first PF, then this value will be ignored by the script.</td>
</tr>
<tr>
<td>sriov_cni_enabled</td>
<td>Boolean</td>
<td>Whether to install SRIOV CNI plugin on target node. This is should be set to true for enabling SR-IOV networking.</td>
</tr>
<tr>
<td>install_ddp_packages</td>
<td>Boolean. Default: “false”</td>
<td>Whether to download and install available DDP packages on each node.</td>
</tr>
<tr>
<td>force_nic_drivers_update</td>
<td>Boolean. Default: “false”</td>
<td>Whether to update NIC drivers. Note: For DDP support, the i40e driver and firmware must be updated to the supported versions or above.</td>
</tr>
</tbody>
</table>
At this point, we have made SR-IOV VF available in the cluster. They are advertised as available to workloads by the SR-IOV Network Device Plugin. To attach a workload to one of these VFs, however, we must first create a Network Attachment Definition which describes the resource required and its configuration.

1. **Create SR-IOV net-attach-def CRs:**

   ```bash
   cat <<EOF | kubectl apply -f -
   apiVersion: "k8s.cni.cncf.io/v1"
   kind: NetworkAttachmentDefinition
   metadata:
     name: sriov-net-gtp
   annotations:
     k8s.v1.cni.cncf.io/resourceName: intel.com/x700_gtp
   spec:
     config: '{
         "cniVersion": "0.3.1",
         "type": "sriov",
         "name": "sriov-gtp"
       }'
   EOF
   ``

   With our network configuration now registered with Kubernetes, we can apply a pod that calls for this network by name in an annotation.

2. **Deploy a pod with SR-IOV VF requesting from the x700_gtp resource pool:**

   ```bash
   cat <<EOF | kubectl apply -f -
   apiVersion: v1
   kind: Pod
   metadata:
     name: testpod-gtp
   labels:
     env: test
   annotations:
     k8s.v1.cni.cncf.io/networks: sriov-net-gtp
   spec:
     containers:
     - name: gtpcntr1
       image: centos/tools
       imagePullPolicy: IfNotPresent
       command: [ "/bin/bash", "--", "--" ]
       args: [ "while true; do sleep 300000; done;" ]
       resources:
         requests:
           intel.com/x700_gtp: 1
         limits:
           intel.com/x700_gtp: 1
   EOF
   ``

   When this pod is created, the K8s* scheduler will look for nodes that have available VF resources with the above name—that is, VFs from an Intel® Ethernet Adapter with the correct DDP profile applied. It then sends a pod creation request to a feasible node where the network is configured, and the specific virtual function attached to the workload.

3. **We can check that a pod using the prescribed DDP profile is created with:**

   ```bash
   $ kubectl get pods --all-namespaces
   NAMESPACE   NAME            READY STATUS    RESTARTS AGE   IP               NODE
   default      testpod-gtp   1/1   Running   0 45s    10.237.222.3 node1 <none>
   <none>
   ```
5 Summary

This guide demonstrates the provisioning and management of DDP enabled Virtual Functions in cloud orchestrator Kubernetes. DDP eliminates the need for the CPU to perform certain kinds of work, such as load balancing or packet classification, on specific widely used network protocols. This reduces latency by keeping network packets out of queues.

The automated workflow described in this document requires little intervention by cluster administrators to put the advanced performance capabilities of DDP devices to work. End users can modify and customize the associated documents and configurations from open source repositories according to need and are encouraged to do so.

Following the outlined steps, cloud network functions can be provisioned quickly to take advantage of DDP technology. This allows high performance and improves utilization not just on the network card, but on the overall hardware platform.