USER GUIDE

Intel Corporation

Enhanced Utilization Telemetry for Polling Workloads with collectd and the Data Plane Development Kit (DPDK)

Authors

David Hunt

Lee Daly

Reshma Pattan

Chris MacNamara

John Browne

Emma Collins

1 Introduction

Understanding the utilization of a platform and application is key to understanding current load and potential overload conditions, giving the network operator an opportunity to monitor and react to meet Service Level Agreements (SLAs).

This paper details new telemetry allowing an operator to understand:
- Platform load and overload
- Application load and overload

In addition, in many cases, applications are deployed in a polling mode to achieve low latency and high throughput. Polling queues, Network Interface Controllers (NICs), and accelerators are just some examples. This paper describes the Data Plane Development Kit (DPDK) use case and provides a solution to the challenge of understanding the amount of work being performed versus CPU utilization specifically when an application is always polling.

For polling applications, operating systems report utilization as 100% busy irrespective of the true traffic rate or the useful work done. The DPDK v19.08 release added a standard metric that applications can populate to inform an orchestration system of busyness or the detection of load and overload situations.
The Management/Orchestration layer, such as Virtual Network Functions Manager (VNFM), Virtualized Infrastructure Manager (VIM), Network Functions Virtualization Orchestrator (NFVO), Software Defined Networking (SDN) controller or analytics systems can then respond by taking the appropriate actions.

Table of Contents

Figures

Tables

1.1 Intended Audience

The intended audience includes developers, engineers, and architects building platforms and monitoring systems for DPDK-based network functions.

The intention of this document is to focus on the ingredients that can be included in a monitoring solution. The goal is to highlight the telemetry enhancements for those interested in monitoring platform metrics and show how Network Functions based DPDK solutions can be enhanced to provide new insights. The upstream components modified include the DPDK and collectd*, with the remaining components being enhanced solely for the purposes of the use case described in this user guide.

1.2 Terminology

Table 1. Terminology

ABBREVIATION	DESCRIPTION
BIOS	Basic Input Output System
DPDK	Data Plane Development Kit
EAL	Environmental Abstraction Layer (in DPDK)
GPSS	General Purpose Simulation System
GUI	Graphical User Interface
JSON*	JavaScript* Object Notation
NFVO	Network Functions Virtualization Orchestrator
NIC	Network Interface Controller
PCPS	Per Core P-States
PCU	Power Control Unit
QSFP (or QSFP+)	Quad Small Form-factor Pluggable (fiber optical transceiver)
SDN	Software Defined Networking
SLA	Service Level Agreement
TSC	Time Stamp Counter
VF	NIC Virtual Function
VIM	Virtualized Infrastructure Manager
VM	Virtual Machine
VNFM	Virtual Network Functions Manager

1.3 Reference Documents

Table 2. Reference Documents

REFERENCE	SOURCE
collectd	https://collectd.org/
DPDK L3 Forwarding with Power Management Sample Application	http://doc.dpdk.org/guides/sample_app_ug/l3_forward_power_man.html
Grafana Analytics & Monitoring	https://grafana.com/
Prometheus monitoring system & time series database	https://prometheus.io/
Telemetry ReadMe	https://github.com/intel/CommsPowerManagement/blob/master/telemetry/README.md
TRex Traffic Generator	https://trex-tgn.cisco.com/

2 Observing the Benefits of Enhanced Telemetry

Using the method described in this user guide, overload conditions can be detected before packet drops occur, improving the SLA for the service. Early detection also means that corrective actions, such as horizontal scaling actions in the VIM, can be triggered.

Figure 1 shows the Grafana* dashboard that provides a visual representation of the use of enhanced telemetry for polling workloads using the DPDK and collectd. The dashboard shows that the system goes into an overloaded state at three different peak points.

At each peak point, an action is taken to scale up another Network Function (NF), virtual machine (VM) or container, to allow the incoming traffic to be divided across the available NFs, reducing the busyness of the overloaded NF.

In addition, it is possible to use running averages to determine if the NFs are overloaded compared to a user set threshold for expected average loads. For example, the threshold could be set to an arbitrary value, such as 70% of the maximum.

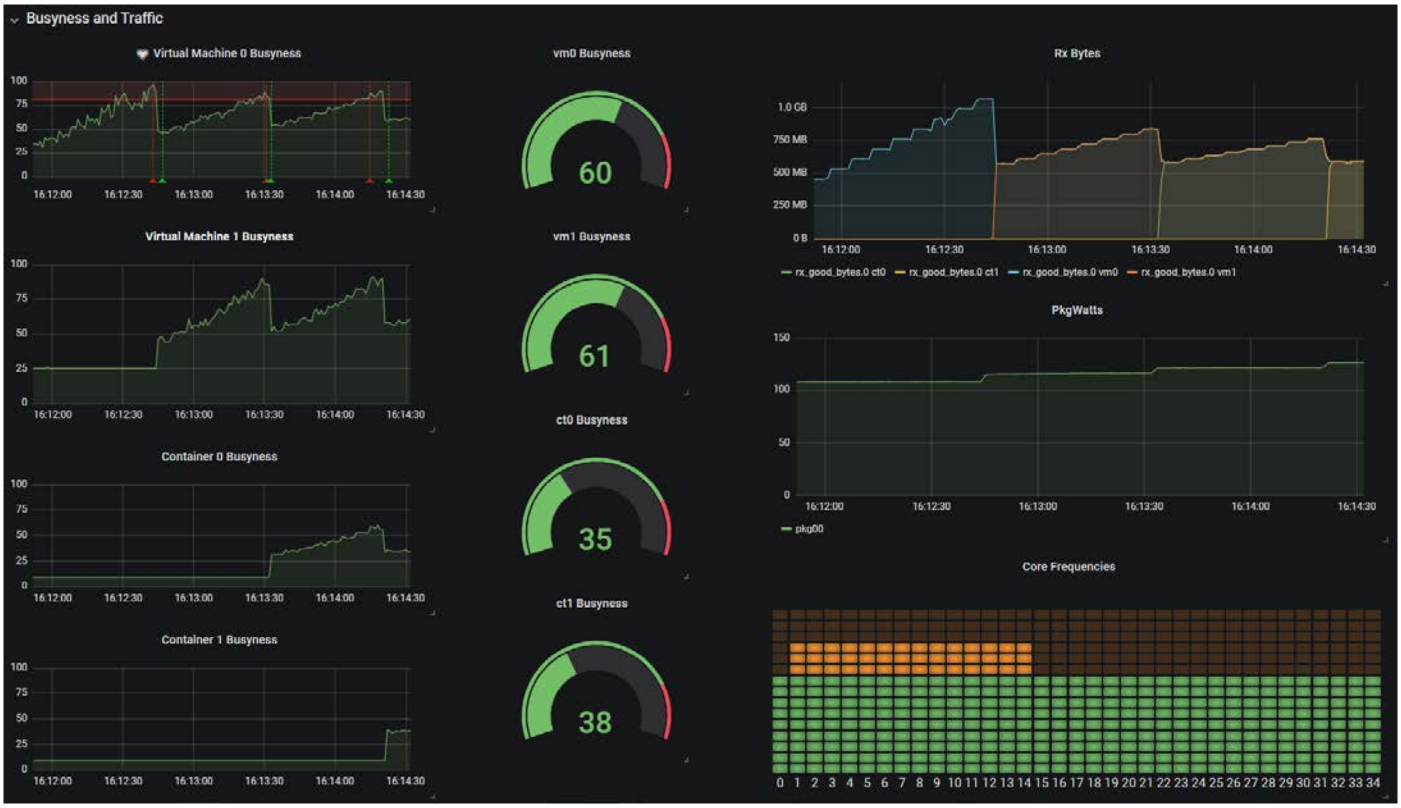

Figure 1. Grafana Dashboard Example Display

Figure 1 shows several workloads, including two VMs and two Containers, with a busyness indictor in the center of the dial for each workload. The busyness metric is new and the subject of discussion in this paper. The figure gives a detailed view of the utilization produced by each DPDK application and is a key enhancement. In previous generations of DPDK, this busyness metric did not exist, and the user had to rely on CPU utilization, which showed 100%, limiting the insight and actions that could be taken.

In addition to network function busyness, the GUI shows activity in three domains:

1. Networking traffic, indicated by Rx Bytes
2. Power, indicated by CPU watts (`PkgWatts`)
3. Performance, indicated by core frequency

This adds a second layer of insight based on platform metrics and allows the user to observe important platform-level load metrics and take actions based on these metrics. Further details are explained throughout this paper.

3 Load and Overload Usage

To improve power management in a Network Function Virtualization (NFV) environment, there are two key metrics that allow a system to decide whether it is in a steady state or it needs to act due to an overload condition. Table 3 shows the metrics and the actions they prompt depending on their states.

Table 3. Key Metric Actions

METRIC#1 BUSY INDICATIONS	METRIC#2 PKGWATTS	ACTION
OK	OK	Steady State
Overload	OK	Back-off
OK	Overload	Back-off

The actions that can be taken due to an overload in application busyness or due to the `pkgwatts` value going beyond a pre-set threshold can indicate to a load balancer or other traffic steering mechanism that action is required to prevent the system entering a phase where packet loss may occur.

The use case described in this user guide shows how the busyness telemetry, which was added in DPDK v19.08, can be used to influence a load balancer, causing it to add additional NFs to help increase the overall throughput of the system, thereby avoiding a packet loss situation. The increase in power as more CPU cores are brought on-line to handle the increasing workload is also shown.

4 Example that Exercises the Telemetry

Figure 2 shows the use case selected to demonstrate how the busy indication can be used to influence Load Balancer traffic.

The following steps describe the flow:

1. collectd gathers data, including `pkgwatts`, from the platform metrics and a busy indication from a VM running a DPDK application.
2. Prometheus*/Grafana* receives these metrics and indicators from collectd.
3. Prometheus/Grafana directs any node or application overload alerts to the Network Load Controller.
4. The Network Load Controller determines the correct action to take in response to the alert and sends a steering decision to the Load Balancer to implement the appropriate action.

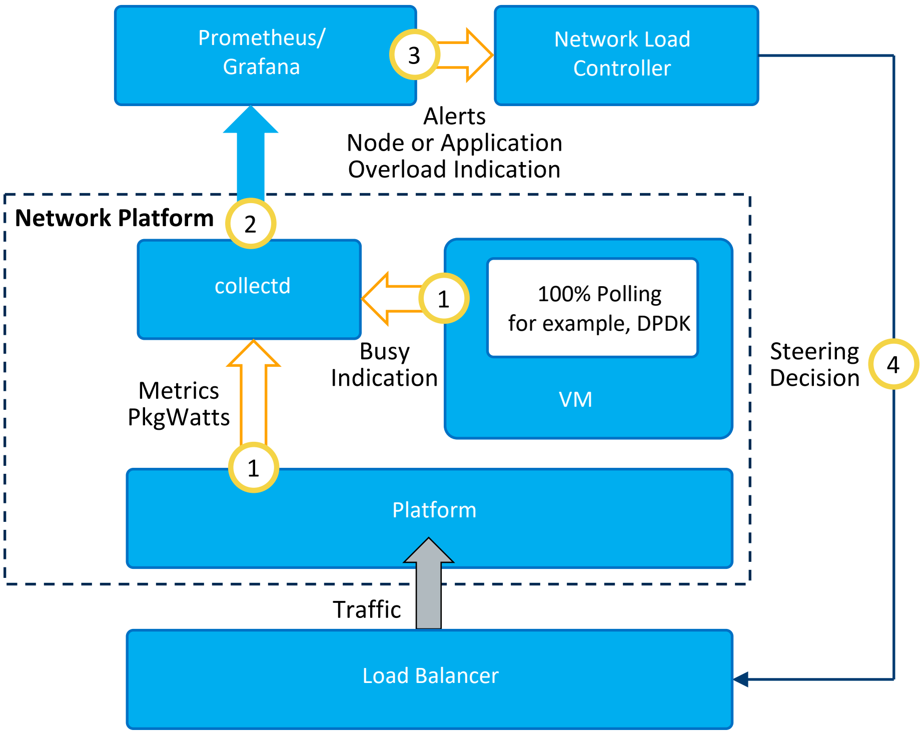

Figure 2. Experience Kit Use Case High-level Block Diagram

5 Use Case Components

Table 4 describes the primary components and their functions.

Table 4. Use Case Components

COMPONENT	DESCRIPTION
DPDK L3 Forwarding with Power Management Sample Application	Provides a busy indication metric to collectd.
collectd (including the DPDK Telemetry Plugin and the `turbostat` plugin)	Gathers metrics from DPDK applications including platform power and frequency metrics.
Prometheus* (including collectd and node exporters)	Gathers metrics from collectd and the platform and presents them in a format that is easily consumable by Grafana.
Grafana* (including `AlertManager`)	Graphically displays metrics, dials, graphs, alerts, and so on. Also, Grafana alerts on pre-defined busy indicators and sends the alerts to the Network Load Controller.
Network Load Controller	Listens for alerts from Grafana and implements the business logic to reconfigure the Load Balancer based on overload conditions.
Load Balancer (a TRex traffic generator with a Python control API)	Generates traffic for the platform at various levels depending on the configuration set by the Network Load Controller.

6 Software Components

The metrics software involves various components, as shown in Figure 3. The subsections following provide an overview of the setup for each component.

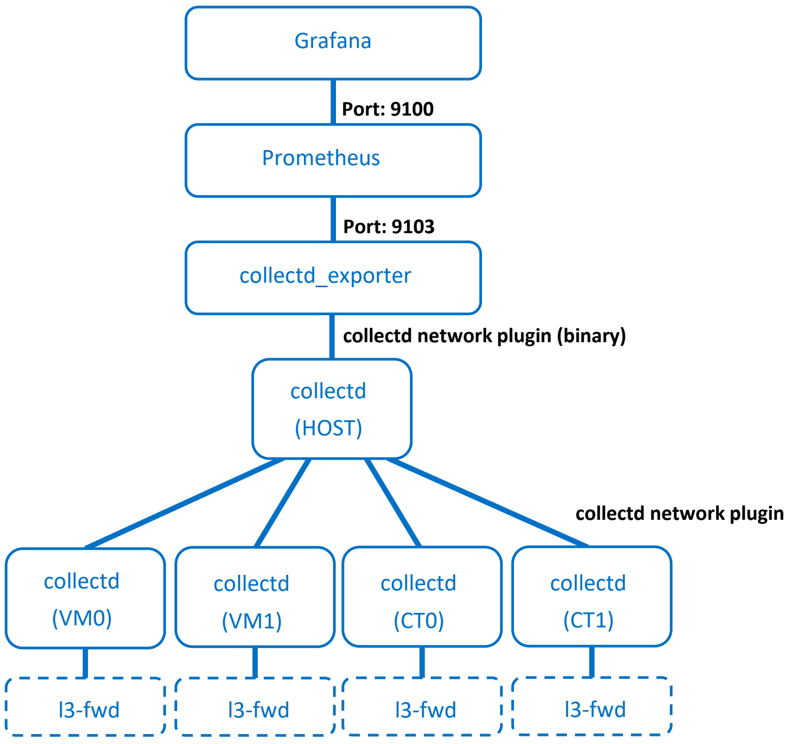

Figure 3. Metrics Software Block Diagram

6.1 collectd Setup

In each VM, collectd is cloned from https://github.com/collectd/collectd and built in the VM. The `dpdk_telemetry` plugin is enabled, and the network plugin is configured to pass on any incoming metrics to the master collectd on the host, as shown in Figure 3.

For the container setup, collectd is built on the host operating system, and is run in the containers using the `docker exec` command.

This results in five instances of collectd in total. Two for the VMs, two for the containers, and one collectd master instance, which gathers metrics from the other four instances and passes them on to the `collectd_exporter`.

The `collectd.conf` configuration file is in the collectd repo.

For `dpdk_telemetry` configuration, extracts are shown here for reference.

```
LoadPlugin dpdk_telemetry

<Plugin dpdk_telemetry>
    ClientSocketPath "/var/run/.client"
    DpdkSocketPath "/var/run/dpdk/rte/telemetry"
</Plugin>
```

Using the `turbostat` plugin, the following configuration is applied. The plugin values are uncommented to allow collectd to retrieve the relevant metrics for `PkgWatts` (CPU watts) and per-core frequency as shown in the Dashboard in Figure 1.

```
LoadPlugin turbostat

<Plugin turbostat>
##      None of the following option should be set manually
##      This plugin automatically detect most optimal options
##      Only set values here if:
##      - The module asks you to
##      - You want to disable the collection of some data
##      - Your (Intel) CPU is not supported (yet) by the module
##      - The module generates a lot of errors "MSR offset 0x… read failed"
##      In the last two cases, please open a bug request
#
#       TCCActivationTemp "100"
#       CoreCstates "392"
#       PackageCstates "396"
#       SystemManagementInterrupt true
#       DigitalTemperatureSensor true
#       PackageThermalManagement true
#       RunningAveragePowerLimit "7"
</Plugin>
```

6.2 Selected Metrics for Low Latency Performance

The `turbostat` plugin interrogates a wide range of metrics including platform, per-CPU and per-core metrics. In very low latency use cases, this level of interrogation adds latency to workers as metrics are collected per core. There are cases where per-core metric retrieval can disrupt the application-- as code is metric collection code is scheduled on the same core as the workload in some cases adding latency. Therefore, in very low latency scenarios, it is recommended to customize the metrics set to only those that are required for the specific use case. One approach to achieve this is to use the `python collectd` plugin. Enabling the `python collectd` plugin and importing a customized script allows the user to deploy selected metrics.

This approach is not used in the setup described in this user guide, but it is worth considering for deployment scenarios where the 20-40 µs of latency is important.

An example of this approach is available on `github` at the following link:

https://github.com/intel/CommsPowerManagement/blob/master/telemetry/pkgpower.py

Refer to *Telemetry Readme* (refer to Table 2) for more information, specifically, the step to enable the `python collectd` plugin in the `collectd.conf` file, and the step to import a script with customized metrics.

6.3 Prometheus Setup

The Prometheus* setup involves downloading the relevant tar file from https://promethus.io/download. Once the archive is extracted, the relevant sources must be added to the `prometheus.yml` configuration file. In our case, the primary source is the `collectd_exporter`, which we have configured to listen on port 9103. This allows Prometheus to gather all the metrics from the main `collectd` instance. The line added to the `prometeus.yml` file is as follows:

```
  - targets: ['localhost:9103']
```

Once the Prometheus binary starts, it begins scraping metrics from the sources, including the `collectd_exporter`.

When we start Prometheus, we also provide a listener address using the `--web.listen-address=:9010` parameter, which is the port used to pull the metrics from Prometheus to Grafana.

6.4 Grafana Setup

Since the options for Grafana configuration are hugely variable, this section provides some examples of the configuration settings used when setting up the use case. Installation is achieved using a package manager (for example, `yum`, `dnf`, or `apt`) to get and install the `grafana` package. Once installed, the Grafana server is running, and most likely listening on port 3000. You can check the `/etc/grafana/grafana.ini` file for confirmation or reconfigure appropriately. Point a browser to the Grafana dashboard at `http://servername:3000`. To start a new dashboard, refer to the *Grafana Analytics & Monitoring* documentation at (refer to Table 2).

6.4.1 Data Sources

Our use case uses the `collectd_exporter` to take metrics from collectd and insert them into Prometheus. Therefore, we must set up a Prometheus source in Grafana. The two key configuration settings for the data source are:

- The URL
- The scrape interval

For the URL, we point at the Prometheus port using `http://localhost:9090`, and because we want quick updates on the dashboard, we set the scrape interval to be 1s (one second).

6.4.2 Queries

Once the source is configured and working, adding metrics to widgets is as simple as typing in the metric name into the query associated with the widget. For example, if you create a simple dashboard, add a graph widget, and start typing "**collectd**", the available metrics appear. For our use case, we are interested in `busy_percent`, which is accessed by:

```
collectd_dpdk_telemetry_total{dpdk_telemetry="busy_percent",exported_instance="xx"}
```

where `xx` is one of: `vm0`, `vm1`, `ct0`, or `ct1`, which are the host names assigned to each NF running on the host.

Because most of the metrics are at web endpoints, we can point a browser at each exporter to see the metrics they provide. For example, the `collectd_exporter` makes its metrics available at port 9103. Therefore, we simply point a browser at `http://server/9103` and we are presented with all the metrics provided by `collectd` through Prometheus.

6.4.3 Actions Based on Telemetry

Once we have the graphs for the busyness set up and running, decide the threshold to set. Once that threshold is exceeded, Grafana triggers an alert, which is caught by our Python script that splits the traffic.

1. Define a notification channel for the dashboard.
   This allows us to create a channel through which alerts arrive when the thresholds are crossed. We create these thresholds later.
2. Create the new webhook notification channel and set the URL to `http://localhost:9201/alert/`.
   This URL is the endpoint created by our alert handler Python script, which listens for alerts on port 9201. The channel edit screen also allows testing, which sends a test alert to the alert handler endpoint.
3. Add a threshold to the graph(s) we are interested in alerting on.
   While editing the widget (in our case, a graph widget for busyness), we edit the alert configuration to evaluate every 1s, and when the average of the last 5 seconds is above a value of 80, a notification is sent to the alert channel.

Note: When editing the notifications, the list of available channels is shown, and select the appropriate channel for our alert.

The graph in Figure 4 shows a red area above 80%, which is the area in which an alert is generated.

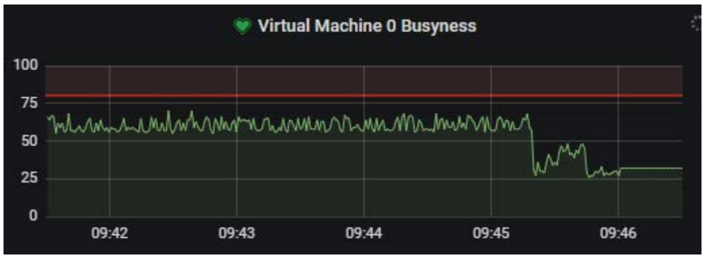

Figure 4. Sample Graph with Threshold Set at 80%

When the metric value goes above the alarm threshold, the graph outline pulsates, giving a clear indication of an alarm condition (see Figure 5), and a JavaScript* Object Notation (JSON*) message is sent to the webbook endpoint (Network Load Controller).

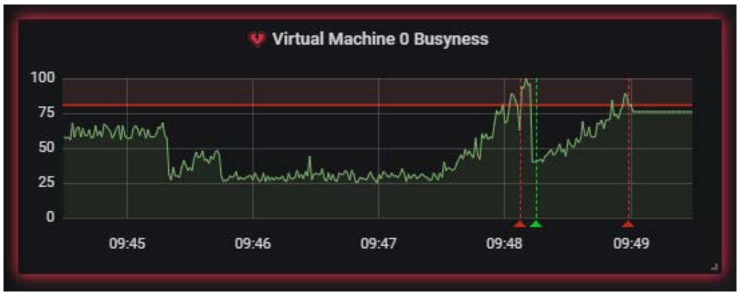

Figure 5. Sample Graph with an Alarm State

6.5 Application Setup

We ran the `l3fwd-power` application in both the VMs and the containers. The build procedure is slightly different for each:

- For the VMs, the DPDK is cloned and built from within the VM. The configuration file (`dpdk/config/common_base`) was modified to set the `CONFIG_RTE_LIBRTE_TELEMETRY` parameter to **y**, and the application was built.
- For the containers, the DPDK was built on the host, then executed in each container using the *`docker exec`* command.

In all cases, when running the application as part of this use case, the `--telemetry` option was passed in both the Environmental Abstraction Layer (EAL) and the application-specific command line options so that the application increments the relevant counters and makes them available to the relevant collectd instance.

For more information on building sample DPDK applications, see https://doc.dpdk.org/guides/linux_gsg/build_sample_apps.html.

6.6 Network Load Controller Setup

The Network Load Controller takes the form of a simple Python script, which acts as a Flask web endpoint listener at one end and calls the T-Rex packet generators Python automation API at the other end. It listens for alerts from Grafana and takes action to change the number of streams sent from T-Rex to the NFs.

On startup, the Network Load Controller initializes a single stream of traffic and sends it to the first VM. The initial traffic transmission rate is 10% of the line rate. The DPDK application updates the busyness metric, which then goes to Grafana through collectd. A separate thread listens for alerts from the Grafana Alert Manager.

The Network Load Controller gradually increase the traffic rate of that initial stream, at a rate of +1% every few seconds.

At the same time, the listener thread awaits an alert that a stream is overloaded, which indicates that an application's busyness indicator has exceeded the 80% threshold. When an alert arrives, an additional traffic stream is added, and the rate is reconfigured to be split evenly between the streams. This has the effect of moving traffic away from the overloaded NF to a previously unused instance.

The traffic gradually increases to 99%, at which stage all NFs (two VMs and two containers) are busy handling traffic. At 100%, the Network Load Controller resets the number of streams back to one, and reduces the traffic rate back to 10%, to repeat the rate increase all over again.

7 Observations

These tests were completed by Intel in June 2020.[1]

Figure 6 shows Grafana* output that demonstrates the traffic increase, the alerts being generated based on the busyness indicator metric from DPDK applications, and the traffic streams being split and reallocated to the additional NFs. The figure shows the four busyness charts; two for the VMs, and two for the containers. From left to right, the effect of the Network Load Controller is seen as the traffic is split into an increasing number of streams as a result of the generated alerts. At the first alert, we can see the second VM come online and start handling traffic. That increases for both VMs until the next alert, at which stage the traffic is split into three, and the first container starts handling traffic (about halfway along the timeline). Finally, when the third alert occurs, the final NF (the second container) starts handling traffic.

Note: The vertical red and green lines in the `vm0` chart indicate when the busyness for `vm0` goes in and out of an alarm state.

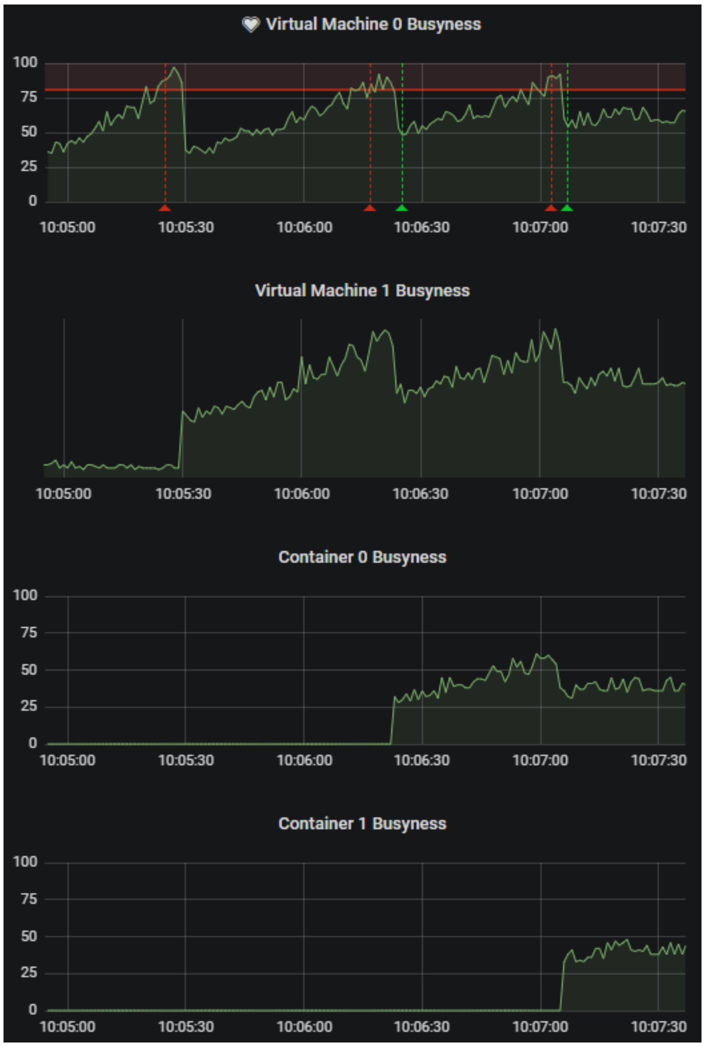

Figure 6. Traffic Stream Split Example

Figure 7 shows a different view of the same events. This view shows the received bytes for each NF. The figure shows the rate for all four traffic streams, and each split is seen as each alert occurs. Initially, `vm0` handles all the traffic. When `vm0` overloads beyond 80%, the alert generated causes the traffic to be split, and `vm0` and `vm1` share the traffic (indicated by the blue and orange lines). The next alert causes the number of streams to increase to three, and container 0 (green line) then takes its share of the traffic. Finally, container 1 (yellow line) takes some of the load when the final alert is triggered.

[1] Refer to Appendix A for configuration details. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks. Refer to http://software.intel.com/en-us/articles/optimization-notice for more information about optimization.

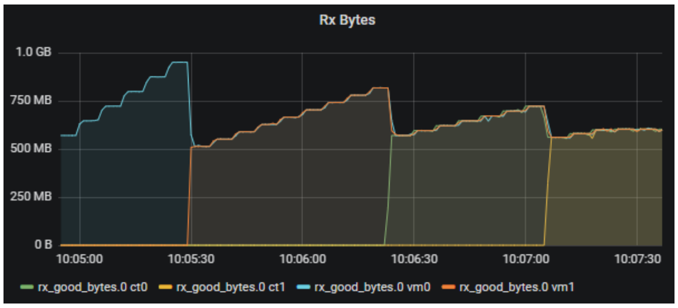

Figure 7. Received Bytes Chart

Figure 10 demonstrates the power increase as the system gets overloaded and the cores on which an NF resides are scaled up in frequency. From the graph we can see the first power increase comes from the point at which `vm0` has become overloaded and the cores on which `vm1` are running have been scaled up in anticipation for the traffic splitting to come. This is repeated as `vm1`, `ct0` and `ct1` become overloaded.

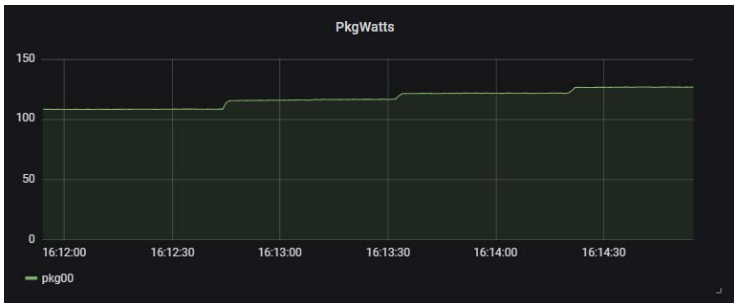

Figure 8. CPU Power

8 Conclusions

The busyness/overload indicator provided by the DPDK in the 19.08 release is a valuable telemetry metric for orchestration systems that need to respond to traffic overload conditions. This user guide focuses on one specific use case. It describes how to use the busyness/overload indicator to influence a Load Balancer. From this single use case, it is hoped that the reader can imagine other use cases where this type telemetry could be useful for, scale-up, scale-down, or scale-out, and so on.

Appendix A Hardware, BIOS and System Setup

A.1 Hardware Configuration

Figure 9 shows the hardware configuration used to model the use case. The configuration consists of a dual-socket Intel® Server Board S2600WFQ system, where one socket is dedicated to traffic generation and metrics handling, and the other socket is dedicated to handling workloads for two VMs and two containers. Each VM and container runs the DPDK `l3fwd-power` sample application.

For each socket, there is an Intel® Ethernet Converged Network Adapter XL710 10/40 GbE with Quad Small Form-factor Pluggable (QSFP+) optical transceiver support. One adapter is for traffic generation from the TRex traffic generator. The other adapter is for traffic consumption and distribution to the VMs and containers.

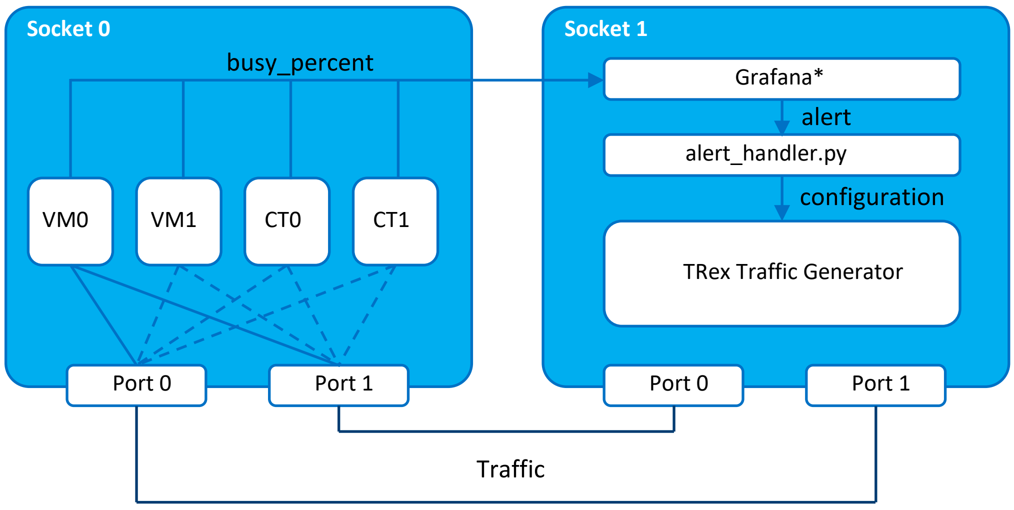

Figure 9. Use Case Hardware Configuration

Table 5. Hardware Components for Tests

ITEM	DESCRIPTION	NOTES
Platform	Intel® Server Board S2600WT	Serial: WO79620L01S014
Processor	2x Intel(R) Xeon(R) Gold 6139	@ 2.30 Ghz
NIC	2x Intel Corporation Ethernet Controller XL710	40GbE QSFP+
BIOS	Intel Corporation SE5C620.86B.00.01.0015.110720180833 Release Date: 11/07/2018	

A.2 Software Configuration

Table 6. Software Components for Tests

SOFTWARE COMPONENT	DESCRIPTION
Linux* Kernel	Kernel: **5.3.11-100.fc29.x86_64**
Data Plane Development Kit (DPDK)	DPDK v19.11
Collectd	collectd 5.11
Prometheus	Prometheus 2.12.0
Grafana	Grafana 6.3.4
Trex	Trex 2.61
Workloads: L3 Forward Power	L3 forward power is a Data Plane Development Kit (DPDK)-based application, which forwards traffic from T-Rex.

A.3 BIOS Configuration

We set up the system for deterministic performance using the BIOS settings in the following table.

Table 7. BIOS Settings

BIOS SETTING	REQUIRED VALUE
HyperThreading	Disabled
CPU Power and Performance Policy	Performance
Workload Configuration	I/O Sensitive
Enhanced Intel® Speedstep Technology	Enabled
Hardware P-States	Enabled
Intel® Turbo Boost Technology	Enabled
Package C-States	C0/C1 State
C1E	Enabled
Processor C6	Enabled
Uncore Frequency Scaling	Enabled
Performance P-Limit	Disabled
Intel® Virtualisation Technology (VT)	Enabled
Intel® VT for Directed I/O (VTd)	Enabled

Note: Some BIOS variants use the term *Per Core P States (PCPS)* instead of *P states* and configuration options reflect that terminology.

A.4 Host and Network Function Setup

System setup encompasses host setup, VM setup, and container setup. The term *Network Function* is used to indicate a VM or container. The following sections provide an overview of the setup. These sections are not intended to be a step-by-step guide to replicating the use case.

Figure 10 shows a simplified version of the Operating System setup with the VMs and containers. The setup for each component is expanded in the subsections that follow.

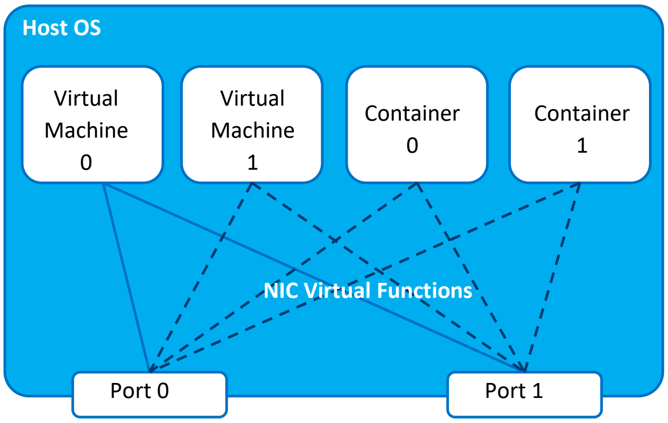

Figure 10. Operating System and Network Function Setup Block Diagram

A.4.1 Host Setup

Some important host configuration steps are required before starting the virtual environments. The workloads need isolation from the kernel, therefore, the host is configured using the kernel boot parameter `isolcpus=cpu_list` to isolate the `l3fwd-power` application cores that are used in VMs and containers. Once the system is up, install the `uio` kernel module, DPDK kernel module `igb_uio`, set up the hugepages, and set up the VFs on the physical devices.

A.4.2 Virtual Machine Setup

The VMs are set up using the `qemu-kvm` emulator with a Fedora 30 image. The VM cores are pinned to a subset of the isolated CPUs in the `isolcpus` list in the host OS, specifically those cores that run the `l3fwd-power` application. Also, the cores are allocated on the socket with direct connection to the Network Interface Controller (NIC) handling the traffic. This avoids any impact resulting from the sending of traffic over the QPI link.

In the host OS, we created virtual functions on the NIC, and allocate two VFs to each VM to give us performant channels for packets coming from the NIC into the VM and from the VM out to the NIC.

The following steps are performed in each VM:

- Add the NIC as a PCI pass-through device so that device control passes to the VM
- Assign each VF a unique MAC address
- Pin virtual cores to isolated physical cores
- Allocate hugepages for use by the DPDK application
- Install the `uio` and `igb_uio` kernel drivers
- Bind the virtual functions to the DPDK `igb_uio` driver

A.4.3 Container Setup

When using containers, it is important to remember that an application still runs on the host machine, but in a contained environment. Containers do not provide virtualization. In this experience kit, we use Docker* to set up the containers.

To run the `l3fwd-power` application in a container, the host must have the `igb_uio` and `uio` modules installed and `hugepages` set up correctly.

We used the standard procedure to create the Docker container to build and run the `l3fwd-power` application.

Similarly, NIC virtual functions are allocated to each container, in the same manner as in the VM setup, for performant networking.

0820/DN/PTI/PDF 627392-001US