A Project Report Submitted

in the Partial Fulfilment of the Requirements

for the Degree of

Master Of Technology

by

Prateek Agarwal

Computer Science and Engineering

Indian Institute of Technology Bombay

June, 2021

Contents

List of Figures

Chapter 1

Introduction

Chapter 2

Problem Statement

Chapter 3

RAN Design

3.1 Core Layout

There are 24 cores on the available machines. 12 cores are available on each NUMA socket. Although there are 2 CPUs (2 hardware threads) available on each core, hyperthreading is kept off for reducing the non deterministic behavior attributed to contention of hardware units and getting repeatable results.

NUMA- node 1 with starting core 12 is currently used by the RAN.

- **CORE_RX_POLL**: Receives all the incoming packets from UPF. These includes the latency packets or data packets in the downlink direction.
- **CORE_TX_START - CORE_TX_END** These cores are used to send data packets in the uplink direction i.e. from RAN to UPF.
- **CORE_RTT** This core sends the data plane latency packets in the uplink direction. These packets are reflected back by the DNN so that end to end latency can be measured.
- **CORE_CP_TRAFFIC** This core sends the control plane traffic. A separate core was used so that a call back can be registered for storing timestamps. These timestamps are used for calculating control plane latency.
- **CORE_MISC and CORE_STAT** These cores handles miscellaneous functions like ARP handling, timer and logging of stats.

Using this layout, a maximum of 7 cores are available for the data forwarding on the same numa node. To increase this number, the functions running on CORE_MISC and CORE_STAT can be run parallely on the same core. Although 7 cores have been found sufficient till now to saturate 40 Gig line with 64 byte packets.

3.2 Control Plane Forwarding

In a 5G-conforming RAN, the session related messages are sent by SMF to UPF. This RAN-emulator's primary role is that of a load generator. Session related messages - session establishment, modification and release messages are directly sent by RAN to the UPF. There are however some functions defined before the start of the load testing mode in the main function that interacts with AMF and to SMF through AMF. These are not removed as they act as a stub for interaction with AMF. They can be deleted after rigorous testing.

The session establishment, modification and release packets have a standard format of the header as well as the payload. These packets are defined as static arrays in the source file dpdk_ran.cpp - pfcpSessionEstablishmentRequestPacket, pfcpSessionModificationRequestPacket and pfcpSessionReleaseRequestPacket. Before sending these packets, the fields pertaining to a particular session are modified in these packets. These fields include source IP, tunnel end point identifier (teid) and session ids. There are counter variables defined in the dpdk class which maintain the next session related fields. Examples include nextUEIPEstablishment, nextTEIdEstablishment etc. These counters are updated after packet is sent. Only one control plane packet is sent at a time i.e. batch size is one.

3.3 Modes of Operation

The load generator can be used in different modes to characterize the different behaviors of the user plane function. The various modes are

- Setup Sessions and Send Data.
- Send only control plane traffic.

- Send Control Plane and Data Plane traffic simultaneously.
- Helper functions.
- QoS functions.

3.3.1 Setup sessions and send data

Initially sessions are setup by sending session establishment and session modification messages directly from the RAN to the UPF. Once the sessions are setup the data packets are forwarded from CORE_TX_START- CORE_TX_END (inclusive). Control plane latency is calculated during the initial setup for all the session related packets. Throughput is also logged in the log file. The number of UEs/sessions that are used for sending data per core is asked to the user. The sessions are partitioned in disjoint sets among the cores before forwarding.

3.3.2 Send only control plane traffic

This mode is used to send only control plane traffic and no data plane forwarding takes place. This is a synthetic traffic i.e. it does not represent any real world scenario. The main motive behind this mode was to saturate control plane and measure control plane handling capabilities of the different designs of the UPF. The latency and throughput values are logged in the file as earlier. This is an open load test in which session establishment, release and modification packets are sent one by one with a user provided inter packet delay (in us). These packets are sent from CORE_CP_TRAFFIC. The open load means that the RAN does not wait for response packets before sending the next packet. Ideally modification message should be sent only once the session establishment response is received. However, open load is a good approximation of the actual behavior and is easy to implement.

3.3.3 Send Control Plane and Data Plane Traffic Simultaneously

This feature is discussed in Chapter **??**

3.3.4 Helper Functions

There is only one helper function right now. This helper function - Delay Estimator - helps in mapping inter batch delay with the data forwarding throughput of the load generator for a given number of cores and sessions. This mapping is used to set the rate of forwarding of the data packets. Intel 40Gbps NICs that we have do not have rate limiting APIs like 10Gbps NIC.

3.3.5 QoS Functions

These functions were developed to check the correctness of QoS algorithms deployed at the UPFs. These functions were tested on 10Gbps NIC systems and may require modification for other NICs.

For QoS testing, it was required to test whether the packet forwarding rate is actually limited by the algorithm. For this, data is forwarded at a low rate and at a high rate. The low rate is lower than the rate limit imposed on the sessions. The high rate is higher than the limit. So when the rate is higher, the output at UPF is limited to the rate limit.

Lower rate - 700 Mbps, High rate - 1200 Mbps, Rate limit -1000 Mbps. When the incoming rate at the UPF is 1200 Mbps, outgoing rate is 1000 Mbps if the QoS algorithm is correct.

Chapter 4

Latency Callbacks

4.0.1 Latency Calculation Algorithms

Key Concept

When packets are transmitted from RAN, the timestamp when the packet is forwarded is stored for the outgoing packet. When the response for the packet arrives at the receiving side of RAN, the difference in time values of the current time and the stored timestamp is calculated and then reported in the log file.

Storing of Timestamp

- **Data Plane** The packet identification field in the IP header is used to identify the packet. These latency packets are sent from the core CORE_RTT. All the established sessions/UEs are used for forwarding the traffic. When the packet is transmitted, a callback function stores the outgoing packet timestamp in a hashmap with packet id as the key. This field is generally used in the fragmentation and reassembly of packet data. The data plane latency packet throughput is substantially reduced by introducing sleep commands. The option of disabling the calculation of latency is removed as latency packets do not significantly affect any other metrics. Data plane latency is independently logged in a different column in the log file.

- **Control Plane** This requires deep packet inspection of control plane packets. These packets are sent from the core CORE_CP_TRAFFIC. There are

three types of control plane packets - session establishment, session modification and session release packets - all of them are used for measuring control plane latency. These packets have session ids stored at different offsets in the payload.

Earlier, a hash map was used for storing the timestamps. The use of hashmap slows down the call backs and unnecessary limits the rate of data forwarding.

Subsequently, fixed length array was used to store timestamps - **TSArray** . The array size is $65536 * 3$ and stores the value of timestamp register **rdtsc** of the outgoing packets. The first 65536 values are used to store establishment packet related timestamps, next 65536 modification packet related timestamps and then release packet timestamps are stored. The bitwise operations can be easily used as 65536 is a power of 2.

A callback function is registered on the transmit queue corresponding to CORE_CP_TRAFFIC which stores the timestamp in the hashmap.

Calculation of Latency

A single callback function is registered on the CORE_RX_POLL which receives the incoming packet.

- **Data Plane** The same outgoing packet is reflected by the DNN packet and stored timestamp is retrieved from the hashmap and the difference is the end to end latency of each packet.

- **Control Plane** Every control plane packet has a response packet - establishment response (51), modification response (53), release response (55). The control plane latency is the time period from when the request packet is sent and the response packet is received. The response packets have message type and either session IDs in their payload. For modification and deletion responses, the session ids have a difference of 3000. Using these information, the index in **TSArray** can be calculated and the corresponding timestamp can be retrieved.

Why do we need different techniques for control plane and data plane?

- **Why can't we use packet identifier for control plane packets?** Both types of packets are received on the same rx queue/core. So, only the packet identifier in ip header is not enough to differentiate control plane and data plane packets. And you will need deep packet inspection to differentiate among the two packets.
- **Why don't we inspect packet payload for data plane packets?** Data plane packet has no useful payload and when the ip header identification field is unused till now, we can use it.
- **Development Sequence** Data plane latency was implemented earlier when the packet id field was unused. Control plane latency calculation is done later.

Callback functions

Call back functions are registered on receive and transmit queues for latency calculation.

- **dataPlaneStoreTimestampCallback** This callback is registered on CORE_RTT for storing the timestamp of outgoing data plane latency packets.
- **controlPlaneStoreTimestampCallback** This callback is registered on CORE_CP_TRAFI for storing the timestamp of outgoing control plane packets.
- **calculateLatencyCallback** This callback is registered on CORE_RX_POLL for inspecting the incoming packets. These packets are either response packets to control plane packets or the data plane packets reflected back by DNN. The difference in the current time and the timestamp stored during the tx callbacks gives the latency values.

Chapter 5

Clean Code

This portion of the problem statement took the maximum effort and time. The inherited code was never cleaned and had various issues which are discussed below.

5.1 Naming of Variables

5.2 Stale Comments

todos, wrong comments.

5.3 Dead Code

if 0

5.4 Deprecated Options

5.5 Copied Code

testpmd.h

5.6 Global Variables

5.6.1 Issue

Around 150 variables, data structures were declared globally in ranMain.cpp. The use of global variables made the code highly coupled and made it difficult to change one procedure without affecting the other. Infact some of them were declared but never used. This made it also difficult to discern the scope of variable usage inside a procedure - whether is s global variable, local variable or a class member.

5.6.2 Resolution

5.7 Directory Structure

5.8 Dead Code

5.9 Poor Refactoring

5.10 Unnecessary Offloads

5.11 Naming of Variables

5.12 Incorrect Stuff

argc == 0, Y(call X one times) , X

5.13 Technical Improvements

High throughput of data plane latency packets.

5.14 File Layout

Chapter 6

Data Plane + Control Plane Traffic

$n1$ sessions are established before the data forwarding takes place. These sessions remain established throughout the run.

$n2$ sessions are established, modified and released while the data forwarding is also taking place. The data packets are sent from all the currently established sessions. The minimum value of currently established sessions is $n1$. The maximum value is $n1 + n2$.

$t1$ is the total duration of the experiment. $t2$ is the duration of the establishment, modification and release cycle of each of the $n2$ sessions.

The duration $t1$ for which all the static sessions $n1$ and the dynamic sessions $n2$ are used is also asked to the user.

Data forwarding starts from the $n1$ sessions at the start. After sleeping for a time (currently 5 seconds), $n2$ threads are started. The role of each thread is to sleep for a random amount of time $t3$ ($< t2$), establish and modify the session, and then sleep for some time $t4$ and release the session. The invariant is $t3 + t4 = t2$. The session is available for data forwarding in the period $t4$. The threads with new session Ids are started once all the previous threads have joined i.e. have finished their task. Each of the data packet forwarding cores use all the existing established sessions/UEs to forward the data. Note that this is different from the case when sessions were partitioned among cores.

Chapter 7

Results