Chapter 3

Overview of DPDK vRouter

2020-06-26

# Motivation

## Comparison with linux stack and NAPI

Let us look at the state of the art in software packet processing about 10 years ago. At that time, the linux kernel networking stack was the only way for packet forwarding in linux based x86 servers. Linux networking stack uses a mechanism called as **New API or NAPI** in short. This mechanism was added to linux 2.4.x kernels to improve the performance under heavy network IO. NAPI was a midway between interrupt and poll mechanism. It mitigated interrupts nicely such that the operating system disabled the interrupts under heavy load and only enabled them when the load is not high. It also deferred packet processing to "bottom half" where the processing can be done safely. Inspite of this, due to the fact that interrupts still exist, the packet processing incurs a lot of overheads. The context switch during raising an interrupt itself takes several thousands of processor cycles. Once the context switch happens, additional latencies are introduced due to the deferral of bottom halves. Still this was a good improvement compared to the previous versions of linux. Once the context is bottom half, the system switches to poll mode and processes a burst of packets. Once the time share is elapsed, it switches back to interrupt mode. This was fine for smaller NICs, say 1G NICs.

## Need for a networking stack

Let us consider best case numbers for conventional linux packet processing. Today, linux can do around 1 million packets per second per core quite easily. For handling a 1G NIC with 64 Byte packets, we need around 2 CPU cores (64 x 8 Bits x 2 cores x 1 MPPS = 1 Gbps). This was sufficient about 10 years ago. But later 10G NIC started coming into the market which required 2 X 10 = 20 cores to handle it. Now we have 25G and 100G NICs. Clearly linux does not scale. Something new needed to be developed to handle these high capacity NICs with a reasonable amount of CPU cores. These were the origins for DPDK.

# Overview of DPDK

Data Plane Development Kit (DPDK) is a framework for fast packet processing in dataplane applications. It consists of a set of userspace libraries and network interface controller drivers which the applications can link. It is currently managed as an open-source project under the Linux Foundation.

## Origin of DPDK

DPDK was initially created by Intel in 2010 and made available under a permissive open source license. The open source community was established at DPDK.org in 2013 by 6WIND and has facilitated the continued expansion of the project. Since then, the community has been continuously growing in terms of the number of contributors, patches, and contributing organizations. DPDK now supports all major CPU architectures and NICs from multiple vendors, which makes it ideally suited to applications that need to be portable across multiple platforms.

In the very first version itself, DPDK L2 forward application was able to give 5 million packets per second (MPPS) per core. It has continuously evolved in the last decade in terms of performance, features, support for different hardware architectures and network interfaces. In the latest LTS of DPDK, the R1911 version, it can give as much as 50 MPPS per core! To put this in perspective, DPDK can handle a 40G NIC using just 1 to 2 CPU cores in the best case. It is an order higher than linux stack. This has become very attractive for Telco NFV use-cases like EPC, VMX, SRX etc. Telcos can now virtualize their hardware based networking elements and at the same time get very good performance.

## Environment Abstraction Layer (EAL)

DPDK creates a EAL which is customized for a particular environment in which it runs in. For example, we can create a DPDK application through compile flags and makefiles which is tuned for x86 64-bit architecture, linux OS. It abstracts hardware dependent portions and provides a generic API for application developers to use. It is a user-space library. Application developers can write their own applications and link it with the DPDK libraries and it runs as a user-space process.

## Why use DPDK?

DPDK has lot of constructs which can be used by application developers readily. It has efficient implementations of most common data structures used in networking like MTRIE, hashtable, lockless-queue etc. It also provides implementation of networking components like Longest prefix match (LPM), fragmentation support, segmentation offloads, user space vhost etc. Developers need not re-implement all these and can make use of these readily. DPDK Provides networking drives for most of the popular NIC drives like intel, broadcom, mellanox etc. Application development is also very easy. It has very good documentation and has example applications which application developers can readily make use of it to develop newer applications. They need not to start from scratch, they can add to an existing example application.

![image](data:image/jpg;base64;base64,)

## Where DPDK cannot be used?

* TCP stack on DPDK is still evolving. It does not have linux socket implementation. TCP applications like web servers, proxy servers are difficult to implement or port to DPDK. Although 6Wind has a TCP stack for DPDK, it is not free.
* DPDK consumes whole CPU cores owing to it’s polling nature. The core is not visible in linux and is bound to a user-io driver which we will see later. Due to this, other applications cannot use this CPU core.
* It is also not easy to run more than one concurrent DPDK application. The support for this is still evolving
* DPDK can only run on network interfaces which has supported DPDK drivers. It is not as exhaustive as what linux supports. But more NICs are supported now.

# How does DPDK achieve good performance?

## Use of polling

DPDK uses the mechanism of polling to perform packet IO. In contrast, linux uses interrupts. Polling has lot of advantages over interrupts

* The whole interrupt processing is avoided. This saves thousands of CPU cycles during interrupt processing and context switching.
* CPU cores continuously poll for packets in the rings of NIC and dequeue them as soon as they arrive. This reduces the latency of packet processing.
* On the virtual machine side, polling avoids expensive VM Exits when the VM is also a running a DPDK application.

### Note about the VMExits

Suppose a guest VM wants to send packet to host, it has to enqueue the packet to it’s tx ring and raise an interrupt to the host and the later needs to dequeue it. This raising of the interrupt triggers a hypercall. Hypercall is like a system call to the hypervisor. Hypervisor then knows that the packet needs to be dequeued. This operation is called VM Exit and is very expensive. It is in the order of thousands of cycles and this whole overhead is avoided using polling. In polling, the cores continuously poll for packets in the rings and so there is no overhead of VMExits.

## Efficient use of Intel x86 architecture

The diagram shows the typical cache and memory layout of a x86 server. The following are the ballpark latency numbers when there is a hit at different levels of caches or memory

* L1 cache hit: 7ns
* L2 cache hit: 11ns
* L3 cache hit: 30ns
* Main memory hit: 60ns
* Different socket: 100ns

![image](data:image/png;base64;base64,)

These are very significant numbers. Let us take an example of 10G NIC. For 64 Byte packet, the Inter arrival rate is around 67ns. That means, every 67ns, there is a new packet arriving at the NIC. To get line rate, the software has to take the packet, process it and send it out in that time. That is the budget. If it does not meet the budget, there will be queueing and subsequently tail drops. If you see the above latency numbers, just a single cache miss is contributing to 60ns when it has to fetch it from main memory. If it has to fetch it from a different sockets, the latency is 100ns which overshoots the budget.

So how does DPDK work? How does it get line rate? Earlier in the chapter, we saw the current DPDK versions are able to 50Mpps.

DPDK solves this by using the below techniques

### Prefetching and Burst processing

Let us say one core is trying to get a packet. It will take 60ns, but in the time it is fetching the packet, DPDK tries to fetch additional 'n' packets. It may not use it at that instant, but it wants to fetch it. So in the time, one packet is fetched, additional set of 'n' packets are fetched. So in the above example, in 60ns, it fetches, say 32 packets. This is called prefetching and burst processing and the packets it receives or sends at one instant is called burst size. By using this approach, the latency is amortized over the number of packets and DPDK is able to get good performance.

## NUMA awareness

DPDK is very NUMA aware. All allocations like mempool, mem ring, mbuf can be done on a per NUMA basis. Developers can specify which NUMA to allocate so as to get optimum performance. That is very important. DPDK can do this since it is highly customized and uses Intel architecture to a very good extent.

## Optimized memory manager

DPDK has a highly optimized memory manager. DPDK works on a group of fixed size objects called a mempoool. Every one of them are pre-allocated. DPDK does not enourage dynamic allocations because it consumes a lot of CPU cycles and is a speed killer. DPDK pre-allocates a set of mbufs and keeps it in a pool called mempool. Mempool has further optimizations. It is very cache friendly. Everything is alligned to the cache and has a some mbufs allocated for each DPDK thread or lcore. This mempool has a ring where all the objects are stored. This ring itself is a highly optimized lockless ring. This ring can be used to communicate with two or more lcores in a multi-producer/multi-consumer kind of scenario without locks. Locking of data structures is a speed killer. By avoiding locks, DPDK gets large performance gains. It makes use of mempools when we want to allocate a mbuf wher packets are stored. Instead of allocating a single mbuf, we do a bulk allocation or bulk free. By doing this, it already having mbufs in the cache and becomes very cache friendly.

## Hugepages

Conventionally, linux works with 4086 byte pages. x86 has a hardware memory cache called Translation lookaside buffer or TLB. This is a fast cahce for page address and virtual addresses generated by the CPU. Once the CPU fetches a page, the address is cached by this TLB. And if the reference is made to the same page again, the memory manager looks up the TLB for the page. If found, multiple page table lookups are avoided. But if the TLB misses, the time it takes to fetch a page is very large. If we look at the linux paging model, it has two or three levels of page tables like midpage table, high page table etc. After these multi-level lookups, it is then memory manager can find the base address of a page to translate the virtual address to a physical address and eventually fetch the page from main memory. This whole process is very expensive. So what DPDK does is, instead of 4K page size, it uses 2M or 1G page size. Because of the use bigger page size, the number of pages in the main memory drastically reduce. So this TLB will suffice to cache or hold the virtual addresses and then most of the lookups of pagetables are avoided, thereby improving performance.

![image](data:image/png;base64;base64,)

![image](data:image/png;base64;base64,)

## UIO drivers

DPDK uses UIO (user IO driver) to communicate with the NICs in user space. UIO driver is a skeletal kernel modules. It takes the BAR(base address) regions of PCI address space the NIC, it’s config regions and other relevant regions and creates the memory map of all these which can be accessed by DPDK applications in user space. By this, the applications can directly access the NIC’s rx/tx rings and config registers directly. Once the setup is done, the linux kernel is bypassed. The DPDK PMD drivers can enqueue a descriptor and point the descriptor to an address space in the user space instead of kernel space and the NIC can DMA from that. Without the support of UIO, this needed to be in the kernel and needed a system call or other kernel mechanisms to do it.

![image](data:image/jpg;base64;base64,)

![image](data:image/jpg;base64;base64,)

## Use of Native instructions

DPDK applications exploit the use of native instructions on the hardware to provide further acceleration. Every DPDK application can have the option of compling on native or generic instruction set. For Example, the recent processors after ivy bridge have vector instructions and support SSE2 instructions set. Operations like CRC32 are at an instruction level and can be accelerated. If the DPDK application is compiled for native architectures, it uses those instructions instead of using generic instructions and provides performance gains.

# Virtio

Virtio is a mechanism of sending packets between virtual machines and host. It is a para-virtualization technology where VM and host need to co-operate. They have to share a common data structure or a memory mapped region called virtio ring so that exchange of packets are possible. It has a front-end driver which runs in the guest and a backend driver which runs in the host. It was first conceived by Rusty Russel in 2009.

## Types of virtio

* Approach 1 - Virtio Ring is shared between Qemu and guest. If guest wants to send packet out, it has to interrupt Qemu. Qemu will then dequeue the packet and makes system call to linux stack and the stack puts it on the tap interface on the host. It involves VMExits and system calls. Owing to this, it is very expensive. We can get less than 100 KPPS using this approach and so it needed improvements. This lead to approach 2.
* Approach 2 - Here, the virtio Ring is moved inside linux kernel. There are kernel processes called vhost processes. Qemu, through ioctl programs the vhost processes about the address of the memory map when the guest is spawned. When a packet is received from the NIC, the vhost processes will directly take the packet. Without going to Qemu, it will enqueue the packet to the virtio ring which is shared with the guest. If a guest wants to send a packet out, it will enqueue to the ring and vhost processes which will be bound to the ring will dequeue the packet and sends it out on a tap interface. This avoids system calls. Ofcourse there will be VMExits still since it has to interrupt to wake up the vhost processes.
* Approach 3 - With qemu 2.1, there was support for user space vhost. It moves the virtio ring from kernel all the way to userspace. The ring is shared between the guest and DPDK vrouter. Qemu sets up this ring as a control plane using unix sockets. Suppose guest is also DPDK, it does polling, so there are no VMExits. There is nothing in kernel here, so there are no system calls. Since both system calls and VM Exits are avoided, the performance boosts significatnly. It will be an order higher. Without user space vhost enhancement in qemu, vRouter was not even possible as a solution also. User space vhost runs in vrouter.

## Server mode Qemu vs Client mode Qemu

There are two options for Qemu -

* Option 1 - Qemu can be client and vRouter needs to be server
* Option 2 - Qemu can be server and vRouter needs to be client

Prior to Qemu 2.8, the only mode supported was option 1, DPDK vRouter had to be server and Qemu had to be client. There were issues like reconnect. So the Qemu community made Qemu as server and DPDK vRouter as client in Qemu 2.8. So it is the responsibility of DPDK vRouter to connect to Qemu and thus the reconnect issues were avoided.

# Contrail vRouter

## History

Initially when vRouter was developed, it was based on linux kernel. But linux kernel is slow. In the best case, it can give a few hundred thousand packets per second. This is not enough for supporting faster NICs like 10G, 25G and so on. At that time, the telco industry was moving towards adapting DPDK in their VNFs. But if the back-end is not DPDK powered, the performance will still be less. So a new vRouter flavor was developed which was accelerated using DPDK. With this, vRouter can easily support 10G and 25G NICs. The performance gain was an order in magnitude higher. Whereas the kernel vRouter could only process a few hundered thousand packets per second, the DPDK vRouter can process millions of packets per second on the same hardware. In summary, the main reason why DPDK vRouter originated is due to the need for faster processing speeds due to the availability of faster NICs.

# Needs refinement

## Architecture of DPDK vRouter

![image](data:image/png;base64;base64,)

Nothing in kernel, entire DPDK vrouter runs in user-space as an application. It talks to the guest which is also an application. Because it runs in the context of qemu. Because of vhost-user, there is a communication channel between the vrouter and the guest. Earlier agent used to talk to vrouter using system calls. Now its unix sockets. Packet enters the NIC. NIC is polled continuously by DPDK, the packet directly enters DPDK. And DPDK does all the encap/decap, all vrouter functionality. To send it to the guest, it uses vhost user and puts in the ring to guest.

DPDK is an multi threaded application. Comes as a set of pthreads which has affinity to the cores. We call it lcores. We also have a bunch of helper pthreads/lcores and bunch of forwarding lcores. At provisioning time, we decide how many forwarding threads are needed for vrouter and spawn them on DPDK initialization. These threads polls NIC, vritio ring etc. Other threads exist - netlink thread, vhost thread etc, pkt0 thread, KNI/tap thread. We have multi producer single consumer implementation. We have one lock.

How to send packets to linux kernel? PAcket comes from NIC and enters vrouter. We use KNI or tap to reinject packets from user space to kernel space. In KNI, there is a kernel module calleed KNI. There is a ring shared between DPDK and KNI. DPDK puts in the ring and KNI module which is polling the ring dequeues it and sends it to the tap interface.