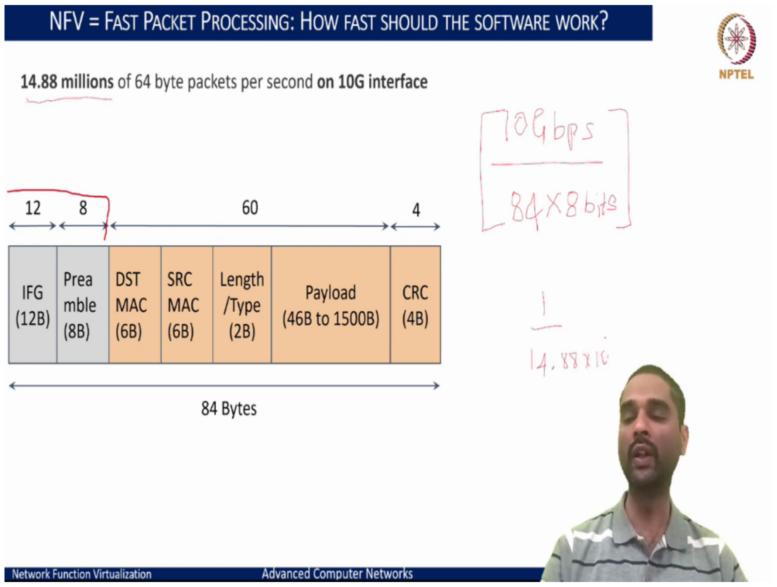
Advanced Computer Networks
Professor Dr. Sameer Kulkarni
Department of Computer Science Engineering
Indian Institute of Technology, Gandhinagar
Lecture 42
High-Performance Network Packet Processing

(Refer Slide Time: 0:17)



NFV = Fast Packet Processing: How fast should the software work?

14.88 millions of 64 byte packets per second **on 10G interface**

IFG (12B)	Preamble (8B)	DST MAC (6B)	SRC MAC (6B)	Length /Type (2B)	Payload (46B to 1500B)	CRC (4B)

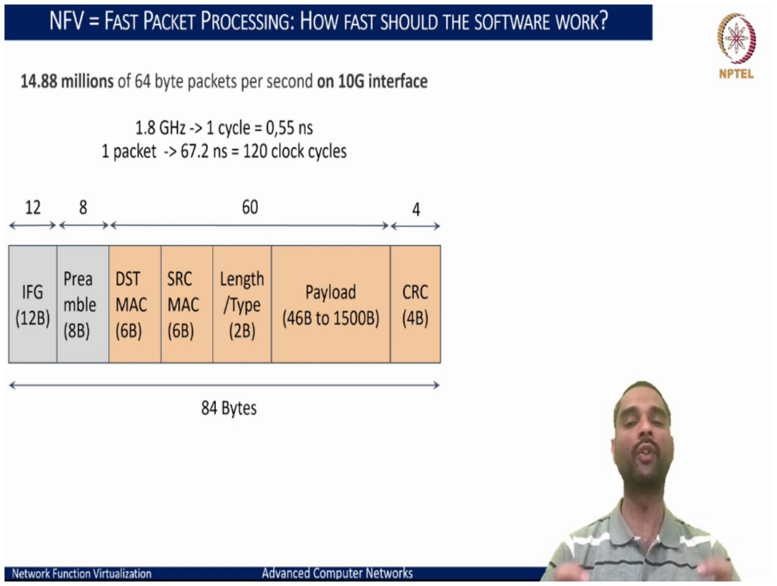
12 | 8 | 60 | 4

84 Bytes

$\left[\frac{10\,Gbps}{84 \times 8\,bits}\right]$

$\frac{1}{14.88 \times 10}$

Network Function Virtualization — Advanced Computer Networks



NFV = Fast Packet Processing: How fast should the software work?

14.88 millions of 64 byte packets per second **on 10G interface**

1.8 GHz -> 1 cycle = 0,55 ns
1 packet -> 67.2 ns = 120 clock cycles

IFG (12B)	Preamble (8B)	DST MAC (6B)	SRC MAC (6B)	Length /Type (2B)	Payload (46B to 1500B)	CRC (4B)

12 | 8 | 60 | 4

84 Bytes

Network Function Virtualization — Advanced Computer Networks

Now, let us try to look and understand what we meant by NFV's packet processing capabilities, what are the needs, and where do we stand when we use just the available commodity hardware and software. For example, if we use a Linux machine, what does it mean, and how do we address these aspects of performance?

So, the figure here is trying to show basically what we see as a typical packet that is exchanged when we send out over an Ethernet, and here what we are trying to show is just a bare minimal packet that we would exchange from one node to the other and once that such a bare minimum packet is typically considered to be offered 64-byte length which is going to be padded with the 20 bytes of the interfering gap and preamble of 12 and 8 bytes, making it basically a 64 byte packet with the 20 bytes of the additional data that is being added before sending it out on the link.

So, a total of 84 bytes is what gets exchanged when we have this very minimal 64-byte packet that we can exchange at layer 2. So, if we take this as a layer 2 packet, that is a 64-byte information at layer 2 and exchanged over a link with a total of 84 bytes.

Now consider if we have just a 10-gigabit interface that is what we have is basically a 10 Gbps link, and on this 10 Gbps link, we are able to send this 84-byte packet that is 84 bytes or 84*8 bits. And if we solve this roughly, we will end up seeing that a 10 gigabit per second interface or 10 gig NIC would allow us for transacting roughly around 15 million or 14.88 million packets per second.

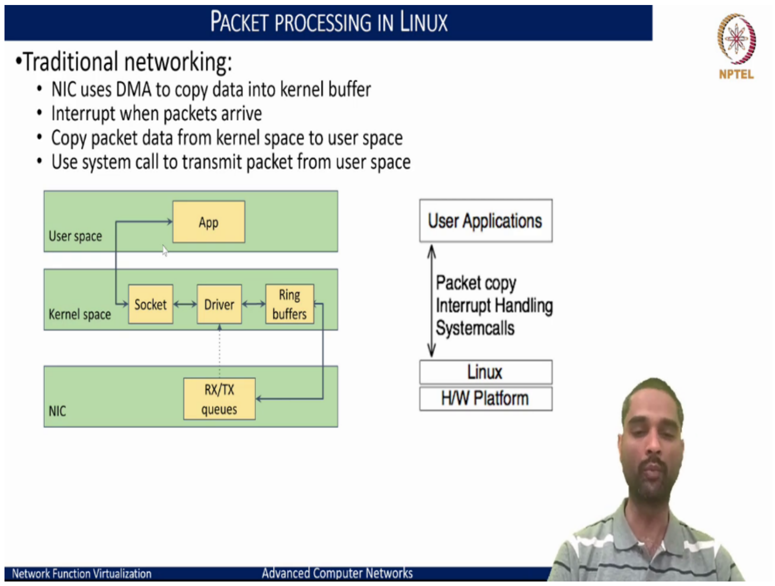
And this 14.88 million packets per second, if we translate that to how much of a time that you would end up with per packet, then you would do $1/(14.88 * 10^6)$ as the time that you would need to process such a packet which would correspond to roughly about 67.2 nanoseconds per packet and that is the amount of time that we should be spending to process 14.88 million packets per second.

And if we look at this and try to see if we have a 1.8 Ghz processor. One cycle roughly corresponds to 0.55 nanoseconds or if we have a 2 GHz, then one cycle corresponds to 0.5 nanoseconds and on such a machine, if we now compute the number of clocks that we would need or the amount of CPU cycle clock cycles that we can really spend to meet the 14.88 million packet processing per second, then it roughly estimates to around 120 clocks on 1.8 gigahertz processor or if we had a 2 gigahertz processor then it would roughly estimate to around 134 to 140 clock cycles.

And this is where we see that the number of clock cycles that we really want to expend are premium, and have to see how we can meet or look at in optimizing these number of clock cycles that we may have to expand once we receive such a packet.

And we need to devise mechanisms to ensure that if we are receiving a packet every 120 clock cycle, we have to also deliver the packet out within those cycles or have a means to process the packets within the stipulated time.

(Refer Slide Time: 4:14)



Packet Processing in Linux

•Traditional networking:

- NIC uses DMA to copy data into kernel buffer
- Interrupt when packets arrive
- Copy packet data from kernel space to user space
- Use system call to transmit packet from user space

So, to understand this better, let us try to see what really happens in traditional networking,, like in the Linux World, once the packets arrive. The typical journey is as the packets arrive onto the NIC, they are queued onto this RX queue or TX queue within the NIC, and then an interrupt would be fired to the driver while the packets are copied to what are called the kernel space ring buffers.

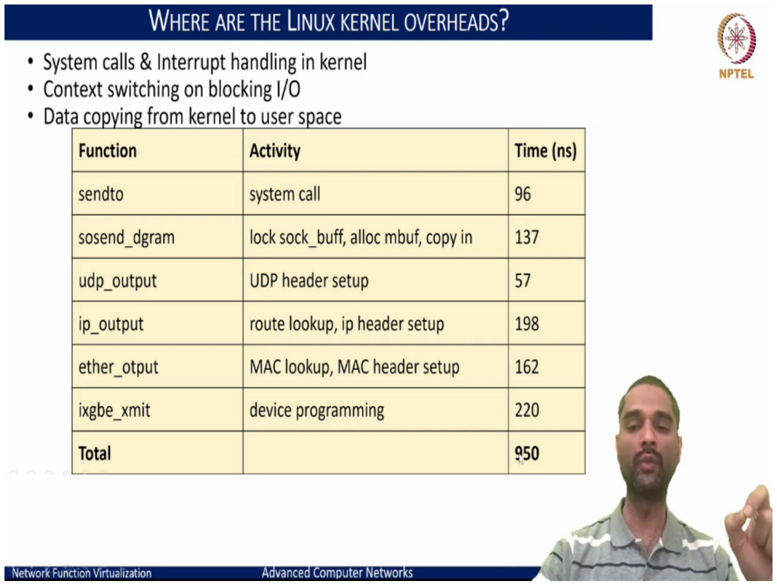
And once the packets are copied onto this ring buffer, an interrupt awakes the driver to start processing on these packets. It would take the packets from this ring buffer and put it onto the socket buffers, which the application would have shared for processing, and once the packets are updated onto the socket buffer, a system call would then enable and awaken the application to start processing the packets.

Note, this is the time that we are speaking is in terms of as the packet arrives to the time that the application processes it, and the next packet that an application needs to process all has to be done within 120 cycles. So, we need to understand this in principle, what are the stages that we are looking at. One is basically the system calls that are involved in making the application up and start to process the packet.

Two, the interrupt that really triggered from the hardware to the kernel space for the driver to wake up and start processing the packet. Three, as the driver has to do the copy from the ring buffers onto the socket for ensuring that the data is delivered to the right applications. So, these we can see as the major components that would lead towards what we can see as key processing overheads that we will be adding.

So, we need to understand the overheads that would be incurred and what are the number of cycles that we would have when we look at this budget that we just spoke about.

(Refer Slide Time: 6:06)

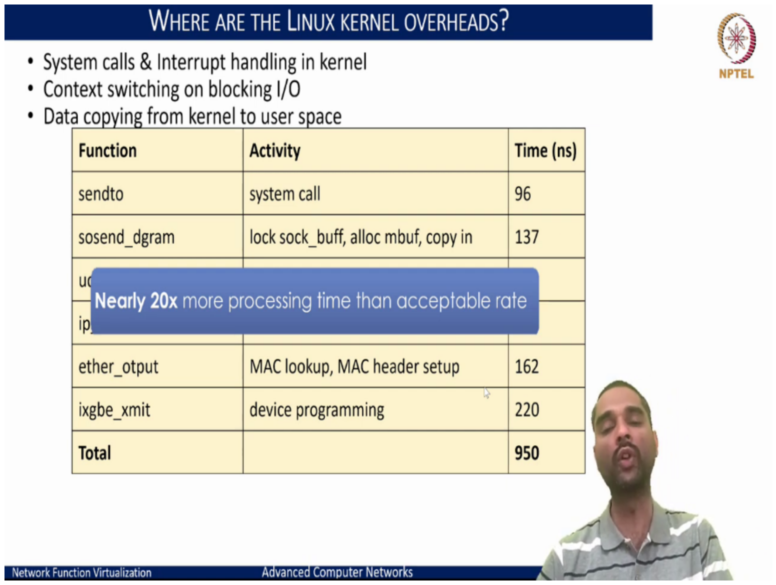


Where are the Linux kernel overheads?

- System calls & Interrupt handling in kernel
- Context switching on blocking I/O
- Data copying from kernel to user space

Function	Activity	Time (ns)
sendto	system call	96
sosend_dgram	lock sock_buff, alloc mbuf, copy in	137
udp_output	UDP header setup	57
ip_output	route lookup, ip header setup	198
ether_otput	MAC lookup, MAC header setup	162
ixgbe_xmit	device programming	220
Total		**950**

Network Function Virtualization   Advanced Computer Networks



Where are the Linux kernel overheads?

- System calls & Interrupt handling in kernel
- Context switching on blocking I/O
- Data copying from kernel to user space

Function	Activity	Time (ns)
sendto	system call	96
sosend_dgram	lock sock_buff, alloc mbuf, copy in	137
ud		
ip		
ether_otput	MAC lookup, MAC header setup	162
ixgbe_xmit	device programming	220
Total		**950**

Nearly 20x more processing time than acceptable rate

Network Function Virtualization Advanced Computer Networks

So, we can do that using the strace functionalities or trying to do the kernel tracing or profiling of the different functions that we have within a system, Linux kernel and the network stack. And if we look at a typical function when a socket wants to send out a packet, it would use sendto system call.

And once this sendto system call is done, how much time does it take for the packet to arrive eventually onto the NIC interface that is out of the ring buffer for that the driver can initiate if it is ixgbe driver which will call ixgbe_xmit to send the packet out on the link.

So, if we analyse these calls. This is a trace on one of the Linux devices, which was done in the lab of early 2014 and we can do that on any of our machines and see how the time would look like, and we can see that the sendto call itself takes around 96 nanoseconds which is much more than what we said about 67.2 nanoseconds of the budget that we had for per-packet processing on a 10 Gbps link.

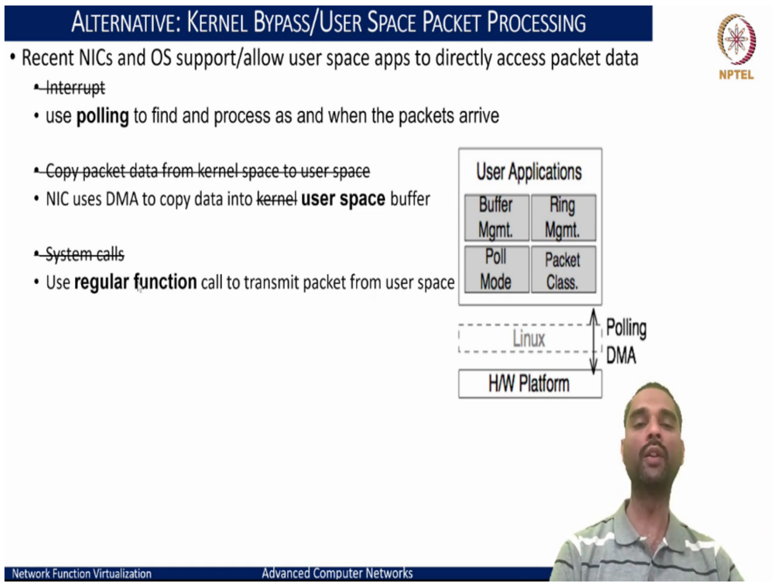
And as it goes further into the network stack of the TCP that starts from the sosend_dgram and then udp_out where udp header processing and allocation of the mbuf in the sock_buff and then eventually processing where the route lookup happens at the layer 3 where to send the data which interface to send the data out and then the layer 2 processing with a MAC lookup and MAC header setup. All of these, we can see that they incur a lot more nanoseconds time than what we really had as a budget, and if we sum it up for the time the packet is being asked to deliver out from the application to the time that packet is really sent out from that device if we sum it up this roughly corresponds to around 950 nanoseconds.

And we can see that this is almost 20x more than our real budget of 67 nanoseconds. Hence we can see clearly that we need newer mechanisms unlike what we said the virtual instances would really work is not true as it is. So, we definitely need certain mechanisms where we can optimize on this what would it really mean is either rewrite the entire kernel stack or rewrite the network stack part so that we can optimize on it or rewrite the system calls, which would take much lesser time and this is a daunting task.

So, we need varied mechanisms wherein even though we may incur the latency in these orders, we may want to hide latency wherever we can, and that is where typically we do the buffering where we process in batch, not just one packet. If we send this functionality of sending just one packet takes around 96. If we send the 100, it might take a few orders more but not 10x the time, and that's where we can really save on latency. That is one of the approaches.

The other, if we are able to see the bulk of the processing overheads are happening in these three network stacks. If you are able to avoid the kernel, we can eliminate lots of these other overheads. So, this is where many of the researchers thought together put together as a consortium of what all options can be brought to minimize this overhead so that we are able to keep up then with the 10 Gbps, but now we need to keep up with 40 or 100 Gbps links at the least.

(Refer Slide Time: 9:40)



Alternative: Kernel Bypass/User Space Packet Processing

- Recent NICs and OS support/allow user space apps to directly access packet data
  - ~~Interrupt~~
  - use **polling** to find and process as and when the packets arrive
  - ~~Copy packet data from kernel space to user space~~
  - NIC uses DMA to copy data into ~~kernel~~ **user space** buffer
  - ~~System calls~~
  - Use **regular function** call to transmit packet from user space

And towards these challenges, many alternatives were thought, and the things that really emerged were what we thought as a kernel bypass because trying to do anything in a kernel was taking multiple copies and processing overheads. So, better avoid this processing in the kernel, and when we look at these middleboxes, they are really the applications now running in the user space. So, why not do the layer 2 packet processing, layer 3 or layer 4, whatever is required right within the user space?

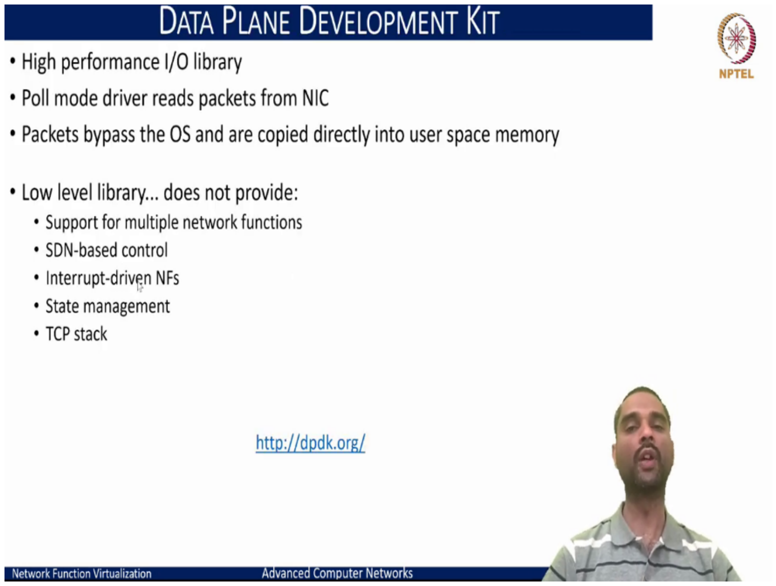
Well this may add more cycles that we would need to process the packets on the application, but it would certainly be tailored to what are the requirements of an NF. Network function has rather than having to do all of the processing for every NF, irrespective of what its requirements are at layer 2, layer 3, or layer 4, to go through this entire stack, and that is where many of the kernel bypass techniques emerged and what they really mean when we say this we bypass the kernel is first, we avoid the interrupt based processing. That means the network interface does not have to interrupt the kernel driver, wake it up, and start processing. Instead, we can basically poll the hardware to see when the packets are arrived or immediately send the packet whenever there is a packet that we want to send it right onto the hardware.

And that is basically through the elimination of the interrupts, which were seen to have major overheads. Second, we avoid the copy of the packet data from kernel space to user space onto the ring buffers that you would want then for that hardware to transmit the packet. Instead rely on the NIC, that is, the network interface card's DMA to copy the data directly into the user space buffer and vice versa when we are trying to transmit a packet from user space onto the NIC memory, where the packets would eventually be transmitted out using the ixgbe_xmit. So, if we are able to avoid the copy overheads, we would be spending much less of a cycles.

And third, avoid doing any of the system calls which would incur a change of processing, and scheduling in terms of a user space, context to the kernel space, and vice versa. So, instead we could use regular functions or library, APIs which would not have to do much with the system but try to manage many things within a user's space.

This although seems simple has a very challenging aspect to build when you want to have realize these software of network functions.

(Refer Slide Time: 12:30)



DATA PLANE DEVELOPMENT KIT

- High performance I/O library
- Poll mode driver reads packets from NIC
- Packets bypass the OS and are copied directly into user space memory
- Low level library... does not provide:
  - Support for multiple network functions
  - SDN-based control
  - Interrupt-driven NFs
  - State management
  - TCP stack

http://dpdk.org/

Network Function Virtualization Advanced Computer Networks

But nonetheless, we can think that if we are able to eliminate these aspects, we would be able to save a lot on the processing latency, and this is where the things emerged and led to what we see as a data plane development kit. And this data plane development kit initially originated from Intel which was meant to provide high-performance I/O and this high-performance I/O as a library so that it can be implemented in user space.

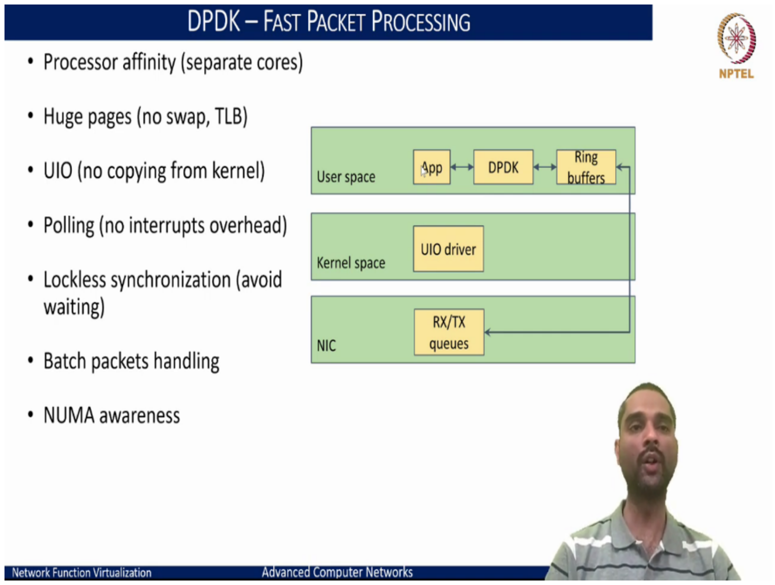
And this also incorporated that you need special support in the hardware so that you are able to poll the device rather than the hardware going to interrupt back to the kernel. So, the extensions were both in the software as well as in the hardware support so that you are able to facilitate polling on the NICs. And then you would also like as you poll, you need to transfer the packets directly onto the user space memory. That also required that you bypass the OS and copy the packets into the user space memory with specific shared memory infrastructure.

So, this was the base of how we could think of having faster packet processing that could be done on the traditional Linux machines. But when we tried to suit this onto the NFV infrastructure, NFV has a lot more requirements than this because we need to support running multiple of the network functions, we need to support having multiple of the network functions being controlled through an external management and orchestration framework wherein the rules of forwarding could be set by an SDN controller and each of these network functions could themselves have to communicate, intercommunicate with them, forward the packets, manage their own state and in fact when we bypass the kernel we no more have the luxury of a TCP stack that used to do the reliable packet handling between the two ends.

Now all of that has to be pushed onto the network functions themselves, that means we need a user-space TCP stack if we have to support TCP connections like HTTP or TLS termination; network functions now need to additionally support this TCP stack if they have to utilize this data plane development kit.

But what DPDK really provides is ensuring that very least latency for a packet that would arrive on the NIC be made available for processing at the user space and ensure that this packet whenever a process wants to send it out, can be immediately sent out on the NIC without having to incur any of the overheads. And this is where we need now the use of DPDK and the combination of the things to further support these aspects that we just discussed.

(Refer Slide Time: 15:14)



DPDK – Fast Packet Processing

- Processor affinity (separate cores)
- Huge pages (no swap, TLB)
- UIO (no copying from kernel)
- Polling (no interrupts overhead)
- Lockless synchronization (avoid waiting)
- Batch packets handling
- NUMA awareness

And DPDK through its, as soon as it came out, became heavily popular and was used in many of the network function frameworks. And this requires us to at least understand in a bit what really changes with the DPDK or why it is really a fast packet processing model, and what it really tries to bring in is that now NIC would handle the RX/TX queues as it is in the earlier hardware.

These are the hardware queues where the packets would be queued up on reception or for transmission, and it would be accordingly processed by the hardware. But whenever the packets arrive here, note now that we have now gotten rid of any of the packets that we were trying to copy in the RX/TX ring buffer or copying back to the socket buffer for the applications. Instead, now the ring buffers where these packets were going to be put are shifted from the kernel space to the user space.

So, what that means is there is just one copy of the packets going directly onto the ring buffers, and once the packets are copied to these ring buffers because they are in user space, the DPDK framework can allow the application, which is basically a library that is hooked with the application to process that packet directly in user space without the need for a system call and without the need for any interrupt from the NIC to the hardware to process those packets.

And thus, it is completely bypassing the kernel space providing the packets onto the user space. And likewise, when you want to transmit a packet, you are making a DPDK API call to push the packet out, and it would copy the packet onto the ring buffer and then which could even be the mem-mapped buffer so that you will eliminate the overheads of copying and only trigger for having the NIC to send it out.

But this requires some updates into the kernel space, and that is in the essence of the user space I/O driver, which basically changes the mode of operation of the NIC, which earlier used to deliver an interrupt and process the package to the kernel space to now change to no interrupts but copy the packets onto the ring buffers where the user space buffers are being shared. And this model became very popular, and we use it excessively in network a function, any of the network functions frameworks.

And this also comes with several of the additional things. The foremost being the processor affinity, wherein any applications that you would want to run because you have the memory that is now running on which will have different amounts of overheads when you are trying to process memory of different regions, especially because now we have multi-socket or multi-core systems where memory access to one region may be cheaper than memory access to other regions and this is where the NUMA awareness is required and trying to bring the affinity of a application to run on specific processors.

So, if I am having a network function that is running in socket 0, that needs to be ensuring that we are having these NICs also tied to the same level so that the memory access between the two is cheaper.

And to avoid especially when we access the packets, they may be huge number of packets that we will be processing in a short span, and if there is a memory access which is a virtual space memory that we are trying to see and there are access overheads especially if you have to swap in or swap out the pages which are typically the 4K pages that we see, it would incur lot more overheads in kernel trying to be involved in the page swap.

That also could be eliminated through the huge use of what we call as the Huge pages and here what we are really trying to change is basically, instead of operating on a small page of 4K size, we want to operate on a megabit or gigabit sized page so that the pages are always resident within and you want to avoid the overheads of swapping.

Further, these Huge pages also help like you now you have minimized the number of pages your TLB, which is very limited in terms of keeping track of how many pages that you are really active can always be resident. So, your TLB need not have to be flushed in and out anytime this swap happens or anytime the context switch happens and this use of huge pages either as a gigabit size pages or in like 2 MB pages are being used heavily with this DPDK framework for further optimizing on how the packets can be really worked out without paying any overheads.

And polling in fact, ensures that we are dedicating a new processor which is going to look for the arrival of any packets on these ring buffers or push the packets immediately onto these ring buffers. And we also said like when we have applications, we may have multi-threaded applications or even we can have multiples of applications that are going to be operating with DPDK and somewhere they would not have to synchronize in terms of when to process a packet when to push the packet or forward a packet to another NF, and there may be a good amount of interaction between multiple of these threads or multiple of the network functions in user space.

So, we would also need key primitives to ensure that we do not have the system calls typically, we would have a mutex where you lock or you have a semaphore that you would use or a barriers which are typically in the system calls that you would do. But now we cannot have that luxury if we want to do the fast packet processing, and hence DPDK also provides what is called as a lockless synchronization fashion in terms of how we can use fast weight mechanisms for multiple of the threads to operate in the DPDK space.

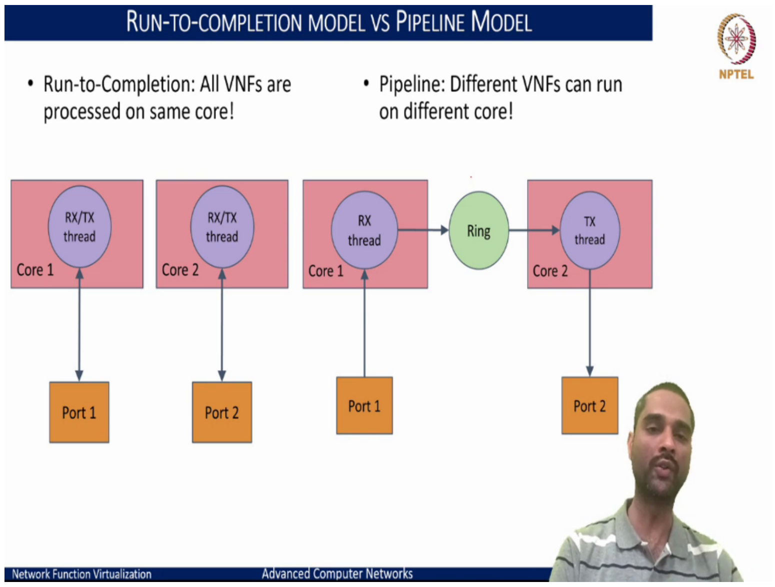
But we also need newer mechanisms, not just as DPDK libraries but maybe some things that we can also augment and build. But the push towards lockless synchronization gained very high attraction, and several of the mechanisms were also pulled in this and like I said before, the best way of latency hiding is always doing the batch mode of packet processing.

Because whatever the time that you need to transmit a packet from one to the other, if I am taking 100x units of time to do that operation and 10x units of time to actually do the copy, then if I do two copies instead of spending 110 cycles, I would just spend 120 cycles because the copy overhead is only 10 cycles while trying to do this operation itself is a hundred cycles of waking up and doing moving the data.

Hence if I increase to a larger batch size, I would end up getting greater benefits, and this is where the batching of packets when we want to really process started has always been used, not that it is a new thing that came with DPDK. This was always been the case with respect to when we build the queues, we want to ensure that you have sufficient packets in the queue, and then you process a batch of packets.

And the same concepts have been borrowed over for DPDK, but we want to ensure that we treat a batch of packets even when we want to process the packets at the application space, and that is how we can see this DPDK as a framework, as a user-space library that enabled for rich and faster packet processing plates.

(Refer Slide Time: 22:30)



RUN-TO-COMPLETION MODEL VS PIPELINE MODEL

- Run-to-Completion: All VNFs are processed on same core!
- Pipeline: Different VNFs can run on different core!

And this DPDK in its essence, comes with two specific kinds of models. One is what we call as the run-to-completion mode. Here, the network functions that we can think of that we want to deploy, are deployed on specific app, and it would have basically as the packets are sent it can process them all at once and send the data out, that means that RX to the TX chain and all the processing happens in one go, on just one dedicated core.

And this way, we can basically ensure that there is no overhead of passing the packets from one core to the other because if you have to access the thing, there are a lot of things that come in between as the caches where the data would already be warmed up and if you move to another socket or another core which has its own separated L2 and L1 cache, then the data that needs to be re-pulled into those L2, L1 cache from L3 that would incur more cycles.

Hence, processing in a single core was seen to be a more effective fashion, and that was called as a run-to-completion model wherein the entire thing, start to end, you do all the processing and send the packet out so that you are optimizing on and evading any of the memory overheads. But this model, although best in terms of performance, creates a problem when you want to have multiple of the network functions that have to interact.

And especially because now you have to glue all the network functions into one processing and make them as a single process which makes it really tedious job to bundle them up in one because we want independent software vendors to develop different network functions.

And if we have different implementations, and different mechanisms being used, this run-to-completion model seems a tedious task to integrate and build. Hence the other alternative is also provided, that is what we call as a pipelined model. Here different virtualized network functions can run on different cores.

What that means, now is as the packets arrive, there will be a RX thread that would poll for the packets that are going to be received, and once they are received, it would copy onto the shared ring buffer and through this ring buffer becomes basically a channel for communicating another thread to really process that packet. In essence, what we can see is we can really change these RX thread ring buffers so that multiple of the network functions would have their own rings and their own RX and TX threads to process the packet and move it out.

So, that way, there is an overhead that we pay in terms of updating the contents onto the ring, but we can see now that you do not have to really copy a packet. If we have a packet and they are all going to be used in a user space, we can have it on mem-map and then just share a pointer to that address. So, here typically, what you are trying to copy is just the metadata of where to access a packet rather than copying the entire packet.

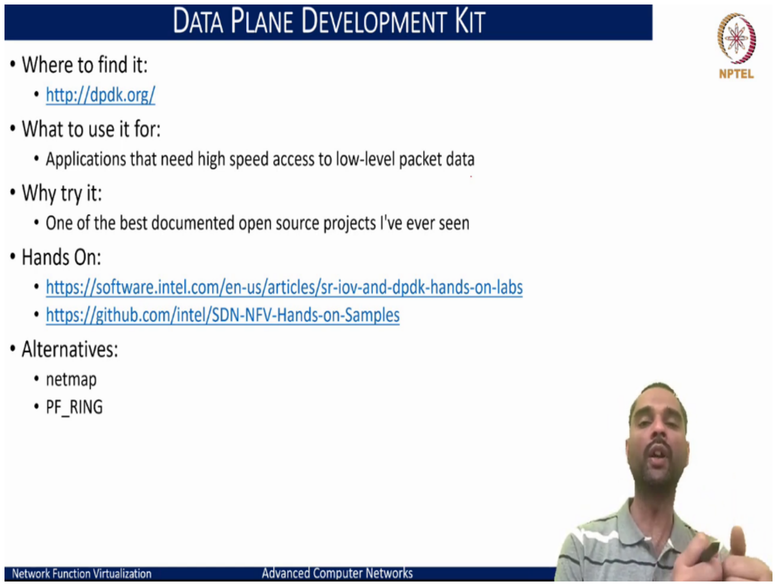
And this is where the pipeline model really helps in augmenting the fast packet processing and, at the same time, ensuring that we can build the NFV applications which are really heterogeneous in nature and can operate together on different cores. Further, we can see that now NFV, if they have a different requirement, a single NF when we deploy as a VM, it may have multi-core requirements, wherein you want to do certain pre-processing, the actual need of whatever the application has to do, state updates and then post-processing, each running on independent cores.

All of this can be readily supported in this pipeline model. Although there is a bit of overhead in terms of the copying and updating of the ring buffers and context switching of different threads are the two overheads that you would see as opposed to what you would see in the run to completion.

But the flexibility that you get with the pipeline model is a lot more and readily usable for deployments of network functions, and once these were seen, there were a lot of research works that came in either of the contexts and one family of works using run to completion for very fast packet processing and the other family of works that use the pipeline model and try to build a framework for developing the NFVs, and both have been equally adopted and used and preferably like what pipeline model provides has greater advantages although comparatively less performant than they run to completion models.

But again, like we spoke of the techniques wherein we were able to mask, we could use similar techniques in the pipeline model to mask the processing overheads or have what we call as latency-hiding techniques that can be developed for better and more efficient packet processing.

(Refer Slide Time: 27:22)



Data Plane Development Kit

- Where to find it:
  - http://dpdk.org/
- What to use it for:
  - Applications that need high speed access to low-level packet data
- Why try it:
  - One of the best documented open source projects I've ever seen
- Hands On:
  - https://software.intel.com/en-us/articles/sr-iov-and-dpdk-hands-on-labs
  - https://github.com/intel/SDN-NFV-Hands-on-Samples
- Alternatives:
  - netmap
  - PF_RING

Network Function Virtualization Advanced Computer Networks

So, to sum up what we have looked at is not in a very depth of the DPDK but the high-level essence of what DPDK constitutes of. But if we have to get into the details of what DPDK, it would be a tutorial on its own of roughly 5-10 hours. But I would encourage that you go look up the DPDK at this dpdk.org which is like I said now, it is taken up by the Linux Foundation, but it initially originated from Intel.

And there have been so many of the documentations, and especially when we see the open source implementations, typically we are wary of how the documentations are. But in my personal view, the DPDK has the best of the documentation in any of the open source.

Although now many of the open sources are coming up with good documentation but I think from day 1, the DPDK when it started, they have the best of the documentation. So, it is very easy to follow it up, what are the key APIs that we would have to use, and how to build on and it is very easy to try out this DPDK as well, and there are so many of the like I said SDN-NFV groups including our group where we have another framework called openNetVM where we can look it up and see how it is been designed.

And this is one of the approaches. There were also alternative approaches that came up. One of them being the netmap and the other being the PF_ring or the Linux and APIs or new APIs to facilitate faster packet processing.