Modifying existing applications for 100 Gigabit Ethernet

Jelte Fennema
jelte.fennema@os3.nl
Monday 11th July, 2016

Abstract

This research investigates the use of DPDK to saturate a 100GbE link. This is first done by looking at Pktgen, an already existing DPDK packet generator. After that iperf3 is modified to use DPDK, as it normally uses the Linux networking stack. The performance of this new iperf3 is then compared to that of the original iperf3. It is shown that the necessary modifications to iperf3 for this were quite straightforward. However, these modifications resulted in less throughput than before. The cause for which is probably some missing features in the DPDK network stack that was used.

1 Introduction

Currently, 100 Gigabit Ethernet (100GbE) Network Interface Cards (NICs) are becoming more common in high performance networks. However, because of these ultra high speeds the CPU overhead of the Linux kernel, required to simply create and handle the packets, becomes quite significant. Even when run on a high performance server, the popular network benchmarking tool iperf3 is not able to utilize the full speed of the network as it is limited by speed of the CPU, since it fully utilizes the core it is running on [1]. This means that specialised and expensive equipment is needed to test the performance of these networks [2].

To combat this limit imposed by the CPU, Intel has developed the Data Plane Development Kit (DPDK) [3], which is a framework for fast packet processing. It is used to access a NIC directly from user space, thereby bypassing the kernel and its overhead, as shown in Figures 1 and 2. DPDK consists of a set of high performance low level libraries, such as memory pool allocator and ring buffer libraries. Apart from this, it also contains special drivers described as poll mode drivers. These drivers do not use interrupts to signal that data is available, but instead the application keeps asking (polling) if data is available. This eliminates the overhead caused by interrupts, but has as a side-effect that a DPDK application is always fully utilizing its CPU core, since it needs to keep asking for data.

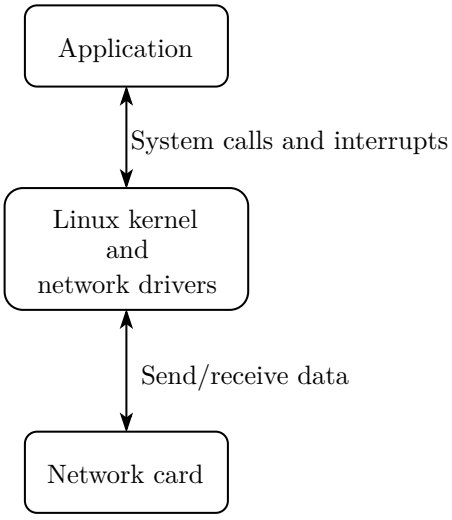

Figure 1: Regular networking on Linux

There are two different packet generators, similar to iperf3, based on DPDK, Pktgen [4] and MoonGen [5]. Using DPDK for these packet generators has proven to be quite fruitful as both show serious performance

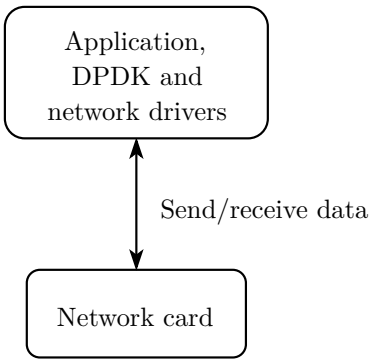

Figure 2: Networking with DPDK

improvements over their former counterparts [5, 6]. However, even for these packet generators no data is available for their performance on 100GbE links.

There are a couple of important differences between these packet generators and iperf3. First of all, both of these applications have been designed from the ground up for use with DPDK. Secondly, they do not open an actual TCP connection like iperf3 does, but instead they simply try to fill the link with a lot of packets, without keeping any state. This means that this approach cannot be used for already existing applications that require an actual TCP stack, but also need the performance provided by DPDK. These applications would need to be rewritten to use a DPDK based networking stack instead of the Linux one.

One such application is the previously mentioned iperf3. It is integrated into perfSONAR, which is a network measurement toolkit used a lot on international research networks [7, 8, 9]. Currently, this toolkit is unable to measure the full speed of 100GbE links, because it cannot saturate them. It could be that modifying iperf3 to use DPDK would result in more throughput, possibly allowing perfSONAR to measure the full speed of these links.

1.1 Research questions

With these problems in mind and the hardware described in Section 3.2 this research will focus on three questions:

1. Can current DPDK packet generators saturate a 100GbE link?
2. What modifications to iperf3 are necessary to let it use DPDK?
3. What effect does the usage of DPDK have on the throughput of iperf3?

2 Related work

Quite some research has already been done into the effects of DPDK applications on high speed networks, but no research is available for using them on 100GbE connections. Both MoonGen and Pktgen have been researched on 10GbE networks [5, 6]. Pktgen is the only one of these which is directly compared to other packet generators. Even though both of them have not been tested on 100GbE connections, in theory MoonGen should be able to handle them, because speeds of up to 120 Gbit/s were reached using multiple 10GbE NICs [5].

Porting existing applications to use the DPDK stack is also not entirely new. Intel has modified Open vSwitch to make it possible to use DPDK. This causes a throughput increase of either two or six times depending on the research [10, 11]. Open vSwitch is similar to MoonGen and Pktgen in the fact that it also does not open TCP connections. It simply forwards packets and optionally does something special with them based on their contents, such as modifying, dropping or duplicating.

Apart from these applications, there is also a more general effort, called Accelerated Network Stack (ANS), to create a full TCP/IP stack based on DPDK [12]. This is done by porting the FreeBSD TCP/IP stack to DPDK. Both Nginx and Redis have been successfully modified to use DPDK in this way [13, 14]. For both of these ports some benchmark results are supplied.

However, these are not compared to equivalent benchmarks for the unmodified versions of the applications.

3 Approach

The research consists of three main parts. First of all, a baseline for performance is set. After that iperf3 is modified to use DPDK, as this is necessary to answer the last two research questions. Lastly, performance experiments are done to compare different versions of iperf3.

3.1 Focus

It is possible to do a multitude of different network throughput tests, but this research focusses on one specific type, namely the performance of a single TCP stream by using a single core. TCP is a commonly used protocol for sending data, because of its built in reliability and congestion control [15]. This makes it an interesting test subject, since the results could also be applicable for TCP applications other than traffic generators as they use the exact same protocol.

The performance of a single TCP stream with a single core is then the most interesting to look at. The reason being that the performance can then be increased further by using multiple streams on multiple cores, when the CPU core is the bottleneck. Although it is not the main focus, the performance of multiple streams on a single core is also compared. This is done, because an increase in throughput for multiple streams could mean that the limiting factor is not the computing power, but instead something else, such as the TCP protocol.

3.2 Setup and versions

The test setup that is used during the experiments consists of two identical 100GbE nodes connected to an Inventec D7032Q28B switch running in OpenFlow mode. A simple overview of the setup can be seen in Figure 3. The nodes have the following hardware:

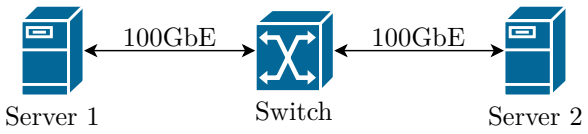

Figure 3: The test setup

Component	Version
Linux Distribution	CentOS 6.8
Linux Kernel	2.6.32 x86_64
DPDK	16.04
iperf3	3.1.3
ANS	22-06-2016 (commit 7f6b0fe494)
Pktgen	3.0.02

Table 1: Exact software versions of the setup

- Supermicro 5018R-MR
- 1 x Intel Xeon E5-1630V3
- 4 x Certified 8GB DDR3 2133mhz ecc reg
- 1 x Mellanox ConnectX-4, 2-port 100GbE, MCX416A-CCAT

The servers are installed using the perfSONAR ISO image. This installs CentOS 6.8 as a Linux distribution together with all the software needed to run the perfSONAR Toolkit. The exact versions of all the used software can be found in Table 1. Some slight modifications have been made to DPDK. This was done to make it compile on this setup, because MoonGen required the changes, and to compile the drivers for the Mellanox NIC. These modifications can be found in Appendix B.

3.3 Performance baseline

To get an idea of the performance that can be achieved by using a DPDK based packet generator a baseline is set by Pktgen. At first, a baseline from MoonGen was also planned, but during installation it became clear that it did not support the 100GbE NIC used in the experiments (Section 3.2).

TCP tests are done with Pktgen, but they differ quite a bit from the iperf3 TCP tests and are actually much more similar to its UDP tests. This is because Pktgen does not actually support TCP stream. It can only send raw packets over the interface. These packets can be in the TCP format, but they will not actually establish a connection over which a stream of data is being sent in accordance with the TCP protocol. It will simply be a packet stream consisting of TCP formatted packets. This makes the performance of iperf3 and Pktgen not directly comparable, but the performance of Pktgen can be used to set a baseline for the maximum speed that can be expected.

A second baseline test is performed, where multiple processes of iperf3 are running at the same time. This has the advantage that multiple cores can be utilized to send traffic, possibly benefiting the total throughput. This total throughput is calculated simply by adding the measured throughput of all processes together. Just like with Pktgen this performance is not directly comparable to the results of the other performance experiments in this research. In this case because it uses multiple processes and cores. However, it is still interesting too see what throughput can be achieved this way and if that is more or less than the other methods.

3.4 Modifying iperf3

DPDK does not have a TCP stack built in to it. It does supply means to generate TCP packets with certain fields, but no means to keep a connection open, which is also why Pktgen does not support this. As iperf3 needs to open a TCP connection to send data for its TCP tests such a stack is required. This is why ANS is be used, as it already contains a full TCP stack that is built on top of DPDK.

A new version of iperf3 is created that uses the TCP stack provided by ANS instead of the one from Linux. This version of iperf3 will be called iperf3-ANS. All the modifications that are necessary to use ANS for iperf3 are investigated and will be described in the results to answer the second research question.

Apart from iperf3-ANS, a second modified version of iperf3 will also be created. This one will still use the Linux networking stack. It will differ from iperf3 by containing the Linux compatible changes that were made to create iperf3-ANS. This version is used to find out what part of the performance difference is caused by DPDK and what part is caused simply by changing a part of the logic in the code.

3.5 Performance of iperf3 versions

After both these new iperf3 versions have been created their performance is compared with that of the original iperf3. Regular iperf3 supports a couple of protocols for its data streams. This research focusses on TCP performance, as described in Section 3.1, so all these tests are done with TCP streams.

3.5.1 Multiple streams with iperf3

One of the features of iperf3 is the possibility of opening multiple data streams by using the `--parallel` command line parameter. In cases where the congestion control features of TCP are causing a bottleneck this should allow for more total throughput, because these features are per stream. So, if the network and hardware can handle the throughput a second stream should have the same performance as the first, thus doubling the total throughput. The amount of streams is increased until no total throughput increase is seen to find out whether the perceived maximum throughput is caused by the CPU or network and not TCP.

3.6 Performance settings

A couple of default network and CPU settings on the two nodes have been changed to achieve better or more reliable performance.

3.6.1 TCP settings

The first of these are TCP features, window sizes, and buffer sizes. The values used for these have been based on the host tuning advice of ESnet [16, 17]. Some NIC specific tuning has also been done according to the tuning guide by the NIC vendor [18]. The exact settings used are shown in Appendix A.

3.6.2 Network settings

Apart from TCP settings, other network related settings have also been changed. The NIC that is used supports jumbo frames, which are Ethernet frames that can carry a payload larger than the maximum of 1500 bytes for regular frames [19]. Using these bigger frames decreases the amount of headers and packets that need to be created and thus the CPU cycles that are spent on it. To utilize this feature the Maximum Transmission Unit (MTU) has been changed from 1500 to 9000 bytes.

A second change that has been made is to enable all offloading capabilities of the NIC with `ethtool` [20]. This moves common network computations, such as checksum calculations, from the CPU to the NIC. This again decreases the load on the CPU, freeing cycles for other computations.

3.6.3 CPU settings

Changes have also been made to the way the CPU cores are used. First of all, CPU affinity is set for the benchmark processes [21]. This way the process will stick to a specific core, eliminating the overhead caused by side effects of moving the process to another core, such as invalidation of the cache and copying the registers. Previous research has shown that the core to which the process is pinned can matter for the performance as well, especially when multiple CPU sockets are used [22]. However, initial experiments done on different cores showed no difference in performance for the cores on our machines.

Secondly, Hyper-Threading is turned off [23]. This is to make sure the physical CPU cores are used instead of Hyper-Threaded ones. Hyper-Threaded cores have lower performance than physical cores, so it is important that the physical ones are used. Disabling it is also recommended by the NIC vendor when doing high-performance computing (HPC) [18].

Lastly, the `isolcpus` boot parameter of Linux is used to make sure nothing else is scheduled on the cores that are used by the benchmark processes [24, 25]. The value `0,1,2` is used for the parameter. This indicates to the kernel that it should not schedule any processes on these cores, except when a process requests it explicitly by using CPU affinity. All other processes will only be run on the cores that are left over, in this case this is only the fourth core.

3.6.4 Using non-optimal settings

During the study it was discovered that ANS currently does not support some of the features described above [26]. These features are NIC offloading, TCP window scaling, and jumbo frames. Because they are not supported they will also not be enabled in any of the tests done with iperf3-ANS. Jumbo frames are also not supported by Pktgen, so an MTU of 1500 is used there as well.

The Linux versions of iperf3 will not only be tested with the features enabled, but also without them. This is done to get an indication of the performance difference that is caused by these different feature sets as opposed to difference caused by the underlying networking stack.

3.6.5 ANS transmit burst buffer length

Finally, there is an ANS specific setting to specify the size of its packet transmit burst buffer [26]. The packet transmit burst buffer is used to temporarily store packets before sending them all at once to the NIC. Without changing this value a strange behaviour occurred. Using two data streams tripled the throughput of iperf3-ANS when compared to a single stream,

while doubling it would be the maximum expected improvement, see Section 4.3.4 for details.

It was suggested by the developer of ANS that the default size of the buffer could be the cause of this behaviour. The default size of 32 packets was chosen for performance reasons for a 10GbE NIC, but he suggested that it could have adverse effects on higher speed NICs such as ours. So finally, some tests are also done with the size of this buffer set to 10, the iperf3 version that uses this will be called iperf3-ANS-small. This size of 10 seemed to be relatively good, based on the single stream performance in initial tests. However, detailed investigation to find the optimal value has not been done.

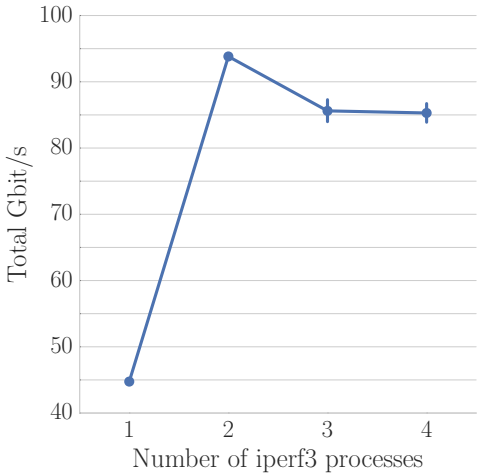

Figure 4: iperf3 test with multiple processes

3.7 Details of the experiments

All throughput measurements are done over a period of 10 seconds. These 10 seconds exclude the three second stabilization period at the beginning of each run. During these three seconds data is being sent, but it is not taken into account when calculating the throughput. This is done to make sure the connection has stabilized and that the TCP slow start is not impacting the performance measurement [27].

For Pktgen, not only the throughput is measured, but also the packet loss, since it does not use TCP. This packet loss is calculated over all packets that are sent, so also the stabilization period. This is done, because Pktgen does not include a communication channel to synchronize the start of the actual experiment.

Almost all of these experiments have been performed 14 times. The only experiments that were repeated a different number of times were the experiments for iperf3-ANS-small. After the first 14 runs the variance was much larger than for the other experiments. So, 16 extra runs were executed to get more reliable results, resulting in a total of 30 runs.

4 Results

The results of the research will be presented in the same order in which they have been described in the previous section. First, the baseline experiments will be shown, then the steps necessary to port iperf3 to DPDK, and finally the results of the different iperf3 versions. All of the plots show the average of the results of the experiments and contain error bars indicating a confidence interval of 95%. These error bars can be very small in cases where there was a low variance in the results.

4.1 Performance baseline

As described in Section 3.3, two performance baseline tests are executed. One with Pktgen and another with multiple iperf3 instances.

The average throughput that could be achieved with Pktgen was 96.3 Gbit/s, with a standard deviation of 0.1 Gbit/s. The packet loss was only 0.000 085 % with a standard deviation of 0.000 041. Although this is not the 100 Gbit/s line rate, it is higher than the speed that could be achieved with a any number of iperf3 process, as can be seen in Figure 4. The

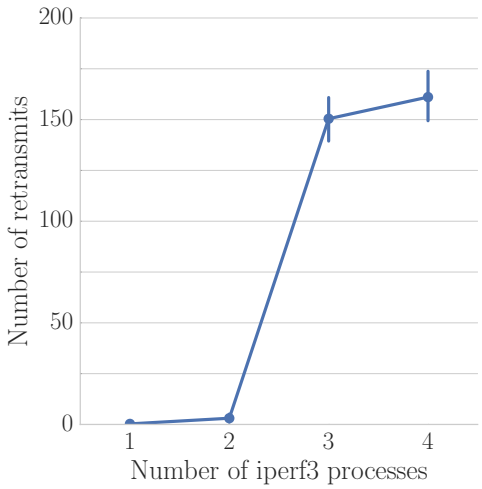

Figure 5: Number of retransmits for multiple iperf3 processes

average throughput that could be achieved with a single iperf3 process is 45 Gbit/s. Opening multiple iperf3 processes increases this throughput. With two processes a total throughput of 94 Gbit/s is achieved, which goes down to 86 Gbit/s with three processes and to 85 Gbit/s with four. The average number of TCP retransmits also increases a lot when using more than two processes, going from almost none to around 150 retransmits as can be seen in Figure 5. This performance drop and rise in retransmits is discussed in Section 5.1 and some possible explanations are given as well.

4.2 Modifying iperf3

The API supplied by ANS is very similar to the networking API available in Linux. It has functions that correspond with the ones from the socket API and the `epoll` event loop API [28, 29]. These corresponding functions are simply prefixed with the string "`anssock_`". This should allow for very easy conversion when the corresponding Linux API is already used. However, when converting iperf3 to the ANS API two main problems were found.

4.2.1 Converting `select` to `epoll`

The first problem was the fact that iperf3 uses an event loop based on `select` instead of `epoll`. This type of event loop has existed longer than `epoll` and it is available on platforms other than Linux as well, but it is not supported by ANS. So the first step was to convert the `select` event loop to an `epoll` one. There are resources available on how to use `epoll` and `select` and some also describe differences between `select` and `epoll`, but none of the resources that were found explained concisely how to convert software from `select` to `epoll` [29, 30, 31, 32, 33]. Using these resources the differences between `epoll` and `select` that were important when converting iperf3 were found, an overview of these conversion techniques can be found in Table 2. Important to note is that, although `epoll` supports a type of events called *edge-triggered*, the default *level-triggered* type is used, since this behaviour is similar to that of `select` [34].

The first one is the way in which a file descriptor (FD) of a socket can be watched for events. For `select` it is necessary to add the FD to one of the three FD sets that are passed to the `select` function when it is called, the read, write and error set. For `epoll` it is necessary that an `epoll` instance is first created. FDs can then be added to this `epoll` instance by calling the `epoll_ctl` function. This has to be done in combination with an `epoll_event` struct that has flags set that specify for which events a notification should be triggered. This conversion is pretty straightforward and an example for read set conversion is shown in the `FD_SET` row of Table 2. The conversion for the write and error set are the same, except that another flag than `EPOLLIN` should be used.

The second difference is the way in which events are received. For `select`, the `select` function has to be called with the FD sets. This call will block until an event happens. Then all file descriptors that have been added to an FD set need to be checked to see if they were the cause of the event by using `FD_ISSET`. After this, something can be done with the FD, such as reading it if a read event occurred. This is quite different for `epoll`. It simply returns a list of the

`select` code	`epoll` code
`FD_SET(socket, read_set)`	`event.events = EPOLLIN;`
	`event.data.fd = socket;`
	`epoll_ctl(epoll_instance, EPOLL_CTL_ADD, socket, event);`
`select(...);`	`epoll_wait(...);`
`FD_ISSET(socket, read_set)`	`(event.events & EPOLLIN) && event.fd == socket`
`FD_CLR(socket, fd_set);`	*Remove*
`FD_ZERO(fd_set);`	*Remove*

Table 2: Basic conversions from `select` to `epoll`

`epoll_event` structs that were triggered. There is no standard way to access the corresponding FD of these events. However, one can easily be created by adding the FD to the data part of the event before it is added to the `epoll` instance with `epoll_ctl`.

The last difference is that `select` requires calls to macros to modify the FD sets after an event has been handled. Otherwise the previously mentioned `FD_ISSET` macro will indicate that an FD triggered the event even when that is not the case. This is not the case for `epoll`, as it only returns the events that have occurred. So all that is needed is simply to remove the macro calls.

4.2.2 Remove blocking sockets

Apart from requiring `epoll` there is a big difference in the way the ANS socket API works, when compared to the POSIX version. This is that all of the functions are nonblocking only, which means that it is not possible to use blocking sockets.

This has an impact on every piece of the codebase where a read or write takes place that requires a follow up action after it completes. With blocking sockets this was as simple as simply placing some code after the call to `read` or `write`. This code would then simply be executed after the call completes. With nonblocking sockets a state needs to be kept and after an event occurs this state needs to be checked to see what the program should do to continue.

The conversion between these two methods can be done by placing the code after the blocking `read` or `write` in a separate function. A state value needs to be saved somewhere, for instance an hash table that is indexed by the FD of the socket. Then, when an event occurs on the socket the state value can be matched to the follow up function.

The method used to convert iperf3 in this research was similar, but less sophisticated. A global state for the connection was already kept, so simply more options were added to this instead of using a hash table. A second solution that was used, was to merge to consecutive writes into a single one, resulting in a single read event at the other side.

4.3 Performance of modified iperf3

In this section the performance of the different versions of iperf3 is compared under different circumstances. To distinguish the different versions in the results they will be called the following:

- iperf3: The original iperf3 without any modifications.
- iperf3-epoll: The version of iperf3 modified to use the Linux `epoll` event loop and only nonblocking sockets.
- iperf3-ANS: The version that is modified to use the ANS library.
- iperf3-ANS-small: The same as iperf3-ANS, but with a smaller transmit burst buffer as described in Section 3.6.5.

In the plots iperf3 is called “Regular” and the others simply have the iperf3 prefix removed, e.g iperf3-ANS is called “ANS”.

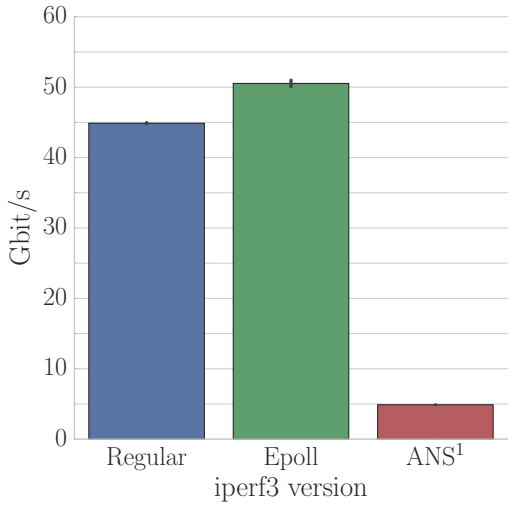

Figure 6: Initial performance comparison with a single stream and optimal settings

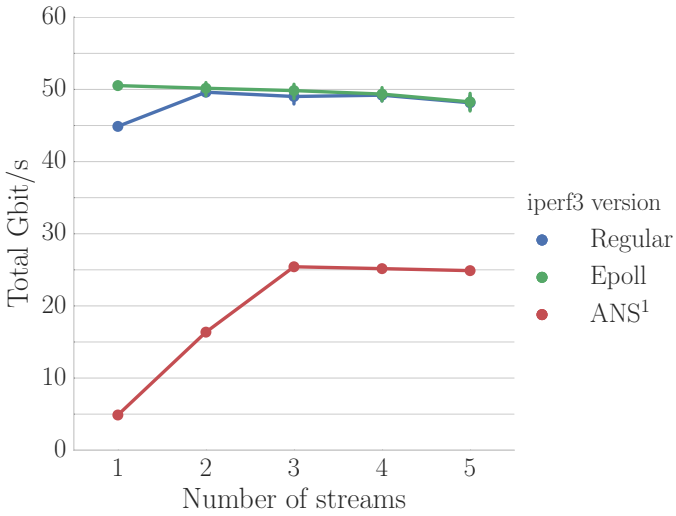

Figure 7: Performance comparison with multiple TCP streams and optimal settings

4.3.1 Initial results

The first results we will look at are the initial results that were achieved with the optimal settings for Linux described in Section 3.6. Keep in mind that iperf3-ANS misses support for a couple of the features that Linux supports and thus does not have these enabled, as described in Section 3.6.4. The results from the experiments where these features are disabled for Linux as well are available in Section 4.3.3. As you can see in Figure 6, iperf3 can reach about 45 Gbit/s and iperf3-epoll is even faster and can reach around 50 Gbit/s. However, iperf3-ANS is much slower than the others and is only able to reach around 5 Gbit/s. So it performs significantly worse than both others, being 9 times slower than iperf3 and 10 times slower than iperf3-epoll.

[1]NIC offloading, TCP window scaling, and jumbo frames are not enabled for this iperf3 version, because ANS does not support them.

4.3.2 Multiple streams

The differences in achievable throughput between versions is much smaller when looking at the results for multiple streams, found in Figure 7. Still, iperf3-ANS is not as fast or faster than the others, but instead of being 9 or 10 times slower it is now only 2 times slower when looking at three or more concurrent streams. Something else to notice is that iperf3 also benefits from using multiple streams. With two or more it performs about the same as iperf3-epoll, with a maximum close to 50 Gbit/s.

4.3.3 Disabling missing features

As explained in Section 3.6.4 some performance features that are available in Linux are not available in ANS. When disabling each of these features separately for Linux it becomes clear that all of them have a noticeable effect on the performance, which can be seen in Figure 8. Keep in mind that iperf3-ANS has all the features disabled in each of the graphs as it does not actually support them.

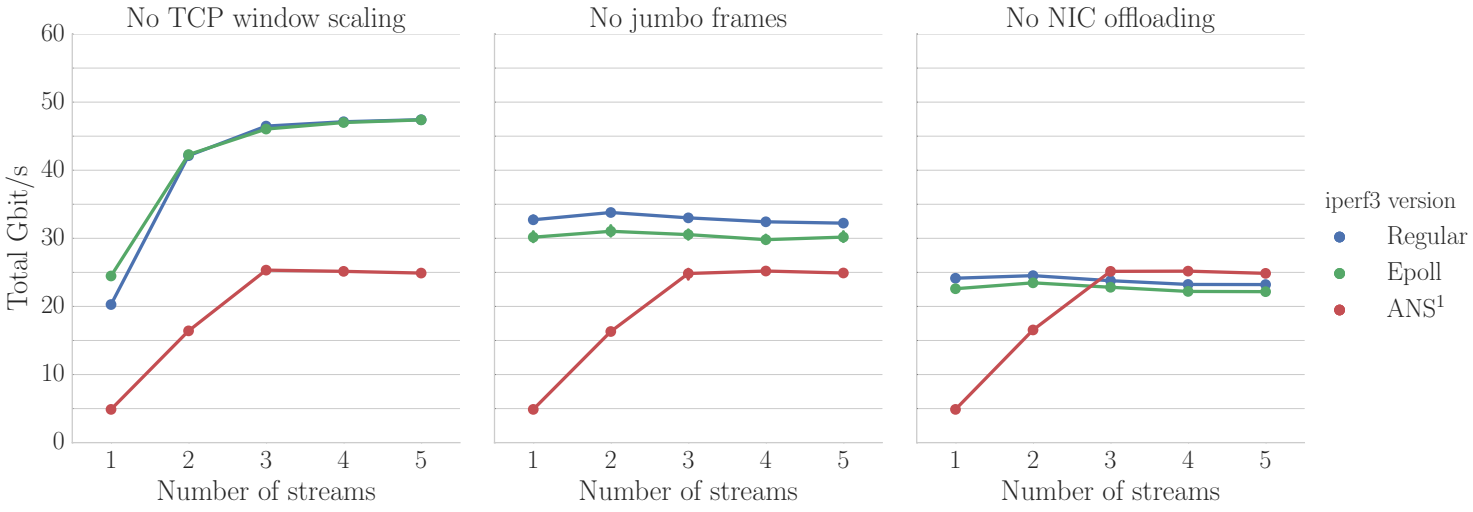

Figure 8: Performance comparison with the missing ANS performance features disabled separately for Linux

Window scaling mostly affects performance of a single stream, since it reaches a maximum of around 48 Gbit/s with higher number of streams, which is close to the 50 Gbit/s maximum that was reached with window scaling enabled. This is not the case for the other two features, jumbo frames and NIC offloading. These features seem to affect the throughput of any amount of streams, by affecting the maximum throughput that can be reached. When disabling one of these two features the maximum is lower than it was when it was enabled.

Disabling jumbo frames lowers the maximum to about 32 to 34 Gbit/s for iperf3 and to about 30 Gbit/s for iperf3-epoll. This is still higher than the maximum of iperf3-ANS. Disabling the computational offloading to the NIC has an even bigger impact. This lowers the maximum to about 24 to 25 Gbit/s resulting in a maximum that is even slightly lower with more streams than the maximum of iperf3-ANS.

Finally, if all these features are disabled at the same time the speed that can be reached is only 6 Gbit/s for iperf3 and 7 Gbit/s for iperf3-epoll, which can be seen in Figure 9. This is much lower than the 25 Gbit/s maximum of iperf3-ANS, but strangely its single stream performance is still slightly worse than that of the other two.

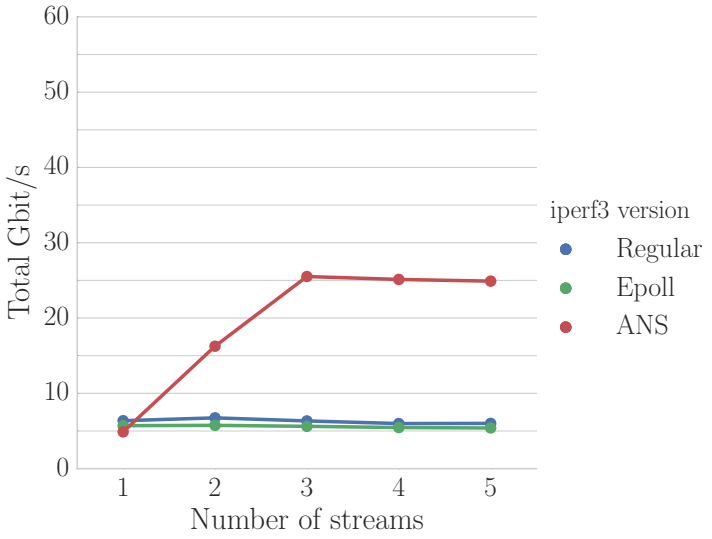

Figure 9: Performance comparison without all the features that are missing in ANS

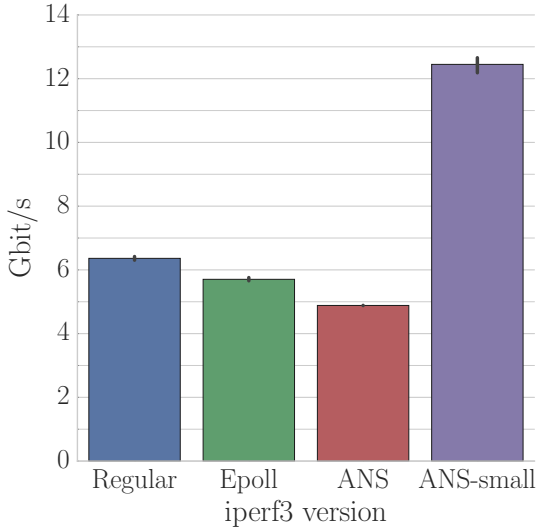

Figure 10: Performance comparison with a single TCP stream with performance features disabled that are missing from ANS

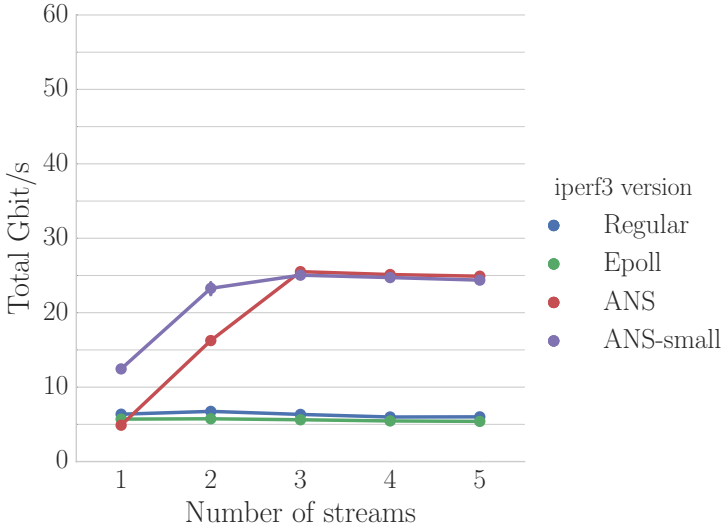

Figure 11: Performance comparison with multiple TCP streams with performance features disabled that are missing from ANS

4.3.4 Decreasing the transmit burst buffer

What is especially strange about the fact that iperf3-ANS can only achieve 5 Gbit/s with single stream is that it can reach 16 Gbit/s with two streams. Thus tripling the throughput instead of doubling it, with each separate stream reaching 8 Gbit/s of throughput instead of 5. A possible method to increase the performance for a single stream was decreasing the transmit burst buffer as explained in Section 3.6.5.

As can be seen in Figure 10 decreasing the size of the transmit buffer did indeed have a positive effect on the throughput of a single stream. The average throughput of a single stream for iperf3-ANS-small is 12 Gbit/s, which is about two times more than that of iperf3.

For two streams an increase is also seen in Figure 11 when comparing iperf3-ANS-small to iperf3-ANS. In this case it goes from around 16 to 23 Gbit/s. However, for three or more streams the performance seems to have degraded a bit, decreasing the maximum to slightly below the maximum of iperf3-ANS.

5 Discussion

5.1 Baseline experiments

In none of the baseline experiments the line rate of 100 Gbit/s was reached. The setup that came the closest to line rate was Pktgen at 96.3 Gbit/s. After that the setup with two simultaneous iperf3 processes was fastest at 94 Gbit/s. However, the total throughput went down with even more processes, which was unexpected. If the throughput had stayed at the same level it would appear that a maximum was reached that could not be passed simply by using more processes. However, as the performance actually went down it seems that using more than two processes in this setup was actually hurting the throughput.

A part of the explanation would be that the number of TCP retransmits also went up when using more than three processes. This indicates that packets are lost and have to be retransmitted. An explanation for this is that more data is being sent than the link can handle, because three processes each sending data

each at 45 Gbit/s would produce output above the line rate of 100 Gbit/s. The retransmissions should then trigger TCP congestion control to lower its transmit speed. So, it could be that the lower throughput is reached, because the different processes constantly increase their speed to what the CPU allows and then lower it again, because of the TCP congestion control.

That four processes did not perform better than three is also unexpected, although not as unexpected as the drop. For the fourth core less performance is to be expected, because context switches are needed to switch between the iperf3 process and the other processes that are scheduled there because of the `isolcpus` value (explained in Section 3.6.3). However, that no performance improvement was gained by using the fourth core is quite strange.

That Pktgen was able to reach at 96.3 Gbit/s without jumbo frames and only a single process is quite impressive, because as iperf3 was only able to get a performance of 33 Gbit/s in those conditions. However, as stated before this is not an entirely fair comparison as iperf3 actually uses a TCP connection and Pktgen only sends raw packets.

5.2 Converting iperf3 to ANS

By using ANS it is possible to convert a regular Linux application to use DPDK with little work. Converting an application that uses a `select` event loop, requires conversion to a `epoll` event loop. This is not difficult, as function calls either have an `epoll` equivalent or they can simply be removed. The conversion to non-blocking sockets is less straightforward, but could be done in a standard way as well.

5.3 Performance of modified iperf3

A single iperf3 process did not come close to the line rate of 100 Gbit/s with a throughput of 45 Gbit/s for a single stream and 50 Gbit/s with multiple streams. However, the two ANS versions performed even worse, reaching at most 12 and 25 Gbit/s for a single stream and multiple streams respectively. This was unexpected, as DPDK was developed to be faster at networking than the Linux kernel and Pktgen also showed that a much higher speed was possible with DPDK.

This unexpected result seems to be caused by the fact that ANS does not support a couple of the performance features that Linux does support. Disabling these features for Linux makes iperf3 perform between 2 and 4 times worse than iperf3-ANS-small, depending on the number of TCP streams, which is more in line with the performance difference that was expected.

With these performance features it is likely that iperf3-ANS-small would be faster than the current Linux iperf3. Although it is hard to say what performance could be expected, some reasonable predictions can be made.

With TCP window scaling enabled the maximum throughput of 25 Gbit/s could probably be achieved with a single stream, instead of requiring multiple streams to reach it. Support for jumbo frames and NIC offloading should both increase this throughput maximum.

With some assumptions it is possible to estimate this new maximum based on the measured increase for the Linux stack. The necessary assumptions are that increase in maximum caused be the features is relative to the maximum without them and that this increase is the same for ANS. For Linux the increase in maximum was from 7 to 50 Gbit/s, which is a relative increase of 7.14 times. So, this would mean that the new maximum of iperf3-ANS-small would be around 178.5 Gbit/s (25×7.14).

This new maximum seems quite high, given that Pktgen was only able to get 96.3 Gbit/s without opening actual TCP connections. However, Pktgen does not support jumbo frames either so its performance would probably be increased as well by using them.

5.4 Future work

There are a couple of directions for future research that are related to this research. The first one would

be to compare the performance of the different iperf3 versions again after ANS supports the currently missing features. It would be very interesting to see if that would indeed make the DPDK based iperf3 versions outperform the Linux ones.

MoonGen is another project that would be interesting to investigate at speeds of 100GbE. This was not possible in this research as MoonGen did not support the available NIC. If this support is added, or if another 100GbE NIC is used it would be interesting to see how it compares against Pktgen.

The chosen transmit burst buffer size for iperf3-ANS-small was already much better than the default one that was used for iperf3-ANS. However, no research was done into finding the perfect size. So, it might be possible that another size works even better and experiments would need to be done to find the best value. A similar experiment can be done for Pktgen, as it also has a transmit burst buffer.

A strange finding was the performance drop that was found when increasing the number of iperf3 processes from two to three. It would be interesting to do more extensive experiments in this area to determine the cause of this performance drop. Experiments could be done on machines with different numbers of cores to see if they behave differently.

Finally, it would be interesting to look at the UDP performance of the different iperf3 versions instead of their TCP performance. This would make the results for iperf3 also directly comparable to those of the two DPDK packet generators, i.e. MoonGen and Pktgen. This would require some more changes to iperf3-ANS though, as it does not support UDP streams at the time of writing.

6 Conclusion

This research has evaluated the use of DPDK as a method to saturate a 100GbE link. It has also investigated what modifications are necessary to iperf3 to make it use DPDK and the effects of these modifications on the throughput.

Using a raw packet stream generated by the DPDK based Pktgen a throughput of 96.3 Gbit/s was reached. This throughput is only 3.7 Gbit/s lower than the maximum that is imposed by the link. It is also more throughput than was achieved with a Linux version of iperf3, which was 45 Gbit/s for a single process and 94 Gbit/s for multiple processes on multiple cores.

Porting iperf3 to DPDK was also successfully done by using the Accelerated Network Stack (ANS). The changes that were required were either straightforward replacements of certain function calls, or could be done using a standardized conversion method. However, even though Pktgen showed that high throughput was possible with DPDK, this ported version of iperf3 did not outperform the original iperf3.

The cause for the relatively bad performance of the modified iperf3 was likely that some performance features are missing from ANS. These features are TCP congestion window scaling, NIC offloading, and jumbo frames. When disabling these features for Linux as well, the modified iperf3 did outperform the original iperf3 by 2 to 4 times, depending on the amount of opened TCP streams. This leads us to believe that ANS might be used in the future to improve iperf3 performance once these features have been added, even though currently the use of ANS only reduces the performance.

References

[1] ESnet. *iperf3*. 2016. URL: http://software.es.net/iperf/index.html (visited on 04/06/2016).

[2] EXFO. *40G/100G Multiservice Test Modules — FTB/IQS-85100G Packet Blazer*. URL: http://www.exfo.com/products/field-network-testing/bu2-transport-datacom/ethernet-testing/ftb-iqs-85100g-packet-blazer (visited on 05/07/2016).

[3] Intel. *Data Plane Development Kit*. 2014. URL: http://dpdk.org (visited on 04/06/2016).

[4] Keith Wiles. *The Pktgen Application*. 2015. URL: https://pktgen.readthedocs.io/en/latest/ (visited on 04/06/2016).

[5] Paul Emmerich et al. 'MoonGen: A Scriptable High-Speed Packet Generator'. In: *Internet Measurement Conference 2015 (IMC'15)*. Tokyo, Japan, Oct. 2015.

[6] Daniel Turull, Peter Sjödin and Robert Olsson. 'Pktgen: Measuring performance on high speed networks'. In: *Computer Communications* 82 (2016), pp. 39–48.

[7] Andreas Hanemann et al. 'Perfsonar: A service oriented architecture for multi-domain network monitoring'. In: *Service-Oriented Computing-ICSOC 2005*. Springer, 2005, pp. 241–254.

[8] Brian Tierney et al. 'perfSONAR: Instantiating a global network measurement framework'. In: *SOSP Wksp. Real Overlays and Distrib. Sys* (2009).

[9] perfSONAR. *About perfSONAR*. URL: http://www.perfsonar.net/about/ (visited on 04/06/2016).

[10] Paul Emmerich et al. 'Assessing soft-and hardware bottlenecks in PC-based packet forwarding systems'. In: *ICN 2015* (2015), pp. 78–83.

[11] Ashok Emani. *Using Open vSwitch* with DPDK for Inter-VM NFV Applications*. 17th Nov. 2015. URL: https://software.intel.com/en-us/articles/using-open-vswitch-with-dpdk-for-inter-vm-nfv-applications (visited on 04/06/2016).

[12] opendp. *TCP/IP stack for dpdk*. 2014. URL: https://github.com/opendp/dpdk-ans (visited on 04/06/2016).

[13] opendp. *dpdk-nginx*. 2015. URL: https://github.com/opendp/dpdk-nginx (visited on 04/06/2016).

[14] opendp. *dpdk-redis*. 2015. URL: https://github.com/opendp/dpdk-redis (visited on 04/06/2016).

[15] Jon Postel. *Transmission Control Protocol*. STD 7. RFC Editor, Sept. 1981. URL: http://www.rfc-editor.org/rfc/rfc793.txt.

[16] ESnet. *Linux Tuning*. URL: https://fasterdata.es.net/host-tuning/linux/ (visited on 18/06/2016).

[17] ESnet. *40G Tuning*. URL: https://fasterdata.es.net/host-tuning/40g-tuning/ (visited on 18/06/2016).

[18] Mellanox Technologies LTD. *Performance Tuning Guidelines for Mellanox Network Adapters Revision 1.17*. 2016. URL: https://www.mellanox.com/related-docs/prod_software/Performance_Tuning_Guide_for_Mellanox_Network_Adapters.pdf (visited on 18/06/2016).

[19] Ethernet Alliance. *Ethernet Jumbo Frames*. 2009. URL: http://www.ethernetalliance.org/wp-content/uploads/2011/10/EA-Ethernet-Jumbo-Frames-v0-1.pdf.

[20] *ethtool(8) - Linux man page*. URL: http://linux.die.net/man/8/ethtool (visited on 06/07/2016).

[21] Robert Love. 'Cpu affinity'. In: *Linux Journal* 111 (2003), pp. 18–21. URL: https://www.linuxjournal.com/article/6799.

[22] Nathan Hanford et al. 'Characterizing the impact of end-system affinities on the end-to-end performance of high-speed flows'. In: *Proceedings of the Third International Workshop on Network-Aware Data Management*. ACM. 2013, p. 1.

[23] Deborah T Marr et al. 'Hyper-Threading Technology Architecture and Microarchitecture.' In: *Intel Technology Journal* 6.1 (2002).

[24] Linuxtopia. *isolcpus — Isolate CPUs from the kernel scheduler*. URL: http://www.linuxtopia.org/online_books/linux_kernel/kernel_configuration/re46.html (visited on 06/07/2016).

[25] Micro Focus. *Isolating CPUs From The General Scheduler*. URL: https://www.novell.com/support/kb/doc.php?id=7009596 (visited on 06/07/2016).

[26] Jelte Fennema and bluestar. *dpdk-ans is slower than regular Linux epoll with 100Gbit/s*. 2016. URL: https://github.com/opendp/dpdk-ans/issues/16 (visited on 06/07/2016).

[27] W. Richard Stevens. *TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms*. RFC 2001. http://www.rfc-editor.org/rfc/rfc2001.txt. RFC Editor, Jan. 1997. URL: http://www.rfc-editor.org/rfc/rfc2001.txt.

[28] *socket - Linux socket interface*. URL: http://man7.org/linux/man-pages/man7/socket.7.html (visited on 06/07/2016).

[29] *epoll - I/O event notification facility*. URL: http://man7.org/linux/man-pages/man7/epoll.7.html (visited on 06/07/2016).

[30] George Yunaev. *select/poll/epoll: practical difference for system architects*. 2014. URL: http://www.ulduzsoft.com/2014/01/select-poll-epoll-practical-difference-for-system-architects/ (visited on 04/06/2016).

[31] Oleksiy Kovyrin. *Using epoll() For Asynchronous Network Programming*. 2006. URL: http://kovyrin.net/2006/04/13/epoll-asynchronous-network-programming/ (visited on 06/07/2016).

[32] Mukund Sivaraman. *How to use epoll? A complete example in C*. 2011. URL: https://banu.com/blog/2/how-to-use-epoll-a-complete-example-in-c/ (visited on 06/07/2016).

[33] *select, pselect, FD_CLR, FD_ISSET, FD_SET, FD_ZERO - synchronous I/O multiplexing*. URL: http://man7.org/linux/man-pages/man2/select.2.html (visited on 06/07/2016).

[34] Jonathan Corbet. *Edge-triggered interfaces are too difficult?* 2003. URL: https://lwn.net/Articles/25137/ (visited on 06/07/2016).

Acronyms

100GbE 100 Gigabit Ethernet. 1–3, 13

10GbE 10 Gigabit Ethernet. 2, 6

ANS Accelerated Network Stack. 2–13

API Application Programming Interface. 7, 8

CPU Central Processing Unit. 1, 3–5, 12

DPDK Data Plane Development Kit. 1–4, 6, 12, 13, 17

FD file descriptor. 7, 8

HPC high-performance computing. 5

IP Internet Protocol. 2

MTU Maximum Transmission Unit. 5

NIC Network Interface Card. 1–3, 5, 6, 9, 10, 12, 13

POSIX Portable Operating System Interface. 8

TCP Transmission Control Protocol. 2–7, 9, 11–13

UDP User Datagram Protocol. 4, 13

A Tuned settings

A.1 `sysctl.conf`

```
# Controls IP packet forwarding
net.ipv4.ip_forward = 0

# Controls source route verification
net.ipv4.conf.default.rp_filter = 1

# Do not accept source routing
net.ipv4.conf.default.accept_source_route = 0

# Controls the System Request debugging functionality of the kernel
kernel.sysrq = 0

# Controls whether core dumps will append the PID to the core filename.
# Useful for debugging multi-threaded applications.
kernel.core_uses_pid = 1

# Controls the use of TCP syncookies
net.ipv4.tcp_syncookies = 1

# Controls the default maxmimum size of a mesage queue
kernel.msgmnb = 65536

# Controls the maximum size of a message, in bytes
kernel.msgmax = 65536

# Controls the maximum shared segment size, in bytes
kernel.shmmax = 68719476736

# Controls the maximum number of shared memory segments, in pages
kernel.shmall = 4294967296
vm.nr_hugepages = 1024

net.core.rmem_max = 134217728
net.core.wmem_max = 134217728

net.ipv4.tcp_rmem = 4096 87380 67108864
net.ipv4.tcp_wmem = 4096 65536 67108864

net.core.netdev_max_backlog = 250000

net.ipv4.tcp_congestion_control=htcp

net.ipv4.tcp_mtu_probing=1
```

```
net.ipv4.tcp_window_scaling=1

# Needed for ANS
kernel.randomize_va_space=0
```

B DPDK modifications

```
diff -ur dpdk-16.04/config/common_base dpdk-16.04-modified/config/common_base
--- dpdk-16.04/config/common_base        2016-04-11 23:56:34.000000000 +0200
+++ dpdk-16.04-modified/config/common_base        2016-06-28 11:30:48.285820155 +0200
@@ -125,7 +125,7 @@
 CONFIG_RTE_LIBRTE_ETHDEV_DEBUG=n
 CONFIG_RTE_MAX_ETHPORTS=32
 CONFIG_RTE_MAX_QUEUES_PER_PORT=1024
-CONFIG_RTE_LIBRTE_IEEE1588=n
+CONFIG_RTE_LIBRTE_IEEE1588=y
 CONFIG_RTE_ETHDEV_QUEUE_STAT_CNTRS=16
 CONFIG_RTE_ETHDEV_RXTX_CALLBACKS=y

@@ -213,7 +213,7 @@
 #
 # Compile burst-oriented Mellanox ConnectX-4 (MLX5) PMD
 #
-CONFIG_RTE_LIBRTE_MLX5_PMD=n
+CONFIG_RTE_LIBRTE_MLX5_PMD=y
 CONFIG_RTE_LIBRTE_MLX5_DEBUG=n
 CONFIG_RTE_LIBRTE_MLX5_SGE_WR_N=4
 CONFIG_RTE_LIBRTE_MLX5_MAX_INLINE=0
diff -ur dpdk-16.04/drivers/net/i40e/i40e_rxtx.c
 ↪  dpdk-16.04-modified/drivers/net/i40e/i40e_rxtx.c
--- dpdk-16.04/drivers/net/i40e/i40e_rxtx.c        2016-04-11 23:56:34.000000000 +0200
+++ dpdk-16.04-modified/drivers/net/i40e/i40e_rxtx.c        2016-06-28 11:30:50.956772983
 ↪  +0200
@@ -50,6 +50,8 @@
 #include <rte_tcp.h>
 #include <rte_sctp.h>
 #include <rte_udp.h>
+#include <rte_branch_prediction.h>
+

 #include "i40e_logs.h"
 #include "base/i40e_prototype.h"
@@ -1619,8 +1621,10 @@
                                                I40E_TX_FLAG_L2TAG1_SHIFT;
                }

-               /* Always enable CRC offload insertion */
```

```
-               td_cmd |= I40E_TX_DESC_CMD_ICRC;
+       /* Enable L2 checksum offload */
+       if (likely(!(ol_flags & PKT_TX_NO_CRC_CSUM)))
+             td_cmd |= I40E_TX_DESC_CMD_ICRC;
+

                /* Enable checksum offloading */
                cd_tunneling_params = 0;
diff -ur dpdk-16.04/drivers/net/ixgbe/ixgbe_rxtx.c
 ↪  dpdk-16.04-modified/drivers/net/ixgbe/ixgbe_rxtx.c
--- dpdk-16.04/drivers/net/ixgbe/ixgbe_rxtx.c        2016-04-11 23:56:34.000000000 +0200
+++ dpdk-16.04-modified/drivers/net/ixgbe/ixgbe_rxtx.c        2016-06-28 11:30:50.821775366
 ↪  +0200
@@ -787,8 +787,10 @@
                 * are only set in the last Data Descriptor:
                 *   - IXGBE_TXD_CMD_RS
                 */
-               cmd_type_len = IXGBE_ADVTXD_DTYP_DATA |
-                       IXGBE_ADVTXD_DCMD_IFCS | IXGBE_ADVTXD_DCMD_DEXT;
+        cmd_type_len = IXGBE_ADVTXD_DTYP_DATA | IXGBE_ADVTXD_DCMD_DEXT;
+        if (!(ol_flags & PKT_TX_NO_CRC_CSUM))
+            cmd_type_len |= IXGBE_ADVTXD_DCMD_IFCS;
+

 #ifdef RTE_LIBRTE_IEEE1588
                if (ol_flags & PKT_TX_IEEE1588_TMST)
diff -ur dpdk-16.04/lib/librte_eal/linuxapp/igb_uio/compat.h
 ↪  dpdk-16.04-modified/lib/librte_eal/linuxapp/igb_uio/compat.h
--- dpdk-16.04/lib/librte_eal/linuxapp/igb_uio/compat.h        2016-04-11 23:56:34.000000000
 ↪  +0200
+++ dpdk-16.04-modified/lib/librte_eal/linuxapp/igb_uio/compat.h        2016-06-28
 ↪  11:30:51.076770862 +0200
@@ -15,13 +15,13 @@
 #define HAVE_PTE_MASK_PAGE_IOMAP
 #endif

-#ifndef PCI_MSIX_ENTRY_SIZE
+#ifndef PCI_MSIX_ENTRY_CTRL_MASKBIT
 #define PCI_MSIX_ENTRY_SIZE             16
-#define  PCI_MSIX_ENTRY_LOWER_ADDR      0
-#define  PCI_MSIX_ENTRY_UPPER_ADDR      4
-#define  PCI_MSIX_ENTRY_DATA            8
-#define  PCI_MSIX_ENTRY_VECTOR_CTRL     12
-#define   PCI_MSIX_ENTRY_CTRL_MASKBIT   1
+#define PCI_MSIX_ENTRY_LOWER_ADDR      0
+#define PCI_MSIX_ENTRY_UPPER_ADDR      4
+#define PCI_MSIX_ENTRY_DATA            8
+#define PCI_MSIX_ENTRY_VECTOR_CTRL     12
```

```
+#define PCI_MSIX_ENTRY_CTRL_MASKBIT   1
 #endif

 #if LINUX_VERSION_CODE < KERNEL_VERSION(2, 6, 34) && \
diff -ur dpdk-16.04/lib/librte_eal/linuxapp/kni/ethtool/igb/kcompat.h
 ↪  dpdk-16.04-modified/lib/librte_eal/linuxapp/kni/ethtool/igb/kcompat.h
--- dpdk-16.04/lib/librte_eal/linuxapp/kni/ethtool/igb/kcompat.h        2016-04-11
 ↪  23:56:34.000000000 +0200
+++ dpdk-16.04-modified/lib/librte_eal/linuxapp/kni/ethtool/igb/kcompat.h        2016-06-28
 ↪  11:30:51.107770314 +0200
@@ -319,11 +319,11 @@
         __be16                h_vlan_encapsulated_proto;
 };
 #define vlan_hdr _kc_vlan_hdr
+#endif /* NETIF_F_HW_VLAN_TX && NETIF_F_HW_VLAN_CTAG_TX */
 #if ( LINUX_VERSION_CODE < KERNEL_VERSION(3,10,0) )
 #define vlan_tx_tag_present(_skb) 0
 #define vlan_tx_tag_get(_skb) 0
 #endif
-#endif /* NETIF_F_HW_VLAN_TX && NETIF_F_HW_VLAN_CTAG_TX */

 #ifndef VLAN_PRIO_SHIFT
 #define VLAN_PRIO_SHIFT 13
diff -ur dpdk-16.04/lib/librte_eal/linuxapp/kni/ethtool/ixgbe/kcompat.h
 ↪  dpdk-16.04-modified/lib/librte_eal/linuxapp/kni/ethtool/ixgbe/kcompat.h
--- dpdk-16.04/lib/librte_eal/linuxapp/kni/ethtool/ixgbe/kcompat.h        2016-04-11
 ↪  23:56:34.000000000 +0200
+++ dpdk-16.04-modified/lib/librte_eal/linuxapp/kni/ethtool/ixgbe/kcompat.h        2016-06-28
 ↪  11:30:51.091770598 +0200
@@ -318,11 +318,12 @@
         __be16                h_vlan_encapsulated_proto;
 };
 #define vlan_hdr _kc_vlan_hdr
+#endif
+
 #if ( LINUX_VERSION_CODE < KERNEL_VERSION(3,10,0) )
 #define vlan_tx_tag_present(_skb) 0
 #define vlan_tx_tag_get(_skb) 0
 #endif
-#endif

 #ifndef VLAN_PRIO_SHIFT
 #define VLAN_PRIO_SHIFT 13
diff -ur dpdk-16.04/lib/librte_eal/linuxapp/kni/kni_misc.c
 ↪  dpdk-16.04-modified/lib/librte_eal/linuxapp/kni/kni_misc.c
--- dpdk-16.04/lib/librte_eal/linuxapp/kni/kni_misc.c        2016-04-11 23:56:34.000000000
 ↪  +0200
```

```
+++ dpdk-16.04-modified/lib/librte_eal/linuxapp/kni/kni_misc.c        2016-06-28
↪  11:30:51.128769943 +0200
@@ -432,9 +432,6 @@
         up_read(&knet->kni_list_lock);

         net_dev = alloc_netdev(sizeof(struct kni_dev), dev_info.name,
-#ifdef NET_NAME_UNKNOWN
-                                                        NET_NAME_UNKNOWN,
-#endif
                                                         kni_net_init);
         if (net_dev == NULL) {
                 KNI_ERR("error allocating device \"%s\"\n", dev_info.name);
diff -ur dpdk-16.04/lib/librte_mbuf/rte_mbuf.c dpdk-16.04-modified/lib/librte_mbuf/rte_mbuf.c
--- dpdk-16.04/lib/librte_mbuf/rte_mbuf.c        2016-04-11 23:56:34.000000000 +0200
+++ dpdk-16.04-modified/lib/librte_mbuf/rte_mbuf.c        2016-06-28 11:30:51.021771834 +0200
@@ -271,6 +271,7 @@
 const char *rte_get_tx_ol_flag_name(uint64_t mask)
 {
         switch (mask) {
+    case PKT_TX_NO_CRC_CSUM: return "PKT_TX_NO_CRC_CSUM";
         case PKT_TX_VLAN_PKT: return "PKT_TX_VLAN_PKT";
         case PKT_TX_IP_CKSUM: return "PKT_TX_IP_CKSUM";
         case PKT_TX_TCP_CKSUM: return "PKT_TX_TCP_CKSUM";
diff -ur dpdk-16.04/lib/librte_mbuf/rte_mbuf.h dpdk-16.04-modified/lib/librte_mbuf/rte_mbuf.h
--- dpdk-16.04/lib/librte_mbuf/rte_mbuf.h        2016-04-11 23:56:34.000000000 +0200
+++ dpdk-16.04-modified/lib/librte_mbuf/rte_mbuf.h        2016-06-28 11:30:51.022771816 +0200
@@ -101,6 +101,10 @@
 /* add new RX flags here */

 /* add new TX flags here */
+/**
+ * Disable CRC checksum offload
+ */
+#define PKT_TX_NO_CRC_CSUM   (1ULL << 48)

 /**
  * Second VLAN insertion (QinQ) flag.
```