Building a 200G Wire-speed Traffic Generator Using 1U Commodity Server

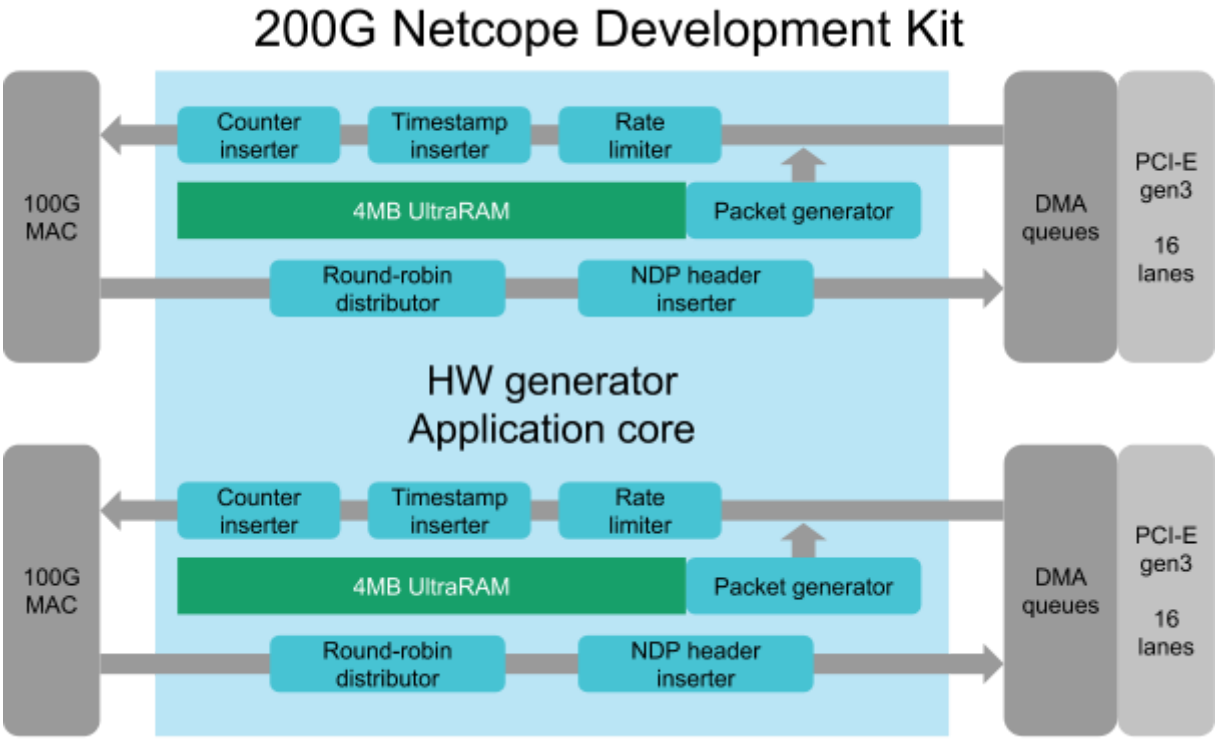
Introduction

The speed of network technologies has grown by hundreds of gigabytes in just a few years. Starting at 10G, we now have 25G, 50G, 100G and 200G, while 400G is already in sight. But as always, this great progress comes at a price. Although CPUs and memories evolve as well, common servers simply cannot provide enough power to do any non-trivial traffic processing at these line-rates. And this is where hardware acceleration bridges the gap.

It may seem that the adoption of hardware acceleration into traffic processing is too difficult or expensive. Let's take a look at a simple use case to see how easy it can be to turn a PC into a 200G traffic generator. Whatever server you have, even with the most powerful CPU inside, there is no way to push much more than 100G through a single PCI-Express gen 3 16-lane interconnection. Although we could use two of these to get 32 PCI-Express gen 3 lanes with up to 256 Gbps of theoretical throughput, we have chosen a different way. Let's use the resources of an FPGA to build a 200G traffic generator requiring no more than zero CPU utilization to run.

Key benefit of this setup from the managerial perspective is a massive reduction in initial purchasing cost compared to hardware traffic generators. High-end commodity server with 2 powerful CPUs and a dual port 100G FPGA network card can be bought for 20 000 USD, while hardware traffic generators cost up to 250 000 USD.

Technology Description



The platform we are going to work with is NFB-200G2QL, the new low-profile network adapter by Netcope Technologies powered by Xilinx Virtex UltraScale+ FPGA. To build an accelerated network application, we are going to use the Netcope Development Kit providing all necessary FPGA peripheries via a simple user interface. Then, the 200G generator will form an application core in the middle of NDK.

The 200G traffic generator is built upon NIC, one of the examples provided with NDK. Like NIC, it uses separate TX and RX paths with 200G full wire-speed throughput and also features a simple round-robin distribution among RX DMA queues. The core functionality is formed by two 100G packet generator modules based on UltraRAM, each with up to 65536 frames of capacity. On top of that, there are components for transmission rate limiting and for insertion of timestamps and counters into frames to byte offsets specified by user.

This approach sends packets that are pre-loaded to UltraRAMs, thus it is independent on CPU preformance and PCI Express throughput. Since NDK supports DPDK API, it can also be easily combined with software-based solutions such as T-Rex. For example, one can imagine a scenario where the (performance-limited) software generates some specific complex traffic pattern, while the FPGA fills the remaining link bandwidth with bulk traffic (eg. simulating a complex attack hidden within large-volume DDoS).

There are three simple configuration tools for the 200G generator. One serves to configure the frames to be generated:

```
generator_configure: HW generator configuration tool

Run 'generator_configure <parameters>' to configure the generator to transmit:
       - PCAP or
       - ICMP of specified size and payload filling

Parameters:
       -h|--help                       Print help
       -d|--device <number>            Device number (default: 0)
       -i|--instance <number>          Instance number (default: 0)

       -t|--interfaces <list>          List of interfaces separated by comma (default:
all)

       -p|--pcap <file>                PCAP file to transmit (tcpdump format, 64-16352
Byte frames)

       -s|--size <value>               Frame size to transmit (64-16352 Bytes, default:
64)
       -S|--size-increment <value>     Frame size increment (default: 0)
       -f|--fill <value>               Frame payload Byte filling (default: random)
       -F|--fill-increment <value>     Frame payload Byte filling increment (default: 0)

By default, 64 bytes long ICMP frames with random payload are generated.
```

The other two serves to start the transmission, control the transmission rate and the offsets for timestamps and counters and to stop the transmission afterwards:

```
generator_run: HW generator configuration tool

Run 'generator_run <parameters>' to run the generator.

Parameters:
       -h|--help                             Print help
       -d|--device <number>                  Device number (default: 0)
       -i|--instance <number>                Instance number (default: 0)

       -s|--speed <value>                    Transmission speed (1-500 = 0.2-100 %,
default: full)
       -c|--counter <offset>                 Insert 16 bit counter to specified offset in
transmitted frames
```

```
       -t|--timestamp <offset>              Insert 64 bit timestamp to specified offset
in transmitted frames
       -l|--loop                            Loop over frames frames configured for
transmission
       -o|--override-sw                     Discard frames sent from SW, don't block
(unsafe if SW is currently transmitting)

       -r|--rx-distribution <queues|NIC>    Distribution of received traffic (Round-robin
or NIC, default: Round-robin over 1 queue)

By default, unmodified configured frames are sent once at full speed while frames sent
from SW are blocked.
```

```
generator_stop: HW generator configuration tool

Run 'generator_stop <parameters>' to stop the generator.

Parameters:
       -h|--help                            Print help
       -d|--device <number>                 Device number (default: 0)
       -i|--instance <number>               Instance number (default: 0)
```

Results

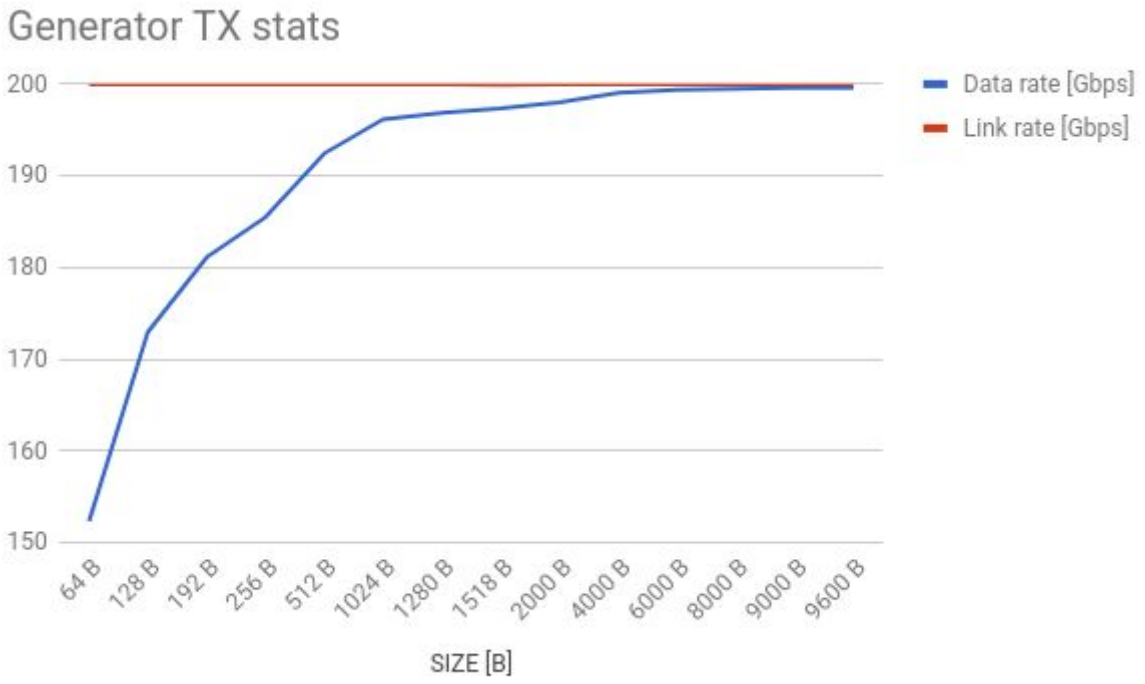


Figure 1: Generator TX stats

The red line represents the link rate at which the generator transmits traffic. The blue line represents the packet data rate at L2 according to the length of the packets generated. The red line shows a steady stream of L1 traffic

at full throughput, irrespective of the blue line. This means that the traffic generator is capable of constant traffic generation of 200 Gbps even on the shortest packets. You can see detailed results in the table below:

SIZE [B]	Size [B]	Frame rate [Mfps]	L2 Data rate [Gbps]	L1 Link rate [Gbps]	Transmitted frames	Error frames
64 B	64	297.619	152.381	200	2980067309	0
128 B	128	168.933	172.987	200	1691021946	0
192 B	192	117.921	181.127	199.995	1180515600	0
256 B	256	90.577	185.502	199.994	906770710	0
512 B	512	46.983	192.444	199.961	470398111	0
1024 B	1024	23.946	196.168	199.999	239702906	0
1280 B	1280	19.227	196.889	199.965	192505664	0
1518 B	1518	16.249	197.338	199.938	162645319	0
2000 B	2000	12.375	198.008	199.988	123891272	0
4000 B	4000	6.22	199.048	200	62258748	0
6000 B	6000	4.153	199.355	200	41573990	0
8000 B	8000	3.116	199.483	199.982	31203583	0
9000 B	9000	2.772	199.584	200	27747793	0
9600 B	9600	2.598	199.573	199.988	26014653	0

Conclusion

At the moment, the 200G traffic generator is more a proof of concept than an application for deployment. However, even at the PoC stage, we are confident to say that it is safe bet to build a traffic generator from commodity server and a single dual port FPGA accelerator. With Netcope Development Kit and Netcope FPGA Board, steady stream of traffic at full throughput is guaranteed. Purchasing cost of such device represents savings of tens of thousands of Dollars compared to hardware traffic generators.

For production purposes, one can add other more complex processing components, either written in VHDL or bought as IP. Since it is widely understood that VHDL development may be too low level for some people, there are other higher level options to define a traffic processing engine. One of them is to describe it using P4 language which can be synthesized into an IP core usable within NDK.