FPGA-Accelerated NAT Project Report

Yongtong Wu
wuyongtong@stu.pku.edu.cn
Peking University
China

Rilin Huang
XXXXXXXX@XXXXXXXX.org
Peking University
China

Jialiang Zhang
XXXXXXXX@XXXXXXXX.org
Peking University
China

1 INTRODUCTION

NAT is a common network function between local-area network (LAN) and wide-area network (WAN). It tracks L4 connections (i.e. TCP and UDP), translates IP address and ports to provide transparent address reuse for different clients in LAN. Implementing NAT in software is convenient, but the massive data traffic (10Gbps or higher) will incur luxurious CPU waste and high packet latency. Therefore, it's common to offload NAT down into hardware such as switches and network interface cards (NIC). However, switches and NICs are usually implemented via application specific integrated circuit (ASIC) which requires a long and expensive cycle from design to production. That makes vendors refuse to put such function into some products with a small audience, such as performant wireless NIC.

Motivation & key ideas. Under the circumstance above, Field Programmable Gate Array (FPGA) comes into consideration naturally. It allows users to implement custom hardware logic including NAT. Therefore, in this project, we explore the possibility of implementing NAT in FPGA, and show the performance gain from hardware-oriented design.

Prior works. We do a survey about prior works including other hardware offloading approach and advance NAT designs. Programmable switches with P4 language support line-rate match-action processing by which NAT can be implemented naturally. [1] Some high-performance NICs such as Nvidia (Mellanox) ConnectX-4/5/6 can support NAT offloading itself with proper configuration. However, they are all wired NIC for data center network. There are also already some trials to implement NAT on FPGA such as NetFPGA platform. More detail can be found at the footnote link. [2]

Evaluation Results. In our evaluation, FPGA-based NAT shows a 10% speed-up in throughput than software-based NAT, and better scalability over connection numbers.

1.1 Contribution Summary

- Exploration and System Design: Yongtong Wu
- Hardware Impl. and Testing: Rilin Huang, Yongtong Wu
- Software and Testbed Configuration: Jialiang Zhang
- Evaluation & Live Demo Video: Yongtong Wu
- Poster: Jialiang Zhang
- Report Writing: Rilin Huang, Yongtong Wu

2 SYSTEM DESIGN

Hello Here is our intro

[1] https://github.com/p4lang/switch/blob/master/p4src/nat.p4
[2] https://www.cl.cam.ac.uk/ osc22/docs/edv10_nat_fpga.pdf

Computer Network (Honor Track), 2023 Fall, Peking University
2024.

3 IMPLEMENTATION

We show our implementation in this section, including testbed setting, the whole workflow and implementation result such as code lines and FPGA resource consumption. We implement our design based on Xilinx official example of Ethernet 10/25G system.

3.1 Testbed Configuration

We implement our NAT with AMD (Xilinx) Zynq UltraScale+ MPSoC ZCU102 (called the board later). The evaluation setup includes the board, a router and a desktop machine. We run a Linux Version on the four PS ARM cores of the board. The desktop runs Ubuntu 18.04 equipped with a Nvidia (Mellanox) ConnectX-3 with driver version ?? whose line rate is 10Gbps. The MTU is configured as 9000B. The performance of router is not critical since we only use it to ssh connect to the board.

Network Topology. The board and the on-board NIC of desktop are both connected to the router and are configured under the same subnet. For data plane, we connect the SPF interface of the board and a interface of ConnectX-3 back-to-back to fully investigate the potential of our FPGA NAT.

3.2 Implementation Workflow

We elaborate our detailed implementation workflow, which includes Verilog design and testing, Vivado block design, Petalinux image building, testbed deployment and Vitis cross-compile in this subsection. We use Vivado 19.02, Vitis 19.02, and Petalinux 19.02. Figure 1 shows our workflow.

Hardware implementation and testing. We use Verilog to implement our design in about 300 lines of code. We use Verilator to test the correctness of our implementation. The testing code is about 1.5k lines in C++.

Vivado block design and synthesis. We use Vivado to insert our Verilog implementation as custom IPs, and then synthesis, implement, and generate bitstream. Finally we export the hardware to an `.xsa` file. The delta of resource consumption between the original example and our modified design is ????????. Note that we use a custom Ethernet protocol number rather than `0x0800` to prevent potential complexities such as NIC checksum-offloading and Linux kernel sanity check. Therefore, we do not need to insert CSU IP.

Petalinux image building. Petalinux is a tool to build tiny embedded linux images. The built image can be placed into an SD card to boot up the whole board. We build the images following the official tutorial in a Ubuntu 18.04 docker container over AMD Ryzen 7 7700 (4.8Ghz full core). It takes up to several hours.

Vitis cross-compilation. We need to cross-compile the C++ source code since the CPUs on the board are ARM core. We do it via Vitis, and scp the generated ELF file to the board directly.

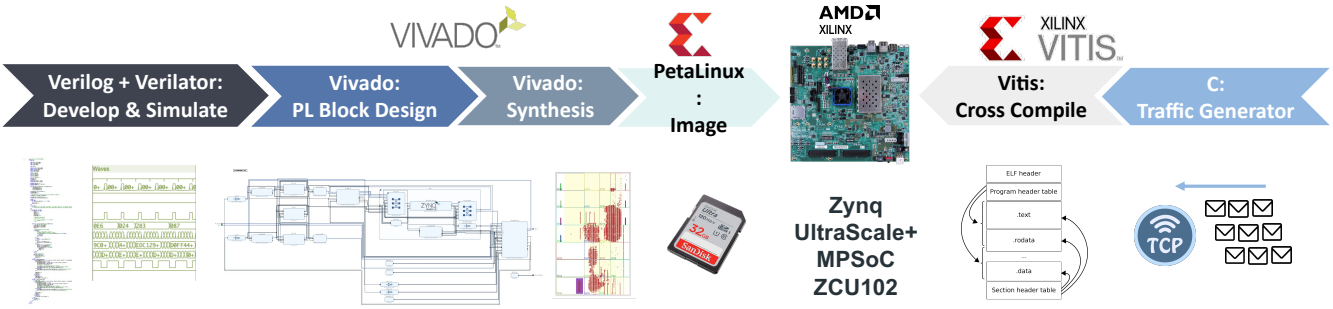

Figure 1: Our implementation workflow.

4 EVALUATION

Hello Here is our intro

5 DISCUSSION

In this section, we will discuss some possible improvement in our system design and evaluation.

Better evaluation method: bypassing PS. Our evaluation result shows that the weak performance of ZCU102 hardware prevents us from fully investigate the potential of our hardware design. Thus, bypassing the PS part of the board may be a better choice to get precise evaluation result. To be specific, the received Ethernet packets should being send back to another SFP interface after they are processed by our custom IPs in the PL logic, instead of passing them to those weak ARM CPUs.

More efficient NAT design. The NAT we implemented use little resource at a cost of wasting more cycles. In fact, we can utilize the hardware parallelism by introducing structure like full-associated cache in our NAT table which promise one cycle latency.

Lack of connection state management. A functional NAT should track connection states to release the NAT table entries in time. Due to time limit, we just discuss and come up with a possible design and did not implement this part.

6 CONCLUSION

In this project, we explore the method to design and implement an FPGA-based NAT. Our evaluation shows about 10% throughput gain and better scalability compared with software-implemented NAT.