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 addresses 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 circumstances above, the 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 survey prior works including other hardware offloading approaches and advanced 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 NICs for Data Center Network. There are also already some trials to implement NAT on FPGA such as the NetFPGA platform. More details 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.

Table 1: ZCU102 Resource Consumption

	LUT	FF	BRAM	DSP
Origin	18923	18315	139.5	0
Ours (table size 1024)	33373	47067	142.0	0
Ours /Total Resource (%)	12.18	8.59	15.57	0

3 IMPLEMENTATION

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

3.1 Testbed Configuration

We implemented 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, whose line rate reaches 10Gbps, with `mlx4` driver version `4.14.142-ubwins+`. The MTU is configured as 9000B to get full performance. The performance of the 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 the desktop are both connected to the router and are configured under the same subnet. For the data plane, we connect the SPF interface of the board and an interface of ConnectX-3 back-to-back to fully investigate the potential of our FPGA NAT.

3.2 Implementation Workflow

We elaborate on 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 **??** shows our workflow.

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

Vivado block design and synthesis. We use Vivado to insert our Verilog implementation as custom IPs, and then synthesize, implement, and generate bitstream. Finally, we export the hardware to an `.xsa` file. The resource consumption is shown in Table 1. The whole design (with table size 1024, original 10/25G Ethernet Subsystem included) consumes 12.18% LUTs, 0.58% LUTRAMs, 8.59% FFs, 15.57% BRAMs. The delta of resource consumption between the original example and our modified design is 14464 LUTs, 13252

FF, 2.5 BRAM, and 0 DSP. 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 (4.8 Ghz 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 cores. Vitis provides cross-compile support for different Zynq platforms. The generated ELF file can be sent to the board via `scp` directly.

4 EVALUATION

Hello Here is our intro

5 DISCUSSION

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

Better evaluation method: bypassing PS. Our evaluation result shows that the weak performance of ZCU102 PS hardware prevents it from fully investigating the potential of our hardware design. Thus, bypassing the PS part of the board may be a better choice to get precise evaluation results. To be specific, the received Ethernet packets should be sent 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 uses little resources at the cost of wasting more cycles. We can utilize the hardware parallelism by introducing structures like a full-associated cache in our NAT table which promises 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 the project time limit, we just discussed and came up with a possible design but 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.

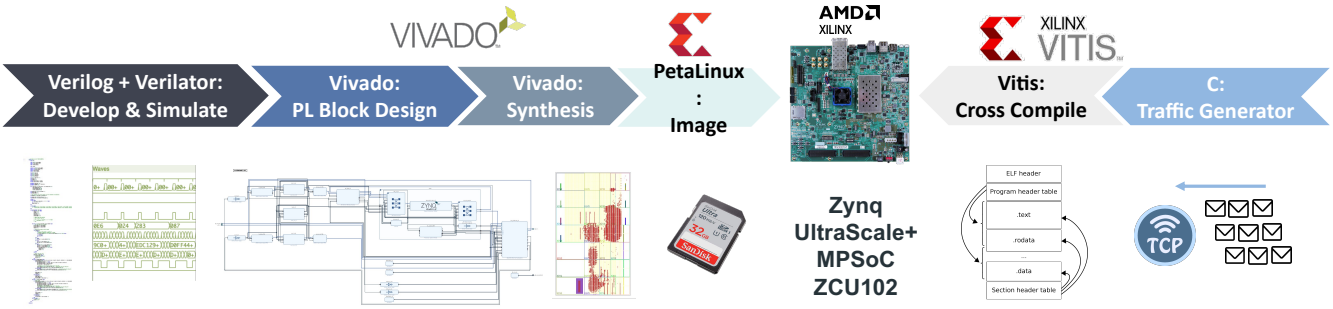

Figure 1: Our implementation workflow.