FPGA-Accelerated NAT Project Report

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

Rilin Huang
2100013095@stu.pku.edu.cn
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

This chapter presents our system design, including the original block design **pl_eth_10g**, and the two custom IP cores that we add.

2.1 Original block design

2.1.1 *Components.*

The design integrates the processing power of the PS, the flexibility of programmable logic in the PL, and specific components like DMA, MAC, and GTH to implement a 10GbE interface on the ZCU102 evaluation board.

PS (Processing System): The processing system in ZCU102 includes ARM Cortex-A53 processors, which are powerful 64-bit processors. These processors handle general-purpose computing tasks and provide the necessary control for the entire system.

PL (Programmable Logic): The programmable logic section is responsible for the custom logic implementation. In this project, it is utilized for creating the 10GbE interface using programmable fabric resources.

- DMA (Direct Memory Access): DMA is used for efficient data transfer between memory and peripherals without involving the processor. It enhances the throughput of data, and offloads the tasks of data transfer from processor.
- MAC (Media Access Control): The MAC layer handles the communication protocol for the Ethernet interface. It manages frame formatting, addressing, error detection, and other protocol-related tasks.
- GTH (Gigabit Transceivers): GTH transceivers are used for high-speed serial data communication. They facilitate the transmission and reception of data at gigabit rates.

2.1.2 *Data flow.*

MAC and GTH are responsible for the transmission of data between the FPGA and the external network. Upon receiving incoming data from MAC and GTH, the DMA takes control of the data transfer process, and moves the data to the designated location in Memory. This stored data can be further processed by the processor or other components of the system.

2.2 NAT IP

Based on the original block design, we introduce a NAT(Network Address Translation) IP to intercept the communication between the MAC/GTH and DMA for our goal. Our NAT IP modifies the IP and TCP headers of the packets according to the NAT rules, and

[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.

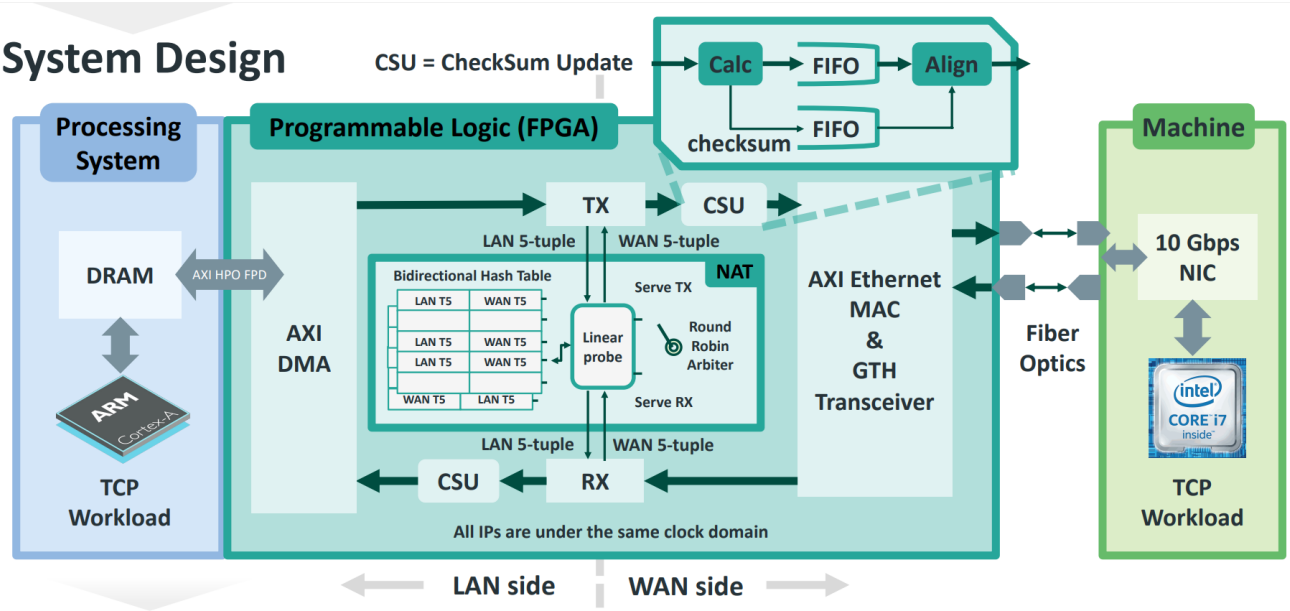

Figure 1: System design

maintains a bidirectional hash table to store the mappings between LAN 5 tuple and WAN 5 tuple.

2.2.1 *Hash Function.*

The NAT IP uses XOR operation on the 5-tuple *(source IP, source port, destination IP, destination port, protocol)* as its hash function. This approach provides a quick and effective way to generate hash values for mapping.

2.2.2 *Linear Probing.*

To handle conflicts within the hash table, the NAT IP utilizes linear probing as a collision resolution strategy. In case of a hash collision, the algorithm linearly searches for the next available slot in the hash table until an empty slot is found.

2.2.3 *Biderectional Hash Table.*

Our NAT IP incorporates two hash tables. One is for the mapping from LAN 5-tuple to WAN 5-tuple, and the other maintains the reverse mapping.

For packets from LAN to WAN, the NAT IP first computes the hash value of the LAN 5-tuple *(source IP, source port, destination IP, destination port, protocol)*, and then searches for a matching entry in the hash table. If no entry is found, the NAT IP allocates a new public port from a predefined range, and creates a new entry in both hash tables with the LAN 5-tuple *(source IP, source port, destination IP, destination port, protocol)* and the WAN 5-tuple *(public IP, public port, destination IP, destination port, protocol)*, then modifies the packet header accordingly. If a matching entry is found, the NAT IP simply replaces *(source IP, source port)* of the packet with *(public IP, public port)* in the corresponding WAN 5-tuple.

For packets from WAN to LAN, the map from WAN 5-tuple to LAN 5-tuple should exist in the normal case. So the NAT IP can modify the packet header accordingly.

2.2.4 *Round Robin Arbiter.*

Round Robin Arbiter is a type of arbiter that try to serve each requester in a circular order, ensuring that each requester can obtain resources fairly. To avoid conflicts caused by TX and RX accessing the bidirectional hash table at the same time, we use a Round Robin Arbiter in the NAT IP to control the access to the hash table.

We use two signals, *ready_tx* and *ready_rx*, to decide which requester should be served. One ready signal is unset only in the next cycle after the end of service for the corresponding requester. In this way, a waiting requester can get served immediately after the end of service for the other. So Round Robin Arbiter achieves fair access control to the bidirectional hash table, avoiding conflicts between TX and RX.

2.3 CSU IP

The CSU(Checksum Update) IP is a custom IP core that recalculates the TCP checksum of the packets after the NAT IP modifies the IP header. It consists of four modules, Calc, FIFO1, FIFO2, and Align.

The Calc module computes the new checksum value based on the modified packet, and outputs the new checksum value and the original packet to FIFO1 and FIFO2 respectively.

The FIFO1 and FIFO2 modules are two FIFO buffers that store the new checksum value and the original packet respectively. They are used to synchronize the data flow between the Calc module and the Align module.

The Align module is responsible for writing the new checksum value into the TCP header of the packet. It reads the data from FIFO1 and FIFO2, and locates the TCP checksum field to replace the old checksum value with the new. m

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.

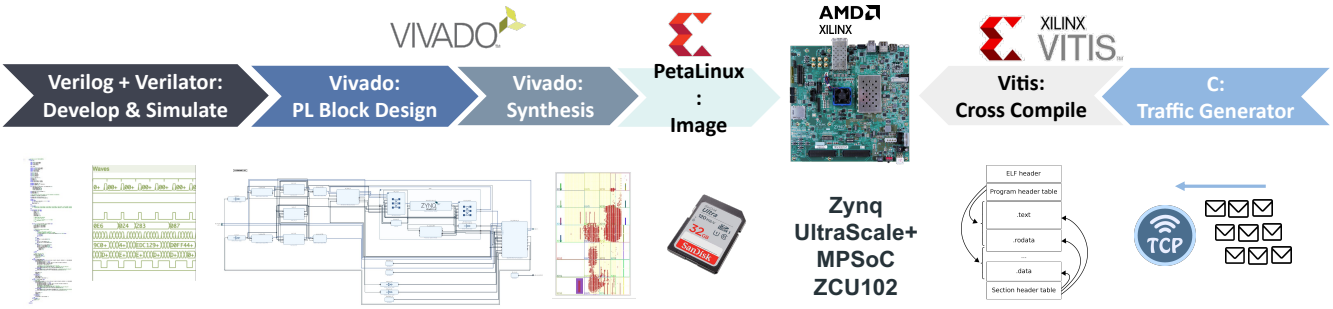

Figure 2: Our implementation workflow.