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
2100012902@stu.pku.edu.cn
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 the 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 up to 7.7% performance gain in throughput than software-based NAT and better scalability over connection numbers (i.e. the load factor).

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

We show our system design in this section. Our design is based on the official example of the Ethernet 10/25G system of Xilinx (referred to as the original design later). [3]

2.1 Original Block Design

Before we elaborate on our design, it is necessary to show how is the original design organized as a background.

Components. The design integrates PS and PL. The processing system (PS) in ZCU102 includes four ARM Cortex-A53 processors. These processors can handle general-purpose computing tasks and provide the necessary control for the entire system. The programmable logic (PL) is responsible for the custom hardware logic implementation. Components like DMA, MAC, and GTH that are implemented in PL form a 10GbE network interface on the ZCU102 evaluation board. For PS, there is a Xilinx handmade Linux driver to drive this interface. We further detail the function of those components implemented in PL here:

- 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 the 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.

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 it into the designated memory location. This stored data can be further processed by the processor or other components of the system.

AXI4-Stream Signals. Most signals between the modules above adopt the AXI4-Stream protocol. AXI4-Stream interfaces appear in pairs: the slave port (start with `s_` below) and the master port (start with `m_` below). The most important handshake mechanism is the `ready` signal of the slave and the `valid` signal of the master. Once these two signals are both active, there is a data transfer. To provide more information, we show the exact definitions of them here.

[1] https://github.com/p4lang/switch/blob/master/p4src/nat.p4
[2] https://www.cl.cam.ac.uk/ osc22/docs/edv10_nat_fpga.pdf
[3] https://github.com/Xilinx-Wiki-Projects/ZCU102-Ethernet/tree/main/2019.2/pl_eth_10g

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

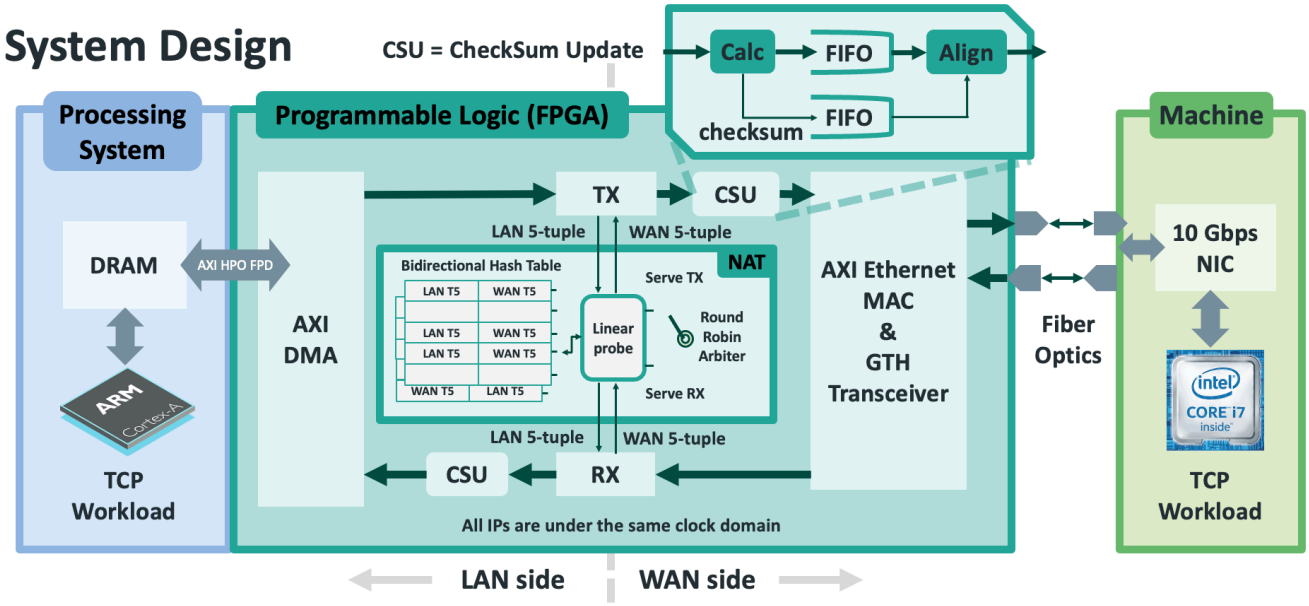

Figure 1: System design

- `s_axis_tdata[63:0]`: Data bus received from the master device, conveying data across the interface.
- `s_axis_tkeep[7:0]`: Byte qualifier received from the master device, indicating whether corresponding bytes of the data bus are part of the data stream. A low level indicates an invalid byte.
- `s_axis_tvalid`: Signal received from the master device, indicating a valid data transfer.
- `s_axis_tlast`: Signal received from the master device, marking the last data transfer and indicating the boundary of the data stream.
- `s_axis_tready`: Signal sent to the master device, indicating its readiness to accept a data transfer in the current clock cycle.
- `m_axis_tdata[63:0]`: Data bus sent to the slave device, providing data across the interface.
- `m_axis_tkeep[7:0]`: Byte qualifier sent to the slave device, indicating whether corresponding bytes of `m_axis_tdata` are part of the data stream. A low level indicates an invalid byte.
- `m_axis_tvalid`: Signal sent to the slave device, indicating the reception of a valid data transfer.
- `m_axis_tlast`: Signal sent to the slave device, marking the last data transfer and indicating the boundary of the data stream.
- `m_axis_tready`: Signal received from the slave device, indicating its readiness to send a data transfer in the current clock cycle.

2.2 TX and RX

Based on the original block design, we introduce TX and RX to intercept the communication between the MAC/GTH and DMA. TX intercepts the communication from DMA to MAC/GTH, and RX intercepts the communication from MAC/GTH to DMA. Both TX and RX check whether the incoming packet is a TCP packet or not.

If the packet is not a TCP packet, TX and RX will simply forward the packet out. Otherwise (i.e. the packet is a TCP packet), both TX and RX will extract the 5-tuple information from the packet, send it to NAT IP, and wait for a 5-tuple response from NAT IP. Upon receiving the response, TX and RX will modify the header of the packet according to the 5-tuple received, and then forward the packet out.

2.3 NAT IP

We introduce a NAT (Network Address Translation) IP to help TX and RX modify the IP and TCP headers of the packets according to the NAT rules. Our NAT IP maintains a bidirectional hash table to store the mappings between LAN 5-tuple and WAN 5-tuple. It communicates with TX and RX through a simple interface.

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.

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.

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. Therefore, the NAT IP can modify the packet header accordingly.

Round Robin Arbiter. It is a type of arbiter that tries 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. Therefore, Round Robin Arbiter achieves fair access control to the bidirectional hash table and avoids synchronization issues between TX and RX.

2.4 CSU IP

The CSU (Checksum Update) IP is a custom IP core that recalculates the IP and TCP checksums of the packets after the NAT IP modifies the 5-tuples. It is necessary since there are sanity checks in hardware and software to validate those checksums. If mismatched, the packet will be dropped silently.

The CSU IP consists of four sub-IPs: 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 buffer the data, waiting for the Align module to fetch. 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 one.

2.5 Put It All Together

All the components of our system design work together to achieve the functionality of a hardware-based NAT system in Figure 1. Their clocks are all synchronized with *tx_clk_out* of MAC/GTH (156Mhz, achieving 10Gbps with a 64-bit data bus).

An incoming TCP packet gets processed by the following workflow: ① Firstly, RX extracts the IP and TCP headers from the *s_axis_tdata* signal, including the source IP, source port, destination IP, destination port, and protocol, to form a 5-tuple. ② Then, RX sends this 5-tuple to the NAT IP core, through a simplified interface only with a 5-tuple and a valid signal. ③ After sending the 5-tuple, RX blocks its input by setting the *s_axis_tdata* signal to 0, indicating that it cannot accept new data. Meanwhile, it waits for the response from the NAT IP core. ④ When the NAT IP core returns a new 5-tuple to RX, RX modifies the header of the packet accordingly and continues receiving data normally. ⑤ All packets from RX are sent to CSU to update the TCP checksum, and finally get into DMA.

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. The board runs Linux 4.19.0 on its four PS ARM cores. The desktop runs Ubuntu 18.04 equipped with Intel i7-9700K and an NVIDIA ConnectX-3 (Mellanox) network interface card, 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's 1Gbps interface and the onboard NIC of the desktop are both connected to the router and are configured under the same subnet. For the data plane, we connect the 10Gbps 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 2 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 key part of our board design is shown in Figure 3 and the netlist is shown in Figure 4. 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

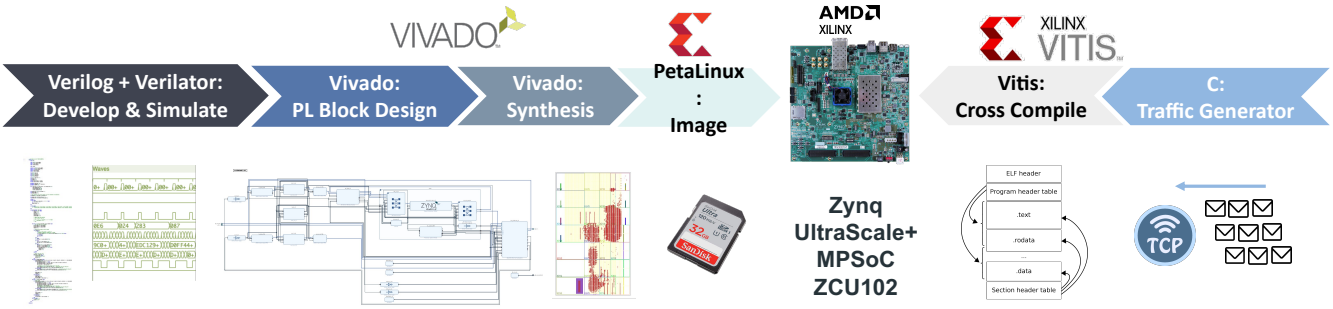

Figure 2: Our implementation workflow.

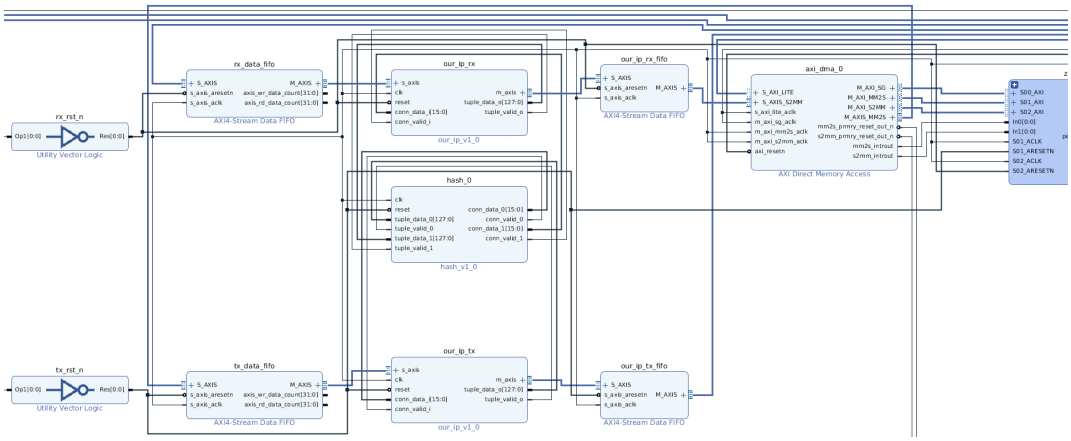

Figure 3: The key part of our board design.

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

We show the evaluation of our NAT system in this section. SW and HW NAT implementation are considered (referred to as SW NAT and HW NAT). Two ablation settings (HW NAT without translation, without NAT) are introduced to measure the overhead of our HW NAT design. With four settings above, we do two evaluations:

- We measure the throughput with different packet sizes from 0 to 5000 bytes.
- We measure the throughput with different numbers of connections from 0 to 1024.

4.1 Evaluation Setting

We detail our settings in two evaluations here. The connection hash table size is configured as 1024 for both evaluations. A traffic generator written in C language continuously sends TCP packets to the desktop machine via a Linux packet socket to the 10GbE interface of the board, which will flow through PL, in an infinite loop. The throughput is measured by dividing the data amount over the time of sending it. The data amount is set to 100 gigabits. The details of the four settings are:

- **HW NAT**: Our design stated in System Design and Implementation sections. It is a hardware-based NAT system that uses our custom NAT IP core, which can intercept the communication between MAC/GTH and DMA, and modify the IP and TCP headers of the packets according to the NAT rules. It uses a bidirectional hash table to store the mappings of LAN 5 tuple and WAN 5 tuple, and uses XOR as the hash function, and linear probing as the collision resolution strategy. It can achieve high-speed data conversion and forwarding, and improve the throughput.

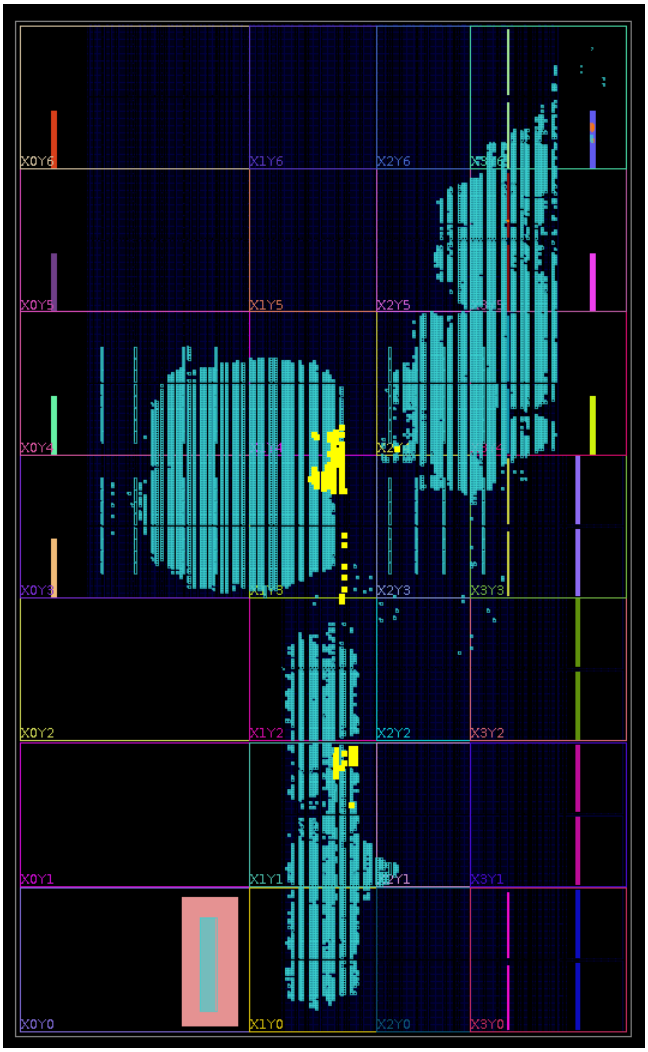

Figure 4: The post-synthesis netlist with NAT part highlighted with yellow.

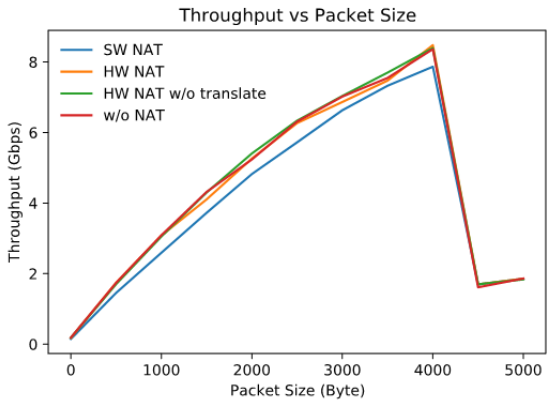

Figure 5: Thput. v.s. pkt size

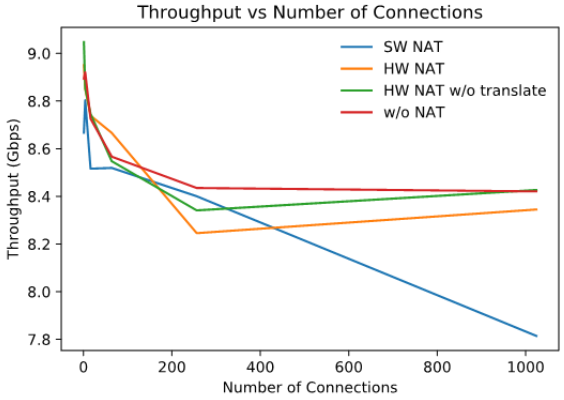

Figure 6: Thput v.s. Connections #

- **SW NAT**: A software-based NAT system that runs on four threads and uses a shared hash table with linear probing. The table size is also 1024. It can be considered as a baseline compared to our HW NAT.
- **HW NAT w/o translate**: The same as **HW NAT**, but we use packets that do not need to translate (e.g. ICMP packets) to measure the performance.
- **w/o NAT**: No NAT at all, packets are directly forwarded between MAC/GTH and DMA. This is the ideal scenario and can be used as an upper bound to evaluate the performance of our system.

Latency. We do not measure the latency since the overhead of HW NAT is predictable, averaging only constant cycles when the load factor is much lower than the table size. Even in extreme situations where the load factor is 1, the maximum latency is only $tableSize \times cycleTime$, which is less than 10 μs when the table size is 1024. It only accounts for about 10% of the normal ping-pong latency (100-200 μs, measured with `ping` command).

4.2 Result

Figure 5 shows the relationship between throughput and packet size. As the packet size increases, the throughput also increases, until it reaches a maximum value at around 4000 bytes, and then suddenly drops. This phenomenon is observed in all four settings.

Figure 6 shows the relationship between throughput and the number of connections. As the number of connections increases, the throughput of SW NAT drops sharply, while the throughput of other scenarios remains stable.

4.3 Interpretation

We draw three main conclusions from our results:

- **The performance of HW NAT is up to 7.7% higher than SW NAT under our setting.** In Figure 5, the maximum throughput of HW NAT is 8.48 Gbps, and the maximum throughput of SW NAT is 7.87 Gbps.

- **Figure 5 shows that SW NAT is a bottleneck and HW NAT is not.** In Figure 5, we can see that SW NAT has a performance gap with no NAT, indicating that SW NAT is a bottleneck that degrades the overall system performance. We can also see that HW NAT has a performance improvement compared to SW NAT, indicating that hardware implementation has better performance than software. And there's almost no performance gap between HW NAT and no NAT, which indicates that hardware implementation has little impact on the original system performance.
- **Figure 6 shows that SW NAT is unscalable and HW NAT is scalable.** SW NAT does not scale well with the number of connections, due to the CPU overhead of software NAT and the synchronization overhead of sharing the NAT table. On the other hand, HW NAT scales well with the number of connections, as it uses a bidirectional hash table and the arbiter design to handle the NAT mappings.

4.4 Anomalies

There are two anomalies in our evaluation results. We try to give some speculative guesses about them.

Why does the packet size need to be large to achieve line rate? We speculate that this is due to the poor performance of the ARM cores and the not-so-good performance of DMA, MAC, and GTH IP. The FPGA side has to process the packets at a high speed, and may not be able to handle the small packets efficiently. Therefore, the packet size needs to be large enough to saturate the bandwidth of the FPGA side.

Why is there a cliff-like drop after 4000 bytes? We speculate that this is due to the existence of a 4K buffer in the link, which may cause extra overflow handling for large packets. When the packet size exceeds 4K, the buffer may not be able to store the whole packet and may have to split it into multiple segments, which may incur additional overhead and reduce the throughput.

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 performant and resource-efficient NAT design. Linear probing requires fewer 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. In addition, we can implement a RAM matrix to further reduce interconnect consumption.

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 7.7% throughput gain and better scalability compared with software-implemented NAT.

7 ACKNOWLEDGEMENT

We want to appreciate Prof. Chenren Xu, TAs, and some seniors for offering logistics and kind help.