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

Network Address Translation (NAT) serves as a crucial network function bridging the gap between local-area networks (LANs) and wide-area networks (WANs). Its primary function involves tracking Layer 4 connections, specifically for protocols like TCP and UDP. NAT achieves transparent address reuse for various clients within a LAN by translating IP addresses and ports. While software-based NAT implementation is convenient, handling massive data traffic, such as at speeds exceeding 10Gbps, can lead to significant CPU resource wastage and high packet latency.

To address these challenges, the common approach is to offload NAT functionality into dedicated hardware, such as switches and network interface cards (NICs). However, the implementation of switches and NICs typically relies on application-specific integrated circuits (ASICs), resulting in a lengthy and expensive cycle from design to production. Consequently, vendors may refrain from incorporating NAT functionality into certain products with a limited audience, such as high-performance wireless NICs.

Motivation & Key Ideas. Given these circumstances, the Field Programmable Gate Array (FPGA) emerges as a natural consideration. FPGA enables users to implement custom hardware logic, including NAT. In this project, we aim to explore the feasibility of implementing NAT in an FPGA, showcasing the performance gains achievable through hardware-oriented design.

Prior Works. Our survey of prior works includes examining alternative hardware offloading approaches and advanced NAT designs. Programmable switches, with P4 language support for line-rate match-action processing, provide a natural avenue for implementing NAT. Examples can be found in the P4 language code for NAT at the footnote link[1]. Additionally, some high-performance NICs, such as Nvidia (Mellanox) ConnectX-4/5/6, can support NAT offloading with proper configuration; however, these are primarily designed for wired NICs in data center networks. Previous attempts to implement NAT on FPGAs, such as the NetFPGA platform, are also documented, and more details can be found in the footnote link[2].

Evaluation Results. In our evaluation, FPGA-based NAT demonstrates a performance gain of up to 7.7% in throughput compared to software-based NAT. Additionally, it exhibits better scalability concerning connection numbers, emphasizing its efficacy under varying load factors.

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

In this section, we present our system design, which is based on the official example of the Ethernet 10/25G system of Xilinx (referred to as the original design hereafter) [3].

2.1 Original Block Design

Before delving into our design, it is essential to provide an overview of the organization of the original design for contextual understanding.

Components. The design integrates the Processing System (PS) and Programmable Logic (PL). The PS in ZCU102 includes four ARM Cortex-A53 processors, capable of handling general-purpose computing tasks and providing control for the entire system. The PL is responsible for implementing custom hardware logic, including components such as DMA, MAC, and GTH, which collectively form a 10GbE network interface on the ZCU102 evaluation board. A Xilinx handmade Linux driver is employed to drive this interface on the PS.

We detail the functions of the components implemented in PL as follows:

- **DMA (Direct Memory Access):** Efficiently transfers data between memory and peripherals without involving the processor, enhancing data throughput and offloading data transfer tasks from the processor.
- **MAC (Media Access Control):** Manages the communication protocol for the Ethernet interface, handling frame formatting, addressing, error detection, and other protocol-related tasks.
- **GTH (Gigabit Transceivers):** Facilitates high-speed serial data communication, enabling the transmission and reception of data at gigabit rates.

Data Flow. MAC and GTH handle the transmission of data between the FPGA and the external network. Upon receiving incoming data from MAC and GTH, DMA takes control of the data transfer process and moves it into the designated memory location, allowing further processing by the processor or other system components.

AXI4-Stream Signals. Most signals between the modules adopt the AXI4-Stream protocol, with interfaces appearing in pairs - the slave port (starting with `s_`) and the master port (starting with `m_`). The `ready` signal of the slave and the `valid` signal of the master

[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.

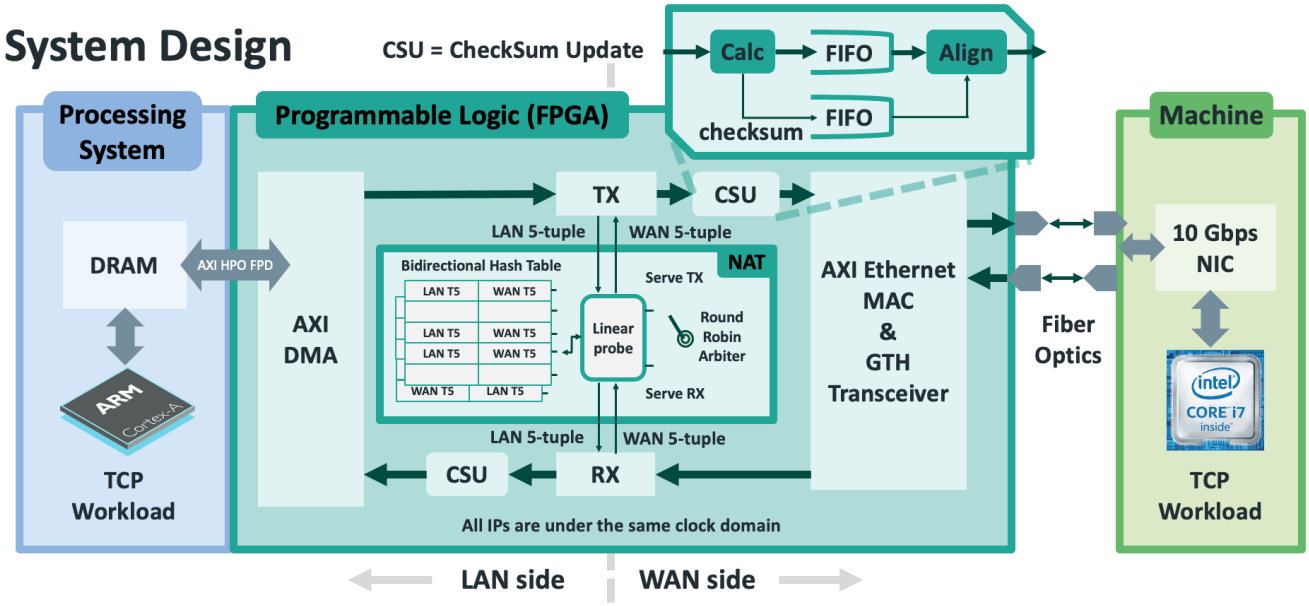

Figure 1: System Design

constitute the primary handshake mechanism, indicating a data transfer when both signals are active.

The exact definitions of these AXI4-Stream signals are as follows:

- `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

Building upon the original block design, we introduce TX and RX to intercept communication between the MAC/GTH and DMA. TX intercepts communication from DMA to MAC/GTH, while RX intercepts communication from MAC/GTH to DMA. Both TX and RX assess whether an incoming packet is a TCP packet.

If the packet is not a TCP packet, TX and RX simply forward the packet. However, if the packet is a TCP packet, TX and RX extract the 5-tuple information, send it to the NAT IP, wait for a 5-tuple response from NAT IP, and upon receiving the response, modify the packet header according to the received 5-tuple before forwarding the packet.

2.3 NAT IP

We introduce a Network Address Translation (NAT) IP to assist TX and RX in modifying the IP and TCP headers of packets based on NAT rules. The NAT IP maintains a bidirectional hash table to store mappings between LAN 5-tuple and WAN 5-tuple. Communication with TX and RX occurs through a simple interface.

Hash Function: The NAT IP employs XOR operation on the 5-tuple *(source IP, source port, destination IP, destination port, protocol)* as its hash function, providing a quick and effective method for generating hash values.

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.

Bidirectional Hash Table: Our NAT IP incorporates two hash tables - one for the mapping from LAN 5-tuple to WAN 5-tuple and the other for the reverse mapping.

For packets from LAN to WAN, the NAT IP computes the hash value of the LAN 5-tuple, searches for a matching entry in the hash

table, allocates a new public port if no entry is found, and modifies the packet header accordingly. If a matching entry is found, the NAT IP 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 NAT IP can modify the packet header based on the mapping from WAN 5-tuple to LAN 5-tuple.

Round Robin Arbiter: To avoid conflicts arising from concurrent access to the bidirectional hash table by TX and RX, a Round Robin Arbiter is employed in the NAT IP to control access to the hash table. Two signals, $ready\_tx$ and $ready\_rx$, determine which requester should be served. Each ready signal is unset only in the next cycle after the end of service for the corresponding requester, ensuring fair access control to the bidirectional hash table and mitigating synchronization issues between TX and RX.

2.4 CSU IP

The CSU (Checksum Update) IP is a custom core that recalculates the IP and TCP checksums of packets after the NAT IP modifies the 5-tuples. This recalibration is essential as sanity checks in hardware and software validate these checksums. Mismatched checksums result in the silent dropping of the packet.

The CSU IP comprises 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. FIFO1 and FIFO2 serve as two FIFO buffers that store the new checksum value and the original packet, awaiting processing by the Align module. The Align module writes the new checksum value into the TCP header of the packet, extracting data from FIFO1 and FIFO2 to locate the TCP checksum field.

2.5 Put It All Together

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

Incoming TCP packets undergo the following workflow: (1) RX extracts the IP and TCP headers from the *s_axis_tdata* signal, forming a 5-tuple. (2) RX sends this 5-tuple to the NAT IP core via a simplified interface 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 its inability to accept new data. Simultaneously, RX awaits a response from the NAT IP core. (4) Upon receiving a new 5-tuple from the NAT IP core, RX modifies the packet header accordingly and resumes normal data reception. (5) All packets from RX are then forwarded to CSU for TCP checksum updating before being processed by DMA.

3 IMPLEMENTATION

In this section, we present our implementation, covering the testbed configuration, the entire workflow, and implementation results such as code lines and FPGA resource consumption. Our design is implemented based on Xilinx's official example of the Ethernet 10/25G system.

3.1 Testbed Configuration

We implemented our NAT system using the AMD (Xilinx) Zynq UltraScale+ MPSoC ZCU102 board (referred to as the board). The evaluation setup consists of the board, a router, and a desktop machine.

The board runs Linux kernel version 4.19.0 on its four ARM Cortex-A53 processors. The desktop runs Ubuntu 18.04 with an Intel i7-9700K processor and an NVIDIA ConnectX-3 (Mellanox) network interface card, capable of achieving a line rate of 10Gbps using the `mlx4` driver version 4.14.142-ubwins+. The MTU is configured as 9000 bytes to ensure optimal performance.

The performance of the router is not critical, as it is only used for SSH connections to the board.

Network Topology. The board's 1Gbps interface and the onboard NIC of the desktop are both connected to the router and 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 explore the potential of our FPGA-based NAT.

3.2 Implementation Workflow

We elaborate on our detailed implementation workflow, including Verilog design and testing, Vivado block design, Petalinux image building, testbed deployment, and Vitis cross-compilation in this subsection. We used Vivado 19.02, Vitis 19.02, and Petalinux 19.02. Figure 2 illustrates our workflow.

Hardware Implementation and Testing. We utilized Verilog and Verilator to implement and test our design, consisting of approximately 300 lines of Verilog code and 1.5k lines of C++ code.

Table 1: ZCU102 Resource Consumption

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

Vivado Block Design and Synthesis. We used Vivado to insert our Verilog implementation as custom IPs, and then synthesized, implemented, and generated a bitstream. Finally, we exported the hardware to an `.xsa` file. The key part of our board design is shown in Figure 3, and the netlist is displayed in Figure 4. Resource consumption details are presented in Table 1. The entire design, including a table size of 1024 and the original 10/25G Ethernet Subsystem, consumes 12.18% LUTs, 0.58% LUTRAMs, 8.59% FFs, and 15.57% BRAMs.

Petalinux Image Building. Petalinux, a tool to build tiny embedded Linux images, was employed to build the images following the official tutorial in a Ubuntu 18.04 Docker container running on an AMD Ryzen 7 7700 (4.8 GHz full core). The image building process took several hours.

Vitis Cross-Compilation. To execute the C++ source code on the board's ARM cores, cross-compilation was necessary. Vitis provides cross-compilation support for various Zynq platforms, resulting in an ELF file that could be sent to the board via `scp`.

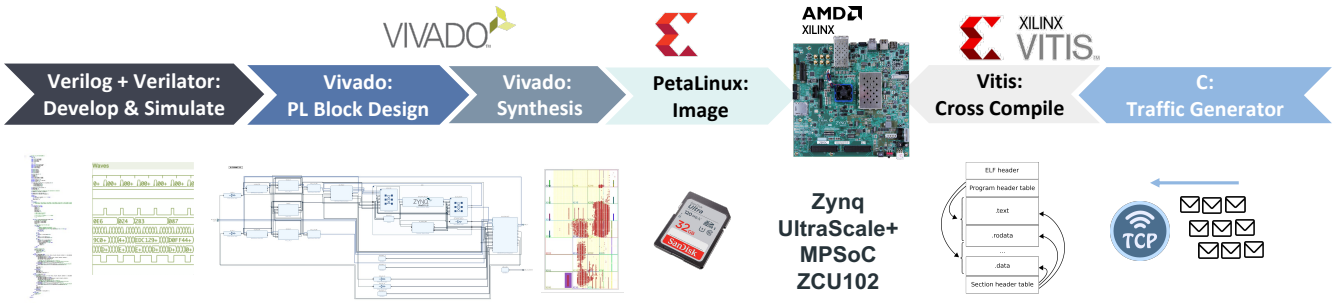

Figure 2: Our implementation workflow.

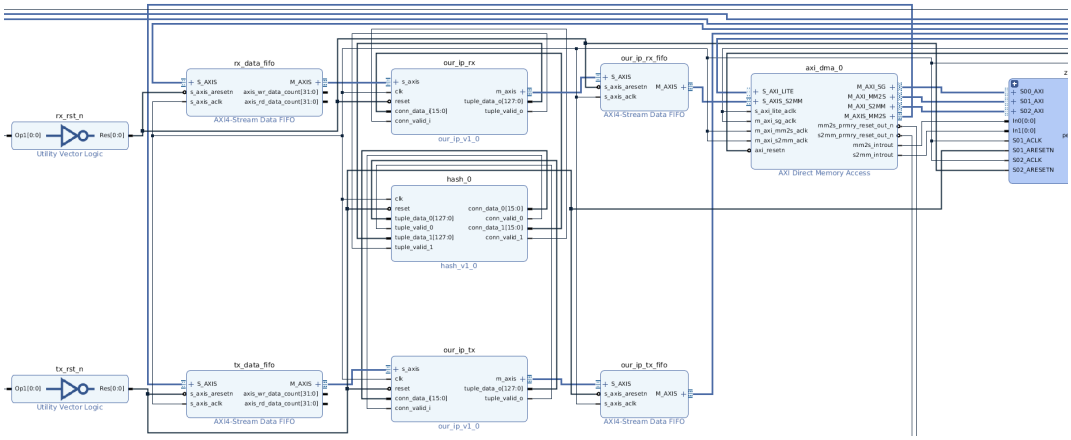

Figure 3: The key part of our board design.

4 EVALUATION

In this section, we present the evaluation of our NAT system, considering both software (SW NAT) and hardware (HW NAT) implementations. Two ablation settings (HW NAT without translation, without NAT) are introduced to measure the overhead of our HW NAT design. We conduct two evaluations:

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

4.1 Evaluation Setting

We provide details for our settings in the two evaluations. The connection hash table size is configured as 1024 for both evaluations. A C language-based traffic generator continuously sends TCP packets to the desktop machine via a Linux packet socket to the 10GbE interface of the board 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 four settings are:

- **HW NAT**: Our hardware-based NAT system as described in the System Design and Implementation sections. It uses a bidirectional hash table to store mappings of LAN 5-tuple and WAN 5-tuple, XOR as the hash function, and linear probing as the collision resolution strategy. It achieves high-speed data conversion and forwarding, improving throughput.
- **SW NAT**: A software-based NAT system running on four threads, using a shared hash table with linear probing. The table size is 1024, serving as a baseline compared to HW NAT.
- **HW NAT w/o translate**: Similar to HW NAT, but using packets that do not require translation (e.g., ICMP packets) to measure performance.
- **w/o NAT**: No NAT at all, with packets directly forwarded between MAC/GTH and DMA. This is the ideal scenario and serves as an upper bound to evaluate the performance of our system.

Latency: We do not measure 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 accounts for about 10% of the normal ping-pong latency (100-200 μs) measured with the `ping` command.

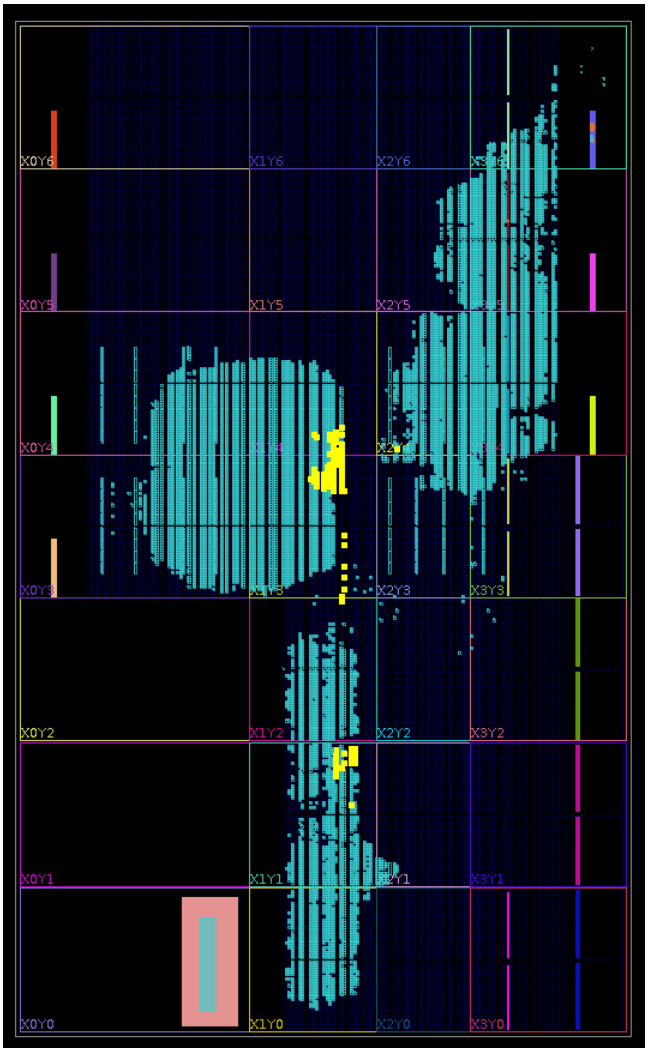

Figure 4: The post-synthesis netlist with the NAT part highlighted in yellow.

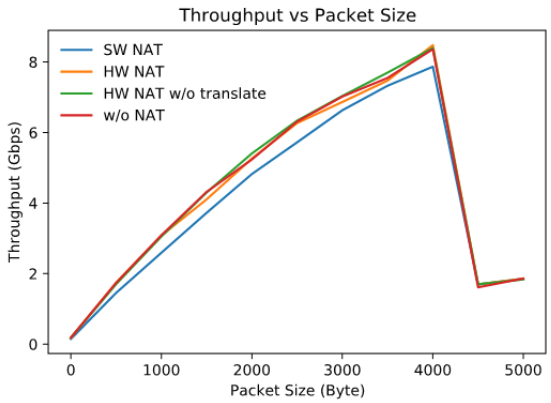

Figure 5: Throughput vs. Packet Size

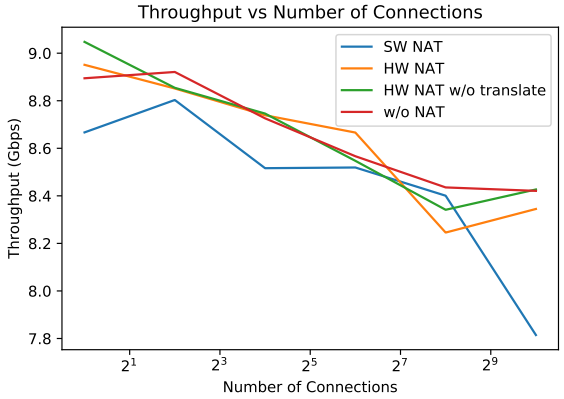

Figure 6: Throughput vs. Number of Connections

4.2 Results

Figure 5 depicts 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, then suddenly drops. This phenomenon is observed in all four settings.

Figure 6 illustrates 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

Our results lead to three main conclusions:

- **The performance of HW NAT is up to 7.7% higher than SW NAT under our setting.** In Figure 5, HW NAT and SW NAT reach maximum throughput under the same packet size. The maximum throughput of HW NAT is 8.48 Gbps, while the maximum throughput of SW NAT is 7.87 Gbps. This demonstrates that the HW NAT system can benefit from CPU savings.
- **Figure 5 shows that SW NAT is a bottleneck, and HW NAT is not.** The performance gap between SW NAT and no NAT indicates that SW NAT is a bottleneck that degrades the overall system performance. Conversely, HW NAT shows performance improvement compared to SW NAT, suggesting that hardware implementation outperforms software. There is almost no performance gap between HW NAT and no NAT, indicating 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. In contrast, HW NAT scales well with the number of connections, using a bidirectional hash table and an arbiter design to handle NAT mappings.

4.4 Anomalies

There are two anomalies in our evaluation results. We offer speculative explanations for 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-efficient performance of DMA, MAC, and GTH IP. The FPGA side has to process the packets at high speed, and may not handle 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, causing extra overflow handling for large packets. When the packet size exceeds 4K, the buffer may not store the entire packet, leading to packet segmentation and additional overhead, which reduces throughput.

5 DISCUSSION

In this section, we discuss potential enhancements to our system design and evaluation.

Improved Evaluation Method: Bypassing Processing System (PS). Our evaluation results reveal that the limited performance of the ZCU102 PS hardware hampers a comprehensive exploration of our hardware design's potential. To obtain more accurate evaluation results, bypassing the PS section of the board could be a more viable option. Specifically, processed Ethernet packets could be redirected to another SFP interface after undergoing processing by our custom IPs in the PL logic, avoiding involvement with the underpowered ARM CPUs.

Enhanced NAT Design for Performance and Resource Efficiency. Linear probing requires fewer resources at the cost of wasting more cycles. We can harness hardware parallelism by introducing structures like a fully associative cache in our NAT table, promising one-cycle latency. Additionally, implementing a RAM matrix may further reduce interconnect consumption.

Addressing Connection State Management. A fully functional NAT should track connection states to release NAT table entries in a timely manner. Due to project time constraints, we discussed and proposed a possible design for this aspect but did not implement it.

6 CONCLUSION

In this project, we delve into the design and implementation of an FPGA-based NAT. Our evaluation demonstrates approximately a 7.7% increase in throughput and enhanced scalability compared to software-based NAT.

7 ACKNOWLEDGEMENT

We would like to express our gratitude to Prof. Chenren Xu, the teaching assistants, and some senior colleagues for providing logistical support and valuable assistance.