Proteus : HLS-based NoC Generator and Simulator

Abhimanyu Rajeshkumar Bambhaniya, Yangyu Chen, Anshuman, Rohan Banerjee, and Tushar Krishna
Georgia Institute of Technology
{abambhaniya3, yangyuchen, vatsanshuman, rbanerjee45}@gatech.edu, tushar@ece.gatech.edu

***Abstract*—Networks-on-chip (NoCs) form the backbone fabric for connecting multi-core SoCs containing several processor cores and memories. Design-space exploration (DSE) of NoCs is a crucial part of the SoC design process to ensure that it does not become a bottleneck. DSE today is often hindered by the inherent trade-off between software simulation vs hardware emulation/evaluation. Software simulators are easily extendable and allow for the evaluation of new ideas but are not able to capture the hardware complexity. Meanwhile, RTL development is known to be time-consuming. This has forced DSE to use simulators followed by RTL development, evaluation and feedback, which slows down the overall design process. In an effort to tackle this problem, we present Proteus, a configurable and modular NoC simulator and RTL generator. Proteus is the first of its kind framework to use HLS compiler to develop NoCs from a C++ description of the NoC circuit. These generated NoCs can be simulated in software and tested on FPGAs. This allows users to do rapid DSE by providing the opportunity to tweak and test NoC architectures in real-time. We also compare Proteus-generated RTL with Chisel-generated and hand-written RTL in terms of area, timing and productivity. The ability to synthesize the NoC design on FPGAs can benefit large designs as the custom hardware results in faster run-time than cycle-accurate software simulators. Proteus is modeled similar to existing state-of-the-art simulators and offers users modifiable parameters to generate custom topologies, routing algorithms, and router microarchitectures.**

***Index Terms*—NoC Generator, Vitis HLS, Chisel, FPGA, NoC Simulator**

I. Introduction

Since the advent of multi-core chips, Network-on-Chips (NoCs) have been an essential part of multi-core designs, and have thus, sparked a plethora of research in academia and industry. Developing a high-performance application-specific NoC is crucial to meeting the power, performance, and area requirement of the target SoC [1].

The fundamental purpose of NoCs is to transact data between cores while maintaining coherency and throughput requirements. NoC implementations become more complex as the number of nodes grows and throughput requirement increases. NoC designers have proposed several techniques for problems like deadlock avoidance, optimal routing for a given topology, flow control, and efficient handling of virtual channels (VCs) and buffers [1]. Commonly used NoC simulators like Garnet [6] and BookSim [7] provide a quick and easy way of determining the effectiveness of the proposed solution. Unfortunately, they do not take the hardware characteristics of the implementations into account; for example, consider a hypothetical algorithm for adaptive routing that collects and broadcasts information about the buffer state for all nodes of an 8x8 Mesh across the network in a single cycle. This implementation might yield promising results in the software simulator, but the actual hardware implementation would not be able to achieve the estimated performance gain due to the lower clock frequency caused by longer critical paths for the complex logic. This would necessitate re-architecting the design and re-evaluating it to devise a solution with multi-cycle traversals taken into account. This motivates the need for accurate hardware modeling during architectural design-space exploration.

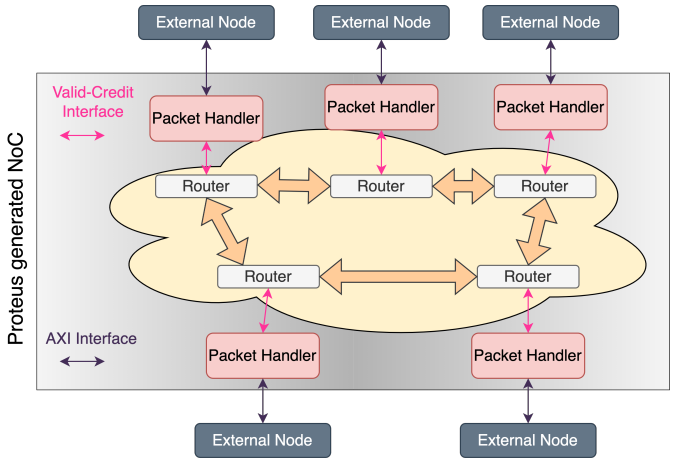

Fig. 1: Proteus Overview.

Hardware Description Languages (HDLs) are used for efficient implementation of the intended hardware but demand prolonged design cycles and more effort as compared to C++ based implementations [9]. There are various NoC implementations available as open source packages [3]–[5]. However, they require HDL coding expertise in languages such as Chisel, Bluespec, or Verilog, which may not be very common.

To mitigate the problem of enormous design efforts, myriads of solutions have been proposed across industry and academia that try to make higher-level abstractions for HDLs in user-friendly languages like C++/C/SystemC. High-level synthesis (HLS) tools such as Xilinx Vitis [11] convert C/C++ based codes into RTL that can be synthesized for FPGAs or ASICs. These tools allow users to perform rapid prototyping and deployment of intended hardware.

In this work, we introduce Proteus[1], an open-source HLS framework that can be used for NoC simulation and RTL generation. Figure 1 shows the overview of the framework. Proteus provides a baseline implementation of parameterizable NoCs that researchers can build upon to test their solutions

This work was supported in part by the ACE Center (SRC JUMP2.0).

[1]https://github.com/synergy-noc-generators/Proteus

TABLE I: Open Source NoC Generators Comparision

	CONNECT [3]	**OpenSoC Fabric [4]**	**OpenSMART [5]**	**Proteus(this work)**
Language	BSV generated Verilog	Chisel	BSV and Chisel	C++(Vitis HLS)
Topology	Arbitrary Typologies	Mesh, Flattened Butterfly	Mesh	Ring, Mesh, Torus
Software Simulation	No	Yes[a]	Yes[a]	Yes
Implementation time	High	Medium	Medium	Low

[a]Chisel/BSV generates C++ code that can be simulated.

in C++ simulation and/or on FPGAs. It can be easily configured/extended to model any desired interconnect. It generates an executable for software simulation and synthesizable RTL of NoC according to input parameters. Proteus thus allows users to perform architectural DSE and at the same time also assess the hardware impact of their solution in terms of area and timing without additional development effort. Since researchers who use NoC simulators already know languages like C++, we believe they can use the Proteus code for developing their design and then generating functional hardware using HLS synthesis flows. We use Xilinx Vitis HLS [11], one of the most popular HLS tools for RTL generation. This is the first work, to the best of our knowledge, that leverages HLS to enable users to perform NoC software simulation and generate synthesizable NoCs at the same time.

As a second contribution, we perform case studies comparing the HLS-generated RTL against Chisel-generated and hand-written RTLs in terms of code complexity and efficiency of the generated designs. A primary concern of HLS-generated RTL is inefficient in terms of area and timing compared to hand-written RTL. While this might be true for complex designs, we observe that HLS does a good job of making functional hardware for NoCs at comparable area cost compared to Chisel and hand-written SystemVerilog. We find that the area of HLS-generated NoC is only $\sim$15% higher than those of hand-written/chisel-generated RTL, and the router area breakdown in all three cases is similar. We thus believe that HLS fulfills the purpose of getting rapid estimates of area/timing overheads to support additional features in the target NoC.

The rest of the paper is organized as follows: Section 2 summarizes previous works on NoC generators/simulators and HDLs. Section 3 introduces our proposed framework, Proteus. In Section 4, we discuss the differences in the implementation of NoCs using HLS, Chisel, and SystemVerilog. In Section 5, we present evaluations comprising a DSE case-study using Proteus , FPGA evaluation, and comparison of area/timing of different HDLs. We conclude our work in Section 6.

II. Background and Related Works

A. High Level HDLs

In the software paradigm, HDLs can be abstracted between assembly language and high-level language. The amount of design effort required for complex hardware designs could be very high, especially for someone who is not an expert at implementing RTL. To mitigate this problem, there is a significant amount of research on high-level HDLs to enable rapid design space exploration(DSE) via software simulation (C++) instead of tedious and time-consuming RTL simulations. Chisel [2] is an open-source HDL based on Scala; it uses constructs and Object-Oriented Programming(OOP) concepts to enable concise and modular code. It generates C++ and synthesizable Verilog based on the Chisel code. Bluespec Verilog (BSV) was developed as an extension of Haskell to handle digital design. Like chisel, BSV generates C++ code for behavioral simulation and Verilog code for synthesis. MYHDL, PyRTL, and PyMTL are all python-based high-level HDLs. Even with a higher level of abstraction, these HDLs require a certain amount of expertise for writing functional code. Generally, for custom applications, describing the circuit at the RTL level is preferred as it generates the most optimal, power-efficient hardware.

HLS is a layer of abstraction above RTL, which allows users to generate synthesizable Verilog code from code written in C/C++. Vitis HLS (Xilinx) [11], Intel HLS Compiler(Intel), Catapult(Mentor), Stratus HLS(Cadence), Symphony C(Synopsys) are some of the popular HLS compilers.

In this work, we compare RTLs generated by Chisel and Vitis HLS with hand-written RTL. More details about the difference in using these three languages are outlined in section IV.

B. NoC Generators

NoC RTL generation suites provide users with a library of modularized components to build routers. These modules are usually parameterized, allowing the user freedom to make their design with a varying number of input/output ports, data widths, and buffer depths. Connect [3] is an FPGA-optimized NoC generator that produces BSV-generated Verilog for user-defined parameters. It exposes a web application to users for taking input and dumping out the RTL. Open SoC Fabric [4] provides an NoC generator designed with Chisel. It supports 2-D meshes and flattened butterfly networks of arbitrary design parameters. Their open-source codebase allows users to modify and extend the Chisel code. OpenSMART [5] is an open-source NoC generator based on BSV implementations for mesh and SMART routers. The aforementioned NoC generators are based on BSV/Chisel. Thus, only users proficient in HDLs can modify and extend the code base. In contrast, Proteus is C++ based code, which is not as esoteric as the other two allowing most users to implement new features and test the updated code. We qualitatively contrast these NoC generation suites with respect to Proteus in Table I.

C. NoC Simulators

NoC simulators provide a fast and efficient way to model network traffic. For faster performance modeling, most network simulators are implemented in high-level languages(e.g. C++) instead of RTL. The simulator must take into account all the latency and contention information when simulating the traffic model. C++ based Garnet [6] is a popular network simulator. It

is incorporated into gem5 [10] and used widely across academia and industry as it supports a variety of topologies and configurations of individual components. BookSim [7] is another simulator that supports a wide variety of parameterized topologies, routing functions, traffic loads, and router components. Since most such simulators are hardware unaware, they are ideal for faster simulation but fail to consider hardware implementation details. This can lead to misleading conclusions.

Compared to these simulators, Proteus can simulate at C++ level at similar speeds as Garnet/BookSim and provide an estimated hardware impact of implemented algorithm. An additional feature that Proteus provides is to test the NoC on FPGAs since it can generate synthesizable RTL. This is indeed beneficial since we see up to 10.73x speed-up in simulation time compared to C++ based simulators.

III. The Proteus NoC Generator

A. Overview

Proteus generates NoC topologies that can be used to study NoC characteristics with synthetic traffic patterns. NoCs generated by Proteus can be used as a modular plugin for multi-core design by connecting cores with NoC routers with AXI interface. We implement base topologies of ring, mesh, and torus as they account for nearly 60% of user-generated topology in open source NoC-generators as prior work shows [3]. Figure 1 shows an overview of Proteus and Table II shows the input parameters currently supported.

TABLE II: Input configuration parameters for Proteus

Input parameter	Implemented configurations
Number of Nodes	2-1024
Topology	Ring, Mesh, Torus
Router Microarchitecture	1-2 Cycle
Link Width	8-1024
VCs per Port	1-16
Routing	XY, YX, North-Last, West-First

B. Design Architecture

We designed Proteus to be hierarchical, parameterizable, and modular. The top module of an NoC is implemented as a C++ function that calls other functions and instantiates various class objects in the NoC. Routers are instantiated as class objects, and sub-modules are functions of the router class. Distinguishable implementation of sub-modules provides the ability to change a module functionality easily; for example, if users want to implement a new routing algorithm, they need to add a C++ logic for the routing function in the router class.

C. Router Micro-architecture

This section will outline the detail of router implementation. The Router, as shown in Figure 2, is a single-cycle router with data registering at input buffer and passing through route compute unit, switch arbiter, and traversing through the switch to output links. We describe the functioning of sub-modules of the router:

Input Port: An input port contains virtual channel buffers and VC multiplexers. Input buffers are the primary data storage

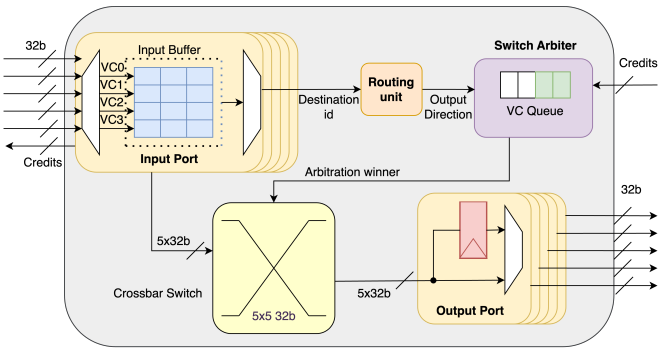

Fig. 2: The router micro-architecture implemented in Proteus

unit of the routers where incoming data from neighbouring nodes is latched. The core injects data into the buffer inside the local input port. This module is also responsible for sending credits back to the port of the previous router.

Routing Unit: Once the new data is latched in the input unit, the next destination is calculated for each valid flit residing in the input buffers. Proteus supports two Dimension Order Routing (DOR) algorithms - XY and YX for mesh and torus. It also supports turn-restricted routing like North-Last/West-First that helps ensure deadlock-free traversal in mesh and torus. We also implement random oblivious routing support for mesh and torus to help users test new deadlock avoidance algorithm.

Switch Arbiter: Switch arbiter is implemented using matrix arbiter [1]. In an N:1 matrix arbiter, N one-bit registers are used to encode priorities among the requesters and are updated after each grant. The switch arbiter will choose a flit with the highest priority in different VCs of multiple ports. The switch arbiter passes arbitration information to the crossbar switch, which helps to enable link traversal from input buffers to the output link. The switch arbiter keeps track of free VC ids using credits from the receiving router. These free VC ids are stored in a queue, and the winner of the switch arbitration picks a VC id from the head of the queue.

Crossbar Switch: The crossbar switch fetches the flits which won the arbitration in the switch arbiter and assigns them to appropriate output links. The output of the crossbar is sent to the output port, where it can be either sent out in the same cycle or registered at the output to make a two-cycle router.

D. Network Interface and Statistics

Proteus generated NoCs can be plugged into external SoC or can be used as a standalone simulator. We describe the functionalities of key modules for enabling both.

Packet Handler: The packet handler (Figure 1) acts as the network interface and breaks incoming packets from external nodes into smaller flits if needed, depending on the channel width. It also aids in statistic collection.

Latency Calculation: Since Proteus serves as a NoC simulator, traffic monitoring an essential feature. Each router provides the following metrics to users in AXI readable registers: an average NoC traversal latency, an average queuing latency, packet traversal statistics.

Traffic Generation: To run standalone NoC simulations, we provide synthetic traffic generators to create packets at user-specified injection rates for destinations determined via patterns such as pseudo-random, bit-complement, bit-reverse, shuffle, transpose, and bit-rotation. We also offer the ability for the NoC to interface with external nodes using AXI through the packet handlers.

Implementation of Random Numbers Using LFSRs: One of the main differences between Proteus and other C++ simulators, like Garnet or Booksim, is the method of generating random numbers. Random numbers are used to generate synthetic traffic to vary the injection rate into the NoC. In an effort to model real hardware, Proteus uses pseudo-random numbers that fulfill the requirement of traffic generation. Proteus uses linear feedback shift registers (LFSRs) to generate pseudo-random numbers. LFSR is a shift register whose input bit is driven by the XOR of some bits of the overall shift register value. Due to the definite nature of the input bit, the output of the next step can be determined by the current state. Thus, it is possible to have a recurring loop of values in LFSR. Depending on the initial seed and function at the input, it is possible to create long loops simulating random behavior.

Deadlock Detection: Each router checks for stalled buffers every cycle and keeps track of the number of cycles for which the buffer is stalled. After a certain user-defined number of cycles, the router is considered to be deadlocked, and the message is passed to the top-level NoC interface.

IV. HLS vs. Chisel vs. SystemVerilog Implementations

SystemVerilog, Chisel, and Vitis HLS are three different levels of abstraction in terms of developing the same hardware functionality. For comparison, we created a 4-router ring NoC with 1-stage router (4-buffer) in all languages[2]. In this section, the implementation of *routing unit*, coded using these different languages is compared. Quantitative evaluations for the full router are presented in Sec. V-E.

HLS: In C++, the routing unit in a ring needs to determine the shortest path to the destination. It also needs to figure out the direction in which the newly injected packets would move. As shown in Listing 1, we use simple if-else statements to check for the number of hops to determine the direction as an output of the routing function. The simplicity of coding with a high-level language such as C++ enables users to focus on the algorithm's functionality rather than on implementation details like clocks, resets, and enable signals. Due to the small number of lines of code, HLS affords rapid design ramp-up and reduced debug times for the network.

Listing 1: HLS Implementation

```
if (dst_id == this->router_id) return EVICT;
if (input_port == LOCAL)
    return go_east_hop > go_west_hop ? WEST : EAST;
else if (input_port == EAST)    return WEST;
else if (input_port == WEST)    return EAST;
else                            return ERROR;
```

[2]We do not directly use OpenSoC [4] generated router to keep the router micro-architecture same across the three implementations.

Verilog: In SystemVerilog, the routing unit is a combinational logic written as series of if and else conditions inside an always block, Listing 2. While the code looks similar to HLS code, there are various details SystemVerilog developers need to consider even in such a simple block of code. Some of these are non-blocking or blocking assignments of the variable. If the developer incorrectly uses these assignments, it can lead to a functionality issue. Also, a major part of SystemVerilog design revolves around sequential logic, for which trigger conditions and edge sensitivity are some of the critical things that need to be considered.

Listing 2: RTL Implementation

```
always @(posedge clk or negedge rst_n) begin
  if (~rst_n) out_dir <= LOCAL;
  else if (en) begin
    if (dest_id == ROUTER_ID) out_dir <= LOCAL;
    else if (IN_PORT == EAST) out_dir <= WEST;
    else if (IN_PORT == WEST) out_dir <= EAST;
    else if (IN_PORT == LOCAL) begin
      out_dir <= (east_hop >= west_hop) ? WEST : EAST;
    end else
      out_dir <= out_dir;
    end
end
```

Listing 3: Chisel Implementation

```
when(flitValid === 1.U) {
  when((router_id === dest_id))
    op_port_reg := LOCAL
  .elsewhen(dest_id > router_id) {
    when((dest_id - router_id) >= (N/2).U)
      op_port_reg := EAST
    .otherwise
      op_port_reg := WEST
  } .otherwise {
    when((router_id - dest_id) >= (N/2).U)
      op_port_reg := WEST
    .otherwise
      op_port_reg := EAST
  }
}
```

Chisel: Listing 3 shows the routing unit in Chisel. We implement it using when/elsewhen statements using router-id and dest-id to calculate the shortest path in the N-node ring. Chisel supports parameterized designs, the use of classes, and inheritance, thereby making the code modular and scalable. Chisel also outputs C++ code used to emulate the design, which can be further used for verification. Although, Chisel supports C++ based emulation, it still requires significant design time for implementation when compared to traditional C++ based code.

V. Evaluations

A. Methodology

We tested the basic functionality of NoCs generated by Proteus using C++ based standalone test benches. Vitis HLS is utilized to compile these test benches which include NoCs as the test unit. These test benches generate traffic at each router at user-specified rates going towards user-specified destinations.

We also validated our generated designs with hardware synthesis tools for ASIC and FPGA design flows. For the ASIC flow, we use Synopsys Design Compiler and Cadence Innovus with the NanGate 15nm open cell library for synthesis

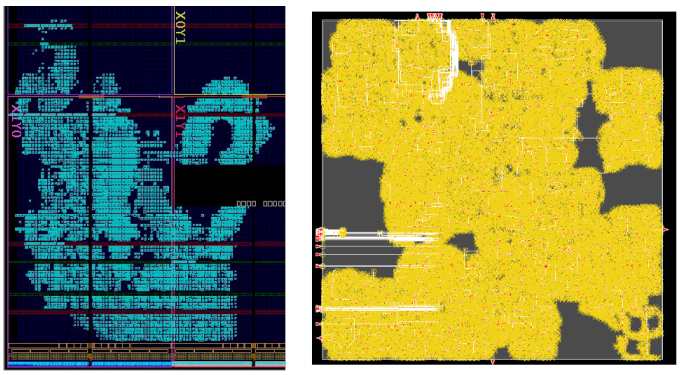

Fig. 3: The layout results of 4x4 mesh in Ultra96v2 FPGA (left) and ASIC using cadence innovus (right).

and place-route respectively. Next for the FPGA validation, we generate bitstreams using Xilinx Vivado Design Suite for Ultra96v2 evaluation board. The hardware synthesis tools provide area, power, and timing closure information. Figure 3 shows the FPGA and ASIC layouts of a 4x4 mesh with 1-cycle router (4 VCs and 48-bit wide channels). The FPGA implementation uses 9252 LUTs (17%) and 10859 FFs (10%). The ASIC implementation uses 68921 um^2 in 15nm.

B. Proteus Validation

For validation of Proteus, we use the synthetic traffic generator mentioned in Sec. III-D to compare network latency for different synthetic traffic. Figure 4 shows comparison of average latency of 4x4 mesh when simulated with Proteus and Garnet. The latency trends of Proteus-generated mesh and Garnet-simulated mesh are similar[3].

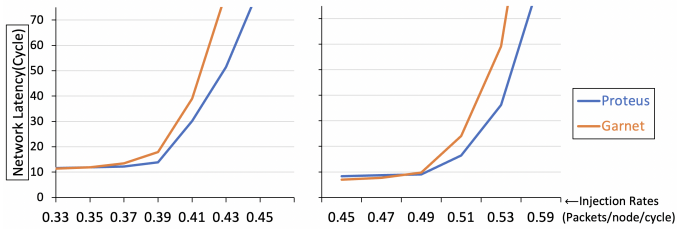

Fig. 4: Average network latency versus injection rate of Proteus generated 4x4 mesh and GARNET for bit complement(left) and bit rotation(right) traffic.

We further exhibit the flexibility of Proteus by creating a functional 1D systolic array using an NoC generated by Proteus. We connect 16 systolic cores and 2 memory units to the routers of an 18-node ring to perform 1D systolic convolution, as shown in Figure 5. Each systolic core is capable of doing 1 multiple and accumulate operations per cycle. Elements of vector B are kept stationary in a core and vector A's elements are streamed from the first memory core. Finally, the output is written back to the second memory unit. We compare the final output with software-generated convolution output is verify the functionality of the NoC.

[3]Proteus shows slightly higher throughput which is due to differences in the arbiter design (Garnet uses round-robin while Proteus uses matrix arbiters).

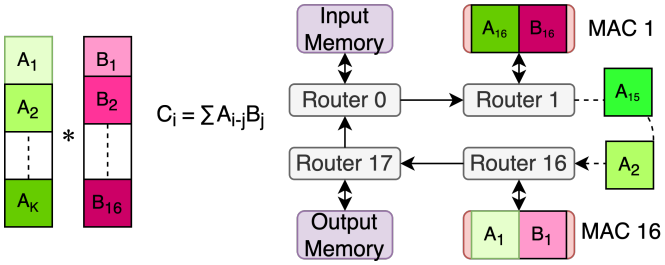

Fig. 5: Proteus generated ring performing vector convolution.

C. Design-space Exploration using proteus

Next, we use Proteus to perform a DSE case study by varying the number of VCs per port. For a 4x4 mesh we sweep the number of VCs from 3 to 16, and observe the relationship of network throughput, area, and timing. We measure the throughput as maximum number of packets received in a given time period. For area and timing, we rely on the reports generated by ASIC synthesis of design. We observe in Figure 6 that the throughput increases as we increase the number of VCs, but it saturates after 16VCs. We can observe an almost linear increase in area as the router area is dominated by area of VC buffers. We also observe a drop in the maximum frequency because of the increase in the size of VC mux caused due to an increase in the VC count.

This case study shows that Proteus can enable researchers to use one framework to perform both performance evaluation and get real timing, area number. This is unlike pure software simulators which can often end up modeling unrealistic hardware, and pure RTL models which limit design-space exploration.

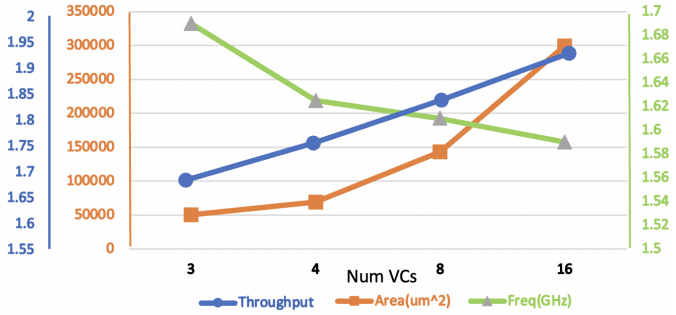

Fig. 6: DSE of number of VCs using Proteus .

D. FPGA evaluations

A key feature distinguishing Proteus from existing NoC simulators is its ability to run the NoC on real hardware. After verifying the functionality of the NoC, users can generate synthesizable RTL. Vivado uses this RTL to generate a hardware configuration for the FPGA. We load the hardware configuration file generated by Vivado to the FPGA board by using a python script. The time taken from generation of the RTL to programming the board is comparable to Garnet build time. Table III compares the simulation speed for running the same topology on a ultra96v2 FPGA board and on Garnet with the same configuration and different synthetic traffics. We see

speedup in the run-time of FPGA up to 10.73 times faster compared to Garnet runtime.

TABLE III: Simulation Speed: Proteus on FPGA vs Garnet

Configuration	FPGA(s)	Garnet(s)	Speedup
Ring: 32 Nodes	1.348-1.526	2.038-3.155	1.51-2.06
Torus: 64 Nodes	0.882-2.256	7.943-9.182	4.07-9
Mesh: 64 Nodes	0.773-2.124	7.866-10.542	4.76-10.73

E. RTL Design evaluations

Finally, we compare the HLS-generated RTL against Chisel-generated and hand-written RTL(SystemVerilog) in terms of code complexity and efficiency of the generated designs. We implemented a 4-node ring with 48-bit links, 2-cycle routers, and four VCs per port - each 1 flit deep. We consider the design area, timing, and productivity as the metrics of comparison.

Area: Figure 7 plots the area breakdown of the various components of the router in the 15nm ASIC flow with three different RTLs. We observe the ratio of the combinational logic to the sequential logic is nearly similar in all three RTLs. Most of the flops are used in the input buffers and the combinational logic for the implementation of the switch crossbar. Area of the router generated by HLS is comparable to area of verilog RTL and chisel-generated RTL. HLS also adds a central controller to enable communication between the Processing System and Programmable Logic side of FPGA. For fairness, we do not include that area in ring area. We observe that the architecture of a NoC router is simple and modular enough that HLS is able to generate competitive designs to hand written RTL.

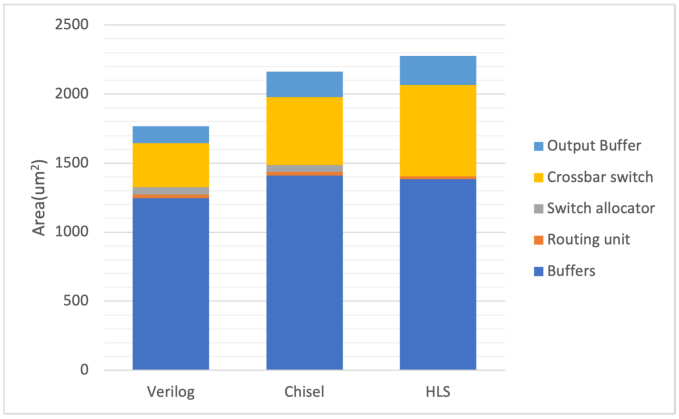

Fig. 7: Router area breakdowns for 3 languages

Timing: Figure 8 plots the maximum achievable frequency of the ring on the 15nm ASIC flow and the ultra96v2 FPGA for the three RTLs. The maximum frequency achieved by RTL generated by Chisel and hand-written RTL is similar to the HLS-generated RTL running at slightly lower rate. The critical path in all three cases is the path through the crossbar switch. It is important to note that the goal of the synthesis tools (both ASIC and FPGA) is to meet the timing, which comes at the cost of larger cells and more buffers.

Productivity: We quantify productivity in terms of the lines of code needed to implement the design, and quality by the time taken in hours to implement and verify functional code. Quality

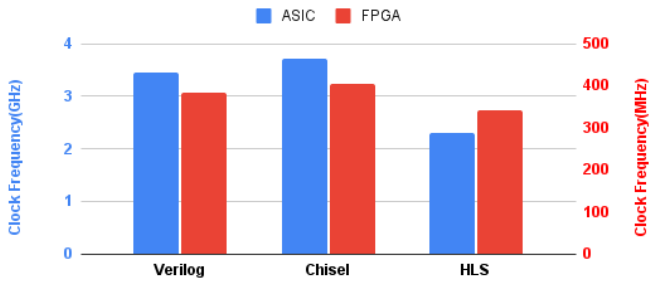

Fig. 8: Max frequency of NoCs in ASIC and FPGA flow

is calculated as the maximum frequency per unit area of the synthesized RTL. We show these parameters in Table IV for the implementation of a 4-node ring for the three languages. HLS outperforms RTL and Chisel in both development time and lines of source code. Although the area and timing of Chisel and RTL are better than HLS, lower design time is vitally impactful in rapid design space exploration. [9]

TABLE IV: Productivity for implementing the ring NoC

Language	LoC	Hours	Freq/Area	Quality/Hours
Hand-Written RTL	932	24	**195.39**	8.14
Chisel	860	18	171.15	9.51
HLS	**578**	**6**	101.04	**16.84**

VI. Conclusion

This work presents Proteus; an HLS-based NoC generator and simulator in C++. It can generate synthesizable NoCs based on user-defined configuration. Proteus generated NoCs can be simulated at the C++ level and tested on FPGAs. We show that Proteus-generated RTL is comparable to hand-written and Chisel-generated RTL. A shorter design cycle with Proteus trumps the marginal area/power savings from HDL/Chisel implementations. Thus, it can be a valuable tool for researchers to do rapid design space exploration of NoCs.

References

[1] Jerger, Natalie Enright, Tushar Krishna, and Li-Shiuan Peh. "On-chip networks." Synthesis Lectures on Computer Architecture 12.3 ,2017.

[2] J. Bachrach et al., "Chisel: Constructing hardware in a Scala embedded language," in DAC 2012

[3] M. K. Papamichael and J. C. Hoe, "CONNECT: re-examining conventional wisdom for designing nocs in the context of FPGAs",FPGA 2012

[4] F. Fatollahi-Fard, D. Donofrio, G. Michelogiannakis and J. Shalf, "OpenSoC Fabric: On-chip network generator," in ISPASS 2016

[5] H. Kwon and T. Krishna, "OpenSMART: Single-cycle multi-hop NoC generator in BSV and Chisel,"in ISPASS 2017

[6] N. Agarwal, T. Krishna, L. -S. Peh and N. K. Jha, "GARNET: A detailed on-chip network model inside a full-system simulator," in ISPASS, 2009

[7] Nan Jiang et al., "A detailed and flexible cycle-accurate Network-on-Chip simulator," in ISPASS, 2013

[8] F. Muñoz-Martínez, J. L. Abellán, M. E. Acacio and T. Krishna, "STONNE: Enabling Cycle-Level Microarchitectural Simulation for DNN Inference Accelerators," in IEEE CAL, 2021.

[9] S. Lahti, P. Sjövall, J. Vanne and T. D. Hämäläinen, "Are We There Yet? A Study on the State of High-Level Synthesis," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, May 2019.

[10] Binkert, Nathan, et al. "The gem5 simulator." ACM SIGARCH computer architecture news 39.2 (2011)

[11] Vitis user guide. 2022. [online] Available at: https://docs.xilinx.com/r/en-US/ug1399-vitis-hls