TPNoC: An Efficient Topology Reconfigurable NoC Generator

Jiangnan Yu
Fudan University
Shanghai, China
jiangnanyu21@m.fudan.edu.cn

Yang Fan*
Fudan University
Shanghai, China
yangfan@fudan.edu.cn

Xiaoling Yi
Fudan University
Shanghai, China
xlyi20@fudan.edu.cn

Chixiao Chen
Fudan University
Shanghai, China
cxchen@fudan.edu.cn

Jun Tao
Fudan University
Shanghai, China
taojun@fudan.edu.cn

Done Xu
ZTE Corporation
Shanghai, China
xu.dong1@zte.com.cn

Xiankui Xiong
ZTE Corporation
Shanghai, China
xiong.xiankui@zte.com.cn

Haitao Yang
Fudan University
Shanghai, China
21212020175@m.fudan.edu.cn

ABSTRACT

With the core count increasing in Chip to support various data-intensive workloads, Network-on-chip (NoC) has become the better solution for addressing on-chip interconnection. Various data-intensive workloads have different traffic patterns that require NoC with different topologies and microarchitectures. On the one hand, topology type selection has a great influence on the final performance, area, and energy. However, it is difficult to change the topology type in the traditional NoC RTL design process once it is determined. On the other hand, NoC platforms have many tunable micro-architecture design parameters, which require careful design space exploration to trade off performance advantages and overhead. Designing and validating each microarchitecture of NoCs to account for various trade-offs will greatly exacerbate the design cost issue.

We propose TPNoC, a topology reconfigurable NoC generator framework. The TPNoC framework is divided into parameterization, verification evaluation, and RTL generation stages, which can be reconfigured into plug-and-play NoC platforms with different topologies and microarchitectures according to the architect's configuration. Architects can efficiently use TPNoC to explore the best NoC for specific workloads. Through verification and evaluation experiments, we demonstrate that TPNoC can be efficiently reconfigured into different NoCs while maintaining high performance.

CCS CONCEPTS

• **Networks** → **Network on chip**; **Network simulations**.

**Corresponding authors

GLSVLSI '23, June 5–7, 2023, Knoxville, TN, USA

ACM ISBN 979-8-4007-0125-2/23/06…$15.00
https://doi.org/10.1145/3583781.3590257

KEYWORDS

network-on-chip generator, modeling, simulation

ACM Reference Format:
Jiangnan Yu, Yang Fan, Xiaoling Yi, Chixiao Chen, Jun Tao, Done Xu, Xiankui Xiong, and Haitao Yang. 2023. TPNoC: An Efficient Topology Reconfigurable NoC Generator. In *Proceedings of the Great Lakes Symposium on VLSI 2023 (GLSVLSI '23), June 5–7, 2023, Knoxville, TN, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3583781.3590257

1 INTRODUCTION

With the development of on-chip manufacturing technologies and the requirements of high-performance computing, the core count is growing quickly in Chip to support various data-intensive workloads. NoC has become the de facto solution for multi/many-core architectures in addressing the communication challenge. The suitable domains vary from multi/many-core chips in HPC supercomputers to high-end servers with dozens to hundreds of homogeneous cores. In fact, there are many existing intellectual properties (IP) cores that require on-chip interconnection communication solutions to expand computing power. For example, Deep Neural Network Models (DNN) model workloads are soaring in complexity over the past years [1][11][9]. OpenAI claims that the amount of computing required to train the most powerful networks doubles every 3.4 months [16]. The computing power of the previously designed single-core DNN accelerators is challenging to meet the increasingly large DNN model. Existing single-core DNN accelerators need a flexible and scalable interconnection system to exert their remaining value. Actually, more and more DNN accelerators have shifted to many-core architectures using NoC systems solutions [6][15][10].

Designing and testing a NoCs system from scratch is labor-intensive and error-prone work. A fully functional NoC RTL generator can significantly simplify the design space exploration and verification process. The output of the NoC RTL generator can be used both as a model providing the most accurate network implementation and timing information and as a final design.

Table 1: Comparison of existing NoC generators

Features	*OpenSoC*	*OpenSmart*	*EAGAN*	*Constellation*	*TPNoC*
Language	Chisel2	Chisel2/BSV	Chisel3	Chisel3	Chisel3
With protocol bridge	No	No	No	Yes	Yes
Topology	Mesh/Fat-Tree	Mesh	Unknown	Arbitrary	Mesh/Ring/Arbitrary
Router Microarchitecture	4 cycles	1-2 Cycles	Unknown	2/4cycles	2cycles

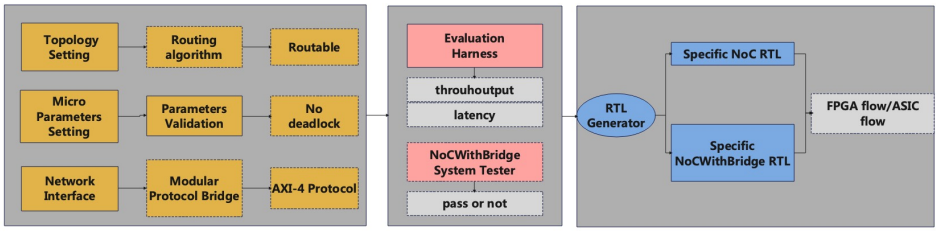


Figure 1: Flowchart of deploying the TPNoC framework.

TPNoC is a NoC generator framework written in the Chisel language. Chisel[2] is an open-source hardware construction language embedded in Scala, which improves the abstraction level for hardware design by providing concepts such as object orientation, inheritance, functional programming, parameterized types, etc. Chisel has been under active development over the last ten years. There have been several examples of successfully manufactured processors designed exclusively using Chisel [3], which illustrates a faithful representation of the Verilog generated by the Chisel from the user's Scala description. There are two generations of Chisel, commonly referred to as Chisel2 and Chisel3. Chisel provides both software and hardware models from the same codebase. Chisel's software models can generate a fast, cycle-accurate RTL simulator implemented in C++, which can be used for simple network design exploration. Taking advantage of Chisel's hardware model tools, the TPNoC structure allows the synthesis of interesting and important network designs for the target ASIC flow or FPGA flow, resulting in accurate power parameters and evaluation of cycle times that software simulators cannot achieve.

In past years, the NoC generator has received much attention from the research community. Table1 compares the features supported by existing NoC RTL generators and our proposed NoC generator TPNoC. OpenSmart [12] focused on proposing low latency router NoC generators. The BSV variant OpenSMART supports arbitrary topologies. However, We found that the open-source Chisel variant of OpenSMART only provides support for fixed 2D Mesh topologies and OpenSMART does not implement any real protocol bridge for plug-and-play use. OpenSoC Fabric [7] provides an NoC generator written in Chisel2, which supports 2-D mesh and flattened butterfly networks of arbitrary size. OpenSoC Fabric claims to be based on the AXI-Lite bus standard for easy connection to ARM and ARM-compatible devices. However, the content of this part of the latest open-source code has been commented on, and there is no specific protocol bridge implementation. EAGEN [17] is a NoC generator written in Chisel3 focusing on proposing a configuration recommendation algorithm that can train models to predict better architectural configurations, without considering the feasibility of design and verification. Constellation [18] integrated into Chipyard SoC design framework that allows system-wide exploration of heterogeneous SoC architectures with irregular memory structures. Chipyard is a huge chip design framework, which brings extremely high learning costs to Constellation. These NoC generators allow the configuration of many microarchitectural parameters and generate microarchitectural designs that can be used to simulate performance and synthesize area and power estimates. The key challenge of these open-source NoC generators is not functional enough that they cannot meet the need for flexible topology reconfiguration, efficient microarchitecture parameters setting, and plug-and-play use of NoC designs.

To address these limitations, we propose a NoC generator framework with parameterization, verification, evaluation, and RTL generation stage. Architects can use TPNoC for efficient design space exploration, reducing the cost of design and verification. At the same time, the TPNoC framework has an optional protocol bridge module that can reuse existing IP cores in a plug-and-play manner. We plan to make the TPNoC open-source.

The remainder of the paper is organized as follows: Section 2 is the overview of TPNoC design. Section 3 describes the parameterization stage of TPNoC, which contains the design details of TPNoC. Section 4 describes the system-level verification and performance evaluation stage. Section 5 shows an example of the RTL stage. Finally, Section 6 concludes the paper.

2 TPNOC DESIGN OVERVIEW

The goal of TPNoC is to serve as a NoC design tool that allows architects to efficiently tune topology and microarchitectural parameters to generate plug-and-play interconnect communication networks. TPNoC takes the "microarchitecture parameters, topology and protocols used by IP" as the configuration input, and the output of TPNoC is NoC RTL configured by the user in the form of Chisel or Verilog. The TPNoC framework is divided into three stages: parametrization, verification and performance evaluation, and RTL Generator which is depicted in Figure 1.

The NoC parameterization stage is orthogonalized into three parts.

- The topology setup part decouples the specification of the physical NoC and the routing algorithms. Different topology settings match different routing policies and router degrees.
- The microarchitecture parameter setting part will check the rationality of the architect configuration parameters.
- The network interface part is to generate the corresponding protocol bridge according to the configuration of the architect.

The verification and evaluation stage generates a tester and performance results that are used for verification and the performance evaluation of the configured NoC.

The RTL generation stage will take the validated parameters to generate a complete RTL for the configured design.

3 PARAMETERIZATION STAGE

The parameterization stage contains the TPNoC design details. This section introduces the details of TPNoC design from the three parts included in the TPNoC parameterization stage.

3.1 Topology Setting

For NoC designs, topology is the first and one of the most important design decisions because it has a significant impact on overall system performance. Topology determines the number of hops (or routers) a message traverses and the length of the interconnection between hops, which can significantly affect network latency. Since energy is consumed by traversing routers and links, the effect of topology on hop count directly affects energy consumption. In the traditional NoC process, once the topology type is determined, it is very difficult to modify it.

TPNoC utilizes Chisel's design language features to achieve the purpose of flexibly configuring topology types. We consider that different topologies require routers to have different numbers of ports and use different routing algorithms. We integrate the parameters affected by the topology into the singleton object provided by the Chisel3 language. As the result, the corresponding parameters under the topology configuration class will be called after the user enters a specific topology type, which will affect the number of router ports, the name of the routing algorithm module used to avoid routing deadlock, the number of routers, etc. Figure 2 shows the topology configuration code in Chisel3. If the architect selects the **Mesh** parameter in the configuration file, TPNoC will call the **MeshConnectConfig** object as shown in Figure 3 for parameter configuration.

```
val ConnectConfigs = (
  topology match { //i is meshsize (i*i=mesh router nums)  , j is ring router nums
    case "Mesh"          => new Configs(MeshConnectConfig(i))
    case "Ring"          => new Configs(RingConnectConfig(j))
    case "RingWithMesh" => new Configs(HybirdConnectConfig(i,j))
  })
```

Figure 2: Topology config.

Extensive research shows that most many-core DNN accelerators use mesh, ring, or ringwithmesh topologies [4][5][14][8], which proves that these topologies satisfy most NoC design requirements

```
object MeshConnectConfig {
  def apply(i:Int) = Map(
    "Topology" -> "Mesh",
    "NUM_USER_SEND_PORTS"  -> i*i,// sender
    "NUM_USER_RECV_PORTS"  -> i*i,//receiver
    "NUM_VCS"              -> 3,//Virtual channel number
    "FLIT_DATA_WIDTH" -> 32,//flit data width
    "FLIT_BUFFER_DEPTH" -> 6, //flit buffer depth
    "MeshDirSize" -> 5, //=mesh router degrees
    "RouterNUM" -> i*i, //router number
    "RouteAlgo" -> "XY" //routing algorithm
  )
}
```

Figure 3: MeshConnectConfig example.

for many-core DNN accelerators. TPNoC takes these three topologies as the main targets for reconfiguration. Due to the modularity of the TPNoC design, architects can mimic existing topologies to add other types of topological configurations. For example, the architect can imitate the mesh topology to write a torus topology, directly add the **Torus** parameter in **ConnectConfigs**, and define the corresponding topology connection object **TorusConnectConfig** by imitating the **MeshConnectConfig** object, which contains parameters such as the routing algorithm to call, router port parameters, etc. Due to the similarity between the torus and the mesh topology, it is also possible to directly call the X-Y routing algorithm module to avoid routing deadlock. If other routing algorithms are needed, the architect can write the routing algorithm required by the custom topology and imitate the existing X-Y routing algorithm module calling method to call the custom routing algorithm and test its correctness through the subsequent verification and evaluation stage.

3.2 Microarchitecture parameterization and modularity

3.2.1 Microarchitectures Parameter Setting. TPNoC has a wealth of user-tunable micro-parameters, which allows architects to quickly start building NoCs with different configurations to generate the myriad NoC microarchitectures required for specific scenarios and the NoC design space exploration, rather than designing NoCs from scratch. TPNoC configuration parameters consist of two parts, IP protocol bridge parameters, and NoC microarchitecture parameters, as shown in Figure 4.

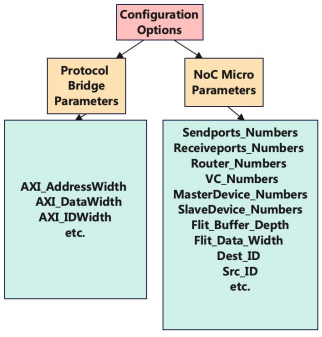


Figure 4: Configuration parameters.

3.2.2 Microarchitectures Modularity. We modularize the code structure of TPNoC to facilitate the adjustment and replacement of TPNoC modules to make the generated NoC perform better. The fundamental router of TPNoC is a state-of-the-art router with virtual channel flow control. The routers consist of input and output modules and have a latency of 2 cycles. The design of our router merges the traditional five-stage pipeline into the two-stage pipeline. Route Computation, virtual channel allocation, and switch allocation are integrated into the input module, and switch traversal and link traversal are integrated into the output module.

Input Module. Each input module consists of a data path module and a replaceable routing algorithm module. The schematic diagram of the input module is shown in Figure 5.

- Input datapath module: Each input datapath module has a configurable number of virtual channels (VCs) that architects can quickly adjust in configuration files. There are three flit types, head flit, data flit, and tail flit. A routing table is maintained in the data path module. For each packet, only the head flit is sent to the routing module for routing direction calculations. When the routing path is turned on, the routing path is recorded and locked. Subsequent flits follow the head flit path walks which speed up the routing of subsequent flits routing, and the path will not be released until the datapath module receives the tail flit of the packet. If VC is full, the module set the corresponding ready interface signal down, creating back pressure on the connection.
- Routing algorithm module: TPNoC supports two dimension order routing (DOR) algorithms - XY / YX for a mesh topology to avoid routing deadlock. In the XY / YX routing module, the routing algorithm module calculates the routing direction according to the head flit information. Each time the head flit arrives, the routing algorithm module will compare the router's address with the target address stored in the header flit, and determine the direction of the next hop depending on the routing algorithm we choose. For irregular topologies, we provide support up-down routing algorithm [13] in the irregular topology algorithm. We enforce strict ordering of nodes in the network by marking the links towards the root node as up, those away as down, and arbitrarily tagging the equidistant ones. By disallowing turns from a down-link to an up-link, all circular dependencies in the network are broken to avoid routing deadlocks.

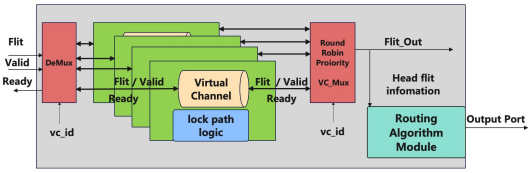


Figure 5: Schematic diagram of the input module.

Output module. The schematic diagram of the output module is shown in Figure 6. Since each output port may receive packets from other input ports of the router and has only one physical output link, the arbitration mechanism is essential. The basic arbitration mechanism used in the design is round-robin arbitration (RRA), which ensures that each input port data has the same probability to obtain the right to use the output port. VCs with the same priority from other input ports are the inputs to the RRA.

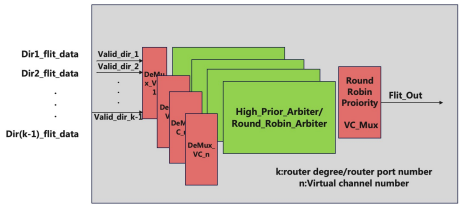


Figure 6: Schematic diagram of the output module.

Table 2: Packet and Flit format

Packet Field	Data width	Flit Field	Data width
None	None	Is head flit	1
None	None	Is tali flit	1
VcID	log2(virtual channel numbers)	VcID	log2(virtual channel numbers)
DestID	log2(receiver numbers)	DestID	log2(receiver numbers)
SrcID	log2(sender numbers)	SrcID	log2(sender numbers)
Packet Payload	uncertain	Flit payload	defined flit data width

3.3 Network Interface

To provide interconnection for IPs using different protocols, TPNoC was designed to be protocol-independent by exposing a minimal packet interface at the endpoints. The format of TPNoC's customized data packet and flit is shown in the table 2.

AXI4 Bridge Module. There are several widely used on-chip communication standards in today's SoCs. Since the Advanced eXtensible Interface 4.0 (AXI4) protocol is widely used at present, we have designed a bridge for the AXI4/AXI4-lit protocol. The primary function of the protocol bridge is to convert standard high-level protocol messages into simple packet formats in TPNoC. Each IP is connected to a router as a master node or a slave node and communicates with each other through transactions. For example, the AXI4 bridge for master IP first converts 5 channels into 5 packets and then converts the 5 AXI4[-Lite] channels into packets in 3 virtual channels (aw/ar channel to a-packet in VC 2, w channel to w-packet in VC 1, and b/r channel to br packets in VC 0). Each router is assigned a node ID that maps the address space of the connected IP. According to the information of the address channel, the bridge assigns a value to the destination node ID (DestID) or source node ID (SrcID) field. All other fields in the protocol message are packed in the payload field. To ensure no deadlock in the protocol, non-blocking between channels needs to be satisfied. By default, multiple channels are multiplexed onto the same routing resource using a round-robin arbiter. The configuration options of the bridge types currently supported by TPNoC are shown in Figure 7. Similar to the configurable process of the topology, the user inputs different bridge type parameters that will make the NoC generated by TPNoC call different bridges. The choice of protocol bridge type is a configurable option so that other types of protocol bridges can be written in the future to support on-chip interconnects of IP using other types of protocols.

```
new NetworkConfigs(
  new Configs(
    protocol match {
    case "AXI4-Lite"    => AXI4LiteWrapperConfig(ConnectConfigs)
    case "AXI4"         => AXI4WrapperConfig(ConnectConfigs)
    case "Simple"       => SimpleWrapperConfig(ConnectConfigs)
    case _ => AXI4WrapperConfig(ConnectConfigs)
  }))
```

Figure 7: Protocol bridge config.

Packer/Unpacker Module. Since the packet sent by the AXI4 channel does not match the defined flit size in the TPNoC flit format, the network interface needs to split the packets into small data flits or reassemble the flits into packets when sending or receiving the data. The Unpacker module splits the packet into head flit/ tail flit/data flit; The packer module accepts flits from the NoC and assembles them into data packets.

4 VERIFICATION AND EVALUATION PHASE

4.1 Verification: System Tester

The TPNoC tester is built based on the test framework chiseltest provided by the Chisel3 project, and the verilator simulator was called by default. The tester of TPNoC has external IP cores models which default use the AXI4 protocol and simulate its behavior of reading and writing data to test the correctness of TPNoC-generated NoCs with the protocol bridge. The master IP core will write a random number to the write data channel of AXI4. Finally, we connect the IP core device and the TPNoC-generated NoC with AXI4 bridge to observe whether the data is sent and received normally.

To demonstrate the testing process, we generate a NoC platform with a 4*4 mesh topology and the AXI4 protocol bridge after the parameterization stage. A schematic diagram of a test instance is shown in Figure 8. We put master IP core 0 on router0, master IP core 1 on router1, slave IP core 0 on router15, and slave IP core 1 on router8. We placed these positions to verify the correctness of the routing algorithm, the maximum transmission delay of the network, and the parallelism of data packets in the case of a 16-core mesh topology. To verify the correctness of system data transmission, we make the master IP core 0/1 model write random data to slave IP core 0/1, and observe whether slave IP core 0/1 successfully receives the data packets sent by the master IP core and whether the response packet of slave cores will be sent successfully to master IP cores.

The verification result is shown in Figure 9. When the clock cycle is 0, the master IP core 0 writes 15 to the write address (aw) channel and forms an "aw" packet. In the next clock cycle, it writes a random number to the write data (w) channel and forms a “w” packet. At the same time, the master IP core 1 writes 8 to the "aw" channel and forms the "aw" packet. In the next clock cycle, writes a random number to the "w" channel and form the "w" packet. In 13 cycles, the slave IP core 0 connected to router 15 successfully received the address packet from the master IP core 0. what's more, the slave IP core 0 successfully forms the corresponding response data packets (br packet). At 27 cycles, the master IP core 0 successfully receives the br packet sent back from slave IP core 0. At the same time, the data packet transmission between the master IP core 1 and the slave IP core 1 is also in progress.

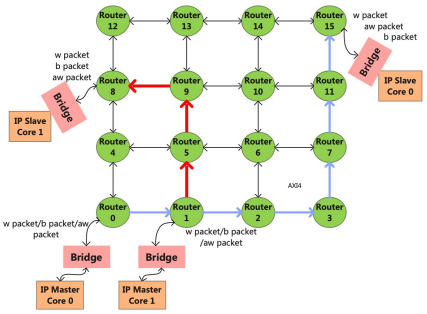


Figure 8: A system-level validation experiment overview.

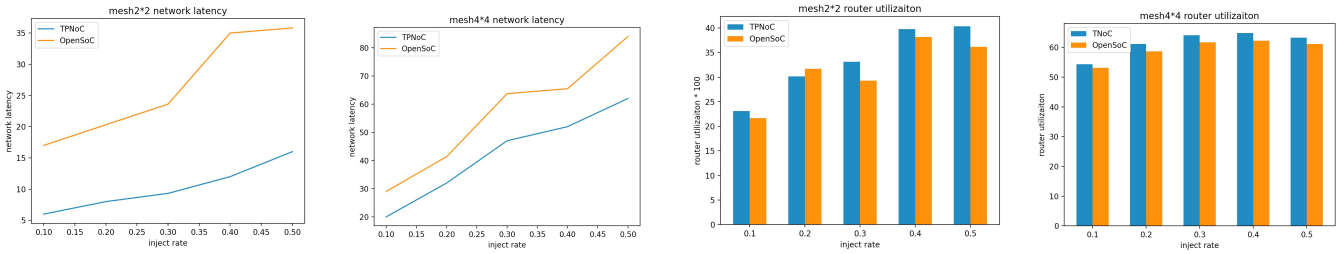
```
✓ Tests passed: 1 of 1 test – 41 sec 450 ms
Elaborating design...
Done elaborating.
Cycle                1: [Master IP core 0] a_packet send,address packet =         15
Cycle                1: [Master IP core 1] a_packet send,address packet =          8
Cycle                2: [Master IP core 0] w_packet send,w packet = 12297829382473034410
Cycle                2: [Master IP core 1] w_packet send,w packet = 12297829382473034410
Cycle               10: [Slave IP core  8]  a_packet receive,aw packet=          8
Cycle               12: [Slave IP core  8] w_packet receive,w packet=12297829382473034410
Cycle               13: [Slave IP core  8] br_packet send, br packet=2360427662797563463613808666
Cycle               13: [Slave IP core 15]  a_packet receive,aw packet=         15
Cycle               15: [Slave IP core 15] w_packet receive,w packet=12297829382473034410
Cycle               16: [Slave IP core 15] br_packet send, br packet=2338893671635677881439354900
Cycle               21: [Master IP core 1] br_packet receive,b pakcet = 2360427662797563463613808666
Cycle               27: [Master IP core 0] br_packet receive,b pakcet = 2338893671635677881439354900
```

Figure 9: Validation result.

4.2 Evaluation Harness: Throughput and Latency

The evaluation stage of TPNoC is a test platform, which can inject flits from each IP port at a user-specified rate under a uniform random traffic pattern to gain the network latency and throughput metrics of the TPNoC-generated NoC to evaluate the performance. The network latency result is calculated by calculating the cycle numbers from ingress NI to egress NI. To compare the throughput, we define the average router utilization as the busy time of each router (flit/cycle/router) to reflect the throughput (flit/node/cycle). The router utilization is measured using a probe module that collects the router's average busy cycle numbers.

To demonstrate the evaluation process, we inject a total of 1024 packets of length 4-flit onto TPNoC-generated NoCs of different sizes in an example test experiment. The size of the NoC generated by TPNoC can be changed quickly in the configuration file. The output is the average latency and router utilization at the corresponding injection rate. For comparison, we implemented the same simulation in the OpenSoC project, extracting network latency results and average router utilization. Figure 10 shows the average latency and utilization results generated by OpenSoC and TPNoC NoC at 2*2 and 4*4 sizes. As the figure shows, TPNoC reduces network latency by an average of 0.604 compared to OpenSoC when the mesh network size is small. As the injection rate increases, the routing utilization of TPNoC and OpenSoC shows an upward trend, and as the network size increases from 2*2 to 4*4, the average delay gap between the two narrows, while the delay of TPNoC is still smaller than that of OpenSoC, and the utilization rate of TPNoC is still higher than that of OpenSoC. The results show that the TPNoC design can be scaled up easily while maintaining performance in terms of throughput and latency.

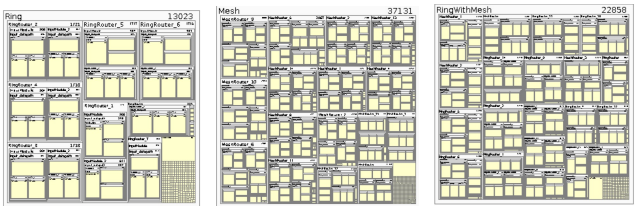


(a) mesh2*2 latency (b) mesh4*4 latency (c) mesh2*2 utilization (d) mesh4*4 utilization

Figure 10: Network latency and utilization results.

5 RTL GENERATION STAGE

When the user's design passed the first two stages, the RTL generation stage can obtain the power consumption or area information of the NoC according to the FPGA flow or ASIC flow. To demonstrate the RTL generation stage, we generated a 4*4 grid NoC RTL, an 8-node ring NoC RTL and a ring (4) grid (2) NoC through the TPNoC parameterization stage and the test evaluation stage, and passed the generated RTL to Xilinx Vivado design flow. Figure 11 shows the layout result.



(a) ring8 (b) mesh4*4 (c) ring(4)mesh(2)

Figure 11: FPGA floorplan results.

6 CONCLUSIONS

In this paper, we proposed TPNoC which is an efficient topology reconfigurable NoC generator with plug play capability framework. Topology reconfigurable, and modular parametric design enables building NoCs from TPNoC for multi/many-core SoC to be much less work than building from scratch. Through parameters, verification evaluation, and RTL generation stages, TPNoC has the ability to significantly reduce the cost of building NoC systems.

ACKNOWLEDGMENTS

This research is supported partly by Science and Technology Commission of Shanghai Municipality Project (22DZ1101100), partly by National Natural Science Foundation of China (NSFC) research projects 62090025, 62141407, 61974032 and 61929102.

REFERENCES

[1] Christof Angermueller, Tanel Pärnamaa, Leopold Parts, and Oliver Stegle. 2016. Deep learning for computational biology. *Molecular systems biology* 12, 7 (2016), 878.

[2] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*. 1216–1225.

[3] Christopher Celio, Pi-Feng Chiu, Borivoje Nikolic, David A Patterson, and Krste Asanovic. 2017. BOOMv2: an open-source out-of-order RISC-V core. In *First Workshop on Computer Architecture Research with RISC-V (CARRV)*.

[4] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM SIGARCH Computer Architecture News* 42, 1 (2014), 269–284.

[5] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits* 52, 1 (2016), 127–138.

[6] Han Cui and Naim Dahnoun. 2019. Real-time stereo vision implementation on Nvidia Jetson TX2. In *2019 8th Mediterranean Conference on Embedded Computing (MECO)*. IEEE, 1–5.

[7] Farzad Fatollahi-Fard, David Donofrio, George Michelogiannakis, and John Shalf. 2016. OpenSoC Fabric: On-chip network generator. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 194–203.

[8] Yatin Hoskote, Sriram Vangal, Arvind Singh, Nitin Borkar, and Shekhar Borkar. 2007. A 5-GHz mesh interconnect for a teraflops processor. *IEEE micro* 27, 5 (2007), 51–61.

[9] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, et al. 2019. Gpipe: Efficient training of giant neural networks using pipeline parallelism. *Advances in neural information processing systems* 32 (2019).

[10] Drago Ignjatović, Daniel W Bailey, and Ljubisa Bajić. 2022. The Wormhole AI Training Processor. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. IEEE, 356–358.

[11] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. 2020. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361* (2020).

[12] Hyoukjun Kwon and Tushar Krishna. 2017. OpenSMART: Single-cycle multi-hop NoC generator in BSV and Chisel. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 195–204.

[13] Doowon Lee, Ritesh Parikh, and Valeria Bertacco. 2014. Brisk and limited-impact NoC routing reconfiguration. In *2014 Design, Automation , Test in Europe Conference , Exhibition (DATE)*. 1–6. https://doi.org/10.7873/DATE.2014.319

[14] Somnath Mazumdar and Alberto Scionti. 2020. Ring-mesh: a scalable and high-performance approach for manycore accelerators. *The Journal of Supercomputing* 76, 9 (2020), 6720–6752.

[15] Jinwook Oh, Sae Kyu Lee, Mingu Kang, Matthew Ziegler, Joel Silberman, Ankur Agrawal, Swagath Venkataramani, Bruce Fleischer, Michael Guillorn, Jungwook Choi, et al. 2020. A 3.0 TFLOPS 0.62 V scalable processor core for high compute utilization AI training and inference. In *2020 IEEE Symposium on VLSI Circuits*. IEEE, 1–2.

[16] Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hobbhahn, and Pablo Villalobos. 2022. Compute trends across three eras of machine learning. In *2022 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

[17] Hao Zhang, Itta Ohmura, and Makoto Taiji. 2020. Implementing a comprehensive networks-on-chip generator with optimal configurations. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 420–421.

[18] Jerry Zhao, Animesh Agrawal, Borivoje Nikolic, and Krste Asanović. 2022. Constellation: An Open-Source SoC-Capable NoC Generator. In *2022 15th IEEE/ACM International Workshop on Network on Chip Architectures (NoCArc)*. IEEE, 1–7.