論文 / 著書情報
Article / Book Information

題目(和文)	
Title(English)	Hardware-Accelerated Modeling of Large-Scale Networks-on-Chip
著者(和文)	Chu Van Thiem
Author(English)	Thiem Van Chu
出典(和文)	学位:博士(工学), 学位授与機関:東京工業大学, 報告番号:甲第10994号, 授与年月日:2018年9月20日, 学位の種別:課程博士, 審査員:吉瀬 謙二,横田 治夫,宮崎 純,渡部 卓雄,金子 晴彦
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第10994号, Conferred date:2018/9/20, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

Hardware-Accelerated Modeling of Large-Scale Networks-on-Chip

by

Thiem Van Chu

A dissertation submitted in partial fulfillment of the requirements for
the degree of Doctor of Engineering

Department of Computer Science
Graduate School of Information Science and Engineering
Tokyo Institute of Technology

Abstract

Networks-on-Chip (NoCs) are becoming increasingly important elements in different types of computing hardware platforms, from general-purpose many-core processors for supercomputers and datacenters to application-specific MultiProcessor Systems-on-Chip (MPSoCs) for embedded applications. They are also integral parts of many emerging accelerators for critically essential applications such as deep neural networks, databases, and graph processing. In such a hardware platform, the NoC is responsible for connecting the other components together and thus has a significant impact on the overall performance. To achieve higher performance and better power efficiency, many-core processors with more and more cores have been developed. For the similar reason and to meet the increasingly stringent requirements of target applications, the number of processing elements, memory and input/output modules integrated on an MPSoC/accelerator is increasing. As the number of components that need to be interconnected increases, the overall performance becomes highly sensitive to the NoC performance. Therefore, research and development of NoCs play a key role in designing future large-scale architectures with hundreds to thousands of components.

A major obstacle to research and development of large-scale NoCs is the lack of fast modeling methodologies that can provide a high degree of accuracy. Analytical models are extremely fast but may incur significant inaccuracy in many cases. Thus, NoC designers often rely on simulation to test their ideas and make design decisions. Unfortunately, while being much more accurate than analytical modeling, conventional software simulators are too slow to simulate large-scale NoCs with hundreds to thousands of nodes in a reasonable time. Because of this, most studies are limited to NoCs with around 100 nodes. To address the simulation speed problem, there have been some attempts to build NoC emulators using Field-Programmable Gate Arrays (FPGAs). However, these NoC emulators suffer from the scalability problem. They cannot scale to large NoCs due to the FPGA logic and memory constraints. A recent study has shown that even an extremely large FPGA does not have enough logic blocks to fit a moderately complex NoC design of around 150 nodes. What is even worse is that emulating a large-scale NoC also requires a large amount of memory for modeling of traffic workloads. However, the on-chip memory capacity of an FPGA is very small, at most from several to around only ten megabytes. Off-chip memory

(usually DRAM) has a larger capacity but is much slower than on-chip memory. The use of off-chip memory may substantially degrade the emulation speed.

This dissertation proposes methods for fast and accurate modeling of NoCs with up to thousands of nodes by FPGA emulation with cycle accuracy, an extremely high degree of emulation accuracy in which target NoCs are emulated on a cycle-by-cycle basis. While the goal of these methods is to enable fast and accurate modeling of large-scale NoCs, they are also beneficial to the modeling of current NoCs with tens to around 100 nodes.

To overcome the FPGA logic constraints, the dissertation proposes a novel use of time-division multiplexing (TDM) where the emulation cycle is decoupled from the FPGA cycle and a network is emulated by time-multiplexing a small number of nodes. This approach makes it possible to emulate NoCs with up to thousands of nodes using a single FPGA. The dissertation focuses on applying the TDM technique to two commonly used network topologies, two-dimensional (2D) mesh and fat-tree (k-ary n-tree), which are the bases of almost all actually constructed network topologies. It thus can be expected that the proposed methods can be extended for a wide range of networks.

While the time-division multiplexing methods enable the emulation of large-scale NoCs, they alone are not sufficient. To achieve a high emulation speed, it is essential to address the memory constraints caused by modeling traffic workloads.

There are two types of workloads used in NoC emulation: synthetic workloads and trace-driven workloads. Synthetic workloads are those based on mathematical modeling of common traffic patterns in real applications. They have a high degree of flexibility and are easy to create. A set of carefully designed synthetic workloads can provide a relatively thorough coverage of the characteristics of the target NoCs. It has also been shown that evaluation on synthetic workloads is indispensable in many cases. For instance, when designing a routing algorithm, the use of synthetic workloads is mandatory for assessing the algorithm on possible corner cases like those under extremely high loads. On the other hand, trace-driven workloads are those based on trace data captured from either a working system or an execution-driven simulation/emulation. They are effective for evaluating target NoCs under the intended applications.

Currently, due to the lack of trace data of large-scale NoC-based systems, using synthetic workloads is practically the only feasible approach for emulating large-scale NoCs with thousands of nodes. To overcome the memory constraints caused by modeling synthetic workloads, the dissertation proposes a method to reduce the amount of required memory so that it is not necessary to use off-chip memory even when emulating NoCs with thousands of nodes. This method not only makes the overall design much simpler but also significantly contributes to the improvement of emulation speed. It and the proposed time-multiplexed emulation methods enable a NoC emulator which can be used to model a mesh-based NoC with 16,384 nodes (128×128 NoC) and a fat-tree-based NoC with 6,144 switch nodes and 4,096 terminal nodes (4-ary 6-tree NoC) and

is up to three orders of magnitude faster than a widely used cycle-accurate software simulator while providing the same results.

The dissertation shows the usability of the developed emulator by designing and modeling an effective routing algorithm for 2D mesh NoCs and evaluating it for various network sizes, from 8×8 to 128×64. The proposed routing algorithm has an oblivious routing scheme and thus a low design complexity. It, however, can achieve high performance by properly distributing the load over two network dimensions and using an efficient deadlock avoidance method. Because of the lack of fast modeling methodologies that can provide a high degree of accuracy, most existing routing algorithms have been evaluated in NoCs of limited size. The developed FPGA-based NoC emulator enables the evaluation in large-scale NoCs with thousands of nodes in a practical time. The evaluation results show that, in the currently common NoC sizes of around 100 nodes, the proposed algorithm significantly outperforms other popular oblivious routing algorithms and can provide comparable performance to a complicated adaptive routing algorithm. However, as the NoC size increases, the performance of the algorithms is strongly affected by the resource allocation policy in the network and the effects are different for each algorithm. This result would not be obtained if modeling of large-scale NoCs could not be performed.

While synthetic workloads can provide a relatively thorough coverage of the characteristics of the emulated NoCs, evaluation on trace-driven workloads is still required in some cases such as assessing some application-specific optimizations. The dissertation takes this into account and extends the proposed NoC emulator to support trace-driven emulation which will be useful for research and development of large-scale NoCs in the future when trace data of large-scale NoC-based systems are available. Since trace data are large, they must be stored in off-chip memory. The dissertation proposes an effective trace data loading architecture and some methods to hide the off-chip memory access time and improve the scalability of the emulation architecture in terms of operating frequency and logic requirements. These proposals are tightly coupled to the time-multiplexed emulation methods. The evaluation results show that the extended NoC emulator is two orders of magnitude faster than the above-mentioned software simulator when emulating an 8×8 NoC with the widely used PARSEC traces while also providing the same results; and the speedup is increased to three orders of magnitude when emulating a 64×64 NoC with trace data created based on a synthetic workload.

The work in this dissertation contributes directly to the formation of infrastructures for research and development of large-scale NoCs, which is crucial for developing more powerful and efficient many-core processors, MPSoCs, and hardware accelerators in the future.

Acknowledgment

The work presented in this thesis would not have been possible without the help and support of so many people to whom I owe a lot of gratitude.

First and foremost, I would like to thank my advisor, Associate Professor Kenji Kise, who has given me this great opportunity to come and work with him at Tokyo Tech. He taught me how to do good research and effectively communicate my ideas to others. He also generously provided the resources and the comfortable environment that enabled me to pursue my ideas to fruition. His passion, hard work, and dedication to the success of his students are truly inspiring. I could not wish for a better advisor.

I would like to thank Professor Haruo Yokota, Professor Jun Miyazaki, Professor Takuo Watanabe, and Associate Professor Haruhiko Kaneko for serving on my defense. Their comments and feedback on the thesis were immensely helpful.

I would also like to thank Assistant Professor Shimpei Sato for his advice and inspiration. He always encouraged me to explore and challenge further. It has been a pleasure to collaborate and co-author with him. I will miss our mid-night discussions.

I am grateful to Ryosuke Sasakawa for proposing the original idea from which I developed the LEF routing algorithm in this thesis. I also would like to thank Shi FA and Myeonggu Kang for their contributions in the development of the LEF routing algorithm.

I was fortunate to have had an opportunity to work together with Professor Tomohiro Yoneda at the National Institute of Informatics and Professor Masashi Imai at Hirosaki University in the project of comparing synchronous and asynchronous Networks-on-Chip. They provided valuable feedback and perspective on my work. This experience makes me feel that doing research with researchers with different perspectives is an extremely good opportunity to enhance my ability.

I would like to thank all the former and current members of my research laboratory. I would like to give special thanks to Associate Professor Shinya Takamaeda-Yamazaki, Assistant Professor Ryohei Kobayashi, Susumu Mashimo, and Tomohiro Misono. Associate Professor Shinya Takamaeda-Yamazaki has made valuable comments on my work. Assistant Professor Ryohei Kobayashi has helped me a lot since I first came to the laboratory. I really enjoyed discussing with Susumu Mashimo. We had great times at two conferences, FCCM 2017 and MICRO 2017, in the United States. He and Tomohiro Misono also carefully reviewed my written Japanese.

I would like to thank Yukiko Asoh and the other assistants in our department for handling all the administrative duties that made things run smoothly for all the students. My abroad business trips would not be possible without them.

I would like to acknowledge the Japan Society for the Promotion of Science for providing the generous funding that supported my work.

I am so blessed to have had truly good friends to lean on in both good and bad times. Thank you, Giang, Viet, Nam, and Quang Anh, for always being there for me.

Finally, I would like to express my gratitude to my family. Their love, support, and encouragement have been an important part of my study at Tokyo Tech.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

1.1.1 The Advent of New Computing Architectures

Over the past several decades, the continuous advances in semiconductor process technology according to Moore's Law [8] coupled with Dennard scaling [9] have been a fundamental driver for improving computing performance. Moore's Law states that the number of transistors that can be cost-effectively integrated on a chip doubles every 24 months, which has been achieved by shrinking the transistors. Dennard scaling observes that transistors consume less power and can switch faster as they get smaller. Therefore, after each generation of technology scaling, with the same cost and power budget, computer architects had more transistors which were faster than the previous generation to be able to incorporate new architectural techniques. This resulted in the exponential improvement of integrated circuit performance. For instance, as shown in Figure 1.1, the processor performance was improved 6,043× from 1978 to 2003 with the average annual growth rate of 22% in the first six years and 52% in the remaining years.

However, recent trends have made it harder to continue to scale computing performance in the conventional way. Dennard scaling has broken down since the middle of the 2000s because the current leakage has become a serious problem as the transistor size decreases. Moore's Law is also reaching its limit. This slowdown in technology scaling is leading to serious consequences. As shown in Figure 1.1, the average annual growth rate of processor performance from 2003 to 2011 was 23% compared to 52% in the previous 17 years. It is further decelerated to 12% in the period from 2011 to 2015 and just 3.5% in recent years.

Nevertheless, the need for computing performance improvement is becoming higher and higher. New technologies like social networking services and Internet of Things devices are resulting in the ever-increasing volume of data that need to be processed. Techniques making use of these data for creating new values such as deep learning [10] also require tremendous

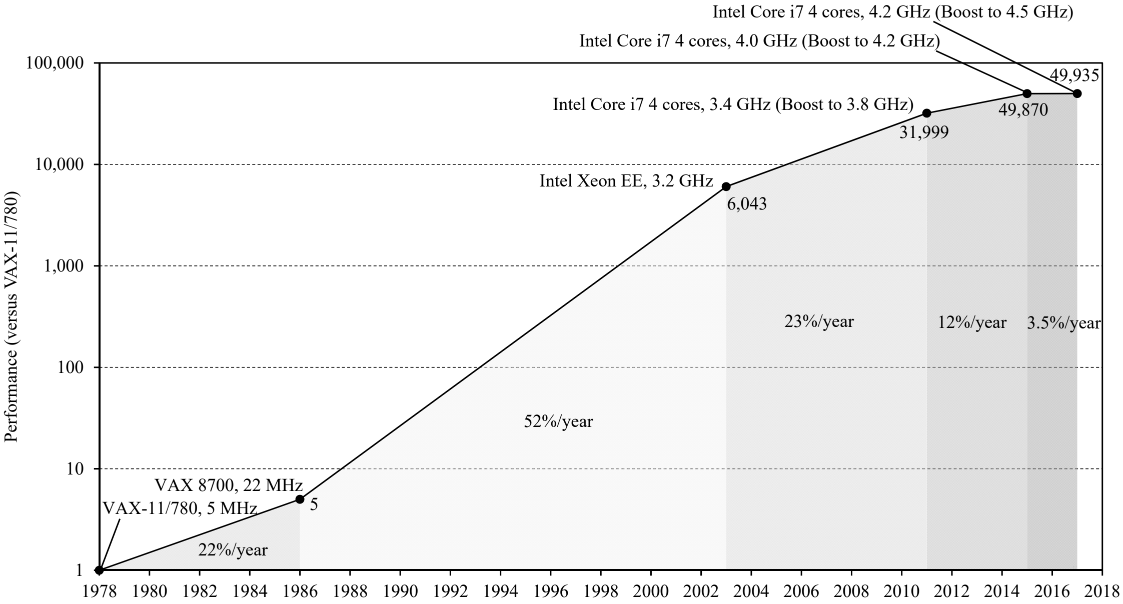


Figure 1.1: Growth in processor performance from 1978 to 2017 (Figure adapted from [1]). The data are collected by running the SPEC integer benchmarks [2] and normalized to the performance of the VAX-11/780 [3].

computing power. Apart from that, many large problems in science and engineering can only be solved if we have much more computational resources than at this time.

In response to the impending end of the benefits of technology scaling and the pressing need for more computing power, we need significant innovation in computing architecture. Two complementary approaches have been shown to be effective: increasing the number of processing elements on a chip and domain/application-specific specialization. Using these approaches, three different types of hardware platforms have been developed: multi/many-core processors, MultiProcessor Systems-on-Chip (MPSoCs), and hardware accelerators.

The microprocessor industry has shifted to integrating multiple processor cores on a chip since the middle of the 2000s. Most processors used in smartphones and personal computers today are multi-core processors with two to around ten cores. Many-core processors with more number of cores have also been developed for use in larger-scale systems like supercomputers and datacenters. For instance, the Intel Xeon Phi Knights Landing many-core processor [11] released in 2016 has up to 72 cores. Sunway TaihuLight, the world's fastest supercomputer as of November 2017, is based on a many-core processor with 260 cores [12]. To achieve higher performance and better power efficiency, many-core processors with more and more cores are being studied at present.

MPSoCs [13] are Systems-on-Chip (SoCs) which have been designed for embedded applications in various domains such as communications, multimedia, networking, and signal processing. An MPSoC often contains a collection of processing elements of multiple types (general-purpose processor cores or domain-specific processing units) which communicate and cooperate to perform the execution of tasks of the target application. Like general-purpose many-core processors, to achieve higher performance, better power efficiency and meet the increasingly stringent requirements of target applications, modern MPSoCs are growing in complexity with more and more processing elements.

Accelerators are hardware units specialized for specific computational tasks. The use of accelerators started in the 1980s with the deployment of floating-point co-processors. Accelerators have also been adopted in designing SoCs including MPSoCs. However, the wide adoption of accelerators in different types of computing systems has not been started until very recently when the slowdown in technology scaling made it hard to meet the increasingly high demand for computing power. There have been numerous accelerators proposed recently for speeding up different tasks in machine learning, databases, graph processing, networking processing, and a variety of other applications [14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35]. Some have even been deployed in commercial systems like Google Tensor Processing Unit [14] and Microsoft's FPGA accelerators [15, 17, 16]. Like in MPSoCs, many accelerators are comprised of multiple processing elements interacting with each other, and they are growing in complexity.

1.1.2 Interconnection Problems

As presented in Section 1.1.1, modern many-core processors, MPSoCs, and many hardware accelerators are composed of multiple processing elements that communicate and cooperate with each other when executing a task. The processing elements also interact with the other components of the system such as the memory and input/output (I/O) modules. The latency and bandwidth of the interconnection between these components are highly critical to the overall performance, especially when the number of components is large.

1.1.2.1 Problems of Traditional On-Chip Interconnection Architectures

The simplest way to connect a set of components together is to use a bus. A key property of buses is *broadcast*: a message transmitted by a node over the bus is received by all other nodes. Also, only one node can be the transmitter at any given time. With these properties, buses have been widely used as the interconnects of small-scale multi-core processors, MPSoCs, and hardware accelerators because of their simplicity and effectiveness. However, as the number of components that need to be interconnected increases, it is hard to make the bus operate at a high speed. Moreover, it is hard to provide enough communications bandwidth since only one message can be sent over the bus at any given time.

Point-to-point interconnects have been used in cases that the required communications band-

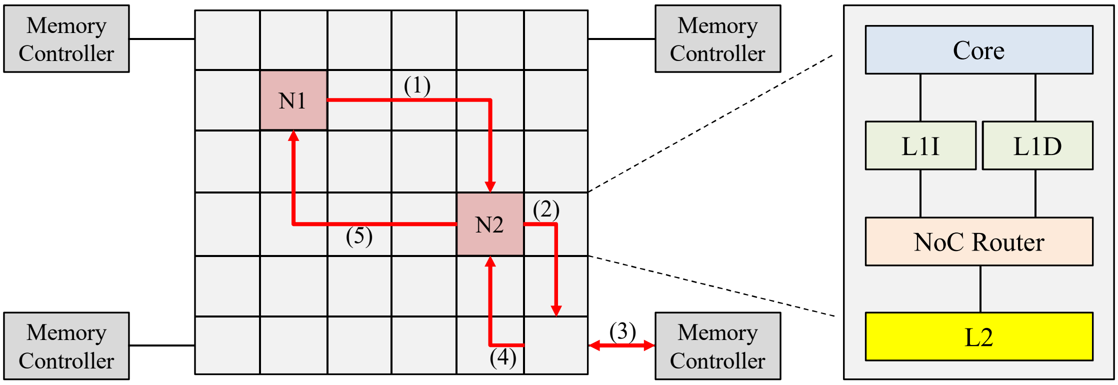


Figure 1.2: An example of a many-core processor architecture with 2D-mesh-based NoC.

width is higher than what a bus can provide. In a point-to-point interconnect, there is a direct link between any pair of nodes. Thus, a high communications bandwidth can be achieved since the transmission between a pair of nodes is not affected by other pairs. However, as the number of components that need to be interconnected increases, the implementation costs (hardware and power requirements) of point-to-point interconnects become prohibitive.

Crossbars are other alternatives to address the communications bandwidth problem of buses. Compared to point-to-point interconnects, the implementation costs of crossbars are cheaper. However, they are still too high when the number of interconnected components is large. Specifically, the cost of a crossbar grows quadratically with the number of I/O ports.

In summary, all of the traditional on-chip interconnection architectures are not scalable. They cannot keep pace with the number of components that need to be interconnected. Therefore, a new class of on-chip interconnects called Networks-on-Chip (NoCs) has been developed [36, 4, 37, 38].

1.1.2.2 Network-on-Chip (NoC)

It has been shown that NoCs can address both the communications performance and implementation cost problems of the traditional on-chip interconnection architectures. They thus have been used as the interconnects of most of modern many-core processors, MPSoCs, and many hardware accelerators [11, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 18, 23, 24, 25, 26, 20, 31].

A NoC is composed of a collection of routers interconnected with each other according to a predetermined topology. To explain how a NoC can be integrated into a system, let us look at an example below.

Figure 1.2 shows an example of a many-core processor architecture with two-dimensional (2D)-mesh-based NoC. This architecture is close to the recently released Intel Xeon Phi Knights Landing [11] and OpenPiton [40, 41] architectures. Each node contains a processor core, a private L1 cache (separated into L1I and L1D), a bank of L2 cache, and a NoC router. The routers of two adjacent nodes are interconnected to each other. L2 is a shared cache and is divided into banks that are distributed among nodes. The organization of L2 is assumed to be Statically-mapped Non-Uniform Cache Architecture (S-NUCA). Specifically, the mapping of data into L2 banks is pre-determined, based on their physical addresses. Figure 1.2 shows how handling an L1 miss at node N1 involves the use of the NoC. Here we assume that the home L2 bank of the data requested is in node N2. The handling steps are as follows. First, a miss request is sent from node N1 to node N2 (step (1)). If hit in the home L2 bank at node N2, then the data are sent back to the L1 cache at node N1 (step (5)). Otherwise, an L2 miss request is sent to the target memory controller (steps (2) and (3)). The memory controller loads the requested data and sends back to the home L2 bank (step (4)). Finally, the data are sent back to the L1 cache from the home L2 bank (step (5)). The request/response messages between node N1, node N2, and the memory controller are transmitted over the NoC.

1.1.3 Challenges in NoC Design

The example in Figure 1.2 described in Section 1.1.2.2 suggests that the NoC performance is highly critical to the overall performance. In [52], Sanchez *et al.* quantitatively show that, in a many-core processor with 128 multithreaded cores, the NoC is responsible for 60% to 75% of the miss latency and thus has a significant impact on the overall performance. The authors also find that improving the NoC can help to boost the overall performance by up to 20% and that the impact of the NoC becomes more pronounced as the number of cores increases. Therefore, research and development of NoCs play a key role in designing future large-scale architectures with hundreds to thousands of cores.

However, a major obstacle to research and development of large-scale NoCs is the lack of fast modeling methodologies that can provide a high degree of accuracy. Analytical models like those proposed in [53, 54] are extremely fast but may incur significant inaccuracy in many cases. Thus, NoC designers often rely on simulation, which can provide much more accurate evaluation results and insights into the designs, to test their ideas and make design decisions.

Software simulators including full-system simulators such as gem5 [55] and stand-alone NoC simulators such as BookSim [5] have been widely used in the NoC research community. Although these simulators offer various advantages such as the flexibility and the broad range of programming tools, they suffer from two serious drawbacks. First, it is hard to validate a complex design because various types of errors may be introduced during specifying, abstracting, and implementing certain details [56]. One can design and implement a feature in a way that would be impractical to implement in hardware. Second, and more important, the simulators are too slow to simulate large-scale designs with hundreds to thousands of nodes in a reasonable time. Sanchez *et al.* [57] mentioned that it would take almost a year for gem5 to simulate

1s of a thousand-core chip. Compared to full-system simulators, stand-alone NoC simulators, which often support much more detailed NoC models, are faster, but simulating a large-scale NoC still requires an excessive amount of time. Because of this, most previous studies are limited to NoCs with around 100 nodes. To investigate novel designs with hundreds to thousands of nodes, it is crucial to improve the simulation speed while maintaining the simulation accuracy. Unfortunately, using existing parallelization techniques to improve the simulation speed without sacrificing the simulation accuracy is hard [58].

To address the simulation speed problem, there have been some attempts to build NoC emulators using Field-Programmable Gate Arrays (FPGAs) [59, 60, 61, 62, 63, 64, 65, 66, 67, 68]. However, these NoC emulators suffer from the scalability problem. They cannot scale to large NoCs due to the FPGA logic and memory constraints.

A recent study [69] has shown that even an extremely large FPGA does not have enough logic blocks to fit a moderately complex NoC design of around 150 nodes. Some prior studies simplify the router models and sacrifice the cycle accuracy, a level of emulation accuracy in which target NoCs are emulated on a cycle-by-cycle basis, to reduce the number of required logic blocks. For instance, DART [65] uses a simple single-stage NoC router with only one output channel to model some multi-stage pipelined NoC routers with five output ports. However, Khan [70] has shown empirical evidence that using a simplified emulation model or sacrificing the cycle accuracy may lead to conclusions that are wrong both quantitatively and qualitatively. Thus, it is necessary to preserve both of them.

Besides the challenge of a large number of required logic blocks, emulating a large-scale NoC also requires a large amount of memory for modeling of traffic workloads. However, the on-chip memory capacity of an FPGA is very small, at most from several to around only ten megabytes. Off-chip memory (usually DRAM) has a larger capacity but is much slower than on-chip memory. The use of off-chip memory may substantially degrade the emulation speed.

There are two types of workloads used in NoC emulation: synthetic workloads and trace-driven workloads. Synthetic workloads are those based on mathematical modeling of common traffic patterns in real applications. They have a high degree of flexibility and are easy to create. A set of carefully designed synthetic workloads can provide a relatively thorough coverage of the characteristics of the target NoCs. It has also been shown that evaluation on synthetic workloads is indispensable in many cases. For instance, when designing a routing algorithm, the use of synthetic workloads is mandatory for assessing the algorithm on possible corner cases like those under extremely high loads. On the other hand, trace-driven workloads are those based on trace data captured from either a working system or an execution-driven simulation/emulation. They are effective for evaluating target NoCs under the intended applications.

To evaluate a NoC on a synthetic workload, it is necessary to use *open-loop measurements* [4] in which a large FIFO buffer called *source queue* is placed after each packet source to decouple the traffic generation and injection process from the NoC. If the source queues are not large enough, then the traffic generation and injection process will be affected by the NoC, and therefore the workload produced by the traffic generators will not be the one originally specified. Because of this, emulating large-scale NoCs under synthetic workloads on FPGAs requires a large amount of memory. Although off-chip memory can be used, the emulation speed may be degraded substantially since accessing off-chip memory takes much more time than on-chip memory.

In the case of emulation with trace-driven workloads, trace data are often much larger than the total capacity of FPGA on-chip memory and thus must be stored in off-chip memory. Most of the existing FPGA-based NoC emulators simplify the control of loading trace data from the off-chip memory, generating messages based on the loaded trace data, and injecting the messages to the NoC by using soft processors like Microblaze or hard processors on SoC FPGAs. The processors are also responsible for manipulating the emulation and making sure that there is no timing error. This approach makes the implementation easy but the emulation speed is drastically reduced with increasing the NoC size [68].

1.2 Thesis Contributions

This thesis proposes methods to overcome the challenges in the FPGA emulation approach described in Section 1.1.3, thereby enabling fast and accurate modeling of NoCs with hundreds to thousands of nodes. While the goal of these methods is to enable fast and accurate modeling of large-scale NoCs, they are also beneficial to the modeling of current NoCs with tens to around 100 nodes.

Novel time-division multiplexing methods for NoC emulation: The thesis proposes a novel use of time-division multiplexing (TDM) where the emulation cycle is decoupled from the FPGA cycle and a network is emulated by time-multiplexing a small number of nodes. This approach helps to overcome the FPGA logic constraints, thereby enabling emulation of large-scale NoCs with hundreds to thousands of nodes using a single FPGA. Although the TDM technique has been adopted in some prior work [60, 63, 65, 67, 68], it has not been discussed thoroughly. The implementation methods in these studies are not scalable in terms of both FPGA logic requirements and emulation speed. Moreover, they consider only some typical direct networks including 2D meshes, 3D tori, hypercubes, and fully connected networks. These direct networks have regular physical arrangements that are intuitively matched to Very-Large-Scale Integration (VLSI) packaging constraints and thus have been employed in many practical systems. On the other hand, indirect networks such as fat-trees have an attractive feature of lower hop counts that imply lower packet transmission delay and have also been widely adopted. Thus, it is highly desirable that indirect network topologies be supported. However, since the connectivity pattern of nodes in an indirect network is generally much more complicated than that in a direct network, using the TDM technique to emulate indirect networks is not trivial. This thesis discusses in detail methods for efficiently applying the TDM technique for both direct and indirect network topologies with the focus on 2D meshes and fat-trees (k-ary n-trees). Because these two network topologies are the bases of almost all actually constructed network topologies [4], it can be expected that the proposed methods can be extended for a wide range of networks.

While the proposed TDM methods enable the emulation of large-scale NoCs, they alone are not sufficient. To achieve a high emulation speed, it is essential to address the memory constraints caused by modeling traffic workloads.

An efficient method for modeling of synthetic workloads: Currently, due to the lack of trace data of large-scale NoC-based systems, using synthetic workloads is practically the only feasible approach for emulating large-scale NoCs with thousands of nodes. The thesis describes how to emulate a NoC under a synthetic workload without requiring a large amount of memory by (1) decoupling the time counter of each packet source from that of the network and (2) properly allowing the network to operate interactively with the packet sources based on the status of the source queues and the relationship between the time counters. This approach makes the emulation take place as if infinite source queues could be used at the expense of stalling the network in some cases. Although the emulation is slowed down in cases of stalling the network, the memory footprint of the emulation is bounded regardless of the offered traffic load. Thus, we can use only FPGA on-chip memory to implement the source queues even when emulating large-scale NoCs. The thesis also presents methods to minimize the number of times the network is stalled.

FNoC – An emulator for NoC emulation under synthetic workloads: Using the proposed time-multiplexed emulation methods and the method for modeling of synthetic workloads, the thesis develops a NoC emulator, called FNoC, on a Virtex-7 XC7VX485T FPGA. The main evaluation results are as follows:

- The size of the largest NoC that can be emulated by FNoC depends on only the on-chip memory capacity of the FPGA used. The thesis demonstrates the emulations of NoCs of various sizes, up to 128×128 (16,384 nodes) for the 2D mesh topology and 4-ary 6-tree (6,144 switch nodes and 4,096 terminal nodes) for the fat-tree topology.
- When emulating a 128×128 NoC and a 4-ary 6-tree NoC under a synthetic workload, FNoC is, respectively, 5,047× and 232× faster than BookSim [5], one of the most widely used software-based NoC simulators.
- Extensive experimentation shows that FNoC and BookSim report exactly the same results in every case. This indicates that the NoCs modeled by FNoC are totally identical to those of BookSim.

A use case of FNoC in design and modeling of a new routing algorithm: The thesis shows the usability of FNoC by designing and modeling an effective routing algorithm, called LEF, for 2D mesh NoCs and evaluating it for various network sizes, from 8×8 to 128×64. LEF is developed based on the idea which is originally proposed in [71]. LEF has an oblivious routing scheme and thus a low design complexity. It, however, can achieve high performance by properly distributing the load over two network dimensions and using an efficient deadlock avoidance method. Because of the lack of fast modeling methodologies that can provide a high degree of accuracy, most existing routing algorithms have been evaluated in NoCs of limited size. FNoC enables the evaluation in large-scale NoCs with thousands of nodes in a practical time. LEF is evaluated against O1TURN [7], one of the best oblivious routing algorithms for 2D meshes, and a complicated adaptive routing algorithm based on the odd-even turn model [72]. The evaluation results show that, in an 8×8 NoC, LEF provides 3.6%–10% higher throughput than O1TURN and comparable performance to the adaptive routing algorithm under three different traffic patterns. LEF is particularly effective when the network is asymmetric. In a 16×8 NoC, LEF outperforms the adaptive routing algorithm and delivers up to 28.3% higher throughput than O1TURN. However, as the NoC size increases, the performance of the algorithms is strongly affected by the resource allocation policy in the network and the effects are different for each algorithm. This result would not be obtained if modeling of large-scale NoCs could not be performed.

Towards NoC emulation under future scenarios with trace-driven workloads: While synthetic workloads can provide a relatively thorough coverage of the characteristics of the emulated NoCs, evaluation on trace-driven workloads is still required in some cases such as assessing some application-specific optimizations. The thesis takes this into account and extends FNoC to support trace-driven emulation which will be useful for research and development of large-scale NoCs in the future when trace data of large-scale NoC-based systems are available. Since trace data are large, they must be stored in off-chip memory. The thesis proposes an effective trace data loading architecture which does not include processors for speeding up trace-driven emulation of NoCs with up to thousands of nodes. The thesis also introduces some methods to effectively hide the off-chip memory access time and improve the scalability of the emulation architecture in terms of operating frequency and FPGA resource requirements. These proposals are tightly coupled to the time-multiplexed emulation methods. The evaluation results show that the extended NoC emulator achieves a speedup of 260× compared to BookSim when emulating an 8×8 NoC with the PARSEC traces [73] collected by Hestness *et al.* [74] while also providing the same results; and the speedup is increased to 5,106× when emulating a 64×64 NoC with trace data created based on a synthetic workload.

The work in this thesis contributes directly to the formation of infrastructures for research and development of large-scale NoCs, which is crucial for developing more powerful and efficient many-core processors, MPSoCs, and hardware accelerators in the future.

1.3 Thesis Organization

The rest of this thesis is organized as follows.

Chapter 2 provides relevant background in NoC, FPGA, and NoC modeling. This chapter also reviews the efforts that have been made over the years in NoC modeling by both software simulation and FPGA emulation and discusses the approach of the thesis. The chapter ends by presenting the NoC emulation model adopted in the thesis.

Chapter 3 proposes novel time-division multiplexing methods to overcome the FPGA logic constraints, thereby enabling emulation of large-scale NoCs with hundreds to thousands of nodes using a single FPGA. The chapter discusses in detail architectures and methods for supporting both direct and indirect network topologies with the focus on 2D meshes and fat-trees (k-ary n-trees). The chapter also describes the basic rules for writing time-multiplexed emulation Register-Transfer Level (RTL) code. The content of this chapter is largely based on the author's published work [75, 76, 77, 78].

Chapter 4 proposes a method for addressing the memory constraints in modeling synthetic workloads that are presently the only feasible workloads for emulating large-scale NoCs with thousands of nodes. The proposed method reduces the amount of required memory so that it is not necessary to use off-chip memory even when emulating NoCs with thousands of nodes. This not only makes the overall design much simpler but also significantly contributes to the improvement of emulation speed since the use of slow off-chip memory is avoided. The chapter also presents a NoC emulator that is built based on the combination of the method for modeling of synthetic workloads and the time-multiplexed emulation methods proposed in Chapter 3. The content of this chapter is largely based on the author's published work [76, 77, 78].

Chapter 5 shows the usability of the NoC emulator proposed in Chapter 4 by designing and modeling an effective routing algorithm for 2D mesh NoCs and evaluating it for various network sizes, from 8×8 to 128×64. The content of this chapter is largely based on the author's published work [79].

Chapter 6 extends the NoC emulator proposed in Chapter 4 to support trace-driven emulation which will be useful for research and development of large-scale NoCs in the future when trace data of large-scale NoC-based systems are available. The chapter proposes an effective trace data loading architecture and some methods to hide the off-chip memory access time and improve the scalability of the emulation architecture in terms of operating frequency and FPGA resource requirements. These proposals are tightly coupled to the time-multiplexed emulation methods introduced in Chapter 3. The chapter also presents the evaluation of the extended NoC emulator. The content of this chapter is largely based on the author's published work [80].

Chapter 7 summarizes the thesis and discusses the future work.

Chapter 2

Background and Related Work

2.1 NoC Basics

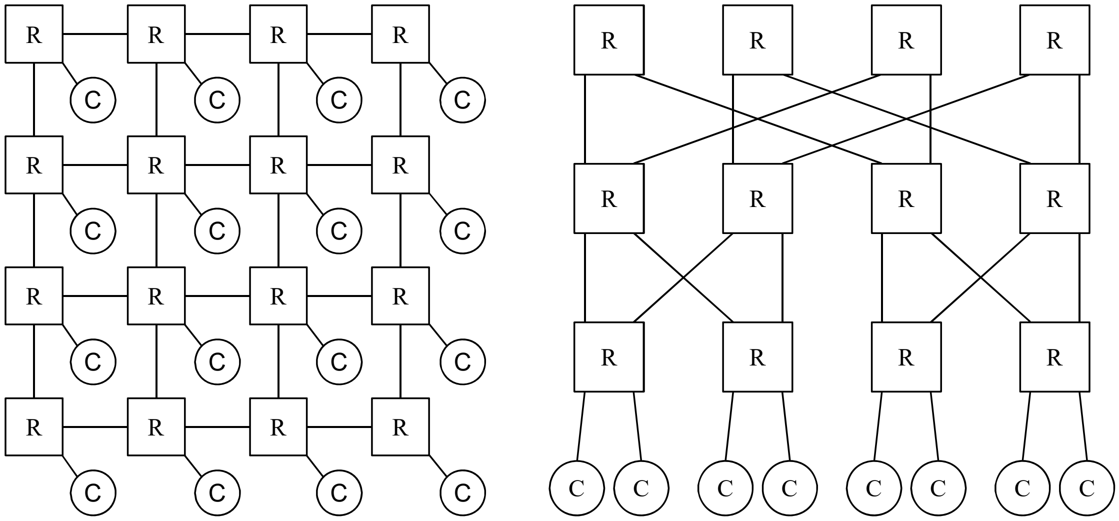
Any NoC architecture can be characterized by four properties: topology, routing, flow control, and router architecture [4, 38]. In this thesis, the components that are interconnected by a NoC in a many-core system (many-core processor, MPSoC, or hardware accelerator) are collectively called *cores*. A core can be a processing element, a memory module, or an I/O module.

2.1.1 Topology

Topology defines how channels and nodes are arranged in a network. Because topology heavily affects the design of the other properties, choosing a topology is usually the first step in designing a NoC. The topology establishes an optimal bound on performance, that is, throughput and latency, of a network. The routing algorithm, flow control protocol, and router architecture determine how closely the optimal performance bound can be approached.

Direct network topology versus indirect network topology: In direct networks such as meshes and tori, each node behaves as both a terminal node and a switch node and thus is composed of a core connected with a router. On the other hand, indirect networks such as butterflies (e.g., flattened butterfly [81]) and trees (e.g., fat-tree [82], Fat H-Tree [83]) distinguish between terminal node and switch node. Specifically, each terminal node in an indirect network is a core while each switch node is a router.

This thesis studies two network topologies: 2D mesh and k-ary n-tree (a commonly used topology in the fat-tree family). Figure 2.1 illustrates the position of routers, cores, and links between routers in each topology. Table 2.1 provides a comparison of the two topologies for three key metrics. The router radix here is defined as the number of ports per router. It can be used as an indicator of the cost of a network. In general, a network with higher router radix is more expensive to implement because it requires more area and energy at each router.



(a) 2D mesh (4×4)

(b) k-ary n-tree ($k = 2, n = 3$)

Figure 2.1: The position of routers (R), cores (C), and links between routers in each of the two network topologies studied in this thesis.

Table 2.1: Comparison of the two network topologies studied in this thesis

	Type	#cores	#routers	Router radix$^{(1)}$
2D mesh ($x \times y$)	Direct network topology	xy	xy	$5^{(2)}$
k-ary n-tree	Indirect network topology	k^n	nk^{n-1}	$2k^{(3)}$

$^{(1)}$: the number of ports per router.

$^{(2)}$: some ports of the routers at the edge are not used.

$^{(3)}$: some ports of the routers at the root of the tree are not used.

An $x \times y$ 2D mesh network comprises xy nodes, each has one core and one radix-5 router[1]. Note that some ports of the routers at the edge of the mesh are not used. These routers can be simplified and thus require less area and energy than the others.

A k-ary n-tree consists of k^n cores (terminal nodes) and n levels of k^{n-1} routers (switch nodes). The router radix in this network is $2k$. Thus, when $k \geq 3$, the cost for implementing each router in this network is higher than in 2D mesh networks. Also, note that some ports of the routers at the root of the tree are not used.

2.1.2 Routing

With the roadmap determined by the topology, routing algorithms define which path a packet takes to reach its destination. A routing algorithm can be categorized as oblivious or adaptive depending on how the routing decisions are determined.

[1] This thesis considers the common case that there is only one core in each node of a 2D mesh network. In general, a designer might choose to put c cores ($c \geq 2$) into each node; in this case, the router radix of the network is $4 + c$.

Oblivious routing algorithms do not use the network's state information in their routing decisions. For example, in 2D mesh networks, a widely used oblivious routing algorithm is the XY dimension-order routing (DOR) algorithm where a packet is routed first in the X dimension and then in the Y dimension to reach its destination. Although this algorithm may produce load imbalance for some traffic patterns, it has been widely used in many commercial and research systems [11, 43, 84, 40, 41] because it is very simple to implement, has short routing delay, requires low hardware overhead, and simplifies the deadlock avoidance problem.

Contrary to oblivious routing algorithms, adaptive routing algorithms consider the network's state information in their routing decisions. Thus, they can adapt to the condition of the network and are often better than oblivious routing algorithms in load balancing. However, adaptive routing algorithms are generally more complex, have longer routing delays, and incur more hardware overhead.

Apart from the requirements of routing delay and hardware overhead, deadlock freedom is an issue that must be taken into account when designing any routing algorithm. A deadlock occurs when multiple packets form a dependency cycle that lasts forever, that is, there is no way that this cycle can be broken. In general, deadlock freedom can be achieved by focusing on the routing algorithm or the flow control protocol.

Achieving deadlock freedom by focusing on the routing algorithm: Deadlocks are avoided by ensuring that the paths produced by the routing algorithm do not form any cycles. For example, Glass and Ni [85] observe that there are eight possible turns in a 2D mesh: from North to West, from West to South, from South to East, from East to North, from North to East, from East to South, from South to West, and from West to North; and by permitting only four of these turns, the XY DOR algorithm does not produce any cycles of paths and thus is deadlock-free. They also find that cycles of paths can be prevented by prohibiting only two of the eight turns. Based on this, they propose three deadlock-free routing algorithms: west-first (prohibiting two turns from North to West and from South to West), north-last (prohibiting two turns from North to West and from North to East), and negative-first (prohibiting two turns from North to West and from East to South). Chiu [72] points out that the routing flexibility provided by these algorithms is not even for all source-destination pairs. Specifically, when any of the algorithms is used, at least half of the source-destination pairs have only one minimal path while this is not the case for the others. Chiu proposes that, by using different sets of prohibited turns for nodes in odd and even columns, the routing flexibility can be improved while deadlock freedom is still guaranteed. For example, when a packet is at a node in an even column, it is prohibited from a set of two turns: from East to North and from North to West. When the packet is at a node in an odd column, it is prohibited from a different set of two turns: from East to South and from South to West.

Achieving deadlock freedom by focusing on the flow control protocol: Deadlocks are avoided by preventing router buffers from being allocated to packets in a way such that a dependency cycle of packets is formed. The deadlock avoidance method proposed in Chapter 5 is based on this approach.

2.1.3 Flow Control

A flow control protocol determines how shared resources in the network such as router buffers and channels are allocated when contention occurs. Most modern NoCs use wormhole and virtual channel (VC) flow control.

In wormhole flow control, router buffers and channels are allocated on a *flit-by-flit* basis. A *flit (flow control digit)* is the smallest unit of flow control. Each packet, the basic unit of transmission of a network, is composed of a head flit, some body flits, and a tail flit. By splitting each packet into multiple flits, large packet sizes can be supported while using small buffers because flits can be transferred to the next hop without waiting for the entire packet.

VC flow control [86] divides each router's input FIFO buffer into several smaller ones (VCs). By this way, each physical channel is associated with multiple small FIFO buffers instead of a deep one. Several VCs may share the bandwidth of a physical channel. Moreover, if a packet is blocked at a FIFO buffer, other packets can still use another FIFO buffer at the same port to pass the blocked packet. VC flow control thus makes efficient use of both physical channels' bandwidth and router buffers.

Because an upstream router can only send a flit to a downstream router if the corresponding buffer in the downstream router has a free space for the flit, a mechanism for managing the agreement across routers is required. One of the most efficient mechanisms is using credits. In credit-based flow control, each router maintains credit counters for tracking the state of the adjacent routers' buffers. Suppose that R_1 and R_3 are two adjacent routers of router R_2. At router R_2, when a flit, previously received from router R_1, leaves a buffer to go to router R_3, the credit counter for the downstream buffer at router R_3 is decremented while a credit is sent to router R_1 to increment the credit counter for the upstream buffer. Because of the delay in sending credits between routers, credit counters are always smaller than the actual numbers of free spaces of corresponding buffers. In routers that have a limited amount of buffers, the impact of this delay on overall performance may be large.

2.1.4 Router Architecture

Router architecture defines the internal organization and pipeline structure of routers in which the routing algorithm and flow control protocol are implemented. Most of the recently proposed NoCs are based on the input-queued VC router [4]. Figure 2.2 shows the architecture of this router. Its operation can be pipelined. The typical five-stage pipeline structure consists of Routing Computation (RC), VC Allocation (VA), Switch Allocation (SA), Switch Traversal (ST), and

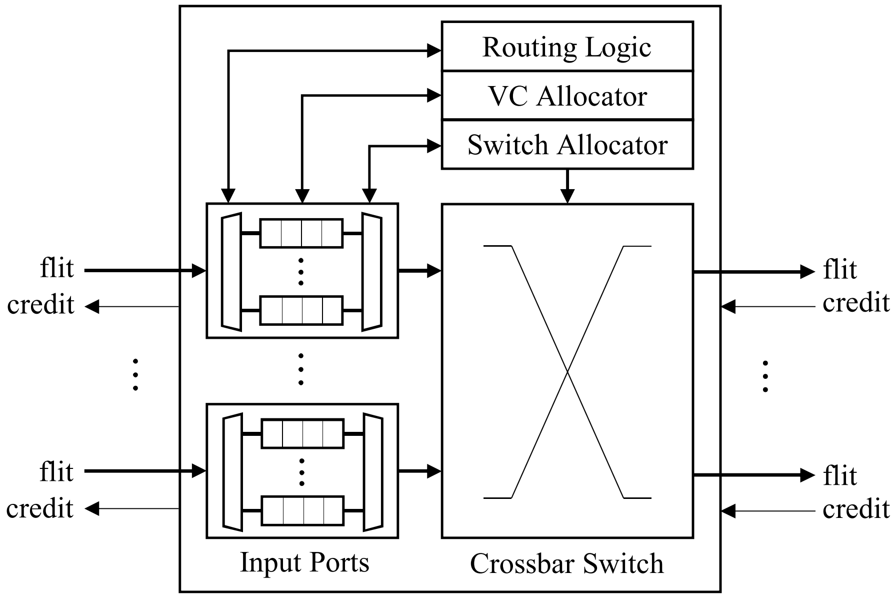


Figure 2.2: The conventional input-queued VC router architecture with credit-based flow control [4].

Link Traversal (LT). Only head flits proceed through the first two stages RC and VA. The remaining three stages are performed for every flit. When the head flit of a packet arrives at a router, stage RC is performed to determine the output port to which the packet is passed. After that, the packet is allocated an output VC at stage VA. Once stage VA is completed, each flit of the packet is allocated a time slot at the crossbar switch, traverses the crossbar switch, and traverses the output link towards the next router at stage SA, ST, and LT, respectively.

The router pipeline structure directly affects the overall latency of the network. The minimum number of cycles that it takes each head flit to traverse a router is equal to the number of pipeline stages. Additional delay may arise due to the contention at two stages VA and SA. The body flits and the tail flit of a packet inherit the output port and output VC from the head flit and thus can skip two stages RC and VA.

There have been numerous efforts to improve the network performance by reducing the number of stages in the router pipeline structure while maintaining a short critical path delay. Some typical approaches include look-ahead routing [4, 87], speculative architecture [88], bypassing [89, 90], and prediction [91]. Chapter 4 will show how the look-ahead routing technique can improve the NoC performance in a case study of the proposed FPGA-based NoC emulator.

2.2 FPGA Basics

FPGAs are programmable devices currently comprising of up to millions of interconnected logic blocks that can be used to implement any logic functions. In Xilinx FPGAs, each of these configurable logic blocks consists of one or several *slices*. For instance, each configurable logic block in Xilinx 7 series FPGAs is composed of two slices [92]. Each slice is formed from some *Look-Up Tables (LUTs)*, flip-flops, multiplexers, and arithmetic carry logic. The programmability of FPGAs comes from the fact that a LUT can be configured to become any logic gate.

In addition to the configurable logic blocks, for higher efficiency, FPGA manufacturers also integrate a large number of hard blocks of common functions fixed into the silicon such as embedded SRAMs (called *block RAMs*, or *BRAMs* for short, in Xilinx FPGAs) and floating-point digital signal processing (DSP) blocks. However, the higher efficiency comes at the expense of programmability. For instance, a BRAM in Xilinx FPGA has only two ports. Thus, a memory with three or more ports cannot be directly implemented using BRAMs. Besides the limitation of the number of ports, BRAMs can only be configured to some predetermined patterns (e.g., 1-bit×32,768; 2-bit×16,384; etc.). Because of this, implementing a memory with the bit-width not included in the predetermined patterns requires at least two BRAMs even the depth of this memory is small.

On FPGAs, a memory can be implemented by using either configurable logic blocks or embedded SRAMs. Configurable logic blocks are flexible but extremely inefficient for implementing memories larger than several kilobytes. Embedded SRAMs are more efficient but have the limitations described above. The on-chip memory capacity of an FPGA is calculated from the number of fixed-size embedded SRAMs and the amount of memory that can be implemented using the configurable logic blocks. Modern FPGAs have very limited on-chip memory capacity. Even large-scale FPGAs have around only several to ten megabytes of on-chip memory. Therefore, in FPGA-based systems, large data must be stored outside of the FPGA (usually off-chip DRAM).

2.3 NoC Modeling

Modeling is the foundation of NoC research and development. It is crucial for architects to test and evaluate their ideas. This section first briefly reviews three basic approaches in NoC modeling, namely, analytical modeling, hardware prototyping, and simulation, and then highlights the urgent need for simulation acceleration.

2.3.1 Analytical Modeling

Analytical modeling is an important evaluation method that is often used in the early stages of the design cycle. In this method, the characteristics of a design are described by using a collection of formulas. For example, assuming there is no contention, the average latency (in cycles) that a packet must experience in a 2D mesh NoC of size $k \times k$ under the uniform random traffic in which a node sends packets to the others with equal probability can be estimated by

$$L = H_{avg} \times D_{hop} + P_{len} - 1$$

where H_{avg}, D_{hop}, and P_{len} are the average minimum hop count of the network, the number of pipeline stages of the router architecture, and the packet length (in flits), respectively. H_{avg} can be calculated by the following formula.

$$H_{avg} = \begin{cases} \frac{2k}{3} & \text{if } k \text{ is an even number} \\ 2(\frac{k}{3} - \frac{1}{3k}) & \text{if } k \text{ is an odd number} \end{cases}$$

Studies such as [53] and [54] provide much more sophisticated analytical models for analyzing NoC performance across some parameters like topology and several microarchitecture decisions.

Analytical modeling has two major advantages. First, it is extremely fast. An estimate can be almost immediately obtained because we only have to compute a limited number of formulas. Second, the formulas often give us fundamental insights into the characteristics of the design. An analogous example is the implication of Amdahl's law [93] in parallel computing which states that the theoretical speedup S of the execution of a task obtained with parallelization can be computed by

$$S = \frac{1}{1 - p + \frac{p}{N}} \tag{2.1}$$

where p ($0 \leq p \leq 1$) is the proportion of execution time that the part benefiting from parallelization originally occupied and N is the speedup of the execution of this part. The insight that can be derived from formula (2.1) is that no matter how hard we try to improve N, we still cannot achieve a speedup larger than $\frac{1}{1-p}$.

However, analytical modeling cannot be used to make design decisions in many cases because either it introduces too much inaccuracy or it is too hard to find appropriate formulas for describing the design decisions. Therefore, NoC designers often use analytical models as the guidance for only high-level decisions.

2.3.2 Hardware Prototyping

A way to evaluate a NoC design is to build a hardware prototype of it [94, 95, 96, 97, 98]. Hardware prototypes can provide extremely accurate evaluation results. However, building them is extremely time-consuming and costly, especially if they are Application-Specific Integrated Circuits (ASICs). Therefore, hardware prototyping is often used in the final stages of the design cycle which are right before the production stage.

2.3.3 Simulation

Simulation is the de facto evaluation method not only in NoC design exploration but also in general computer architecture. Simulation can provide much more accurate evaluation results than analytical modeling while having a relatively cheap development cost compared to hardware prototyping.

2.3.3.1 Simulation Methodologies

Existing simulation methodologies can be classified into three categories [4]: execution-driven simulation, simulation with synthetic workloads, and simulation with trace-driven workloads. They have been used in a complementary manner.

The execution-driven simulation approach gives the highest degree of simulation accuracy but has three major drawbacks. First, developing and controlling full-system simulators that support execution-driven simulations is difficult because of the involvement of processing elements, memory modules, and I/O modules. Second, the simulation speed is generally very slow, especially for complicated and large designs. Third, it is hard to identify bottlenecks in the simulated NoC because any design change affects not only the NoC itself but also the workload.

Due to the above reasons, two approaches, simulation with synthetic workloads and trace-driven simulation, have been commonly used in the evaluation of NoCs. Synthetic workloads capture the salient features of the execution-driven workloads while remaining flexible. However, because of the high level of abstraction, in some cases, the simulation results may not reveal the characteristics of the NoC under the intended applications. This motivates the use of the trace-driven simulation approach in which a NoC simulator replays a sequence of messages captured from either a working system or an execution-driven simulation.

This thesis focuses on simulation with synthetic workloads and trace-driven simulation. Supporting execution-driven simulation is left as future work.

2.3.3.2 Cycle Accuracy

Cycle accuracy is a level of simulation accuracy in which the target design is simulated on a cycle-by-cycle basis. With the same input, all of the state elements (e.g., memories, registers) in the simulation contain the same values as those in a real hardware implementation at every clock cycle. Thus, the evaluation results obtained from cycle-accurate simulations are totally reliable.

Cycle-accurate simulations have been shown to be extremely important in NoC research specifically and in computer architecture research generally. For instance, in [70], Khan has provided empirical evidence that sacrificing the cycle accuracy may lead to conclusions that are wrong both quantitatively and qualitatively. Therefore, preserving the cycle accuracy is a crucial issue.

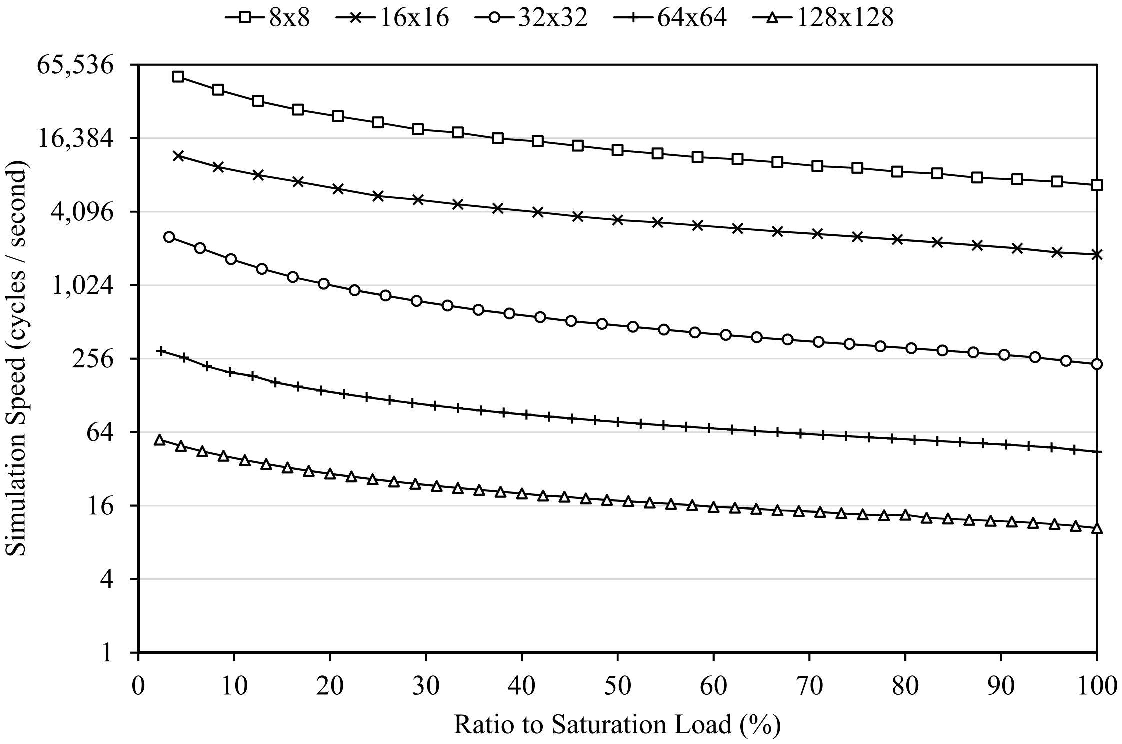


Figure 2.3: Simulation speed of BookSim [5], one of the most widely used cycle-accurate NoC simulators, for different network sizes.

2.3.4 The Need for Simulation Acceleration

Cycle-accurate simulators, while being reliable for making design decisions, are too slow, especially when the target design is large. Figure 2.3 shows the simulation speed of BookSim [5], one of the most widely used cycle-accurate NoC simulators, when simulating five NoCs of different sizes with a synthetic workload on a Core i7 4770 PC. The results in the case of simulation with trace-driven workloads are almost the same. We can see that the simulation speed of BookSim is around 6,700–52,000 simulation cycles per second when simulating the 8×8 NoC and is reduced to just 10–55 simulation cycles per second when simulating the 128×128 NoC. At this speed, running a long simulation would take an excessive amount of time. For example, in the case of trace-driven simulation, the length of a meaningful trace for 8×8 NoCs is typically several billions of cycles and becomes longer for larger network sizes; the simulation time is thus impractical.

The above analysis shows that there is a great need for accelerating NoC simulation. This thesis addresses this problem with the FPGA emulation approach.

2.3.5 FPGA Emulation: A Hardware-Accelerated Approach to Simulation

FPGA emulation is becoming a promising approach for accelerating NoC simulation due to three reasons. First, with the programmability of FPGAs, FPGA-based hardware can be developed in an incremental way which is analogous to software development. Second, the fine-grain parallelism of FPGAs can be leveraged to achieve orders-of-magnitude speedup relative to software simulators. This has been shown in the literature which will be discussed in Section 2.4.2. Third, FPGA development tools are becoming better and better, which helps to significantly reduce the effort that designers need to spend on writing, testing, and debugging code.

2.4 Related Work

2.4.1 Software Simulators

Full-system simulators such as gem5 [55] and MARSS [99] enable the study of full many-core systems, including NoCs and other components, with real applications. They can be designed to be cycle-accurate, but are so slow that most studies are restricted to designs with less than 64 cores. For example, MARSS achieves a speed of less than 200 Kilo Instructions Per Second (KIPS) when simulating an 8-core design. At this speed, simulating large designs would take an excessive amount of time.

Most parallel full-system simulators sacrifice accuracy for speed. For instance, ZSim [57] proposes a parallelization technique that divides the simulation into many small intervals of several thousand cycles. In each interval, processor cores are simulated in parallel while ignoring resource contentions and using zero-load latencies for all memory accesses. The loss of accuracy can be small if there are only a few interactions between instructions from different processor cores. However, maintaining a high degree of accuracy is challenging in many cases.

Stand-alone NoC simulators often provide more detailed network models than full-system simulators do. GARNET [100] is a NoC simulator that has been incorporated into gem5. BookSim [5] is a detailed and cycle-accurate NoC simulator that provides a wide variety of parameterized network components. BookSim is designed to avoid mechanisms that are impractical to implement in hardware. For example, the communication between two adjacent routers is established via a channel with a parameterized delay rather than a global variable. Therefore, compared to other NoC simulators like GARNET, BookSim is slightly slower. Noxim [101] is a NoC simulator written in SystemC. What makes Noxim unique is that it supports emerging wireless NoC architectures. GARNET, Noxim, and BookSim are open source and have been widely used in the NoC research community. Other open source and notable NoC simulators include NOCulator [102], VisualNoC [103], and Access Noxim [104]. Like GARNET, BookSim, and Noxim, NOCulator provides cycle-accurate performance models for a wide range of topologies and routers. VisualNoC can visualize the detailed operations of routers, packets, and processing elements and is thus very useful for designers to analyze their routing and task mapping algorithms. Access Noxim is a simulator supporting 3D NoC architectures. Besides performance models, Access Noxim incorporates both power and thermal models.

In general, stand-alone NoC simulators are much faster than full-system simulators. However, as described in Section 2.3.4, they are still slow for simulating designs with hundreds to thousands of nodes in a reasonable time.

Existing parallelization techniques scale poorly because of the high synchronization cost. For instance, HORNET [58], a parallel NoC simulator, divides the simulation into parallel threads and periodically synchronizes all threads on a barrier twice per simulated cycle to preserve 100% timing accuracy. In this way, HORNET's speed scales almost linearly up to five threads when simulating a 64-node NoC. By performing the synchronizations less frequently, the simulation speed can scale well up to around 15 threads, but the simulation accuracy is no longer guaranteed.

2.4.2 FPGA-Based Emulators

There have been some attempts of using FPGAs to emulate full multi/many-core systems. Studies in the RAMP project [105], such as RAMP Red [106, 107], RAMP White [108, 109], ProtoFlex [110, 111], RAMP Gold [112], and HAsim [113], have shown that an unprecedented emulation performance can be achieved by leveraging the fine-grain parallelism of FPGAs. Several other studies also reported the great potential of the FPGA-based approach. For example, Arete [114] achieves an emulation speed of 55 MIPS when emulating an 8-core design on four FPGAs which is multiple orders of magnitude faster than conventional software simulators. However, there are only a few studies supporting detailed NoC models. HAsim proposes a TDM scheme where a system is emulated based on A-Ports [115], a method for abstracting the physical core as well as the physical router into separate modules connected by FIFO queues, and a novel use of permutations that represent the communication pattern between routers in the system. This TDM scheme enables HAsim to emulate a 16-core design with detailed core pipelines, cache hierarchy, and NoC using a Virtex-5 FPGA. Another study that considers detailed NoC models is Heracles [116]. Besides the NoC, Heracles provides four different types of processing units including one simple injector core dedicated for NoC simulation and three MIPS cores together with a parameterized memory system. On an FPGA, Heracles can emulate a 25-core NoC-based design with a seven-stage MIPS core, one level of private instruction/data caches, and 32KB of local memory per core.

Currently, emulating full many-core systems with over hundreds of cores is still an open research problem. The focus of the rest of this subsection is on stand-alone NoC emulators.

Table 2.2 highlights the differences between the state-of-the-art emulators and the emulator

Table 2.2: Comparison of FPGA-based NoC emulators

NoC emulator	Largest NoC	Fully pipelined router	Adaptive routing	TDM for direct NoCs	TDM for indirect NoCs	Using soft processor	Using off-chip memory	Dedicated hardware	Traffic	Approximate emulation speed (cycles/s)
AdapNoC [67]	32×32	No	Yes	Yes	No	Yes	Yes	No	Syn+Trace	30K–200K(1)
DuCNoC [68]	128×64	No	Yes	Yes	No	No	Yes	Yes	Syn+Trace	200K–375K(2)
Drewes *et al.* [66]	8×8	No	-	No	No	No	Yes	Yes	Syn+Trace	16K(1)
DART [65]	9×9	No	No	Yes	No	No	No	No	Syn+Trace	5,500K–16,000K(1)
FIST [64]	24×24	No	No	No	No	No	No	No	-	-
Papamichael [63]	4×4×4×4	No	No	Yes	No	Yes	Yes	No	Trace	-
AcENoCs [62]	5×5	No	No	No	No	Yes	Yes	No	Syn+Trace	20K–55K(2)
DRNoC [61]	4×4	No	No	No	No	No	No	Yes	Syn	-
Wolkotte *et al.*[60]	16×16	-	No	Yes	No	No	Yes	Yes	Syn	12K–35K(1)
FNoC (Proposal)	128×128 4-ary 6-tree	Yes	Yes	Yes	Yes	No	Syn:No Trace:Yes	No	Syn+Trace	11,580K–15,000K(1) 80.9K–97.7K(3) 7.9K–8.1K(4)

(1): speeds of emulating 8×8 NoCs under synthetic workloads.

(2): speeds of emulating 5×5 NoCs under synthetic workloads.

(3): speed of emulating a 128×128 NoC under a synthetic workload.

(4): speed of emulating a 4-ary 6-tree NoC under a synthetic workload.

proposed in this thesis (FNoC).

AdapNoC [67] is a NoC emulator focusing on 2D meshes. AdapNoC uses two Microblaze soft processors to implement the traffic generators/receptors and monitor the emulated network. AdapNoC employs the TDM approach proposed in the original work of this thesis [75, 76, 77, 78] in which a network is emulated by using a cluster of a small number of nodes. In AdapNoC, the processing power of the soft processors and the transmission overhead between them and the emulated NoC are the performance bottlenecks. Because of this, the emulation speed of AdapNoC is heavily affected by the offered traffic load. Moreover, increasing the cluster size may not improve the emulation speed because a larger cluster means that more data need to be sent/received to/from the soft processors. AdapNoC achieves a speed of from around 30K to around 200K emulation cycles per second (depending on the offered traffic load) when emulating an 8×8 NoC under uniform random traffic.

DuCNoC [68] is an upgraded version of AdapNoC. Different from AdapNoC, DuCNoC is built on a SoC FPGA (Xilinx Zynq-7000). It thus uses two ARM processors on the SoC FPGA, which are more powerful than the Microblaze soft processors used by AdapNoC, for implementing the traffic generators/receptors and manipulating the emulation. As a result, DuCNoC is faster than AdapNoC. The speed of DuCNoC when emulating a 5×5 NoC is around 200K–375K emulation cycles per second while that of AdapNoC is around 90K–280K emulation cycles per second. However, the emulation speed still decreases dramatically with respect to the NoC size, even at a much faster pace than software simulators like BookSim. The authors of DuCNoC report a speedup of 66× over BookSim when emulating a 512-node NoC using a cluster of 16 nodes; however, the speedup is reduced to just 3× when the NoC size is 8,192-node.

Drewes *et al.* [66] propose a NoC emulator with the same approach as DuCNoC: using a Xilinx Zynq-7000 SoC FPGA and utilizing the ARM processors on the SoC FPGA for implementing the traffic generators/receptors and monitoring the emulated NoC. This emulator can emulate a largest NoC of 64 nodes at a speed of around 16K emulation cycles per second.

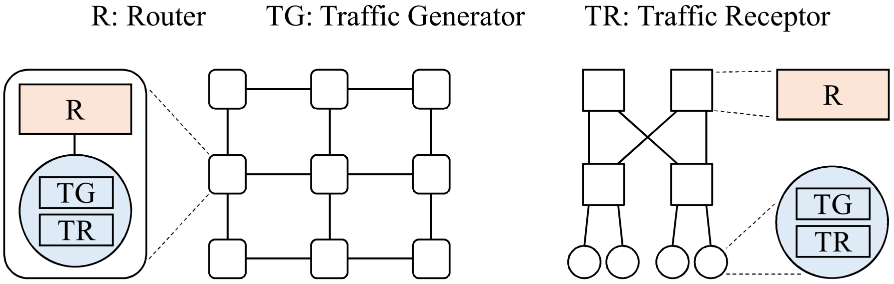
DART [65] is a NoC emulator which provides a global interconnect between all nodes together with a table-based routing scheme. With the global interconnect, by configuring the routing tables appropriately using a software tool on a host PC, DART can emulate any topology without re-synthesizing the design as long as the router radix in the topology is smaller than a predetermined value. However, this advantage comes at the expense of hardware overhead. The global interconnect leads to a large amount of FPGA resource usage. DART reduces the cost of global interconnect by grouping several nodes into a partition and using a crossbar for the partitions instead of using a full crossbar for all nodes. Despite this, it is difficult to scale DART to support large NoCs because the area cost of the crossbar increases quadratically with respect to the number of I/O ports. On a Virtex-6 FPGA, DART can emulate the largest network of 49 nodes at 50MHz. DART provides a TDM option in which a network is emulated using one DART node. As a result, the number of nodes that can be emulated is increased to 81. DART uses a simple NoC model in which the modeled router has only one output port and is a single-stage router.

FIST [64] is a lightweight packet latency estimator designed to be used within full-system simulation environments. FIST abstractly models each router in a network as a set of load-delay curves obtained by offline or online training. The latency of a packet is estimated by using the latencies obtained from the load-latency curves at the routers that the packet traversed. Because of the abstract modeling approach, FIST is not suitable for studying networks in detail.

Papamichael [63] proposes a NoC emulator using two approaches: direct-mapped implementation, and virtualized implementation. In the former approach, the emulated NoCs are directly implemented on an FPGA. The later approach adopts the TDM technique. In both approaches, a single-stage router architecture is implemented. A Microblaze soft processor is used to initialize the traffic tables as well as the emulation parameters and monitor the emulation result. Off-chip DRAM is required to store the traffic tables. Hence, the performance of the overall system may be restricted by the DRAM access latency and bandwidth.

AcENoCs [62] is another NoC emulator which uses a soft processor to control the traffic generation and injection process and monitor the emulation result. What separates AcENoCs from other emulators is that it supports Globally Asynchronous Locally Synchronous NoCs. However, like DART and [63], AcENoCs adopts a single-stage router model.

Like DuCNoC and [66], several other NoC emulators are also based on dedicated hardware. DRNoC [61] leverages the partial reconfiguration capability of several modern FPGAs to model different NoCs without re-synthesizing the design. Wolkotte *et al.* [60] propose a NoC emulator



(a) Direct network (b) Indirect network

Figure 2.4: Emulation models of direct and indirect networks.

on a hardware platform consisting of an FPGA board and a SoC board. The FPGA board contains a Virtex-II 8000 FPGA while the SoC board has two ARM9 processors. On the FPGA board, the TDM technique is employed to sequentially emulate all routers of the target NoC using a single router. Software on one or both ARM9 processors generates traffic, controls the NoC on the FPGA, and analyzes output packets. Thus, FPGA resources can be used solely for implementing the NoC. However, the off-chip communication between the FPGA board and the SoC board is the performance bottleneck.

Most of the NoC emulators mentioned above use a simple single-stage router model and thus are not able to cycle-accurately emulate state-of-the-art pipelined router architectures. They emulate functional behavior rather than cycle-accurate behavior. In contrast, FNoC, the NoC emulator proposed in this thesis, is cycle-accurate and can deal with pipelined routers.

As discussed in Chapter 1, emulating large-scale NoCs is challenging due to the FPGA logic and on-chip memory capacity constraints. The largest network sizes that can be supported by DRNoC, AcENoCs, and the emulator in [66] are 4×4, 5×5, and 8×8 respectively. Wolkotte *et al.* [60], Papamichael [63], Wang *et al.* [65], and Kamali *et al.* [67, 68] use the TDM technique to scale to larger networks. Papamichael [63] supports four topologies including a 2D mesh (11×23, 253 nodes), a 3D torus (4×7×9, 252 nodes), a hypercube (4×4×4×4, 256 nodes), and a fully connected network (16 nodes), while the others support only 2D mesh topology. None of these studies considers indirect network topologies. On the contrary, this thesis discusses methods for effectively applying the TDM technique for both direct and indirect network topologies, focusing on 2D meshes and fat-trees.

2.5 NoC Emulation Model

2.5.1 Basic Components

Our emulation model consists of three basic components: *router*, *traffic generator*, and *traffic receptor*. As shown in Figure 2.4(a), in direct networks such as meshes and tori, each node is modeled by a router, a traffic generator, and a traffic receptor. In indirect networks such as fat-trees and butterflies, each switch node is a router while each terminal node is modeled by a traffic generator and a traffic receptor as shown in Figure 2.4(b). A pair of traffic generator and traffic receptor simulates the behavior of a core.

2.5.1.1 Router

The primary router model in this study is the input-queued pipelined VC router [4] described in Section 2.1.4 because it is the baseline of most of the recently proposed ASIC-style NoC architectures. However, note that the proposed methods are independent of the emulation target NoC router; they can be integrated with FPGA-friendly routers like Hoplite [117, 118] as well.

A detailed model of all router components including input buffers, routing logic, VC allocator, switch allocator, and crossbar switch is provided. These components are designed to be easily modified for realizing novel architecture optimizations. For instance, the modeled VC allocator can handle the most general case in which a packet at an input VC can send requests to all available output VCs at all output channels. This enables a wide range of routing algorithms and VC allocation policies, which cannot be achieved if a simple VC allocator model is used. For example, if we adopt a model in which there is only one candidate output VC for each packet, then the VC allocator will become much simpler, and hence a significant amount of FPGA resources will be saved. However, in this case, it is impossible to support routing algorithms that return multiple candidate output VCs for each packet.

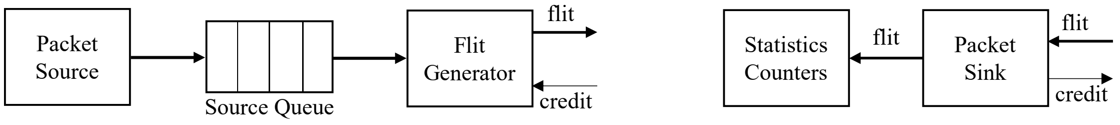
The thesis does not address a specific target design even this may help to reduce a lot of FPGA resources required. For example, when the 2D mesh topology and the XY DOR algorithm are used, the implementation of the switch allocator and crossbar switch can be significantly simplified based on the observation that a packet at the north or the south input port will not go to the east and the west output ports. However, this makes it difficult to support other topologies and routing algorithms. Therefore, the thesis does not optimize the router model for a specific topology and routing algorithm.

2.5.1.2 Traffic Generator

As mentioned before, the focus of this thesis is on synthetic and trace-driven workloads. This subsection describes the high-level traffic generator architecture for modeling these workloads. The detailed modeling methods will be described in Chapter 4 and Chapter 6.

As shown in Figure 2.5(a), each traffic generator consists of a *packet source*, a *source queue*, and a *flit generator*.

In the case of synthetic workloads, the packet sources generate packet descriptors according to the specified injection processes. Periodic processes are the simplest injection processes. In



(a) Traffic Generator (b) Traffic Receptor

Figure 2.5: Architecture of each traffic generator and traffic receptor.

a periodic process, the period T between injections is a constant. Therefore, a traffic generator does not need a large source queue to make sure that no injection of the packet source is dropped. A one-entry source queue is enough because the next injection time can be easily calculated by just adding T to the current injection time. However, periodic processes are too simple. They do not incorporate randomness which might be expected from a real injection process. More complex injection processes such as Bernoulli and Markov modulated processes [4] that incorporate randomness are commonly used in NoC simulation/emulation. Because of the randomness of injections, we typically need a large source queue after every packet source to model the injection processes accurately. If the source queues are not large enough, then some injections of the packet sources may have to be dropped, and therefore, the produced workload may not be the one originally specified. Chapter 4 will propose a novel method for accurately emulating NoCs under synthetic workloads with injection processes that incorporate randomness while using small source queues.

In the case of trace-driven workloads, the packet sources get packet descriptors from the modules called *trace loaders* which are outside of the traffic generators. The interaction between these modules and the off-chip memory will be described in detail in Chapter 6.

When the network is ready, the flit generator reads a packet descriptor from the source queue and generates flits based on the information encoded inside this packet descriptor. The generated flits are injected one by one into the network. The flit generator does not read another packet descriptor until all flits generated from the previous packet descriptor have been injected into the network. The status of the network is tracked by using the incoming flow control credits.

2.5.1.3 Traffic Receptor

Traffic receptors (Figure 2.5(b)) are responsible for ejecting packets from their destinations and collecting performance characteristics of the emulated NoC using some statistics counters. When a packet arrives at its destination, it is forwarded to the traffic receptor. Here, the desired performance characteristics such as packet latency are recorded. The latency of a packet is calculated according to the generation timestamp, which indicates the time at which the packet is generated and injected into the source queue, and thus includes the waiting time in the source queue. The traffic receptor is also responsible for sending back flow control credits to the network.

Table 2.3: Flit model

Valid	1 bit
Look-ahead routing information	w bits
Flit type	2 bits
VC	x bits
Other control information	y bits
Data payload	z bits

2.5.2 Flit Model

Table 2.3 shows the flit model used in this study. Each flit is composed of six fields: valid (1 bit), look-ahead routing information (w bits), type (2 bits), VC (x bits), other control information (y bits), and data (z bits). The valid bit determines whether the flit exists. If the emulated router architecture employs look-ahead routing, then we need w bits (e.g., w is equal to 3 if the network topology is 2D mesh) for storing routing information passed between routers. Otherwise, w is set to zero. The 2-bit type defines the type of flit (head, body, or tail). The x-bit VC determines which VC the flit is stored into. The value of x depends on the number of VCs per port. We reserve y bits for other control information. For example, 1 bit of control information is used to implement an adaptive routing algorithm described in the evaluation section of Chapter 4. In the case no other control information is required, y is set to zero. Finally, the z-bit data are the flit payload. The destination address of a packet is carried in the payload of the head flit. Thus, the flit payload size z determines the number of nodes that can be addressed.

If the minimum packet length is one flit, the payload of a head flit must also carry the information necessary for recording the desired performance characteristics of the emulated NoC (e.g., the packet's generation timestamp) besides the destination address of the packet. Otherwise, the payload of a head flit carries only the packet's destination address. Other information is carried by the body and tail flits. By this way, we can reduce the memory usage since the flit size directly affects the amount of memory required for the routers' buffers.

In the case studies described in the evaluation section of Chapter 4, where the packet length is eight flits, the flit payload size (z) is set to 14 bits to be able to address all nodes of a 128×128 NoC ($2^{14} = 16,384$), the largest NoC can be emulated by the current version of the proposed emulator. Also in these case studies, two flits are used to carry each packet's generation timestamp to be able to increase the maximum number of emulation cycles to 2^{28} without increasing the flit payload size. The flit payload size should be as small as possible because the amount of memory needed to implement the router buffers and the buffers responsible for storing the data transferred in the emulated network (*in/out buffer* in Figures 3.1 and 3.2 in Chapter 3) is almost proportional to the flit size. To understand the impact of the flit size on the total amount of memory required for implementation, let us consider a specific instance: emulating the 128×128-2vc-5stage-xy NoC (detailed parameters are shown in Tables 4.1 and 4.2 in Chapter 4) using a cluster of four nodes. In the current implementation, the flit size is 18 bits (14 bits for the payload field). With this flit size, the number of 36Kb BRAMs required for implementing the router buffers and the in/out buffers is 363 and that for implementing the other memories is 574 (the total number of 36Kb BRAMs on the currently used FPGA is 1,030). If we, for example, want to emulate 1,000,000 cycles but use only one flit to carry each packet's generation timestamp, the flit size will have to be increased to 24 bits (20 bits for the payload field). By estimation, the total number of 36Kb BRAMs required for implementation will be $\frac{363 \times 24}{18} + 574 = 1,058$, and thus we will not have enough BRAMs on the currently used FPGA to implement the emulator. In this case, we need an FPGA with more BRAMs. On the other hand, by using two flits to carry each packet's generation timestamp, we do not have to increase the payload size z as long as the number of emulation cycles is smaller than 2^{2z}.

In the case studies in the evaluation section in Chapter 6, because the packet length of the traces varies between 2-flit and 18-flit, the approach of using two flits to carry each packet's generation timestamp cannot be used as in the case studies in Chapter 4. Instead, each packet's generation timestamp is carried by only one flit. The flit payload size is set to 26 bits to be able to increase the maximum number of emulation cycles to 2^{26}.

DART [65] proposes a measurement method which makes the maximum number of emulation cycles independent of the packet's generation timestamp width. Specifically, besides the packet's generation timestamp, each flit contains a second timestamp which is updated at each router on the route from source to destination to reflect the router pipeline latency and delay caused by resource contention. When a flit arrives at the destination traffic receptor, the second timestamp indicates its arrival time. The latency of a packet is calculated by subtracting the packet's generation timestamp from the arrival time of the tail flit. In this way, if the bit width of each timestamp is t bits, DART can provide correct latency measurement results as long as the latency of every packet does not exceed 2^t cycles (t is set to 10 in DART). However, this is not guaranteed in all cases. Thus, this thesis does not adopt this measurement method. Instead, the method described above is used to be able to measure latencies of all packets.

Chapter 3

Novel Time-Division Multiplexing Methods

3.1 Introduction

The straightforward way to emulate a NoC on an FPGA is to fully replicate the nodes and connect them to the network. However, such a direct approach requires vast FPGA logic blocks and thus does not scale to networks with hundreds to thousands of cores. This chapter proposes a novel use of time-division multiplexing (TDM) where the emulation cycle is decoupled from the FPGA cycle and a network is emulated by time-multiplexing a small number of nodes.

This chapter describes methods for efficiently applying the TDM technique for both direct and indirect networks. This is different from previous studies which consider only direct networks. In the case of direct networks, the focus is on 2D meshes. The described methods, however, can be applied to other k-ary n-cubes. In the case of indirect networks, the focus is on fat-trees (in particular k-ary n-trees). The connectivity pattern of switching nodes (routers) in a k-ary n-tree is similar to that in a k-ary n-fly, except that all channels in the k-ary n-tree are bidirectional while all channels in the k-ary n-fly are unidirectional. Thus, the described methods can be easily modified to apply to k-ary n-flies. Since almost all actually constructed networks are derived from k-ary n-cubes and k-ary n-flies [4], it can be expected that the proposed methods can be extended for a wide range of networks.

In general, applying the TDM technique in the case of direct networks is different from that in the case of indirect networks. We can emulate a direct network of N_{node} nodes by time-multiplexing N_{node} logical nodes on a physical node. Some direct networks such as meshes and tori can be emulated by time-multiplexing on a group of several physical nodes which is called a *physical cluster*. On the other hand, since a node in an indirect network can be either a switching node (router) or a terminal node (core), we cannot use a single physical node to emulate the

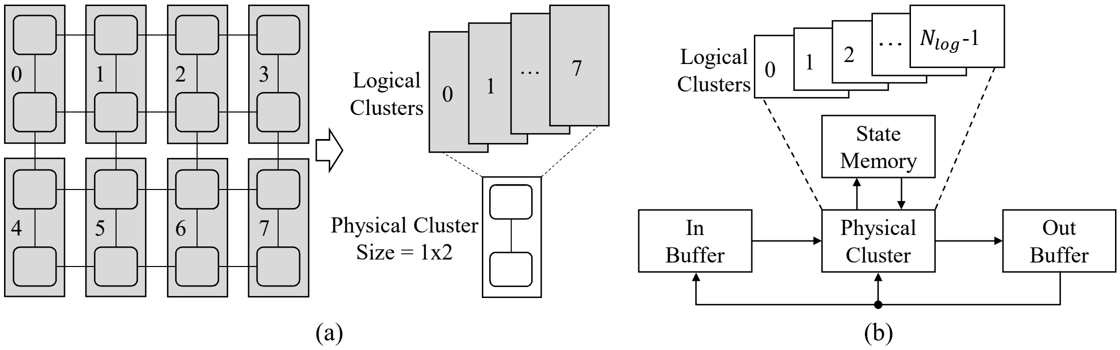


Figure 3.1: Emulating 2D mesh networks: (a) a 4×4 mesh NoC is emulated using two physical nodes and (b) high-level datapath.

behavior of the entire network. For indirect networks, we have to separate switch nodes from terminal nodes.

The major constraint of topology on the TDM emulation is that the inter-cluster/router emulation buffers (*out buffer* and *in buffer* in Figures 3.1(b) and 3.2(b)) can be efficiently implemented on FPGAs. For instance, Section 3.4 will show that, these buffers can be trivially mapped to BRAMs when emulating meshes, but this is not the case when emulating k-ary n-trees. We introduce the concepts of *physical port ID* and *logical port ID* in Section 3.4.3 to solve the problem.

3.2 High-Level Datapath for Emulating 2D Meshes

Figure 3.1(a) shows an example where a 4×4 mesh is emulated by using two physical nodes. The group of physical nodes is called the *physical cluster*. To complete one emulation cycle, the physical cluster sequentially emulates eight *logical clusters*. In general, larger physical clusters will reduce the number of logical clusters, and hence improve the emulation speed.

Figure 3.1(b) shows the high-level datapath when the TDM technique is employed. In addition to the physical cluster, there are three other components: *state memory*, *out buffer*, and *in buffer*.

The *state memory* is used to store the states of all logical clusters. The physical cluster emulates different logical clusters by using different states loaded from the state memory. When the emulation of a logical cluster is finished, its state in the state memory is overwritten by the new state which will be used in the next emulation cycle.

To emulate a logical cluster, in addition to the state data read from the state memory, the physical cluster needs appropriate data from other logical clusters. For example, in Figure 3.1(a), emulating logical cluster 5 requires data from logical clusters 1, 4, and 6. Thus, the outgoing data of all logical clusters are stored in a buffer which we call the *out buffer*.

However, using only the out buffer is not sufficient. To explain why, let us suppose that logical cluster 1 is emulated after logical cluster 0 and logical cluster 1 depends on the outgoing data of logical cluster 0. Let d_0^c be the outgoing data of logical cluster 0 after emulation cycle $c - 1$. In emulation cycle c, logical cluster 1 uses d_0^c. If only the out buffer is used, d_0^c will be overwritten by d_0^{c+1} before being used by logical cluster 1 since logical cluster 0 is emulated before logical cluster 1. To prevent such conflicts, as shown in Figure 3.1(b), the *in buffer* is used. In the above example, we copy d_0^c to the in buffer before overwriting it with d_0^{c+1} in the out buffer.

In some cases, we do not have to copy all data from the out buffer to the in buffer. The insight here is that, before writing the new data of a logical cluster into the out buffer, we only copy the old data that are necessary for emulating the subsequent logical clusters. In the case of 2D meshes, we can optimize the amount of data needed to be copied by choosing an appropriate emulation order. For instance, the emulation order in the example in Figure 3.1(a) requires us to copy data of only the east direction and the south direction instead of all four directions. Thus, the size of the in buffer can be reduced by half in this case. This emulation order is an optimal order to minimize the size of the in buffer because the outgoing data of any logical cluster affect at least two other logical clusters.

3.3 High-Level Datapath for Emulating k-Ary n-Trees

As mentioned before, a k-ary n-tree consists of k^n cores (terminal nodes) and n levels of k^{n-1} radix-$2k$ routers (switch nodes). The n levels are consecutively numbered starting from 0 at the root up to the leaves. The routers in each level are numbered from 0 at the leftmost position to $k^{n-1} - 1$ at the rightmost position.

A k-ary n-tree is emulated by applying the TDM technique with a physical router and a physical core and buffering data transferred between routers as well as between routers and cores. Figure 3.2(a) shows an example where a 2-ary 3-tree is emulated by time-multiplexing 12 *logical routers* and 8 *logical cores* on a *physical router* and a *physical core*, respectively. In general, let N_R and N_C be the numbers of routers and cores, respectively, in a k-ary n-tree; then each cycle of the network is completed after processing all N_R logical routers and N_C logical cores.

The datapath for emulating k-ary n-trees is given in Figure 3.2(b). Like in the datapath for 2D meshes, a *state memory* is used for storing the states of all logical routers and cores. The communication between routers is also performed through two buffers *out buffer* and *in buffer*, and the size of the in buffer can be optimized by choosing an appropriate emulation order. For example, the emulation order in the example in Figure 3.2(a) ($R_0 \rightarrow R_1 \rightarrow \cdots \rightarrow R_{11}$) does not require us to copy all data from the out buffer to the in buffer. We only have to copy the output data of the down ports. This emulation order is also an optimal order to minimize the size of the in buffer because the outgoing data of any logical router affect at least k other logical routers.

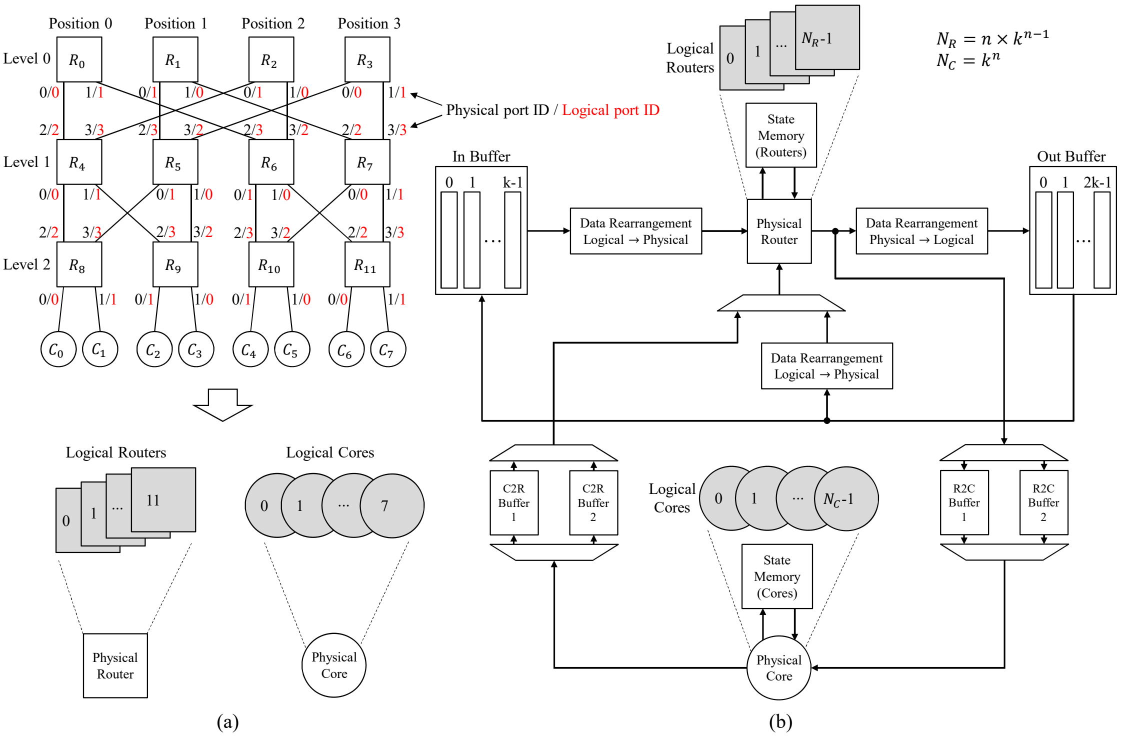


Figure 3.2: Emulating k-ary n-trees: (a) a 2-ary 3-tree is emulated by time-multiplexing 12 logical routers and 8 logical cores; (b) high-level datapath.

In a k-ary n-tree, the cores are connected to the level $n-1$ routers. Thus, when emulating a level $n-1$ router, we need data not only from some other routers but also from some cores. For example, in Figure 3.2(a), when emulating router R_8, we need data from routers R_4, R_5 and cores C_0, C_1. As shown in Figure 3.2(b), data transferred from the cores to the level $n-1$ routers are stored in two buffers *C2R Buffer 1* and *C2R Buffer 2*. In an emulation cycle, the cores write data to a buffer while the routers read data from the other. The two buffers exchange their roles after each emulation cycle. In particular, if a buffer is a write (read) buffer in emulation cycle c, it will become a read (write) buffer in emulation cycle $c+1$. By this way, we can ensure that, in any emulation cycle, data are not overwritten before being used. This has the same meaning as the use of the out buffer and the in buffer. Similarly, the data transferred from the level $n-1$ routers to the cores are stored in two buffers *R2C Buffer 1* and *R2C Buffer 2*.

As explained above, when processing a logical router, the input data of the up ports are loaded from the in buffer because they have been already overwritten in the out buffer. On the other hand, the input data of the down ports may be loaded from the out buffer or one of the C2R Buffers. Thus, as shown in Figure 3.2(b), a multiplexer is used to select data from the out buffer or the C2R Buffers.

In addition to the modules explained above, the datapath in Figure 3.2(b) contains three *Data Rearrangement* modules. These modules are responsible for rearranging the input and output data of the out buffer and the output data of the in buffer. This will be explained in detail in Section 3.4.3.

3.4 Inter-Cluster/Router Emulation Buffers

3.4.1 Essential Characteristic for Efficient Mapping to BRAMs

In Figures 3.1(b) and 3.2(b), the inter-cluster/router emulation buffers (out buffer and in buffer) are responsible for storing communication data between logical clusters/routers. Before going into details of the essential characteristic for efficient mapping these buffers to BRAMs, this section first presents formulas for calculating their sizes. Let s_{flit} be the flit size (bits) and n_{vcs} be the number of VCs per port.

In the case of 2D meshes, suppose that the physical cluster size is $x \times y$ and that the number of nodes is N_{node}. In each emulation cycle, a router may send a flit and v bits of flow control information to an adjacent router. Thus, the total bit-widths of the outgoing data at the north/south and east/west directions of a logical cluster are $(s_{flit} + n_{vcs}) \times x$ and $(s_{flit} + n_{vcs}) \times y$, respectively. Since the number of logical clusters is $N_{node}/(x \times y)$, the sizes of the out buffer and the in buffer are

$$M_{obuf}^{mesh} = 2 \times (s_{flit} + n_{vcs}) \times x \times \frac{N_{node}}{x \times y} + 2 \times (s_{flit} + n_{vcs}) \times y \times \frac{N_{node}}{x \times y}, \tag{3.1}$$

$$M_{ibuf}^{mesh} = \frac{M_{obuf}^{mesh}}{2}. \tag{3.2}$$

The formulas for k-ary n-trees are simpler than formulas (3.1) and (3.2) since only one physical router is used in the time-multiplexed emulation. Let $N_R = n \times k^{n-1}$ be the number of routers. We have

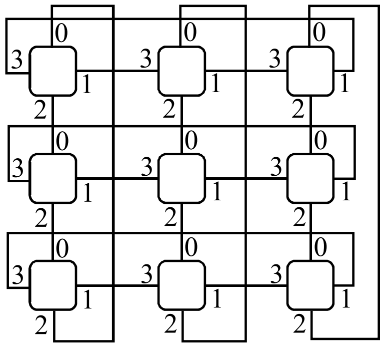
$$M_{obuf}^{tree} = (s_{flit} + n_{vcs}) \times 2k \times N_R, \tag{3.3}$$

$$M_{ibuf}^{tree} = \frac{M_{obuf}^{tree}}{2}. \tag{3.4}$$

From formulas (3.1), (3.2), (3.3), and (3.4), it is clear that emulating larger networks requires larger inter-cluster/router emulation buffers. For networks with over hundreds of nodes, it is infeasible to implement these buffers using other resources rather than BRAMs.

Now, let us look at the essential characteristic that allows an efficient buffer mapping to BRAMs. This is based on the fact that a BRAM has only two ports. On FPGAs, any memory with more than two ports cannot be directly implemented using BRAMs.

Let us consider a network of N_R routers $R_0, R_1, \cdots R_{N_R-1}$. Suppose that each router has P ports numbered from 0 to $P-1$ (in the case of direct networks, the port connected to the



Every router has the same port mapping function g:
$g(0) = 2$
$g(1) = 3$
$g(2) = 0$
$g(3) = 1$

Figure 3.3: Every router in a torus has the same port mapping function and this function is bijective.

core is excluded), and that g_i $(0 \leq i \leq N_R - 1) : [0, P-1] \rightarrow [0, P-1]$ is the port mapping function determining how P ports of router R_i are connected to its adjacent routers' ports (port j of router R_i is connected to port $g_i(j)$ of an adjacent router of R_i). **The essential characteristic that allows an efficient buffer mapping to BRAMs is that every port mapping function g in $\{g_0, g_1, \cdots g_{N_R-1}\}$ is bijective, that is, $\forall j_1, j_2 \in [0, P-1]$, if $j_1 \neq j_2$, then $g(j_1) \neq g(j_2)$ and vice versa (*)**.

If a topology follows rule (*) and we divide the out buffer as well as the in buffer into multiple smaller buffers according to the port ID (buffer j stores outgoing data at port j), we can ensure that each port of the currently emulated router reads data from a different buffer inside the out buffer and the in buffer. Thus, every buffer inside the out buffer and the in buffer has only one read port and one write port and can be efficiently mapped to BRAMs.

3.4.2 2D Mesh

Obviously, rule (*) works perfectly for tori. As shown in Figure 3.3, every router in a torus has the same port mapping function and this function is bijective. Rule (*) can be also applied to meshes because we can ignore the ports at the borders that are not connected to any routers.

The discussion until now is in the case of time-multiplexed emulation with the physical cluster of only one physical node. However, we can easily see that it can also be applied to the case that the physical cluster is composed of multiple interconnected physical nodes.

3.4.3 k-Ary n-Tree

3.4.3.1 Using Logical Port IDs instead of Physical Port IDs

In general, rule (*) does not work for k-ary n-trees, k-ary n-flies, and clos networks. To explain why rule (*) does not work for k-ary n-trees, let us revisit Figure 3.2(a). Here, two new concepts of *physical port ID* and *logical port ID* are introduced. Physical port IDs are the port IDs that we have considered so far. In Figure 3.2(a), in each router, the physical IDs of the down ports are from 0 at the leftmost position to $k-1$ at the rightmost position while the physical IDs of the up ports are from k at the leftmost position to $2k-1$ at the rightmost position. One example which shows that rule (*) does not work is the port mapping function of router R_4 (g_4): we have $0 \neq 1$ but $g_4(0) = g_4(1) = 2$. Because of this, even we divide the out buffer and the in buffer into multiple smaller buffers according to the physical port ID, we still need memories with three ports (one write port and two read ports). As k increases, the required memories will have more ports. Implementing such memories directly on FPGAs not only requires vast registers, LUTs, and multiplexers but also makes the overall operating frequency decrease substantially. Although the replication approach can be used to reduce to the numbers of read ports of those memories to one, it is not scalable because the number of required BRAMs is increased multiple times. By introducing the concept of logical port IDs, the thesis makes it possible to map the out buffer and the in buffer into BRAMs with small overhead.

The logical port IDs of the routers in a k-ary n-tree are defined as follows:

1. In a router, each down port has a unique logical port ID in $[0, k-1]$ while each up port has a unique logical port ID in $[k, 2k-1]$.

2. $\forall l \in [0, n-2], \forall j \in [0, k-1]$, output port j (logical port ID = j) of a router in level l is connected to input port $j+k$ (logical port ID = $j+k$) of a router in level $l+1$.

3. $\forall l \in [1, n-1], \forall j \in [k, 2k-1]$, output port j (logical port ID = j) of a router in level l is connected to input port $j-k$ (logical port ID = $j-k$) of a router in level $l-1$.

Figure 3.2(a) shows an example. We can see that the physical port ID assignment is the same for every router while any two routers might have different logical port ID assignments. For instance, the logical port ID assignment of router R_4 is different from that of router R_5.

By the above definition, it is obvious that rule (*) works for k-ary n-trees if we consider logical port IDs instead of physical port IDs. Thus, by using logical port IDs, the out buffer and the in buffer can be efficiently mapped to BRAMs.

3.4.3.2 Procedure for Calculating Logical Port IDs

This section describes a procedure to calculate the logical port IDs of every router in a k-ary n-tree. The proof of correctness of this procedure is provided in Section 3.4.4.

In the proposed procedure, the logical ID of a port of a router is calculated recursively from the base of a function which maps the physical IDs of the down ports of the level $n-1$ routers to logical IDs. Let R be the set of routers in the k-ary n-tree network, and $f : R \times [0, 2k-1] \rightarrow [0, 2k-1]$ be a function which maps each physical port ID of each router to a logical port ID. The proposed procedure is as follows.

- **Step 1**: calculating the logical port IDs of the down ports of the routers in level $n-1$. $\forall$ router r in level $n-1$, let p be the position of r in the level ($p \in [0, k^{n-1}-1]$). $\forall$ physical port ID $\texttt{phyid} \in [0, k-1]$, the logical port ID which corresponds to $\texttt{phyid}$ is given by

$$f(r, \texttt{phyid}) = \left(\texttt{phyid} + \left\lfloor \frac{p}{k^{n-2}} \right\rfloor + \left\lfloor \frac{p\%k^{n-2}}{k^{n-3}} \right\rfloor + \left\lfloor \frac{\left(p\%k^{n-2}\right)\%k^{n-3}}{k^{n-4}} \right\rfloor + \cdots + \left\lfloor \frac{\left(\left(\left(p\%k^{n-2}\right)\%k^{n-3}\right)\%\cdots\right)\%k^{1}}{k^{0}} \right\rfloor\right)\%k. \tag{3.5}$$

The idea of this formula is to avoid duplication of logical port IDs in every router when applying the rules described in Steps 2 and 3. For example, the logical port ID corresponding to the physical port ID 0 of router R_9 in the 2-ary 3-tree ($p = 1, k = 2, n = 3$) in Figure 3.2(a) is

$$f(R_9, 0) = \left(0 + \left\lfloor \frac{1}{2^1} \right\rfloor + \left\lfloor \frac{1\%2^1}{2^0} \right\rfloor\right)\%2 = 1\%2 = 1.$$

- **Step 2**: calculating the logical port IDs of the up ports of the routers in level $n-1$. $\forall$ router r in level $n-1$, $\forall$ physical port ID $\texttt{phyid} \in [k, 2k-1]$, the logical port ID which corresponds to $\texttt{phyid}$ is given by

$$f(r, \texttt{phyid}) = 2k - 1 - f(r, 2k - 1 - \texttt{phyid}). \tag{3.6}$$

Here, $f(r, 2k-1-\texttt{phyid})$ has been determined in Step 1. For example, the logical port ID corresponding to the physical port ID 3 of router R9 in the 2-ary 3-tree ($k = 2, n = 3$) in Figure 3.2(a) is given by

$$\begin{aligned} f(R_9, 3) &= 2 \times 2 - 1 - f(R_9, 2 \times 2 - 1 - 3) \\ &= 3 - f(R_9, 0) \\ &= 3 - 1 = 2. \end{aligned}$$

- **Step 3**: calculating the logical port IDs of the down ports of the routers in level $n-2$. $\forall$ router r in level $n-2$, $\forall$ physical port ID $\texttt{phyid} \in [0, k-1]$, let r' be the router at level $n-1$, which is connected to physical port $\texttt{phyid}$ of router r at physical port $\texttt{phyid}'$. Then, the logical port ID corresponding to $\texttt{phyid}$ is given by

$$f(r, \texttt{phyid}) = f(r', \texttt{phyid}') - k. \tag{3.7}$$

Here, $f(r', \texttt{phyid}')$ has been determined in Step 2. For example, in Figure 3.2(a), physical port 1 of router R_5 is connected to physical port 3 of router R_9. Thus, the logical port ID corresponding to the physical port ID 1 of router R_5 is given by

$$\begin{aligned} f(R_5, 1) &= f(R_9, 3) - 2 \\ &= 2 - 2 = 0 \end{aligned}$$

- **Step 4**: calculating the logical port IDs of the routers in the remaining levels like Step 2 and 3.

3.4.3.3 Conversion between Logical Port IDs and Physical Port IDs

When emulating k-ary n-trees, since both the physical and logical port IDs are used, the arrangement of input/output data of the out buffer as well as the in buffer is different from that of the physical router. Specifically, the input/output data of the out buffer as well as the in buffer are arranged according to the logical port IDs. In contrast, the input/output data of the physical router are arranged according to the physical port IDs. Therefore, as shown in Figure 3.2(b), three *Data Rearrangement* modules for rearranging the input/output data of the out buffer and the output data of the in buffer are used. These modules rearrange data based on the conversion between logical port IDs and physical port IDs.

The conversion between logical port IDs and physical port IDs is performed by using two conversion tables: one for converting from physical port IDs to logical port IDs and the other for converting from logical port IDs to physical port IDs. These tables are respectively called *Phy2Log* and *Log2Phy*. The approach of using the conversion tables is adopted because, as described in Section 3.4.3.2, calculating logical port IDs from physical port IDs and vice versa involves complex modulo, multiplication, division, and exponent operations that are hard to implement on FPGAs. The details of the proposed approach are as follows. A software tool is developed to calculate all entries of the port ID conversion tables. The software tool writes each conversion table to a file. The contents of the files are loaded into ROM memories, which are actually implemented using BRAMs, before the emulation starts (using *$readmemb* or similar functions in Verilog). In this way, the emulation can proceed by using the offline-calculated data stored in the ROM memories.

Finally, let us look at the memory overhead of the port ID conversion tables. The number of entries of each conversion table is equal to the number of routers of the emulated k-ary n-tree which is $n \times k^{n-1}$. Each entry of *Phy2Log* is composed of $2k$ logical port IDs of a router. Similarly, each entry of *Log2Phy* is composed of $2k$ physical port IDs of a router. In the implementation in this thesis, the bit-width of each port ID is $\lceil \log_2(2k+1) \rceil$. Thus, the size of each

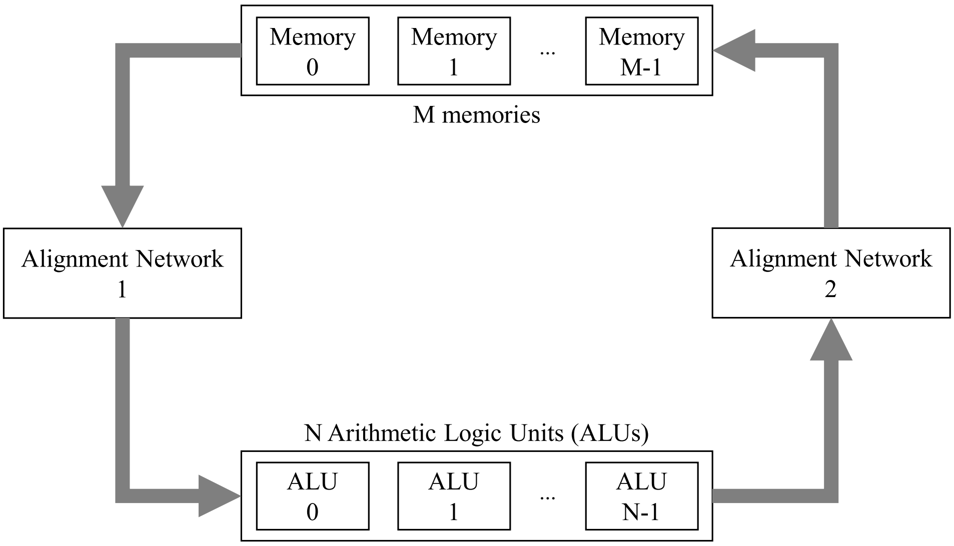


Figure 3.4: An array processor [6].

d_0^0	d_0^1	d_0^2	d_0^3
d_1^0	d_1^1	d_1^2	d_1^3
d_2^0	d_2^1	d_2^2	d_2^3
d_3^0	d_3^1	d_3^2	d_3^3

(a)

Memory

0	1	2	3	4	5	6	7
d_0^0		d_0^1		d_0^2		d_0^3	
	d_1^3		d_1^0		d_1^1		d_1^2
d_2^1		d_2^2		d_2^3		d_2^0	
	d_3^0		d_3^1		d_3^2		d_3^3

(b)

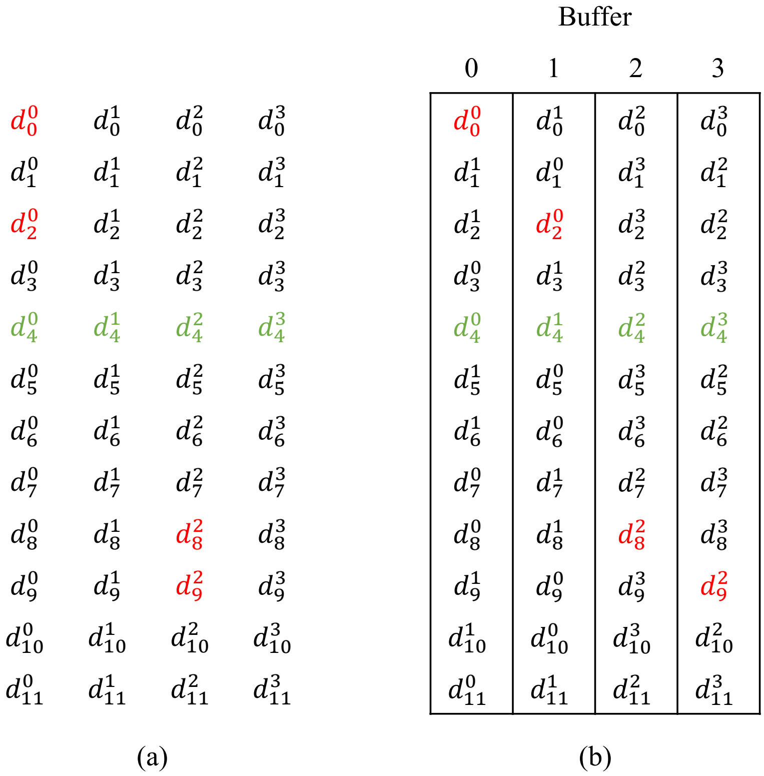
Figure 3.5: (a) A 4×4 array of data. (b) An example of distributing the 4×4 array of data in figure (a) to eight memories in an array processor with four ALUs so that any row, any column, any 2×2 square block, the forward diagonal, and the backward diagonal can be accessed without conflicts.

conversion table is

$$X = (\lceil \log_2(2k+1) \rceil \times 2k) \times (n \times k^{n-1}) \text{ bits.}$$

The total memory overhead for the two conversion tables is $2X$ bits. For example, to emulate a 4-ary 6-tree NoC (the largest k-ary n-tree that the current version of the proposed emulator supports), 393,216 bits of memory for the two tables are required. In actual implementation, the number of 36Kb BRAMs required is 16. This is only 1.55% of the total number of BRAMs of the currently used FPGA. Therefore, the memory overhead for the port ID conversion is small.

The idea of rearranging data for efficient mapping to memories can be traced back to the array processor architecture first proposed in the 1970s [6]. Figure 3.4 shows an array processor with N Arithmetic Logic Units (ALUs) connected to M memories. The two alignment networks



d_0^0	d_0^1	d_0^2	d_0^3
d_1^0	d_1^1	d_1^2	d_1^3
d_2^0	d_2^1	d_2^2	d_2^3
d_3^0	d_3^1	d_3^2	d_3^3
d_4^0	d_4^1	d_4^2	d_4^3
d_5^0	d_5^1	d_5^2	d_5^3
d_6^0	d_6^1	d_6^2	d_6^3
d_7^0	d_7^1	d_7^2	d_7^3
d_8^0	d_8^1	d_8^2	d_8^3
d_9^0	d_9^1	d_9^2	d_9^3
d_{10}^0	d_{10}^1	d_{10}^2	d_{10}^3
d_{11}^0	d_{11}^1	d_{11}^2	d_{11}^3

(a)

Buffer

0	1	2	3
d_0^0	d_0^1	d_0^2	d_0^3
d_1^1	d_1^0	d_1^3	d_1^2
d_2^1	d_2^0	d_2^3	d_2^2
d_3^0	d_3^1	d_3^2	d_3^3
d_4^0	d_4^1	d_4^2	d_4^3
d_5^1	d_5^0	d_5^3	d_5^2
d_6^1	d_6^0	d_6^3	d_6^2
d_7^0	d_7^1	d_7^2	d_7^3
d_8^0	d_8^1	d_8^2	d_8^3
d_9^1	d_9^0	d_9^3	d_9^2
d_{10}^1	d_{10}^0	d_{10}^3	d_{10}^2
d_{11}^0	d_{11}^1	d_{11}^2	d_{11}^3

(b)

Figure 3.6: (a) Inter-router communication data in the emulation of a 2-ary 3-tree (Figure 3.2(a)): d_r^p indicates the output data of physical port p of router r; the output and input data of router R_4 (output data: d_4^0, d_4^1, d_4^2, d_4^3; input data: d_0^0, d_2^0, d_8^2, d_9^2) are highlighted in green and red, respectively. (b) Read conflicts are avoided by dividing the out buffer and the in buffer into multiple smaller buffer memories according to the logical port ID instead of the physical port ID: buffer i ($i = 0; 1; 2; 3$) stores output data of logical port i of the routers.

are responsible for rearranging data transferred between the ALUs and the memories. In [6], Lawrie describes how N and M should be chosen as well as how the alignment networks should be designed so that data can be distributed to the memories in a way that, for certain patterns, the ALUs can simultaneously access the memories without conflicts, that is, none of the memories is simultaneously accessed by two or more ALUs. Lawrie also shows an example of distributing a 4×4 array of data to eight memories in an array processor with four ALUs (M = 8, N = 4). Figure 3.5 depicts this example. By distributing the data array as in Figure 3.5(b), any row, any column, any 2×2 square block, the forward diagonal, and the backward diagonal can be accessed without memory conflicts. For example, four slices of data of column 2, $d_0^2, d_1^2, d_2^2, d_3^2$ are stored in four different memories 4, 7, 2, and 5, respectively. Thus, the four ALUs can access this column (each ALU accesses a slice of data) without memory conflicts. The alignment networks rearrange outgoing and incoming data of the ALUs so that for the ALUs the arrangement of data is as shown in Figure 3.5(a).

To describe the difference between the rearrangement of data in the array processor architecture and that in the proposed architecture for emulating k-ary n-trees, let us revisit Figure 3.2. Figure 3.6(a) illustrates the inter-router communication data in the emulation of the 2-ary 3-tree shown in Figure 3.2(a). These data are stored in the out buffer and the in buffer as shown in Figure 3.2(b). In Figure 3.6(a), d_r^p indicates the output data of physical port p of router r. For example, the output data of router R_4 are d_4^0, d_4^1, d_4^2, and d_4^3, which are highlighted in green in Figure 3.6. Emulating R_4 requires d_0^0, d_2^0, d_8^2, and d_9^2, which are highlighted in red in Figure 3.6. Figure 3.6(b) shows the contents of the buffer memories numbered from 0 to $2k - 1$ ($k = 2$ here) inside the out buffer and the in buffer; buffer i (i = 0, $1, \cdots, 2k - 1$) stores output data of logical port i of the routers. We can see that by dividing the out buffer and the in buffer into smaller buffer memories according to the logical port ID instead of the physical port ID, d_0^0, d_2^0, d_8^2, and d_9^2 are stored in different buffers. In this way, read conflicts are avoided, that is, none of the buffers 0, $1, \cdots$, and $2k - 1$ is simultaneously read by two or more readers.

The difference between the rearrangement of data in the array processor architecture and that in the proposed architecture for emulating k-ary n-trees arises from two facts. First, while the array processor architecture considers regular memory access patterns like reading/writing a row or a column of data, this is not the case for the proposed architecture where the read pattern varies according to the k-ary n-tree topology and is complicated. Second, while the number of memory units in the array processor architecture is set sufficiently large so that the data mapping can be realized by some simple functions, the number of memory units in the proposed architecture is equal to the number of threads that simultaneously access them. The proposed data mapping functions do not require adding extra memories like in the array processor architecture. Although two conversion tables from physical port IDs to logical port IDs and vice versa are needed in the rearrangement of data, as described above, their overhead is small.

3.4.4 Proof of Correctness of the Procedure for Calculating Logical Port IDs in k-Ary n-Trees

This section provides a formal proof of correctness of the procedure for calculating the logical port IDs of a k-ary n-tree. The section starts by describing in detail the connection of routers in a k-ary n-tree. After that, the proof is organized in a series of six lemmas and one theorem.

3.4.4.1 Connection of Routers in a k-Ary n-Tree

As mentioned earlier, a k-ary n-tree consists of k^n cores (terminal nodes) connected by n levels of k^{n-1} radix-$2k$ routers (switch nodes). Figure 3.7 shows a 3-ary 3-tree with physical port ID assignment. As illustrated in the figure, the n levels of the tree are consecutively numbered starting from 0 at the root up to the leaves. The routers in each level are numbered from 0 at the leftmost position to $k^{n-1} - 1$ at the rightmost position. In each router, the physical IDs of the

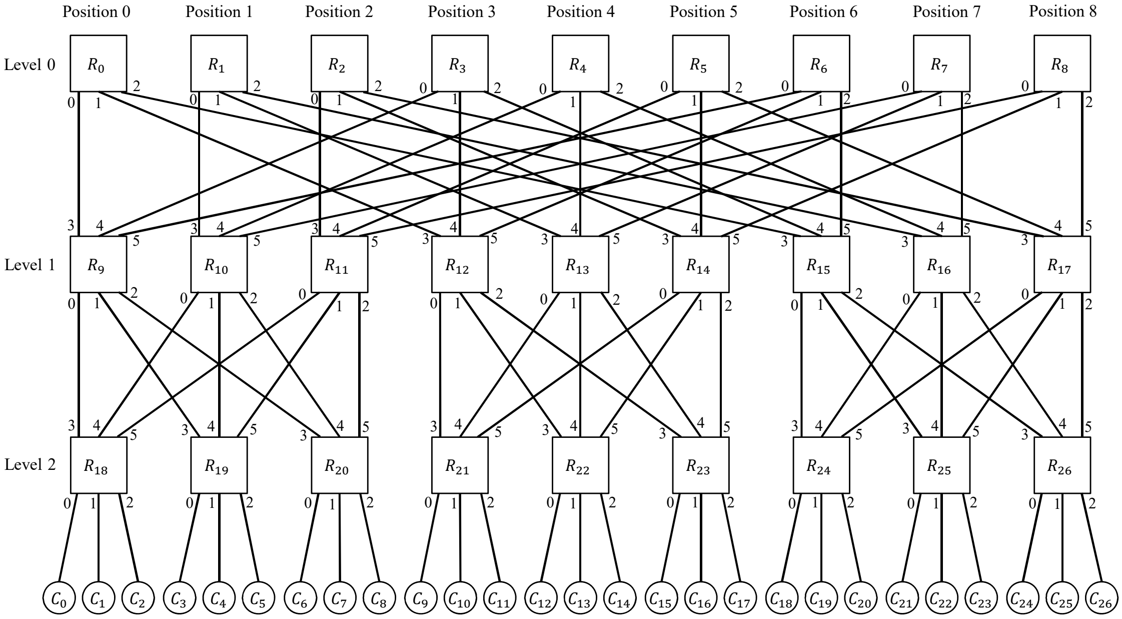


Figure 3.7: k-ary n-tree with physical port ID assignment. A 3-ary 3-tree consists of $k^n = 3^3 = 27$ cores connected by $n = 3$ levels of $k^{n-1} = 3^{3-1} = 9$ radix-$2k$ (radix-6) routers. The levels are consecutively numbered starting from 0 at the root up to the leaves. The routers in each level are numbered from 0 at the leftmost position to $k^{n-1} - 1 = 3^{3-1} - 1 = 8$ at the rightmost position. In each router, the physical IDs of the down ports are from 0 at the leftmost position to $k - 1 = 3 - 1 = 2$ at the rightmost position while the physical IDs of the up ports are from $k = 3$ at the leftmost position to $2k - 1 = 2 \times 3 - 1 = 5$ at the rightmost position.

down ports are from 0 at the leftmost position to $k-1$ at the rightmost position while the physical IDs of the up ports are from k at the leftmost position to $2k - 1$ at the rightmost position.

The connection of routers in a k-ary n-tree is as follows. Port `phyid` (physical port ID = `phyid`) of router R_i such that

$$0 \le \texttt{phyid} \le 2k - 1$$

$$0 \le i \le (n-1) \times k^{n-1}$$

$$R_i \text{ lies at position } p = i\%k^{n-1} \text{ in level } l = \left\lfloor \frac{i}{k^{n-1}} \right\rfloor$$

is connected to port `phyid`′ (physical port ID = `phyid`′) of router $R_{i'}$ (position p' in level l') where

$$\texttt{phyid}' = k + \left\lfloor \frac{p\%k^{n-1-l}}{k^{n-1-l-1}} \right\rfloor \tag{3.8}$$

$$i' = l' \times k^{n-1} + p'$$

$$l' = l + 1$$

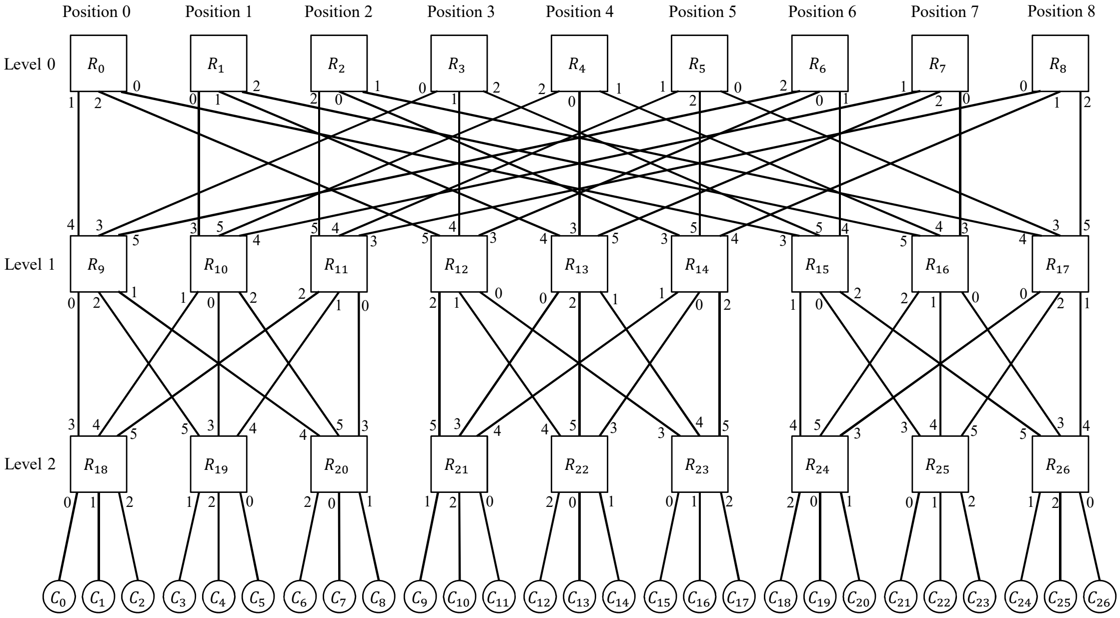


Figure 3.8: k-ary n-tree (3-ary 3-tree) with logical port ID assignment. The logical port ID assignment of a router may be different from that of another.

$$p' = \left\lfloor \frac{p}{k^{n-1-l}} \right\rfloor \times k^{n-1-l} + p\%k^{n-1-l-1} + \texttt{phyid} \times k^{n-1-l-1} \tag{3.9}$$

Figure 3.8 shows the physical port ID assignment in a k-ary n-tree ($k = 3, n = 3$) obtained by using the procedure described in Section 3.4.3.2. This figure and Figure 3.7 are used in the reference examples throughout the proof below.

3.4.4.2 Proof

Lemma 3.4.1.

$$\forall a, b \in \mathbb{N}; b \neq 0 \quad \textit{we have:} \quad \left\lfloor \frac{a}{b} \right\rfloor = \frac{a - a\%b}{b}$$

Proof.

$$\forall a, b \in \mathbb{N} \quad \exists q, r \text{ such that } 0 \leq r < b \text{ and } a = b \times q + r$$

We have:

$$\begin{aligned} \left\lfloor \frac{a}{b} \right\rfloor &= \left\lfloor \frac{b \times q + r}{b} \right\rfloor \\ &= \left\lfloor q + \frac{r}{b} \right\rfloor \\ &= q + \left\lfloor \frac{r}{b} \right\rfloor \end{aligned}$$

$$= q \quad \text{(since } 0 \leq r < b\text{)}$$

Since $a = b \times q + r$ and $0 \leq r < b$, we have $a\%b = r$. Thus,

$$\begin{aligned} \frac{a - a\%b}{b} &= \frac{(b \times q + r) - r}{b} \\ &= \frac{b \times q}{b} \\ &= q \end{aligned}$$

Therefore,

$$\left\lfloor \frac{a}{b} \right\rfloor = \frac{a - a\%b}{b} = q$$

□

Lemma 3.4.2.

$$\forall a, k, m, n \in \mathbb{N}; k \neq 0 \quad \textit{we have:} \quad (a\%k^m)\%k^n = (a\%k^n)\%k^m$$

Proof. Without the loss of generality, we assume that $m \geq n$.

Let $r_1 = a\%k^m$ and $r_2 = r_1\%k^n$ ($0 \leq r_1 < k^m$ and $0 \leq r_2 < k^n$). We have:

$$(a\%k^m)\%k^n = r_1\%k^n = r_2$$

$\forall a, k, m, n \in \mathbb{N} \quad \exists q_1, q_2 \in \mathbb{N}$ such that

$$\begin{aligned} a &= k^m \times q_1 + r_1 \\ r_1 &= k^n \times q_2 + r_2 \end{aligned}$$

Thus,

$$\begin{aligned} a &= k^m \times q_1 + k^n \times q_2 + r_2 \\ &= k^n \times (k^{m-n} \times q_1 + q_2) + r_2 \end{aligned}$$

Since $0 \leq r_2 < k^n \leq k^m$, we have: $(a\%k^n)\%k^m = r_2$. Therefore,

$$(a\%k^m)\%k^n = (a\%k^n)\%k^m = r_2$$

□

Lemma 3.4.3. $\forall a, k, m, n \in \mathbb{N}; k \neq 0; m > n$ *we have:*

$$(((a\%k^m)\%k^{m-1})\%\cdots)\%k^n = a\%k^n$$

Proof. This lemma can be derived from Lemma 3.4.2. □

Lemma 3.4.4. $\forall a, k, m, n \in \mathbb{N}; k \neq 0; m \geq n$ *we have:*

$$((a\%k^m)\%k^n) = a\%k^n$$

Proof. This lemma can also be derived from Lemma 3.4.2. □

Lemma 3.4.5. *We consider a k-ary n-tree. Let* $\texttt{pid1}$, $\texttt{pid2}$ *(*$0 \leq \texttt{pid1}, \ \texttt{pid2} \leq k-1$*;* $\texttt{pid1} < \texttt{pid2}$*) be the physical IDs of two down ports of a router in level l (*$0 \leq l \leq n-2$*). Let p be the position of the router in level l and* $\Delta\texttt{pid} = \texttt{pid2} - \texttt{pid1}$*. Suppose that:*

- *Port* $\texttt{pid1}$ *is connected to port* $\texttt{pid1}_{l+1}$ *(physical port ID =* $\texttt{pid1}_{l+1}$*) of the router at position* $\mathrm{p1}_{l+1}$ *in level* $l+1$.
- $\forall i$ *such that* $1 \leq i \leq n-l-2$*, port* $2k-1-\texttt{pid1}_{l+i}$ *(physical port ID =* $2k-1-\texttt{pid1}_{l+1}$*) of the router at position* $\mathrm{p1}_{l+i}$ *in level* $l+i$ *is connected to port* $\texttt{pid1}_{l+i+1}$ *(physical port ID =* $\texttt{pid1}_{l+i+1}$*) of the router at position* $\mathrm{p1}_{l+i+1}$ *in level* $l+i+1$.
- *Port* $\texttt{pid2}$ *is connected to port* $\texttt{pid2}_{l+1}$ *(physical port ID =* $\texttt{pid2}_{l+1}$*) of the router at position* $\mathrm{p2}_{l+1}$ *in level* $l+1$.
- $\forall i$ *such that* $1 \leq i \leq n-l-2$*, port* $2k-1-\texttt{pid2}_{l+i}$ *(physical port ID =* $2k-1-\texttt{pid2}_{l+1}$*) of the router at position* $\mathrm{p2}_{l+i}$ *in level* $l+i$ *is connected to port* $\texttt{pid2}_{l+i+1}$ *(physical port ID =* $\texttt{pid2}_{l+i+1}$*) of the router at position* $\mathrm{p2}_{l+i+1}$ *in level* $l+i+1$.

Then:

- $\texttt{pid1}_{l+i} = \texttt{pid2}_{l+i} \quad \forall i$ *such that* $1 \leq i \leq n-l-1$.
- $\mathrm{p2}_{l+i} - \mathrm{p1}_{l+i} = \Delta\texttt{pid} \times k^{n-l-2} \quad \forall i$ *such that* $1 \leq i \leq n-l-1$.
- $\left\lfloor \frac{\mathrm{p1}_{l+i}}{k^{n-t}} \right\rfloor = \left\lfloor \frac{\mathrm{p2}_{l+i}}{k^{n-t}} \right\rfloor = \left\lfloor \frac{p}{k^{n-t}} \right\rfloor \quad \forall i, t$ *such that* $1 \leq i \leq n-l-1$ *and* $1 \leq t \leq l+1$

Example. *In Figure 3.7 (3-ary 3-tree;* $k=3, n=3$*), we consider two ports 0 and 2 of router* R_1 *(*$l=0, p=1, \texttt{pid1}=0, \texttt{pid2}=2, \Delta\texttt{pid}=2$*). We have:*

- $\texttt{pid1}_{l+1} = 3$ *(port 3 of router* R_{10}*) and* $\texttt{pid2}_{l+1} = 3$ *(port 3 of router* R_{16}*). So,* $\texttt{pid1}_{l+1} = \texttt{pid2}_{l+1}$.

- $p1_{l+1} = 1$ *(router* R_{10} *is at position 1 in level 1) and* $p2_{l+1} = 7$ *(router* R_{16} *is at position 7 in level 1). So,* $p2_{l+1} - p1_{l+1} = 6 = \Delta\texttt{pid} \times k^{n-l-2}$.

- $\texttt{pid1}_{l+2} = 4$ *(port 4 of router* R_{20}*) and* $\texttt{pid2}_{l+2} = 4$ *(port 4 of router* R_{26}*). So,* $\texttt{pid1}_{l+2} = \texttt{pid2}_{l+2}$.

- $p1_{l+2} = 2$ *(router* R_{20} *is at position 2 in level 2) and* $p2_{l+2} = 8$ *(router* R_{26} *is at position 8 in level 2). So,* $p2_{l+2} - p1_{l+2} = 6 = \Delta\texttt{pid} \times k^{n-l-2}$.

- *We can see that* $\left\lfloor \frac{p1_{l+i}}{k^{n-t}} \right\rfloor = \left\lfloor \frac{p2_{l+i}}{k^{n-t}} \right\rfloor = \left\lfloor \frac{p}{k^{n-t}} \right\rfloor \quad \forall i, t$ *such that* $1 \leq i \leq n - l - 1$ *and* $1 \leq t \leq l + 1$.

Proof. We will prove by induction.

Basis: level $l + 1$**.** By using formula (3.8) and (3.9) presented in section 3.4.4.1, we have:

$$\begin{aligned}
\texttt{pid1}_{l+1} &= \texttt{pid2}_{l+1} \\
&= k + \left\lfloor \frac{p\%k^{n-1-l}}{k^{n-1-l-1}} \right\rfloor \\
p2_{l+i} - p1_{l+i} &= (\texttt{pid2} - \texttt{pid1}) \times k^{n-1-l-1} \\
&= \Delta\texttt{pid} \times k^{n-l-2}
\end{aligned}$$

Next, we will prove that $\left\lfloor \frac{p1_{l+1}}{k^{n-t}} \right\rfloor = \left\lfloor \frac{p2_{l+1}}{k^{n-t}} \right\rfloor = \left\lfloor \frac{p}{k^{n-t}} \right\rfloor \quad \forall t$ such that $1 \leq t \leq l + 1$. Using formula (3.9), we have:

$$\begin{aligned}
\left\lfloor \frac{p1_{l+1}}{k^{n-t}} \right\rfloor &= \left\lfloor \frac{\left\lfloor \frac{p}{k^{n-1-l}} \right\rfloor \times k^{n-1-l} + p\%k^{n-1-l-1} + \texttt{pid1} \times k^{n-1-l-1}}{k^{n-t}} \right\rfloor \\
&= \left\lfloor \frac{\left\lfloor \frac{p}{k^{n-l-1}} \right\rfloor \times k^{n-l-1} + p\%k^{n-l-2} + \texttt{pid1} \times k^{n-l-2}}{k^{n-t}} \right\rfloor \\
&= \left\lfloor \frac{p - p\%k^{n-l-1} + p\%k^{n-l-2} + \texttt{pid1} \times k^{n-l-2}}{k^{n-t}} \right\rfloor \text{ (Lemma 3.4.1)} \\
&= \left\lfloor \frac{p - p\%k^{n-t} + p\%k^{n-t} - p\%k^{n-l-1} + p\%k^{n-l-2} + \texttt{pid1} \times k^{n-l-2}}{k^{n-t}} \right\rfloor \\
&= \left\lfloor \frac{p - p\%k^{n-t}}{k^{n-t}} + \frac{p\%k^{n-t} - p\%k^{n-l-1} + p\%k^{n-l-2} + \texttt{pid1} \times k^{n-l-2}}{k^{n-t}} \right\rfloor \\
&= \left\lfloor \left\lfloor \frac{p}{k^{n-t}} \right\rfloor + \frac{p\%k^{n-t} - p\%k^{n-l-1} + p\%k^{n-l-2} + \texttt{pid1} \times k^{n-l-2}}{k^{n-t}} \right\rfloor \\
&\quad \text{(Lemma 3.4.1)} \\
&= \left\lfloor \frac{p}{k^{n-t}} \right\rfloor + \left\lfloor \frac{p\%k^{n-t} - p\%k^{n-l-1} + p\%k^{n-l-2} + \texttt{pid1} \times k^{n-l-2}}{k^{n-t}} \right\rfloor
\end{aligned}$$

We will prove that $(p\%k^{n-t} - p\%k^{n-l-1} + p\%k^{n-l-2} + \texttt{pid1} \times k^{n-l-2}) < k^{n-t}$. We have:

$$\begin{aligned}
&p\%k^{n-t} - p\%k^{n-l-1} + p\%k^{n-l-2} + \texttt{pid1} \times k^{n-l-2}\\
=&p\%k^{n-t} - (p\%k^{n-t})\%k^{n-l-1} + p\%k^{n-l-2} + \texttt{pid1} \times k^{n-l-2}\\
&\text{(Lemma 3.4.4 with } n-t \geq n-l-1\text{)}\\
=&\left\lfloor \frac{p\%k^{n-t}}{k^{n-l-1}} \right\rfloor + p\%k^{n-l-2} + \texttt{pid1} \times k^{n-l-2}\\
&\text{(Lemma 3.4.1)}
\end{aligned}$$

We also have:

$$\begin{aligned}
p\%k^{n-l-2} + \texttt{pid1} \times k^{n-l-2} &< k^{n-l-2} + (k-1)k^{n-l-2}\\
&= k^{n-l-1}
\end{aligned}$$

- If $t = l+1$, then

$$\begin{aligned}
\left\lfloor \frac{p\%k^{n-t}}{k^{n-l-1}} \right\rfloor + p\%k^{n-l-2} + \texttt{pid1} \times k^{n-l-2} &= \left\lfloor \frac{p\%k^{n-t}}{k^{n-t}} \right\rfloor + p\%k^{n-l-2} +\\
&\quad \texttt{pid1} \times k^{n-l-2}\\
&= p\%k^{n-l-2} + \texttt{pid1} \times k^{n-l-2}\\
&< k^{n-l-1}\\
&= k^{n-t}
\end{aligned}$$

  Thus, $(p\%k^{n-t} - p\%k^{n-l-1} + p\%k^{n-l-2} + \texttt{pid1} \times k^{n-l-2}) < k^{n-t}$

- If $1 \leq t \leq l$, then

$$\begin{aligned}
\left\lfloor \frac{p\%k^{n-t}}{k^{n-l-1}} \right\rfloor + p\%k^{n-l-2} + \texttt{pid1} \times k^{n-l-2} &< \frac{k^{n-t}}{k^{n-l-1}} + p\%k^{n-l-2} + \texttt{pid1} \times k^{n-l-2}\\
&< \frac{k^{n-t}}{k^{n-l-1}} + k^{n-l-1}\\
&= k^{l+1-t} + k^{n-l-1}\\
&< k^{n-2+1-t} + k^{n-l-1} \text{ (since } l \leq n-2\text{)}\\
&< k^{n-t-1} + k^{n-t-1} \text{ (since } t \leq l\text{)}\\
&= 2 \times k^{n-t-1}\\
&\leq k^{n-t} \text{ (since we consider } k \geq 2 \text{ only)}
\end{aligned}$$

Thus, $(p\%k^{n-t} - p\%k^{n-l-1} + p\%k^{n-l-2} + \texttt{pid1} \times k^{n-l-2}) < k^{n-t}$

Therefore, $\forall t$ such that $1 \leq t \leq l+1$, we have $(p\%k^{n-t} - p\%k^{n-l-1} + p\%k^{n-l-2} + \texttt{pid1} \times k^{n-l-2}) < k^{n-t}$. So,

$$\begin{aligned} &\left\lfloor \frac{p\%k^{n-t} - p\%k^{n-l-1} + p\%k^{n-l-2} + \texttt{pid1} \times k^{n-l-2}}{k^{n-t}} \right\rfloor = 0 \\ \Rightarrow &\left\lfloor \frac{p1_{l+1}}{k^{n-t}} \right\rfloor = \left\lfloor \frac{p}{k^{n-t}} \right\rfloor \end{aligned}$$

Similarly, we can prove that

$$\left\lfloor \frac{p2_{l+1}}{k^{n-t}} \right\rfloor = \left\lfloor \frac{p}{k^{n-t}} \right\rfloor$$

Therefore,

$$\left\lfloor \frac{p1_{l+1}}{k^{n-t}} \right\rfloor = \left\lfloor \frac{p2_{l+1}}{k^{n-t}} \right\rfloor = \left\lfloor \frac{p}{k^{n-t}} \right\rfloor$$

Inductive step: Assume that the lemma holds for level $l+i$, that is,

- $\texttt{pid1}_{l+i} = \texttt{pid2}_{l+i}$
- $p2_{l+i} - p1_{l+i} = \Delta\texttt{pid} \times k^{n-l-2}$
- $\left\lfloor \frac{p1_{l+i}}{k^{n-t}} \right\rfloor = \left\lfloor \frac{p2_{l+i}}{k^{n-t}} \right\rfloor = \left\lfloor \frac{p}{k^{n-t}} \right\rfloor$

We must prove that the lemma holds for level $l+i+1$, that is,

- $\texttt{pid1}_{l+i+1} = \texttt{pid2}_{l+i+1}$
- $p2_{l+i+1} - p1_{l+i+1} = \Delta\texttt{pid} \times k^{n-l-2}$
- $\left\lfloor \frac{p1_{l+i+1}}{k^{n-t}} \right\rfloor = \left\lfloor \frac{p2_{l+i+1}}{k^{n-t}} \right\rfloor = \left\lfloor \frac{p}{k^{n-t}} \right\rfloor$

Using formula (3.8) presented in Section 3.4.4.1, we have:

$$\begin{aligned} \texttt{pid2}_{l+i+1} &= k + \left\lfloor \frac{p2_{l+i}\%k^{n-1-(l+i)}}{k^{n-1-(l+i)-1}} \right\rfloor \\ &= k + \left\lfloor \frac{p2_{l+i}\%k^{n-l-i-1}}{k^{n-l-i-2}} \right\rfloor \\ &= k + \left\lfloor \frac{(p1_{l+i} + \Delta\texttt{pid} \times k^{n-l-2})\%k^{n-l-i-1}}{k^{n-l-i-2}} \right\rfloor \\ &= k + \left\lfloor \frac{(p1_{l+i}\%k^{n-l-i-1} + (\Delta\texttt{pid} \times k^{n-l-2})\%k^{n-l-i-1})\%k^{n-l-i-1}}{k^{n-l-i-2}} \right\rfloor \end{aligned}$$

$$\begin{aligned}
&= k + \left\lfloor \frac{(p1_{l+i}\%k^{n-l-i-1})\%k^{n-l-i-1}}{k^{n-l-i-2}} \right\rfloor \quad \text{(since } k^{n-l-2} \text{ divides } k^{n-l-i-1}\text{)}\\
&= k + \left\lfloor \frac{p1_{l+i}\%k^{n-l-i-1}}{k^{n-l-i-2}} \right\rfloor\\
&= k + \left\lfloor \frac{p1_{l+i}\%k^{n-1-(l+i)}}{k^{n-1-(l+i)-1}} \right\rfloor\\
&= \mathtt{pid1}_{l+i+1}
\end{aligned}$$

Using formula (3.9) presented in Section 3.4.4.1, we have:

$$\begin{aligned}
p1_{l+i+1} &= \left\lfloor \frac{p1_{l+i}}{k^{n-1-(l+i)}} \right\rfloor \times k^{n-1-(l+i)} + p1_{l+i}\%k^{n-1-(l+i)-1} +\\
&\quad (2k-1-\mathtt{pid1}_{l+i}) \times k^{n-1-(l+i)-1}\\
&= \left\lfloor \frac{p1_{l+i}}{k^{n-l-i-1}} \right\rfloor \times k^{n-l-i-1} + p1_{l+i}\%k^{n-l-i-2} + (2k-1-\mathtt{pid1}_{l+i}) \times k^{n-l-i-2}\\
&= \frac{p1_{l+i} - p1_{l+i}\%k^{n-l-i-1}}{k^{n-l-i-1}} \times k^{n-l-i-1} + p1_{l+i}\%k^{n-l-i-2} +\\
&\quad (2k-1-\mathtt{pid1}_{l+i}) \times k^{n-l-i-2} \text{ (Lemma 3.4.1)}\\
&= p1_{l+i} - p1_{l+i}\%k^{n-l-i-1} + p1_{l+i}\%k^{n-l-i-2} + (2k-1-\mathtt{pid1}_{l+i}) \times k^{n-l-i-2}
\end{aligned}$$

Similarly, we have:

$$p2_{l+i+1} = p2_{l+i} - p2_{l+i}\%k^{n-l-i-1} + p2_{l+i}\%k^{n-l-i-2} + (2k-1-\mathtt{pid2}_{l+i}) \times k^{n-l-i-2}$$

Since $\mathtt{pid1}_{l+i} = \mathtt{pid2}_{l+i}$ and $p2_{l+i} - p1_{l+i} = \Delta\mathtt{pid} \times k^{n-l-2}$ (inductive hypothesis), we have:

$$\begin{aligned}
p2_{l+i+1} - p1_{l+i+1} &= p2_{l+i} - p2_{l+i}\%k^{n-l-i-1} + p2_{l+i}\%k^{n-l-i-2} -\\
&\quad p1_{l+i} + p1_{l+i}\%k^{n-l-i-1} - p1_{l+i}\%k^{n-l-i-2}\\
&= p1_{l+i} + \Delta\mathtt{pid} \times k^{n-l-2} - (p1_{l+i} + \Delta\mathtt{pid} \times k^{n-l-2})\%k^{n-l-i-1} +\\
&\quad (p1_{l+i} + \Delta\mathtt{pid} \times k^{n-l-2})\%k^{n-l-i-2} -\\
&\quad p1_{l+i} + p1_{l+i}\%k^{n-l-i-1} - p1_{l+i}\%k^{n-l-i-2}\\
&= \Delta\mathtt{pid} \times k^{n-l-2} -\\
&\quad (p1_{l+i}\%k^{n-l-i-1} + (\Delta\mathtt{pid} \times k^{n-l-2})\%k^{n-l-i-1})\%k^{n-l-i-1} +\\
&\quad (p1_{l+i}\%k^{n-l-i-2} + (\Delta\mathtt{pid} \times k^{n-l-2})\%k^{n-l-i-2})\%k^{n-l-i-2} +\\
&\quad p1_{l+i}\%k^{n-l-i-1} - p1_{l+i}\%k^{n-l-i-2}
\end{aligned}$$

$$\begin{aligned}
&= \Delta\texttt{pid} \times k^{n-l-2} - (p1_{l+i}\%k^{n-l-i-1} + 0)\%k^{n-l-i-1} + \\
&\quad (p1_{l+i}\%k^{n-l-i-2} + 0)\%k^{n-l-i-2} + \\
&\quad p1_{l+i}\%k^{n-l-i-1} - p1_{l+i}\%k^{n-l-i-2} \\
&\quad (\text{since } k^{n-l-2} \text{ divides } k^{n-l-i-1} \text{ and } k^{n-l-i-2}) \\
&= \Delta\texttt{pid} \times k^{n-l-2} - p1_{l+i}\%k^{n-l-i-1} + p1_{l+i}\%k^{n-l-i-2} + \\
&\quad p1_{l+i}\%k^{n-l-i-1} - p1_{l+i}\%k^{n-l-i-2} \\
&= \Delta\texttt{pid} \times k^{n-l-2}
\end{aligned}$$

By a similar proof as in the basis step, we have:

$$\left\lfloor \frac{p1_{l+i+1}}{k^{n-t}} \right\rfloor = \left\lfloor \frac{p1_{l+i}}{k^{n-t}} \right\rfloor$$
$$\left\lfloor \frac{p2_{l+i+1}}{k^{n-t}} \right\rfloor = \left\lfloor \frac{p2_{l+i}}{k^{n-t}} \right\rfloor$$

Using the inductive hypothesis, we have:

$$\left\lfloor \frac{p1_{l+i}}{k^{n-t}} \right\rfloor = \left\lfloor \frac{p2_{l+i}}{k^{n-t}} \right\rfloor = \left\lfloor \frac{p}{k^{n-t}} \right\rfloor$$

Therefore,

$$\left\lfloor \frac{p1_{l+i+1}}{k^{n-t}} \right\rfloor = \left\lfloor \frac{p2_{l+i+1}}{k^{n-t}} \right\rfloor = \left\lfloor \frac{p}{k^{n-t}} \right\rfloor$$

□

Lemma 3.4.6. *We consider a k-ary n-tree. Let $\texttt{pid}$ ($0 \leq \texttt{pid} \leq k-1$) be the physical ID of a down port of a router in level l ($0 \leq l \leq n-2$). We call this router r. Suppose that:*

- *Port $\texttt{pid}$ is connected to port $\texttt{pid}_{l+1}$ (physical port ID = $\texttt{pid}_{l+1}$) of the router at position p_{l+1} in level $l+1$.*
- *$\forall i$ such that $1 \leq i \leq n-l-2$, port $2k-1-\texttt{pid}_{l+i}$ (physical port ID = $2k-1-\texttt{pid}_{l+1}$) of the router at position p_{l+i} in level $l+i$ is connected to port $\texttt{pid}_{l+i+1}$ (physical port ID = $\texttt{pid}_{l+i+1}$) of the router at position p_{l+i+1} in level $l+i+1$.*
- *r_{l+i} ($1 \leq i \leq n-l-1$) is the router at position p_{l+i} in level $l+i$.*

Then, the logical ID of the port with physical ID $\texttt{pid}$ of router r is

- *$f(r_{n-1}, 2k-1-\texttt{pid}_{n-1})$ if $(n-l)\%2 \neq 0$, or*

- $k-1-f(r_{n-1}, 2k-1-\texttt{pid}_{n-1})$ *otherwise.*

where f is defined in formula (3.5) in Section 3.4.3.2.

Example 1. *In Figure 3.7 (3-ary 3-tree; $k=3, n=3$), we consider physical port 0 of router R_3 ($l=0, r=R_3, \texttt{pid}=0$). We have:*

- $\texttt{pid}_1 = 4, r_1 = R_9$ *(physical port 4 of router R_9)*
- $\texttt{pid}_2 = 3, r_2 = R_{19}$ *(physical port 3 of router R_{19})*

Since $n-l=3-0=3\%2=1 \neq 0$, the logical ID of the port with physical ID 0 of router R3 is given by

$$\begin{aligned} f(R_{19}, 2k-1-\texttt{pid}_2) &= f(R_{19}, 2) \\ &= \left(2 + \left\lfloor \frac{1}{3^1} \right\rfloor + \left\lfloor \frac{1\%3^1}{3^0} \right\rfloor\right)\%3 \\ &= 3\%3 = 0 \textit{ (shown in Figure 3.8)} \end{aligned}$$

Example 2. *In Figure 3.7 (3-ary 3-tree; $k=3, n=3$), we consider physical port 1 of router R_9 ($l=1, r=R_9, \texttt{pid}=1$). We have:*

- $\texttt{pid}_2 = 3, r_2 = R_{19}$ *(physical port 3 of router R_{19})*

Since $n-l=3-1=2\%2=0$, the logical ID of the port with physical ID 1 of router R9 is given by

$$\begin{aligned} k-1-f(R_{19}, 2k-1-\texttt{pid}_2) &= 3-1-f(R_{19}, 2) \\ &= 3-1-0 = 2 \textit{ (shown in Figure 3.8)} \end{aligned}$$

Proof. Using formula (3.6) and formula (3.7) in Section 3.4.3.2, we have:

$$\begin{aligned} f(r_{n-1}, \texttt{pid}_{n-1}) &= 2k-1-f(r_{n-1}, 2k-1-\texttt{pid}_{n-1}) \\ f(r_{n-2}, 2k-1-\texttt{pid}_{n-2}) &= f(r_{n-1}, \texttt{pid}_{n-1}) - k \\ &= k-1-f(r_{n-1}, 2k-1-\texttt{pid}_{n-1}) \qquad (3.10) \\ fr_{n-2}, \texttt{pid}_{n-2}) &= 2k-1-f(r_{n-2}, 2k-1-\texttt{pid}_{n-2}) \\ &= 2k-1-k+1+f(r_{n-1}, 2k-1-\texttt{pid}_{n-1}) \\ &= k+f(r_{n-1}, 2k-1-\texttt{pid}_{n-1}) \\ f(r_{n-3}, 2k-1-\texttt{pid}_{n-3}) &= f(r_{n-2}, \texttt{pid}_{n-2}) - k \\ &= k+f(r_{n-1}, 2k-1-\texttt{pid}_{n-1}) - k \\ &= f(r_{n-1}, 2k-1-\texttt{pid}_{n-1}) \qquad (3.11) \end{aligned}$$

By repeating the above calculation, we will eventually reach physical port $\texttt{pid}$ of router r and the logical port ID corresponding to the physical port ID $\texttt{pid}$ is calculated by either formula (3.10) or formula (3.11) depending on level l, that is,

- $f(r_{n-1}, 2k-1-\texttt{pid}_{n-1})$ if $(n-l)\%2 \neq 0$, or
- $k-1-f(r_{n-1}, 2k-1-\texttt{pid}_{n-1})$ otherwise.

□

Theorem 3.4.7. *We consider a k-ary n-tree. Let $\texttt{pid1, pid2}$ ($0 \leq \texttt{pid1, pid2} \leq k-1$; $\texttt{pid1} < \texttt{pid2}$) be the physical IDs of two down ports of a router in level l ($0 \leq l \leq n-1$). We call this router r. Let R be the set of routers in the network, and $f : R \times [0, 2k-1] \to [0, 2k-1]$ be the port ID mapping funtion described in Section 3.4.3.2 (f maps each physical port ID of each router to a logical port ID). Then:*

$$f(r, \texttt{pid1}) \neq f(r, \texttt{pid2})$$

In other words, there is no duplication of logical port IDs in every router or the procedure for calculating the logical port IDs described in Section 3.4.3.2 is correct (because the logical port IDs of the up ports are calculated based on the logical port IDs of the down ports as shown in formula (3.6) in Section 3.4.3.2).

Proof. We will prove this theorem using the lemmas presented above.
We first rewrite formula (3.5) in Section 3.4.3.2 as follows.

$$f(r, \texttt{phyid}) = h(r, \texttt{phyid})\%k$$

where

$$h(r, \texttt{phyid}) = \texttt{phyid} + \left\lfloor \frac{p}{k^{n-2}} \right\rfloor + \left\lfloor \frac{p\%k^{n-2}}{k^{n-3}} \right\rfloor + \left\lfloor \frac{\left(p\%k^{n-2}\right)\%k^{n-3}}{k^{n-4}} \right\rfloor + \cdots + \left\lfloor \frac{\left(\left(\left(p\%k^{n-2}\right)\%k^{n-3}\right)\%\cdots\right)\%k^{1}}{k^{0}} \right\rfloor$$

Now we prove the theorem. Let p be the position of r in level l and $\Delta\texttt{pid} = \texttt{pid2} - \texttt{pid1}$. Since $0 \leq \texttt{pid1, pid2} \leq k-1$ and $\texttt{pid1} < \texttt{pid2}$, we have: $1 \leq \Delta\texttt{pid} \leq k-1$.
Case 1: $l = n-1$**.** It is trivial that:

$$\begin{aligned} h(r, \texttt{pid2}) - h(r, \texttt{pid1}) &= \texttt{pid2} - \texttt{pid1} \\ &= \Delta\texttt{pid} \end{aligned}$$

Since $1 \leq \Delta\texttt{pid} \leq k-1$, we have: $(h(r,\texttt{pid2}) - h(r,\texttt{pid1}))\%k \neq 0$. Thus,

$$h(r,\texttt{pid1})\%k \neq h(r,\texttt{pid2})\%k$$
$$\text{that is, } f(r,\texttt{pid1}) \neq f(r,\texttt{pid2})$$

Case 2: $l \leq n-2$**.** Suppose that:

- Port $\texttt{pid1}$ is connected to port $\texttt{pid1}_{l+1}$ (physical port ID = $\texttt{pid1}_{l+1}$) of the router at position $p1_{l+1}$ in level $l+1$.
- $\forall i$ such that $1 \leq i \leq n-l-2$, port $2k-1-\texttt{pid1}_{l+i}$ (physical port ID = $2k-1-\texttt{pid1}_{l+1}$) of the router at position $p1_{l+i}$ in level $l+i$ is connected to port $\texttt{pid1}_{l+i+1}$ (physical port ID = $\texttt{pid1}_{l+i+1}$) of the router at position $p1_{l+i+1}$ in level $l+i+1$.
- Port $\texttt{pid2}$ is connected to port $\texttt{pid2}_{l+1}$ (physical port ID = $\texttt{pid2}_{l+1}$) of the router at position $p2_{l+1}$ in level $l+1$.
- $\forall i$ such that $1 \leq i \leq n-l-2$, port $2k-1-\texttt{pid2}_{l+i}$ (physical port ID = $2k-1-\texttt{pid2}_{l+1}$) of the router at position $p2_{l+i}$ in level $l+i$ is connected to port $\texttt{pid2}_{l+i+1}$ (physical port ID = $\texttt{pid2}_{l+i+1}$) of the router at position $p2_{l+i+1}$ in level $l+i+1$.
- $r1_{l+i}$ $(1 \leq i \leq n-l-1)$ is the router at position $p1_{l+i}$ in level $l+i$.
- $r2_{l+i}$ $(1 \leq i \leq n-l-1)$ is the router at position $p2_{l+i}$ in level $l+i$.

For example, in Figure 3.7 (3-ary 3-tree; $k=3, n=3$), we consider two physical ports 0 and 2 of router R_1 ($l=0, p=1, \texttt{pid1}=0, \texttt{pid2}=2, \Delta\texttt{pid}=2$). We have:

- $\texttt{pid1}_{l+1} = \texttt{pid1}_1 = 3; r1_{l+1} = r1_1 = R_{10}$ (physical port 3 of router R_{10}) and $\texttt{pid2}_{l+1} = \texttt{pid2}_1 = 3; r2_{l+1} = r2_1 = R_{16}$ (physical port 3 of router R_{16}).
- $p1_{l+1} = p1_1 = 1$ (router R_{10} is at position 1 in level 1) and $p2_{l+1} = p2_1 = 7$ (router R_{16} is at position 7 in level 1).
- $\texttt{pid1}_{l+2} = \texttt{pid1}_2 = 4; r1_{l+2} = r1_2 = R_{20}$ (physical port 4 of router R_{20}) and $\texttt{pid2}_{l+2} = \texttt{pid2}_2 = 4; r2_{l+2} = r2_2 = R_{26}$ (physical port 4 of router R_{26}).
- $p1_{l+2} = p1_2 = 2$ (router R_{20} is at position 2 in level 2) and $p2_{l+2} = p2_2 = 8$ (router R_{26} is at position 8 in level 2).

According to Lemma 3.4.6, either

$$f(r,\texttt{pid1}) = f(r1_{n-1}, 2k-1-\texttt{pid1}_{n-1}), \text{ and}$$
$$f(r,\texttt{pid2}) = f(r2_{n-1}, 2k-1-\texttt{pid2}_{n-1}) \text{ (if } (n-l)\%2 \neq 0)$$

or

$$f(r,\texttt{pid1}) = k - 1 - f(r1_{n-1}, 2k - 1 - \texttt{pid1}_{n-1}), \text{ and}$$
$$f(r,\texttt{pid2}) = k - 1 - f(r2_{n-1}, 2k - 1 - \texttt{pid2}_{n-1}) \text{ (if } (n-l)\%2 = 0)$$

In both cases, $f(r,\texttt{pid1}) \neq f(r,\texttt{pid2})$ if and only if $f(r1_{n-1}, 2k-1-\texttt{pid1}_{n-1}) \neq f(r2_{n-1}, 2k-1-\texttt{pid2}_{n-1})$. Thus, below we will prove that $f(r1_{n-1}, 2k-1-\texttt{pid1}_{n-1}) \neq f(r2_{n-1}, 2k-1-\texttt{pid2}_{n-1})$.

Let $x = 2k - 1 - \texttt{pid1}_{n-1}$ and $y = p1_{n-1}$. According to Lemma 3.4.5, we have:

$$\texttt{pid1}_{n-1} = \texttt{pid2}_{n-1}, \text{ and}$$
$$p2_{n-1} - p1_{n-1} = \Delta\texttt{pid} \times k^{n-l-2}$$

Thus,

$$2k - 1 - \texttt{pid2}_{n-1} = x, \text{ and}$$
$$p2_{n-1} = y + \Delta\texttt{pid} \times k^{n-l-2}$$

Since $p1_{n-1}$ and $p2_{n-1}$ are the positions of routers $r1_{n-1}$ and $r2_{n-1}$ in level $n-1$, both y and $y + \Delta\texttt{pid} \times k^{n-l-2}$ are smaller than k^{n-1} (due to the fact that the number of routers per level is k^{n-1}).

We have:

$$\begin{aligned}
h(r1_{n-1}, 2k-1-\texttt{pid1}_{n-1}) &= h(r1_{n-1}, x) \\
&= x + \left\lfloor \frac{y}{k^{n-2}} \right\rfloor + \left\lfloor \frac{y\%k^{n-2}}{k^{n-3}} \right\rfloor + \left\lfloor \frac{\left(y\%k^{n-2}\right)\%k^{n-3}}{k^{n-4}} \right\rfloor + \cdots + \\
&\quad \left\lfloor \frac{\left(\left(\left(y\%k^{n-2}\right)\%k^{n-3}\right)\%\cdots\right)\%k^{1}}{k^{0}} \right\rfloor \\
&= x + \left\lfloor \frac{y\%k^{n-1}}{k^{n-2}} \right\rfloor + \left\lfloor \frac{y\%k^{n-2}}{k^{n-3}} \right\rfloor + \\
&\quad \left\lfloor \frac{\left(y\%k^{n-2}\right)\%k^{n-3}}{k^{n-4}} \right\rfloor + \cdots + \\
&\quad \left\lfloor \frac{\left(\left(\left(y\%k^{n-2}\right)\%k^{n-3}\right)\%\cdots\right)\%k^{1}}{k^{0}} \right\rfloor \text{ (since } y < k^{n-1}) \\
&= x + \left\lfloor \frac{y\%k^{n-1}}{k^{n-2}} \right\rfloor + \left\lfloor \frac{y\%k^{n-2}}{k^{n-3}} \right\rfloor + \\
&\quad \left\lfloor \frac{y\%k^{n-3}}{k^{n-4}} \right\rfloor + \cdots + \left\lfloor \frac{y\%k^{1}}{k^{0}} \right\rfloor
\end{aligned} \tag{3.12}$$

(Lemma 3.4.3)

$$
\begin{aligned}
h(r2_{n-1}, 2k-1-\mathtt{pid2}_{n-1}) &= h(r2_{n-1}, x) \\
&= x + \left\lfloor \frac{y + \Delta\mathtt{pid} \times k^{n-l-2}}{k^{n-2}} \right\rfloor + \\
&\quad \left\lfloor \frac{(y + \Delta\mathtt{pid} \times k^{n-l-2})\%k^{n-2}}{k^{n-3}} \right\rfloor + \\
&\quad \left\lfloor \frac{\left((y + \Delta\mathtt{pid} \times k^{n-l-2})\%k^{n-2}\right)\%k^{n-3}}{k^{n-4}} \right\rfloor + \cdots + \\
&\quad \left\lfloor \frac{\left(\left(\left((y + \Delta\mathtt{pid} \times k^{n-l-2})\%k^{n-2}\right)\%k^{n-3}\right)\%\cdots\right)\%k^{1}}{k^{0}} \right\rfloor \\
&= x + \left\lfloor \frac{(y + \Delta\mathtt{pid} \times k^{n-l-2})\%k^{n-1}}{k^{n-2}} \right\rfloor + \\
&\quad \left\lfloor \frac{(y + \Delta\mathtt{pid} \times k^{n-l-2})\%k^{n-2}}{k^{n-3}} \right\rfloor + \\
&\quad \left\lfloor \frac{\left((y + \Delta\mathtt{pid} \times k^{n-l-2})\%k^{n-2}\right)\%k^{n-3}}{k^{n-4}} \right\rfloor + \cdots + \\
&\quad \left\lfloor \frac{\left(\left(\left((y + \Delta\mathtt{pid} \times k^{n-l-2})\%k^{n-2}\right)\%k^{n-3}\right)\%\cdots\right)\%k^{1}}{k^{0}} \right\rfloor \\
&\quad \text{(since } y + \Delta\mathtt{pid} \times k^{n-l-2} < k^{n-1}\text{)} \\
&= x + \left\lfloor \frac{(y + \Delta\mathtt{pid} \times k^{n-l-2})\%k^{n-1}}{k^{n-2}} \right\rfloor + \\
&\quad \left\lfloor \frac{(y + \Delta\mathtt{pid} \times k^{n-l-2})\%k^{n-2}}{k^{n-3}} \right\rfloor + \\
&\quad \left\lfloor \frac{(y + \Delta\mathtt{pid} \times k^{n-l-2})\%k^{n-3}}{k^{n-4}} \right\rfloor + \\
&\quad \left\lfloor \frac{(y + \Delta\mathtt{pid} \times k^{n-l-2})\%k^{1}}{k^{0}} \right\rfloor
\end{aligned}
\tag{3.13}
$$

(Lemma 3.4.3)

Using Lemma 3.4.1, we have: $\forall a, b \in \mathbb{N}; b \neq 0$

$$
\begin{aligned}
\left\lfloor \frac{a}{b} \right\rfloor &= \frac{a - a\%b}{b} \\
\Rightarrow a\%b &= a - \left\lfloor \frac{a}{b} \right\rfloor \times b
\end{aligned}
\tag{3.14}
$$

By using formula (3.14), we have: $\forall i = 1, 2, \cdots, n-1$

$$\begin{aligned}\left\lfloor \frac{y\%k^{n-i}}{k^{n-i-1}} \right\rfloor &= \left\lfloor \frac{y - \lfloor \frac{y}{k^{n-i}} \rfloor \times k^{n-i}}{k^{n-i-1}} \right\rfloor \\ &= \left\lfloor \frac{y}{k^{n-i-1}} - \left\lfloor \frac{y}{k^{n-i}} \right\rfloor \times k \right\rfloor \\ &= \left\lfloor \frac{y}{k^{n-i-1}} \right\rfloor - \left\lfloor \frac{y}{k^{n-i}} \right\rfloor \times k \end{aligned} \tag{3.15}$$

Similarly, $\forall i = 1, 2, \cdots, n-1$

$$\left\lfloor \frac{(y + \Delta\texttt{pid} \times k^{n-l-2})\%k^{n-i}}{k^{n-i-1}} \right\rfloor = \left\lfloor \frac{y + \Delta\texttt{pid} \times k^{n-l-2}}{k^{n-i-1}} \right\rfloor - \left\lfloor \frac{y + \Delta\texttt{pid} \times k^{n-l-2}}{k^{n-i}} \right\rfloor \times k \tag{3.16}$$

By applying formula (3.15) to formula (3.12), we have:

$$\begin{aligned} h(r1_{n-1}, x) &= x + \left\lfloor \frac{y}{k^0} \right\rfloor - \left\lfloor \frac{y}{k^{n-1}} \right\rfloor \times k - (k-1) \times \sum_{i=2}^{n-1} \left\lfloor \frac{y}{k^{n-i}} \right\rfloor \\ &= x + y - (k-1) \times \sum_{i=2}^{n-1} \left\lfloor \frac{y}{k^{n-i}} \right\rfloor \\ &\quad \text{(since } \left\lfloor \frac{y}{k^{n-1}} \right\rfloor = 0 \text{ due to the fact that } y < k^{n-1}\text{)} \end{aligned}$$

Similarly, by applying formula (3.16) to formula (3.13), we have:

$$h(r2_{n-1}, x) = x + (y + \Delta\texttt{pid} \times k^{n-l-2}) - (k-1) \times \sum_{i=2}^{n-1} \left\lfloor \frac{y + \Delta\texttt{pid} \times k^{n-l-2}}{k^{n-i}} \right\rfloor$$

Thus,

$$\begin{aligned} & h(r2_{n-1}, 2k-1-\texttt{pid2}_{n-1}) - h(r1_{n-1}, 2k-1-\texttt{pid1}_{n-1}) \\ &= h(r2_{n-1}, x) - h(r1_{n-1}, x) \\ &= \Delta\texttt{pid} \times k^{n-l-2} - (k-1) \times \sum_{i=2}^{n-1} \left(\left\lfloor \frac{y + \Delta\texttt{pid} \times k^{n-l-2}}{k^{n-i}} \right\rfloor - \left\lfloor \frac{y}{k^{n-i}} \right\rfloor \right) \end{aligned}$$

- If $l + 2 \le i \le n-1$, then

$$\begin{aligned} \left\lfloor \frac{y + \Delta\texttt{pid} \times k^{n-l-2}}{k^{n-i}} \right\rfloor &= \left\lfloor \frac{y}{k^{n-i}} + \Delta\texttt{pid} \times k^{i-l-2} \right\rfloor \\ &= \left\lfloor \frac{y}{k^{n-i}} \right\rfloor + \Delta\texttt{pid} \times k^{i-l-2} \end{aligned}$$

Thus,

$$\left\lfloor \frac{y + \Delta\texttt{pid} \times k^{n-l-2}}{k^{n-i}} \right\rfloor - \left\lfloor \frac{y}{k^{n-i}} \right\rfloor = \Delta\texttt{pid} \times k^{i-l-2}$$

- If $2 \leq i \leq l+1$, then

$$\begin{aligned} \left\lfloor \frac{y + \Delta\texttt{pid} \times k^{n-l-2}}{k^{n-i}} \right\rfloor &= \left\lfloor \frac{p2_{n-1}}{k^{n-i}} \right\rfloor \\ &= \left\lfloor \frac{p}{k^{n-i}} \right\rfloor \text{ (Lemma 3.4.5)} \\ \left\lfloor \frac{y}{k^{n-l-2}} \right\rfloor &= \left\lfloor \frac{p1_{n-1}}{k^{n-i}} \right\rfloor \\ &= \left\lfloor \frac{p}{k^{n-i}} \right\rfloor \text{ (Lemma 3.4.5)} \end{aligned}$$

Thus,

$$\left\lfloor \frac{y + \Delta\texttt{pid} \times k^{n-l-2}}{k^{n-i}} \right\rfloor - \left\lfloor \frac{y}{k^{n-i}} \right\rfloor = 0$$

Therefore, we have:

$$\begin{aligned} & h(r2_{n-1}, 2k-1-\texttt{pid2}_{n-1}) - h(r1_{n-1}, 2k-1-\texttt{pid1}_{n-1}) \\ &= \Delta\texttt{pid} \times k^{n-l-2} - (k-1) \times \sum_{i=2}^{n-1} \left(\left\lfloor \frac{y + \Delta\texttt{pid} \times k^{n-l-2}}{k^{n-i}} \right\rfloor - \left\lfloor \frac{y}{k^{n-i}} \right\rfloor \right) \\ &= \Delta\texttt{pid} \times k^{n-l-2} - (k-1) \times \sum_{i=l+2}^{n-1} \left(\Delta\texttt{pid} \times k^{i-l-2} \right) \\ &= \Delta\texttt{pid} \times k^{n-l-2} - (k-1) \times \Delta\texttt{pid} \times \sum_{i=l+2}^{n-1} \left(k^{i-l-2} \right) \\ &= \Delta\texttt{pid} \times k^{n-l-2} - (k-1) \times \Delta\texttt{pid} \times \left(k^0 + k^1 + \cdots + k^{n-l-3} \right) \\ &= \Delta\texttt{pid} \times k^{n-l-2} - (k-1) \times \Delta\texttt{pid} \times \frac{k^{n-l-2}-1}{k-1} \\ &= \Delta\texttt{pid} \times k^{n-l-2} - \Delta\texttt{pid} \times (k^{n-l-2}-1) \\ &= \Delta\texttt{pid} \end{aligned}$$

Since $1 \leq \Delta\texttt{pid} \leq k-1$, we have:

$$\begin{aligned} & (h(r2_{n-1}, 2k-1-\texttt{pid2}_{n-1}) - h(r1_{n-1}, 2k-1-\texttt{pid1}_{n-1})) \,\%k \neq 0 \\ \Rightarrow & h(r1_{n-1}, 2k-1-\texttt{pid1}_{n-1})\%k \neq h(r2_{n-1}, 2k-1-\texttt{pid2}_{n-1})\%k \\ \Rightarrow & f(r1_{n-1}, 2k-1-\texttt{pid1}_{n-1}) \neq f(r2_{n-1}, 2k-1-\texttt{pid2}_{n-1}) \end{aligned}$$

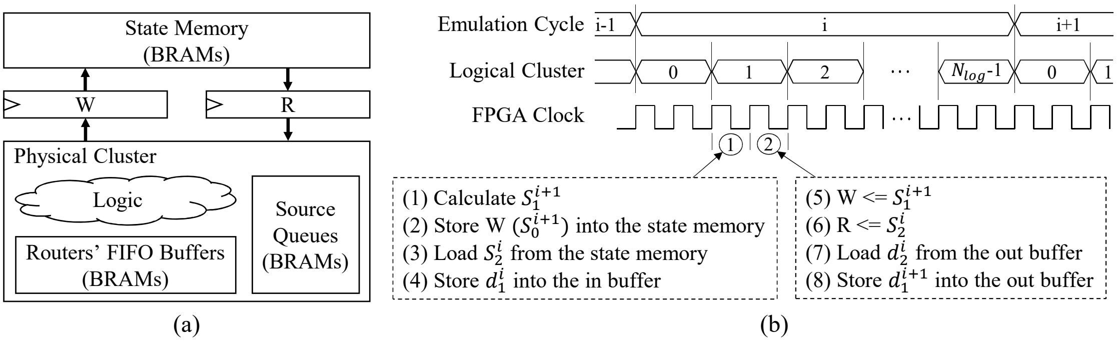

State Memory (BRAMs)
W
R
Physical Cluster
Logic
Routers' FIFO Buffers (BRAMs)
Source Queues (BRAMs)
(a)
Emulation Cycle
i-1
i
i+1
Logical Cluster
0
1
2
···
N_{log}-1
0
1
FPGA Clock
1
2
(1) Calculate S_1^{i+1}
(2) Store W (S_0^{i+1}) into the state memory
(3) Load S_2^i from the state memory
(4) Store d_1^i into the in buffer
(5) W <= S_1^{i+1}
(6) R <= S_2^i
(7) Load d_2^i from the out buffer
(8) Store d_1^{i+1} into the out buffer
(b)

Figure 3.9: (a) Datapath between the state memory and the physical cluster in time-multiplexed emulation of 2D meshes. (b) Timing diagram: S_j^i and d_j^i are the state and the outgoing data respectively of logical cluster j after emulation cycle $i-1$; both S_j^i and d_j^i are used at emulation cycle i.

$$\Rightarrow f(r, \texttt{pid1}) \neq f(r, \texttt{pid2})$$

We have proved that, in every case, there is no duplication of logical port IDs of down ports in every router. Because the logical port IDs of the up ports are calculated based on the logical port IDs of the down ports as shown in formula (3.6) in Section 3.4.3.2, there is also no duplication of logical port IDs of up ports in every router. Therefore, the procedure for calculating the logical port IDs is correct. □

3.5 Detailed Timing

This section first focuses on 2D meshes and then discusses the differences between 2D meshes and k-ary n-trees.

3.5.1 2D Mesh

As shown in Figure 3.9(a), the physical cluster in Fig. 3.1(b) can be abstracted to three parts: *logic*, *routers' FIFO buffers*, and *source queues*. The state data of the physical cluster are divided into two groups: memory (routers' FIFO buffers and traffic generators' source queues) and register (e.g., credit counters, allocators' states). To emulate N_{log} logical clusters, a memory of x-entry is enlarged to ($x \times N_{log}$)-entry, each series of x-entry for a logical cluster. Since only one logical cluster is emulated at each FPGA cycle, each enlarged memory also has one read port and one write port as the original one and thus can be implemented using BRAMs.

Each logical cluster has a set of registers stored in one entry of the state memory. The physical cluster emulates a logical cluster by feeding its set of registers loaded from the state memory to the logic part. For simplicity, below the set of registers of a logical cluster is considered as its state.

The TDM technique helps to effectively utilize BRAMs to implement FIFO buffers in routers and source queues in traffic generators because each BRAM can be shared between many nodes. BRAMs are also used to implement the state memory, the out buffer, and the in buffer. In fact, BRAMs are used much more extensively than other FPGA resources. For instance, as will be seen in the evaluation section of Chapter 4, 91.0% of BRAMs are required when emulating a 128×128 mesh using a four-node physical cluster, whereas only 1.9% of slice registers and 5.2% of slice LUTs are occupied. BRAMs in the Virtex-7 and other modern FPGA architectures are typically arranged in parallel columns due to both design and manufacturing constraints [119]. Therefore, if BRAMs are used much more extensively than registers and LUTs, then the design will be spread out according to the distribution of BRAMs, that is, along the parallel columns. This problem makes the placement and routing tasks much more difficult. Critical paths in the design are the paths between BRAMs. The timing of the design is dominated by the net delay, not by the logic delay.

In our design, because of the above problem, a path between a BRAM inside the state memory and a BRAM inside the physical cluster may be a critical path. Therefore, two registers R and W are inserted between the state memory and the physical cluster as shown in Figure 3.9(a).

Figure 3.9(b) shows the timing diagram. N_{log} here is the number of logical clusters. S_j^i and d_j^i are the state and the outgoing data respectively of logical cluster j after emulation cycle $i-1$. Both S_j^i and d_j^i are used at emulation cycle i. Two FPGA cycles are used to process each logical cluster. At the first FPGA cycle, (1) the state of the current logical cluster is updated to a new state. After that, at the second FPGA cycle, (5) the new state is stored into register W. (2) The value of register W will be stored into the state memory at the first FPGA cycle of emulating the next logical cluster in the emulation sequence, and will be used in the next emulation cycle. The state of the next logical cluster is (3) loaded from the state memory at the first FPGA cycle and (6) stored into register R at the second FPGA cycle. Also at the second FPGA cycle, (8) the new data of the current logical cluster become available and thus are stored into the out buffer. On the other hand, the old data are (7) loaded from the out buffer at the second FPGA cycle of emulating the previous logical cluster in the emulation sequence and (4) stored into the in buffer at the first FPGA cycle of emulating the current logical cluster. The next logical clusters in the emulation sequence will retrieve the current logical cluster's data from the in buffer. After the emulation of logical cluster $N_{log}-1$ is completed, the emulation cycle is incremented.

Although using two FPGA cycles for processing each logical cluster may decrease the emulation speed, it results in a simpler design because of the following two reasons. (1) If there is data dependency between the last and the first logical clusters in the emulation sequence (logical clusters $N_{log}-1$ and 0 in Figure 3.9(b)), then using one FPGA cycle for emulating each logical cluster will require complicated control logic to deal with the case where $d_{N_{log}-1}^{i+1}$ is still not available in both the out buffer and the in buffer when emulating logical cluster 0 in emulation cycle $i + 1$. (2) In emulation cycle i, the new data d_j^{i+1} of logical cluster j become available at the same time that the calculation of the new state S_j^{i+1} is completed. If a part of d_j^{i+1} is not registered, using one FPGA cycle for processing each logical cluster will require additional combinational logic to maintain this part because the state in the physical cluster has already changed to S_{j+1}^i (the state of logical cluster $j + 1$) when S_j^{i+1} is available. Another advantage of using two FPGA cycles for processing each logical cluster is that it becomes easy to improve the operating frequencies of BRAMs by adding an extra stage of registers to their outputs. As discussed above, since BRAMs are critical resources in our design, a higher overall operating frequency can be achieved.

Since the emulation of each logical cluster needs two FPGA cycles, the total number of FPGA cycles required to emulate one cycle of the NoC is $2 \times N_{log}$. However, as discussed in Chapter 4 and Chapter 6, since the network may be stalled, the actual number of FPGA cycles may be greater than $2 \times N_{log}$.

3.5.2 k-Ary n-Tree

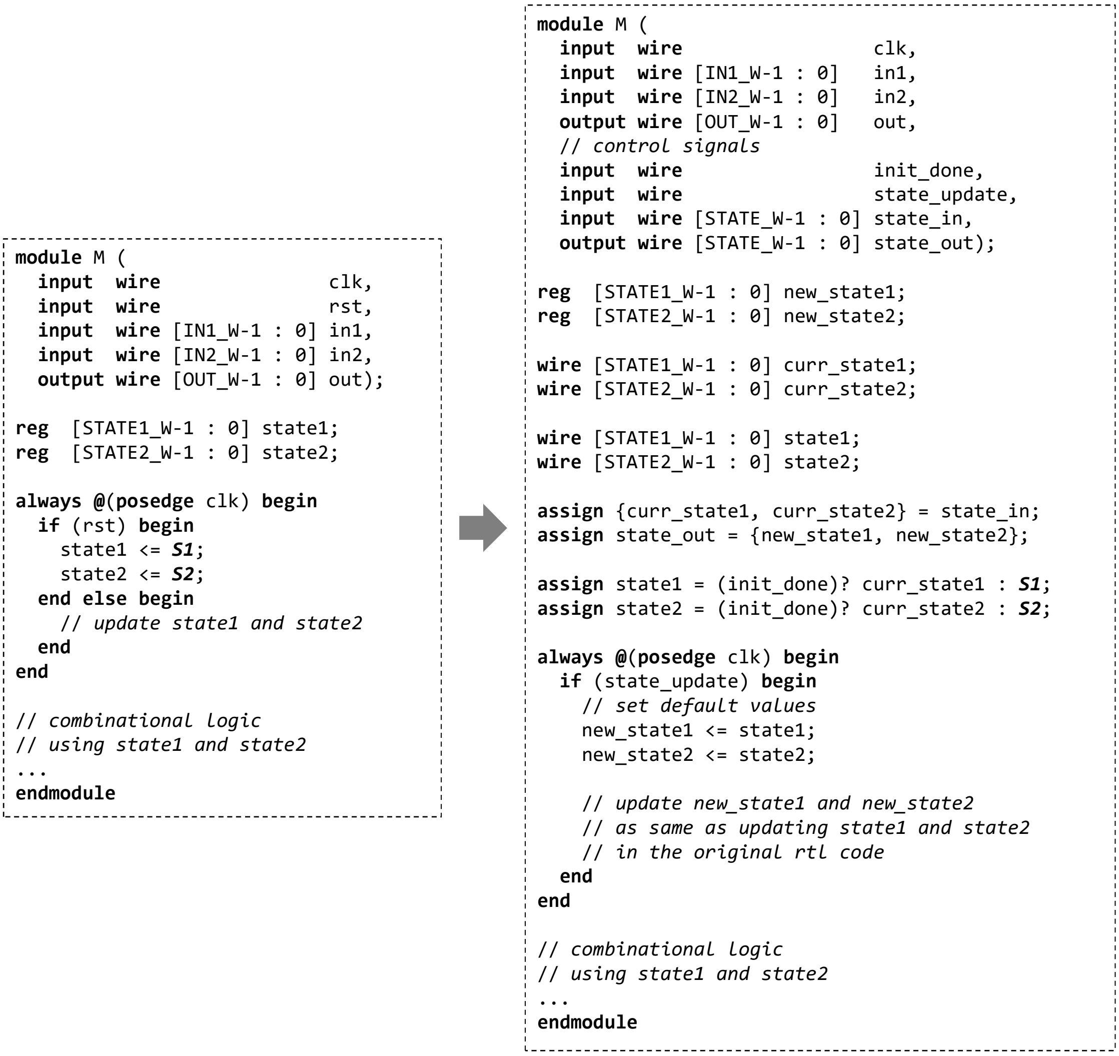
The detailed timing of emulating k-ary n-trees is basically the same as that described for 2D meshes in Section 3.5.1. The differences arise from the fact that, in a k-ary n-tree, the number of routers ($N_R = n \times k^{n-1}$) may be different from the number of cores ($N_C = k^n$).

In Figure 3.2(b), the physical router can be abstracted to *logic* and *FIFO buffers*, while the physical core can be abstracted to *logic* and *source queues*. To emulate N_R logical routers, a FIFO buffer of x-entry is enlarged to ($x \times N_R$)-entry, each series of x-entry for a logical router. Similarly, to emulate N_C logical cores, a source queue of x-entry is enlarged to ($x \times N_C$)-entry.

Two FPGA cycles are also used for processing each logical router/core. Each logical router is processed in parallel with a logical core. The read/write timing of the state memory and how data are copied from the out buffer to the in buffer are the same as those described in Section 3.5.1. If N_R is greater than N_C, the emulation cycle is incremented after the emulation of the last logical router is completed. Otherwise, the emulation cycle is incremented after the emulation of the last logical core is completed. Except for the case where the network is stalled due to the methods described in Chapter 4 and Chapter 6, the total number of FPGA cycles required to emulate one cycle of the NoC is $2 \times \max\{N_R, N_C\}$.

3.6 Emulation Code Translation

This section shows the basic rules for translating from the original Register-Transfer Level (RTL) code to the time-multiplexed emulation RTL code. These rules are independent of the target



```
module M (
  input  wire                clk,
  input  wire                rst,
  input  wire [IN1_W-1 : 0] in1,
  input  wire [IN2_W-1 : 0] in2,
  output wire [OUT_W-1 : 0] out);

reg  [STATE1_W-1 : 0] state1;
reg  [STATE2_W-1 : 0] state2;

always @(posedge clk) begin
  if (rst) begin
    state1 <= S1;
    state2 <= S2;
  end else begin
    // update state1 and state2
  end
end

// combinational logic
// using state1 and state2
...
endmodule
```

(a)

```
module M (
  input  wire                 clk,
  input  wire [IN1_W-1 : 0]   in1,
  input  wire [IN2_W-1 : 0]   in2,
  output wire [OUT_W-1 : 0]   out,
  // control signals
  input  wire                 init_done,
  input  wire                 state_update,
  input  wire [STATE_W-1 : 0] state_in,
  output wire [STATE_W-1 : 0] state_out);

reg  [STATE1_W-1 : 0] new_state1;
reg  [STATE2_W-1 : 0] new_state2;

wire [STATE1_W-1 : 0] curr_state1;
wire [STATE2_W-1 : 0] curr_state2;

wire [STATE1_W-1 : 0] state1;
wire [STATE2_W-1 : 0] state2;

assign {curr_state1, curr_state2} = state_in;
assign state_out = {new_state1, new_state2};

assign state1 = (init_done)? curr_state1 : S1;
assign state2 = (init_done)? curr_state2 : S2;

always @(posedge clk) begin
  if (state_update) begin
    // set default values
    new_state1 <= state1;
    new_state2 <= state2;

    // update new_state1 and new_state2
    // as same as updating state1 and state2
    // in the original rtl code
  end
end

// combinational logic
// using state1 and state2
...
endmodule
```

(b)

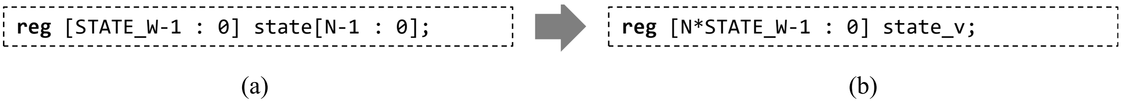
Figure 3.10: Translating from (a) the original RTL code to (b) the time-multiplexed emulation RTL code.

NoCs and might be automatically performed by software.

3.6.1 Register

Figure 3.10 shows an example where the original RTL code is translated to the time-multiplexed emulation RTL code. There are two states *state1* and *state2*, which are declared as registers in the original RTL code. Before the translation, we assume that the declarations of all register arrays have been already converted to the standard form as shown in Figure 3.11.

The initial values of *state1* and *state2* are ***S1*** and ***S2***, respectively. In the emulation RTL code, four control signals are added: *init_done*, *state_update*, *state_in*, and *state_out*.



```
reg [STATE_W-1 : 0] state[N-1 : 0];
```

(a)

```
reg [N*STATE_W-1 : 0] state_v;
```

(b)

Figure 3.11: Preprocessing before the code translation: a register array in (a) is converted to the standard form in (b).

- **init_done**. This control signal is used to avoid resetting all entries of the state memory at the beginning of each emulation. The need for resetting all entries of the state memory at the beginning of each emulation arises from the fact that, for correct operation, all state registers in the NoC emulator must be initialized to predetermined values before running any emulation. Because the state memory is very large when emulating large-scale NoCs with hundreds to thousands of nodes, it must be mapped to BRAMs. However, a memory cannot be mapped to BRAM if it has a reset input. Thus, the state memory needs to be implemented without a reset input. This can be achieved by the approach of initialization at power-up time (using *$readmemb* or similar functions in Verilog) if the emulator runs only one emulation after being powered on. However, this is usually not the case. For instance, in the case of emulation with synthetic workloads, an auto-reset mechanism is implemented for emulating a target NoC under different the traffic loads without re-synthesizing the FPGA design. By using the control signal *init_done* as below, it becomes unnecessary to reset the state memory and so, this memory can be mapped to BRAMs. The value of *init_done* is set to zero when the reset signal is activated and when the emulation cycle counter is equal to zero. Whenever *init_done* is zero, initial states are used instead of states loaded from the state memory. With the use of *init_done*, the reset signal *rst* becomes unnecessary, and thus is omitted from the emulation RTL code in Figure 3.10(b).

- **state_update**. To process a logical instance (a logical cluster in the case of emulating 2D meshes; or a logical router/core in the case of emulating k-ary n-trees) in an emulation cycle, we need at least two FPGA cycles. More than two FPGA cycles are required when the network is stalled as discussed in Chapter 4 and Chapter 6. The value of *state_update* is one only at the first FPGA cycle so that each logical instance is processed only one time per emulation cycle.

- **state_in**. This is the current state loaded from register R (Figure 3.9(a)).

- **state_out**. This is the new state which will be stored into the state memory after being temporarily stored in register W (Figure 3.9(a)).

3.6.2 Memory

As discussed in Section 3.5, to emulate N logical instances, an original memory of x-entry needs to be enlarged to a memory of $x \times N$-entry in the emulation code. In the $x \times N$-entry memory, each consecutive x entries are for one logical instance. Specifically, entries from address 0 to $x - 1$ belong to logical instance 0, entries from address x to $2x - 1$ belong to logical instance 1, and so on. Besides, we also have to add some logic for calculating the read address and write address of the enlarged memory.

3.7 Summary

This chapter proposed time-multiplexed emulation methods to emulate a large-scale NoC using a limited number of logic blocks that can be fit into a single FPGA. The chapter discussed in detail architectures and methods for supporting both direct and indirect network topologies, focusing on 2D meshes and k-ary n-trees. The chapter also described the basic rules for translating from the original RTL code to the time-multiplexed emulation RTL code. The proposed architectures and methods will be used to build a NoC emulator evaluated in Chapter 4 and Chapter 6.

Chapter 4

FNoC: An Emulator for NoC Emulation under Synthetic Workloads

4.1 Introduction

This chapter focuses on NoC emulation with synthetic workloads that are created based on mathematical modeling of common traffic patterns in real applications. Synthetic workloads have a high degree of flexibility, are easy to create, and thus have been commonly used. Currently, due to the lack of trace data of large-scale NoC-based systems, using synthetic workloads is practically the only feasible approach for emulating large-scale NoCs with thousands of nodes. A set of carefully designed synthetic workloads can provide a relatively thorough coverage of the characteristics of the target NoCs. Besides traditional synthetic traffic patterns such as uniform random and permutation, some effective synthetic traffic generation methodologies such as Synfull [120] and that introduced by Yin *et al.* [121] have been proposed to create richer traffic patterns that provide very close results compared to full-system simulations. It has been shown that evaluation on synthetic workloads is indispensable in many cases. For instance, when designing a routing algorithm, the use of synthetic workloads is mandatory for assessing the algorithm on possible corner cases like those under extremely high loads.

To evaluate a NoC on a synthetic workload, open-loop measurements are required [4]. To perform an open-loop measurement, the conventional way is to make the traffic generation and injection process independent of the rest of network and generate traffic according to the specified pattern without caring about the condition of the network. Specifically, a large source queue is placed at the output of each packet source to deal with the case that the packet source injects a packet at a time when the network cannot accept traffic (Figure 2.5(a)). In every emulation cycle, the packet source has an opportunity to generate and inject a packet into the source queue according to the specified injection process. Generated packets are stored in the source queues

```
if $squeue.empty()$ then
    $gen \leftarrow$ false
    while $!gen \;\&\; psource.time \leq network.time$ do
        if $rand() <$ THRESHOLD then
            $packet \leftarrow generate\_packet()$
            $squeue.push(packet)$
            $gen \leftarrow$ true
        end if
        $psource.time$++
    end while
end if
$network.time$++
```

Figure 4.1: Pseudo-code for simulating each packet source and the corresponding source queue with Bernoulli process in BookSim. This code is executed every simulation cycle.

until they can be accepted by the network.

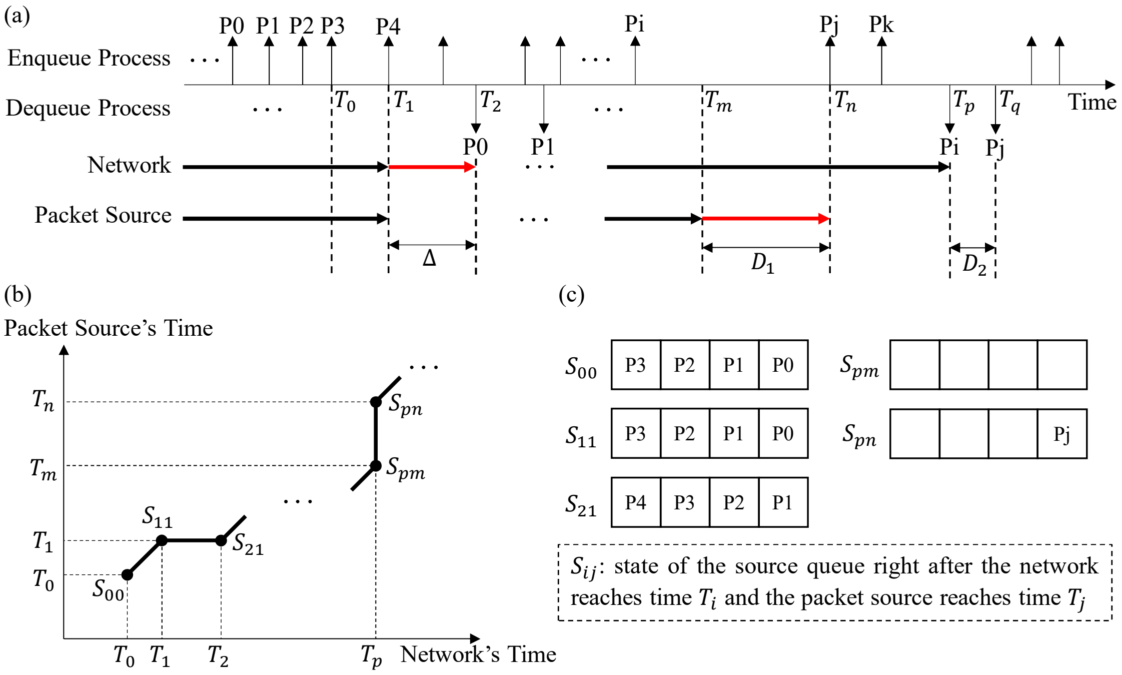
Ideally, the length of every source queue must be infinite, so that the packet sources can properly operate according to the specified injection process and none of the generated packets is dropped when the rate at which the network can accept packets is slower than the rate at which packets are generated. However, in practice, since the number of emulation cycles is finite, we only have to ensure that these source queues do not become full during the emulation. Thus, the length of each source queue can be finite but the upper bound cannot be determined before actually running the emulation. In general, this length must be very large to ensure proper emulation at high traffic loads. Additionally, a longer emulation time will require larger source queues if the traffic load is beyond the saturation point of the network.

Modern servers and PCs typically have a large amount of memory. Thus, most software-based NoC simulators can simply use the dynamic memory allocation approach to implement the large source queues. For instance, this approach is used by Noxim [101], a popular software-based NoC simulator. On the other hand, FPGA-based NoC emulators cannot directly use such approach. Kamali and Hessabi [67] and Papamichael [63] use a MicroBlaze soft processor for controlling the traffic injection process by software. Wolkotte *et al.* [60] use two ARM9 processors on a SoC board to implement the traffic generators by software. This is also the approach of Kamali in [68].

Compared with other software simulators, BookSim [5] uses a different approach which has been discussed by Dally and Towles in [4]. Figure 4.1 shows the pseudo-code for simulating each packet source and the corresponding source queue with Bernoulli process in BookSim. The time counter of the packet source ($psource.time$) is decoupled from the time counter of the network ($network.time$). The network advances its time counter cycle by cycle. On the other hand, the time of the packet source is advanced only when the corresponding source queue ($squeue$) is empty (the source queue becomes empty when the tail flit of the most recently created packet has been injected into the network). At a simulation cycle, if the source queue is empty, the packet source will run until a packet is generated or until its time counter is equal to the network's time counter. In this way, each source queue always contains at most one packet. Therefore, in BookSim, the memory footprint for the source queues is bounded.

However, it is extremely difficult to implement the above method on FPGAs due to the following reason. If the network and the packet source run at the same clock frequency, then it is impossible to execute the while loop from line 3 to line 10 in Figure 4.1 in one clock cycle since the packet source can advance only one step per clock cycle. A possible solution is to use a much faster clock for the packet source. However, this is extremely hard because of the limitation of increasing clock frequency in FPGAs and the unbounded number of iterations of the while loop.

This chapter proposes a method to overcome the problem of large source queues in the case of FPGA-based NoC emulation. Like the method used in BookSim, the time counter of each packet source is decoupled from the rest of the network. The key ideas are as follows. First, the network is made to operate interactively with the packet sources based on the status of the source queues and the relationship between the time counters. The packet sources are allowed to run behind the network to avoid dropping packets. Moreover, since it is impossible to run a while loop in one clock cycle on FPGAs, the network is stalled when there exists an empty source queue and the time of the packet source corresponding to this empty source queue is behind the network's time. By this way, we can make sure that the emulation is correct because, at any time, at least one of the following conditions holds: (1) all packet sources are not slower than the network and (2) none of the source queues is empty. Second, in the proposed method, the packet generation is run in parallel with injecting generated packets into the network. In BookSim, once a packet is generated, the packet source will wait until all flits of the generated packet are injected into the network. Different from BookSim, the packet sources in the proposed emulator do not generate flits directly, but instead, generate packet descriptors. A source queue may hold more than one packet descriptor at a time. When a source queue becomes full, the corresponding packet source may not have to be stopped immediately. The packet source is stopped only when it wants to generate a packet descriptor but the source queue is full. We have a module called flit generator (Figure 2.5(a)) placed after each source queue. The flit generator generates flits based on packet descriptors read from the source queue. This approach has two major advantages: (1) we do not have to store the whole packets in the source queues; (2) the impact of stalling the network on emulation performance is significantly reduced.



S_{ij}: state of the source queue right after the network reaches time T_i and the packet source reaches time T_j

Figure 4.2: Timeline of the network and a packet source. The enqueue process describes how the packet source generates and injects packet descriptors into the source queue while the dequeue process describes how packet descriptors are ejected from the source queue by the network. The state of the source queue is determined by both the network's time and the packet source's time.

4.2 Efficient Method for Modeling of Synthetic Workloads

Before going into details of the method, let us consider the trivial case. If the source queues are large enough to never become full, the packet sources will not have to stop working anytime. This means that the time counters of the packet sources are always equal to that of the network, and therefore, we do not have to stall the network anytime. Clearly, the emulation is correct. This case happens when the offered traffic load is lower than a threshold which depends on the source queue length, the workload, and the emulated NoC. Below we discuss the other cases where the packet sources are stopped to prevent packet descriptors from being dropped.

Figure 4.2 shows the timeline of the network and a packet source. Here, only one packet source is considered. The similar discussion can be applied to the other packet sources with the note that each packet source has a different time counter. The enqueue process describes how the packet source generates and injects packet descriptors into the source queue while the dequeue process describes how packet descriptors are ejected from the source queue by the network. For a given workload and a given NoC design, both the enqueue and dequeue process are deterministic since we are using pseudo-random number generators.

In Figure 4.2, the network's time indicates the emulation cycle that the network has reached. Similarly, the packet source's time indicates the emulation cycle that the packet source has reached. The source queue of four entries becomes full right after both the network and the packet source reach time T_0 (state S_{00} in Figure 4.2(b), (c)). However, the packet source is not stopped immediately. It continues to advance its time counter until time T_1 when it must generate and inject packet descriptor P4 into the source queue according to the enqueue process. Now, since the source queue is full (state S_{11} in Figure 4.2(b), (c)), the packet source has to temporarily stop all of its operations and wait for the network to reach time T_2 when one space in the source queue becomes available since packet descriptor P0 is ejected. The packet source can now generate and inject packet descriptor P4 into the source queue. It then continues execution from time T_1 and thus is behind the network which continues execution from time T_2. As the emulation time goes on, there are two possible cases described below.

Case 1. If each source queue always contains at least one packet descriptor, then the emulation is correct. This case happens when the traffic load is higher than a threshold. For a given workload, this threshold depends on the source queue length and the emulated NoC.

Case 2. If the traffic load is not high as the previous case but can sometimes make the source queues full, then the emulation may be not correct if we do not stall the network. This can be seen in the example in Figure 4.2. Let Δ be the distance between the network's time and the packet source's time. When the network is at time T_2 and the packet source is at time T_1, we have $\Delta = T_2 - T_1$. If only the packet source is stalled, then Δ will become larger and larger as time goes on. When Δ is large enough, the following problem can occur. Suppose that the network reaches time T_p when the packet source reaches time T_m. According to the enqueue process, the last packet descriptor generated and injected into the source queue by the packet source before time T_m is Pi. When the network's time is T_p and the packet source's time is T_m, Pi is ejected from the source queue according to the dequeue process. The source queue thus becomes empty (state S_{pm} in Figure 4.2(b), (c)). According to the enqueue process, the packet source will not generate any packet descriptors until it reaches time T_n when Pj is generated. On the other hand, according to the dequeue process, Pj will be ejected from the source queue when the network reaches time T_q ($T_q > T_n$). Let $D_1 = T_n - T_m$ and $D_2 = T_q - T_p$. If D_1 is greater than D_2, then the network will reach time T_q before packet descriptor Pj is generated and injected into the source queue by the packet source. In other words, the packet source has become too slow to be able to generate packet descriptors according to the injection process, and therefore the generated workload is not the one originally specified. To overcome this problem, the network is forced to stop all of its operations when both of two following conditions hold: (1) the source queue is empty, and (2) the current time counter of the packet source is smaller than the current time counter of the network. When the network is stalled, the packet source continues its operations. As shown in Figure 4.2, the network is stalled at time T_p until the packet source reaches time T_n when packet descriptor Pj is generated and injected into the source queue (state S_{pn} in Figure 4.2(b), (c)). By this way, we can ensure that either the packet source is not slower

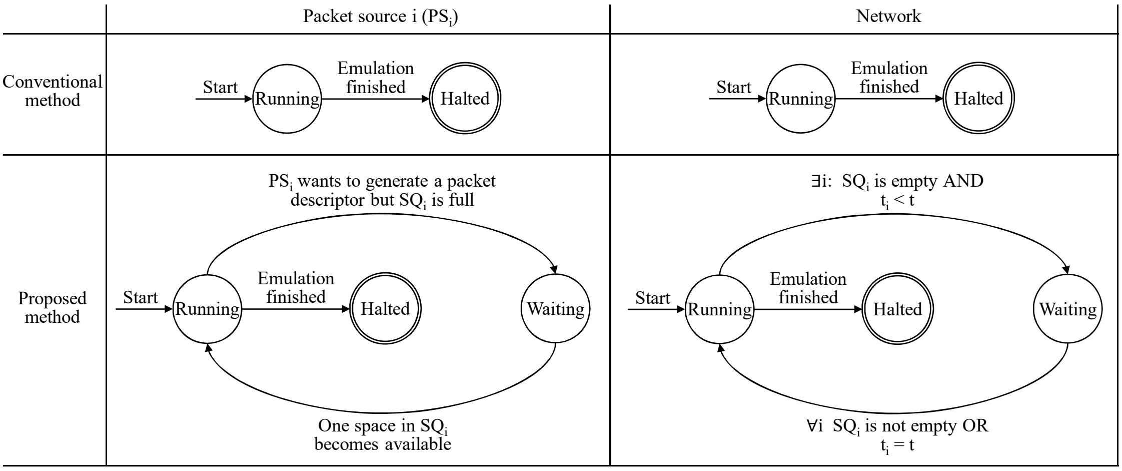

Packet source i (PS$_i$)
Network
Conventional method
Proposed method
Start
Running
Emulation finished
Halted
Waiting
PS$_i$ wants to generate a packet descriptor but SQ$_i$ is full
One space in SQ$_i$ becomes available
∃i: SQ$_i$ is empty AND t$_i$ < t
∀i SQ$_i$ is not empty OR t$_i$ = t

Figure 4.3: The state transition diagrams of each packet source and the network in the conventional method and the proposed method for modeling synthetic workloads. Here, packet source i and its corresponding source queue are denoted by PS_i and SQ_i, respectively. The current time counters of packet source i and the network are denoted by t_i and t, respectively.

than the network or the source queue contains at least one packet. Thus, the emulation is correct.

In the first case, since the source queues never become empty, the network is not stalled. Thus, the emulation can take place without any penalty cycles. On the other hand, the network may be stalled many times in the second case. For a given NoC design and a given workload, the number of penalty cycles depends on the length of the source queues. Larger source queues will reduce the number of penalty cycles, and hence reduce the stall time, but also require more memory. This issue will be discussed in more detail in Section 4.3.3.

Figure 4.3 summarizes the proposed method in comparison with the conventional one in the form of state transition diagrams. In the conventional method, the packet sources and the network keep running from the start to the end of the emulation without any interruptions. On the other hand, in the proposed method, they may transition between the running state and the waiting state during the emulation. The state transitions occur in response to certain events. A packet source transitions from the running state to the waiting state when it wants to generate a packet descriptor but the corresponding source queue is full. The packet source stays at the waiting state until one space in the source queue becomes available when it returns to the running state. The network transitions from the running state to the waiting state when there exists an empty source queue and the time of the packet source connected to this empty source queue is behind the time of the network. It transitions back to the running state when the condition for transitioning from the running state to the waiting state does not hold anymore.

Table 4.1: Common parameters of the target NoCs and emulation parameters

Network parameters		Emulation parameters	
Router architecture	Input-queued VC router	Packet length	8-flit
VC/Switch allocator	iSLIP [122]	Injection process	Bernoulli process
Arbiter type	Round-robin	Traffic pattern	Uniform random & hotspot
VC size	4-flit	# of warmup / measurement cycles	200,000 / 200,000
Flow control	Credit-based	Source queue length	8-entry

Table 4.2: Individual parameters of each of the target NoCs

NoC	#cores	#routers	Router radix	Flit size	NoC	#cores	#routers	Router radix	Flit size
128×128-2vc-5stage-xy	16,384	16,384	5	18bit	64×64-1vc-5stage-xy	4,096	4,096	5	18bit
128×128-2vc-5stage-oe	16,384	16,384	5	19bit	64×64-1vc-4stage-xy	4,096	4,096	5	21bit
128×128-2vc-4stage-xy	16,384	16,384	5	21bit	4ary6tree-2vc-5stage-nca	4,096	6,144	8	18bit
128×128-1vc-5stage-xy	16,384	16,384	5	18bit	4ary6tree-1vc-5stage-nca	4,096	6,144	8	18bit
128×128-1vc-4stage-xy	16,384	16,384	5	21bit	4ary5tree-2vc-5stage-nca	1,024	1,280	8	18bit
64×64-2vc-5stage-xy	4,096	4,096	5	18bit	4ary5tree-1vc-5stage-nca	1,024	1,280	8	18bit
64×64-2vc-5stage-oe	4,096	4,096	5	19bit	2ary10tree-2vc-5stage-nca	1,024	5,120	4	18bit
64×64-2vc-4stage-xy	4,096	4,096	5	21bit	2ary10tree-2vc-5stage-nca	1,024	5,120	4	18bit

xy: dimension-order routing algorithm.
oe: minimal adaptive routing algorithm based on the odd-even turn model [72].
nca: nearest common ancestor routing algorithm.

4.3 Evaluation

A NoC emulator, called FNoC, is developed on a Xilinx VC707 board using the methods proposed in this chapter and Chapter 3. Vivado 2015.4 is used for synthesizing, implementing, and generating FPGA bitstream files. The synthesis and implementation strategies are set as *Flow_PerfOptimized_High* and *Performance_ExplorePostRoutePhysOpt*, respectively. The emulation results are transferred to a host PC via an RS232C interface at a data rate of 0.5 Mbps.

Tables 4.1 and 4.2 show the parameters of the 16 target NoCs and the emulation parameters. There are two pipelined architectures. In the five-stage architecture, the five pipeline stages consist of routing computation, VC allocation, switch allocation, switch traversal, and link traversal as described in Chapter 2. In the four-stage architecture, the look-ahead routing technique is used to perform two stages routing computation and VC allocation in parallel, thereby reducing the number of pipeline stages from five to four. In the look-ahead routing technique, the router at hop i of a route performs the routing computation for the next router, which is at hop $i+1$ of the route, and passes the result along with the head flit. Therefore, the flit size used in the four-stage router is larger than that used in the five-stage pipelined router. Besides the XY dimension-order routing algorithm, a minimal adaptive routing algorithm based on the odd-even turn model [72] is also implemented. To implement this algorithm, one control bit is added to the flit structure.

Table 4.3: Overhead and speed of emulating each of the target NoCs under uniform and hotspot traffics

	LUTs		Regs		Slices		BRAMs		Vivado's	Avg. speed
	#	%	#	%	#	%	#	%	runtime	(cycles/s)
128×128-2vc-5stage-xy-uniform-8×4phy	122,294	40.3	87,312	14.4	39,090	51.5	923	89.6	80min	97.0K
128×128-2vc-5stage-xy-uniform-4×4phy	61,584	20.3	43,984	7.2	23,597	31.1	965	93.7	50min	48.7K
128×128-2vc-5stage-xy-uniform-2×2phy	15,846	5.2	11,334	1.9	8,036	10.6	937	91.0	23min	12.2K
128×128-2vc-5stage-xy-hotspot-8×4phy	151,976	50.1	87,344	14.4	46,105	60.7	923	89.6	88min	77.8K
128×128-2vc-5stage-oe-uniform-8×4phy	126,465	41.7	89,269	14.7	39,843	52.5	939	91.2	85min	96.4K
128×128-2vc-5stage-oe-hotspot-8×4phy	156,128	51.4	89,269	14.7	46,822	61.7	939	91.2	90min	77.2K
128×128-2vc-4stage-xy-uniform-8×4phy	126,808	41.8	93,295	15.4	39,239	51.7	956	92.8	82min	97.1K
128×128-1vc-5stage-xy-uniform-8×4phy	75,053	24.7	61,286	10.1	27,699	36.5	635	61.7	55min	96.8K
128×128-1vc-4stage-xy-uniform-8×4phy	79,028	26.0	65,094	10.7	27,069	35.7	651	63.2	53min	96.9K
64×64-2vc-5stage-xy-uniform-8×4phy	120,573	39.7	87,185	14.4	36,824	48.5	635	61.7	77min	387.8K
64×64-2vc-5stage-xy-uniform-4×4phy	60,245	19.8	43,857	7.2	19,366	25.5	368	35.7	43min	194.6K
64×64-2vc-5stage-xy-uniform-2×2phy	15,685	5.2	11,217	1.8	5,575	7.3	246	23.9	13min	48.8K
64×64-2vc-5stage-xy-hotspot-8×4phy	152,386	50.2	87,228	14.4	44,344	58.4	635	61.7	83min	291.1K
64×64-2vc-5stage-oe-uniform-8×4phy	125,406	41.3	89,167	14.7	38,979	51.4	651	63.2	77min	384.9K
64×64-2vc-5stage-oe-hotspot-8×4phy	157,016	51.7	89,197	14.7	45,707	60.2	651	63.2	83min	288.7K
64×64-2vc-4stage-xy-uniform-8×4phy	126,020	41.5	93,050	15.3	40,991	54.0	668	64.9	103min	389.0K
64×64-1vc-5stage-xy-uniform-8×4phy	74,374	24.5	61,073	10.1	25,718	33.9	507	49.2	47min	386.6K
64×64-1vc-4stage-xy-uniform-8×4phy	77,942	25.7	64,920	10.7	27,079	35.7	523	50.8	52min	385.0K
4ary6tree-2vc-5stage-nca-uniform	8,890	2.9	5,818	1.0	4,557	6.0	871	84.6	34min	8.1K
4ary6tree-1vc-5stage-nca-uniform	5,451	1.8	3,846	0.6	3,150	4.2	591	57.4	12min	8.1K
4ary5tree-2vc-5stage-nca-uniform	8,707	2.9	5,247	0.9	3,642	4.8	273	26.5	11min	39.0K
4ary5tree-1vc-5stage-nca-uniform	5,345	1.8	3,445	0.6	2,360	3.1	206	20.0	8min	39.0K
2ary10tree-2vc-5stage-nca-uniform	3,651	1.2	2,758	0.5	1,971	2.6	369	35.8	10min	9.8K
2ary10tree-1vc-5stage-nca-uniform	2,865	0.9	2,121	0.3	1,416	1.9	255	24.8	8min	9.8K

When there are two available output ports, the implemented selection strategy selects the port with more free VCs.

All the RTL code is developed from scratch. The emulation RTL code is created based on the rules described in Section 3.6.

Table 4.3 summarizes the overhead and speed of emulating each of the 16 target NoCs. Four 2D mesh NoCs 128×128-2vc-5stage-xy/oe and 64×64-2vc-5stage-xy/oe are evaluated on two traffic patterns: uniform random and hotspot. In the uniform random traffic, the destination of each packet is randomly selected with equal probability among all nodes in the network. In the hotspot traffic pattern, hotspot nodes are located at the left-up corner of the network and receive four times more traffic than the other nodes. The numbers of hotspot nodes used in evaluating the 128×128 and 64×64 NoCs are 256 (a 16×16 cluster) and 64 (a 8×8 cluster), respectively. When emulating the two NoCs 128×128-2vc-5stage-xy and 64×64-2vc-5stage-xy under the uniform random traffic, three different physical cluster sizes are used: 8×4, 4×4, and 2×2. The total times of synthesizing, implementing and generating the FPGA bitstream files on a Core i7 4770 PC are included for reference. The emulation speeds are the average speeds at all traffic loads used in latency measurements.

4.3.1 Resource Requirements

We first look at the hardware overhead when emulating the 2D meshes. There are four major points here. (1) FNoC uses BRAMs much more extensively than slice registers and slice LUTs. (2) The numbers of occupied slice registers and slice LUTs are roughly proportional to the physical cluster size. (3) When the emulated network is not large enough, many occupied BRAMs are underutilized. To understand this, we have to understand the characteristics of BRAMs. Each BRAM in Xilinx 7 series FPGAs can be configured to only one of the following depths: $2^9, 2^{10}, \cdots, 2^{15}$. When employing the TDM technique, as described in Chapter 3, the depth of the state memory as well as the out buffer and the in buffer is N_{log} where N_{log} is the number of logical clusters. Also, a memory of x-entry is enlarged to ($x \times N_{log}$)-entry. If N_{log} or $x \times N_{log}$ is not a multiple of $2^9, 2^{10}, \cdots, 2^{15}$, many of the occupied BRAMs will be underutilized. For example, when emulating a 64×64 NoC with physical cluster sizes 8×4, 4×4, and 2×2, the numbers of logical clusters are $2^7, 2^8$, and 2^{10}, respectively. Thus, many of the occupied BRAMs are underutilized when the physical cluster size is 8×4 or 4×4. On the other hand, when emulating a 128×128 NoC with physical cluster sizes 8×4, 4×4, and 2×2, the numbers of logical clusters are $2^9, 2^{10}$, and 2^{12} respectively. Thus, most occupied BRAMs are fully utilized. Therefore, there is only minor difference in the number of required BRAMs when emulating a 128×128 NoC with different physical cluster sizes, while emulating a 64×64 NoC with a smaller physical cluster requires less BRAMs. (4) With the same architecture, physical cluster size, and traffic pattern, emulating a larger network only requires more BRAMs. For example, when the architecture, physical cluster size, and traffic pattern are 2vc-5stage-xy, 2×2, and uniform, respectively, emulating the 64×64 NoC requires 23.9% of BRAMs while emulating the 128×128 NoC requires 91.0% of BRAMs. However, the numbers of required slice LUTs, as well as slice registers, are almost the same. The size of the largest NoC that can be emulated by FNoC depends on only the total amount of BRAMs on the FPGA. By using an FPGA with more BRAMs, larger NoCs can be supported. This is an important point because the amount of BRAMs embedded on an FPGA is doubling every two years [123].

When emulating the k-ary n-trees, BRAMs are also the most used resources. Note that the router radix of a k-ary n-tree is different from that of a 2D mesh. For example, the router radix of a 4-ary 6-tree is eight while that of a 2D mesh is five. Thus, implementing a physical router of the 4-ary 6-tree requires more resources.

The operating frequency of FNoC is highly dependent on three factors: (1) the traffic workload that needs to be modeled, (2) the critical path of the router pipeline, and (3) the amount of required BRAMs. As discussed in Section 3.5 in Chapter 3, since BRAMs are used much more extensively than other FPGA resources, the design is spread out according to the distribution of BRAMs, that is, along parallel columns in which BRAMs are placed. Although this problem is mitigated by inserting registers into the paths between BRAMs as described in Section 3.5, the paths to BRAMs might be still very long if most BRAMs of the FPGA are occupied. In this evaluation, the timing constraint is set to 100 MHz in the case the traffic pattern is uniform. 100 MHz is the maximum operating frequency when emulating the 4ary6tree-2vc-5stage-nca NoC. This is because the router pipeline in this NoC has the longest critical path (due to the high radix), and 84.6% of BRAMs are occupied while only 2.9% of slice LUTs and 1.0% of slice registers are required. Higher frequencies might be achieved when emulating the other NoCs. In the case the traffic pattern is hotspot, the frequency decreases to 80 MHz (128×128 NoCs) and 75 MHz (64×64 NoCs) due to the use of modulo operations with non-power-of-2 operands.

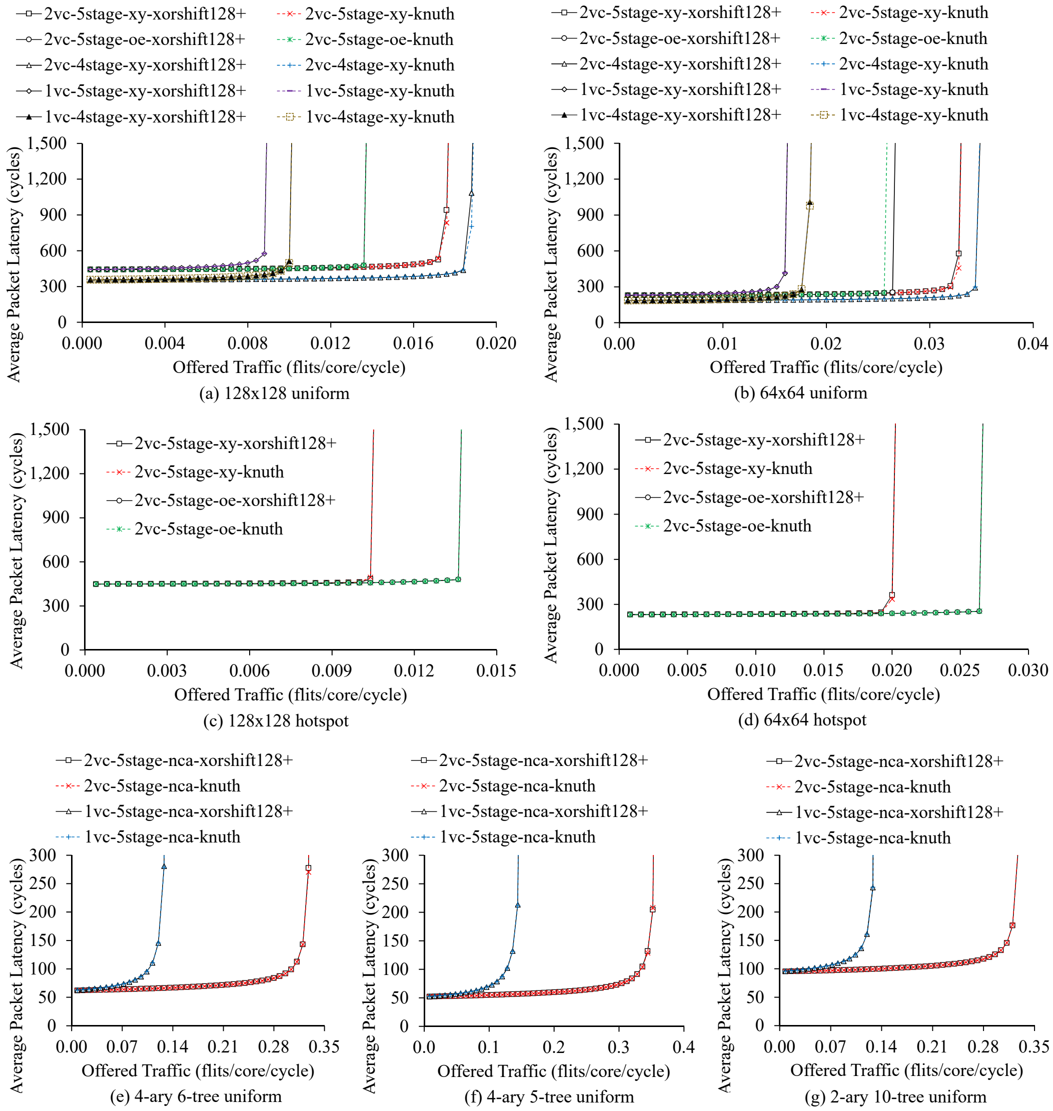
4.3.2 Emulation Accuracy

The emulation results of FNoC are compared to those of BookSim. For each traffic load, 200,000 warmup cycles, 200,000 measurement cycles, and a drain phase are emulated. We are limited to emulating these numbers of warmup and measurement cycles because BookSim is extremely slow. As mentioned in Section 2.5 in Chapter 2, the current parameters allow us to execute up to 2^{28} cycles. For every emulation, it is verified that all packets that are sent are received.

As mentioned earlier, three physical cluster sizes are used to emulate two 2D meshes. However, note that the physical cluster size does not have any impact on the emulation results that are always the same as those obtained when not using the TDM technique. A larger physical cluster only makes the emulation faster.

The evaluation results show that, when using the same random number generator, FNoC and BookSim report exactly the same results, including both latency and throughput results, in every case. This is evident in that the developed NoC models are totally identical to those of BookSim. Currently, FNoC uses the xorshift128+ pseudo-random number generator [124] for all FPGA implementations while the default pseudo-random number generator used in BookSim is Knuth's [125]. Figure 4.4 shows the latency graphs obtained when emulating the NoCs described in Table 4.2 under uniform and hotspot traffics. Figure 4.5 shows the distributions of packet latency obtained when emulating the 128×128-2vc-4stage-xy at two different traffic loads. For reference, the results of both random number generators are included. We can see that the results obtained when using the xorshift128+ pseudo-random number generator are almost the same as those obtained when using Knuth's pseudo-random number generator.

Figure 4.4 shows that some results such as those regarding the numbers of VCs and pipeline stages from small-scale NoCs can be also applied to large-scale NoCs. For the minimal adaptive routing algorithm based on the odd-even turn model, Figure 4.4 also shows the same trend



(a) 128x128 uniform

(b) 64x64 uniform

(c) 128x128 hotspot

(d) 64x64 hotspot

(e) 4-ary 6-tree uniform

(f) 4-ary 5-tree uniform

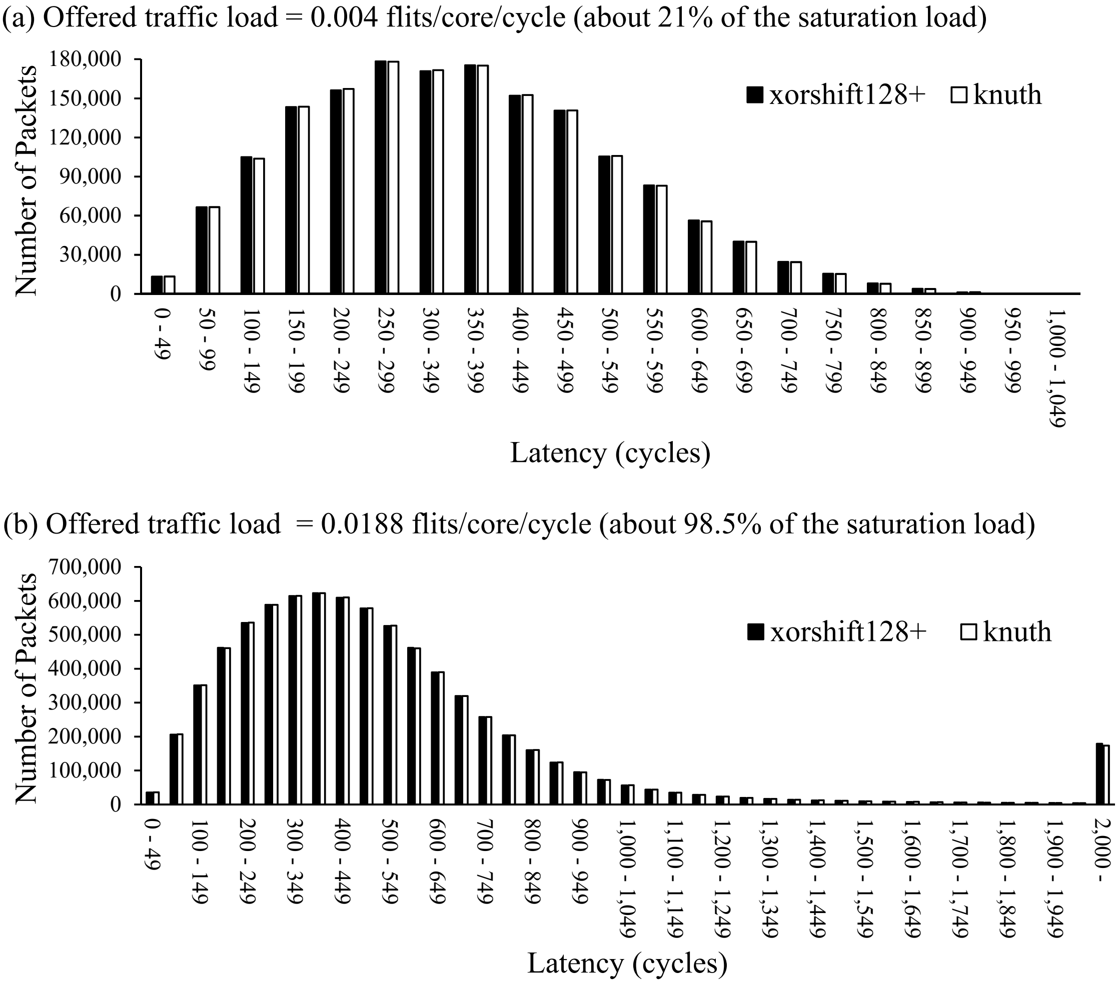
(g) 2-ary 10-tree uniform

Figure 4.4: Average packet latency: emulating 16 target NoCs with the parameters described in Tables 4.1 and 4.2.

reported in [72] and [126] for small-scale NoCs: compared to XY routing, the adaptive routing algorithm performs worse under uniform traffic but better under hotspot traffic.

4.3.3 Emulation Performance

This section first presents a formula for calculating the emulation speed S of FNoC. In the formula, N_{phy}, N_{node}, and α are defined as follows. In the case of emulating a 2D mesh, N_{phy} and N_{node} are the physical cluster size (the number of physical nodes) and the number of nodes of the NoC, respectively. In the case of emulating a k-ary n-tree, $N_{phy} = 1$ and N_{node} is $\max\{N_R, N_C\}$ where $N_R = n \times k^{n-1}$ and $N_C = k^n$ are the numbers of routers and cores re-



(a) Offered traffic load = 0.004 flits/core/cycle (about 21% of the saturation load)

(b) Offered traffic load = 0.0188 flits/core/cycle (about 98.5% of the saturation load)

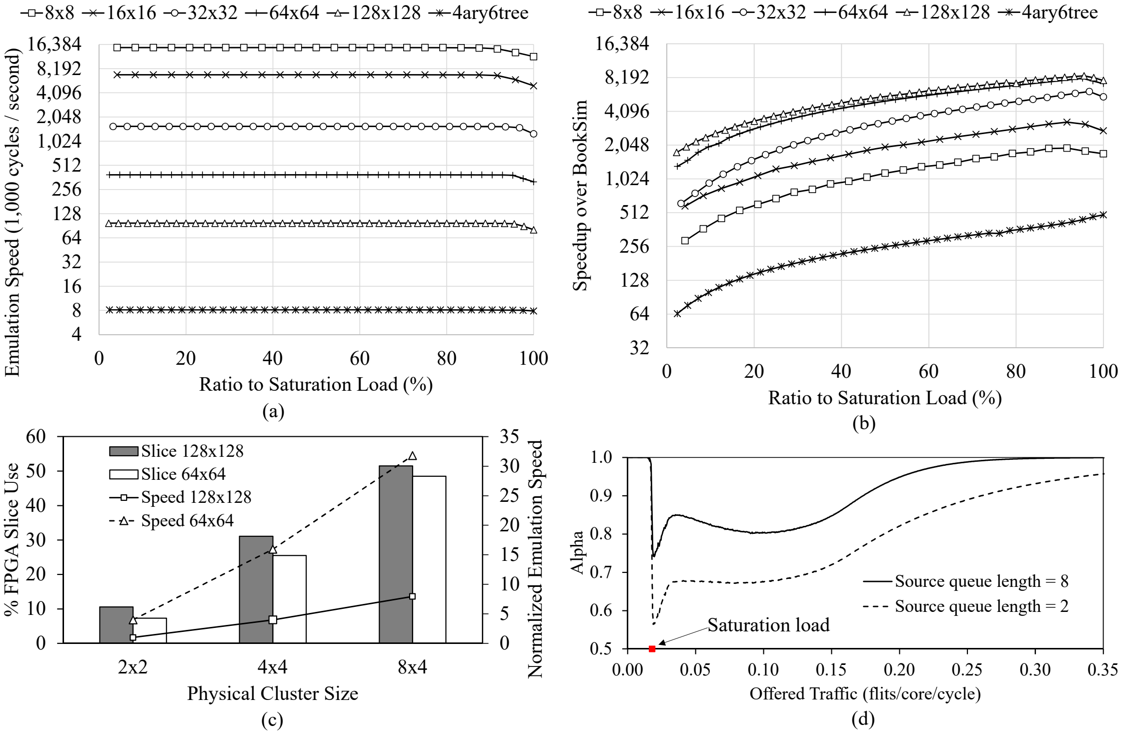
Figure 4.5: Distributions of packet latency: emulating the 128×128-2vc-4stage-xy (detailed parameters are described in Tables 4.1 and 4.2) under uniform traffic.

spectively of the NoC. Finally, α $(0 < \alpha \leq 1)$ is a coefficient reflecting the stalling effect caused by using the method described in Section 4.2. $\alpha = 1$ means that there is no stalling effect. If the network is stalled during the emulation, then α will be smaller than 1. Since two FPGA cycles are used for processing each logical instance (cluster/router/core), the emulation speed S (emulation cycles per second) of FNoC is

$$S = \alpha \times \frac{F}{2 \times \frac{N_{node}}{N_{phy}}}, \tag{4.1}$$

where N_{phy}, N_{node}, α are described above and F is the operating frequency (Hz) of the system. From this formula, we can calculate the time T (seconds) needed to execute C emulation cycles by the following formula.

$$T = \frac{C}{S}$$



(a) (b) (c) (d)

Figure 4.6: (a) FNoC's emulation speed for different network sizes; (b) FNoC's speedup over BookSim; (c) FNoC's normalized emulation speed versus % FPGA slice use for different physical cluster sizes; (d) The stalling effect coefficient α with different traffic loads and source queue lengths in the case of emulating the 128$\times$128-2vc-5stage-xy.

In the evaluation below, the traffic pattern used is uniform random.

Figure 4.6(a) shows the emulation speed of FNoC for different network sizes. The 128×128 and 4ary6tree NoCs here are the 128×128-2vc-5stage-xy and 4ary6tree-2vc-5stage-nca described in Tables 4.1 and 4.2. The other 2D meshes have the same parameters as the 128×128 NoC except the network size. The 8×8 NoC is emulated using a 4×4 physical cluster while the other 2D meshes are emulated using a 8×4 physical cluster. FNoC operates at 120 MHz when emulating the 8×8 NoC, 110 MHz when emulating the 16×16 NoC and 100 MHz when emulating the other NoCs. In each case, the stalling effect coefficient α is measured and the emulation speed is derived from formula (4.1). For the 2D meshes, FNoC's speed varies from 11,580K–15,000K (cycles/s) for the 8×8 NoC to 80.9K–97.7K (cycles/s) for the 128×128 NoC. For the 4-ary 6-tree NoC, the speed decreases to 7.9K–8.1K since only one physical router and one physical core are used to emulate the entire network. In every case, we can see that the emulation speed drops slightly at traffic loads near the saturation load. This is because of the stalling effect, which will be analyzed in detail below.

Figure 4.6(b) shows the speedup of FNoC over BookSim. For this comparison, BookSim is run on a Core i7 4770 PC. The evaluation results show that BookSim's speed decreases with increasing traffic load. This is because a higher traffic load requires more operations to be executed. For example, the lower the traffic load is, the more likely that the body of an if statement can be skipped. On the other hand, as shown in Figure 4.6(a), FNoC's speed is almost constant when the traffic load is not high. Therefore, as shown in Figure 4.6(b), the speedup is generally higher when the traffic load is higher. The stalling effect makes the speedup decrease at some traffic loads near the saturation load. However, as will be explained later, the stalling effect tends to zero and thus FNoC's speed increases back to the zero-load speed as the traffic load is further increased. Since BookSim's speed decreases with increasing the traffic load, the speedup of FNoC over BookSim at an extremely high traffic load is greater than at a low traffic load. When emulating the 128×128 NoC, the maximum and average (geomean) speedups over BookSim are 8,469× and 5,047× respectively. In terms of emulation time, the time needed to draw the latency graph of the NoC (shown in Figure 4.4(a)) is reduced from around 13.2 days to around 3.2 minutes. When emulating the 4-ary 6-tree NoC, a maximum of 489× and an average (geomean) of 232× speedups are achieved. The time needed to draw the latency graph of the NoC (shown in Figure 4.4(e)) is reduced from around 6.2 days to around 34.5 minutes.

Figure 4.6(c) shows the Pareto chart of FNoC's normalized speed versus % FPGA slice use in the case of emulating the 128×128 NoC and 64×64 NoC with different physical cluster sizes. The emulation speeds are normalized to the case of emulating the 128×128 NoC using a 2×2 physical cluster. We can see that FNoC's speed is almost proportional to the physical cluster size.

Finally, we analyze the stalling effect coefficient α (in formula (4.1)). For a given workload, α depends on the source queue length and the emulated NoC. Figure 4.6(d) shows how α varies with different traffic loads and source queue lengths (note that all results presented above are obtained with the source queue length = 8) in the case of emulating the 128×128 NoC. This result is matched with the analysis in Section 4.2. Specifically, at low traffic loads, $\alpha = 1$ because we do not have to stall the network. α starts to decrease when the offered traffic load is close to the saturation load. At extremely high traffic loads, the network does not stall because, as the traffic load increases, it is highly likely that every source queue always contains at least one packet descriptor after it becomes full for the first time. Thus, α converges to 1 as the offered traffic load tends to 1.

4.3.4 Comparison with other FPGA-Based NoC Emulators

This section first compares how FNoC and the other FPGA-based NoC emulators are justified in terms of emulation accuracy. Among all emulators mentioned in Section 2.4.2 in Chapter 2, only DART [65], AdapNoC [67], and DuCNoC [68] are verified against a well-known simulator, which is also BookSim. However, all of them report the comparison results for only one 3×3 NoC with XY routing and a simple router model under uniform traffic. Although the comparison results (latency graphs) show that these emulators track BookSim closely, the differences are not zero. For instance, DART reports a noticeable difference in average packet latency and distribution of packet latencies at high traffic loads mainly due to the difference in the router models. In contrast, FNoC reports emulation results of 16 target NoCs with both deterministic and adaptive routing algorithms and fully pipelined router models under both uniform and non-uniform traffic patterns. When using the same random number generator, the results reported by FNoC are totally identical to those reported by BookSim.

The section next compares the emulation speeds of FNoC, DART, AdapNoC, DuCNoC, and the NoC emulator proposed by Drewes *et al.* [66]. Because of some differences such as in the router architectures and the FPGAs used, the comparison here is not strictly quantitative. The thesis aims to qualitatively show that the emulation speed of FNoC is comparable to the state-of-the-art emulators.

DART's emulation speed varies from around 9,000K to around 22,500K emulation cycles per second for a 3×3 NoC and decreases to from around 5,500K to around 16,000K emulation cycles per second for an 8×8 NoC. By using the TDM approach, DART can emulate a 9×9 NoC. However, the authors of DART do not provide results for this NoC.

AdapNoC's emulation speed varies from around 175K to around 400K emulation cycles per second for a 3×3 NoC and decreases to from around 30K to around 200K emulation cycles per second for an 8×8 NoC. The authors of AdapNoC state that AdapNoC can emulate a 32×32 NoC using the TDM approach. However, they do not provide any results for NoCs larger than 8×8.

DuCNoC's speed for a 5×5 NoC varies from around 200K to around 375K emulation cycles per second. For larger NoCs, the authors of DuCNoC do not report the absolute emulation speeds but instead the speedups compared to BookSim. Their evaluation results show that DuCNoC's speedup over BookSim decreases with increasing the target NoC size; the maximum speedups when emulating a 512-node NoC and an 8,196-node NoC (the largest NoC that can be emulated by DuCNoC) are 265× and 3×, respectively.

The size of the largest NoC that can be emulated by Drewes *et al.*'s NoC emulator is 8×8. The speed when emulating this NoC under a uniform random traffic pattern is around 16.6K emulation cycles per second.

As presented in Section 4.3.3, when emulating an 8×8 NoC, the speed of FNoC is from 11,580K to 15,000K emulation cycles per second. The largest NoC size that currently can be emulated by FNoC is 128×128 (16,384-node). When emulating this NoC, FNoC's speed is from 80.9K to 97.7K emulation cycles per second. Different from DuCNoC, FNoC's speedup over BookSim increases with increasing the target NoC size; the speedup when emulating the 128×128 NoC is 5,047×.

4.4 Summary

This chapter presented a method for emulating a NoC under a synthetic workload without requiring a large amount of memory. This method and the time-multiplexed emulation methods proposed in Chapter 3 enable fast and cycle-accurate emulation of large-scale NoCs with sophisticated router architectures on a single FPGA without using off-chip memory. An extensive evaluation with 16 target NoCs was performed. Compared to BookSim, a widely used NoC simulator, the proposed NoC emulator was 5,047× faster when emulating a 128×128 NoC and 232× faster when emulating a 4-ary 6-tree NoC, while providing the same results.

The method presented in this chapter considers the most popular version of synthetic workloads in which the traffic generation process of a node is independent of those of all the other nodes. However, one may want to create a synthetic workload in which the traffic generation process of a node may affect another. To support such synthetic workloads, the proposed method needs to be modified.

Chapter 5

A Use Case of FNoC in Design and Modeling of a New Routing Algorithm

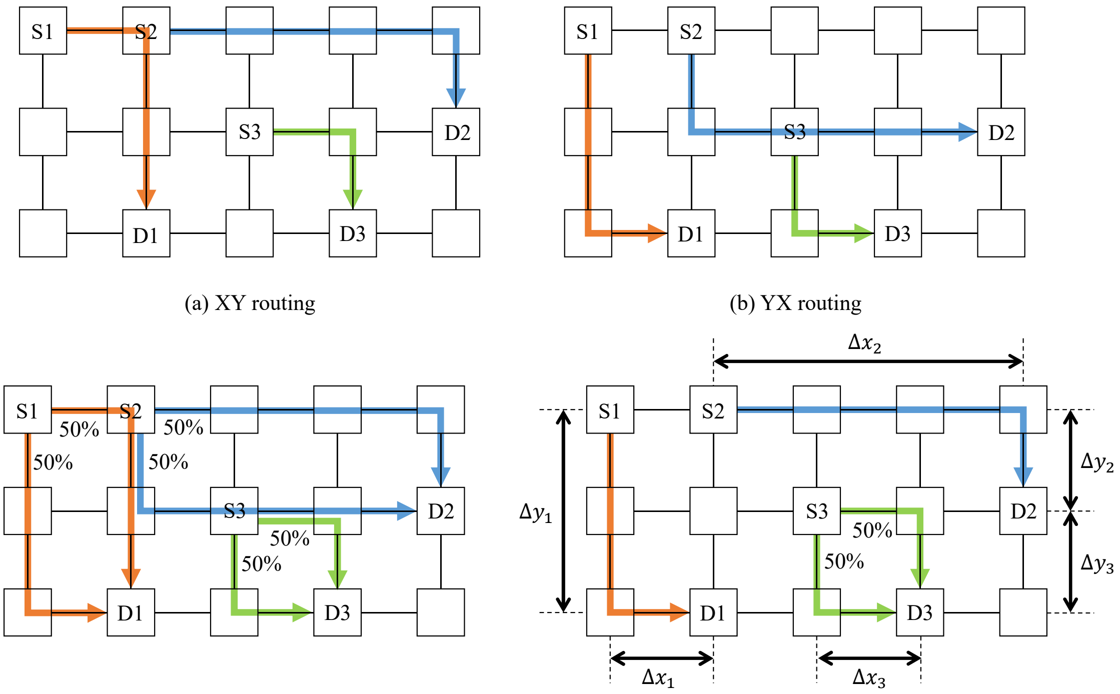
5.1 Introduction

This chapter shows the usability of the FPGA-based NoC emulator proposed in Chapter 4 by designing and modeling an effective routing algorithm for 2D mesh NoCs and evaluating it for various network sizes, from currently popular middle-scale sizes to future large-scale sizes.

In any NoC, the routing of messages sent between nodes plays a key role in achieving high performance. It is one of the major factors that determine how closely the network operates to the performance boundary set by the choice of topology. This chapter focuses on routing algorithms for 2D meshes which constitute an important class of NoCs that have been widely used in both commercial and research systems [11, 51, 43, 84, 40, 41, 44].

As explained in Chapter 2, existing routing algorithms can be classified into two categories: oblivious routing and adaptive routing. Oblivious routing algorithms do not use the network's state information in their routing decisions. On the other hand, adaptive routing algorithms consider the state of the network when determining a path for a packet from the source node to the destination node. Thus, they are potentially better in terms of performance. Despite this, oblivious routing algorithms are easier to implement, have much shorter routing computation delay, and do not incur much hardware overhead. Therefore, they are commonly used in practice.

The simplest and most popular oblivious routing algorithm for 2D meshes is the dimension-order routing (DOR) algorithm. As shown in Figure 5.1(a), in the XY DOR algorithm, packets are routed first in the X dimension and then in the Y dimension to reach their destinations. On the other hand, in the YX DOR algorithm (Figure 5.1(b)), packets are routed first in the Y dimension and then in the X dimension. Besides the simplicity, the DOR algorithm has the major advantage of being deadlock-free and thus has been employed in many practical systems.



(a) XY routing

(b) YX routing

(c) O1TURN

(d) Long Edge First (LEF)

Figure 5.1: (a) XY DOR: packets are routed first in the X dimension and then in the Y dimension to reach their destinations. (b) YX DOR: packets are routed first in the Y dimension and then in the X dimension. (c) O1TURN [7] combines XY DOR and YX DOR: the first dimension of traversal is chosen randomly. (d) LEF: a packet is routed in the X dimension first (XY DOR) if the difference of the X coordinates of the source node and the destination node (Δx) is greater than the difference of the Y coordinates (Δy); otherwise, if Δy is greater than Δx, the packet is routed in the Y dimension first (YX DOR); in the case that Δx is equal to Δy, the first dimension of traversal is chosen randomly like in O1TURN.

Since the DOR algorithm offers no path diversity, it does a poor job of load balancing the network under many traffic patterns. To address this problem, several oblivious routing algorithms including Valiant's [127], ROMM [128], and O1TURN [7] have been proposed.

Valiant's algorithm [127] offers a high path diversity by routing each packet from the source node to the destination node via a randomly chosen intermediate node. With the high path diversity, this routing algorithm achieves optimal worst-case throughput[1]. However, it suffers from low average-case throughput and high latency because a packet may traverse from source to destination through a non-minimal path.

ROMM [128] is similar to Valiant's algorithm in which it routes each packet from the source node to the destination node via an intermediate node[2]. However, the intermediate node in ROMM must be within the minimum rectangle defined by the source node and the destina-

[1] The minimum throughput over all traffic patterns.

[2] Here, we consider the two-phase ROMM which is the most popular version of ROMM. In general, in the n-phase ROMM algorithm, a packet is routed from source to destination via $n-1$ intermediate nodes.

tion node whereas the intermediate node in Valiant's algorithm can be anywhere in the network. Because of this, packets are always routed through minimal paths in ROMM. Thus, ROMM has better average-case throughput and latency than Valiant's algorithm. However, it has been demonstrated that, in the worst case, ROMM may deliver lower performance than the DOR algorithm in 2D tori and meshes [129, 7].

O1TURN [7] is one of the most effective oblivious routing algorithms for 2D meshes known so far. As shown in Figure 5.1(c), this routing algorithm combines XY DOR and YX DOR. The first dimension of traversal is chosen randomly. It has been shown that O1TURN can deliver higher performance than the DOR algorithm as well as Valiant's algorithm and ROMM.

Most of the routing algorithms proposed for 2D meshes have been evaluated on only symmetric networks of size $n \times n$. However, in practice, the network size can be $m \times n$ where $m \neq n$. For example, the 2D mesh network used in the Intel Xeon Phi Knights Landing architecture [11] is 6×9. Moreover, when multiple parallel applications are mapped into a 2D-mesh-based system, each application does not necessarily have to be mapped into a symmetric sub-mesh region. Thus, it is crucial to evaluate routing algorithms on both symmetric and asymmetric networks.

This chapter proposes *LEF (Long Edge First)*, a new oblivious routing algorithm for 2D meshes. LEF offers high throughput with low design complexity and is especially effective when the communication is within an asymmetric mesh. LEF's basic idea comes from conventional wisdom in choosing the appropriate DOR algorithm for supercomputers with asymmetric mesh or torus interconnects [130, 131, 132]: routing longest dimensions first provides better performance than other strategies; that is, for example, if the interconnect is an asymmetric 2D mesh with the X dimension longer than the Y dimension, then the XY DOR is preferred over the YX DOR. This wisdom has also been adopted in some commercial many-core processors. For instance, the 2D-mesh-based Intel Xeon Phi Knights Landing architecture [11] employs the YX DOR because the Y dimension is longer than the X dimension.

Routing longest dimensions first can deliver higher performance because the pressure on the channel buffers is reduced as packets tend not to move from a lightly loaded channel to a heavily loaded one when changing from a dimension to another [130, 131, 132]. The reason for this is that a channel on a longer dimension is potentially shared by more flows and hence more heavily utilized. The results in this thesis confirm that, in a 16×8 network (a mesh with the X dimension longer than the Y dimension), the XY DOR algorithm offers 10.5% and 24.7% higher throughput than the YX DOR algorithm under a uniform and a hotspot traffic, respectively.

Unlike the conventional wisdom, decisions of selecting the appropriate DOR algorithm in LEF are not fixed to the network shape but instead made on a per-packet basis. The first dimension of traversal of a packet is the one in which the packet needs to traverse more hops. In this way, like O1TURN, LEF distributes the load over both XY and YX paths and thus outperforms the DOR algorithm when the traffic pattern is non-uniform. Moreover, in the cases of global communication in an asymmetric network and local communication within an asymmetric region, there are more source-destination pairs of which the distance in the longer dimension is greater than that in the shorter dimension. Thus, in LEF, the majority of packets traverse the longer dimension, which is more heavily loaded, first. Therefore, the pressure on the channel buffers is lower than in O1TURN where 50% of packets traverse the longer dimension first and the remaining 50% traverse the shorter dimension first. By balancing between distributing the load over both XY and YX paths and reducing the pressure on the channel buffers, LEF can achieve higher throughput than other oblivious routing algorithms including O1TURN.

When LEF is used, deadlock may occur because of the combination of XY DOR and YX DOR. The thesis proposes an efficient deadlock avoidance method for LEF in which the use of virtual channels (VCs) is more flexible than in the conventional method used by O1TURN. Using the proposed method, we can expect a more effective utilization of VCs which contributes to improving the overall performance.

Due to the lack of fast modeling methodologies that can provide a high degree of accuracy, most existing routing algorithms have not been evaluated in large-scale NoCs which consist of thousands of nodes. This chapter uses the fast and cycle-accurate FPGA-based NoC emulator proposed in the previous chapters to evaluate LEF and its counterparts in networks of various sizes ranging from 8×8 to 128×64.

The original idea of LEF is proposed by Sasakawa and Kise in [71]. This thesis makes the following major extensions.

First, the thesis optimizes the selection scheme of XY and YX DOR in LEF. By treating packets of which routed paths lie on only one dimension different from the others, the thesis relaxes the restrictions that must be imposed for deadlock freedom. Moreover, in [71], XY DOR is always selected when the difference of the X coordinates of the source node and the destination node is equal to the difference of the Y coordinates; however, the thesis finds that this selection strategy does not perform well under traffics patterns like *transpose* where most (or all) of the source-destination pairs have the distance in the X dimension equal to that in the Y dimension. Therefore, the proposed selection scheme is not fixed to XY DOR when the difference of the X coordinates of the source node and the destination node is equal to the difference of the Y coordinates but instead made randomly between XY DOR and YX DOR.

Second, the thesis proposes a more efficient deadlock avoidance method for LEF in which the use of VCs is more flexible. The evaluation results show that this deadlock avoidance method outperforms the conventional method and can also be used to improve the performance of O1TURN.

Third, the thesis provides much more detailed analysis and evaluation results. The thesis compares LEF against not only the DOR algorithm but also O1TURN, one of the best oblivious routing algorithms for 2D meshes known so far, and a minimal adaptive routing algorithm based on the odd-even turn model [72]. By using the proposed FPGA-based NoC emulator, the thesis

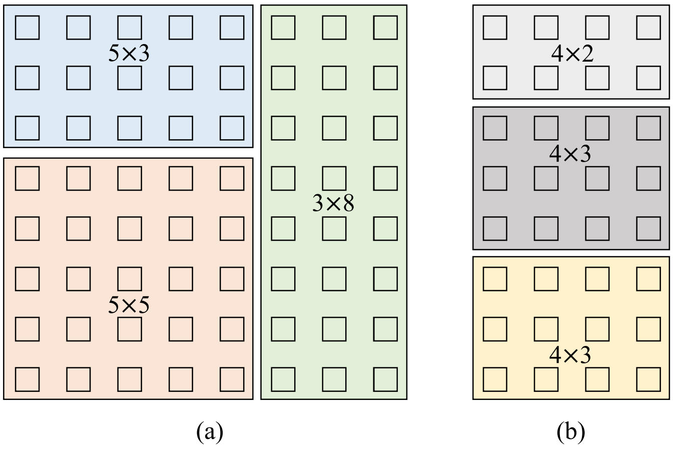


Figure 5.2: (a) An example of mapping three parallel applications into an 8×8 mesh. (b) An example of mapping three parallel applications into a 4×8 mesh.

examines the routing algorithms on NoCs of various sizes from 8×8 to 128×64 and shows that their performance is strongly affected by the resource allocation policy in the network and the effects are different for each routing algorithm. This result would not be obtained if modeling of large-scale NoCs could not be performed.

5.2 The LEF Routing Algorithm

5.2.1 Selection of XY DOR and YX DOR

Figure 5.1(d) shows the basic idea of the selection of XY DOR and YX DOR in LEF. When routing a packet, which DOR algorithm is chosen depends on the relative position between the source node and the destination node. The XY DOR will be chosen if the distance of the X coordinates of the source node and the destination node (Δx) is greater than the distance of the Y coordinates (Δy). Otherwise, if Δy is greater than Δx, the YX DOR will be chosen. In the case that Δx is equal to Δy, the first dimension of traversal will be selected randomly.

In the example in Figure 5.1(d), because Δy_1 is greater than Δx_1, packets sent from node S1 to node D1 are routed with YX DOR. On the other hand, packets sent from node S2 to node D2 are routed with XY DOR because Δx_2 is greater than Δy_2. Finally, packets sent from node S3 to node D3 can be routed with both XY DOR and YX DOR (randomly chosen) because Δx_3 is equal to Δy_3.

By selecting XY DOR or YX DOR for each packet as described above, LEF naturally distributes the load over both XY and YX paths. It thus achieves better load balancing than the DOR algorithm which routes every packet in a fixed dimension first. Moreover, the XY DOR and YX DOR selection strategy makes LEF especially effective in the case that the nodes participating in communication lie on an asymmetric mesh. This includes the global communication in an asymmetric network and the local communication occurring when multiple parallel applications are mapped into a network (may be symmetric or asymmetric) like in Figure 5.2.

Figure 5.2(a) illustrates the situation of mapping three parallel applications into three sub-meshes 5×3, 5×5, and 3×8 of an 8×8 mesh. Interestingly, if we use the XY DOR algorithm for this network, the application running in the 5×3 sub-mesh will get benefited but the one running in the 3×8 sub-mesh will be harmed. On the other hand, if the YX DOR algorithm is used, the application running in the 3×8 sub-mesh will get benefited but the one running in the 5×3 sub-mesh will be harmed. By using LEF, the performance of both applications in the two sub-meshes will be boosted.

Figure 5.2(b) illustrates the situation of mapping three parallel applications into three sub-meshes 4×2, 4×3, and 4×3 of a 4×8 mesh. According to the conventional wisdom explained in Section 5.1, we should use the YX DOR algorithm for this network because the Y dimension (8) is longer than the X dimension (4). However, for the mapping pattern in Figure 5.2(b), the XY DOR algorithm is more suitable since every sub-mesh has the X dimension longer than the Y dimension. By using LEF, this problem will disappear. This is because LEF is effective regardless of which dimension is longer.

Besides the better load balancing, the preceding discussion reveals a major advantage of LEF over the DOR algorithm: LEF is effective under any application mapping pattern. This advantage is significant since the application mapping pattern often changes over time.

In the case that the nodes participating in communication lie on an asymmetric mesh, LEF can be more effective than O1TURN because LEF causes a lower pressure on the channel buffers. For simplicity, we assume that the X dimension of the asymmetric mesh is longer than the Y dimension. In this case, a channel in the X dimension is potentially shared by more flows and thus more heavily loaded than a channel in the Y dimension. Also, there are more source-destination pairs of which the distance in the X dimension is greater than that in the Y dimension. Therefore, in LEF, the majority of packets traverse the X dimension first and thus tend to move from a heavily loaded channel to a lightly loaded one when turning to the Y dimension. This results in a lower pressure on the channel buffers than in O1TURN where 50% of packets tend to move from a lightly loaded channel to a heavily loaded one when turning from the Y dimension to the X dimension.

5.2.2 Deadlock Avoidance

In the explanations below, a packet that is routed with the XY DOR is called an *XY packet*. Similarly, a *YX packet* is the one routed with the YX DOR.

Since LEF combines XY DOR with YX DOR, dependency cycles involving XY packets and YX packets may be formed, and thus deadlock may occur. This is the same as in O1TURN.

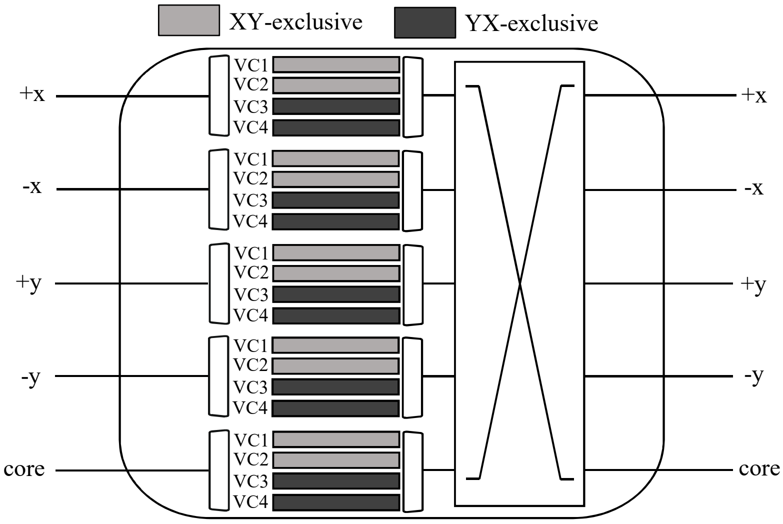


Figure 5.3: O1TURN [7] avoids deadlock by completely separating XY packets and YX packets. In each physical channel, half of the VCs are for XY packets while the other half for YX packets.

As shown in Figure 5.3, O1TURN avoids deadlock by completely separating XY packets and YX packets. It requires two or more VCs per physical channel. In each physical channel, half of the VCs are for XY packets while the other half for YX packets. In this way, it is clear that no dependency cycle can be formed, and therefore, deadlock does not occur.

5.2.2.1 Proposed Deadlock Avoidance Method

This thesis proposes a new deadlock avoidance method in which the use of VCs is more flexible than in the O1TURN's method described above. The proposed method requires that the atomic VC allocation policy is used. In this policy, a VC can be re-allocated to a new packet only when it is empty. Thus, at any given time, a VC can be occupied by only one packet. This is also an essential requirement for making fully adaptive routing algorithms deadlock-free [133].

Figure 5.4 shows two typical deadlock situations that occur when a non-atomic VC allocation policy is used. In the first situation shown in the left-hand side of Figure 5.4, a deadlock occurs because four packets P1, P2, P3, P4 form a dependency cycle. The second deadlock situation shown in the right-hand side of Figure 5.4 involves eight packets P1, P2, P3, P4, P5, P6, P7, and P8.

Like in the O1TURN's method, the number of VCs per physical channel (v) in the proposed method must be also greater than or equal to two ($v \geq 2$). Two design options are provided:

1. **Y-restricted**: reserve k VCs ($1 \leq k \leq v - 1$) in each physical channel in the Y dimension exclusively for XY packets (Figure 5.5(a)).

2. **X-restricted** – reserve k VCs ($1 \leq k \leq v - 1$) in each physical channel in the X dimension

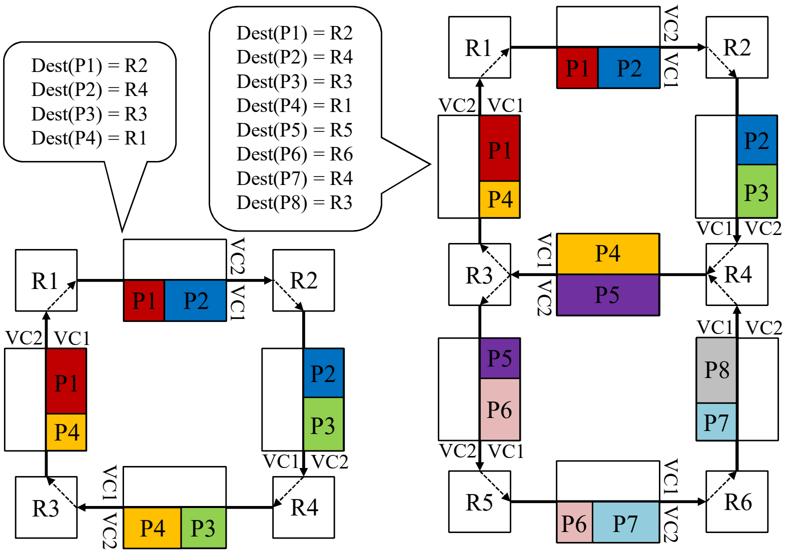


Figure 5.4: Two typical deadlock situations that occur when a non-atomic VC allocation policy is used, that is, a VC can be occupied by two or more packets at the same time.

exclusively for YX packets (Figure 5.5(b)).

The VCs that are reserved exclusively for XY packets and YX packets are called *XY-exclusive* VCs and *YX-exclusive* VCs, respectively. In the Y-restricted approach, YX packets are prohibited from using XY-exclusive VCs in the Y dimension while there is no restriction on VCs that XY packets can use. Similarly, in the X-restricted approach, XY packets are prohibited from using YX-exclusive VCs in the X dimension while YX packets can use any VC in both the X and Y dimensions. The proof of deadlock freedom will be presented in Section 5.2.2.2.

The decision of which design option, Y-restricted or X-restricted, is chosen is taken according to the shape of the network. In the case the X dimension is longer than the Y dimension, the Y-restricted approach is chosen. This is because there are fewer Y channels than X channels. Also, there is a high probability that a channel on the Y dimension is shared by fewer flows and thus less heavily utilized than a channel on the X dimension. Thus, using the Y-restricted approach is better. For the similar reasons, the X-restricted approach is chosen in the case the Y dimension is longer than the X dimension. For symmetric networks, either approach can be chosen[3].

As will be described in the proof of deadlock freedom in Section 5.2.2.2, exclusive VCs are the keys that prevent potential deadlock situations from becoming real ones. The number of exclusive VCs per channel k affects how fast a potential deadlock situation can be broken. Using too few exclusive VCs may make it too slow to break potential deadlock situations, especially at high loads, which may hurt the overall network performance. On the other hand, using too many exclusive VCs may make the non-exclusive VCs in the same dimension with the exclusive VCs

[3] In the evaluation in Section 5.3, the Y-restricted approach is adopted for symmetric networks

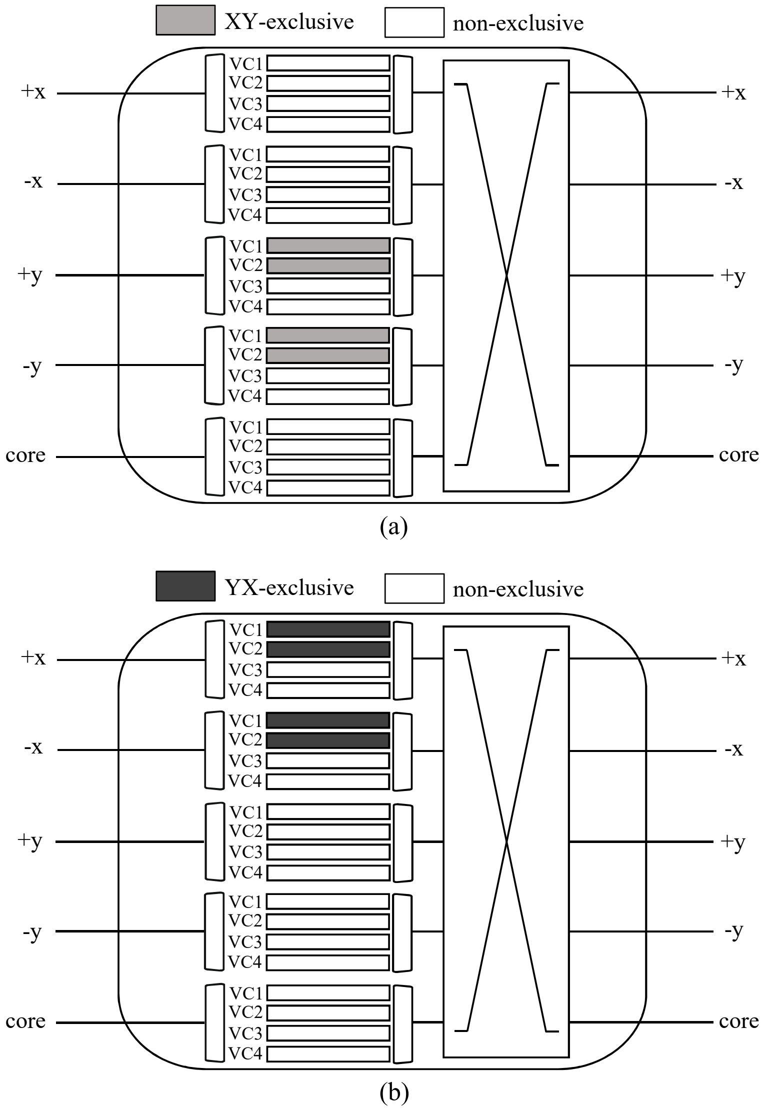


Figure 5.5: The use of VCs in the proposed deadlock avoidance method. Two design options are provided: (a) *Y-restricted* and (b) *X-restricted*.

too congested, which in turn also leads to poor performance. Because of the above reasons, in the evaluation in Section 5.3, k is set to $v/2$ where v is the number of VCs per channel.

Using the proposed method, a more effective utilization of VCs than in the O1TURN's method can be expected. This contributes to improving the overall performance, which we will see in Section 5.3.

5.2.2.2 Proof of Deadlock Freedom

This section will prove that deadlock does not occur in LEF when the proposed deadlock avoidance method in Section 5.2.2.1 is used. The proof for the X-restricted approach (Figure 5.5(b)) is provided. The proof for the Y-restricted approach (Figure 5.5(a)) is almost the same. The proof

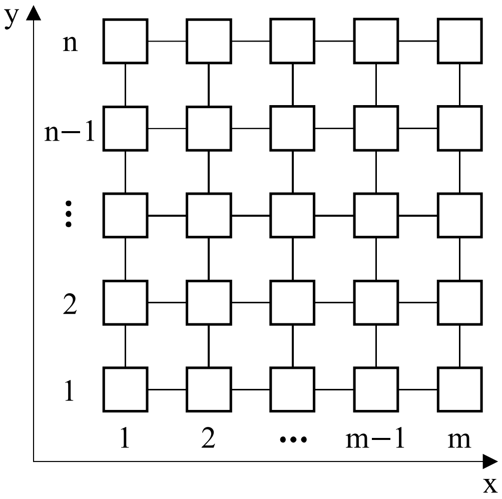


Figure 5.6: Coordinates of nodes in an m×n mesh.

assumes an m×n mesh shown in Figure 5.6 and is organized in a series of three lemmas and one theorem.

Lemma 5.2.1. *Any YX-exclusive VC is eventually released.*

Proof. As shown in Fig. 5.5(b), YX-exclusive VCs are located at the input ports +x and -x. They can be occupied by only YX packets. Thus, there are only two cases for a packet that is currently in a YX-exclusive VC.

- Case 1: the packet already arrived at its destination.
- Case 2: the packet will proceed to the next channel in the X dimension.

In case 1, the packet will be forwarded to the processing core of the current node. Thus, the YX-exclusive VC that the packet is occupying will be released. Below we focus on case 2.

i) input port -x of node (m,1). First, we consider the YX-exclusive VCs in input port -x of node (m,1). Case 2 never happens since output port +x of node (m,1) is not connected to any other nodes. Thus, the YX-exclusive VCs in input port -x of node (m,1) are eventually released.

ii) input port -x of node (m-1,1). Next, we consider the YX-exclusive VCs in input port -x of node (m-1,1). When case 2 happens, the packet should be transmitted to node (m,1). At least, the YX-exclusive VCs in input port -x of node (m,1) are eventually released by i). Thus, the packet will be certainly forwarded to node (m,1) and the YX-exclusive VC in input port -x of node (m-1,1) currently occupied by the packet will be released.

iii) input ports -x of the remaining nodes in row 1 (nodes from (m-2,1) to (1,1)). With similar arguments as in i) and ii), we can prove that the YX-exclusive VCs in the input ports -x of nodes from (m-2,1) to (2,1) are surely released. Input port -x of node (1,1) is not connected to any other nodes and thus can be ignored.

iv) input ports -x of nodes in rows 2, 3, $\cdots$, n. The proofs for rows 2, 3, $\cdots$, n are similar to that of row 1 (from i) to iii)).

v) input ports +x. By the similar arguments as from i) to iv), we can prove that the YX-exclusive VCs in input ports +x of all nodes are also eventually released.

By i) – v), we can conclude that any YX-exclusive VC is eventually released. □

Lemma 5.2.2. *Any VC in the Y dimension is eventually released.*

Proof. As shown in Fig. 5.5(b), all VCs in the Y dimension are non-exclusive VCs. Since both XY packets and YX packets can utilize non-exclusive VCs, there are three cases for a packet that is currently in a VC in an input port +y or -y.

- Case 1: the packet already arrived at its destination.

- Case 2: the packet will proceed to the next channel in the Y dimension.

- Case 3: the packet will proceed to the next channel in the X dimension.

As same as in the proof of Lemma 5.2.1, case 1 is trivial. Below we focus on case 2 and case 3.

i) input port -y of node (1,n). First, we consider the VCs in input port -y of node (1,n). Case 2 never happens since output port +y of node (1,n) is not connected to any other nodes. When case 3 happens, the packet should be transmitted to the next channel via output port +x or -x (note that output port -x of node (1,n) is not connected to any other nodes; we add it here for the generality of explanations). In this situation, the packet can be surely sent to the X dimension since at least the YX-exclusive VCs in input port -x or +x of the next node is surely released by Lemma 5.2.1. Thus, the VCs in input port -y of node (1,n) are eventually released.

ii) input port -y of node (1,n-1). Next, we consider the VCs in input port -y of node (1,n-1). When case 2 happens, the next node of the packet is (1,n). The VCs in input port -y of node (1,n-1) are eventually released since the VCs in input port -y of node (1,n) are eventually released by i). When case 3 happens, the VCs in input port -y of node (1,n-1) are also eventually released as similarly discussed in i). Therefore, the VCs in input port -y of node (1,n-1) are eventually released in every case.

iii) input ports -y of the remaining nodes in column 1 (nodes from (1,n-2) to (1,1)). As similarly discussed in i) and ii), we can prove that the VCs in the input ports -y of nodes from (1,n-2) to (1,2) are surely released. Input port -y of node (1,1) is not connected to any other nodes and thus can be ignored.

iv) input ports -y of nodes in columns 2, 3, $\cdots$, m. The proofs for columns 2, 3, $\cdots$, m are similar to that of column 1 (from i) to iii)).

v) input ports +y. All VCs in the input ports -y of all nodes are eventually released by i) – iv). In the same way, we can prove that all VCs in the input ports +y of all nodes are also eventually released.

By i) – v), we can conclude that any VC in the Y dimension is eventually released. □

Lemma 5.2.3. *Any non-exclusive VC in the X dimension is eventually released.*

Proof. This lemma will be proved using Lemma 5.2.1 and Lemma 5.2.2. There are three cases for a packet that is currently in a non-exclusive VC in the X dimension.

- Case 1: the packet already arrived at its destination.
- Case 2: the packet will proceed to the next channel in the X dimension.
- Case 3: the packet will proceed to the next channel in the Y dimension.

As same as in the proof of Lemma 5.2.1, case 1 is trivial. For case 2, the VC is eventually released because at least the YX-exclusive VCs in input port +x or -x of the next node is eventually released by Lemma 5.2.1. For case 3, the VC is also eventually released because all VCs in input port +y or -y of the next node are eventually released by Lemma 5.2.2. Thus, any non-exclusive VC in the X dimension is eventually released. □

We can easily derive the following theorem using Lemma 5.2.1, Lemma 5.2.2, and Lemma 5.2.3.

Theorem 5.2.4. *All VCs in the network are eventually released.*

Since all VCs in the network are eventually released by Theorem 5.2.4, LEF is deadlock-free. The proposed deadlock avoidance method can also be used for any other routing algorithms that combine XY DOR and YX DOR. Section 5.3 will show that it helps to improve the performance of O1TURN considerably.

5.2.3 Optimization on the Selection of XY DOR and YX DOR

Figure 5.7 shows the straightforward algorithm for selecting XY DOR and YX DOR. The behavior of this algorithm is the same as that described in Section 5.2.1. This section will first show that it may cause unnecessary restrictions on the VC usage of certain packets. Then a solution to eliminate those restrictions is proposed.

When combining XY DOR and YX DOR together, if deadlock arises, it is caused by only packets of which routed paths lie on both the X and Y dimensions. However, the VC usage restriction in the deadlock avoidance method described in Section 5.2.2.1 is applied to all packets. According to the algorithm in Figure 5.7, packets of which routed paths lie only on the X dimension ($\Delta x > 0$ and $\Delta y = 0$) are determined as XY packets. If the X-restricted approach

```
1:  **Input**: source node $S\,(x_S, y_S)$ and destination node $D\,(x_D, y_D)$
2:  **Output**: routing method $R$
3:  $\Delta x = |x_D - x_S|$
4:  $\Delta y = |y_D - y_S|$
5:  if $\Delta x > \Delta y$ then
6:      $R =$ XY DOR
7:  else if $\Delta x < \Delta y$ then
8:      $R =$ YX DOR
9:  else
10:     $R =$ Random(XY DOR or YX DOR)
11: end if
```

Figure 5.7: The straightforward algorithm for selecting XY DOR and YX DOR described in Section 5.2.1.

(Figure 5.5(b)) is used, these packets will be prevented from using the YX-exclusive VCs which are exclusively reserved for YX packets. Likewise, packets of which routed paths lie only on the Y dimension ($\Delta x = 0$ and $\Delta y > 0$) are determined as YX packets. Thus, if the Y-restricted approach (Figure 5.5(a)) is used, these packets will be prevented from using the XY-exclusive VCs which are exclusively reserved for XY packets. These restrictions are unnecessary and may make the exclusive VCs underutilized while the non-exclusive VCs are congested, especially when the traffic is dominated by packets of which routed paths lie on only one dimension.

The unnecessary restrictions described above are eliminated by slightly modifying the algorithm for selecting XY DOR or YX DOR as shown in Figure 5.8. Here we assume that the Y-restricted deadlock avoidance method (Figure 5.5(a)) is used. When the path of a packet lies only on the Y dimension ($\Delta x = 0$), it is determined as an XY packet and thus can use all VCs in the Y dimension. The algorithm for the case the X-restricted deadlock avoidance method (Figure 5.5(b)) is used is similar to that in Figure 5.8. We only have to modify lines 5 and 6 as follows:

```
if $\Delta y == 0$ then
    $R =$ YX DOR.
```

This means that, when the path of a packet lies only on the X dimension ($\Delta y = 0$), it is determined as a YX packet and thus can use all VCs in the X dimension. In both cases of the deadlock avoidance method, the selection of XY routing and YX routing for packets that must traverse both X and Y dimensions to reach their destinations is the same as in the straightforward algorithm in Figure 5.7.

```
 1: **Input**: source node $S\,(x_S, y_S)$ and destination node $D\,(x_D, y_D)$
 2: **Output**: routing method $R$
 3: $\Delta x = |x_D - x_S|$
 4: $\Delta y = |y_D - y_S|$
 5: **if** $\Delta x == 0$ **then**
 6:     $R$ = XY DOR
 7: **else**
 8:     **if** $\Delta x > \Delta y$ **then**
 9:         $R$ = XY DOR
10:     **else if** $\Delta x < \Delta y$ **then**
11:         $R$ = YX DOR
12:     **else**
13:         $R$ = Random(XY DOR or YX DOR)
14:     **end if**
15: **end if**
```

Figure 5.8: The algorithm for selecting XY DOR and YX DOR in LEF for the case the Y-restricted deadlock avoidance method (Figure 5.5(a)) is used. For the case the X-restricted deadlock avoidance method (Figure 5.5(b)) is used, lines 5 and 6 should be modified as follows: **if** $\Delta y == 0$ **then** R = YX DOR.

5.2.4 LEF Implementation

The conventional input-queued VC router architecture with five pipeline stages described in Chapter 2 is used as the baseline. The routing information is carried by the head flit.

LEF is implemented by adding one bit to the flit structure. This bit indicates whether a packet is an XY or a YX packet, which is determined at the source node and passed along with the head flit to all nodes of the route. In this way, every packet is always aware of its type (XY or YX), and therefore, it always knows which VCs it can use since the information of which VCs are XY/YX-exclusive and which VCs are non-exclusive is predetermined. In the router implementation, some logic is added to ensure that an XY packet does not send requests for YX-exclusive VCs (and similarly, a YX packet does not send requests for XY-exclusive VCs) to the VC allocator.

5.3 Evaluation

5.3.1 Evaluation Methodology

As mentioned in Section 5.1, the proposed FPGA-based NoC emulator is used in the evaluation. Apart from the advantages of emulation speed and accuracy, the proposed NoC emulator supports

Table 5.1: Emulation parameters

Middle-scale NoCs	8×8, 16×8, 16×16
Large-scale NoCs	64×64, 128×64
Router architecture	Input-queued VC router
Router pipeline	5-stage
# of VCs per physical channel	4
VC size	4-flit
VC/Switch allocator	iSLIP [122]
Arbiter type	Round-robin
Flow control	Wormhole & credit-based
Packet length	16-flit
Injection process	Bernoulli process
Traffic pattern	Uniform, hotspot, transpose

asymmetric networks which are not officially supported by some widely used software-based NoC simulators like BookSim [5].

Table 5.1 shows the emulation parameters. Both symmetric and asymmetric meshes are considered. The results for five network sizes are presented: 8×8, 16×8, 16×16, 64×64, and 128×64. Each physical channel has four VCs, each can hold four flits.

Three traffic patterns are considered: uniform, hotspot, and transpose. Under the uniform traffic, the destination of each packet is randomly selected with equal probability among all nodes in the network. Under the hotspot traffic, there are four hotspot nodes (a 2×2 cluster) in the middle-scale NoCs and 256 hotspot nodes (a 16×16 cluster) in the large-scale NoCs. In every case, the hotspot nodes are located at a corner of the network and receive four times more traffic than the other nodes. The transpose traffic pattern is only used with symmetric meshes. Under this traffic pattern, node (i, j) only sends messages to node (j, i).

For each traffic pattern, each of the NoCs is emulated with different flit injection rates: from a low injection rate at which there is almost no contention in the network to a high injection rate at which the network is saturated. The emulation at each injection rate is executed in three phases: warm-up (middle-scale NoCs: 100,000 cycles; large-scale NoCs: 200,000 cycles), measurement (middle-scale NoCs: 100,000 cycles; large-scale NoCs: 200,000 cycles), and drain. The average packet latency is calculated based on latencies of all packets generated during the measurement phase. The network throughput is calculated from the number of packets arriving at their destinations during the measurement phase.

5.3.2 LEF Performance

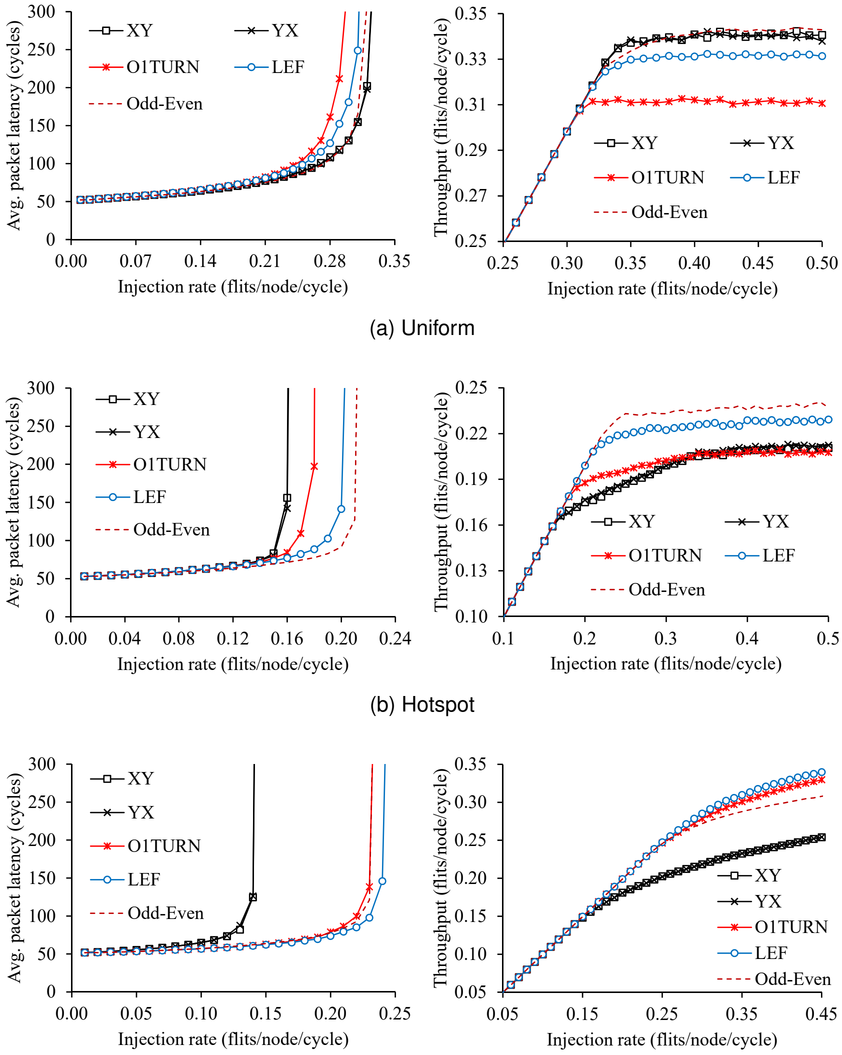
5.3.2.1 Middle-Scale NoCs

LEF is compared against the DOR algorithm (both XY DOR and YX DOR), O1TURN [7], and a minimal adaptive routing algorithm based on the odd-even turn model [72]. In the adaptive routing algorithm, when there are two available output ports, the selection strategy selects the port with more free VCs. A more complicated selection strategy may provide better performance but can also produce diminishing results if it makes the router critical path longer. In the description below and in the result graphs, the adaptive routing algorithm is denoted by *Odd-Even*. The results of Odd-Even are included to show that LEF can provide comparable or even slightly better performance than complex adaptive routing algorithms. The main objective is to compare LEF with O1TURN and the DOR algorithm which are in the same category of oblivious routing algorithms as LEF.

Figures 5.9, 5.10, and 5.11 show the average packet latency and throughput results of three network sizes 8×8, 16×8, and 16×16, respectively. The results vary with both the shape of the network and the offered traffic pattern.

In the symmetric 8×8 NoC, the XY DOR and YX DOR algorithms provide almost the same results regardless of the offered traffic pattern. They outperform the other routing algorithms under the uniform traffic. This result is the same as those reported in [126] and [72] which indicate the superiority of the DOR algorithm under the uniform traffic. The main reason for this is that the DOR algorithm incorporates more global, long-term information about the characteristics of the uniform traffic pattern and thus can make the traffic distribution more even [126, 72]. LEF, O1TURN, and Odd-Even show their strengths with the non-uniform traffics. Under the hotspot traffic, thanks to the adaptability of routing decisions to the state of the network, Odd-Even performs better than the other routing algorithms. Under the transpose traffic, LEF, O1TURN, and Odd-Even deliver by far better performance than the DOR algorithm. In this traffic pattern, since all source-destination pairs have the distance in the X dimension equal to the distance in the Y dimension, LEF randomly selects XY DOR and YX DOR for every packet, which is the same as O1TURN. However, LEF is slightly better than O1TURN thanks to its effective deadlock avoidance method described in Section 5.2.2.1. This will be discussed in more details in Section 5.3.3. In summary, compared to the DOR algorithm, the throughput provided by LEF is around 3.3% lower under the uniform traffic but around 13.7% and 43.4% higher under the hotspot and transpose traffics, respectively. Compared to O1TURN, the throughput provided by LEF is around 5.7%, 10.8%, and 3.6% higher under the uniform, hotspot, and transpose traffics, respectively.

In the asymmetric 16×8 NoC, LEF is the best-performing routing algorithm. As expected, the XY DOR is better than YX DOR since the X dimension of the network is longer than the Y dimension. Under the uniform traffic, LEF delivers almost the same throughput as the XY



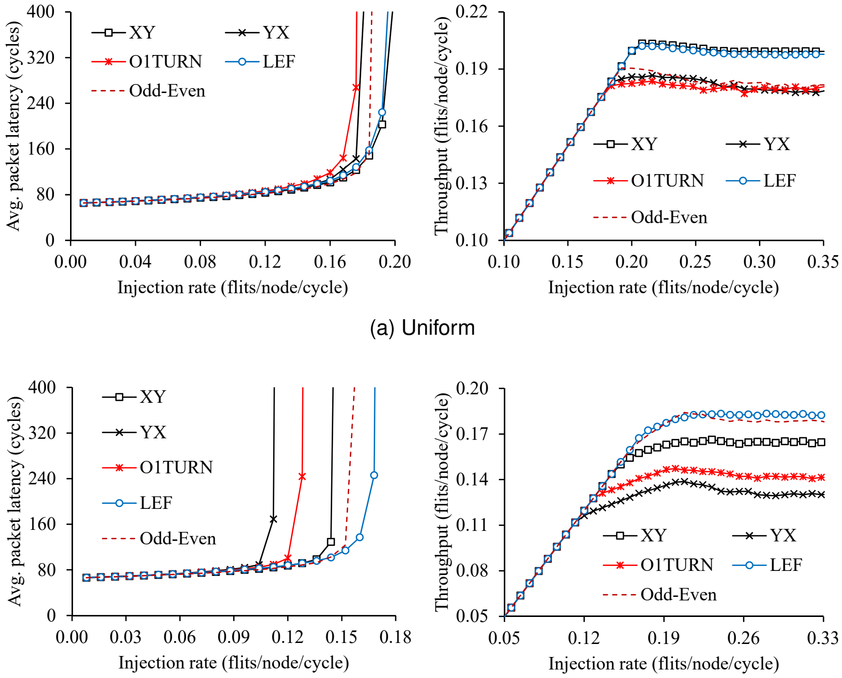
(a) Uniform

(b) Hotspot

(c) Transpose

Figure 5.9: 8×8 NoC: average packet latency and throughput results with three different traffic patterns.

DOR and around 9.3%, 10.1%, and 7.8% higher than the YX DOR, O1TURN, and Odd-Even, respectively. Under the hotspot traffic, the throughput provided by LEF is higher than all other algorithms with the approximate differences of 11%, 38.3%, 28.3%, and 2.2% compared to the XY DOR, YX DOR, O1TURN, and Odd-Even, respectively. Interestingly, O1TURN is outperformed by the XY DOR and LEF is slightly better than Odd-Even. The performance gain of LEF comes from two sources. First, as explained in Section 5.1, LEF balances between distributing the load over both XY and YX paths and reducing the pressure on the channel buffers by routing a packet along the dimension in which the source-destination distance is longer first. Second, LEF uses a more effective deadlock avoidance method than the conventional one which is used by O1TURN.



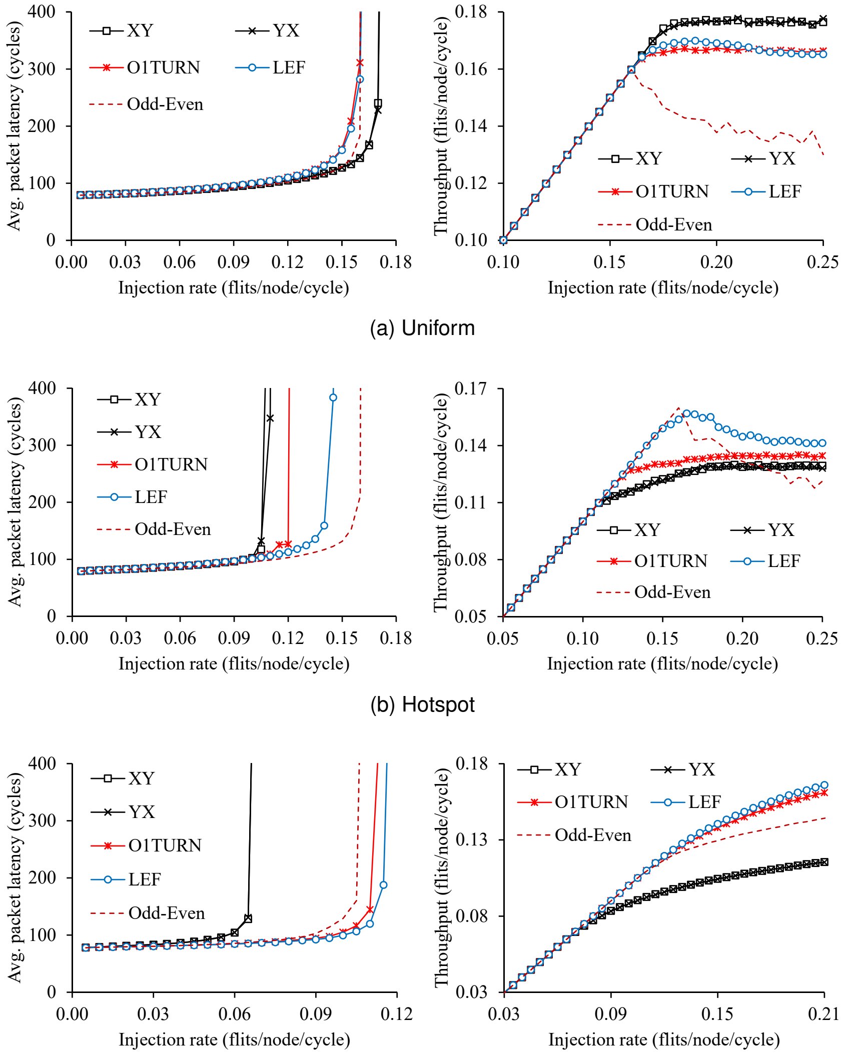
(a) Uniform

(b) Hotspot

Figure 5.10: 16×8 NoC: average packet latency and throughput results with two different traffic patterns.

The results in the 16×16 NoC are almost the same as those in the 8×8 NoC. However, there is a phenomenon that does not occur in the 8×8 NoC. When Odd-Even is used, the network becomes unstable at extremely high loads (Figures 5.11(a) and 5.11(b)). This phenomenon is caused by the selection of using round-robin arbiters for VC and link allocation. At high loads, it is likely that most VCs in the network are occupied at most the time. When Odd-Even is used, packets tend to have to compete for VC and link allocation more than in the case the DOR algorithm is used. For instance, with the XY DOR, a packet at input port +y or -y will not go to output ports +x and -x since the packet is routed first in the X dimension and then in the Y dimension to reach its destination. Because of this, when the XY DOR is used, there is less competition for the VCs and links in the X dimension. However, this is not the case for Odd-Even. Because of the higher competition for VCs and links, Odd-Even requires a better VC and link allocation strategy, especially for large networks where there exist many long paths. Currently, since round-robin arbiters are used, the VC and link arbitration is locally fair but globally unfair. The latency of a long-path packet may be extremely high because it is treated the same as short-path packets in every hop from source to destination.

Like Odd-Even, LEF also causes the 16×16 NoC slightly unstable at high loads. The reason is the same as for Odd-Even. However, compared to Odd-Even, LEF is less affected by the globally unfair resource allocation because half of the VCs in the X dimension are YX-exclusive.



(a) Uniform

(b) Hotspot

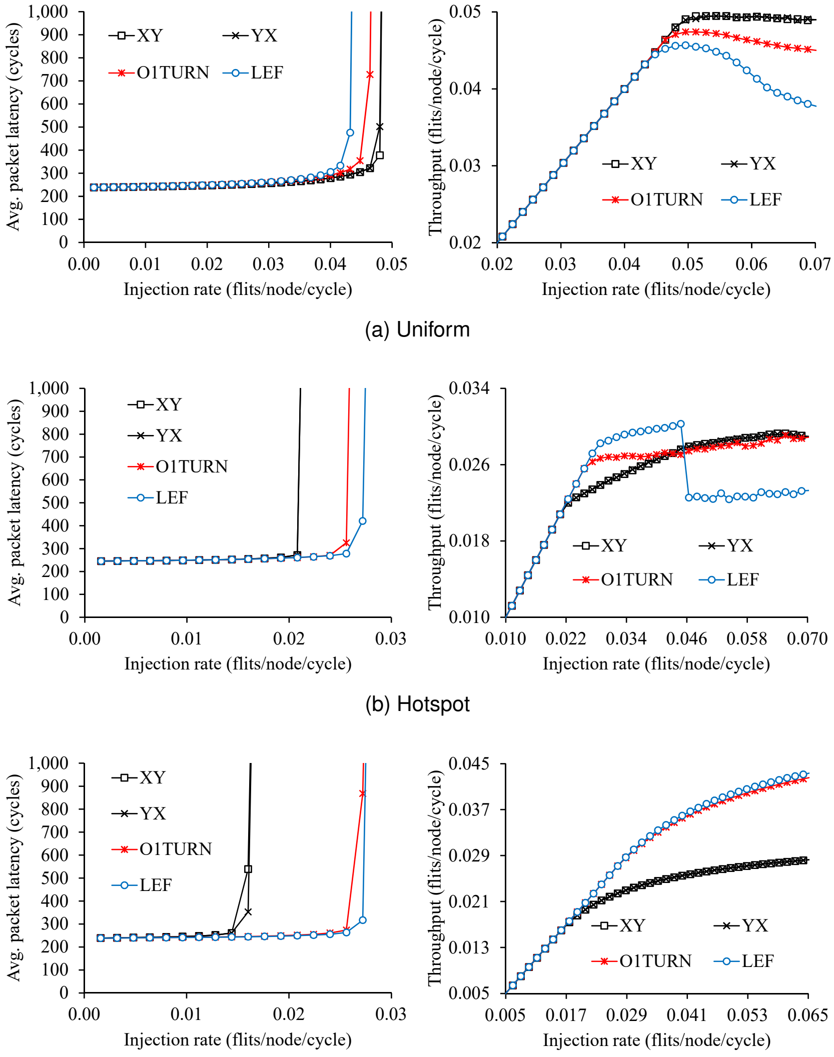
(c) Transpose

Figure 5.11: 16×16 NoC: average packet latency and throughput results with three different traffic patterns.

The competition for these VCs is low since only YX packets can occupy them. Despite this, we will see in Section 5.3.2.2 that the effect of the globally unfair resource allocation on LEF is significant when the network is large.

5.3.2.2 Large-Scale NoCs

This section presents the preliminary results of evaluating LEF on large-scale NoCs. Figures 5.12 and 5.13 show the average packet latency and throughput results of two networks 64×64 and 128×64, respectively. In most cases, the trend of average packet latency is the same as in the middle-scale networks. LEF is still particularly effective when the network is asymmetric. However, the effect of the global unfairness of resource allocation makes LEF perform worse than



(a) Uniform

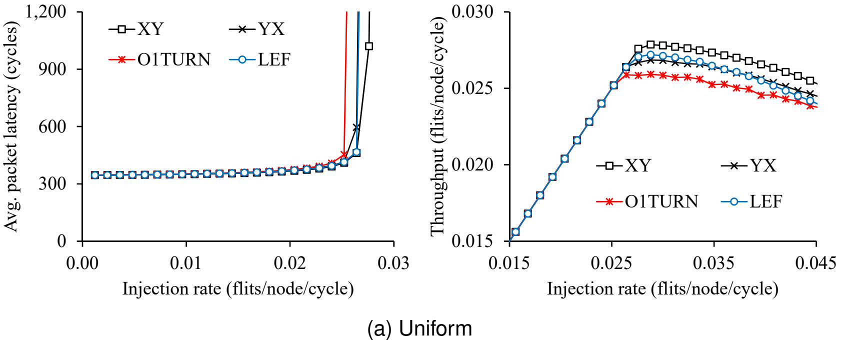
(b) Hotspot

(c) Transpose

Figure 5.12: 64×64 NoC: average packet latency and throughput results with three different traffic patterns.

in the middle-scale networks. For instance, under the uniform traffic, LEF is outperformed by O1TURN in the 64×64 NoC while the result in the 8×8 NoC is opposite. In terms of throughput, we can see that the large-scale NoCs are unstable at high loads when LEF is used. The reason is the same as that used to explain the behavior of Odd-Even in the 16×16 NoC in Section 5.3.2.1. To alleviate the problem, it is necessary to use a different arbitration policy which can handle long-path packets well.

O1TURN is less affected by the globally unfair resource allocation than LEF because it uses the deadlock avoidance method in which the VCs are completely separated into two layers, one for XY packets and the other for YX packets. In this way, there is less competition for each VC in O1TURN than in LEF and the effect of the globally unfair resource allocation is smaller.



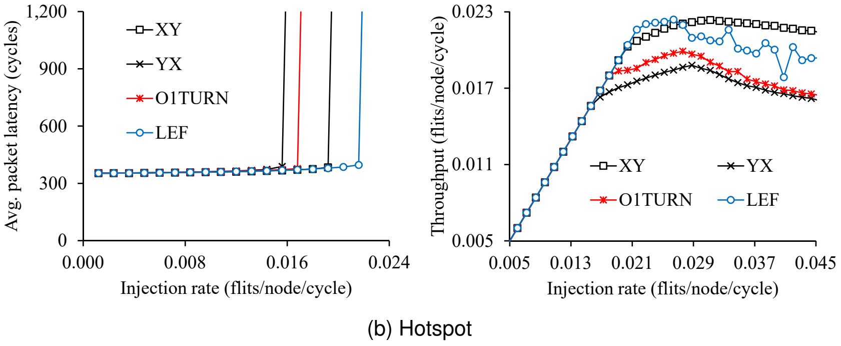


Figure 5.13: 128×64 NoC: average packet latency and throughput results with two different traffic patterns.

The results presented above make it clear that the performance of the routing algorithms is strongly affected by the resource allocation policy in the network and the effects are different for each algorithm. To promote the potential of LEF in large-scale NoCs, it is necessary to take into account the design of the resource allocation policy. The details are left as future work.

5.3.3 The Effectiveness of the Proposed Deadlock Avoidance Method

Figure 5.14 illustrates the effectiveness of the proposed deadlock avoidance method over the conventional method which is used by O1TURN [7]. The network size and traffic pattern here are 16×8 and hotspot, respectively. As mentioned earlier, the proposed method can also be applied to O1TURN. In the graphs in Figure 5.14, O1TURN++ indicates O1TURN with the proposed deadlock avoidance method while LEF-- indicates LEF with the conventional deadlock avoidance method. Thus, LEF and O1TURN++ use the same deadlock avoidance method. This is also the case for LEF-- and O1TURN.

Figure 5.14 shows that the throughputs provided by LEF and O1TURN++ are around 18.1% and 12%, respectively, higher than those provided by LEF-- and O1TURN. Therefore, the proposed deadlock avoidance method is effective not only for LEF but also for O1TURN.

Another result obtained from Figure 5.14 is that, when using the same deadlock avoidance method, LEF outperforms O1TURN. This indicates that the strategy of selecting the XY DOR

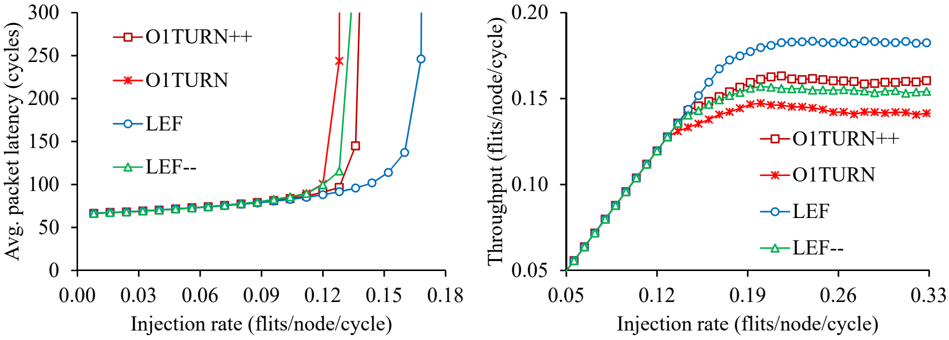


Figure 5.14: Comparison of the proposed deadlock avoidance method and the conventional method which is used by O1TURN [7]. In the graphs, O1TURN++ indicates O1TURN with the proposed deadlock avoidance method while LEF-- indicates LEF with the conventional deadlock avoidance method.

and YX DOR in LEF is better than that in O1TURN.

5.4 Summary

This chapter proposed LEF, a new oblivious routing algorithm for 2D meshes, and an effective deadlock avoidance method for it. By routing a packet along the dimension in which it needs to traverse more hops first, LEF balances between distributing the load over both XY and YX paths and reducing the pressure on the channel buffers, which leads to better load balancing than other oblivious routing algorithms like O1TURN. This, together with the proposed deadlock avoidance method, enables LEF to achieve higher performance than the DOR algorithm and O1TURN and provide comparable performance to a complicated adaptive routing algorithm. The evaluation results showed that, in an 8×8 NoC, the throughput provided by LEF was from around 3.6% to around 10.8% higher than O1TURN under three different traffic patterns. In a 16×8 NoC, LEF delivered up to around 28.3% higher throughput than O1TURN and was even better than the adaptive routing algorithm. The evaluation results also showed the effectiveness of the proposed deadlock avoidance method over the conventional method. The use of the proposed FPGA-based NoC emulator makes it possible to examine LEF and the other routing algorithms in NoCs with thousands of nodes in a practical time. The preliminary results showed that the performance of these routing algorithms was strongly affected by the resource allocation policy in the network and the effects were different for each algorithm. To maximize the potential of LEF in such large-scale NoCs, it is necessary to pay more attention to the resource allocation policy.

Like many previous studies on routing algorithms [72, 7, 134], the thesis presents evaluation results under only synthetic traffics. Although synthetic workloads can provide a relatively thorough coverage of the characteristics of the routing algorithms, evaluation results under real traffics may be required for some certain application-specific optimizations. Therefore, in the next chapter, the thesis extends the proposed NoC emulator to support emulation under trace-driven workloads that are based on trace data of real applications.

Chapter 6

Towards NoC Emulation under Trace-Driven Workloads

6.1 Introduction

In the trace-driven emulation approach, a NoC emulator replays a sequence of messages, called trace, captured from either a working system or an execution-driven simulation/emulation. However, due to the lack of trace data of large-scale NoC-based systems, using synthetic workloads is presently the only feasible approach for emulating large-scale NoCs with thousands of nodes. While synthetic workloads can provide a relatively thorough coverage of the characteristics of the target NoCs, evaluation on trace-driven workloads is still required in some cases such as assessing application-specific optimizations. The thesis takes this into account and extends the NoC emulator proposed in Chapter 4 to support trace-driven emulation which will be useful for research and development of large-scale NoCs in the future when trace data of large-scale NoC-based systems are available.

Besides the necessity of emulation with workloads created from realistic applications' trace data, it is crucial for an FPGA-based NoC emulator to support trace-driven emulation due to the following reason. Some synthetic workloads, especially those based on complex but effective synthetic workload generation methodologies like Synfull [120] and that proposed by Yin *et al.* [121], are difficult to model on FPGAs because they require complicated probability density functions and modulo, multiplication, division, and exponent operations. Even it is feasible to implement these functions and operations on FPGAs, it is likely that the operating frequency of the emulator drops substantially. Therefore, if the emulator supports trace-driven emulation, developing a software tool to generate trace data based on the descriptions of the complex synthetic workloads and then using the trace-driven emulation mode of the emulator would be a better approach.

It is difficult to effectively support trace-driven NoC emulation on FPGAs because trace data are often much larger than the total capacity of FPGA on-chip memory and thus must be stored in off-chip memory. Most of the existing FPGA-based NoC emulators simplify the control of loading trace data from the off-chip memory, generating messages based on the loaded trace data, and injecting the messages to the NoC by using soft processors like Microblaze or hard processors on SoC FPGAs. AcENoCs [62] and AdapNoC [67] use two Microblaze soft processors to be able to alleviate the off-chip memory access time. Specifically, one processor is responsible for loading trace data while the other is responding for generating and injecting messages to the NoC. In this way, the trace load operation is overlapped with the message generation and injection operations, and therefore, the effect of the off-chip memory access time is reduced. DuCNoC [68] also adopts this approach but uses two ARM processors on a Xilinx Zynq-7000 SoC FPGA instead of the Microblaze soft processors. This is also the case for the NoC emulator proposed by Drewes *et al.* [66]. While using processors (either soft or hard cores) makes the implementation easy, it significantly degrades the emulation speed. For instance, in [66], Drewes *et al.* report an average emulation speedup of around 39K cycles per second for an 8×8 NoC with the trace data collected from the full-system simulation of the PARSEC benchmark suite [73]. This is even slower than the speed of BookSim [5], one of the most widely used software-based NoC simulators, measured on a Core i7 4770 PC; the measurement results of the thesis shows that the average speed of BookSim when simulating an 8×8 NoC with the PARSEC traces is around 62.5K cycles per second. Another problem with the approach of using processors is that the emulation speed decreases even faster than software-based simulators with increasing the target NoC size. For instance, DuCNoC reports a maximum speedup of 265× over BookSim when emulating a 512-node NoC and the speedup is reduced to just 3× when the target NoC size is 8,196-node (these numbers are obtained with a synthetic workload; the emulation speed with trace-driven workloads has the similar trend). Therefore, to speed up the emulation of large-scale NoCs, a new approach is required.

This chapter adopts the time-division multiplexing (TDM) scheme introduced in Chapter 3 and proposes an effective architecture for speeding up trace-driven emulation of NoCs with up to thousands of nodes on FPGAs. The proposed architecture helps to virtually eliminate the negative effect of the off-chip memory on the emulation performance and allows the emulator to operate at a high frequency. The major contributions are as follows.

1. Most of the existing FPGA-based NoC emulators rely on soft or hard processors to support trace-driven emulation. Because of this, their emulation speeds are limited, especially when the target NoC is large. This thesis proposes an effective architecture which does not include processors for speeding up trace-driven emulation of NoCs with up to thousands of nodes.

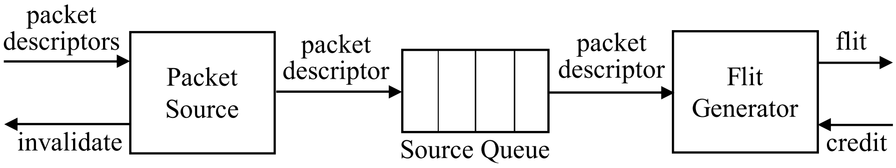


Figure 6.1: Traffic generator architecture for supporting trace-driven emulation.

2. The thesis introduces some methods to effectively hide the off-chip memory access time and improve the scalability of the emulation architecture in terms of operating frequency and FPGA resource requirements.

3. Using the proposed architecture, the thesis extends the NoC emulator proposed in Chapter 4 to support trace-driven emulation. The evaluation results show that the developed NoC emulator is 260× faster than BookSim when emulating an 8×8 NoC with the PARSEC traces, and the speedup is increased to 5,106× when emulating a 64×64 NoC with trace data created based on the uniform random traffic.

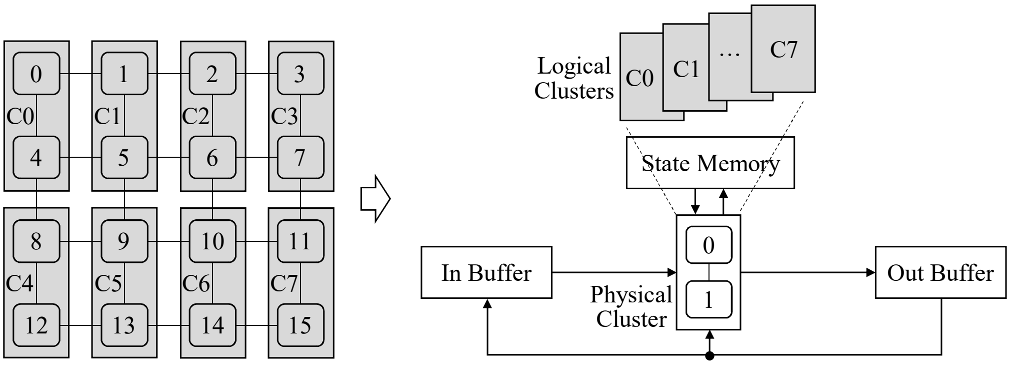
This chapter focuses on emulation of 2D meshes. However, the proposed architecture can be easily modified for emulation of k-ary n-trees as well.

6.2 Background

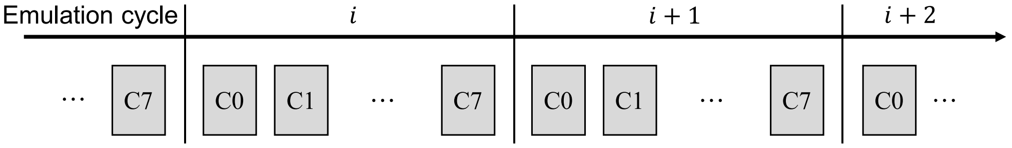
This section starts by describing the traffic generator architecture for supporting trace-driven emulation. After that, the TDM scheme is briefly reviewed because it is tightly coupled to the proposed architecture.

6.2.1 Traffic Generator Architecture

Figure 6.1 shows the traffic generator architecture for supporting trace-driven emulation. The *packet source* gets trace data from a module called *trace loader* and sends back a control signal, called *invalidate*, which will be described in detail in Section 6.3. In this thesis, a trace is defined as a set of *packet descriptors*, each encodes a message sent from one node to another in the many-core system from which the trace was recorded. In the implementation presented in Section 6.4, a packet descriptor is 64-bit length and composed of five fields: valid bit (1 bit), packet generation timestamp (30 bits), source address (14 bits), destination address (14 bits), and packet length (5 bits). These widths limit the maximum number of emulation cycles, the largest NoC size, and the maximum number of packet lengths to 2^{30}, 2^{14}, and 2^{5}, respectively. However, this is not a fundamental limitation; larger numbers of emulation cycles, NoC sizes, and numbers of packet lengths can be supported with only minor changes in the source code. The packet generation timestamp, source/destination addresses, and packet length are the information necessary



(a) The architecture for time-multiplexed emulation of 2D meshes.



(b) The emulation cycle is incremented after all logical clusters have been processed.

Figure 6.2: The TDM scheme for 2D meshes introduced in Chapter 3.

for carrying out the trace-driven emulation. Designers can add other information to the packet descriptor structure for collecting their desired performance characteristics of the emulated NoC.

When a valid packet descriptor arrives at a traffic generator, the packet source stores it into the *source queue*. When the network is ready, the *flit generator* reads a packet descriptor from the source queue and compares the encoded packet generation timestamp with the current time of the emulation to check whether it is time to generate a packet and inject it to the network. A packet can be injected into the network when its generation timestamp is smaller than the current time of the network. The flit generator tracks the status of the network based on the incoming flow control credits.

6.2.2 Time-Multiplexed Emulation

Figure 6.2 shows the TDM scheme for 2D meshes introduced in Chapter 3. A network is emulated using a small number of interconnected nodes called *physical cluster*. A node in the physical cluster is called a *physical node*. In each time slot, the physical cluster processes a part of the network which is called a *logical cluster*. The state of each logical cluster is stored in one entry of the *state memory*. The physical cluster switches from processing a logical cluster to processing another by changing its state with the state data loaded from the state memory. The *in buffer* and the *out buffer* are responsible for storing communication data between logical clusters.

Two FPGA cycles are used for processing each logical cluster. Let N_{phy} be the number of physical nodes; then emulating each cycle of a NoC with N_{node} nodes costs $2N_{node}/N_{phy}$ FPGA cycles. However, in the proposed architecture, because the emulation may be stalled when the speed of loading trace data from off-chip memory cannot keep pace with the emulation speed, emulating a cycle of an N_{node}-node NoC using a cluster of N_{phy} physical nodes may take more than $2N_{node}/N_{phy}$ FPGA cycles.

6.3 Proposed Trace-Driven Emulation Architecture

Before going into details of the proposed architecture, this section describes how a trace is organized on a host PC before it is sent to the FPGA side. As mentioned in Section 6.2.1, a trace is a set of packet descriptors. The order of packet descriptors in a trace is decided by first their source addresses and then their packet generation timestamps. Let p_s^t be the packet descriptor with source address s and packet generation timestamp t. For any two packet descriptors p_{s1}^{t1} and p_{s2}^{t2}, if $s1 < s2$ then p_{s2}^{t2} is behind p_{s1}^{t1}; in the case that the source addresses of the packet descriptors are the same ($s1 = s2$), they are sorted by the packet generation timestamps.

6.3.1 Architecture Overview

Figure 6.3 shows the overview of the proposed trace-driven emulation architecture on a Xilinx VC707 board in which the default 1GB DDR3 DRAM is replaced with a larger one (4GB DDR3 DRAM). The *trace receiver* is responsible for receiving packet descriptors from a host PC and writing them to the DRAM on the FPGA board. Packet descriptors with the same source address are stored in a continuous region of the DRAM and in the order of the packet generation timestamp. The emulation starts when the trace receiver finishes writing the whole trace data into the DRAM.

Since packet descriptor p_s^t describes a packet generated by node s, in the explanations below, we say that p_s^t belongs to s.

Each physical node in the physical cluster is connected to a *trace loader* which is responsible for loading packet descriptors belonging to the nodes processed by that physical node. For example, in Figure 6.2(a), because physical node 0 is responsible for processing nodes 0, 1, 2, 3, 8, 9, 10, and 11 of the network, trace loader 0 will load packet descriptors belonging to these nodes during the emulation. Section 6.3.2 will describe in detail how packet descriptors are loaded.

In the current implementation, the DRAM controller returns a block of 512 bits for each read request. Since the packet descriptor size is 64 bits, a trace loader will get eight packet descriptors after issuing a read request to the DRAM controller. Asynchronous FIFOs and the double flopping technique are used for passing data between different clock domains [135].

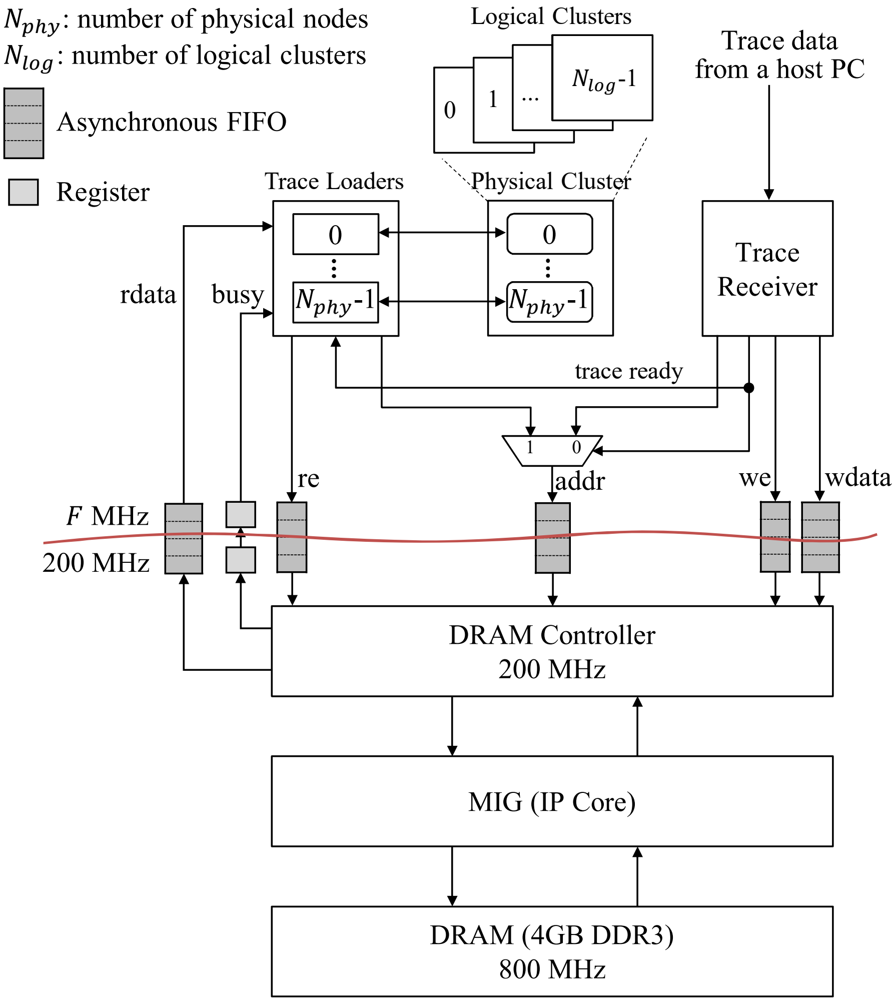


Figure 6.3: Overview of the proposed trace-driven emulation architecture on a Xilinx VC707 board in which the default 1GB DDR3 DRAM is replaced with a larger one (4GB DDR3 DRAM).

6.3.2 Trace Loader Architecture

Figure 6.4 shows the datapath surrounding the trace loaders. In a trace loader, there are four memory modules: *data, offset, valid*, and *reqstatus*. Each of these memories consists of N_{log} entries where N_{log} is the number of logical clusters, each entry for a node processed by the physical node which the trace data loader is connected to. For example, in Figure 6.2(a), we have two physical nodes 0 and 1. Since physical node 1 processes nodes 4, 5, 6, 7, 12, 13, 14, and 15, trace loader 1 is responsible for loading packet descriptors belonging to these nodes. In trace loader 1, *data*[0], *offset*[0], *valid*[0], and *reqstatus*[0] are for node 4; *data*[1], *offset*[1], *valid*[1], and *reqstatus*[1] are for node 5; and so on.

Data: 512-bit-wide memory. *Data* stores packet descriptors loaded from the DRAM.

Offset: x-bit-wide memory where x depends on the size of the trace. *Offset*$[i]$ stores a relative

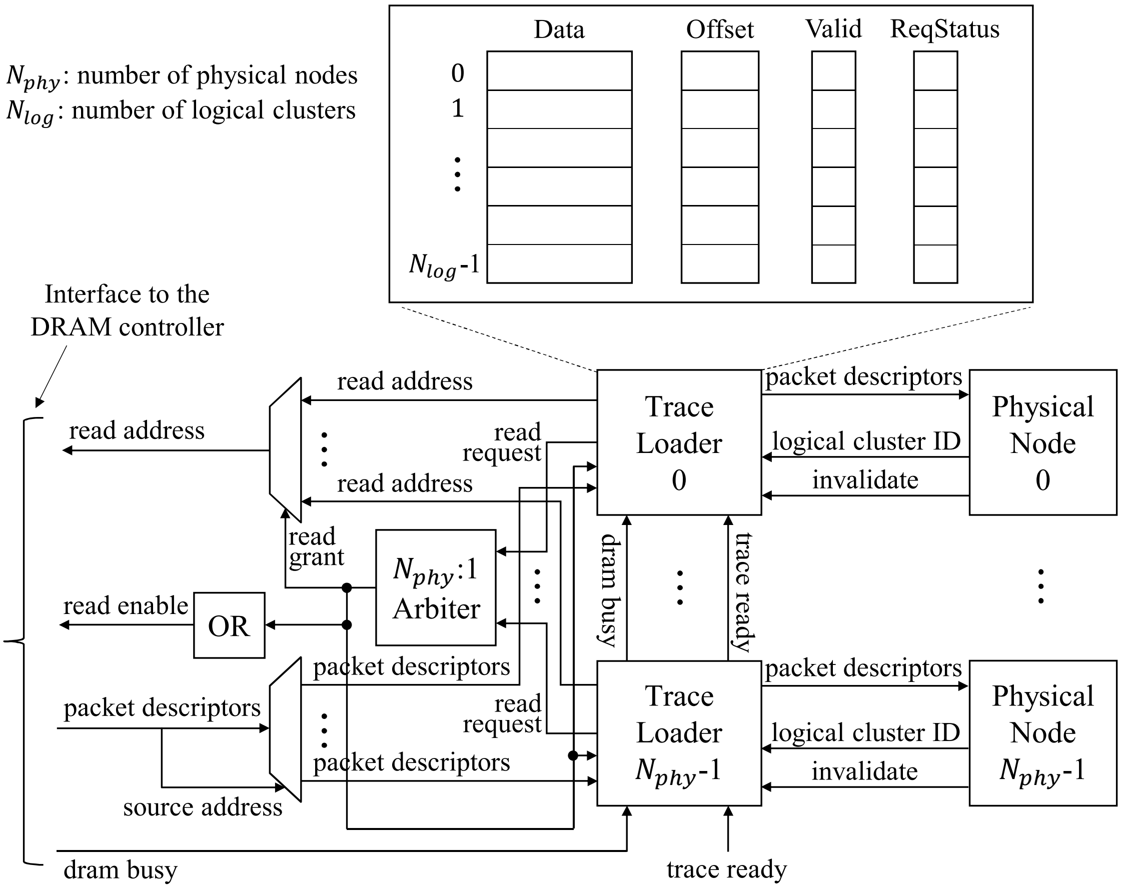


Figure 6.4: Datapath surrounding the trace loaders.

address which is used to calculate the address for loading the next *data*$[i]$; the base address is determined by the ID of the node that *data*$[i]$, *offset*$[i]$, *valid*$[i]$, and *reqstatus*$[i]$ are for.

Valid: 1-bit-wide memory. *Valid*$[i]$ indicates whether *data*$[i]$ is valid or not. *Valid*$[i]$ is initialized with 0 and set to 1 each time *data*$[i]$ is filled. For the simplicity of description, below, we assume that *data*$[i]$, *offset*$[i]$, *valid*$[i]$, and *reqstatus*$[i]$ are for node s. In each emulation cycle, if valid$[i]$ is 1 and the source queue of the traffic generator in node s is not full, one packet descriptor will be extracted from *data*$[i]$ and put into the source queue by the packet source (Figure 6.1). In the packet source, a counter is maintained to track the number of packet descriptors that have been extracted from *data*$[i]$. When the packet source extracts the eighth packet descriptor (the last one) from *data*$[i]$, this counter is reset. At the same time, an invalidate request is issued to the trace loader to reset *valid*$[i]$ to zero.

ReqStatus: 1-bit-wide memory. A trace loader is not blocked after issuing a read request to the DRAM controller. Instead of waiting for data from the DRAM controller, the trace loader continues to issue more read requests if necessary and possible. In this way, the impact of the DRAM access time on the emulation performance is effectively reduced. *ReqStatus* is used to

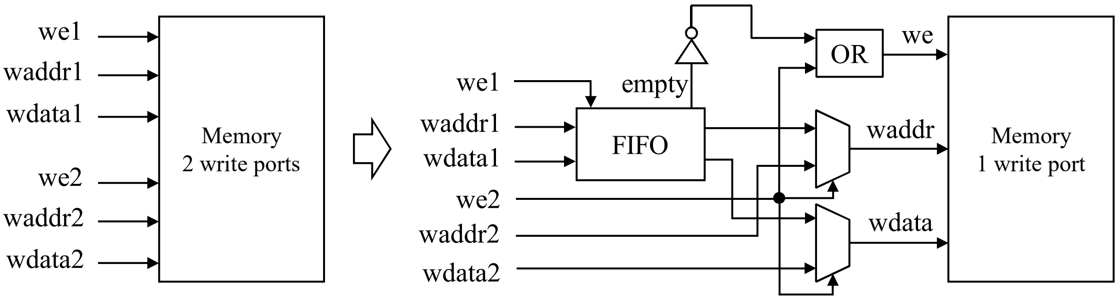


Figure 6.5: The idea for reducing the number of write ports of the *valid* memory and the *reqstatus* from two to one.

make sure that there are no duplicate read requests. *ReqStatus*$[i]$ indicates whether a read request for *data*$[i]$ has been issued to the DRAM controller. *ReqStatus*$[i]$ is set to 1 when the DRAM controller accepts the read request for *data*$[i]$ and reset to 0 when *data*$[i]$ is filled. The trace loader issues a read request for *data*$[i]$ when both *valid*$[i]$ and *reqstatus*$[i]$ are zero.

Both the *data* memory and the *offset* memory have only one write port and one read port and can be efficiently implemented using FPGA on-chip block RAMs (BRAMs). However, this is not the case for the *valid* memory and the *reqstatus* memory.

As described above, the *valid* memory has two write ports: one for the DRAM controller side (update requests – active when new packet descriptors arrive) and the other for the physical node side (invalidate requests). It also has two read ports: one for determining whether a read request should be issued and the other is for determining the status of the packet descriptors sent to the physical node.

The *reqstatus* memory has two write ports and one read port. One of the write port is for updating when the DRAM controller accepts the read request of the trace loader. The other write port is for updating when new packet descriptors arrive. The read port is for determining whether a read request should be issued to the DRAM controller.

Since *valid* and *reqstatus* are 1-bit-wide memories, it seems that having multiple write ports and read ports is not a serious problem. However, the experimental results show that the FPGA resource requirements and the peak operating frequency are significantly affected when their depth is high, that is, the number of logical clusters N_{log} is large. This makes it difficult to support emulation of large-scale NoCs.

The thesis proposes a method to make it possible to implement the *valid* memory and the *reqstatus* memory efficiently. The proposed method is based on the following observations. In the *valid* memory, it is not necessary to immediately process invalidate requests from the physical node. An invalidate request can be stalled for $2N_{log}$ FPGA cycles, which is the time needed to emulate one cycle of the network. When we stall an invalidate request and all subsequent ones (if there are any), there are at most $N_{log} - 1$ update requests from the DRAM controller side. This is because a read request for *data*$[i]$ is not issued until *valid*$[i]$ is invalidated (reset to zero). Therefore, we can reduce the number of write ports of the *valid* memory from two to one using the idea illustrated in Figure 6.5. We store invalidate requests from the physical node side in an N_{log}-entry FIFO. Update requests from the DRAM controller side are always serviced immediately. When there is no update request, an invalidate request is popped from the FIFO and serviced. Similarly, we can also reduce the number of write ports of the *reqstatus* memory to one. Since the *valid* memory is only 1-bit-wide, we simply duplicate it to reduce the number of read ports from two to one. Consequently, all memories in the proposed architecture have only one write port and one read port, and therefore, they can be implemented efficiently on FPGAs.

As shown in Figure 6.4, the read requests of the trace loaders are arbitrated by an N_{phy}:1 arbiter where N_{phy} is the number of physical nodes of the physical cluster used to emulate the network. Packet descriptors provided by the DRAM controller are sent to the appropriate trace loaders based on the source addresses encoded inside them.

6.3.3 Emulation Methodology

Similar to in emulation with synthetic workloads, the time of the network is separated from the times of the traffic generators. Each traffic generator, as well as the network, has its own time counter.

The time counter of a traffic generator is updated each time the traffic generator gets a packet descriptor from the corresponding trace loader and puts it into the source queue. The value used to update the time counter is the packet generation timestamp of the packet descriptor. For the correctness, the emulation of the network is stalled when both of the following two conditions hold: (1) the source queue is empty, and (2) the time counter of the traffic generator is smaller than or equal to the time counter of the network. This occurs when the DRAM access speed is too slow compared to the emulation speed.

With the above approach, the traffic generators are allowed to run ahead of the network. This helps to minimize the impact of stalling the network on the emulation performance (that is, hiding the DRAM access time). However, the source queues may contain packet descriptors with packet generation timestamps larger than the current time counter of the network. To prevent incorrect emulation, some logic is added to make sure that a packet descriptor is dequeued from the source queue in which it is stored if and only if its packet generation timestamp is smaller than the current time counter of the network.

One problem that might arise when the above emulation method is used is that the emulation might not be finished successfully if the largest packet generation timestamp in a node is smaller than the time to which we want to emulate. For example, assume that we want to emulate a 10,000-cycle trace and the largest packet generation timestamp in node 0 is 7,000. At emulation

Table 6.1: Parameters of the target NoCs

Router architecture	Input-queued VC router
Router pipeline	5-stage
Routing algorithm	XY, odd-even
# of VCs per port	2, 4
VC size	4-flit
Flit size	30-bit, 31-bit, 32-bit
VC/Switch allocator	iSLIP [122]
Arbiter type	Round-robin
Flow control	Wormhole & credit-based

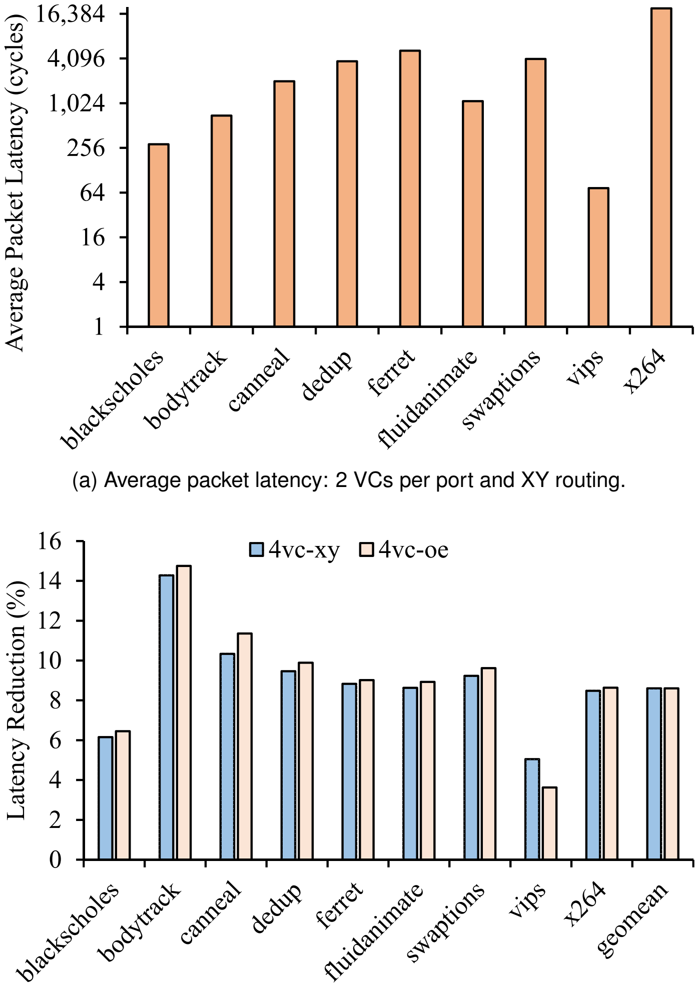
cycle t ($t > 7,000$) when the source queue in node 0 becomes empty, according to the emulation method explained above, the network is stalled since the time counter of the traffic generator in node 0 is 7,000 and smaller than t. If we do not provide any valid packet descriptors with packet generation timestamp larger than t, the network will be stalled forever. To deal with this problem, some dummy packet descriptors with the largest possible packet generation timestamp are appended to the end of the trace of each node. The number of appended dummy packet descriptors is set to $L + 1$ where L is the source queue length to prevent the corresponding trace loader from issuing unnecessary read requests to the DRAM controller.

6.4 Evaluation

Using the proposed architecture, the thesis extends FNoC, the NoC emulator proposed in Chapter 4, to support trace-driven emulation. As mentioned in Section, the default 1GB DDR3 DRAM on the used Xilinx VC707 board is replaced with a larger one with a capacity of 4GB. Vivado 2017.4 is used for synthesizing, implementing, and generating FPGA bitstream files.

Table 6.1 shows the parameters of the NoCs evaluated in this chapter. Two routing algorithms are implemented: the dimension-order routing algorithm (denoted by *XY*) and a minimal adaptive routing algorithm based on the odd-even turn model (denoted by *odd-even*) [72]. In the odd-even routing algorithm, when there are two available output ports, the port with more free VCs is selected. The flit size depends on the routing algorithm used and the number of VCs per port. When the routing algorithm and the number of VCs per port are XY and two, respectively, the flit size is 30-bit. Increasing the number of VCs per port to four requires a flit size of 31-bit. Changing the routing algorithm from XY to odd-even adds one more bit to the flit structure.

The trace data of the PARSEC benchmark suite [73] collected by Hestness *et al.* [74] are used for evaluating the implemented 8×8 NoC. In these trace data, the packet length varies between 2-flit and 18-flit. In each trace, the focus is on the region which represents the parallel portion of the application. To evaluate NoCs larger than 8×8, trace data of synthetic workloads are created.



(a) Average packet latency: 2 VCs per port and XY routing.

(b) Latency reduction when increasing the number of VCs per port from 2 to 4 (4vc-xy) and using the odd-even routing algorithm (4vc-oe).

Figure 6.6: Results obtained when emulating an 8×8 NoC (the parameters are shown in Table 6.1) with the PARSEC traces.

6.4.1 Emulation Accuracy

The emulation accuracy of FNoC in trace-driven emulation is verified by running extensive emulations with the PARSEC traces and various traces created based on synthetic workloads. The evaluation results show that, like in the case of emulation with synthetic workloads, FNoC and BookSim report exactly the same results in every case. This indicates that the NoC models of FNoC are totally identical to those of BookSim.

Figure 6.6(a) shows the average packet latencies obtained when emulating an 8×8 NoC (the parameters are shown in Table 6.1; the routing algorithm, number of VCs per port, and flit size are XY, 2, and 30-bit, respectively) with the PARSEC traces. For each trace, the latency data are collected over 20 million warm-up cycles and 20 million measurement cycles. We can see that applications like *x264*, *ferret*, and *dedup* exhibit high average packet latencies. This is because

Table 6.2: FPGA resource requirements and operating frequencies when emulating different NoC sizes using a cluster of 16 nodes (4×4)

NoC	LUTs		Regs		Slices		BRAMs		Freq.
	#	%	#	%	#	%	#	%	[MHz]
8×8	65,913	21.71%	53,551	8.82%	22,233	29.29%	442	42.91%	130
16×16	66,377	21.86%	54,073	8.91%	22,247	29.31%	442	42.91%	130
32×32	66,624	21.94%	54,298	8.94%	21,993	28.98%	442	42.91%	120
64×64	69,020	22.73%	54,736	9.01%	23,837	31.41%	562	54.56%	120
128×64	70,847	23.34%	55,053	9.07%	23,828	31.39%	690	66.99%	120

they have a high degree of data sharing and data exchange during the executions.

Figure 6.6(b) shows how the average packet latency can be reduced when increasing the number of VCs per port from 2 to 4 (4vc-xy) and using the odd-even routing algorithm (4vc-oe). The figure shows that increasing the number of VCs helps to reduce an average of 8.61% packet latency. However, changing the routing algorithm from XY to odd-even has only a slight impact.

6.4.2 Resource Requirements and Scalability

Table 6.2 shows the FPGA resource requirements and operating frequencies when emulating different NoC sizes using a physical cluster of 16 nodes (4×4). The 8×8 NoC here is the one evaluated in Figure 6.6(a). The other NoCs have the same parameters as the 8×8 NoC except for the network size.

We can see that the numbers of required LUTs and registers are almost the same in every case. Emulating a larger NoC only requires more BRAMs. The size of each BRAM is fixed. When the emulated NoC is small, many occupied BRAMs are underutilized. This is the reason why the number of required BRAMs does not change when increasing the NoC size from 8×8 to 32×32.

The above result indicates that the scalability of the proposed architecture depends on only the total capacity of BRAMs on the FPGA used. Larger NoCs can be supported by using an FPGA with more BRAMs. This result is similar to the result in the case of emulation with synthetic workloads presented in Chapter 4.

As shown in Table 6.2, the proposed architecture can operate at very high frequencies, 130 MHz when the emulated NoC size is 8×8 and 16×16 and 120 MHz in the other cases. This helps to improve the emulation performance.

Next, let us evaluate the impact of the method for efficiently implementing the *valid* and *reqstatus* memories described in Section 6.3.2 on the overall FPGA resource requirement and operating frequency. Figure 6.7 shows that, when the emulated NoC is small, there is almost

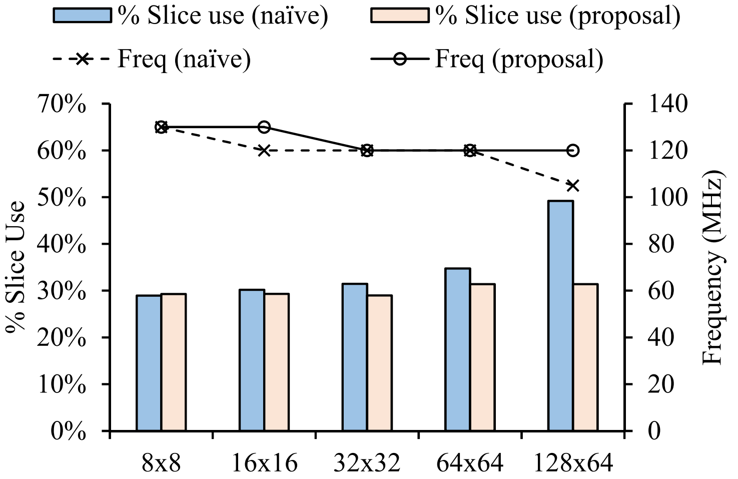


Figure 6.7: Impact of the method for efficiently implementing the *valid* and *reqstatus* memories on the overall FPGA resource requirement and operating frequency.

no difference between using and not using the proposed method. However, as the NoC size increases, the effectiveness of the proposed method is clearly established. When emulating the 128×64 NoC, the naïve approach of using the *valid* and *reqstatus* memories with multiple write ports and read ports requires 49.2% of FPGA slices. In contrast, the proposed method requires only 31.4% FPGA slices, which is almost the same as when emulating the 8×8 NoC. The proposed method also helps to improve the operating frequency.

6.4.3 Emulation Performance

To evaluate the performance of FNoC in trace-driven emulation, this section first proposes a performance model. Let N_{node}, N_{phy}, and N_{log} be the number of nodes of the emulated NoC, the physical cluster size (the number of physical nodes), and the number of logical clusters, respectively; then $N_{node} = N_{phy} \times N_{log}$. We define α $(0 < \alpha \leq 1)$ as a coefficient reflecting the impact of DRAM on the emulation speed. $\alpha = 1$ means that the DRAM access time is completely hidden. α is smaller than 1 when the network is stalled during the emulation as described in Section 6.3.3. Since two FPGA cycles are used for emulating each logical cluster, the emulation speed S (emulation cycles per second) of FNoC is given by

$$S = \alpha \times \frac{F}{2 \times N_{log}} = \alpha \times \frac{F}{2 \times \frac{N_{node}}{N_{phy}}}, \tag{6.1}$$

where F is the operating frequency (Hz).

Figure 6.8 shows the speed of FNoC when emulating the 8×8 NoC which is evaluated in Figure 6.6(a) with the PARSEC traces. The physical cluster size used here is 4×4, and the emulator operates at a frequency of 130 MHz. Like in the case of emulation with synthetic workloads,

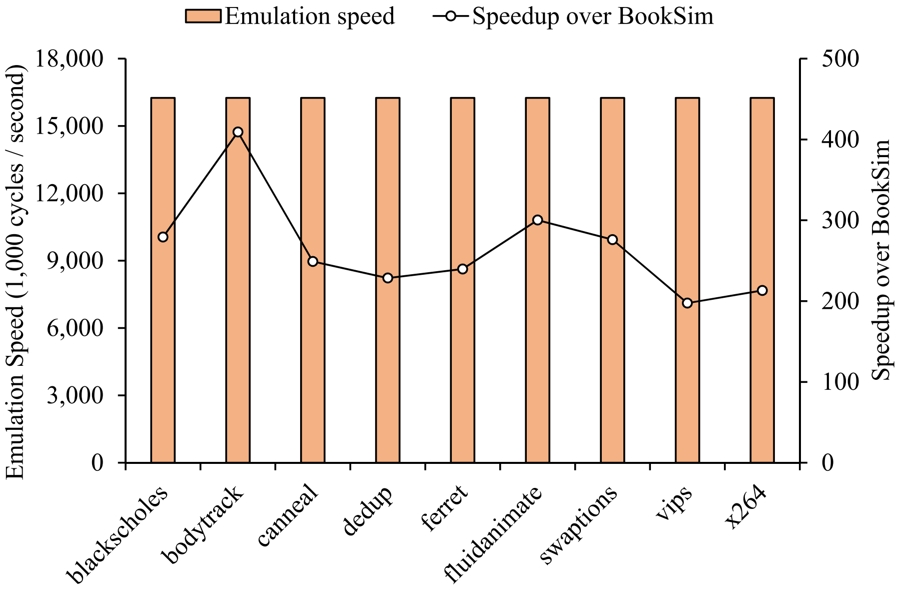


Figure 6.8: Speed of FNoC when emulating the 8×8 NoC which is evaluated in Figure 6.6(a) with the PARSEC traces. The physical cluster size used is 4×4.

a comparison with BookSim is made. In this comparison, BookSim is also run on a Core i7 4770 PC. The results show that the speed of FNoC is almost constant at around 16,250K emulation cycles per second in every case. In contrast, BookSim's speed varies with the applications. This is because each application features a different traffic load and BookSim's speed decreases with increasing traffic load. Consequently, the speedup of FNoC over BookSim varies with the applications. The average (geomean) speedup is 260×.

To the extent of the author's knowledge, at the moment, there does not exist any traces of realistic applications for emulation of large-scale NoCs with thousands of nodes. Since creating such traces is not trivial and extremely time-consuming, the thesis adopts the approach of using traces created based on synthetic workloads to estimate the speed of FNoC when emulating large-scale NoCs.

Figure 6.9 shows the speed of FNoC when emulating the 64×64 NoC which has the same parameters as the 8×8 NoC evaluated in Figure 6.6(a) with traces created based on the uniform random traffic under different loads. The physical cluster size used here is 8×4, and the emulator operates at a frequency of 105 MHz. Similar to the evaluation result of the 8×8 NoC above, the speed of FNoC is almost constant at around 410K emulation cycles per second in every case. The speedup over BookSim ranges from 2,235× to 9,005× with the average (geomean) of 5,106×.

Finally, let us discuss the impact of DRAM on the emulation speed, that is, the coefficient α in formula (6.1). There are two factors that influence α. The first factor is the difference between the DRAM read bandwidth required for an ideal emulation speed (B_{req}) and the maximum read bandwidth that can be achieved with the proposed emulation architecture (B_{max}). The second

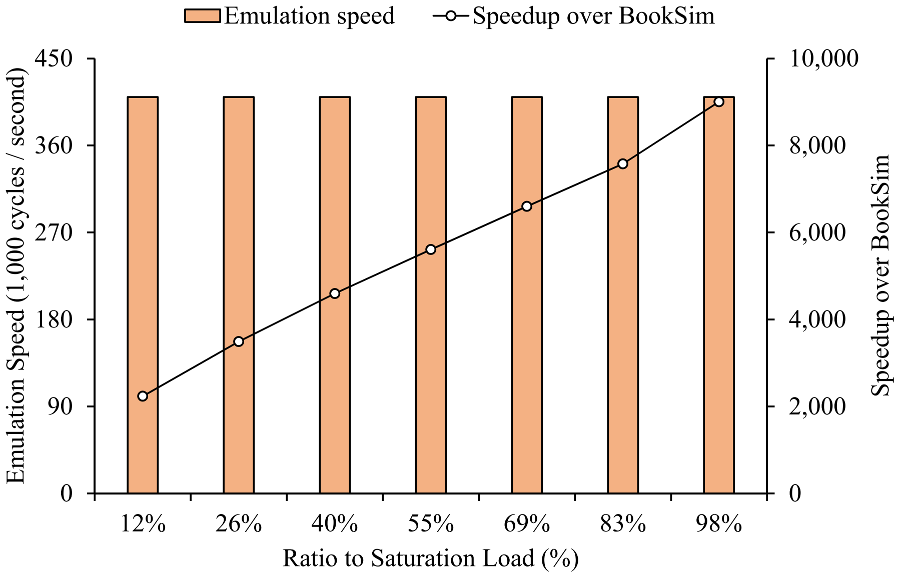


Figure 6.9: Speed of FNoC when emulating the 64×64 NoC which has the same parameters as the 8×8 NoC evaluated in Figure 6.6(a) with traces created based on the uniform random traffic. The physical cluster size used is 8×4.

factor is how well the DRAM read latency can be hidden. α is equal to 1 if B_{req} is not greater than B_{max} and the DRAM read latency can be completely hidden. Otherwise, α will be smaller than 1.

The thesis provides a formula for calculating the required DRAM read bandwidth B_{req}. Let l (flits/node/cycle), p (flits), and d (bits) be the traffic load accepted by the emulated NoC under the offered trace-driven workload, the average packet length, and the size of each packet descriptor, respectively. As same as in formula (6.1), N_{node}, N_{phy}, and F (Hz) are the number of nodes of the NoC, the physical cluster size (the number of physical nodes), and the operating frequency of the emulator, respectively. N_{phy} is also the number of trace loaders. In each emulation cycle, the NoC accepts $(l \times N_{node})/p$ packets; thus, $(l \times N_{node})/p$ packet descriptors, that is, $(l \times N_{node} \times d)/p$ bits need to be loaded from the DRAM. Since each emulation cycle is equivalent to $2 \times N_{node}/N_{phy}$ FPGA cycles in the ideal case and each FPGA cycle is equivalent to $1/F$ seconds, the required DRAM read bandwidth B_{req} can be calculated by

$$\begin{aligned} B_{req} &= \frac{\frac{l \times N_{node} \times d}{p}}{2 \times \frac{N_{node}}{N_{phy}} \times \frac{1}{F}} \\ &= \frac{l \times d \times N_{phy} \times F}{2p} \quad \text{(bps).} \end{aligned} \tag{6.2}$$

We can see that B_{req} is proportional to the physical cluster size N_{phy} and the operating frequency F of the emulator. A larger physical cluster and a higher operating frequency will make B_{req}

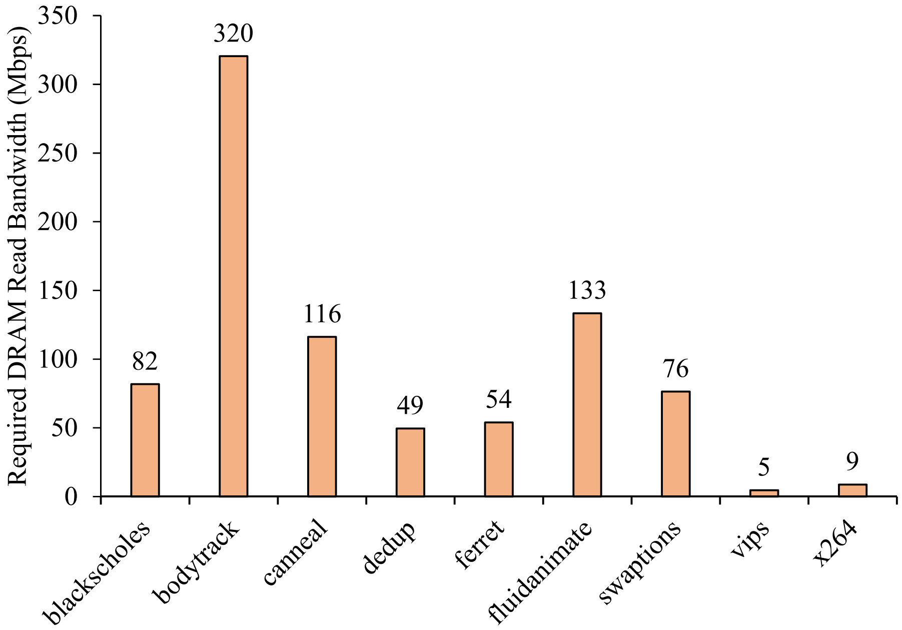


Figure 6.10: The DRAM read bandwidth required for an ideal speed (B_{req}) when emulating the 8×8 NoC which is evaluated in Figure 6.6(a) with the PARSEC traces. The physical cluster size used is 4×4.

increase, and thus more pressure is put on the DRAM.

Figures 6.10 and 6.11 show B_{req} in the cases of emulating the 8×8 and 64×64 NoCs that are used to evaluate FNoC's emulation speed above. Here, B_{req} is measured according to formula (6.2). In Figure 6.10, B_{req} varies from 5 Mbps to 320 Mbps. The analysis shows that the variation is because each application exhibits a different traffic load l and an average packet size p. In Figure 6.11, the packet size is fixed and thus the variation of B_{req} (from 54 Mbps to 441 Mbps) is caused by only the changes in the traffic load l.

The measurement results show that the sequential and random read bandwidths of the currently used DRAM are around 85,539 Mbps and 24,889 Mbps, respectively. Thus, the maximum read bandwidth that can be achieved with the proposed emulation architecture B_{max} is estimated to be much larger than the required bandwidths shown in Figures 6.10 and 6.11. Therefore, the DRAM can provide sufficient bandwidth for the emulation of the 8×8 and 64×64 NoCs with the current parameters.

Figure 6.12 shows the estimation of the required DRAM read bandwidth B_{req} when emulating the 64×64 NoC mentioned in Figure 6.11 with different physical cluster sizes. Here, the traffic load l is 98$\%$ of the saturation load. The packet descriptor size d, the operating frequency F, and the average packet length p are assumed to be fixed when the physical cluster size N_{phy} is changed. The estimation is performed using formula (6.2). We can see that the required DRAM read bandwidth B_{req} becomes larger than the DRAM random read bandwidth when the physical

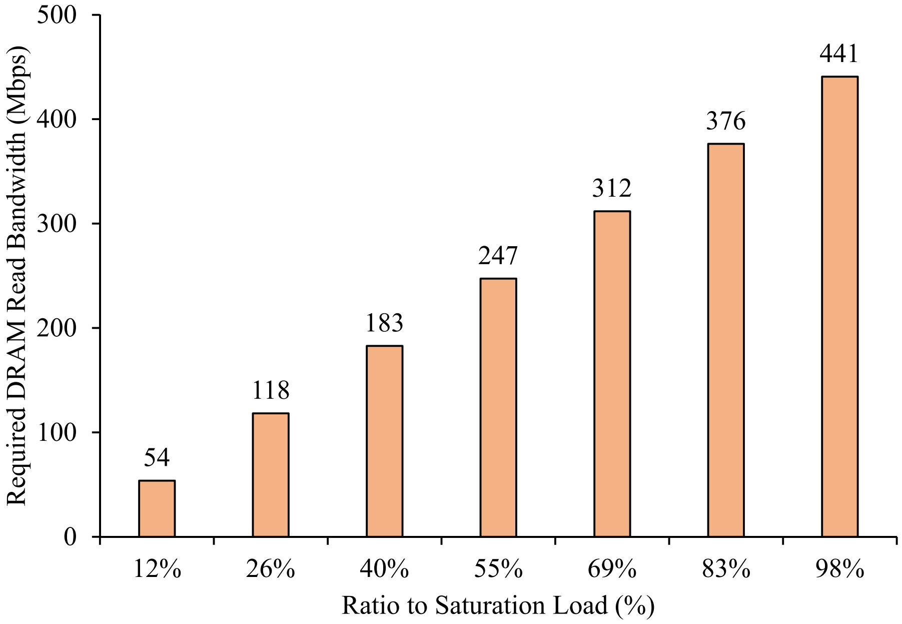


Figure 6.11: The DRAM read bandwidth required for an ideal speed (B_{req}) when emulating the 64×64 NoC which has the same parameters as the 8×8 NoC evaluated in Figure 6.6(a) with traces created based on the uniform random traffic. The physical cluster size used is 8×4.

cluster size N_{phy} is equal to 2,048.

As mentioned above, apart from the DRAM read bandwidth, the coefficient α also depends on the DRAM read latency. In the proposed emulation architecture, there are three parameters that may affect the DRAM read latency: the packet descriptor size, the bit-width of the *data* memories in the trace loaders, and the length of the source queues in the traffic generators. In all evaluations in this chapter, these parameters are set to 64 bits, 512 bits, and 2-entry, respectively. The results show that, when emulating the 8×8 NoC, we always have $\alpha > 0.9999$. Similarly, when emulating the 64×64 NoC, α is always greater than 0.9997. Thanks to the effective emulation architecture and methods proposed in Section 6.3, the negative impact of the DRAM read latency is virtually eliminated.

6.4.4 Comparison with other FPGA-Based NoC Emulators

Among all of the FPGA-based NoC emulators mentioned in Section 2.4.2 in Chapter 2 that support trace-driven emulation, only DuCNoC [68] reports results obtained when emulating a NoC with traces of realistic applications and provides a comparison with a well-known simulator which is also BookSim. The comparison results show that DuCNoC tracks BookSim closely. However, the differences are not zero. In contrast, the results reported by FNoC are totally identical to those reported by BookSim.

The thesis next compares the speed of FNoC with those of DART [65], AdapNoC [67], DuC-

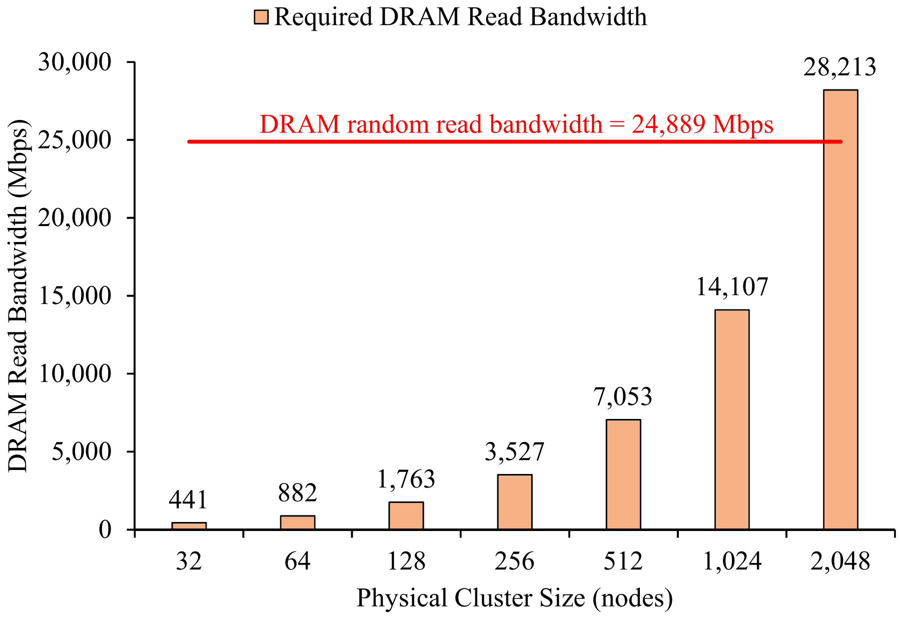


Figure 6.12: Estimation of the required DRAM read bandwidth B_{req} when emulating the 64×64 NoC mentioned in Figure 6.11 with different physical cluster sizes. Here, the traffic load l is 98% of the saturation load. The packet descriptor size d, the operating frequency F, and the average packet length p are assumed to be fixed when the physical cluster size N_{phy} is changed. The estimation is performed using formula (6.2).

NoC [68], and the NoC emulator proposed by Drewes *et al.* [66]. These are the most recently proposed FPGA-based NoC emulators that support trace-driven emulation. The comparison here is not strictly quantitative but qualitative because of the following reasons. First, the router architectures modeled by the emulators are not the same. Second, the emulators are implemented on different FPGAs. Third, although DART, AdapNoC, and DuCNoC support trace-driven emulation, they only report the speeds of synthetic workload emulations.

When the emulated NoC size is 8×8, the emulation speed ranges of DART and AdapNoC are around 5,500K–16,000K and 30K–200K emulation cycles per second, respectively. Although the largest NoC sizes that can be emulated by these two emulators are 9×9 and 32×32, respectively, their authors do not provide any results for NoCs larger than 8×8. DuCNoC's speed for a 5×5 NoC varies from around 200K to around 375K emulation cycles per second. For larger NoCs, the authors of DuCNoC do not report the absolute emulation speeds but instead the speedups compared to BookSim. As mentioned in Section 6.1, DuCNoC's speed decreases faster than BookSim's speed with increasing the target NoC size. DuCNoC achieves a maximum speedup of 265× over BookSim when emulating a 512-node NoC but the speedup is reduced to just 3× when the target NoC size is 8,196-node. Different from DART, AdapNoC, and DuCNoC, Drewes *et al.* report the speed of their emulator in the case of emulation with trace-driven workloads, which is around 39K emulation cycles per second for an 8×8 NoC with the PARSEC traces. As presented in Section 6.4.3, the speed of FNoC when emulating an 8×8 NoC with the PARSEC traces is 16,250K emulation cycles per second. When the NoC size is increased to 64×64, the speed is reduced to 410K emulation cycles per second. Contrary to DuCNoC, FNoC's speed decreases much slower than BookSim's speed with increasing the target NoC size; the speedup over BookSim when emulating the 64×64 NoC is 5,106×.

6.5 Summary

This chapter proposed an effective architecture for speeding up trace-driven emulation of NoCs with up to thousands of nodes on FPGAs. The chapter introduced some methods to effectively hide the off-chip memory access time and improve the scalability of the emulation architecture in terms of operating frequency and FPGA resource requirements. The developed NoC emulator achieved a speedup of 260× compared to BookSim, a widely used NoC simulator, when emulating an 8×8 NoC with the PARSEC traces. The speedup was increased to 5,106× when emulating a 64×64 NoC with trace data created based on a synthetic workload.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

NoCs have become integral parts of modern many-core processors, MPSoCs, and many hardware accelerators of intrinsically important applications. Research and development of NoCs play a key role in designing future large-scale architectures with hundreds to thousands of components that need to be interconnected. However, a major obstacle to research and development of large-scale NoCs is the lack of fast modeling methodologies that can provide a high degree of accuracy. The FPGA-based emulation approach has been shown promising but scaling to large-scale NoCs is hard due to the FPGA logic and memory constraints. This dissertation proposed methods and architectures to address this problem, thereby enabling fast and accurate emulation of NoCs with up to thousands of nodes. Specifically, the dissertation has made the following contributions.

In Chapter 3, the dissertation proposed a novel use of time-division multiplexing where the emulation cycle was decoupled from the FPGA cycle and a network was emulated by time-multiplexing a small number of nodes. This approach helps to overcome the FPGA logic constraints, thereby enabling emulation of large-scale NoCs with hundreds to thousands of nodes using a single FPGA. The dissertation discussed in detail methods for efficiently applying the time-division multiplexing technique for both direct and indirect network topologies with the focus on 2D meshes and fat-trees (k-ary n-trees). Because these are the bases of almost all actually constructed network topologies, it can be expected that the proposed methods can be extended for a wide range of networks. While the proposed time-division multiplexing methods enable the emulation of large-scale NoCs, they alone are not sufficient. To achieve a high emulation speed, it is essential to address the memory constraints caused by modeling traffic workloads.

In Chapter 4, the dissertation addressed the memory constraints caused by modeling synthetic workloads that are presently the only feasible workloads for emulating large-scale NoCs with thousands of nodes due to the lack of trace data of large-scale NoC-based systems. Synthetic workloads are created based on mathematical modeling of common traffic patterns in real applications and have been used to analyze NoCs in a wide range of situations. A set of carefully designed synthetic workloads can provide a relatively thorough coverage of the characteristics of the target NoCs. It has also been shown that evaluation on synthetic workloads is indispensable in many cases. For instance, when designing a routing algorithm, the use of synthetic workloads is mandatory for assessing the algorithm on possible corner cases like those under extremely high loads. However, existing methods for modeling synthetic workloads require a large amount of memory. The dissertation proposed a method to reduce the amount of required memory so that it was not necessary to use off-chip memory even when emulating NoCs with thousands of nodes. This method not only makes the overall design much simpler but also significantly contributes to the improvement of emulation speed since accessing on-chip memory is much faster than off-chip memory.

Also in Chapter 4, the dissertation developed a NoC emulator, called FNoC, on a Xilinx VC707 FPGA board using the proposed time-multiplexed emulation methods and the method for modeling of synthetic workloads. The evaluation results showed that (1) the size of the largest NoC that could be emulated by FNoC depended on only the on-chip memory capacity of the FPGA used; the largest NoCs that could be emulated by FNoC on the currently used FPGA were a mesh-based NoC with 16,384 nodes (128×128 NoC) and a fat-tree-based NoC with 6,144 switch nodes and 4,096 terminal nodes (4-ary 6-tree NoC); (2) When emulating a 128×128 NoC and a 4-ary 6-tree NoC under a synthetic workload, FNoC was, respectively, 5,047× and 232× faster than BookSim, one of the most widely used software-based NoC simulators, while providing the same results.

In Chapter 5, the dissertation showed the usability of FNoC by designing and modeling an effective routing algorithm, called LEF, for 2D mesh NoCs and evaluating it for various network sizes, from currently popular middle-scale sizes to future large-scale sizes. LEF has an oblivious routing scheme and thus a low design complexity. It, however, can achieve high performance by properly distributing the load over two network dimensions and using an efficient deadlock avoidance method. FNoC enabled the evaluation of LEF and some other routing algorithms for comparison in large-scale NoCs with thousands of nodes in a practical time. The evaluation results showed that, in an 8×8 NoC, LEF provided 3.6%–10% higher throughput than O1TURN, one of the best oblivious routing algorithms for 2D meshes known so far, and comparable performance to a relatively complex adaptive routing algorithm under three different traffic patterns. LEF is particularly effective when the network is asymmetric. In a 16×8 NoC, LEF outperformed the adaptive routing algorithm and delivered up to 28.3% higher throughput than O1TURN. However, as the NoC size increased, the performance of the routing algorithms was strongly affected by the resource allocation policy in the network and the effects were different for each algorithm. This result would not be obtained if modeling of large-scale NoCs could not be performed.

In Chapter 6, the dissertation extended FNoC to support trace-driven emulation which will be useful for research and development of large-scale NoCs in the future when trace data of large-scale NoC-based systems are available. While synthetic workloads can provide a relatively thorough coverage of the characteristics of the target NoCs, evaluation on trace-driven workloads is still required in some cases such as assessing some application-specific optimizations. In trace-driven NoC emulation, trace data are often much larger than the total capacity of FPGA on-chip memory and thus must be stored in off-chip memory. The dissertation proposed an effective trace data loading architecture and some methods to hide the off-chip memory access time and improve the scalability of the emulation architecture in terms of operating frequency and FPGA resource requirements. These proposals are tightly coupled to the time-multiplexed emulation methods introduced in Chapter 3. The evaluation results showed that the extended NoC emulator achieved a speedup of 260× compared to BookSim when emulating an 8×8 NoC with the PARSEC traces while also providing the same results; and the speedup was increased to 5,106× when emulating a 64×64 NoC with trace data created based on a synthetic workload.

The work in this dissertation contributes directly to the formation of infrastructures for research and development of large-scale NoCs, which is crucial for developing more powerful and efficient many-core processors, MPSoCs, and hardware accelerators in the future.

7.2 Future Work

The focus of this dissertation is on proposing effective methods and architectures for emulating NoCs with up to thousands of nodes on FPGAs. The dissertation does not aim at proposing a complete NoC emulation environment. For such purpose, some extensions, such as adding the ability to change some important parameters (e.g., the number of VCs per port, VC size) without re-synthesizing the design, are required.

The time-multiplexed emulation methods described in this dissertation are limited to homogeneous NoCs where all routers have the same hardware components. For heterogeneous NoCs, the methods need to be modified. Fortunately, a heterogeneous NoC is often composed of only several types of routers, and the routers of the same type are often positioned according to a specific pattern. Thus, by time-multiplexing on at least one router for each type, we can emulate the behavior of the entire network.

Besides the advantages of accuracy and speed, the author believes that the FPGA-based approach has the following limitations that need to be tackled. First, developing and debugging FPGA-based systems are generally difficult. Second, since the FPGA synthesis time is long, methods for avoiding re-synthesizing as much as possible are required. Third, observing intermediate states of FPGA-based emulators in detail is technically possible but requires more effort to implement than in the case of using software simulators.

While emulations with synthetic workloads and trace-driven workloads have their unique roles in the design cycle and are effective in assisting architects to make their design decisions, execution-driven emulation is still required in many cases. Therefore, supporting execution-driven emulation is also an important issue that needs to be addressed.

Bibliography

[1] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Morgan Kaufmann Publishers Inc., 2017.

[2] Standard Performance Evaluation Corporation, "SPEC's Benchmarks," 2018. [Online]. Available: https://www.spec.org/benchmarks.html

[3] W. D. Strecker, "VAX-ll/780 – A Virtual Address Extension to the DEC PDP-11 Family," in *Proceedings of the National Computer Conference*, 1978, pp. 967–980.

[4] W. J. Dally and B. Towles, *Principles and Practices of Interconnection Networks*. Morgan Kaufmann Publishers, 2003.

[5] N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J. Kim, and W. J. Dally, "A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2013, pp. 86–96.

[6] D. H. Lawrie, "Access and Alignment of Data in an Array Processor," *IEEE Transactions on Computers (TC)*, vol. C-24, no. 12, pp. 1145–1155, 1975.

[7] D. Seo, A. Ali, W.-T. Lim, N. Rafique, and M. Thottethodi, "Near-Optimal Worst-Case Throughput Routing for Two-Dimensional Mesh Networks," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA)*, 2005, pp. 432–443.

[8] G. Moore, "Cramming More Components onto Integrated Circuits," *Electronics*, vol. 38, no. 8, 1965.

[9] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of Ion-Implanted MOSFET's with Very Small Physical Dimensions," *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 9, no. 5, pp. 256–268, 1974.

[10] Y. LeCun, Y. Bengio, and G. Hinton, "Deep Learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[11] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu, "Knights Landing: Second-Generation Intel Xeon Phi Product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[12] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, W. Zhao, X. Yin, C. Hou, C. Zhang, W. Ge, J. Zhang, Y. Wang, C. Zhou, and G. Yang, "The Sunway TaihuLight Supercomputer: System and Applications," *Science China Information Sciences*, vol. 59, no. 7, 2016.

[13] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor System-on-Chip (MPSoC) Technology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 27, no. 10, pp. 1701–1713, 2008.

[14] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 1–12.

[15] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, 2014, pp. 13–24.

[16] J. Fowers, K. Ovtcharov, M. Papamichael, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, L. Adams, M. Ghandi, S. Heil, P. Patel, A. Sapek, G. Weisz, L. Woods, S. Lanka, S. K. Reinhardt, A. M. Caulfield, E. S. Chung, and D. Burger, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *Proceeding of the 45th Annual International Symposium on Computer Architecuture (ISCA)*, 2018, pp. 1–14.

[17] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, D. Firestone, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov,

M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger, “Configurable Clouds,” *IEEE Micro*, vol. 37, no. 3, pp. 52–61, 2017.

[18] Y. H. Chen, T. Krishna, J. S. Emer, and V. Sze, “Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks,” *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 52, no. 1, pp. 127–138, 2017.

[19] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, “Minerva: Enabling Low-power, Highly-accurate Deep Neural Network Accelerators,” in *Proceedings of the 43rd International Symposium on Computer Architecture (ISCA)*, 2016, pp. 267–278.

[20] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, “Q100: The Architecture and Design of a Database Processing Unit,” in *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014, pp. 255–268.

[21] C. Tan, M. Karunaratne, T. Mitra, and L.-S. Peh, “Stitch: Fusible Heterogeneous Accelerators Enmeshed with Many-Core Architecture for Wearables,” in *Proceedings of the 45th International Symposium on Computer Architecture (ISCA)*, 2018, pp. 575–587.

[22] M. Mahmoud, B. Zheng, A. D. Lascorz, F. Heide, J. Assouline, P. Boucher, E. Onzon, and A. Moshovos, “IDEAL: Image Denoising Accelerator,” in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2017, pp. 82–95.

[23] T. Luo, S. Liu, L. Li, Y. Wang, S. Zhang, T. Chen, Z. Xu, O. Temam, and Y. Chen, “DaDianNao: A Neural Network Supercomputer,” *IEEE Transactions on Computers (TC)*, vol. 66, no. 1, pp. 73–88, 2017.

[24] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, “ShiDianNao: Shifting Vision Processing Closer to the Sensor,” in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 92–104.

[25] F. Akopyan, J. Sawada, A. Cassidy, R. Alvarez-Icaza, J. Arthur, P. Merolla, N. Imam, Y. Nakamura, P. Datta, G. J. Nam, B. Taba, M. Beakes, B. Brezzo, J. B. Kuang, R. Manohar, W. P. Risk, B. Jackson, and D. S. Modha, “TrueNorth: Design and Tool Flow of a 65 mW 1 Million Neuron Programmable Neurosynaptic Chip,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 34, no. 10, pp. 1537–1557, 2015.

[26] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A Scalable Processing-in-memory Accelerator for Parallel Graph Processing," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 105–117.

[27] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based Accelerator Design for Deep Convolutional Neural Networks," in *Proceedings of the 23rd ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2015, pp. 161–170.

[28] M. Alwani, H. Chen, M. Ferdman, and P. Milder, "Fused-layer CNN Accelerators," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 22:1–22:12.

[29] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, "From High-level Deep Neural Models to FPGAs," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016, pp. 17:1–17:12.

[30] X. Ma, D. Zhang, and D. Chiou, "FPGA-Accelerated Transactional Execution of Graph Workloads," in *Proceedings of the 25th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2017, pp. 227–236.

[31] J. Thong and N. Nicolici, "SAT Solving Using FPGA-Based Heterogeneous Computing," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015, pp. 232–239.

[32] J. Casper and K. Olukotun, "Hardware Acceleration of Database Operations," in *Proceedings of the 22nd ACM/SIGDA International Symposium on Field-programmable Gate Arrays (FPGA)*, 2014, pp. 151–160.

[33] W. Song, D. Koch, M. Lujan, and J. Garside, "Parallel Hardware Merge Sorter," in *Proceedings of the 24th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2016, pp. 95–102.

[34] S. Mashimo, T. V. Chu, and K. Kise, "High-Performance Hardware Merge Sorter," in *Proceedings of the 25th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2017, pp. 1–8.

[35] M. Saitoh, E. A. Elsayed, T. V. Chu, S. Mashimo, and K. Kise, "A High-Performance and Cost-Effective Hardware Merge Sorter without Feedback Datapath," in *Proceedings of the 26th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2018, pp. 197–204.

[36] W. J. Dally and B. Towles, “Route Packets, Not Wires: On-chip Interconnection Networks,” in *Proceedings of the 38th Design Automation Conference (DAC)*, 2001, pp. 684–689.

[37] L. Benini and G. D. Micheli, “Networks on Chips: A New SoC Paradigm,” *Computer*, vol. 35, no. 1, pp. 70–78, 2002.

[38] N. E. Jerger, T. Krishna, and L. S. Peh, *On-Chip Networks, Second Edition*. Morgan Claypool, 2017.

[39] J. Howard, S. Dighe, S. R. Vangal, G. Ruhl, N. Borkar, S. Jain, V. Erraguntla, M. Konow, M. Riepen, M. Gries, G. Droege, T. Lund-Larsen, S. Steibl, S. Borkar, V. K. De, and R. V. D. Wijngaart, “A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling,” *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 46, no. 1, pp. 173–183, 2011.

[40] J. Balkind, M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, A. Lavrov, M. Shahrad, A. Fuchs, S. Payne, X. Liang *et al.*, “OpenPiton: An Open Source Manycore Research Framework,” in *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016, pp. 217–232.

[41] M. McKeown, Y. Fu, T. Nguyen, Y. Zhou, J. Balkind, A. Lavrov, M. Shahrad, S. Payne, and D. Wentzlaff, “Piton: A Manycore Processor for Multitenant Clouds,” *IEEE Micro*, vol. 37, no. 2, pp. 70–80, 2017.

[42] B. K. Daya, C.-H. O. Chen, S. Subramanian, W.-C. Kwon, S. Park, T. Krishna, J. Holt, A. P. Chandrakasan, and L.-S. Peh, “SCORPIO: A 36-core Research Chip Demonstrating Snoopy Coherence on a Scalable Mesh NoC with In-network Ordering,” in *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA)*, 2014, pp. 25–36.

[43] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. C. Miao, J. F. B. III, and A. Agarwal, “On-Chip Interconnection Architecture of the Tile Processor,” *IEEE Micro*, vol. 27, no. 5, pp. 15–31, 2007.

[44] B. Bohnenstiehl, A. Stillmaker, J. J. Pimentel, T. Andreas, B. Liu, A. T. Tran, E. Adeagbo, and B. M. Baas, “KiloCore: A 32-nm 1000-Processor Computational Array,” *IEEE Journal of Solid-State Circuits (JSSC)*, vol. 52, no. 4, pp. 891–902, 2017.

[45] G. Kurian, J. E. Miller, J. Psota, J. Eastep, J. Liu, J. Michel, L. C. Kimerling, and A. Agarwal, “ATAC: A 1000-Core Cache-Coherent Processor with On-Chip Optical Network,” in *Proceedings of the 19th Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 477–488.

[46] R. Busseuil, L. Barthe, G. M. Almeida, L. Ost, F. Bruguier, G. Sassatelli, P. Benoit, M. Robert, and L. Torres, "Open-Scale: A Scalable, Open-Source NOC-based MPSoC for Design Space Exploration," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2011, pp. 357–362.

[47] J. Ax, G. Sievers, J. Daberkow, M. Flasskamp, M. Vohrmann, T. Jungeblut, W. Kelly, M. Porrmann, and U. Ruckert, "CoreVA-MPSoC: A Many-Core Architecture with Tightly Coupled Shared and Local Data Memories," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 29, no. 5, pp. 1030–1043, 2018.

[48] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an Ecosystem for a Scalable, Modular and High-Efficiency Embedded Computing Accelerator," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2012, pp. 983–987.

[49] A. Olofsson, "Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip," *CoRR*, vol. abs/1610.01832, 2016.

[50] A. Olofsson, T. Nordstrom, and Z. Ul-Abdin, "Kickstarting High-Performance Energy-Efficient Manycore Architectures with Epiphany," in *Proceedings of the 48th Asilomar Conference on Signals, Systems and Computers*, 2014, pp. 1719–1726.

[51] B. D. de Dinechin, R. Ayrignac, P. E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, and T. Strudel, "A Clustered Manycore Processor Architecture for Embedded and Accelerated Applications," in *Proceedings of the IEEE High Performance Extreme Computing Conference (HPEC)*, 2013, pp. 1–6.

[52] D. Sanchez, G. Michelogiannakis, and C. Kozyrakis, "An Analysis of On-chip Interconnection Networks for Large-scale Chip Multiprocessors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 7, no. 1, pp. 4:1–4:28, 2010.

[53] L.-S. Peh and W. J. Dally, "A Delay Model for Router Microarchitectures," *IEEE Micro*, vol. 21, no. 1, pp. 26–34, 2001.

[54] U. Y. Ogras, P. Bogdan, and R. Marculescu, "An Analytical Approach for Network-on-Chip Performance Analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 29, no. 12, pp. 2001–2013, 2010.

[55] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 Simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[56] A. Gutierrez, J. Pusdesris, R. G. Dreslinski, T. Mudge, C. Sudanthi, C. D. Emmons, M. Hayenga, and N. Paver, "Sources of Error in Full-System Simulation," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 13–22.

[57] D. Sanchez and C. Kozyrakis, "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems," in *Proceeding of the 40th Annual International Symposium on Computer Architecuture (ISCA)*, 2013, pp. 475–486.

[58] P. Ren, M. Lis, M. H. Cho, K. S. Shim, C. W. Fletcher, O. Khan, N. Zheng, and S. Devadas, "HORNET: A Cycle-Level Multicore Simulator," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, vol. 31, no. 6, pp. 890–903, 2012.

[59] N. Genko, D. Atienza, G. De Micheli, J. M. Mendias, R. Hermida, and F. Catthoor, "A Complete Network-On-Chip Emulation Framework," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2005, pp. 246–251 Vol. 1.

[60] P. T. Wolkotte, P. K. F. Holzenspies, and G. J. M. Smit, "Fast, Accurate and Detailed NoC Simulations," in *Proceedings of the 1st International Symposium on Networks-on-Chip (NOCS)*, 2007, pp. 323–332.

[61] Y. E. Krasteva, F. Criado, E. de la Torre, and T. Riesgo, "A Fast Emulation-Based NoC Prototyping Framework," in *Proceedings of the International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, 2008, pp. 211–216.

[62] S. Lotlikar, V. Pai, and P. V. Gratz, "AcENoCs: A Configurable HW/SW Platform for FPGA Accelerated NoC Emulation," in *Proceedings of the 24th International Conference on VLSI Design (VLSID)*, 2011, pp. 147–152.

[63] M. K. Papamichael, "Fast Scalable FPGA-Based Network-on-Chip Simulation Models," in *Proceedings of the 9th IEEE/ACM International Conference on Formal Methods and Models for Codesign (MEMOCODE)*, 2011, pp. 77–82.

[64] M. K. Papamichael, J. C. Hoe, and O. Mutlu, "FIST: A Fast, Lightweight, FPGA-Friendly Packet Latency Estimator for NoC Modeling in Full-System Simulations," in *Proceedings of the 5th International Symposium on Networks-on-Chip (NOCS)*, 2011, pp. 137–144.

[65] D. Wang, C. Lo, J. Vasiljevic, N. E. Jerger, and J. Gregory Steffan, "DART: A Programmable Architecture for NoC Simulation on FPGAs," *IEEE Transactions on Computers (TC)*, vol. 63, no. 3, pp. 664–678, 2014.

[66] T. Drewes, J. M. Joseph, and T. Pionteck, "An FPGA-Based Prototyping Framework for Networks-on-Chip," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, 2017, pp. 1–7.

[67] H. M. Kamali and S. Hessabi, “AdapNoC: A Fast and Flexible FPGA-based NoC Simulator,” in *Proceedings of the 26th International Conference on Field-Programmable Logic and Applications (FPL)*, 2016, pp. 1–8.

[68] H. M. Kamali, K. Z. Azar, and S. Hessabi, “DuCNoC: A High-Throughput FPGA-Based NoC Simulator Using Dual-Clock Lightweight Router Micro-Architecture,” *IEEE Transactions on Computers (TC)*, vol. 67, no. 2, pp. 208–221, 2018.

[69] S. Abba and J. Lee, “A Parametric-based Performance Evaluation and Design Trade-offs for Interconnect Architectures Using FPGAs for Networks-on-chip,” *Microprocessors and Microsystems*, vol. 38, no. 5, pp. 375–398, 2014.

[70] A. I. Khan, “Cycle-Accurate Modeling of Multicore Processors on FPGAs,” Ph.D. dissertation, Massachusetts Institute of Technology, 2013.

[71] R. Sasakawa and K. Kise, “LEF: Long Edge First Routing for Two-dimensional Mesh Network on Chip,” in *Proceedings of the 6th International Workshop on Network on Chip Architectures (NoCArc)*, 2013, pp. 5–10.

[72] G.-M. Chiu, “The Odd-Even Turn Model for Adaptive Routing,” *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 11, no. 7, pp. 729–738, 2000.

[73] C. Bienia, “Benchmarking Modern Multiprocessors,” Ph.D. dissertation, Princeton University, 2011.

[74] J. Hestness, B. Grot, and S. W. Keckler, “Netrace: Dependency-driven Trace-based Network-on-chip Simulation,” in *Proceedings of the 3rd International Workshop on Network on Chip Architectures (NoCArc)*, 2010, pp. 31–36.

[75] T. V. Chu, S. Sato, and K. Kise, “KNoCEmu: High Speed FPGA Emulator for Kilo-Node Scale NoCs,” in *Proceedings of the 8th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2014, pp. 215–222.

[76] T. V. Chu, S. Sato, and K. Kise, “Enabling Fast and Accurate Emulation of Large-scale Network on Chip Architectures on a Single FPGA,” in *Proceedings of the 23rd IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2015, pp. 60–63.

[77] T. V. Chu, S. Sato, and K. Kise, “Ultra-Fast NoC Emulation on a Single FPGA,” in *Proceedings of the 25th International Conference on Field-Programmable Logic and Applications (FPL)*, 2015, pp. 1–8.

[78] T. V. Chu, S. Sato, and K. Kise, “Fast and Cycle-Accurate Emulation of Large-Scale Networks-on-Chip Using a Single FPGA,” *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 10, no. 4, pp. 27:1–27:27, 2017.

[79] T. V. Chu, M. Kang, S. FA, and K. Kise, "Enhanced Long Edge First Routing Algorithm and Evaluation in Large-Scale Networks-on-Chip," in *Proceedings of the 11th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, 2017, pp. 83–90.

[80] T. V. Chu and K. Kise, "An Effective Architecture for Trace-Driven Emulation of Networks-on-Chip on FPGAs," in *Proceedings of the 28th International Conference on Field-Programmable Logic and Applications (FPL)*, 2018, pp. 419–426.

[81] J. Kim, J. Balfour, and W. Dally, "Flattened Butterfly Topology for On-Chip Networks," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007, pp. 172–182.

[82] C. E. Leiserson, "Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing," *IEEE Transactions on Computers (TC)*, vol. C-34, no. 10, pp. 892–901, 1985.

[83] H. Matsutani, M. Koibuchi, Y. Yamada, D. F. Hsu, and H. Amano, "Fat H-Tree: A Cost-Efficient Tree-Based On-Chip Network," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 20, no. 8, pp. 1126–1141, 2009.

[84] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs," *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002.

[85] C. J. Glass and L. M. Ni, "The Turn Model for Adaptive Routing," in *Proceedings of the 19th Annual International Symposium on Computer Architecture (ISCA)*, 1992, pp. 278–287.

[86] W. J. Dally, "Virtual-Channel Flow Control," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 3, no. 2, pp. 194–205, 1992.

[87] M. Galles, "Spider: A High-Speed Network Interconnect," *IEEE Micro*, vol. 17, no. 1, pp. 34–39, 1997.

[88] L.-S. Peh and W. J. Dally, "A Delay Model and Speculative Architecture for Pipelined Routers," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture (HPCA)*, 2001, pp. 255–266.

[89] A. Kumar, L.-S. Peh, P. Kundu, and N. K. Jha, "Express Virtual Channels: Towards the Ideal Interconnection Fabric," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, 2007, pp. 150–161.

[90] D. Park, R. Das, C. Nicopoulos, J. Kim, N. Vijaykrishnan, R. Iyer, and C. R. Das, “Design of a Dynamic Priority-Based Fast Path Architecture for On-Chip Interconnects,” in *Proceedings of the 15th Annual IEEE Symposium on High-Performance Interconnects (HOTI)*, 2007, pp. 15–20.

[91] H. Matsutani, M. Koibuchi, H. Amano, and T. Yoshinaga, “Prediction Router: A Low-Latency On-Chip Router Architecture with Multiple Predictors,” *IEEE Transactions on Computers (TC)*, vol. 60, no. 6, pp. 783–799, 2011.

[92] Xilinx, “7 Series FPGAs Configurable Logic Block User Guide,” 2018. [Online]. Available: https://www.xilinx.com/support/documentation/user_guides/ug474_7Series_CLB.pdf

[93] G. M. Amdahl, “Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities,” in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, 1967, pp. 483–485.

[94] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar, “An 80-Tile 1.28TFLOPS Network-on-Chip in 65nm CMOS,” in *Proceedings of the IEEE International Solid-State Circuits Conference (ISSCC). Digest of Technical Papers*, 2007, pp. 98–589.

[95] H. G. Lee, N. Chang, U. Y. Ogras, and R. Marculescu, “On-chip Communication Architecture Exploration: A Quantitative Evaluation of Point-to-point, Bus, and Network-on-chip Approaches,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 12, no. 3, pp. 23:1–23:20, 2008.

[96] U. Y. Ogras and R. Marculescu, “"It’s a Small World after All": NoC Performance Optimization via Long-Range Link Insertion,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 7, pp. 693–706, 2006.

[97] U. Y. Ogras, R. Marculescu, H. G. Lee, P. Choudhary, D. Marculescu, M. Kaufman, and P. Nelson, “Challenges and Promising Results in NoC Prototyping Using FPGAs,” *IEEE Micro*, vol. 27, no. 5, pp. 86–95, 2007.

[98] T. Le and M. A. S. Khalid, “NoC Prototyping on FPGAs: A Case Study Using an Image Processing Benchmark,” in *Proceedings of the IEEE International Conference on Electro/Information Technology*, 2009, pp. 441–445.

[99] A. Patel, F. Afram, S. Chen, and K. Ghose, “MARSS: A Full System Simulator for Multicore x86 CPUs,” in *Proceedings of the 48th Design Automation Conference (DAC)*, 2011, pp. 1050–1055.

[100] N. Agarwal, T. Krishna, L. S. Peh, and N. K. Jha, "GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009, pp. 33–42.

[101] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti, "Cycle-Accurate Network on Chip Simulation with Noxim," *ACM Transactions on Modeling and Computer Simulation*, vol. 27, no. 1, pp. 4:1–4:25, 2016.

[102] CMU-SAFARI, "NOCulator," 2018. [Online]. Available: https://github.com/CMU-SAFARI/NOCulator

[103] J. Wang, Y. Huang, M. Ebrahimi, L. Huang, Q. Li, A. Jantsch, and G. Li, "VisualNoC: A Visualization and Evaluation Environment for Simulation and Mapping," in *MES*, 2016, pp. 18–25.

[104] Access IC Lab, "Access Noxim," 2018. [Online]. Available: http://access.ee.ntu.edu.tw/noxim/index.html

[105] J. Wawrzynek, D. Patterson, M. Oskin, S. L. Lu, C. Kozyrakis, J. Hoe, D. Chiou, and K. Asanovic, "RAMP: Research Accelerator for Multiple Processors," *IEEE Micro*, vol. 27, no. 2, pp. 46–57, 2007.

[106] N. Njoroge, J. Casper, S. Wee, Y. Teslyar, D. Ge, C. Kozyrakis, and K. Olukotun, "ATLAS: A Chip-multiprocessor with Transactional Memory Support," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*, 2007, pp. 3–8.

[107] S. Wee, J. Casper, N. Njoroge, Y. Tesylar, D. Ge, C. Kozyrakis, and K. Olukotun, "A Practical FPGA-based Framework for Novel CMP Research," in *Proceedings of the 15th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2007, pp. 116–125.

[108] D. Chiou, D. Sunwoo, J. Kim, N. Patil, W. H. Reinhart, D. E. Johnson, and Z. Xu, "The FAST Methodology for High-speed SoC/Computer Simulation," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2007, pp. 295–302.

[109] D. Chiou, D. Sunwoo, J. Kim, N. A. Patil, W. Reinhart, D. E. Johnson, J. Keefe, and H. Angepat, "FPGA-Accelerated Simulation Technologies (FAST): Fast, Full-System, Cycle-Accurate Simulators," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2007, pp. 249–261.

[110] E. S. Chung, E. Nurvitadhi, J. C. Hoe, B. Falsafi, and K. Mai, "A Complexity-effective Architecture for Accelerating Full-system Multiprocessor Simulations Using FPGAs," in

Proceedings of the 16th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA), 2008, pp. 77–86.

[111] E. S. Chung, M. K. Papamichael, E. Nurvitadhi, J. C. Hoe, K. Mai, and B. Falsafi, "ProtoFlex: Towards Scalable, Full-System Multiprocessor Simulations Using FPGAs," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 2, no. 2, pp. 15:1–15:32, 2009.

[112] Z. Tan, A. Waterman, R. Avizienis, Y. Lee, H. Cook, D. Patterson, and K. Asanović, "RAMP Gold: An FPGA-based Architecture Simulator for Multiprocessors," in *Proceedings of the 47th Design Automation Conference (DAC)*, 2010, pp. 463–468.

[113] M. Pellauer, M. Adler, M. Kinsy, A. Parashar, and J. Emer, "HAsim: FPGA-Based High-Detail Multicore Simulation Using Time-division Multiplexing," in *Proceedings of the 17th International Symposium on High-Performance Computer Architecture (HPCA)*, 2011, pp. 406–417.

[114] A. Khan, M. Vijayaraghavan, S. Boyd-Wickizer, and Arvind, "Fast and Cycle-Accurate Modeling of A Multicore Processor," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2012, pp. 178–187.

[115] M. Pellauer, M. Vijayaraghavan, M. Adler, Arvind, and J. Emer, "A-Ports: An Efficient Abstraction for Cycle-Accurate Performance Models on FPGAs," in *Proceedings of the 16th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2008, pp. 87–96.

[116] M. A. Kinsy, M. Pellauer, and S. Devadas, "Heracles: A Tool for Fast RTL-Based Design Space Exploration of Multicore Processors," in *Proceedings of the 21st ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2013, pp. 125–134.

[117] N. Kapre and J. Gray, "Hoplite: Building Austere Overlay NoCs for FPGAs," in *Proceedings of the 25th International Conference on Field-Programmable Logic and Applications (FPL)*, 2015, pp. 1–8.

[118] N. Kapre and J. Gray, "Hoplite: A Deflection-Routed Directional Torus NoC for FPGAs," *ACM Transactions Reconfigurable Technology Systems*, vol. 10, no. 2, pp. 14:1–14:24, 2017.

[119] E. S. Chung, "CoRAM: An In-Fabric Memory Architecture for FPGA-Based Computing," Ph.D. dissertation, Carnegie Mellon University, 2011.

[120] M. Badr and N. E. Jerger, "SynFull: Synthetic Traffic Models Capturing Cache Coherent Behaviour," in *Proceedings of the 41st Annual International Symposium on Computer Architecture (ISCA)*, 2014, pp. 109–120.

[121] J. Yin, O. Kayiran, M. Poremba, N. E. Jerger, and G. H. Loh, “Efficient Synthetic Traffic Models for Large, Complex SoCs,” in *Proceedings of the 22nd IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2016, pp. 297–308.

[122] N. McKeown, “The iSLIP Scheduling Algorithm for Input-Queued Switches,” *IEEE/ACM Transactions on Networking*, vol. 7, no. 2, pp. 188–201, 1999.

[123] L. Shannon, V. Cojocaru, C. N. Dao, and P. H. W. Leong, “Technology Scaling in FPGAs: Trends in Applications and Architectures,” in *Proceedings of the 23rd IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2015, pp. 1–8.

[124] S. Vigna, “Further scramblings of Marsaglia’s xorshift generators,” *Journal of Computational and Applied Mathematics*, vol. 315, pp. 175–181, 2017.

[125] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[126] J. Hu and R. Marculescu, “DyAD: Smart Routing for Networks-on-Chip,” in *Proceedings of the 41st Design Automation Conference (DAC)*, 2004, pp. 260–263.

[127] L. G. Valiant and G. J. Brebner, “Universal Schemes for Parallel Communication,” in *Proceedings of the 13th Annual ACM Symposium on Theory of Computing (STOC)*, 1981, pp. 263–277.

[128] T. Nesson and S. L. Johnsson, “ROMM Routing on Mesh and Torus Networks,” in *Proceedings of the 7th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 1995, pp. 275–287.

[129] B. Towles and W. J. Dally, “Worst-Case Traffic for Oblivious Routing Functions,” *IEEE Computer Architecture Letters*, vol. 1, no. 1, pp. 4–4, 2002.

[130] S. Kumar, Y. Sabharwal, R. Garg, and P. Heidelberger, “Optimization of All-to-All Communication on the Blue Gene/L Supercomputer,” in *Proceedings of the 37th International Conference on Parallel Processing (ICPP)*, 2008, pp. 320–329.

[131] D. Chen, N. A. Eisley, P. Heidelberger, R. M. Senger, Y. Sugawara, S. Kumar, V. Salapura, D. L. Satterfield, B. Steinmacher-Burow, and J. J. Parker, “The IBM Blue Gene/Q Interconnection Network and Message Unit,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2011, pp. 26:1–26:10.

[132] D. Chen, N. Eisley, P. Heidelberger, S. Kumar, A. Mamidala, F. Petrini, R. Senger, Y. Sugawara, R. Walkup, B. Steinmacher-Burow, A. Choudhury, Y. Sabharwal, S. Singhal, and

J. J. Parker, “Looking Under the Hood of the IBM Blue Gene/Q Network,” in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, pp. 69:1–69:12.

[133] S. Ma, N. E. Jerger, and Z. Wang, “Whole Packet Forwarding: Efficient Design of Fully Adaptive Routing Algorithms for Networks-on-chip,” in *Proceedings of the 18th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2012, pp. 1–12.

[134] M. Li, Q.-A. Zeng, and W.-B. Jone, “DyXY: A Proximity Congestion-aware Deadlock-free Dynamic Routing Method for Network on Chip,” in *Proceedings of the 43rd Annual Design Automation Conference (DAC)*, 2006, pp. 849–852.

[135] S. Kilts, *Advanced FPGA Design: Architecture, Implementation, and Optimization*. Wiley-IEEE Press, 2007.

Published Material

The work in this dissertation is largely based on the following publications:

Peer-reviewed Journal Papers

- Thiem Van Chu, Shimpei Sato, and Kenji Kise, "Fast and Cycle-Accurate Emulation of Large-Scale Networks-on-Chip Using a Single FPGA," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, Volume 10, Issue 4, pp. 27:1–27:27, December 2017.

Peer-reviewed Conference Papers

- Thiem Van Chu, Shimpei Sato, and Kenji Kise, "KNoCEmu: High Speed FPGA Emulator for Kilo-Node Scale NoCs," in *Proceedings of the 8th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pp. 215–222, Aizu-Wakamatsu, Japan, September 2014.
- Thiem Van Chu, Shimpei Sato, and Kenji Kise, "Enabling Fast and Accurate Emulation of Large-scale Network on Chip Architectures on a Single FPGA," in *Proceedings of the 23rd IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 60–63, Vancouver, British Columbia, Canada, May 2015. (Short Paper)
- Thiem Van Chu, Shimpei Sato, and Kenji Kise, "Ultra-Fast NoC Emulation on a Single FPGA," in *Proceedings of the 25th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 1–8, London, UK, September 2015.
- Thiem Van Chu, Myeonggu Kang, Shi FA, and Kenji Kise, "Enhanced Long Edge First Routing Algorithm and Evaluation in Large-Scale Networks-on-Chip," in *Proceedings of the 11th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*, pp. 83–90, Seoul, Korea, September 2017.

- Thiem Van Chu and Kenji Kise, "An Effective Architecture for Trace-Driven Emulation of Networks-on-Chip on FPGAs," in *Proceedings of the 28th International Conference on Field-Programmable Logic and Applications (FPL)*, pp. 419–426, Dublin, Ireland, August 2018.