TOKYO INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Ultra-Fast and Accurate Simulation for Large-Scale Many-Core Processors

Thiem Van Chu

A thesis submitted in partial fulfillment of
the requirements for the degree of Master of Engineering

Advisor: Associate Professor Kenji Kise

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

July 2015

Abstract

Many-core processor architectures are becoming mainstream. With the many-core trend, Network-on-Chip (NoC) has become the de facto on-chip communication fabric, replacing traditional bus-based architectures. Targeting thousands of cores in near future many-core architectures, large-scale NoC designs need to be modeled and evaluated fast and accurately to understand their performance characteristics as well as the impact on the overall system.

This thesis proposes novel methods for emulating large-scale NoC architectures on a single FPGA (Field-Programmable Gate Array). The scalability is improved without simplifying the emulated architectures or using off-chip resources. The thesis first describes how to accurately model synthetic workloads on FPGA by separating the time of the emulated network and the times of the traffic generation units. The thesis next proposes a novel use of time-multiplexing in emulating the entire network using a small number of physical nodes. Finally, the thesis shows the basic steps for applying the proposed methods to emulate different NoC architectures.

The proposed methods enable ultra-fast and accurate emulations of large-scale NoC architectures with up to thousands of nodes using only on-chip resources of a single FPGA. The evaluation results show that more than 5,000× simulation speedup over BookSim, one of the most widely used software-based NoC simulators, is achieved while the simulation accuracy is maintained.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With the continuous advances in semiconductor process technology, for decades, microprocessors' performance has been improved mainly by increasing clock frequency and extensively exploiting instruction-level parallelism. However, the direction changed in the 2000s due to many issues such as the power wall and the limitation of instruction-level parallelism techniques. Instead of trying to increase performance of single-core processors, the microprocessor industry has shifted to integrating multiple processor cores on a chip.

Many-core processors have now become main stream. As the number of cores integrated on a chip increases, traditional on-chip communication architectures such as bus-based architectures fail to keep pace with the increasing demand of high communication performance. To date, Network-on-Chip (NoC) [1–3] (packet-switched interconnection network) has been widely regarded as the de facto on-chip communication architecture for many-core systems. This thesis focuses on NoC simulation rather than full-system simulation. Because integrating more cores on a chip makes the interconnection between cores more important, NoC simulation is a critical part of evaluating trade-offs in designing many-core processors with hundreds to thousands of cores.

Software-based simulators have been widely used for architectural exploration. Full-system simulators [4–9] model the overall system, from the interconnection network to processor cores, memory hierarchy and cache, and can execute realistic workloads. However, they either are too slow to simulate architectures with more than 100 cores in practical time, or do not support accurate simulation. To reduce complexity, most existing full-system simulators simplify the contention in the network. They use a fixed latency value for any condition of the network. Although this simplification is tolerable for some cases, especially when the number of cores is small, studying an NoC architecture and its effect to the overall system requires detailed models. To address this problem, some stand-alone network simulators such as BookSim [10], GARNET [11], Noxim [12], SICOSYS [13] have been proposed. They are useful for studying NoC designs independently, and several orders of magnitude faster than full-system simulators.

One of the essential advantages of software-based simulators is the wide variety of programming tools. Additionally, they are flexible and easy to debug. However, software-based simulators have several drawbacks. It is easy to produce designs that would be impractical to implement in hardware. And more importantly, software-based simulators are slow. Simulation speeds of typical full-system simulators range from several KIPS (Kilo Instructions Per Second) to hundreds of KIPS depending on the detail level of the simulations. At these speeds, the simulation of one second of one core may take several hours or days to complete. For instance, if we use a 50-KIPS simulator to simulate a relatively slow 1-GIPS (Giga Instructions Per Second) processor core, it will take 20,000 seconds (more than 5.5 hours) to complete one second of the core. For thousands of cores, the simulation time is thus impractical.

Even stand-alone network simulators are getting slower rapidly as the number of simulated cores increases. This can be seen in an analysis of the performance of BookSim, one of the most popular software-based NoC simulator, in Figure 1.1. The figure shows how the simulation speed of BookSim [10] decreases when changing the simulated NoC's size from 16 nodes to 144 nodes. As shown in Figure 1.1(a), BookSim's simulation speed depends on both the simulated NoC's size and the amount of network activity which is represented by the *flit injection rate*. Higher amount of network activity requires longer simulation time because more tasks need to be executed. Figure 1.1(b) shows the slowdown in simulation speed of BookSim at an injection rate of 0.025 (*flits/node/cycle*) when changing the simulated NoC's size from 16 nodes to 144 nodes. The slowdown is normalized to the case where the number of nodes is 16. The NoC architecture and evaluation environment here are same as the primary architecture and the environment that will be described in Chapter 4. We can see that the simulation speed of BookSim decreases more than 17 times when the number of nodes is

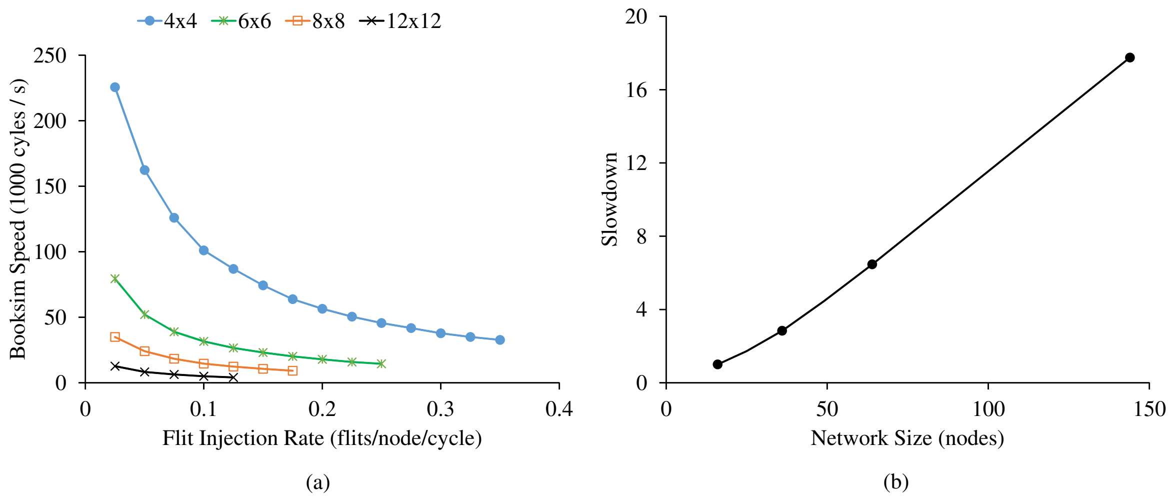


Figure 1.1: (a) Simulation speed of BookSim with different network sizes and traffic injection rates. (b) Slowdown in simulation speed of BookSim at an injection rate of 0.025 (*flits/node/cycle*) when changing the simulated NoC's size from 16 nodes to 144 nodes.

increased nine times.

Unfortunately, it is difficult to significantly improve the simulation speed by using thread-level parallelism because of the huge amount of synchronizations in the simulated network. Parallelizing a simulator and leveraging modern multi-core processors to increase the simulation speed is non-trivial. Without sacrificing accuracy, only a limited degree of parallelization can be achieved. On the other hand, compromising on accuracy may make the simulator unsuitable for studying new architectural proposals because some software/hardware errors that only occur under certain timing conditions may be hidden.

Targeting thousands of cores in the near future many-core architectures, large-scale NoC designs need to be modeled and evaluated fast and accurately to understand their performance characteristics as well as the impact on the overall system. With the ever increasing capacity of FPGAs (Field-Programmable Gate Arrays), FPGA-based emulation is becoming a promising approach. By using FPGAs, an ultra-fast simulation speed can be achieved because many operations can be simulated simultaneously in a tick of FPGAâĂŹs clock. Moreover, adding detail to a model requires more hardware, but does not necessary degrade performance. This is a significant difference compared to software-based simulators which are slower when the model is more sophisticated.

However, scaling up to a large number of NoC nodes is still a challenging task due to the FPGA capacity constraints. If a single FPGA is used, the available resources are limited. Even a large FPGA can fit only around 100 NoC routers of moderate complexity. To reduce the amount of required FPGA resources, some FPGA-based NoC emulators simplify the emulated router architectures. For instance, DART [14] reduces the number of channels and pipeline depth. FIST [15] abstractly models each router in the network as a set of load-delay curves obtained by offline or online training.

The limitation of FPGA resources can be mitigated by using multiple FPGAs [16] and off-chip memory (usually DRAM) [17, 18]. However, these approaches not only lead to a higher cost, but also make the entire system more complex and slower. The off-chip communication restricts performance of the entire system.

This thesis proposes novel methods which enable ultra-fast and accurate emulations of large-scale NoC designs with up to thousands of nodes using only on-chip resources of a single FPGA. The thesis first describes how to accurately model synthetic workloads on FPGA by separating the time of the emulated network and the times of the traffic generation units. Synthetic workloads are flexible and very useful in early stages of network evaluation. They can be used to quickly stress the emulated NoC designs and capture their bottlenecks. In general, to properly emulate a network under a specific synthetic workload in open-loop simulations, a large FIFO buffer (*source queue* in Figure 3.1) is needed between each traffic generation unit and the network to make sure that no packet is dropped in the traffic generation process. Every packet generated by traffic generation unit *i* is stored in source queue *i*, the FIFO buffer between traffic generation unit *i* and the network, until it can be accepted by the network. The large FIFO buffers cannot be implemented using only FPGA on-chip memory. The proposed method allows us to manage the effect of the feedback from the network to the packet generation units using small FIFO buffers, and thus the use of off-chip memory can be avoided.

The thesis next proposes a novel use of time-multiplexing in emulating the entire network using a small number of physical nodes. The number of physical nodes can be flexible. More physical nodes will improve the simulation speed but consume more hardware resources. Although the time-division multiplexing (TDM) technique has been adopted in several previous work [14, 17, 18], it has not been discussed thoroughly. In contrast, the thesis provides detailed discussions of how to effectively apply the TDM technique in FPGA-based NoC emulations.

Although the TDM technique can reduce the use of combinational logic, it does not help to reduce the total amount of required memory. In contrast, an additional memory is needed to store data passed between logical nodes. Without the first proposed method, which helps to manage the effect of the network to the traffic generation process using only a limited amount of memory, the use of off-chip memory is unavoidable. The combination of this method and the method based on time-multiplexing is the key factor to scale to NoC designs with thousands of nodes on a single FPGA.

Finally, the thesis shows how the proposed methods can be applied to emulate different NoC architectures.

The thesis has three main contributions.

(1) The thesis proposes an ultra-fast and accurate FPGA-based NoC emulator which is able to emulate large-scale NoC designs with up to thousands of nodes using a single FPGA. The evaluation results show that, when emulating an 128x128 mesh network (16,384 nodes) with state-of-the-art router architectures, more than 5,000× simulation speedup over BookSim, one of the most widely used software-based NoC simulators, is achieved while the simulation accuracy is maintained. The thesis also shows the basic steps for applying the proposed methods to emulate other NoC architectures.

(2) The thesis proposes a novel method for accurately emulating NoC designs under synthetic workloads without using a large amount of memory. With this method, the use of off-chip memory can be avoided.

(3) The thesis presents and discusses a novel use of the TDM technique in NoC emulations on FPGA. This method together with the method in (2) allow us to emulate NoC designs with up to thousands of nodes on a single FPGA.

The proposed methods are independent of the emulated NoC architectures, and do not require a specific type of FPGAs. In addition to the NoC designs demonstrated in this thesis, they can be applied to emulate other NoC designs including emerging 3D NoC designs such as MIRA [19]. Besides, the proposed emulator can be used to evaluate interconnection networks of not only many-core processors but also many other systems including Systems-on-Chip (SoCs) and large-scale supercomputers.

The rest of this thesis is organized as follows. Chapter 2 provides an overview of the many-core trend as well as the basics of NoC, and summarizes state-of-the-art approaches in NoC simulation/emulation. In Chapter 3, the emulation model is first described. After that, two methods which enable ultra-fast and accurate emulations of large-scale NoC designs with up to thousands of nodes are proposed. The first method helps to eliminate the memory constraint in NoC simulation, which has been described earlier, and thus helps to avoid using off-chip memory. The second method is based on time-multiplexing in which a small number of interconnected nodes are used to emulate the entire network by sequentially emulating every part of the network. The novel use of time-multiplexing helps to utilize FPGA resources effectively while maintaining the simulation accuracy. Chapter 3 also describes the basic steps for applying the proposed methods to emulate different NoC designs. The evaluation results are analyzed in Chapter 4. Finally, Chapter 5 summarizes the contributions of this thesis and discusses future work.

Chapter 2

Background

As the number of cores integrated on a chip increases, the interconnection between cores becomes a major concern. Therefore, NoC designs need to be modeled and evaluated in various aspects to understand their performance characteristics and trade-offs involved in designing the overall system. This chapter first describes the many-core trend and key basic concepts of NoC. After that, state-of-the-art approaches in NoC simulation/emulation are summarized and discussed.

2.1 Many-Core Architectures

For decades, the continuous improvements in semiconductor process technology according to Moore's law coupled with Dennard scaling have been a fundamental driver for increasing microprocessors' performance. Moore's law states that the number of transistors that can be cost-effectively integrated on a chip doubles every 24 months. At the same time, Dennard scaling observes that transistors' power density stays constant when their size decreases. Therefore, architects could improve microprocessors' performance exponentially by increasing clock frequency and extensively exploiting instruction-level parallelism techniques such as out-of-order execution and aggressive branch prediction.

However, recent trends have made it extremely hard to increasingly improve microprocessors' performance in the conventional way. Dennard scaling appears to have broken down due to the current leakage at small sizes. Therefore, even smaller and faster transistors can still be produced, increasing the number of transistors while maintaining the same clock frequency would require more power. The power wall has become a fundamental obstacle in increasing performance of microprocessors. On the other hand, all instruction-level parallelism techniques have their own limitations. The Intel Pentium 4 was widely known to have gone beyond the point of diminishing returns in exploiting instruction-level parallelism techniques.

Instead of trying to increase performance of single-core processors, all processor manufactures have now shifted to multi/many-core architectures. Most processors used in smartphones, PCs, servers, and supercomputers today are multi/many-core processors. Because many applications in some areas such as cloud computing and multimedia can easily gain significant benefits from using processors with a large number of cores, there already exist some commercial many-core processors with up to hundred cores. For instance, the EZchip's TILE-Gx72 processor [20] targeting networking and security workloads contains 72 64-bit RISC cores. The recently released EZchip's TILE-MX processor [21] offers massive computing power with 100 ARM v8 64-bit cores. The Intel Xeon Phi coprocessors [22–24] provide up to more than 60 cores and have been used in many supercomputers such as Tianhe-2, the fastest supercomputer in the world according to the TOP500 list in June 2015 [25].

Many researchers predicted that processors with a large number of cores would be the future. The near future architectures will have up to thousands of cores on a chip [8, 26–28]. Moving towards many-core architectures, however, requires many challenges to be solved. One of them is the interconnection between cores which is the main focus of this thesis.

2.2 NoC Basics

Any NoC architecture can be characterized by four properties: topology, routing, flow control, and router architecture [2, 3].

2.2.1 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 an NoC. The topology establishes an optimal bound on performance, i.e. throughput and latency, of a network. The routing algorithm, flow control mechanism, and router architecture determine how closely the optimal performance bound can be approached.

Direct network vs. indirect network. 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 [29]) and trees (e.g., Fat H-Tree [30]) distinguish between terminal node and switch node. In particular, each terminal node is a core while each switch node is a router.

2.2.2 Routing

With the road map determined by the topology, routing algorithms define which path a packet takes to reach its destination. A routing algorithm can be categorized as deterministic, oblivious, or adaptive, depending on the number of paths between each pair of nodes and how these paths are determined. A deterministic routing algorithm defines a single static path between each pair of nodes. For example, in 2-dimensional mesh networks, a widely used deterministic routing algorithm is dimension-order routing (XY routing) where a packet is routed first in the x-dimension and then in the y-dimension to reach its destination. Although dimension-order routing may produce load imbalance for some traffic patterns, it is preferred in many cases because it is very simple to implement and simplifies the deadlock avoidance problem.

Contrary to deterministic routing algorithms, both oblivious and adaptive routing algorithms allow multiple paths between each pair of nodes. However, while the routing decisions in oblivious routing algorithms are made regardless of the condition of the network, adaptive routing algorithms use network's congestion information to choose an appropriate path for each packet.

2.2.3 Flow Control

A flow control mechanism determines how shared resources in the network such as routers' buffers and channels are allocated when contention occurs. Most NoCs use wormhole and virtual channel (VC) flow control. In wormhole flow control, routers' 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.

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 necessary. 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.

VC flow control [31] 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 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 routers' buffers.

2.2.4 Router Architecture

Router architecture defines the internal organization and pipeline structure of routers in which the routing algorithm and flow control mechanism are implemented. Figure 2.1 shows the architecture of a canonical input-queued VC router consisting of the following components: input FIFO buffers, routing logic, VC allocator, switch allocator, and crossbar switch. The router is pipelined at the flit level. The typical five-stage pipeline structure consists of Routing Computation (RC), VC Allocation (VA), Switch Allocation (SA), Switch Traversal (ST), and

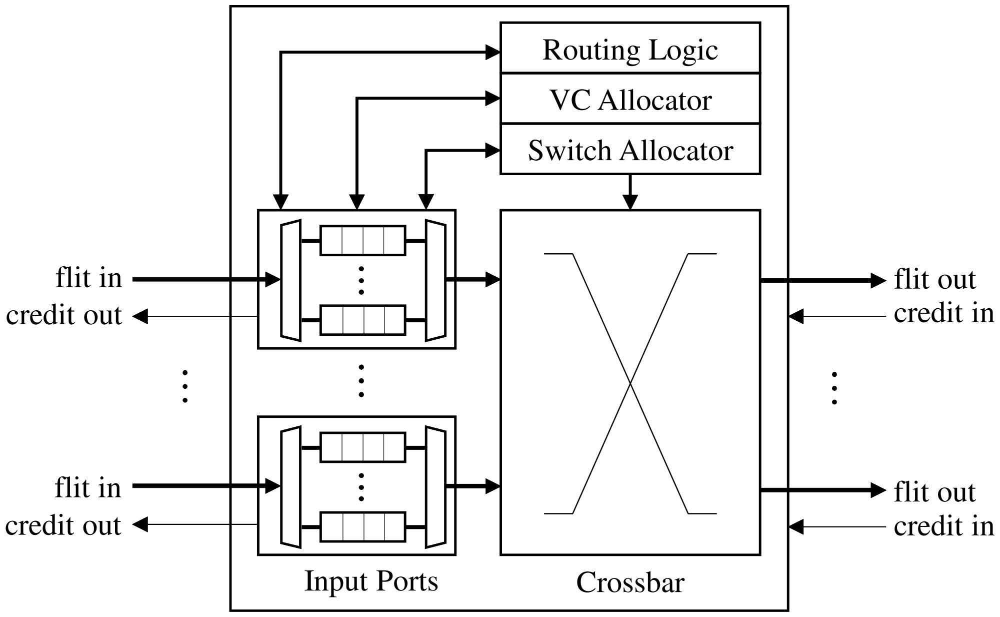


Figure 2.1: A canonical input-queued VC router architecture.

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 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, traverses the crossbar, 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 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. Some typical approaches include speculative architecture [32], look-ahead routing [2, 33], bypassing [34, 35], prediction [36]. Chapter 4 will show how the look-ahead routing technique can improve the network performance by using the proposed FPGA-based NoC emulator.

2.3 Related Work

2.3.1 Software-Based Models

Software-based simulators are very flexible and can leverage the wide variety of programming tools. Thus, they have been widely used by researchers. Sequential full-system simulators such as gem5 [4] and MARSS [5] can be designed to be very accurate. However, they are so slow that most studies are restricted to architectures with less than 64 cores. Simulating architectures with hundreds to thousands of cores would take months to years to complete, and thus is impractical.

Unfortunately, existing parallelization techniques scale poorly because of the high synchronization cost. Most parallel full-system simulators sacrifice simulation accuracy for speed. For instance, ZSim [8] accelerates the simulation of processor cores using instruction-driven timing models instead of cycle-driven or event-driven timing models. ZSim also proposes a two-phase parallelization technique, which divides the simulation into many small intervals of several thousand cycles and allows to simulate processor cores in parallel for each interval while ignoring resource contentions and using zero-load latencies for all memory accesses. In an interval, the loss of accuracy of the parallelization technique 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.

Compared to full-system simulators, stand-alone network simulators provide more detailed network models and are general several orders of magnitude faster. GARNET [11] is an event-driven network simlator that has been incorporated into gem5 [4]. BookSim [10] is a detailed and cycle-accurate network 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. Compared to event-driven network simulators such as GARNET, BookSim provides a higher level of accuracy, but is slower when the network load is low. BookSim has been validated against a well-known RTL NoC router [37].

Although software-based simulators offer many advantages, they are too slow to simulate architectures with up to thousands of cores. In general, parallelization is challenging because of the following two reasons. First, within one simulation cycle, the number of activities that can be parallelized increases rapidly with increasing the number of simulated cores. Second, a high amount of communication among parallelized activities occurs across simulation cycles, which is hard to manage efficiently using typical communication methods such as shared memory.

2.3.2 FPGA-Based Models

Addressing the simulation speed problem of software-based full-system simulators, some FPGA-based models have been proposed. However, most of them do not support detailed NoC models and can scale to only several tens of cores. RAMP Gold [38] can emulate up to 64 SPARC cores (without a detailed model of NoC interconnect) on a Virtex-5 FPGA, but with some compromises on accuracy. HAsim [39] and Heracles [40] use NoC for the communication fabric, but the number of cores that can be emulated is less than 32. Several many-core emulators such as Arete [41] are built using multiple FPGAs. However, this approach leads to much more complex designs. Moreover, the off-chip communication between FPGAs restricts performance of the entire system.

DART [14] is an FPGA-based NoC emulator supporting both trace-driven workloads and synthetic workloads. DART provides a global interconnect between all nodes. 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. However, 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 of this, it is difficult to scale DART to simulate large NoC designs because the area cost of the crossbar increases quadratically with respect to the number of input and output ports. Moreover, the routing tables become larger when increasing the number of nodes. With thousands of nodes, the use of off-chip memory is unavoidable.

DART provides a TDM option which simulates the entire network using one DART node.

As a result, the number of emulated nodes is increased. The TDM technique helps to reduce the required combinational logic, but it cannot reduce the amount of required memory. Although block RAMs (BRAMs) can be utilized more effectively, the limitation of memory cannot be solved. To further reduce the hardware resource usage, DART simplifies the emulated router architecture. In particular, the emulated router has only one output port and is a single-stage router. On the other hand, the FPGA-based emulator proposed in this thesis can emulate NoC architectures containing thousands of nodes with state-of-the-art pipelined router architectures.

FIST is an FPGA-based NoC emulator proposed by Papamichael et al. [15]. Compared to DART, FIST adopts a different approach. Rather than cycle-accurately emulating the NoC's operation, FIST abstractly models each router in the network as a set of load-delay curves. For a given network configuration and traffic pattern, these load-delay curves are 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. This approach helps FIST to achieve a higher simulation speed and significantly reduces the amount of required FPGA resources. FIST can emulate a 20×20 mesh NoC on a Xilinx Virtex 5 LX155T and up to 24×24 mesh on a Virtex 6 LX760 FPGA. However, the main disadvantage of FIST is that it does not provide cycle-accurate emulation.

Papamichael also proposed another FPGA-based NoC emulator [17] using two approaches: direct-mapped implementation, and virtualized implementation. In the former approach, the emulated NoC is directly implemented on an FPGA. The later approach adopts the TDM technique. In both approaches, a single-stage router architecture is implemented to minimize the FPGA resource usage. Additionally, Microblaze, a soft processor, is used to initialize the traffic tables as well as simulation parameters of the NoC emulator, and monitor the simulation result. Off-chip DRAM is required to store the traffic tables. Hence, performance of the entire system is restricted by the off-chip DRAM access time and bandwidth.

Wolkotte et al. [18] proposed an NoC simulator on a hardware platform consisting of an FPGA board and an 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 simulate all routers of the network using a single router. Software on one or both

ARM9 processors generates traffic, controls the network of routers on FPGA, and analyzes output packets. In this approach, a dedicated SoC board is required. Additionally, off-chip communication is the performance bottleneck.

Chapter 3

Proposal of Novel Emulation Methods

3.1 Emulation Model

Figure 3.1(a) shows the general emulation model containing three basic components: *router*, *traffic generator*, and *traffic sink*. In direct networks such as meshes and tori, each node can be abstracted by a router, a traffic generator, and a traffic sink, as shown in Figure 3.1(b). In indirect networks, e.g. butterflies, each terminal node can be modeled by a traffic generator and a traffic sink while each switch node is a router.

3.1.1 Router

The primary router model in this work is the input-queued pipelined VC router. Chapter 4 will demonstrate emulations of the canonical input-queued VC router which has been described in Section 2.2.4. Two router pipeline structures will be analyzed:

(1) Five-stage pipeline structure: the five pipeline stages consist of routing computation, VC allocation, switch allocation, switch traversal, and link traversal.

(2) Four-stage pipeline structure: the look-ahead routing technique is used to perform routing computation and VC allocation in parallel, thereby reducing the number of pipeline stages from five to four.

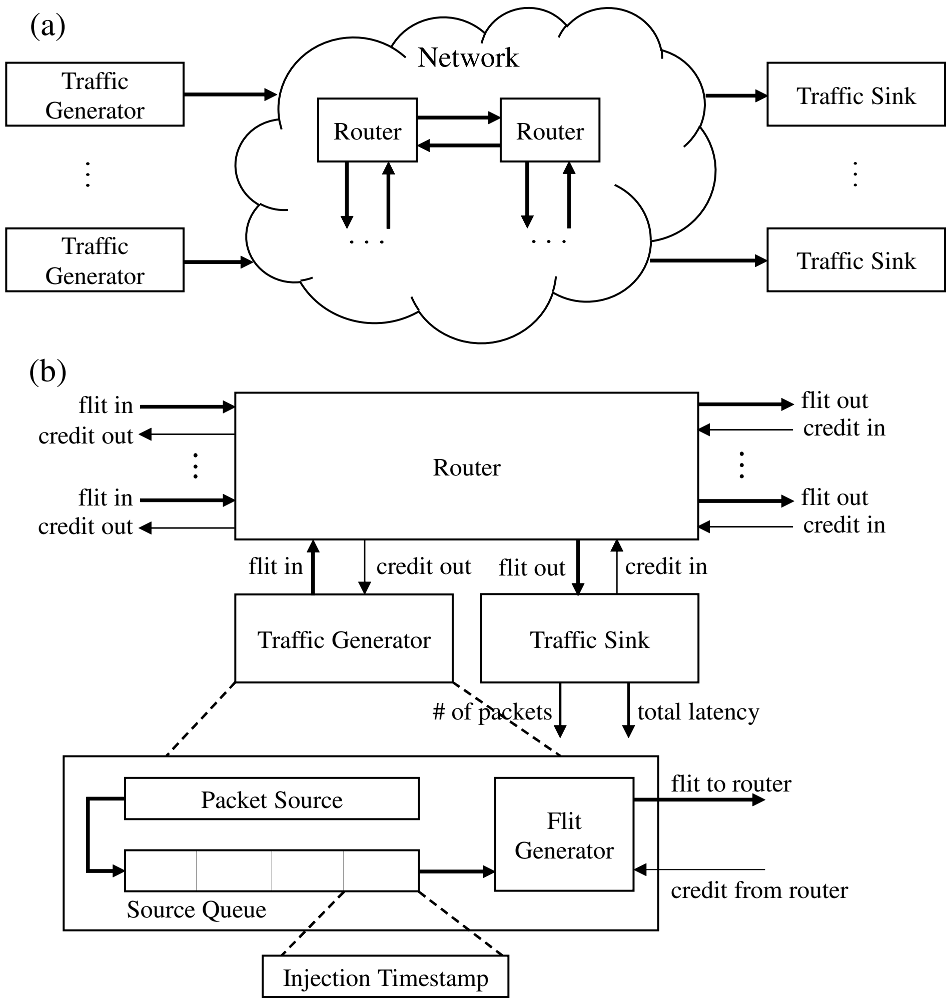


Figure 3.1: (a) General emulation model. (b) Architecture of each node in direct networks (such as meshes and tori).

While some previous work simplifies the emulated router architectures to reduce the hardware resource usage, the methods proposed in this thesis allow us to emulate NoC designs with up to thousands of nodes without any compromise on emulation accuracy.

3.1.2 Traffic Generator

Traffic generators receive flow control credits and send back generated flits to the network. As shown in Figure 3.1(b), each traffic generator is composed of a *packet source*, a *source queue*, and a *flit generator*.

Packet sources model injection processes of synthetic workloads. There are three typical types of injection processes: periodic process, Bernoulli process, and Markov modulated process. In a periodic process, the period T between injections is a constant. Therefore, a traffic generator does not need a large source queue to store all packets generated by the packet source. An 1-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.

Bernoulli processes and Markov modulated processes incorporate randomness into the injection process. Bernoulli processes are the most common injection processes used in NoC simulations. In a Bernoulli process, the probability of injecting a packet is equal to the packet injection rate. In addition to the randomness in injecting packets, a Markov modulated process can model time-varying traffic using a Markov chain.

Because of the randomness, every Bernoulli and Markov modulated process requires a large source queue between each packet source and the network to accurately model specified synthetic workloads. In Section 3.2, a novel method is proposed for eliminating this memory constraint. With the proposed method, NoC designs can be accurately emulated under synthetic workloads with Bernoulli and Markov modulated injection processes while using small source queues.

In the conventional NoC simulation/emulation model, each packet source generates and pushes packets to the corresponding source queue. In our model, the packet sources generate packets' injection timestamps which are the only information needed to be stored in the source queues. Each injection timestamp indicates the time at which a packet was created. These timestamps are stored in the source queues until they can be used to generate flits to be injected into the network. The destination address and the length of each packet are determined at the flit generator.

Flit model. As shown in Figure 3.2, each flit is composed of five fields: valid (one bit), type (two bits), VC (x bits), look-ahead routing information (y bits), and data (z bits). The valid bit determines whether the flit exists. The 2-bit type defines the type of flit (head, body, or tail). The x-bit VC determines which VC (virtual channel) the flit is stored into. The value

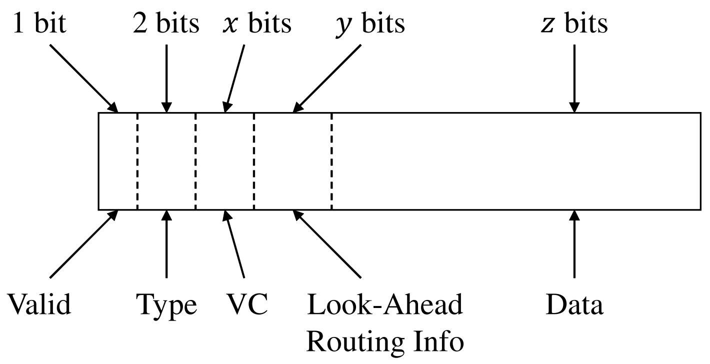


Figure 3.2: The flit structure used in the emulation model.

of x depends on the number of VCs per port. If the emulated router architecture employs look-ahead routing, we need y bits for storing routing information passed between routers. Otherwise, y is equal to zero. Finally, z-bit data is the data carried by the flit.

The routing information of a packet is calculated based on the z-bit destination address attached at the head flit. To address all nodes in an 128×128 mesh network (16,384 nodes), z must be greater than or equal to 14 ($2^{14} = 16,384$).

The injection timestamp of a packet, the time at which the packet was created and is used to calculate the latency of the packet, is divided into multiple parts. The current implementation divides each injection timestamp into two parts. One is attached at the tail flit. The other is carried by the last body flit which is adjacent to the tail flit. As a result, we need a way to separate the body flit that carries a part of the injection timestamp from other body flits. As shown in Figure 3.2, we use two bits for determining the type of a flit (head, body, or tail). A simple separation method is described below.

- 2'b10: a head flit.
- 2'b00: a body flit that does not carry any part the injection timestamp of the packet.
- 2'b11: a body flit that carries a part of the injection timestamp of the packet.
- 2'b01: a tail flit.

Because the packet length is typically greater than two, using two flits to carry an injection timestamp is not a limitation. In contrast, this approach allows us to increase the number of simulation cycles while using a small flit size. In general, the maximum number of simulation cycles depends on the timestamps' bit width. In particular, with a naive implementation, using t-bit timestamps allows us to run 2^t simulation cycles. Besides, the flit size directly affect the amount of memory required for the routers' FIFO buffers. Therefore, if each timestamp is carried by only one flit, increasing the maximum number of simulation cycles will require significantly more memory to implement the routers' FIFO buffers. On the other hand, by using multiple flits to carry a timestamp, a relatively small flit size can be used to realize a large enough number of simulation cycles. For example, if each timestamp is divided into two parts, each is attached to a flit, 2^{2z} simulation cycles, where z is the bit width of the data field of each flit, can be realized. When z is equal to 14, the minimum bit width that can be used to address all nodes of of an 128×128 mesh network, the maximum number of simulation cycles is equal to 2^{28} (268,435,456). This is enough for simulating NoC designs with up to thousands of nodes.

3.1.3 Traffic Sink

Traffic sinks are responsible for ejecting packets from their destinations and collecting performance characteristics of the emulated NoC design using some statistic counters. When a packet arrives its destination, the corresponding traffic sink calculates the latency of the packet based on the injection timestamp attached at the last body flit and the tail flit, increments the packet counter, and sends back flow control credits to the network.

3.2 Decoupling Time Counters

In the conventional model of emulating NoC designs under synthetic workloads, a large source queue is used between each packet source and the network to remove the feedback of the network to the injection process. Every packet generated by packet source i is stored in the corresponding source queue i until it can enter the network.

Ideally, the length of every source queue must be infinite. However, in practice, since the number of emulation cycles is finite, we only have to ensure that all source queues will not become full during the emulation. The length of each source queue thus can be finite. However,

```
1: network.time++
2: if squeue.empty() then
3:     gen ← false
4:     while !gen & psource.time ≤ network.time do
5:         if rand() < THRESHOLD then
6:             packet ← generate_packet()
7:             squeue.push(packet)
8:             gen ← true
9:         end if
10:        psource.time++
11:    end while
12: end if
```

Figure 3.3: Algorithm for simulating each packet source and the corresponding source queue with Bernoulli process in BookSim. This code is executed at every simulation cycle.

it is generally very large to properly emulate network behavior at high traffic injection rates. Also, longer emulation time will require larger source queues when the traffic injection rate 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. On the other hand, FPGA-based NoC emulators cannot directly use such approach. Papamichael [17] uses off-chip DRAM for storing all traffic data and Microblaze soft processor for controlling the traffic injection process by software. Wolkotte et al. [18] use ARM9 processor to implement the traffic generators by software.

Contrary to other software-based NoC simulators, BookSim [10] uses a different approach which has been discussed by Dally and Towles [2]. Figure 3.3 shows the algorithm for simulating each packet source and the corresponding source queue with Bernoulli process in BookSim. The time counter of the network (*network.time* in Figure 3.3) is separated from the time counters of the packet sources (*psource.time* in Figure 3.3). Also, a packet source advances only when its corresponding source queue (*squeue* in Figure 3.3) is empty. In particular, at a simulation cycle, if the source queue becomes empty, the packet source will run until a packet is generated or until its time counter is equal to the network's time counter. Using this algorithm, each source queue always contains at most one packet. Thus, only 1-entry source queues are required.

However, it is difficult to implement the algorithm described in Figure 3.3 on FPGAs due to the following reason. If the network and the packet source are synchronized by the same clock, it is impossible to execute the while loop from line 4 to line 11 in Figure 3.3 in one clock cycle because the packet source can advance only one step per clock cycle. One possible solution is to synchronize the packet source by a much faster clock. However, it is extremely hard to realize that solution because of the limitation of increasing clock frequency in FPGAs and the difficulty in determining the upper bound of the number of iterations in the while loop.

This section proposes a novel method for accurately emulating NoC designs under synthetic workloads without using large source queues. This method enhances the idea of separating the time counter of the network and the time counters of the packet sources. By decoupling the time counters, we allow each packet source and the network to have two states: running and waiting. The state transitions are based on the status of the source queues and the relationship between the time counters. The method is described in detail below.

First of all, the feedback of the network can temporarily affect the packet sources. When a packet source is going to generate and inject a packet into the corresponding source queue, it checks the status of the source queue. If the source queue is full due to the congestion of the network, the packet source will wait until there is one space in the source queue to insert the new packet. This may take several cycles depending on the congestion level of the network.

Figure 3.4 shows the timeline of the network and a packet source. The enqueue process describes how packets are injected into the source queue while the dequeue process is about ejecting packets from the source queue. The role of the packet source is to generate and inject packets into the source queue according to the enqueue process. For a given synthetic workload and a given NoC design, both the enqueue and dequeue process are deterministic since we are using pseudo-random number generators.

In Figure 3.4, the source queue of four entries becomes full at time T_0. At time T_1, the packet source is going to generate and inject packet P4 into the source queue according to the

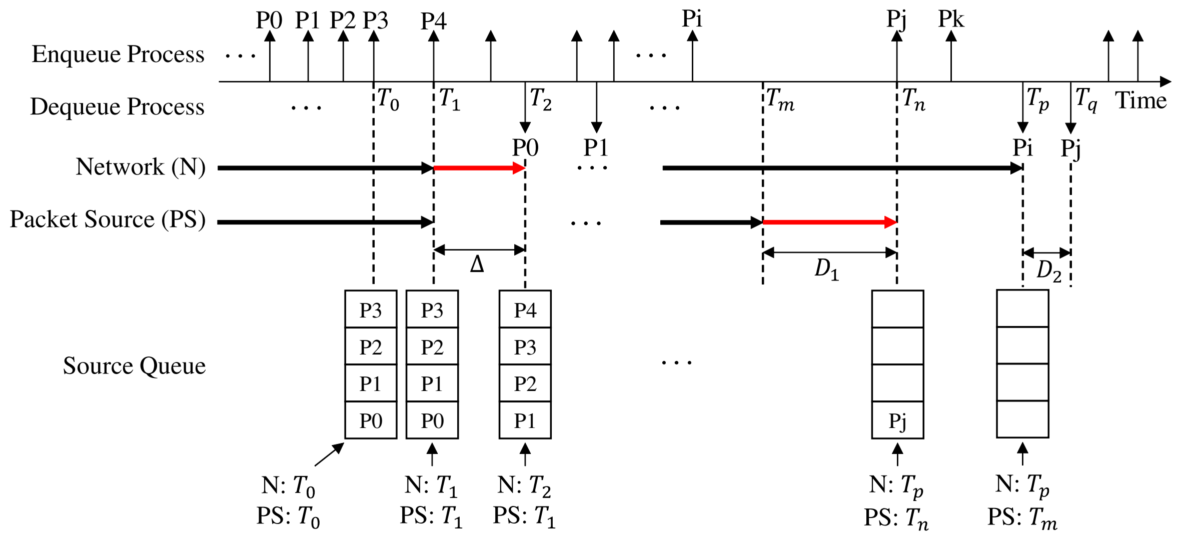


Figure 3.4: Timeline of a packet source and the network. The packet source is allowed to run behind the network. The state of the source queue is determined by both the network's time and the packet source's time. For example, (N: T_p, PS: T_n) indicates the state of the source queue when the network is at time T_p and the packet source is at time T_n.

enqueue process. However, it has to wait until there is a free space in the source queue. At time T_2, packet P0 is ejected from the source queue. The packet source can now generate and inject packet P4. It then continues execution from time T_1, and thus is behind the network.

Case 1: If all source queues never become full, the packet sources do not have to stop working any time. Hence, the emulation is correct. This case happens when the traffic injection rate is low. The feedback of the network does not affect the packet sources.

Case 2: If each source queue always contains at least one packet after the first time it becomes full, correct emulation is also achieved. This case happens when the traffic injection rate is higher than a threshold value. For a given synthetic workload, this value depends on the length of the source queue and the emulated NoC design. In this case, the feedback of the network does affect the packet sources. However, the effect does not cause any problem because it is ensured that all packets are generated on time.

Case 3: If the traffic injection rate is not high as the previous case but still can make source queues become full, the emulation may be not correct. This can be seen in the example in Figure 3.4. Here, we consider only one packet source. The same discussion can be applied to other packet sources with the note that each packet source has a different time counter. 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, Δ will become larger and larger as time goes on. When Δ is large enough, the following problem can occur. Suppose that the source queue becomes empty after the network takes packet Pi at time T_p and the packet source has reached time T_m. Also, at time T_q, the network will take packet Pj, which will be generated by the packet source at time T_n, from the source queue. Let $D_1 = T_n - T_m$ and $D_2 = T_q - T_p$. If $D_1 > D_2$, the network will reach time T_q before packet Pj is generated and injected into the source queue by the packet source. In other words, the packet source is too slow to properly emulate the specified synthetic workload. To overcome this problem, we force the network to stop all of its operations when both of the 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. As shown in Figure 3.4, when the source queue becomes empty at time T_p, the network is stalled because the time counter of the packet source is smaller than the time counter of the network. At time T_n, packet Pj is generated and injected into the source queue by the packet source. The network can now return to its normal operations. By this way, we can ensure that either the packet source's time is same as the network's time or the source queue contains at least one packet. Thus, the specified synthetic workload is emulated accurately.

In the first case and the second case, it is ensured that the network is not stalled. Thus, the emulation can take place without any penalty cycle. On the other hand, the network may be stalled many times in the third case. For a given NoC design and a given injection process, 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. Chapter 4 will evaluate the effect of the proposed method to performance of the implemented NoC emulator.

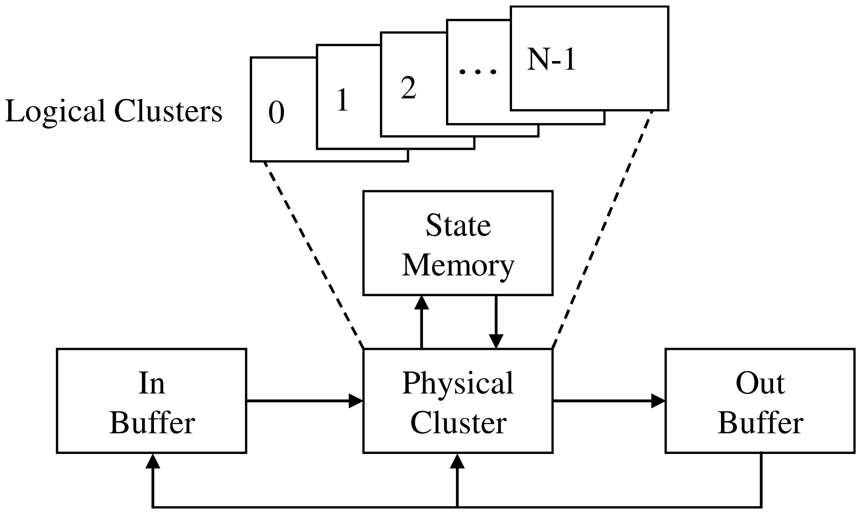


Figure 3.5: High-level datapath of the NoC emulator.

3.3 Novel Use of Time-Division Multiplexing

The straightforward way to emulate an NoC design on an FPGA is to fully replicate the nodes and connect them into the network. However, such direct approach is very expensive. Using the TDM technique is an approach to reduce the FPGA resource usage.

3.3.1 High-Level Datapath

Figure 3.5 shows the high-level datapath of the proposed NoC emulator composed of four components: *physical cluster*, *state memory*, *out buffer*, and *in buffer*. The physical cluster is a group of physical nodes used to emulate the behavior of the entire network. This work focuses on direct networks in which every node is both a terminal and a switch. For indirect networks, two physical clusters are needed: one for terminal nodes, and the other for switch nodes.

Many direct networks such as meshes and tori can be emulated using multiple interconnected physical nodes. Figure 3.6 shows an example where a 4×4 mesh NoC is emulated by using (a) one physical node, (b) two physical nodes, or (c) four physical nodes. To complete one emulation cycle, the physical cluster sequentially emulates a number of *logical clusters*. Larger physical cluster will reduce the number of logical clusters, and hence improve the emulation speed.

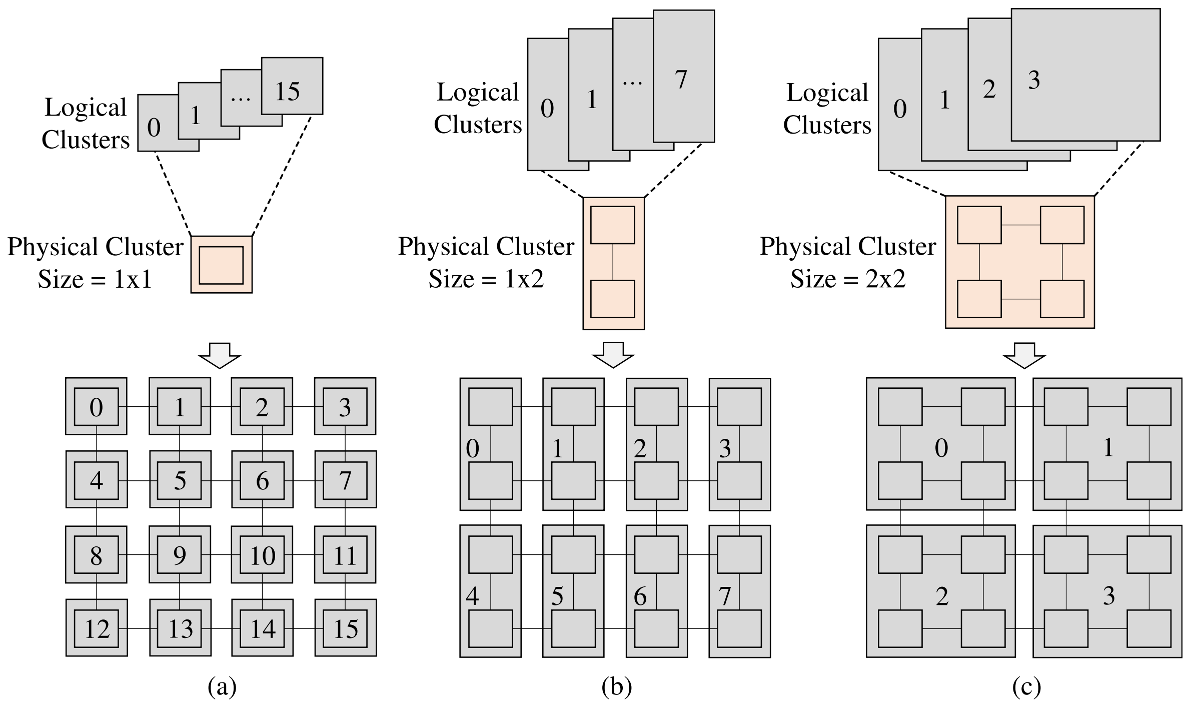


Figure 3.6: A 4×4 mesh NoC emulated by using (a) one physical node, (b) two physical nodes, or (c) four physical nodes.

The state memory stores 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 of the NoC emulator.

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. We store the data produced by all logical clusters in the out buffer. For example, in Figure 3.6(a), logical cluster 6 may communicate with logical cluster 2, 5, 7, and 10. In particular, the output data of logical cluster 6 in emulation cycle $i-1$ is the input data of logical cluster 2, 5, 7, and 10 in emulation cycle i.

However, using only the out buffer is not sufficient. Suppose that logical cluster 1 is emulated after logical cluster 0 and logical cluster 1 depends on the output data of logical cluster 0. Let d_0^i be the output data of logical cluster 0 after emulation cycle $i-1$. In emulation cycle i, logical cluster 1 uses d_0^i. If only the out buffer is used, d_0^i will be overwritten by d_0^{i+1} before being used by logical cluster 1 because logical cluster 0 is emulated before logical cluster 1. To prevent such conflicts, as shown in Figure 3.5(a), the in buffer is used.

The double-buffering scheme is also used by Papamichael [17]. However, in some cases, we do not have to copy all data from the out buffer to the in buffer. Our insight is that, before overwriting the new data of a logical cluster into the out buffer, we only copy the old data which is necessary for emulating the subsequent logical clusters. Some topologies such as mesh allow us to 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 3.6 requires us to copy only data at the east direction and the south direction instead of all four directions. Thus, the size of the in buffer can be reduced. This optimization is important because topology is a fixed component in many NoC designs.

3.3.2 Detailed Timing

As shown in Figure 3.7(a), the physical cluster contains three parts: *logic*, *routers' FIFO buffers*, and *source queues*. We divide states of the physical cluster into two groups: memory (routers' FIFO buffers and source queues) and register (e.g., credit counters, allocators' states). To emulate N logical clusters, a memory of k-entry is enlarged to $k \times N$-entry, each series of k-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 which is 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, we consider the set of registers of a logical cluster 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, we use BRAMs much more extensively than other FPGA resources (registers, LUTs). For instance, as will be seen in Chapter 4, 79% of BRAMs are required when emulat-

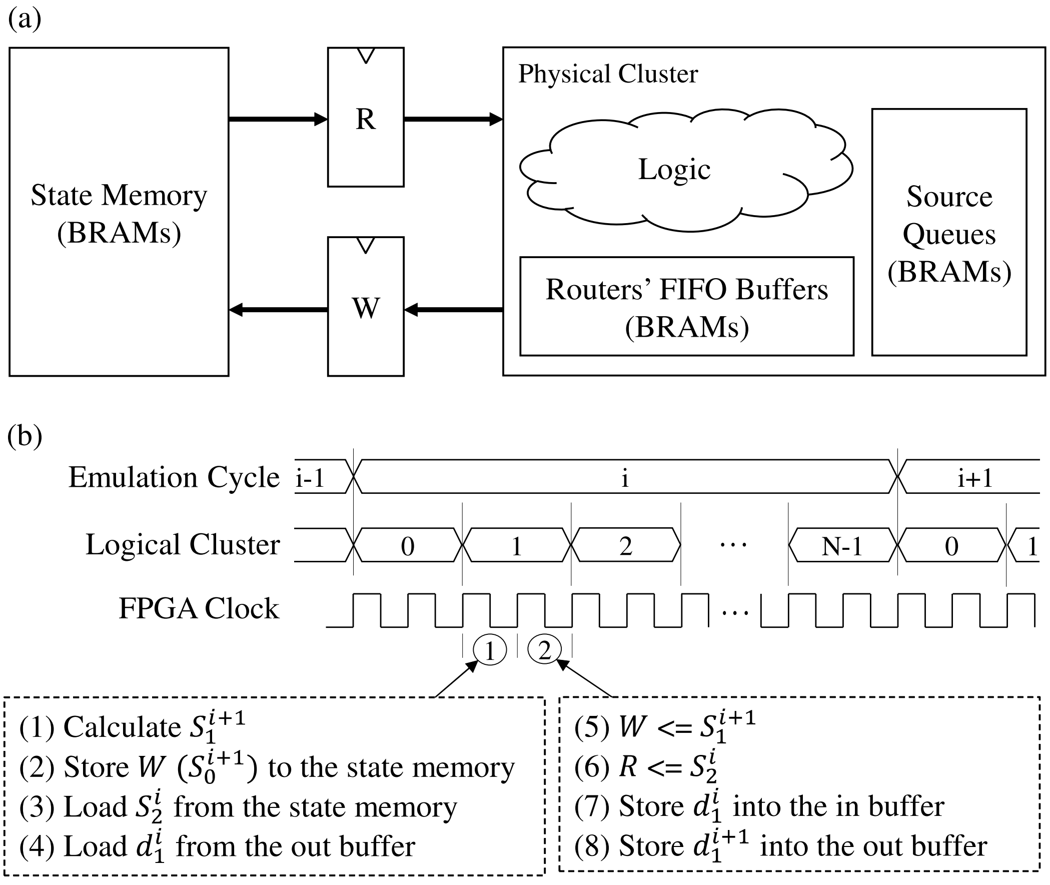

(a)
State Memory (BRAMs)
R
W
Physical Cluster
Logic
Routers' FIFO Buffers (BRAMs)
Source Queues (BRAMs)
(b)
Emulation Cycle
i-1
i
i+1
Logical Cluster
0
1
2
N-1
0
1
FPGA Clock
1
2
(1) Calculate S_1^{i+1}
(2) Store W (S_0^{i+1}) to the state memory
(3) Load S_2^i from the state memory
(4) Load d_1^i from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

Figure 3.7: (a) Using register R and register W between the state memory and the physical cluster to make the routing task easier. (b) Timing diagram of the NoC emulator when using TDM. S_j^i and d_j^i are the state and the output 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.

ing an 128×128 mesh NoC design with a canonical input-queued 5-stage pipelined VC router using four physical nodes (physical cluster size = 2×2) while only 1% of slice registers and 4% of slice LUTs are occupied. Figure 3.8 shows how a Virtex-7 FPGA is actually implemented. BRAMs in the Virtex-7 architecture are typically arranged in parallel columns. As shown in Figure 3.8, the design is spread out according to the distribution of BRAMs. This problem makes the routing task much more difficult. Critical paths in the design are the paths between BRAMs (e.g. a path between the state memory and a FIFO buffer of a router). The timing of the design is dominated by the net delay, not by the logic delay.

To overcome the above problem, as shown in Figure 3.7(a), we insert register R and register

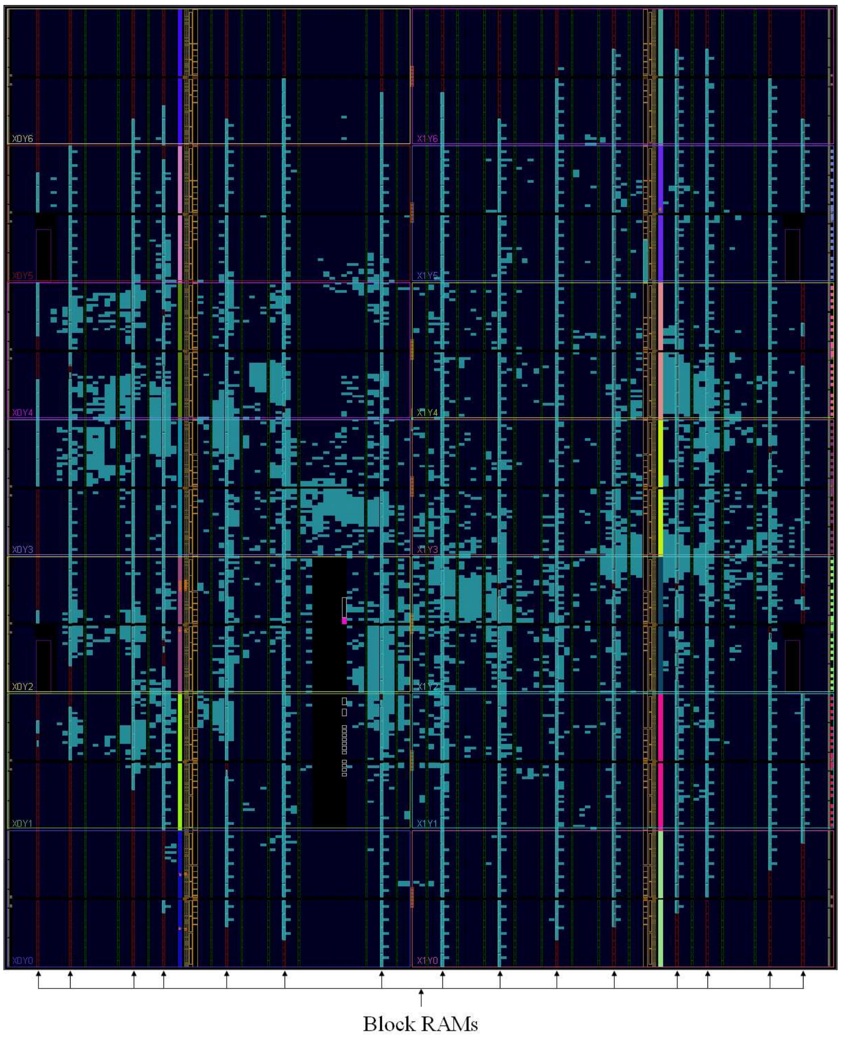


Figure 3.8: FPGA resource usage when emulating an 128×128 mesh NoC design with 5-stage pipelined VC router (2 VCs / port) using 4 physical nodes.

W between the state memory and the physical cluster. The detailed timing is described below.

Figure 3.7(b) shows the timing diagram of the proposed NoC emulator. *N* here is the number of logical clusters. Two FPGA cycles are used to emulate 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 becomes available, and thus is stored into the out buffer. On the other hand, the old data is (4) loaded from the out buffer at the first FPGA cycle and (7) stored into the in buffer at the second FPGA cycle. The next logical clusters in the emulation sequence will retrieve the current cluster's data from the in buffer. After the emulation of logical cluster $N-1$, the emulation cycle is incremented.

Although using two FPGA cycles for emulating each logical cluster may decrease the simulation 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 cluster in the emulation sequence (logical cluster $N-1$ and 0 in Figure 3.7(b)), using one FPGA cycle for emulating each logical cluster will require complicated control logics to deal with the case where d_{N-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 becomes available at the same time that the new state S_j^{i+1} is calculated. If a part of d_j^{i+1} is not registered, using one FPGA cycle for emulating 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} when S_j^{i+1} is available.

Because 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 $2 \times N$. However, as discussed in Section 3.2, since the network may be stalled, the actual number of FPGA cycles required to emulate one cycle of the NoC may be greater than $2 \times N$.

3.3.3 Implementation of the State Memory

When emulating NoC designs with thousands of nodes, the state memory is very large. Thus, it should be implemented using BRAMs. To save the FPGA synthesis time, an auto-reset mechanism is used for automatically updating the traffic injection rate. The NoC emulator is reset every time the traffic injection rate is updated. Each reset time requires all entries of the state memory to be reset to determined values.

If we reset all entries of the state memory, the implementation is limited to discrete registers rather than either BRAMs or distributed RAMs. As a result, large multiplexers for these discrete registers are required. Discrete registers and large multiplexers not only lead to a large amount of required FPGA resources but also significantly decrease the operating frequency of the NoC emulator.

To avoid resetting all entries of the state memory when resetting the NoC emulator, an 1-bit register *init_done* is used to identify the current state of the NoC emulator. The value of this register is zero when the NoC emulator is reset and when the clock counter of the NoC emulator is zero. Whenever the value of *init_done* is zero, we ignore all states read from the state memory and use the initial states. By this way, the emulation can be performed properly without resetting the state memory. Although additional multiplexers are required, it is a necessary trade-off for implementing the state memory by BRAMs.

3.4 Emulation RTL Code Translation

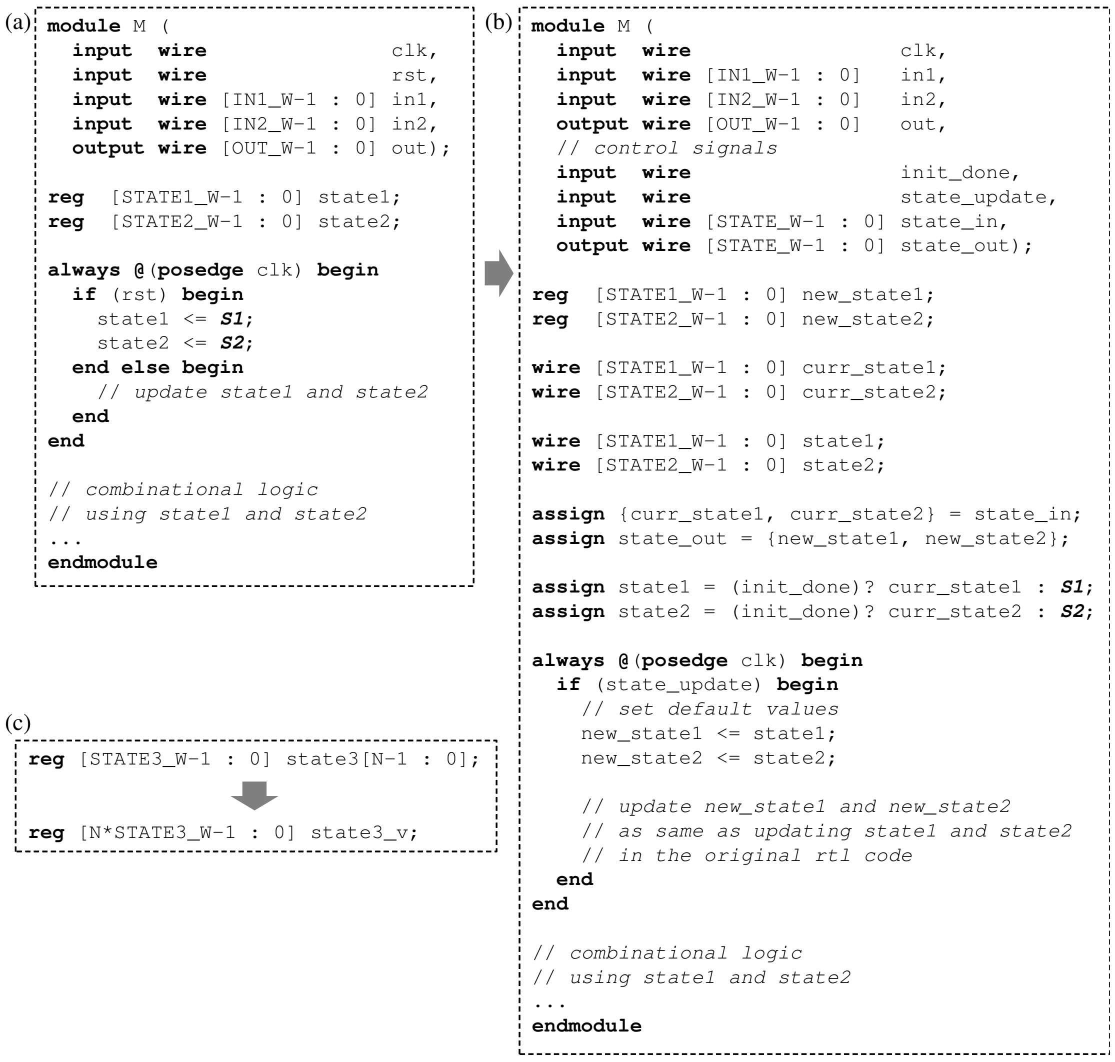
This Section will show the basic steps to translate from the original RTL code to the emulation RTL code.

3.4.1 Register

Figure 3.9 shows an example where (a) the original RTL code is translated to (b) the 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 all registers have a same standard form as *state1* and *state2*. Therefore, a register array must be converted to the standard form as shown in Figure 3.9(c).

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

(1) **init_done.** This control signal is used to avoid resetting all entries of the state memory as described in Section 3.3.3. When *init_done* is equal to zero, i.e. when the NoC emulator is being reset and when the clock counter of the NoC emulator is zero, initial states are used instead of states loaded from the state memory. With the use of *init_done*, the reset



(a)
```
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
```

(b)
```
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
```

(c)
```
reg [STATE3_W-1 : 0] state3[N-1 : 0];

reg [N*STATE3_W-1 : 0] state3_v;
```

Figure 3.9: Translating from (a) original RTL code to (b) emulation RTL code. In (c), a register array is converted to the standard form.

signal *rst* becomes unnecessary, and thus is omitted from the emulation RTL code.

(2) **state_update.** To emulate a logical cluster 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 Section 3.2. The value of *state_update* is one only at the first FPGA cycle so that each logical cluster is emulated one time per emulation cycle.

(3) **state_in.** This is the state used in the current emulation cycle and is loaded from register *R* (Figure 3.7(a)).

(4) **state_out.** This is the new state which will be stored into the state memory after being temporarily stored in register W (Figure 3.7(a)).

3.4.2 Memory

Since distributed RAMs are too area-expensive for implementing large storages, to emulate NoC designs with up to thousands of nodes, all memories (routers' FIFO buffers and source queues) should be implemented using BRAMs. As discussed in Section 3.3, to emulate N logical clusters, an original memory of k-entry needs to be enlarged to a memory of $k \times N$-entry in the emulation RTL code. In the $k \times N$-entry memory, each consecutive k entries (from the first address) are for one logical cluster. We have to add some logic for calculating the read address and write address of the new memory.

We also have to analyze the relationship between the timing of the overall design and the timing of the memory that we want to translate to emulation RTL code. Suppose that *posedge* is used in the entire design. If *posedge* is also used at the read port of the memory, the data read from the memory at emulation cycle i needs to be saved into the state memory as described in Section 3.4.1 to be used in the next emulation cycle (i + 1). On the other hand, using *negedge* at the read port does not require to save the read data into the state memory because the data read from the memory at an emulation cycle is used only at that emulation cycle.

Chapter 4

Evaluation and Analysis

4.1 Implementation

The proposed methods are adopted to build an NoC emulator on a Xilinx VC707 Evaluation Board. As discussed in Section 3.3.3, for a given NoC design, the NoC emulator emulates a range of traffic injection rates without re-synthesizing the design. The emulation results are transferred to a host PC via an RS232C interface at a data rate of 0.5 Mbps. For each traffic injection rate, the NoC emulator reports the number of emulated packets, total latency, and the number of emulated cycles.

We use Vivado 14.4.1 for synthesizing, implementing, and generating the FPGA bitstream file. The synthesis and implementation strategy are set as *Flow_PerfOptimized_High* and *Performance_Explore*, respectively.

Table 4.1 shows the configuration parameters. The NoC emulator can emulate up to 16,384 nodes connected by the 128×128 mesh topology. This chapter demonstrates emulations of two router architectures: a canonical input-queued 5-stage pipelined VC router and a canonical input-queued 4-stage pipelined VC router. The former has been described in Section 2.2.4, and is the primary router model of the NoC emulator. The latter uses the look-ahead routing technique to perform two stages routing computation and VC allocation in parallel, thereby reducing the pipeline depth from 5-stage to 4-stage. With the ability to emulate different router architectures with different pipeline structures, the proposed NoC emulator outperforms other emulators such as DART [14] which simplify the emulated router architectures.

Table 4.1: Configuration parameters

Topology	128×128 mesh
Router architecture	Input-queued VC router with or without look-ahead routing
Routing algorithm	Dimension-order (XY)
Flow control	Credit-based
Router pipeline latency	5-stage or 4-stage
VC/Switch allocator	Separable output first
Arbiter type	Fixed priority
Flit size	18-bit or 21-bit
# of VCs per port	1 or 2
VC size	4-flit
Packet length	8-flit
Injection process	Bernoulli process
Traffic pattern	Uniform random
Source queue length	8-entry

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 router employing look-ahead routing is larger than the one used in the original 5-stage router (21-bit versus 18-bit, as shown in Table 4.1).

We evaluate each router architecture in two cases: 2 VCs per port and 1 VC per port. Therefore, four router designs are studied in this evaluation.

(1) **5-stage 2-VC**: the canonical input-queued 5-stage pipelined VC router with 2 VCs per port.

(2) **5-stage 1-VC**: the canonical input-queued 5-stage pipelined VC router with 1 VC per port.

(3) **4-stage 2-VC**: the canonical input-queued 4-stage pipelined VC router employing look-ahead routing with 2 VCs per port.

Table 4.2: Implementation results with three different physical cluster sizes: 2×2 (4-phy), 4×4 (16-phy), and 8×4 (32-phy).

		4-phy		16-phy		32-phy	
		#	%	#	%	#	%
5-stage 2-VC	Slices	5,844	8%	16,533	22%	27,970	37%
	Regs	5,849	1%	21,916	4%	43,283	7%
	LUTs	12,123	4%	42,833	14%	83,784	28%
	BRAMs	813	79%	815	79%	780	76%
	Freq	100 MHz		100 MHz		100 MHz	
5-stage 1-VC	Slices	4,657	6%	13,373	18%	22,693	30%
	Regs	5,042	1%	18,513	3%	36,399	6%
	LUTs	9,608	3%	33,896	11%	66,983	22%
	BRAMs	587	57%	567	55%	572	56%
	Freq	100 MHz		100 MHz		100 MHz	
4-stage 2-VC	Slices	6,664	9%	18,144	24%	30,980	41%
	Regs	6,456	1%	23,976	4%	47,228	8%
	LUTs	13,271	4%	46,484	15%	92,260	30%
	BRAMs	913	89%	930	90%	894	87%
	Freq	100 MHz		100 MHz		100 MHz	
4-stage 1-VC	Slices	5,171	7%	14,447	19%	26,019	34%
	Regs	5,422	1%	19,890	3%	39,037	6%
	LUTs	10,683	4%	36,922	12%	74,200	24%
	BRAMs	644	63%	631	61%	668	65%
	Freq	100 MHz		100 MHz		100 MHz	

(4) **4-stage 1-VC**: the canonical input-queued 4-stage pipelined VC router employing look-ahead routing with 1 VC per port.

In all cases, the number of physical nodes used to emulate the entire 128×128 mesh network can be 4, 16, and 32 (physical cluster size = 2×2, 4×4, and 8×4, respectively).

Table 4.2 shows the implementation results. On a PC with Core i7 4770 CPU, the total time of synthesizing, implementing and generating the FPGA bitstream file is about 30 minutes when the physical cluster size is 2×2, and is still less than 80 minutes for other cases.

The implementation results show that the number of slices is roughly proportional to the number of physical nodes while there is only minor difference in the number of required BRAMs. For example, in the case of emulating the 5-stage pipelined VC router with 2 VCs per port (5-stage 2-VC in Table 4.2), about 80% of BRAMs are occupied. For a same configuration, the 4-stage pipelined VC router requires more FPGA resources than the 5-stage one because we need some additional logic and a larger flit size to perform look-ahead routing. The proposed methods allow us to use BRAMs extensively for emulating large-scale NoC designs with up to thousands of nodes on a single FPGA.

In all cases, the NoC emulator runs at 100 MHz. The novel use of the time-multiplexing technique, which has been presented in Section 3.3, makes it possible to easily achieve this high operating frequency.

4.2 Measurement Methodology

For verification, the simulation results of the proposed FPGA-based NoC emulator are compared to those of BookSim [10]. Because BookSim has been widely used in NoC research, the comparison will provide more confidence in the accuracy of the NoC emulator.

For each simulation, we simulate 100,000 warm-up cycles, 100,000 measurement cycles, and a drain phase. The warm-up phase brings the network to a steady state. All packets generated in this phase are not counted. The measurement phase is run after the warm-up phase is completed. Packets generated during this phase are referred as *measurement packets*. Finally, the drain phase is executed for all measurement packets to reach their destinations.

The average packet latency is calculated based on latencies of all measurement packets. Although packets generated during the warm-up and drain phases are not counted, they do affect the measurement by providing background traffic for measurement packets.

The drain phase may take a large number of cycles. In the worst case, it may never complete if the network is subject to starvation. To prevent such case, the simulation is forced to be terminated after the measurement phase if the average packet latency has become greater than a given threshold. In this case, the simulation is referred as an unstable simulation. Users can check whether a simulation is unstable by observing the state of an LED on the FPGA board.

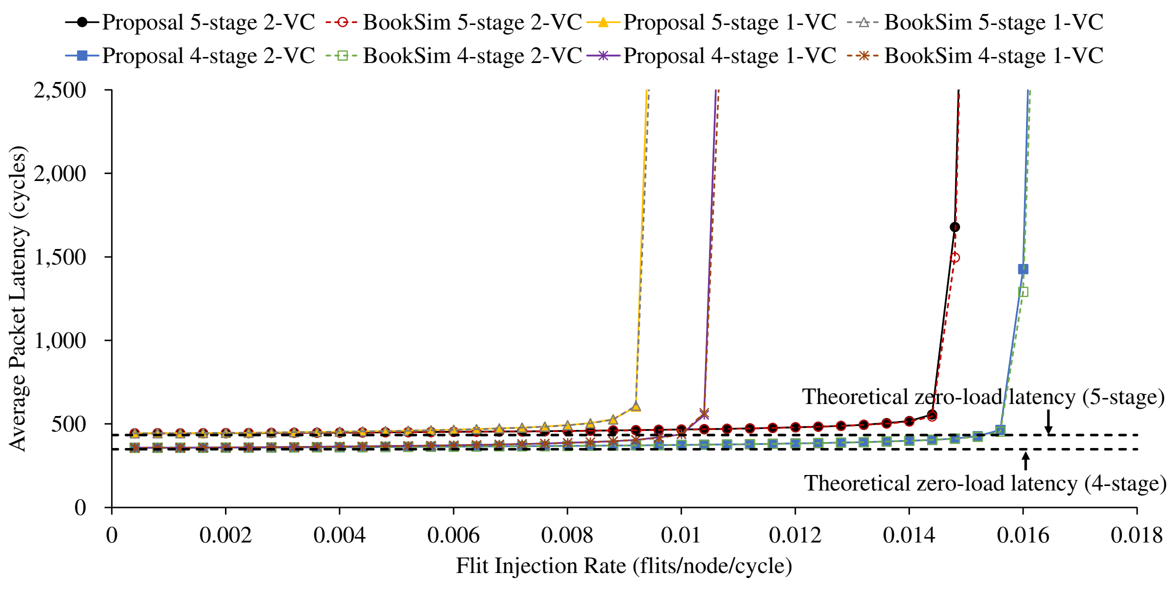


Figure 4.1: Average packet latency reported by the proposed FPGA-based NoC emulator and BookSim for four different 128×128 NoC designs described in Section 4.1.

4.3 Accuracy

First of all, the proposed methods do not affect the simulation accuracy. They do not involve any simplification of the emulated architectures. By decoupling the time counters, synthetic workloads can be modeled accurately without using a large amount of memory. By using the method based on time-multiplexing, FPGA resources can be effectively utilized. Indeed, we make no compromise in the simulation accuracy.

Figure 4.1 shows the average packet latency graphs of four different 128×128 NoC designs, which have been described in Section 4.1, with the configuration parameters shown in Table 4.1. These graphs are obtained using the proposed FPGA-based NoC emulator and BookSim.

We can see that the simulations using the NoC emulator and BookSim exhibit nearly identical average packet latency. For each NoC design, there is a minor difference between two graphs. The difference is not zero because BookSim uses Knuth's pseudo-random number generators [42] while the proposed NoC emulator employs xorshift+ pseudo-random number generators [43] which provide good statistical properties and are easy to efficiently implement on FPGA.

DART [14], a typical FPGA-based NoC emulator, has been also compared to BookSim. However, because DART models a simplified single-stage router with one output port, the resource contentions are not modeled accurately. To emulate a normal router with n input ports and n output ports by the router model of n input ports and a single output port, DART needs n clock cycles to performs the all-to-all switching between input ports and output ports. Additionally, the allocation of all resources in the single-stage router is performed in one stage, and thus is significantly different from that in a pipelined router. As a result, DART provides very different average packet latencies, especially at high injection rates, even when the routing delay, VC allocation delay, and switch allocation delay in BookSim are set appropriately so that both DART and BookSim have the same overall router latency. An important note here is that the largest NoC design which can be emulated by DART has less than 100 nodes.

The average packet latency graphs are also compared to the theoretical zero-load latencies which can be calculated based on the average minimum hop count of the network (because we use XY routing, a minimal routing algorithm), the numbers of pipeline stages of the router architectures, and the packet length. Figure 4.1 shows that the zero-load latencies obtained by emulations are almost similar to the theoretical ones.

Under the uniform random traffic, the theoretical zero-load latency of the network is given by:

$$L_{zero-load} = H_{avg} * D_{hop} + P_{len} - 1$$

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, respectively. In real emulations, several additional cycles are required because all generated packets are stored in the source queues before entering the network.

For the 2-dimensional mesh topology (k-ary n-cube where $n = 2$), the average minimum hop count can be calculated by the following equation:

$$H_{avg} = \begin{cases} \frac{nk}{3} & k \textbf{ even} \\ n(\frac{k}{3} - \frac{1}{3k}) & k \textbf{ odd} \end{cases}$$

For the 128×128 mesh topology ($H_{avg} = \frac{128*2}{3}$) and 8-flit packet length ($P_{len} = 8$), the theoretical zero-load latencies of the networks using 5-stage pipelined router ($L^{5-stage}_{zero-load}$) and 4-stage pipelined router ($L^{4-stage}_{zero-load}$) are as follows.

$$\begin{aligned}
L^{5-stage}_{zero-load} &= H_{avg} * D_{hop} + P_{len} - 1 \\
&= \frac{128 * 2}{3} * 5 + 8 - 1 \\
&= 433.7 \\
L^{4-stage}_{zero-load} &= H_{avg} * D_{hop} + P_{len} - 1 \\
&= \frac{128 * 2}{3} * 4 + 8 - 1 \\
&= 348.3
\end{aligned}$$

4.4 Simulation Performance

We measure the simulation time of BookSim on a single PC with Core i7 4770 CPU and 32GB memory for only one 128×128 NoC design (5-stage pipelined VC router with 2 VCs per port) because the simulations using BookSim are extremely slow. For other designs, we run BookSim on several PCs, and hence do not measure the simulation time. The evaluation results, however, are applicable to these and other NoC designs as well.

Figure 4.4 shows the speedup of the NoC emulator over BookSim when simulating the 128×128 mesh NoC with different physical cluster sizes and traffic injection rates. We can see that the speedup is roughly proportional to the size of the physical cluster (the number of physical nodes). However, as shown in Table 4.2, using a larger physical cluster will require more FPGA resources to implement the NoC emulator.

As discussed in Chapter 1, because a high traffic injection rate requires more tasks to be executed than a low traffic injection rate does, the simulation speed of software-based simulators in general, and BookSim in particular, decreases when the traffic injection rate increases. However, this is not true for the FPGA-based NoC emulator. Therefore, as shown in Figure 4.2, the speedup is generally higher when the traffic injection rate is higher. On the other hand, we can see that the speedup drops slightly when the traffic injection rates is greater than around 0.014 flits/node/cycle. This problem is caused by the first proposed method in which we properly stall the network to eliminate the memory constraint in modeling synthetic workloads on FPGA. It is discussed in detail below.

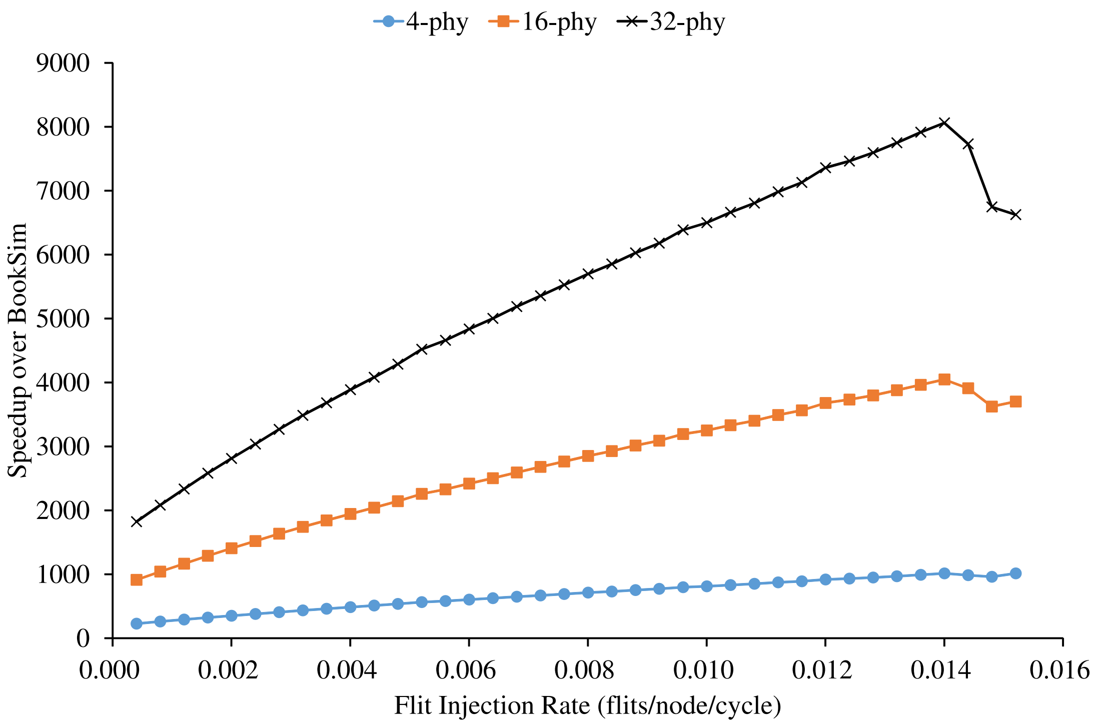


Figure 4.2: Speedup of the proposed FPGA-based NoC emulator over BookSim when simulating an 128×128 mesh NoC with different physical cluster sizes and traffic injection rates.

We can calculate the time it takes for the NoC emulator to finish a simulation (T_{sim}) (at a traffic injection rate) as the following equation:

$$T_{sim} = \frac{2 \times N \times C}{N_{phy} \times f} + T_{stalled} \quad \text{(in seconds)}$$

where N and N_{phy} is the total number of nodes and the number of physical nodes, respectively. C is the total number of cycles of the simulation. f is the operating frequency of the NoC emulator (in Hz). And finally, $T_{stalled}$ is the stalled time of the network during the simulation. The transmission time from the FPGA board to the host PC is less than 0.01 seconds, and therefore can be ignored.

Let T_{ideal} be the ideal simulation time which is achieved when source queues with infinite size can be used.

$$T_{ideal} = T_{sim} - T_{stalled} = \frac{2 \times N \times C}{N_{phy} \times f}$$

Figure 4.3 shows the graph of the ratio of the total simulation time over ideal simulation time (also in case of emulating the 5-stage pipelined VC router architecture with 2 VCs per port) at

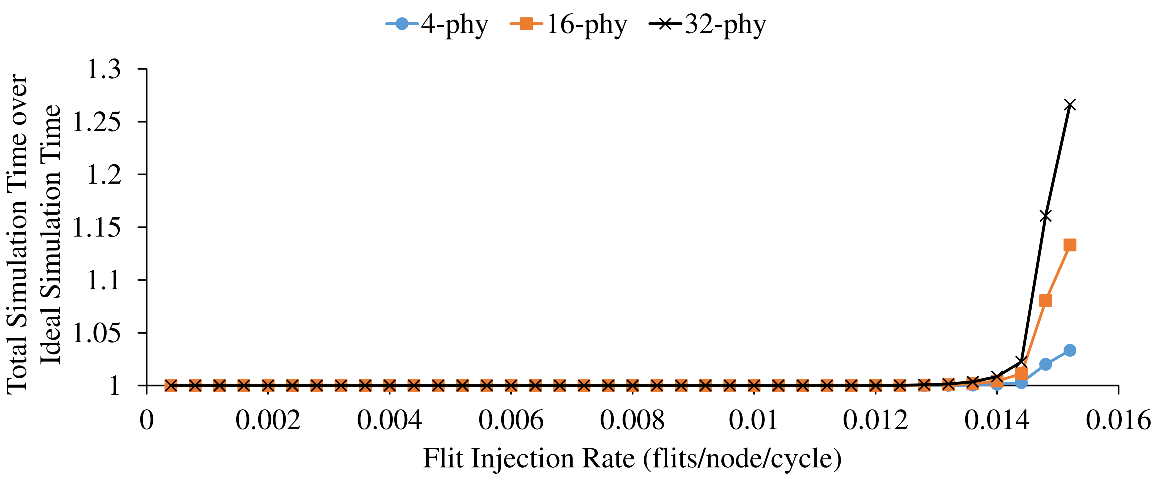


Figure 4.3: The ratio of the total simulation time over ideal simulation time (T_{sim}/T_{ideal}) at different traffic injection rates in three cases: physical cluster size = 2×2 (4-phy), 4×4 (16-phy), and 8×4 (32-phy).

different traffic injection rates in three cases: physical cluster size = 2×2, 4×4, and 8×4. For this evaluation, each source queue has a length of 8-entry.

When the size of the physical cluster increases, the number of FPGA cycles (without the stalled cycles) needed to complete each simulation, decreases. Thus, larger physical cluster will lead to smaller ideal simulation time (T_{ideal}). On the other hand, the total time in which the network is stalled does not change much with different physical cluster sizes. Therefore, as shown in Figure 4.3, the ratio (T_{sim}/T_{ideal}) is greater when the physical cluster is larger. In other words, the effect of the proposed method to simulation performance is higher when the size of the physical cluster increases. Therefore, as shown in Figure 4.2, the speedup of the proposed FPGA-based NoC emulator over BookSim at traffic injection rates greater than 0.014 flits/node/cycle drops most significantly when the physical cluster size is 32 nodes.

Figure 4.3 also shows that the ratio T_{sim}/T_{ideal} is equal to one at most traffic injection rates. Additionally, even its maximum value is still smaller than 1.3. Therefore, the actual effect of the proposed method on overall performance is small. Increasing the length of the source queues will reduce the ratio T_{sim}/T_{ideal} but also requires more BRAMs.

Figure 4.4 shows the relative speed of the proposed FPGA-based NoC emulator compared to BookSim. The speedup here is calculated as the total simulation time for all traffic injection

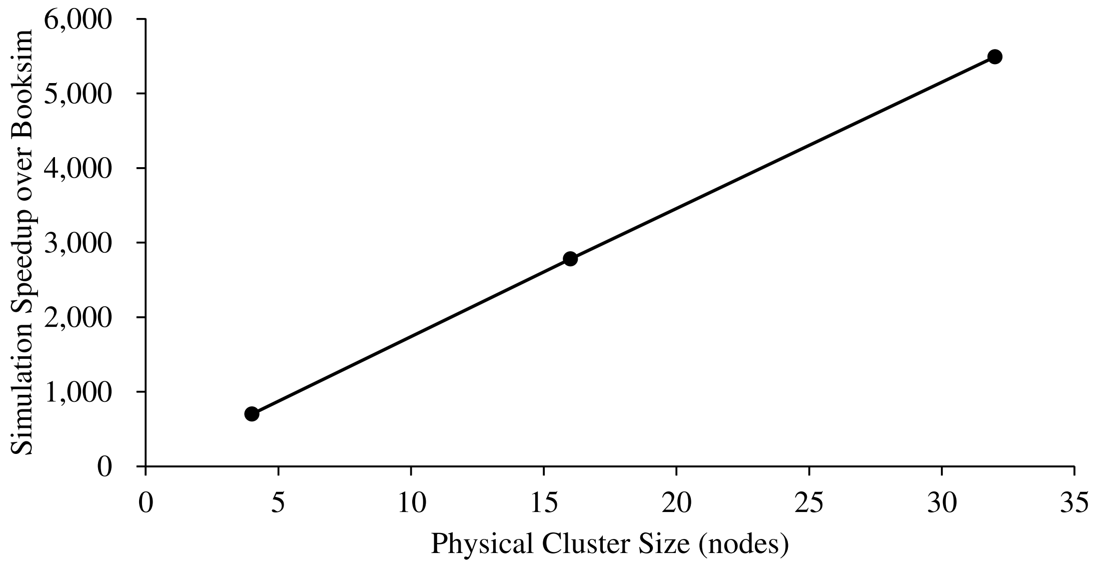


Figure 4.4: Speedup of the proposed FPGA-based NoC emulator over BookSim when simulating an 128×128 mesh NoC.

rates of BookSim divided by the one of the NoC emulator. 5,490× simulation speedup is achieved when the number of physical nodes is 32. In particular, BookSim needs more than 131 hours (about 5.5 days) to simulate the NoC design for all traffic injection rates while the total simulation time of the NoC emulator is 86.2 seconds (less than 1.5 minutes).

Chapter 5

Conclusion

This thesis presented two novel methods to enable ultra-fast and accurate emulation of large-scale NoC architectures with up to thousands of nodes on a single FPGA. The first method helps to eliminate the memory constraint, and thus helps to avoid using off-chip memory in modeling synthetic workloads for NoC emulations on FPGA. The second method leverages the time-multiplexing technique to utilize FPGA resources effectively while maintaining the simulation accuracy. The thesis also showed how the proposed methods can be applied to emulate different NoC designs. The evaluation results show that the proposed FPGA-based NoC emulator can achieve up to 5,490× simulation speedup over BookSim, a widely used software-based NoC simulator.

Because this work focuses on NoC, the communication fabric of many-core processors, simple cores, which can generate synthetic workloads such as uniform random, are modeled instead of detailed processor cores. Synthetic workloads are flexible and can be used to quickly stress NoC designs and capture their bottlenecks. However, to fully support many-core architecture and software research, it is necessary to model detailed processor cores as well as the memory hierarchy and cache, and run a wide range of realistic workloads.

My future work aims to fully support research of large-scale many-core processors by providing cycle-accurate simulation with a wide range of benchmarks/workloads and more than three orders of magnitude speedup over conventional software-based simulators.

Acknowledgement

First and foremost, I would like to thank my advisor, Prof. Kenji Kise, for his guidance and support throughout my years in graduate school at Tokyo Tech. He has helped me in so many ways. His insight and incisive comments added greatly to this thesis. Doing research with him has been an invaluable learning experience.

I also want to thank Dr. Shimpei Sato for his advice and inspiration. He encouraged me to explore and challenge further. It has been a pleasure to collaborate and co-author with him. I also thank my thesis committee members, Prof. Haruo Yokota, Prof. Jun Miyazaki, Prof. Haruhiko Kaneko, and Prof. Takuo Watanabe, for serving on my defense.

I have learned a lot from my friends at Tokyo Tech. I would like to thank all the members of the Kise Laboratory. They have made the laboratory a comfortable place to do research. I want to thank Mrs. Yukiko Asoh for helping me a lot from the beginning of my days at Tokyo Tech.

Last and most importantly, I thank my family for the sacrifices that they have made for me. Their love, support, and encouragement have been an important part of my journey at Tokyo Tech.

Bibliography

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

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

[3] N. Enright Jerger and L.-S. Peh, *On-Chip Networks*. Morgan Claypool, 2009.

[4] 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,” *SIGARCH Comp. Arch. News*, vol. 39, no. 2, pp. 1–7, 2011.

[5] A. Patel, F. Afram, S. Chen, and K. Ghose, “MARSS: A Full System Simulator for Multicore x86 CPUs,” in *Proceedings of the 38th Annual Design Automation Conference (DAC-48)*, 2011, pp. 1050–1055.

[6] P. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner, “Simics: A Full System Simulation Platform,” *Computer*, vol. 35, no. 2, pp. 50–58, 2002.

[7] P. Ren, M. Lis, M. H. Cho, K. S. Shim, C. 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*, vol. 31, no. 6, pp. 890–903, 2012.

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

[9] Y. Fu and D. Wentzlaff, “PriME: A Parallel and Distributed Simulator for Thousand-Core Chips,” in *Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 116–125.

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

[11] N. Agarwal, T. Krishna, L.-S. Peh, and N. Jha, “GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator,” in *Proceedings of the 2009 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009, pp. 33–42.

[12] Noxim. [Online]. Available: http://noxim.sourceforge.net

[13] V. Puente, J. Gregorio, and R. Beivide, “SICOSYS: An Integrated Framework for Studying Interconnection Network Performance in Multiprocessor Systems,” in *Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing*, 2002, pp. 15–22.

[14] D. Wang, C. Lo, J. Vasiljevic, N. E. Jerger, and J. G. Steffan, “DART: A Programmable Architecture for NoC Simulation on FPGAs,” *IEEE Transactions on Computers*, vol. 63, no. 3, pp. 664–678, 2014.

[15] M. Papamichael, J. 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 IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, 2011, pp. 137–144.

[16] A. Nejad, M. Martinez, and K. Goossens, “An FPGA Bridge Preserving Traffic Quality of Service for On-Chip Network-Based Systems,” in *Proceedings of the Design, Automation and Test in Europe (DATE)*, 2011, pp. 1–6.

[17] M. 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.

[18] P. Wolkotte, P. HoÌĹlzenspies, and G. J. M. Smit, “Fast, Accurate and Detailed NoC Simulations,” in *Proceedings of the 1st IEEE/ACM International Symposium on Networks-on-Chip (NOCS)*, 2007, pp. 323–332.

[19] D. Park, S. Eachempati, R. Das, A. K. Mishra, Y. Xie, N. Vijaykrishnan, and C. R. Das, “MIRA: A Multi-Layered On-Chip Interconnect Router Architecture,” in *Proceedings of the 35th Annual International Symposium on Computer Architecture (ISCA-35)*, 2008, pp. 251–261.

[20] TILE-Gx72. [Online]. Available: http://www.tilera.com/products/?ezchip=585&spage=618

[21] TILE-MX. [Online]. Available: http://www.tilera.com/products/?ezchip=585&spage=686

[22] Intel Xeon Phi Product Family. [Online]. Available: http://www.intel.com/content/www/us/en/processors/xeon/xeon-phi-detail.html

[23] G. Chrysos, “Knights Corner, Intel’s First Many Integrated Core (MIC) Architecture Product,” in *Hot Chips*, 2012.

[24] R. Hazra, “Accelerating Insights in The Technical Computing Transformation,” in *International Supercomputing Conference*, 2014.

[25] TOP500 List June 2015. [Online]. Available: http://www.top500.org/lists/2015/06/

[26] S. Borkar, “Thousand Core Chips: A Technology Perspective,” in *Proceedings of the 44th Annual Design Automation Conference (DAC-44)*, 2007, pp. 746–749.

[27] 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 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 477–488.

[28] D. Sanchez and C. Kozyrakis, "SCD: A Scalable Coherence Directory with Flexible Sharer Set Encoding," in *Proceedings of the 18th International Symposium on High Performance Computer Architecture (HPCA-18)*, 2012, pp. 1–12.

[29] J. Kim, W. J. Dally, and D. Abts, "Flattened Butterfly: A Cost-efficient Topology for High-radix Networks," in *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA-34)*, 2007, pp. 126–137.

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

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

[32] 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-7)*, 2001, pp. 255–266.

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

[34] 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-34)*, 2007, pp. 150–161.

[35] 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.

[36] 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*, vol. 60, no. 6, pp. 783–799, 2011.

[37] Open-Source Network-on-Chip Router RTL. [Online]. Available: http://nocs.stanford.edu/cgi-bin/trac.cgi/wiki/Resources/Router

[38] 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-47)*, 2010, pp. 463–468.

[39] 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-17)*, 2011, pp. 406–417.

[40] 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.

[41] A. Khan, M. Vijayaraghavan, S. Boyd-Wickizer, and Arvind, “Fast and Cycle-Accurate Modeling of A Multicore Processor,” in *Proceedings of the 2012 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2012, pp. 178–187.

[42] D. E. Knuth, *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[43] S. Vigna, “Further scramblings of Marsaglia’s xorshift generators,” *CoRR*, 2014.

Publication

Conference Papers

1. **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. 343–350*, London, UK, September 2015.

2. **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.

3. **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.

Posters

1. Hiroshi Nakatsuka, Yuichiro Tanaka, **Thiem Van Chu**, Shinya Takamaeda-Yamazaki, and Kenji Kise, "Ultrasmall: The Smallest MIPS Soft Processor", *the 24th International Conference on Field-Programmable Logic and Applications (FPL)*, Munich, Germany, September 2014.

Others

1. **Thiem Van Chu** and Kenji Kise, "A Novel Time-Division Multiplexing Approach for Emulating NoC Architectures on FPGAs", *the 77th National Convention of Information Processing Society of Japan*, March 2015. (**Honorable Mention**)

2. **Thiem Van Chu**, Shimpei Sato, and Kenji Kise, "Challenge for Ultrafast 10K-Node NoC emulation on FPGA", *IEICE Technical Reports RECONF2014-21, vol. 114, no. 223, pp. 23–28*, September 2014.