An Effective Architecture for Trace-Driven Emulation of Networks-on-Chip on FPGAs

Thiem Van Chu*† and Kenji Kise*
*Tokyo Institute of Technology, †JSPS Research Fellow
thiem@arch.cs.titech.ac.jp, kise@c.titech.ac.jp

***Abstract*—Modern many-core systems use Networks-on-Chip (NoCs) to move data around their cores. As the number of cores increases, the overall performance becomes highly sensitive to the NoC performance. Research and development of NoCs thus play a key role in designing future systems with hundreds to thousands of cores. However, current methodologies for evaluating NoCs are not scalable with respect to the system complexity. Conventional software simulators are too slow for evaluating middle- and large-scale NoCs. Recent FPGA-based emulators provide promising emulation speedups over software simulators. However, emulating large-scale NoCs with hundreds to thousands of nodes on FPGAs is a challenging problem because of the FPGA capacity constraints. Moreover, supporting trace-driven emulation is not trivial because trace data must be stored outside of the FPGA (usually in off-chip DRAM). Most of the existing FPGA-based NoC emulators rely on soft processors like Microblaze or hard processors on SoC FPGAs for loading trace data from the off-chip memory, generating messages, injecting the messages to the target NoC, manipulating the emulation, and making sure that there is no timing error. This approach makes the implementation easy but drastically degrades the emulation speed. This paper proposes an effective architecture for trace-driven emulation of NoCs on FPGAs. We present methods to scale to large NoCs and effectively hide the off-chip memory access latency. Our evaluation results show that (1) the proposal achieves a speedup of 260× compared to BookSim, one of the most widely used NoC simulators, when emulating an 8×8 NoC with trace data collected from full-system simulation of the PARSEC benchmark suite; and (2) the speedup is increased to three orders of magnitude when emulating a 64×64 NoC.**

I. Introduction

Networks-on-Chip (NoCs) are the critical components of modern many-core systems. They are proposed to replace traditional interconnects such as buses and crossbars that have been widely used in systems with a small number of cores. As the core count increases, designing NoCs to meet requirements in various aspects such as communication bandwidth and latency becomes an intrinsically important issue.

Traditionally, NoC designers rely on simulation to test their ideas and make design decisions. Existing simulation methodologies can be classified into three categories [1]: execution-driven simulation, simulation with synthetic workloads, and trace-driven simulation.

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

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

Most of the existing NoC evaluation environments are software-based simulators. Some of the most popular ones include BookSim [2], GARNET [3], and Noxim [4]. Despite being much faster than full-system simulators like gem5 [5], these systems are still too slow to simulate middle- and large-scale NoCs in practical time. For instance, our evaluation results show that the speed of BookSim is around 40K–80K simulation cycles per second when simulating an 8×8 NoC and is reduced to just 50–180 simulation cycles per second when simulating a 64×64 NoC. At this speed, running a trace-driven simulation would take an excessive amount of time because the length of a meaningful trace is typically over billions of cycles and often increases with increasing the network size. Existing parallelization techniques can be used to improve the simulation speed but only to a very limited degree because of the high synchronization cost caused by the communication-intensive nature of NoCs [6].

To address the simulation speed problem, some FPGA-based NoC emulators have been proposed [7]–[14]. A problem of the FPGA-based approach is that the FPGA capacity constraints restrict the maximum NoC size that can be emulated. For instance, the largest NoC sizes that can be emulated by DRNoC [15], AcENoCs [8], and DART [10] are 4×4, 5×5, and 9×9, respectively. Wolkotte et al. [7], Papamichael [9], Chu et al. [11], [13], and Kamali et al. [12], [14] use a technique called Time-Division Multiplexing (TDM) to scale to larger networks. In this technique, a network is emulated using a small number of nodes, and hence the amount of required FPGA resources is reduced. As a result, two state-of-the-art emulators, DuCNoC [14] and FNoC [13], can support emulation of NoCs with up to thousands of nodes. DuCNoC

can emulate an 8,196-node NoC but the emulation speed decreases dramatically with respect to the NoC size. Specifically, the authors of DuCNoC report a speedup of 66× over BookSim when emulating a 512-node NoC using 16 nodes; however, the speedup is reduced to just 3× when the NoC size is 8,192-node. This is because DuCNoC is designed on a SoC FPGA and it relies on the ARM processors of the SoC to generate traffic workloads and manipulate the emulation. On the other hand, FNoC is not based on dedicated hardware and does not rely on soft or hard processors. FNoC can emulate a 16,384-node NoC at a speedup of 5,047× over BookSim. However, FNoC does not support trace-driven emulation.

Besides the necessity of emulation with workloads created from realistic applications' trace data, it is crucial for an FPGA-based NoC emulator to support trace-driven emulation due to the following reason. Some synthetic workloads, especially those based on complex but effective synthetic workload generation methodologies like Synfull [16], are difficult to model on FPGAs because they require complicated probability density functions and modulo, multiplication, division, and exponent operations. Even it is feasible to implement these functions and operations on FPGAs, it is likely that the operating frequency of the emulator drops substantially. Therefore, if the emulator supports trace-driven emulation, developing a software tool to generate trace data based on the descriptions of the complex synthetic workloads and then using the trace-driven emulation mode of the emulator would be a better approach.

It is difficult to effectively support trace-driven NoC emulation on FPGAs because trace data are often much larger than the total capacity of FPGA on-chip memory and thus must be stored in off-chip memory. Most of the existing FPGA-based NoC emulators simplify the control of loading trace data from the off-chip memory, generating messages based on the loaded trace data, and injecting the messages to the NoC by using soft processors like Microblaze or hard processors on SoC FPGAs. AcENoCs [8] and AdapNoC [12] use two Microblaze soft processors to be able to alleviate the off-chip memory access time. Specifically, one processor is responsible for loading trace data while the other is responding for generating and injecting messages to the NoC. In this way, the trace load operation is overlapped with the message generation and injection operations, and therefore, the effect of the off-chip memory access time is reduced. DuCNoC [14] also adopts this approach but uses two ARM processors on the SoC FPGA instead of the Microblaze soft processors. As mentioned before, while using processors makes the implementation easy, it significantly degrades the emulation speed. To speed up the emulation of large-scale NoCs, a new approach is required.

In this paper, we adopt the TDM scheme introduced by Chu et al. in [13] and propose an effective architecture for speeding up trace-driven emulation of NoCs with up to thousands of nodes on FPGAs. The proposed architecture helps to virtually eliminate the negative effect of the off-chip memory on the emulation performance and allows the emulator to operate at a high frequency. We make the following major contributions.

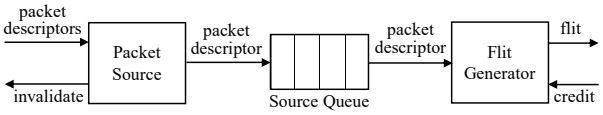


Fig. 1: Proposed traffic generator architecture.

1) Most of the existing FPGA-based NoC emulators rely on soft or hard processors to support trace-driven emulation. Because of this, their emulation speeds are limited, especially when the target NoC is large. We propose an effective architecture which does not include processors for speeding up trace-driven emulation of NoCs with up to thousands of nodes.
2) We introduce some methods to effectively hide the off-chip memory access time and improve the scalability of the emulation architecture in terms of operating frequency and FPGA resource requirements.
3) We implement a NoC emulator using the proposed architecture on a Virtex-7 FPGA. The evaluation results show that our NoC emulator is 260× faster than BookSim [2] when emulating an 8×8 NoC with trace data collected from full-system simulation of the PARSEC benchmark suite [17]; and the speedup is increased to 5,106× when emulating a 64×64 NoC with trace data created based on the uniform random traffic.

II. Background

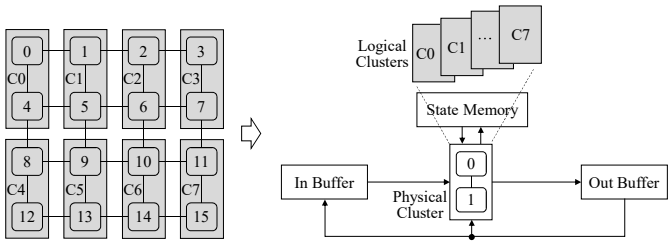
In this section, we first explain our emulation model. After that, we briefly describe the TDM scheme used by FNoC [13], which we adopt to support emulation of large-scale NoCs.

A. Emulation Model

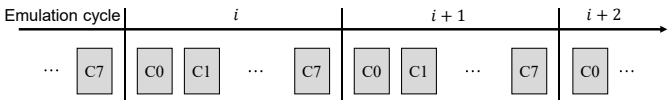
Except the modules dedicated to supporting trace-driven emulation which will be explained in Section III, there are three basic components in our emulation model: router, traffic generator, and traffic receptor. In two-dimensional NoCs which we focus in this paper, each node is modeled by a router, a traffic generator, and a traffic receptor.

1) Router: We use the conventional input-queued pipelined virtual channel (VC) router [1] with five pipeline stages (routing computation, VC allocation, switch allocation, switch traversal, and link traversal) as the primary router model because it is the baseline of most of the recently proposed ASIC-style NoC architectures. However, note that our proposed methods are independent of the emulation target NoC router; they can be integrated with FPGA-friendly routers like Hoplite [18] as well.

2) Traffic Generator: Fig. 1 shows our traffic generator architecture. The *packet source* gets trace data from a module called *trace loader* and sends back a control signal, called *invalidate*, which will be described in detail in Section III. We define a trace as a set of *packet descriptors*, each encodes a message sent from one node to another in the many-core system from which the trace was recorded. In our implementation, a packet descriptor is 64-bit length and



(a) The architecture for implementing the TDM technique in FNoC.



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

Fig. 2: FNoC's TDM scheme [13].

composed of five fields: valid bit (1 bit), packet generation timestamp (30 bits), source address (14 bits), destination address (14 bits), and packet length (5 bits). These widths limit the maximum number of emulation cycles, the largest NoC size, the maximum number of packet lengths to 2^{30}, 2^{14}, and 2^5, respectively. However, this is not a fundamental limitation; larger numbers of emulation cycles, NoC sizes, and numbers of packet lengths can be supported with only minor changes in the source code. The packet generation timestamp, source/destination addresses, and packet length are the information necessary for carrying out the trace-driven emulation. Designers can add other information to the packet descriptor structure for collecting their desired performance characteristics of the emulated NoC.

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

3) Traffic Receptor: When a packet arrives at its destination, it is forwarded to the traffic receptor. Here, the desired performance characteristics such as packet latency are collected. The traffic receptor is also responsible for sending back flow control credits to the network.

B. FNoC's TDM Scheme

Fig. 2 shows how the TDM technique is implemented in FNoC [13]. A network is emulated using a small number of interconnected nodes called *physical cluster*. A node in the physical cluster is called a *physical node*. In each time slot, the physical cluster processes a part of the network which is called a *logical cluster*. The state of each logical cluster is stored in one entry of the *state memory*. The physical cluster switches from processing a logical cluster to processing another by changing its state with the state data loaded from the state memory. The *in buffer* and the *out buffer* are responsible for storing communication data between logical clusters. FNoC uses two FPGA cycles for processing each logical cluster. Let N_{phy} be the number of physical nodes; then emulating each cycle of a NoC with N_{node} nodes costs $2N_{node}/N_{phy}$ FPGA cycles.

We adopt the above TDM scheme. The rules for creating the TDM-based RTL code are described in the original work of FNoC [11]. In our implementation, because the emulation may be stalled when the speed of loading trace data from off-chip memory cannot keep pace with the emulation speed, emulating a cycle of a N_{node}-node NoC using a cluster of N_{phy} physical nodes may take more than $2N_{node}/N_{phy}$ FPGA cycles.

III. Proposed Trace-Driven Emulation Architecture

Before going into details of the proposed architecture, we describe how a trace is organized on a host PC before it is sent to the FPGA side. As mentioned in Section II-A2, a trace is a set of packet descriptors. The order of packet descriptors in a trace is decided by first their source addresses and then their packet generation timestamps. Let p_s^t be the packet descriptor with source address s and packet generation timestamp t. For any two packet descriptors p_{s1}^{t1} and p_{s2}^{t2}, if $s1 < s2$ then p_{s2}^{t2} is behind p_{s1}^{t1}; in the case that the source addresses of the packet descriptors are the same ($s1 = s2$), they are sorted by the packet generation timestamps.

A. Architecture Overview

Fig. 3 shows the overview of the proposed trace-driven emulation architecture on a Xilinx VC707 board in which we use a 4GB DDR3 DRAM. The *trace receiver* is responsible for receiving packet descriptors from a host PC and writing them to the DRAM on the FPGA board. Packet descriptors with the same source address are stored in a continuous region of the DRAM and in the order of the packet generation timestamp. The emulation starts when the trace receiver finishes writing the whole trace data into the DRAM.

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

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

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

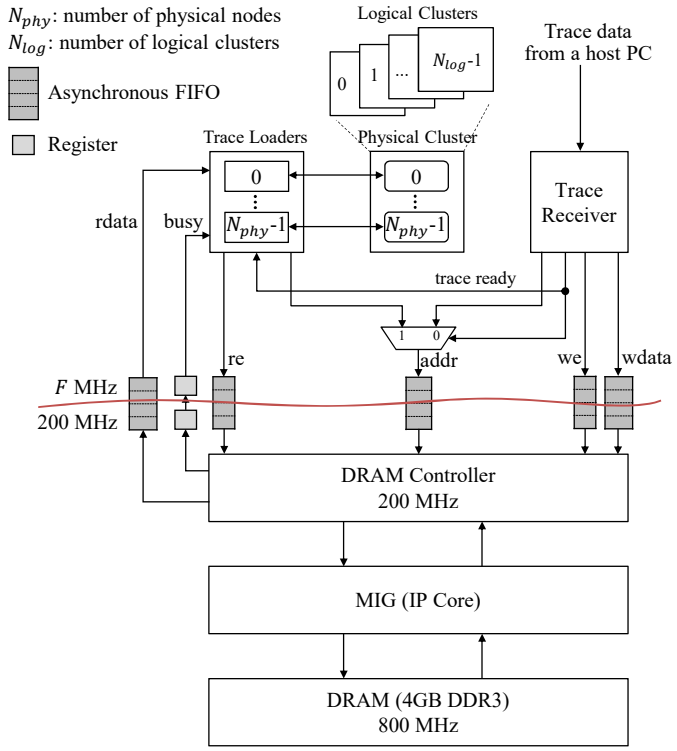


Fig. 3: Overview of the proposed trace-driven emulation architecture on a Xilinx VC707 board in which we use a 4GB DDR3 DRAM.

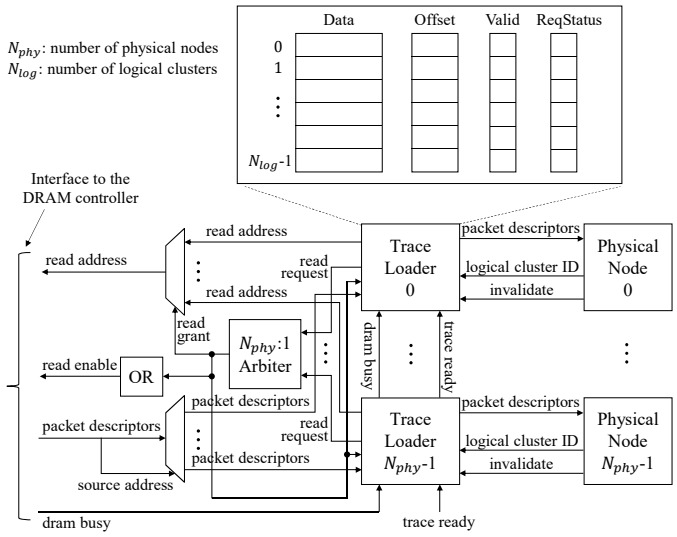


Fig. 4: Datapath surrounding the trace loaders.

B. Trace Loader Architecture

Fig. 4 shows the datapath surrounding the trace loaders. In a trace loader, there are four memory modules: *data, offset, valid*, and *reqstatus*. Each of these memories consists of N_{log} entries where N_{log} is the number of logical clusters, each entry for a node processed by the physical node which the trace data loader is connected to. For example, in Fig. 2(a), we have two physical nodes 0 and 1. Since physical node 1 processes nodes 4, 5, 6, 7, 12, 13, 14, and 15, trace loader 1 is responsible for loading packet descriptors belonging to these nodes. In trace loader 1, *data*[0], *offset*[0], *valid*[0], and *reqstatus*[0] are for node 4; *data*[1], *offset*[1], *valid*[1], and *reqstatus*[1] are for node 5; and so on.

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

Offset: x-bit-wide memory where x depends on the size of the trace. *Offset*[i] stores a relative address which is used to calculate the address for loading the next *data*[i]; the base address is determined by the ID of the node that *data*[i], *offset*[i], *valid*[i], and *reqstatus*[i] are for.

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

ReqStatus: 1-bit-wide memory. In our implementation, a trace loader is not blocked after issuing a read request to the DRAM controller. Instead of waiting for data from the DRAM controller, the trace loader continues to issue more read requests if necessary and possible. In this way, the impact of the DRAM access time on the emulation performance is effectively reduced. *ReqStatus* is used to make sure that there are no duplicate read requests. *ReqStatus*[i] indicates whether a read request for *data*[i] has been issued to the DRAM controller. *ReqStatus*[i] is set to 1 when the DRAM controller accepts the read request for *data*[i] and reset to 0 when *data*[i] is filled. The trace loader issues a read request for *data*[i] when both *valid*[i] and *reqstatus*[i] are zero.

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

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

The *reqstatus* memory has two write ports and one read port. One of the write port is for updating when the DRAM controller accepts the read request of the trace loader. The other write port is for updating when new packet descriptors

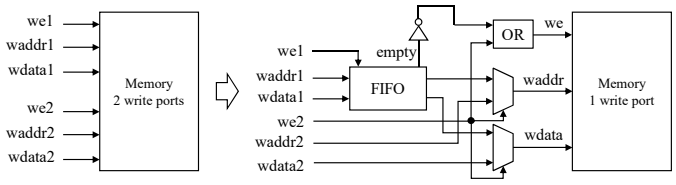


Fig. 5: Our idea for reducing the number of write ports of the *valid* memory and the *reqstatus* from two to one.

TABLE I: Parameters of the target NoCs

Router architecture	Input-queued VC router
Router pipeline	5-stage
Routing algorithm	XY, odd-even
# of VCs per port	2, 4
VC size	4-flit
Flit size	30-bit, 31-bit, 32-bit
VC/Switch allocator	iSLIP [19]
Arbiter type	Round-robin
Flow control	Wormhole & credit-based

arrive. The read port is for determining whether a read request should be issued to the DRAM controller.

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

We propose a method to make it possible to implement the *valid* memory and the *reqstatus* memory efficiently. Our method is based on the following observations. In the *valid* memory, it is not necessary to immediately process invalidate requests from the physical node. An invalidate request can be stalled for $2N_{log}$ FPGA cycles, which is the time needed to emulate one cycle of the network. When we stall an invalidate request and all subsequent ones (if there are any), there are at most $N_{log}-1$ update requests from the DRAM controller side. This is because a read request for $data[i]$ is not issued until $valid[i]$ is invalidated (reset to 0). Therefore, we can reduce the number of write ports of the *valid* memory from two to one using the idea illustrated in Fig. 5. We store invalidate requests from the physical node side in an N_{log}-entry FIFO. Update requests from the DRAM controller side are always serviced immediately. When there is no update request, an invalidate request is popped from the FIFO and serviced. Similarly, we can also reduce the number of write ports of the *reqstatus* memory to one. Since the *valid* memory is only 1-bit-wide, we simply duplicate it to reduce the number of read ports from two to one. Consequently, all memories in the proposed architecture have only one write port and one read port, and therefore, they can be implemented efficiently on FPGAs.

As shown in Fig. 4, the read requests of the trace loaders are arbitrated by an N_{phy}:1 arbiter where N_{phy} is the number of physical nodes of the physical cluster used to emulate the network. Packet descriptors provided by the DRAM controller are sent to the appropriate trace loaders based on the source addresses encoded inside them.

C. Emulation Methodology

We separate the time of the network from the times of the traffic generators. Each traffic generator, as well as the network, has its own time counter.

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

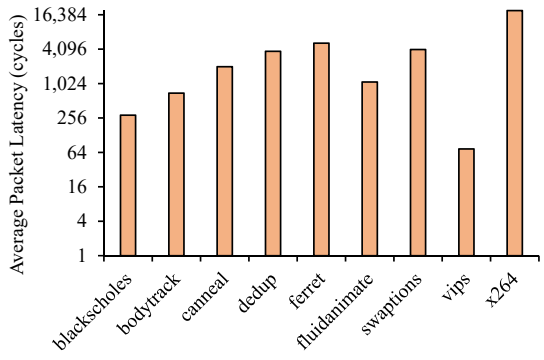
With the above approach, we allow the traffic generators to run ahead of the network. This helps to minimize the impact of stalling the network on the emulation performance (that is, hiding the DRAM access latency). However, the source queues may contain packet descriptors with packet generation timestamps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

One problem that might arise when the above emulation method is used is that the emulation might not be finished successfully if the largest packet generation timestamp in a node is smaller than the time to which we want to emulate. For example, assume that we want to emulate a 10,000-cycle trace and the largest packet generation timestamp in node 0 is 7,000. At emulation cycle t ($t > 7,000$) when the source queue in node 0 becomes empty, according to the emulation method explained above, the network is stalled since the time counter of the traffic generator in node 0 is 7,000 and smaller than t. If we do not provide any valid packet descriptors with packet generation timestamp larger than t, the network will be stalled forever. To deal with this problem, we append some dummy packet descriptors with the largest possible packet generation timestamp to the end of the trace of each node. The number of appended dummy packet descriptors is set to $L+1$ where L is the source queue length to prevent the corresponding trace loader from issuing unnecessary read requests to the DRAM controller.

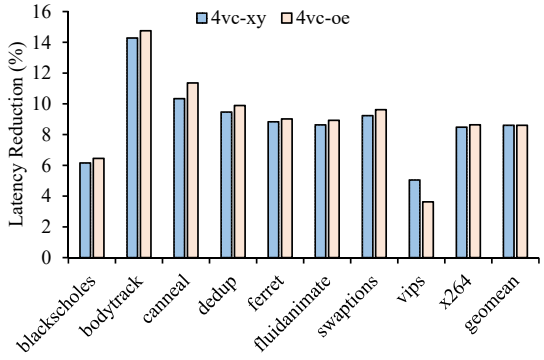
IV. Evaluation

Using the proposed architecture, we build a NoC emulator on a Xilinx VC707 board. Vivado 2017.4 is used for synthesizing, implementing, and generating FPGA bitstream files.

Table I shows the parameters of the NoCs evaluated in this paper. We implement two routing algorithms: the dimension-order routing algorithm (denoted by *XY*) and a minimal adaptive routing algorithm based on the odd-even turn model



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



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

Fig. 6: Results obtained when emulating an 8×8 NoC (the parameters is shown in Table I) with the PARSEC traces.

(denoted by *odd-even*) [20]. In the odd-even routing algorithm, when there are two available output ports, we select the port with more free VCs. The flit size depends on the routing algorithm used and the number of VCs per port. When the routing algorithm and the number of VCs per port are XY and two, respectively, the flit size is 30-bit. Increasing the number of VCs per port to four requires a flit size of 31-bit. Changing the routing algorithm from XY to odd-even adds one more bit to the flit structure.

We use the trace data of the PARSEC benchmark suite [17] collected by Hestness et al. [21] for evaluating the implemented 8×8 NoC. In these trace data, the packet length varies between 2-flit and 18-flit. In each trace, we focus on the region which represents the parallel portion of the application. To evaluate NoCs larger than 8×8, we create trace data of synthetic workloads.

A. Correctness

We verify the correctness of the implemented NoC emulator by running extensive emulations with the PARSEC traces and various traces created based on synthetic workloads. We confirmed that our implemented NoC emulator and BookSim reported exactly the same results in every case. This indicates that our NoC models are totally identical to those of BookSim.

Fig. 6(a) shows the average packet latencies obtained when emulating an 8×8 NoC (the parameters is shown in Table I; the routing algorithm, number of VCs per port, flit size are XY, 2, and 30-bit, respectively) with the PARSEC traces. For each trace, the latency data are collected over 20 million warm-up cycles and 20 million measurement cycles. We can see that applications like *x264, ferret*, and *dedup* exhibit high average packet latencies. This is because they have a high degree of data sharing and data exchange during the executions.

Fig. 6(b) shows how the average packet latency can be reduced when increasing the number of VCs per port from 2 to 4 (4vc-xy) and using the odd-even routing algorithm (4vc-oe). The figure shows that increasing the number of VCs helps to reduce an average of 8.61% packet latency. However, changing the routing algorithm from XY to odd-even has only a slight impact.

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

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

B. Resource Requirements and Scalability

Table II shows the FPGA resource requirements and operating frequencies when emulating different NoC sizes using a physical cluster of 16 nodes (4×4). The 8×8 NoC here is the one evaluated in Fig. 6(a). The other NoCs have the same parameters as the 8×8 NoC except for the network size.

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

The above result indicates that the scalability of the proposed architecture depends on only the total capacity of BRAMs on the FPGA used. Larger NoCs can be supported by using an FPGA with more BRAMs.

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

Next, we evaluate the impact of the method for efficiently implementing the *valid* and *reqstatus* memories described in Section III-B on the overall FPGA resource requirements and operating frequency. Fig. 7 shows that, when the emulated NoC is small, there is almost no difference between using and not using the proposed method. However, as the NoC size increases, the effectiveness of the proposed method is clearly established. When emulating the 128×64 NoC, the naïve approach of using the *valid* and *reqstatus* memories with multiple write ports and read ports requires 49.2% of FPGA

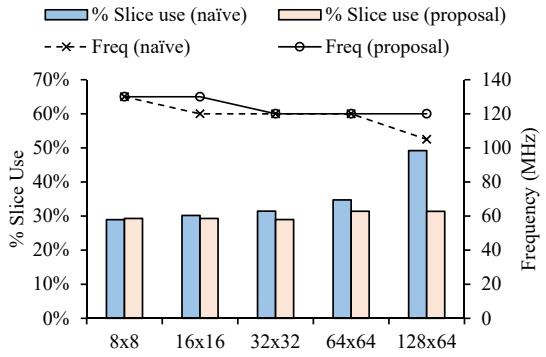


Fig. 7: Impact of the method for efficiently implementing the *valid* and *reqstatus* memories on the overall FPGA resource requirement and operating frequency.

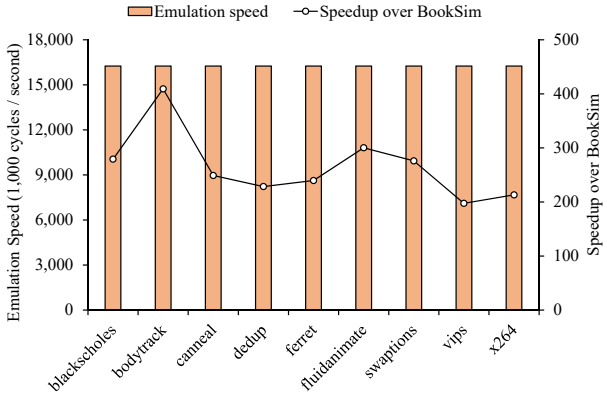


Fig. 8: Speed of our emulator when emulating the 8×8 NoC which is evaluated in Fig. 6(a) with the PARSEC traces. The physical cluster size used is 4×4.

slices. In contrast, the proposed method requires only 31.4% FPGA slices, which is almost the same as when emulating the 8×8 NoC. The proposed method also helps to improve the operating frequency.

C. Emulation Performance

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

$$S = \alpha \times \frac{F}{2 \times N_{log}} = \alpha \times \frac{F}{2 \times \frac{N_{node}}{N_{phy}}}, \qquad (1)$$

where F is the operating frequency (Hz) of the emulator.

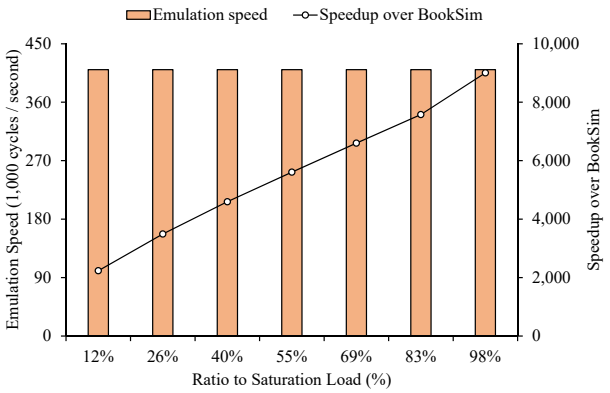


Fig. 9: Speed of our emulator when emulating the 64×64 NoC, which has the same parameters as the 8×8 NoC evaluated in Fig. 6(a), with traces created based on the uniform random traffic. The physical cluster size used is 8×4.

Fig. 8 shows the speed of our emulator when emulating the 8×8 NoC which is evaluated in Fig. 6(a) with the PARSEC traces. The physical cluster size used here is 4×4, and the emulator operates at a frequency of 130 MHz. We also make a comparison with BookSim which is run on a Core i7 4770 PC. We observe that the speed of our emulator is almost constant at around 16,250K emulation cycles per second in every case. In contrast, BookSim's speed varies with the applications. This is because each application features a different traffic load and it is widely known that BookSim's speed decreases with increasing traffic load [10], [13]. Consequently, the speedup of our emulator over BookSim varies with the applications. The average (geomean) speedup is 260×.

To the best of our knowledge, there does not exist any traces of realistic applications for emulation of large-scale NoCs with thousands of nodes. Since creating such traces is not trivial and extremely time-consuming, we adopt the approach of using traces created based on synthetic workloads to estimate the speed of our emulator when emulating large-scale NoCs.

Fig. 9 shows the speed of our emulator when emulating the 64×64 NoC, which has the same parameters as the 8×8 NoC evaluated in Fig. 6(a), with traces created based on the uniform random traffic under different loads. The physical cluster size used here is 8×4, and the emulator operates at a frequency of 105 MHz. Similar to the evaluation result of the 8×8 NoC above, the speed of our emulator is almost constant at around 410K emulation cycles per second in every case. The speedup over BookSim ranges from 2,235× to 9,005× with the average (geomean) of 5,106×.

Finally, we discuss the impact of DRAM on the emulation speed, that is, the coefficient α in formula (1). For a given trace, α depends on the length of the source queues in the traffic generators, the operating frequency of the emulator, and the emulated NoC. In all evaluations in this paper, the source queue length is set to 2-entry. A higher operating frequency will make α decrease because the pressure on the DRAM is increased.

Our evaluation results show that, when emulating the 8×8 NoC, we always have $\alpha > 0.9999$. Similarly, when emulating the 64×64 NoC, α is always greater than 0.9997. Thanks to the effective emulation architecture and methods proposed in Section III, the negative impact of the DRAM access latency is virtually eliminated.

D. Comparison with other FPGA-Based NoC Emulators

Among all emulators mentioned in Section I that support trace-driven emulation, only DuCNoC [14] reports results obtained when emulating a NoC with traces of realistic applications and provides a comparison with a well-known simulator which is also BookSim. The comparison results show that DuCNoC tracks BookSim closely. However, the differences are not zero. In contrast, the results reported by our emulator are totally identical to those reported by BookSim.

Next, we compare the speed of our emulator with those of DART [10], AdapNoC [12], FNoC [13], and DuCNoC [14], the most recently proposed FPGA-based NoC emulators. DART, AdapNoC, and DuCNoC support both emulation with synthetic workloads and trace-driven emulation while FNoC supports only emulation with synthetic workloads. The comparison here is not strictly quantitative but qualitative because of the following reasons. First, the router architectures modeled by the emulators are not the same. Second, the emulators are implemented on different FPGAs. Third, although DART, AdapNoC, and DuCNoC support trace-driven emulation, they only report the speeds of synthetic workload emulations.

When the emulated NoC size is 8×8, the emulation speed ranges of DART, AdapNoC, and FNoC are around 5,500K–16,000K, 30K–200K, and 11,580K–15,000K emulation cycles per second, respectively. Although the largest NoC sizes that can be emulated by DART and AdapNoC are 9×9 and 32×32, respectively, the authors of these emulators do not provide any results for NoCs larger than 8×8. On the contrary, the authors of FNoC also report the results of large-scale NoCs. FNoC's speed varies from around 319K to around 391K emulation cycles per second for a 64×64 NoC. DuCNoC's speed for a 5×5 NoC varies from around 200K to around 375K emulation cycles per second. For larger NoCs, the authors of DuCNoC do not report the absolute emulation speeds but instead the speedups compared to BookSim. As presented in Section IV-C, the speed of our emulator when emulating an 8×8 NoC with the PARSEC traces is 16,250K emulation cycles per second. When the NoC size is increased to 64×64, the speed is reduced to 410K emulation cycles per second.

V. Conclusion

Trace-driven simulation is an effective approach for studying NoCs but may require an excessive amount of time, especially when the target NoC size is large. We propose an effective architecture for speeding up trace-driven emulation of NoCs with up to thousands of nodes on FPGAs. We introduce some methods to effectively hide the off-chip memory access time and improve the scalability of the emulation architecture in terms of operating frequency and FPGA resource requirements. We achieve a speedup of 260× compared to BookSim, a widely used NoC simulator, when emulating an 8×8 NoC with the PARSEC traces. The speedup is increased to 5,106× when emulating a 64×64 NoC with trace data created based on a synthetic workload.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Numbers JP16H02794, JP17J09956.

References

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

[2] N. Jiang, D. U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D. E. Shaw, J. Kim, and W. J. Dally, "A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator," in *ISPASS*, 2013, pp. 86–96.

[3] N. Agarwal, T. Krishna, L. S. Peh, and N. K. Jha, "GARNET: A Detailed On-Chip Network Model inside a Full-System Simulator," in *ISPASS*, 2009, pp. 33–42.

[4] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti, "Cycle-Accurate Network on Chip Simulation with Noxim," *ACM Trans. on Modeling and Computer Simulation*, vol. 27, no. 1, pp. 4:1–4:25, 2016.

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

[6] P. Ren, M. Lis, M. H. Cho, K. S. Shim, C. W. Fletcher, O. Khan, N. Zheng, and S. Devadas, "HORNET: A Cycle-Level Multicore Simulator," *IEEE TCAD*, vol. 31, no. 6, pp. 890–903, 2012.

[7] P. T. Wolkotte, P. K. F. Holzenspies, and G. J. M. Smit, "Fast, Accurate and Detailed NoC Simulations," in *NOCS*, 2007, pp. 323–332.

[8] S. Lotlikar, V. Pai, and P. V. Gratz, "AcENoCs: A Configurable HW/SW Platform for FPGA Accelerated NoC Emulation," in *VLSID*, 2011, pp. 147–152.

[9] M. K. Papamichael, "Fast Scalable FPGA-Based Network-on-Chip Simulation Models," in *MEMOCODE*, 2011, pp. 77–82.

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

[11] T. V. Chu, S. Sato, and K. Kise, "Ultra-Fast NoC Emulation on a Single FPGA," in *FPL*, 2015, pp. 1–8.

[12] H. M. Kamali and S. Hessabi, "AdapNoC: A Fast and Flexible FPGA-based NoC Simulator," in *FPL*, 2016, pp. 1–8.

[13] T. V. Chu, S. Sato, and K. Kise, "Fast and Cycle-Accurate Emulation of Large-Scale Networks-on-Chip Using a Single FPGA," *ACM TRETS*, vol. 10, no. 4, pp. 27:1–27:27, 2017.

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

[15] Y. E. Krasteva, F. Criado, E. de la Torre, and T. Riesgo, "A Fast Emulation-Based NoC Prototyping Framework," in *ReConFig*, 2008, pp. 211–216.

[16] M. Badr and N. E. Jerger, "SynFull: Synthetic Traffic Models Capturing Cache Coherent Behaviour," in *ISCA*, 2014, pp. 109–120.

[17] C. Bienia, "Benchmarking Modern Multiprocessors," Ph.D. dissertation, Princeton University, 2011.

[18] N. Kapre and J. Gray, "Hoplite: Building Austere Overlay NoCs for FPGAs," in *FPL*, 2015, pp. 1–8.

[19] N. McKeown, "The iSLIP Scheduling Algorithm for Input-Queued Switches," *IEEE/ACM Trans. on Networking*, vol. 7, no. 2, pp. 188–201, 1999.

[20] G. M. Chiu, "The Odd-Even Turn Model for Adaptive Routing," *IEEE TPDS*, vol. 11, no. 7, pp. 729–738, 2000.

[21] J. Hestness, B. Grot, and S. W. Keckler, "Netrace: Dependency-driven Trace-based Network-on-chip Simulation," in *NoCArc*, 2010, pp. 31–36.