1.5em

AN ABSTRACT OF THE DISSERTATION OF

Fawaz Alazemi for the degree of Doctor of Philosophy in Computer Science presented on June 12, 2019.

Title: Routerless Network-on-chip

Abstract approved: _____

Bella Bose Lizhong Chen

Traditional bus-based interconnects are simple and easy to implement, but the scalability is greatly limited. While router-based networks-on-chip (NoCs) offer superior scalability, they also incur significant power and area overhead due to complex router structures. In this thesis, a new class of on-chip networks, referred to as *Routerless (RL) NoCs*, where routers are completely eliminated is explored. A novel design that utilizes on-chip wiring resources smartly to achieve comparable hop count and scalability to that of the router-based NoCs is proposed. Several effective techniques that significantly reduce the resource requirement to avoid new network abnormalities in routerless NoC designs are presented. Evaluation results show that, compared with a conventional mesh, the proposed routerless NoC achieves 9.5X reduction in power, 7.2X reduction in area, 2.5X reduction in zero-load packet latency, and 1.7X increase in throughput. In addition, compared with the state-of-the-art low-cost NoC design, the proposed approach achieves 7.7X reduction in power, 3.3X reduction in area, 1.3X reduction in zero-load packet latency, and 1.6X increase in throughput. Moreover, the shrinking features sizes due to technology innovation results in increasing link failure rates. For RL NoC this is a major concern due to a large number of wires that RL NoC employs. To address a solution to this problem, a fault tolerance technique for permanent link failures without using

redundant wires is introduced and this technique requires a minimal overhead to the existing RL NoC. The performance of the technique after a fault recovery is extensively assessed using synthetic traffic patterns and PARSEC workloads. On average, latency is increased by 5.2% and 10.03% for synthetic patterns and PARSEC workloads, respectively. Moreover, this technique requires only an additional 4% area and 20.3% of total power.

Routerless Network-on-chip

by

Fawaz Alazemi

A DISSERTATION

submitted to

Oregon State University

in partial fulfillment of
the requirements for the
degree of

Doctor of Philosophy

Presented June 12, 2019
Commencement June 2019

Doctor of Philosophy dissertation of Fawaz Alazemi presented on June 12, 2019.

APPROVED:

Major Professor, representing Computer Science

Head of the School of Electrical Engineering and Computer Science

---

Dean of the Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Fawaz Alazemi, Author

ACKNOWLEDGEMENTS

I am extremely thankful to my advisors Prof. Bella Bose and Prof. Lizhong Chen for their endless support, excellence mentorship, and inspiration through my doctoral degree voyage. It was my honor to have them both as my advisors. I also would like to thank:

- my wife, Arwa, for her care, support, sacrifice through all the ups and downs in this journey. Through the journey we were bless by two beautiful girls, Mariam and Noura, which brought to us a lot of joy and funny moments. Without their understanding and encouragement this journey would not finish.
- all my colleges in the STAR lab Ryan Gambord, Yongbin Gu, Yunfan Li, Arash Azizi Mazreah, Drew Penney, and Aashish Adhikari.
- my parents, Mohammed and Saud, and my brothers Hamid, Jaber, and Abdulrahman and my sisters Afrah, Abrar, and Rawan.
- my Ph.d committee Prof. Ben Lee, Prof. Rakesh Bobba, and William H. Warnes.
- Prof. Ziad Najem for being a role model.
- Prof. Mehmet Karaata for all the pleasent night we spent in Shaiwkh campus.
- Kuwait University for the generaous scholarship.
- Oregon State University for offering an excellent academic environment.
- all my friends in Kelly and Corvallis.

From the bottom of my heart, Thank you.

TABLE OF CONTENTS

Page

1 Introduction 1

1.1 Background and Motivation . . . 3
1.1.1 Related Work . . . 3
1.1.2 Need for New Routerless NoC Designs . . . 9

1.2 Analysis on Wiring Resources . . . 12
1.2.1 Metal Layers . . . 12
1.2.2 Wiring in NoC . . . 13
1.2.3 Usable Wires for NoCs . . . 15

1.3 NoC Simulators . . . 17

2 Routerless network-on-chip design 22

2.1 Designing Routerless NoCs . . . 22
2.1.1 Basic Idea . . . 22
2.1.2 Examples . . . 29
2.1.3 Formal Procedure . . . 31

2.2 Implementation Details . . . 35
2.2.1 Injection Process . . . 35
2.2.2 Ejection Process . . . 38
2.2.3 Avoiding Network Abnormalities . . . 39

2.3 Deadlock Avoidance . . . 40
2.3.1 Interface Hardware Implementation . . . 42

2.4 Evaluation methodology . . . 44

2.5 Results and Analysis . . . 48
2.5.1 Ejection Links and Extension Buffers . . . 48
2.5.2 Synthetic Workloads . . . 51
2.5.3 PARSEC and SPLASH-2 Workloads . . . 51
2.5.4 Power . . . 53
2.5.5 Area . . . 54

2.6 Discussion . . . 58
2.6.1 Scalability and Regularity . . . 58
2.6.2 Average Overlapping . . . 58
2.6.3 Impact on Latency distribution . . . 59
2.6.4 RL for $n \times m$ Chip . . . 60

2.7 Highlights on implementing Routerless in Booksim . . . 61
2.7.1 How Booksim works . . . 61

TABLE OF CONTENTS (Continued)

Page

2.7.2 Routerless implementation 65
2.8 Conclusion 73

3 Reliability of Routerless NoC 74
3.1 Faults in Networks-on-Chip 74
3.1.1 Motivation 81
3.2 Addressing reliability in RL NoC 82
3.2.1 A fault example in RL 85
3.2.2 Fault detection and switches 90
3.3 Evaluation and results 91
3.3.1 Evaluation methodology 91
3.4 Results and Analysis 92
3.4.1 Synthetic workloads 92
3.4.2 PARSEC Workloads 94
3.4.3 PARSEC traces 96
3.4.4 Power and area analysis 96
3.5 Discussion 98
3.5.1 Multiple faults in NoCs 98
3.6 Highlight on implementing RL in Gem5 101
3.7 Conclusion 101

4 Conclusion and future work 103
4.1 Future work 104
4.1.1 Routerless tiles 104
4.1.2 Optimal width of a link 104
4.1.3 Use AI to find better set of loops 105
4.1.4 Application mapping for RL networks 105
4.1.5 Multicast and broadcast 105

Bibliography 105

Appendices 110
Patent 111
Routerless Network-on-chip HPCA 2018 146

TABLE OF CONTENTS (Continued)

Page

Routerless loops for 4× 4 158
Routerless loops for 8× 8 159
Routerless loops for 16× 16 161
Topology script for Routerless in Gem5 170

LIST OF FIGURES

Figure		Page
1.1	An example of loops in a 4×4 grid.	6
1.2	A long wire in NoCs with repeaters.	7
2.1	Layers of an 8×8 grid.	23
2.2	Loops in L_1, and $M_2 = L_1$.	24
2.3	Loops in Layer 2 (L_2). $M_4 = L_2 \cup L_1$.	24
2.4	Loops in L_3. $M_6 = L_3 \cup L_2 \cup L_1$.	26
2.5	Loops in L_4. $M_8 = L_4 \cup L_3 \cup L_2 \cup L_1$.	28
2.6	Routerless interface components.	33
2.7	Injecting a long packet requires a packet-sized buffer per loop at each hop in prior implementation (X, Y and Z are interfaces).	34
2.8	Throughput of routerless NoC under different number of ejection links and extension buffers (EXBs).	46
2.9	Performance comparison for synthetic traffic patterns.	47
2.10	PARSEC and SPLASH-2 benchmark performance results (y-axis represents average pack latency in cycles.) RL is compared with different Mesh configurations, EVC, and IMR in (a), (b) and (c). In (d), RL is also compared with a 3D Cube.	50
2.11	Breakdown of power consumption for different PARSEC and SPLASH-2 workloads (normalized to Mesh).	52
2.12	Area comparison under 15nm technology.	56
2.13	Average hop count for synthetic workloads.	57
2.14	Latency distribution of benchmarks for RL 8×8 NoC.	59
2.15	Call hierarchy for `TimedModule` objects in every clock cycle	63
2.16	Router's pipline implemented by `Evaluate()` in Booksim	64

LIST OF FIGURES (Continued)

Figure		Page
3.1	RL Algorithm grows the network in concentric layers	76
3.2	Layer 1 (2x2) contains two counter-rotating loops	77
3.3	Layer 2 (4x4) loops provide connectivity to new outer nodes	77
3.4	Layer 3 (6x6) loops showing alternating direction of column loops	78
3.5	Percentage of node pairs that share a single loop	79
3.6	Only one loop shared between gray nodes and the black node	80
3.7	Passive links offer redundancy in the event of link failure	83
3.8	Supplemental short loops deactivate and provide donor links to bypass failed links in RL loops	83
3.9	Proposed auxiliary's Hamiltonian loop to provide redundancy in the event of loop failure	84
3.10	Only one loop shared between gray nodes and the black node	87
3.11	Two loops are joined into one after detecting a faulty link	88
3.12	High level BIST design to detect faults in links.	90
3.13	Average latency of traffic patterns at injection rate 0.005.	93
3.14	Average hop count prior to fault vs. after a fault. Average hop count depends on fault location expected and worst case average hop count are shown.	94
3.15	Percentage of execution time change with and without a fault	94
3.16	Expected average hop count for 8×8 RL	95
3.17	Average latency of PARSEC workloads with and without a fault	95
3.18	Breakdown of power consumption for PARSEC workloads	98
3.19	Fusing two loops more than once will result into disconnected loops.	99
3.20	Example of ten faults in 4x4 NoC	100

LIST OF TABLES

Table		Page
1.1	Number of unidirectional loops in $n \times n$ grid [3].	5
1.2	Wiring resources in a many-core processor chip.	8
1.3	Major features of Gem5 simulator [2]	18
2.1	Average overlapping and loops/rings in RL/IMR	59
2.2	Loops for 4×4 NoC	72
3.1	Key configuration parameters of simulation	91

List of Algorithms

1 RLrec 32
2 Evaluation 66
3 Input Queuing 66
4 Ejection Module 67
5 Input Module 68
6 Injection Module 69
7 Route Traffic Module 70
8 Reset Flags Module 71

Chapter 1: Introduction

As technologies continue to advance, tens of processing cores on a single chip-multiprocessor (CMP) has already been commercially offered. Intel Xeon Phi Knight Landing [16] is an example of a single CMP that has 72 cores. With hundreds of cores in a CMP around the corner, there is a pressing need to provide efficient networks-on-chip (NoCs) to connect the cores. In particular, recent chips have exhibited the trend to use many but simple cores (especially for special-purpose many-core accelerators), as opposed to a few but large cores, for better power efficiency. Thus, it is imperative to design highly scalable and ultra-low cost NoCs that can match with many simple cores.

Prior to NoCs, buses have been used to provide on-chip interconnects for multi-core chips [10, 11, 18, 22, 43, 44]. While many techniques have been proposed to improve traditional buses, it is hard for their scalability to keep up with modern many-core processors. In contrast, NoCs offer a decentralized solution by the use of routers and links. Thanks to the switching capability of routers to provide multiple paths and parallel communications, the throughput of NoCs is significantly higher than that of buses. Unfortunately, routers have been notorious for consuming a substantial percentage of chip's power and area [25, 26]. Moreover, the cost of routers increases rapidly as link width increases. Thus, except for a few ad hoc designs, most on-chip networks do not employ link width higher than 256-bit or 512-bit, even though additional wiring resources may be available. In fact, our study shows that, a 6x6 256-bit Mesh only uses 3% of the

total available wiring resources (more details in Section 1.2).

The high overhead of routers motivates researchers to develop *routerless NoCs* that eliminate the costly routers but use wires more efficiently to achieve scalable performance. While the notion of routerless NoC has not been formally mentioned before, prior research has tried to remove routers with sophisticated use of buses and switches, although with varying success. The goal of routerless NoCs is to select a set of smartly placed loops (composed of wires) to connect cores such that the average hop count is comparable to that of conventional router-based NoCs. However, the main roadblocks are the enormous design space of loop selection and the difficulty in avoiding deadlock with little or no use of buffer resources (otherwise, large buffers would defeat the purpose of having routerless NoCs).

In this thesis, we explore efficient design and implementation to materialize the promising benefits of routerless NoCs. Specifically, we propose a layered progressive method that is able to find a set of loops meeting the requirement of connectivity and the limitation of wiring resources. The method progressively constructs the design of a large routerless network from good designs of smaller networks, and is applicable to any $n \times m$ many-core chips with superior scalability. Moreover, we propose several novel techniques to address the challenges in designing routerless interface to avoid network abnormalities such as deadlock, livelock and starvation. These techniques result in markedly reduced buffer requirement and injection/ejection hardware overhead. In addition, compared with a conventional router-based Mesh, the proposed routerless design achieves 9.48X reduction in power, 7.2X reduction in area, 2.5X reduction in zero-load packet latency, and 1.73X increase in throughput. In addition, compared with the current state-of-the-art

scheme that tries to replace routers with less costly structures (IMR [33]), the proposed scheme achieves 7.75X reduction in power, 3.32X reduction in area,1.26X reduction in zero-load packet latency, and 1.6X increase in throughput.

The rest of the paper is organized as follows. In Sections 2, we provide background on wiring resource and discuss the related work. Section 3 illustrates the concept of routerless NoCs and the associated design challenges. In Section 4, we explain the details of a routerless NoC design followed by implementation details in Section 5. Section 6 describes the evaluation methodology and Section 7 presents the evaluation results. Finally, some further discussion is provided in Section 8, and Sections 9 concludes the paper.

1.1 Background and Motivation

1.1.1 Related Work

Prior work on on-chip interconnects can be classified into *bus-based* and *network-based*. The latter can be further categorized as *router-based NoCs* and *routerless NoCs*. The main difference between bus-based interconnects and routerless NoCs is that bus-based interconnects use buses in a direct, simple and primitive way, whereas routerless NoCs use a network of buses in a sophisticated way and typically need some sort of switching that earlier bus systems do not need. Each of the three categories is discussed in more detail below.

Bus-based Interconnects are centralized communication systems that are straight-

forward and cheap to implement. While buses work very well for a few cores, the overall performance degrades significantly as more cores are connected to the bus [22, 43]. The two main reasons for such degradation are the length of the bus and its capacitive load. Rings [10, 11, 18] can also be considered as variants of bus-based systems where all the cores are attached to a single bus/ring. IBM Cell processor [44] is an improved bus-based system which incorporates a number of bus optimization techniques in a single chip. Despite having a better performance over conventional bus/ring implementations, IBM Cell process still suffers from serious scalability issues [6].

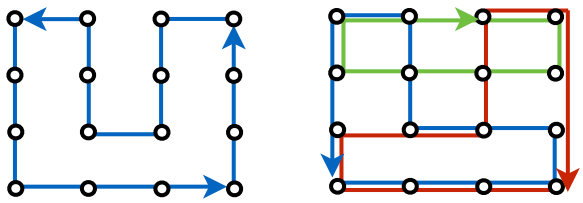
Router-based NoCs are decentralized communication systems. A great deal of research has gone into this (e.g., [14, 17, 23, 28, 30, 31, 36, 38], too many to cite all here). The switching capability of routers provides multiple paths and parallel communications to improve throughput, but the overhead of routers is also quite substantial. Bufferless NoC (e.g., [20]) is a recent interesting line of work. In this approach, buffer resources in a router are reduced to the minimal possible size (i.e. one flit buffer per input port). Although bufferless NoC is a clever approach to reduce area and power overhead, the router still has other expensive components that are eliminated in the routerless approach (Section 2.5.5 compares the hardware cost).

Routerless NoCs aim to eliminate the costly routers while having scalable performance. While the notion of routerless NoC has not been formally mentioned before, there are several works that try to remove routers with sophisticated use of buses and switches. However, as discussed below, the hardware overhead in these works is quite high, some requiring comparable buffer resources as that of conventional routers, thus not truly materializing the benefits of routerless NoCs. One approach is presented in [40],

where the NoC is divided into segments. Each segment is a bus, and all the segments are connected by a central bus. Segments and central bus are linked by a switching element. In large NoCs, either the segments or the central bus may suffer from scalability issues due to their bus-based nature. A potential solution is to increase the number of wires in the central bus and the number of cores in a segment. However, for NoCs larger than 8×8, it would be challenging to find the best size for the segments and central bus without affecting scalability. Hierarchical rings (HR) [21] has a similar design approach as that of [40]. The NoC is divided into disjoint sets of cores, and each set is connected by a ring. Such rings are called local rings. Additionally, a set of global rings bring together the local rings. Packets switch between local and global rings through a low-cost switching element. Although the design has many nice features, the number of switching element is still not small. For example, for an 8×8 NoC, there are 40 switching elements, and this number is close to the number of routers in the 8×8 network. Recently, a multi-ring-based NoC called isolated multiple rings (IMR) is proposed in [33] and has been shown to be superior than the above Hierarchical rings. To our knowledge, this is the latest and best scheme so far along the line of work on removing routers. While the proposed concept is promising, the specific IMR design has several major issues and the results are far from optimal, as discussed in the next subsection.

Table 1.1: Number of unidirectional loops in $n \times n$ grid [3].

n	# of loops	n	# of loops
1	0	2	2
3	26	4	426
5	18,698	6	2,444,726
7	974,300,742	8	1,207,683,297,862



(a) (b)

Figure 1.1: An example of loops in a 4×4 grid.

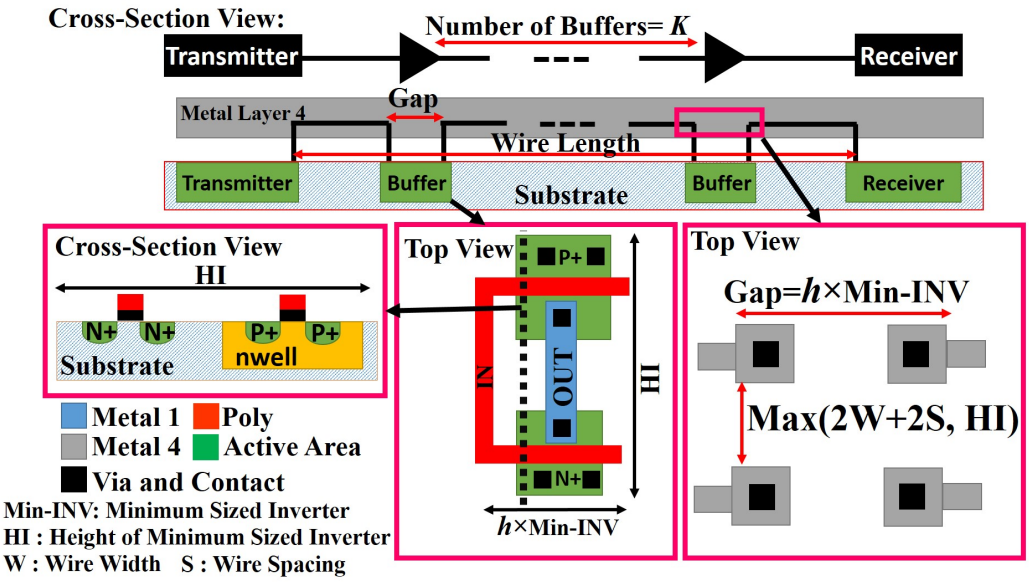


Figure 1.2: A long wire in NoCs with repeaters.

Table 1.2: Wiring resources in a many-core processor chip.

Many Core Processor	**Xeon Phi, Knights Landing**	
Number of Cores	72	
NoC Size	6×6	
Die Area	(31.9mm x 21.4mm) 683 mm^2 [4]	
Technology	FinFET 14nm	
Interconnect	13 Metal Layers	
Inter-core Metal Layers	Metal Layer	Pitch [27] [35]
	M4	80nm
	M5	104nm

1.1.2 Need for New Routerless NoC Designs

1.1.2.1 Principles and Challenges

We use Figure 1.1 to explain the basic principles of routerless NoCs. This figure depicts an example of a 16-core chip. The 4×4 layout specifies only the positions of the cores, not any topology. A straightforward but naive way to achieve routerless NoC is to use a long loop (e.g., a Hamiltonian cycle) that connects every node on the chip as shown in Figure 1.1(a). Each node injects packets to the loop and receives packets from the loop through a simple interface (referred to as RL interface hereafter). Apparently, even if a flit on the loop can be forwarded to the next node at the speed of one hop per cycle, this design would still be very slow because of the average $O(n^2)$ hop count, assuming an $n \times n$ many-core chip. Scalability is poor in this case, as the conventional topology such as Mesh has an average hop count of $O(n)$.

To reduce the hop count, we need to select a better set of loops to connect the nodes, while guaranteeing that every pair of nodes is connected by at least one loop (so that a node can reach another node directly in one loop). Figure 1.1(b) shows an example with the use of three loops, and this technique satisfies the connectivity requirement and reduces the all-pair average hop count by 46% compared to the technique shown in Figure 1.1 (a). Note that, when injecting a packet, a source node chooses a loop that connects to the destination node. Once the packet is injected into a loop, it stays on this loop and travels at the speed of one hop per cycle all the way to the destination node. No changing loops is needed at RL interfaces, thus avoiding the complex switching hardware and per-hop contention that may occur in the conventional router-based on-chip

networks.

Several key questions can be asked immediately. Is the design in Figure 1.1(b) optimal? Is it possible to select loops that achieve comparable hop count as conventional NoCs such as Mesh? Is there a generalized method that we can use to find the loops for any $n \times n$ network? How can this be done without exceeding the available on-chip wiring resources? Unfortunately, answering these questions is extremely challenging due to the enormous design space. We calculated the number of possible loops for $n \times n$ chips based on the method used in [3], where a loop can be any unidirectional circular path with the length between 4 and n. Table 1.1 lists the results up to $n = 8$. As can be seen, the number of possible loops grows extremely rapidly. To overcome this problem, we had to choose a subset of loops such that there is at least one loop connecting any part of nodes.

Meanwhile, any selected final set of loops needs to comfortably fit in the available wiring resources on the chip. Specifically, when loops are superpositioned, the number of overlapped loops between any neighboring node pairs should not exceed a limit. In what follows, we use *overlapping* to refer to the number of overlapped loops between two neighboring nodes (e.g., in Figure 1.1(b) some neighboring nodes have two loops passing through them while others have only one loop passing), and use *overlapping cap* to refer to the limit of the overlapping. Note that the cap should be much lower than the theoretical wiring resources on chip due to various practical considerations (analyzed in Section 1.2). As an example, if the overlapping cap is 1, then Figure 1.1(a) has to be the final set. If the overlapping cap increases to 2, it provides more opportunity for improvement, e.g., the better solution in Figure 1.1(b). The overlapping cap is a hard limit

and should not be violated. However, as long as this cap is met, it is actually beneficial to approach this cap for as many neighboring node pairs as possible. Doing this indicates that more wires are being utilized to connect nodes and the hop count is reduced.

1.1.2.2 Major Issues in Current State-of-the-Art

There are several major issues that must be addressed in order to achieve effective routerless NoCs. We use IMR [33] as an example to highlight these issues. IMR is a state-of-the-art design that follows the above principle to deploy a set of rings such that each ring joins a subset of cores. While IMR has been shown to outperform other schemes with or without the use of routers, the fundamental issues in IMR prevent it from realizing the true potential of routerless NoCs. This calls for substantial research on this topic to develop more efficient routerless designs and implementations.

(1) *Large overlapping*. For example, IMR uses a large number of superpositioned rings (equivalent to the above-defined overlapping cap of 16) without analyzing the actual availability of wiring resources on-chip.

(2) *Extremely slow search*. A genetic algorithm is used in IMR to search the design space. This general-purpose search algorithm is very slow (taking several hours to generate results for 16×16, and is not able to produce good results in a reasonable time for larger networks). Moreover, the design generated by the algorithm is far from optimal with high hop counts, as evaluated in Section 2.4. Thus, efforts are much needed to utilize clever heuristics to speed up the process.

(3) *High buffer requirement*. Currently, the network interface of IMR needs one packet-

sized buffer per ring to avoid deadlock. Given that up to 16 rings can pass through an IMR interface, the total number of buffers at each interface is very close to a conventional router.

The above issues are addressed in the next three sections. Section 1.2 analyzes the main contributing factors that determine the wiring availability in practice, and estimates reasonable overlapping caps using a contemporary many-core processor. Section 2.1 proposes a layered progressive approach to select a set of loops, which is able to generate highly scalable routerless NoC designs in less than a second (up to 128×128). Section 2.2 presents our implementation of routerless interface. This includes a technique that requires only one flit-sized buffer per loop (as opposed to one packet-sized buffer per loop). This technique alone can save buffer area by multiple times.

1.2 Analysis on Wiring Resources

1.2.1 Metal Layers

As technology scales to smaller dimensions, it provides a higher level of integration. With this trend, each technology comes with an increasing number of routing metal layers to meet the growing demand for higher integration. For example, Intel Xeon Phi (Knights Landing) [1] and KiloCore [13] are fabricated in the process technology with 11 and 13 metal layers, respectively. Each metal layer has a pitch size which defines the minimum wire width and the space between two adjacent wires. The physical difference between metal layers results in various electrical characteristics. This allows designers

to meet their design constraints such as delay on the critical nets by switching between different layers. Typically, lower metal layers have narrower width and are used for local interconnects (e.g., within a circuit block); higher metal layers have wider width and are used for global interconnects (e.g., power supply, clock); middle metal layers are used for semi-global interconnects (e.g., connecting neighboring cores). Table 1.2 lists several key physical parameters of Xeon Phi including the middle layers that can be used for on-chip networks.

1.2.2 Wiring in NoC

To estimate the actual wiring resources that can used for routing, several important issues should be considered when placing wires on the metal layers.

Routing strategy: In general, two approaches can be considered for routing interconnects over cores in NoCs. In the first approach, dedicated routing channels are used to route wires in NoCs. This method of routing was widely used in earlier technology nodes where only three metal layers were typically provided [42], and it has around 20% area overhead. In the second approach, wires are routed over the cores at different metal layers [37]. In the modern technology nodes with six to thirteen metal layers, this approach of routing over logic becomes more common for higher integration. This can be done in two ways: 1) several metal layers are dedicated for routing wires, and 2) a fraction of each metal layer is used to route the wires. The first way is preferable given that many metal layers are available in advanced technology nodes [37, 42].

Repeater: Wires have parasitic resistance and capacitance which increase with the length of wires. To meet a specific target frequency, a long wire needs to be split into several segments, and repeaters (inverters) are inserted between the segments, as shown in Figure 1.2. The size of repeaters should be considered in estimating the available wiring resources. For a long wire in the NoC, the size of each repeater (h times of an inverter with minimum size) is usually not small, but the number of repeaters (k) needed is small [32]. In fact, it has been shown that increasing k has negligible improvement in reducing the delay [32]. For a 2GHz operating frequency, using only one repeater with the size of 40 times W/L of the minimum sized inverter can support a wire length of 2mm [37], which is longer than the typical distance between two cores in a many-core processor [41].

Coping with cross-talk: Cross-talk noises can occur either between the wires on the same metal layer or between the wires on different metal layers, both of which may affect the number of wires that can be placed. The impact of cross-talk noises on voltage can be calculated by Equation (1) as the voltage changes on a floated victim wire [24].

$$\Delta V_{victim} = \frac{C_{adj}}{C_{victim} + C_{adj}} \times \Delta V_{aggressor} \tag{1.1}$$

where ΔV_{victim} is the voltage variation on the victim wire, $\Delta V_{aggressor}$ is the voltage variation on the aggressor, C_{victim} is the total capacitance (including load capacitance) of the victim wire, and C_{adj} is the coupling capacitance between the aggressor and the victim. It can be observed from Equation (1) that the impact of cross-talk on the victim wire depends on the ratio of C_{adj} to C_{victim}. Hence, the cross-talk on the same layer

has much larger impact on the power, performance, and functionality of the NoC since the adjacent wires which run in parallel on the same metal layer has larger coupling capacitance (C_{adj}) [24]. There are two major techniques to mitigate cross-talk noises, shielding and spacing. In the shielding approach, crosstalk noises are largely avoided between two adjacent wires by inserting another wire (which is usually connected to the ground or supply voltage) between them. In the spacing approach, adjacent wires are separated by a certain distance that would keep the coupling noise below a level tolerable by the target process and application. Compared with spacing, shielding is much more effective as it can almost remove crosstalk noises [8]. However, shielding also incurs more area overhead as the distance used in the spacing approach is usually smaller than that of inserting a wire.

1.2.3 Usable Wires for NoCs

To gain more insight on how many wiring resources are usable for on-chip networks under current manufacturing technologies, we estimated the number of usable wires by taking into account the above factors. The estimation is based on using two metal layers to route wires over the cores. The area overhead of the repeater insertion including the via contacts and the area occupation of the repeaters are considered based on the layout design rules of each metal layer. We used the conservative way of shielding to reduce crosstalk noises (and the inserted wires are not counted towards usable wires), although spacing may likely offer more usable wires. In addition, in practice, 20% to 30% of each dedicated metal layer for routing wires over the cores is used for I/O

signals, power, and ground connections [37]. This overhead is also accounted for. The maximum values of h and k are used for worst-case estimation. As such, the above method gives a very conservative estimation of the usable wires. Assuming that there is a chip with similar physical configuration as Table 1.2, the two metal layers M4 and M5 under 14nm technology can provide 101,520 wires in the cross-section. This translates into 793 unidirectional links of 128-bit, or 396 unidirectional links of 256-bit, or 198 unidirectional links of 512-bit in the cross-section. In contrast, a 6×6 mesh only uses 12 unidirectional 256-bit links in the bisection, which is about 3% of the usable wires. It is important to note that the conventional router-based NoCs do not use very wide links for good reasons. For instance, router complexity (e.g., the number of crosspoints in switches, the size of buffers) increases rapidly as the link width increases. Also, although wider links provide higher throughput, it is difficult to capitalize on wider links for lower latency. The reduction in serialization latency by using wider links quickly becomes insignificant as link width approaches the packet size. This motivates the need for designing routerless NoCs where wiring resources can be used more efficiently.

The above estimation of the number of usable wires helps to decide the overlapping cap mentioned previously. To avoid taxing too much on the usable wiring resources and to have a scalable design, we propose to use an overlapping cap of n for $n \times n$ chips. In the above 6×6 case, this translates into 4.5% of the usable wires for 128-bit loop width, or 9.1% for 256-bit loop width. This parameterized overlapping cap helps to provide the number of loops that is proportional to chip size, so the quality of the routerless designs can be consistent for larger chips.

1.3 NoC Simulators

Computer architecture simulators are key players in the advancements of computer architecture research. Computer architects use simulators to construct prototypes of their ideas for testing and verifications purposes, to avoid huge costs associated with building real systems, and as a performance evaluation methodology. Gem5 [12] is well-known open-source simulator among computer architects which is the result of merging M5 and GEMS simulators. The project of Gem5 is emanated from efforts generated by various major companies such as HP, MIPS, ARM and top academic institutions such as Princeton, MIT, Wisconsin, Michigan Ann Arbor, and Texas Austin. Gem5 is still backed up by major companies (ARM, Hewlett-Packard, Intel, AMD, and Microsoft.)

Gem5 has a modular platform and includes system-level architecture as well as processor microarchitecture. In addition, it is flexible and capable of evaluating a wide variety of components. For example:

1. Gem5 supports four different CPU models (single CPI in-order, out-of-order, KVM) where each exists in a different point of speed-vs-accuracy spectrum
2. support multiple ISA model (Alpha, ARM, SPARC, MIPS, POWER, RISC-V and x86)
3. System-call Emulation mode and full-system mode capabilities
4. GPU integrations
5. Power and energy models

Table 1.3: Major features of Gem5 simulator [2]

1	Feature	Description
2	Multiple interchangeable CPU models.	Simple one-CPI, in-order, out-of-order (O3), and KVM-based CPU.
3	GPU integration model	Run actual ISA and share virtual memory with the host CPU.
4	NoMali GPU model	NoMali GPU model is integrated in Gem5 which is compatible with Linux and Android GPU driver stack.
5	Event-driven memory system	Support of caches, crossbars, snoop filters, and a fast and accurate DRAM controller model.
6	A trace-based CPU model	Timing annotated traces for memory-system throughput analysis.
7	Homogeneous and heterogeneous multi-core	Arbitrary topologies for CPU models and caches coherence protocols.
8	Multiple ISA support	Supports for Alpha, ARM, SPARC, MIPS, POWER, RISC-V and x86 ISAs.
9	Full-system capability	Support for full system simulation for major CPUs including ALPHA, ARM, SPARC, and x86.
10	Multi-system capability	Gem5 can simulate multple systems in a single simulation process.
11	Power and energy modeling	Support for OS-controller Dynamic Voltage and Frequency (DVFS) scaling.
12	Co-simulation with SystemC	with SystemC support, Gem5 can run as single thread in SystemC simulation to interoperate SoC component models, such as interconnects, devices, and accelerators.

Table 1.3 list all main features provided by Gem5 according to [2].

Gem5 has several design features and behaviors. The features include python integration, utilization of standard interface, pervasive object orientation design, and use of domain-specific languages whereas the behaviors include check-pointing/serialization, initialization, configuration, and statistics-based behaviors for each SimObject in Gem5. Despite a majority of gem5 being written in C++, the role of Python in the initialization, configuration, and simulation control. The pervasive object-oriented design focusses on the aspect of flexibility that is established through creating independent objects in which their integration lead to multi-system and multi-core modeling. DSLs (Domain Specific Languages) offers a powerful way of expressing several solutions. This design feature also offers idioms and knowledge that are relevant to the concerned problem space. Two types of DSLs are used in this case (Cache Coherence DSL and the ISA DSL.) Another important design feature is the Standard Interfaces which are fundamentals to the object-oriented design. These features closely work together with the Domain Specific Languages. Two central interfaces are referred to in this case, that is the port interface and the message buffer interface. One of port interface uses is to connect the memory objects in Gem5. That is, connecting CPUs and caches, Caches to busses, and busses to memories and devices as well. Another usage of the port interfaces is to access data (timing, atomic, and functional), feed Gem5 with interconnection topology, and debugging.

As network-on-chip (NoC) architectures become important in microprocessor development and the number of cores and modules on a single chip multiply, the support of a NoC module become vital for any simulator. GARNET [5] is a detailed cycle-accurate

network simulator that was incorporated into the Gem5 simulator and is also available as a standalone network simulator. GARNET is implemented as an event-driven simulator that can model NoCs in detail up to the microarchitecture level. It also offers a five-stage classical pipelined router by default under the virtual channel flow control mechanism. Booksim [29] is another stand-alone NoC simulator which is cycle-accurate and modular, and also offers a large set of configuration parameters. Booksim better match RTL model of a canonical NoC router than GARNET. In general, the accuracy and speed of execution are two conflicting parameters and this is true for Booksim.

Any NoC simulator must support at least router pipeline, routing implementation, the flow control, saturation throughput, and custom topology. Router pipeline along with routing algorithm and flow control defines the router microarchitecture. Saturation throughput is used to measure the performance of the network at high level injection rates. Another metrics commonly used in measuring network performance in zero-load latency which is used as a lower bound when it comes to network latency. The two metrics are either obtained from drawn latency versus throughput curves or analytically estimated. Topology is another key element in network simulators. It imposes bisection bandwidth, latency bound, and initial throughput, which is governed by the bandwidth as measured from the network diameter. The routing algorithm is also critical in this case as it defines the communication within the network topology. It also offers various trade-offs on achieving recommendable performance and other limitations as well. Routing algorithms must be free of any network abnormalities such as deadlocks, starvation, and livelock.

In this thesis, we used Booksim and Gem5 to study RL design performance. Due to the major differences between RL design and Router-based designs, we had to replace

router microarchtecture components, for Booksim and GARNET, with an implementation that matches the interface of RL design

Chapter 2: Routerless network-on-chip design

2.1 Designing Routerless NoCs

2.1.1 Basic Idea

Our proposed routerless NoC design is based on what we call *layered progressive* approach. The basic idea is to select the loop set in a progressive way where the design of a large routerless network is built on top of the design of smaller networks. Each time the network size increments, the newly selected loops are conceptually bundled as a layer that is reused in the next network size.

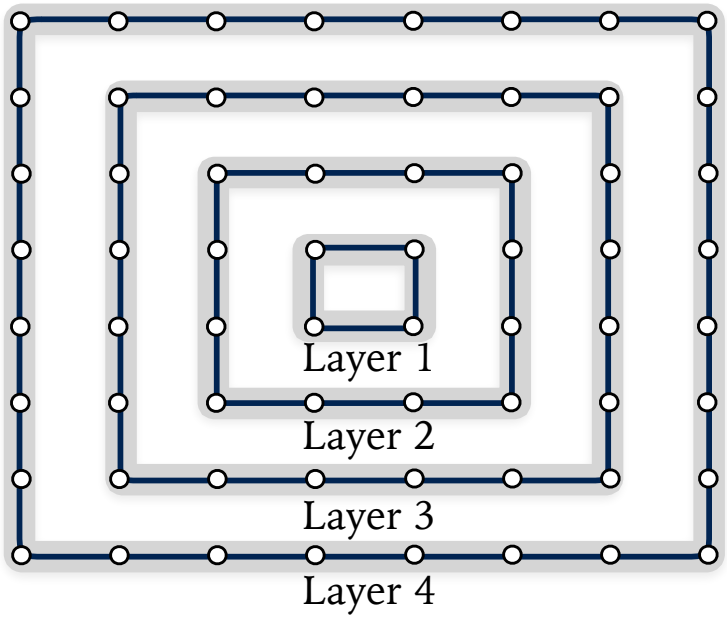


Figure 2.1: Layers of an 8×8 grid.

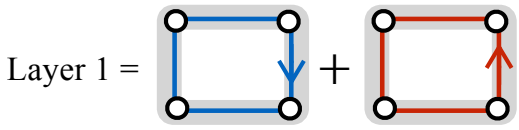


Figure 2.2: Loops in L_1, and $M_2 = L_1$.

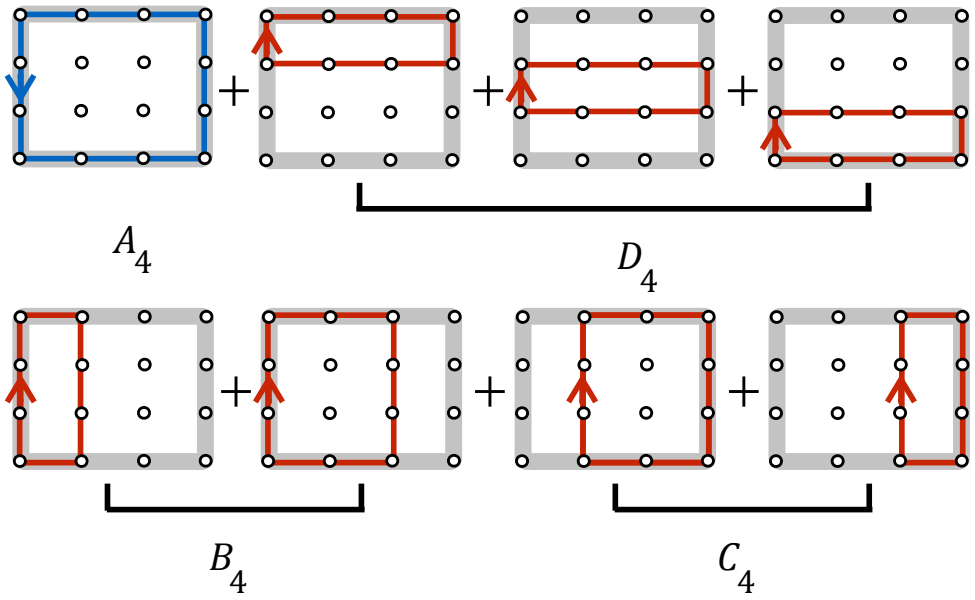


Figure 2.3: Loops in Layer 2 (L_2). $M_4 = L_2 \cup L_1$.

Specifically, let M_k be the final set of selected loops for $k \times k$ grid ($2 \leq k \leq n$) that meets the connectivity, overlapping and low hop count requirements. We construct M_{k+2} by combining M_k with a new set (i.e., layer) of smartly placed loops. The new layer utilizes new wiring resources that are available when expanding from $k \times k$ to $(k+2) \times (k+2)$. The resulting M_{k+2} can also meet all the requirements and deliver superior performance. For example, as shown in Figure 3.1, the grid is logically split into multiple layers with increasing sizes. Let L_k be the set of loops selected for Layer k. Firstly, suppose that we already find a good set of loops for 2×2 grid that connects all the nodes with a low hop count and does not exceed an overlapping of 2 between any neighboring nodes. That set of loops is M_2, which is also L_1 as this is the base case. Then we find another set of loops L_2, together with M_2, can form a good set of loops for 4×4 grid (i.e., $M_4 = L_2 \cup M_2$). The resulting M_4 can connect all the nodes with a low hop count and do not exceed an overlapping of 4 between any neighboring nodes. And so on so forth, until reaching the targeted $n \times n$ grid. In general, we have $M_n = L_{\lfloor n/2 \rfloor} \cup M_{n-2} = L_{\lfloor n/2 \rfloor} \cup L_{\lfloor n/2 \rfloor - 2} \cup M_{n-4} = \ldots = L_{\lfloor n/2 \rfloor} \cup L_{\lfloor n/2 \rfloor - 2} \cup L_{\lfloor n/2 \rfloor - 4} \cup \ldots \cup L_1$.

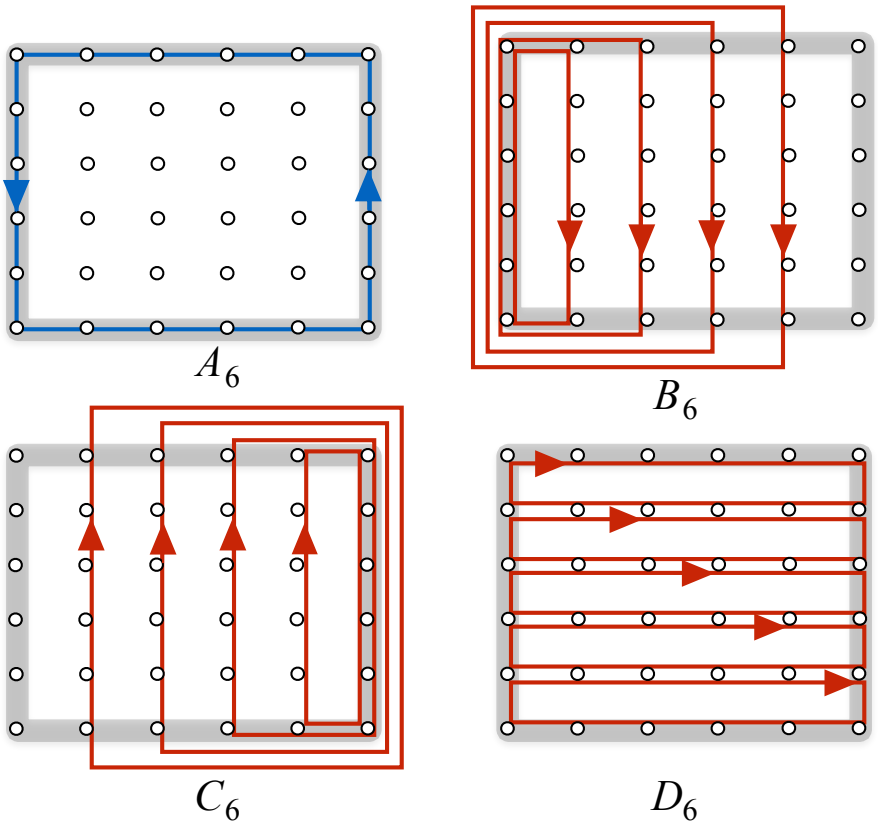


Figure 2.4: Loops in L_3. $M_6 = L_3 \cup L_2 \cup L_1$.

Apparently, the key step in the above progressive process is how to select the set of loops in Layer k, which enables the progression to the next sized grid with low hop count and overlapping. In the next subsections, we walk through several examples to illustrate how it is done to progress from 2×2 grid to 8×8 grid.

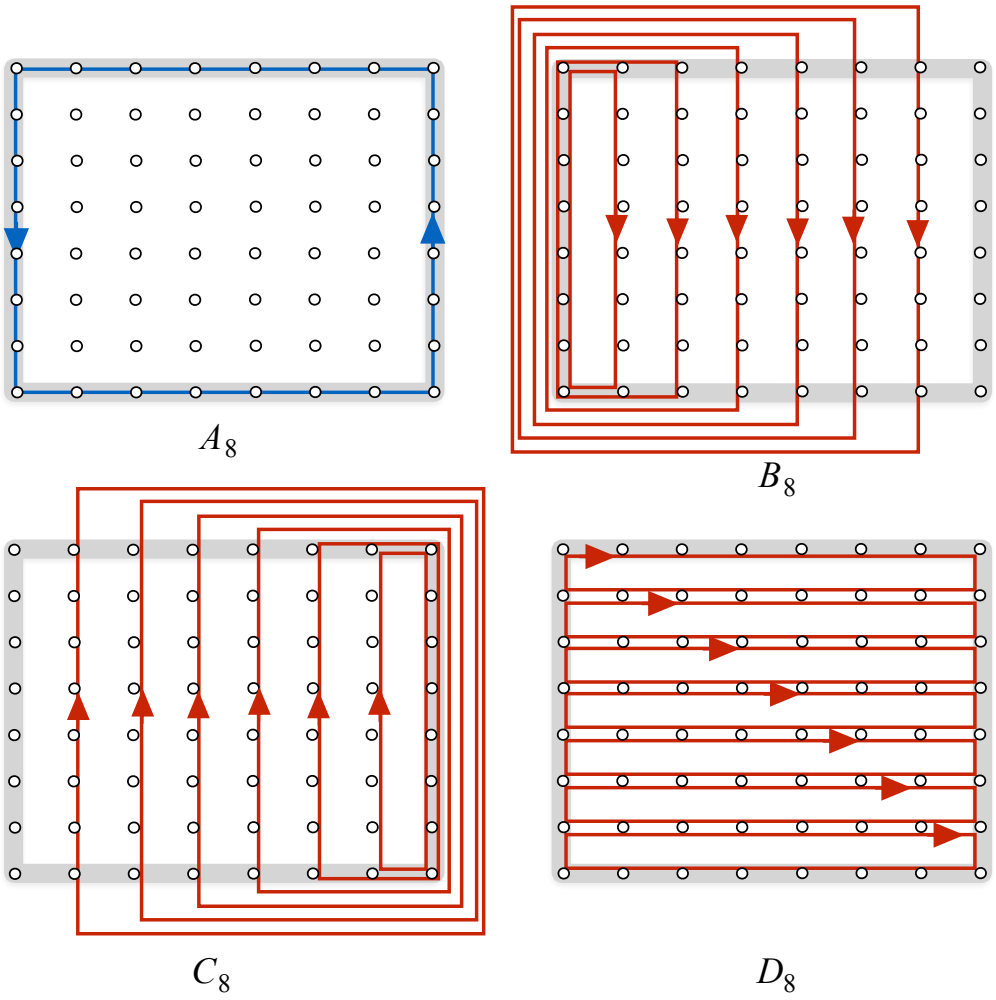


Figure 2.5: Loops in L_4. $M_8 = L_4 \cup L_3 \cup L_2 \cup L_1$.

2.1.2 Examples

2.1.2.1 2×2 Grid

This is the base case with one layer. There are exactly two possible loops, one in each direction, in a 2×2 grid. Both of them are included in $M_2 = L_1$, as shown in Figure 2.2. The resulting M_2 satisfies the requirement that every source and destination pair is connected by at least one loop. The maximum number of loops overlapping between any neighboring nodes is 2, which meets the overlapping cap. This set of loops achieves a very low all-pair average hop count of 1.333, which is as good as the Mesh.

2.1.2.2 4×4 Grid

M_4 consists of loops from two layers. Based on our layered progressive approach, L_1 is from M_2. We select 8 loops to form L_2, as illustrated in Figure 2.3. The 8 loops fall into four groups (from this network size and forward, each new layer is constructed using four groups with the similar heuristics as discussed below). The first group, A_4 (the subscript indicates the size of the grid), has only one anti-clockwise loop. It provides connectivity among the 12 new nodes when expanding from Layer 1 to Layer 2. The loops in the second group, B_4, have the first column as the common edge of the loops, but the opposite edge of the loops moves gradually towards the right (this is more evident in group B_6 in Figure **??**). Similarly, the third group, C_4, uses the last column as the common edge of the loops and gradually moves the opposite edge towards the left. It can be verified that groups B_4 and C_4 provide connectivity between the 12 new nodes in

Layer 2 and the 4 nodes in Layer 1. Since the connectivity among the 4 inner nodes has already been provided by L_1, the connectivity requirement of 4×4 grid is met by having L_1, A_4, B_4 and C_4. The fourth group, D_4, offers additional "shortcuts" in the horizontal dimension.

A very nice feature of the selected M_4 is that the wiring resources are efficiently utilized, as the overlapping between many neighboring node pairs is close to the overlapping cap of 4. For example, for the first (or the last) column, each group of loops has exactly one loops passing through that column, totaling an overlapping of 4, which is the same as the cap. Thus, no overlapping "ration" is under-utilized. For the second column (or the third) column, groups A_4 and D_4 have no loop passing through, and groups B_4 and C_4 have two loops passing through in total. However, note that the final M_4 also includes L_1 which has two loops passing through the second (or the third) column. Hence, the total overlapping of the middle columns is also 4, exactly the same as the cap. Simple counting can show that the overlapping on the horizontal dimension is also 4 for each row. Owing to this efficient use of wiring resource "ration", the all-pair average hop count is 3.93 for the selected set of loops in M_4. The final set is $M_4 = L_2 \cup M_2 = L_2 \cup L_1$.

2.1.2.3 6×6 Grid

M_6 consists of loops from three layers. L_1 and L_2 are from M_4, and L_3 is formed in a similar fashion as 4×4 grid from four groups, as illustrated in Figure **??**. Again, connectivity is provided by M_4 and groups A to C. Together with group D, the number of overlapping on each column and row is 6, thus fully utilizing the allocated wiring

resources.

Additionally, for the purpose of reducing hop count and balancing horizontal and vertical wiring utilization, when we combine M_4 and L_3 to form M_6, every loop in M_4 is reversed and then rotated 90° clockwise[1]. If this slightly changed M_4 is denoted as M_4', the final set can be expressed as $M_6 = L_3 \cup M_4' = L_3 \cup (L_2 \cup L_1)'$, with an all-pair average hop count of 6.07.

2.1.2.4 8×8 Grid

Similar to earlier examples, L_4 consists of loops shown in Figure 2.5. The final set M_8 is $M_8 = L_4 \cup M_6' = L_4 \cup \big(L_3 \cup (L_2 \cup L_1)'\big)'$ with an all-pair average hop count of 8.32.

2.1.3 Formal Procedure

For an $n \times n$ grid, the loops for a routerless NoC design can be recursively found by the procedure shown in Algorithm 1. The procedure is recursive and denoted as RLrec. The procedure begins by generating loops for the outer layer, say layer i, and then it recursively generates loops for layer $i-1$ and so on until the base case is reached or the layer has a single node or empty. Procedure $G(r_1, r_2, c_1, c_2, d)$ is a simple function that generates a rectangular shape loop with corners (r_1, c_1), (r_1, c_2), (r_2, c_1) and (r_2, c_2) and direction d. When processing each layer in this algorithm, procedure G is called repeatedly to generate four groups of loops. Additionally, the generated loops rotate 90

[1] In 4×4 grid, reversal and rotation of M_2 is not necessary because M_2 and M_2' have the same effect on L_1.

Algorithm 1: RLrec

```
   Input  : N_L, N_H; the low and high numbers
1  begin
2      if N_L = N_H then
3          return {}
4      Let M = {}
5      if N_H − N_L = 1 then
6          M = M ∪ G(N_L, N_H, N_L, N_H, clockwise)
7          M = M ∪ G(N_L, N_H, N_L, N_H, anticlockwise)
8          return M
9      M = M ∪ G(N_L, N_H, N_L, N_H, anticlockwise)    // Group A
10     for i = N_L + 1 → N_H − 1 do
11         M = M ∪ G(N_L, N_H, N_L, i, clockwise)      // Group B
12         M = M ∪ G(N_L, N_H, i, N_H, clockwise)      // Group C
13     for i = L → H − 1 do
14         M = M ∪ G(i, i + 1, N_L, N_H, clockwise)    // Group D
15     M′ = RLrec(N_L+1, N_H-1)
16     Reverse and rotate for 90° every loop in M′
17     return M ∪ M′
```

degrees and reverse directions after processing each layer to balance wiring utilization and reduce hop count, respectively. The final loops generated by the RLrec algorithm have an overlapping of at most n.

While it would be ideal if an analytical expression can be derived to calculate the average hop count for this heuristic approach, this seems to be very challenging at the moment. However, it is possible to calculate the average hop count numerically. This result is presented in the evaluation, which shows that our proposed design is highly scalable.

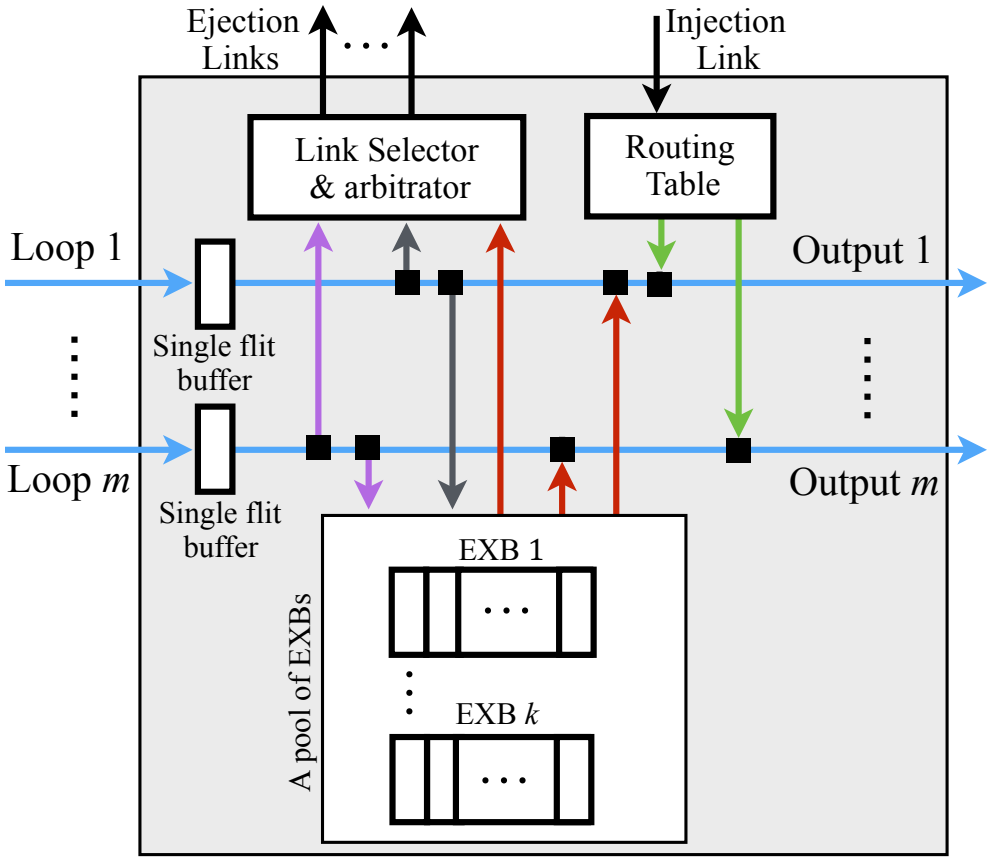


Figure 2.6: Routerless interface components.

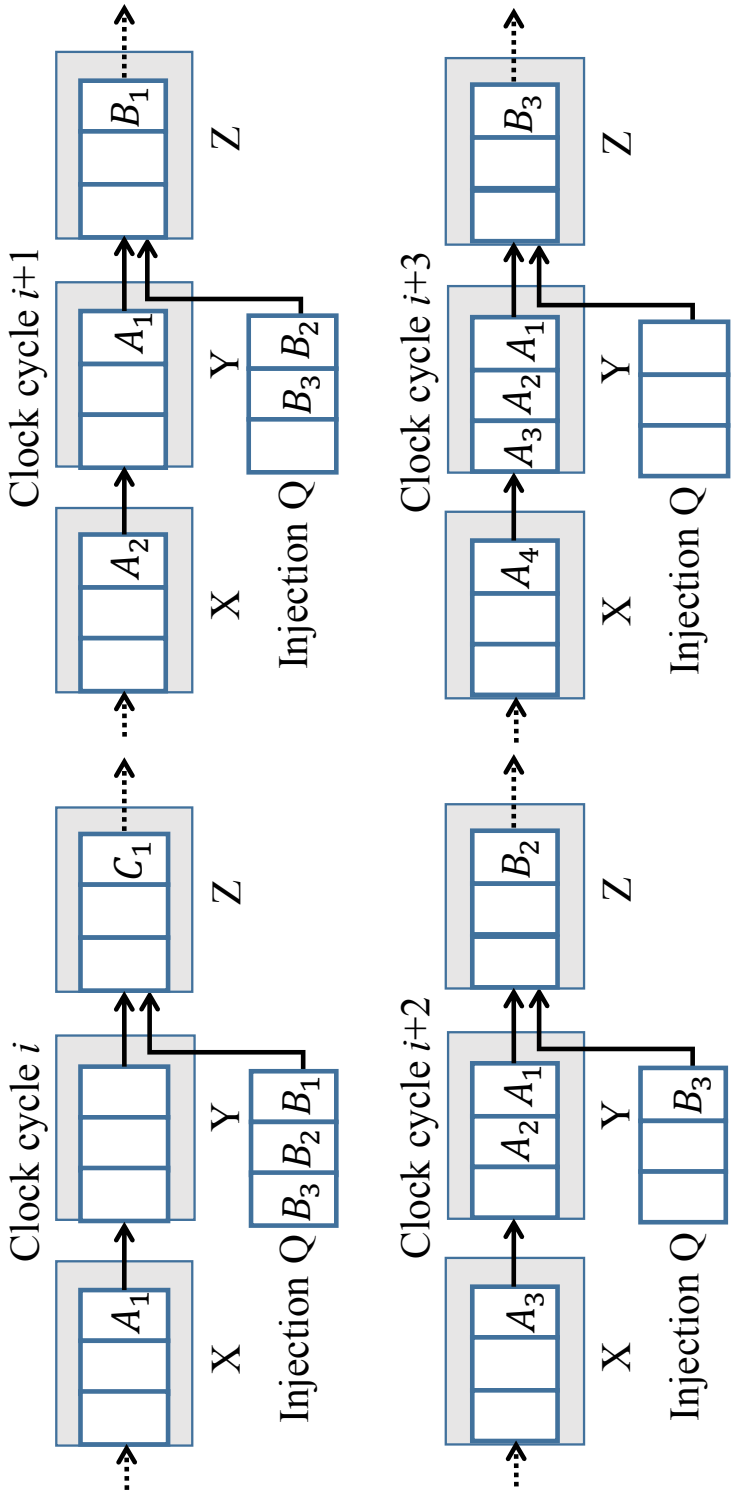


Figure 2.7: Injecting a long packet requires a packet-sized buffer per loop at each hop in prior implementation (X, Y and Z are interfaces).

2.2 Implementation Details

After addressing the key issue of finding a good set of loops, the next important task is to efficiently implement the routerless NoC design in hardware. Because of the routerless nature, no complex switching or virtual channel (VC) structure is needed at each hop, so the hardware between nodes and loops has a small area footprint in general. However, due to various potential network abnormalities such as deadlock, livelock, and starvation, a certain number of resources are required to guarantee correctness. If not addressed appropriately, this may cause substantial overhead that is comparable to router-based NoCs. In this section, we propose a few effective techniques to minimize those overhead.

In a routerless NoC, each node uses an interface (RL interface) to interact with one or multiple loops that pass through this node. Figure 2.6 shows the main components of a RL interface. While details are explained in the following subsections, the essential function of the interface includes injecting packets into a matching loop based on connectivity and availability, forwarding packets to the next hop on the same loop, and ejecting packets at the destination node. Notice that packets cannot switch loops once injected. All the loops have the same width (e.g., 128-bit wires).

2.2.1 Injection Process

2.2.1.1 Extension Buffer Technique

A loop is basically a bundle of wires connected with flip-flops at each hop (Figure 2.6). At clock cycle i, a flit arriving at the flip-flop of loop l must be consumed immediately by

either being ejected at this node or forwarded to the next hop on loop l through output l. If no flit arrives at loop l (thus not using output l), the RL interface can inject a new flit on loop l through output l. However, it is possible that an injecting packet consists of multiple flits and requires several cycles to finish the injection, during which other flits on loop l may arrive at this RL interface. Therefore, addition buffer resources are needed to hold the incoming flits temporarily.

If routerless NoC uses the scheme proposed in prior ring-based work (e.g., IMR [33]), a full packet-sized buffer per loop at each hop would be needed to ensure correctness, which is very inefficient. As illustrated in Figure 2.7, a long packet B with multiple flits is waiting for injection (there is no issue if it is a short single-flit packet). At clock cycle i, the injection is allowed because packet B sees that no other flit in Interface Y is competing with B for the output to Interface Z. From cycle $i+1$ to $i+3$, the flits of B are injected sequentially. However, while packet B is being injected during these cycles, another long packet A may arrive at Interface Y. Because RL interfaces do not employ flow control to stop the upstream node, Interface Y needs to provide a packet-sized buffer to temporarily store the entire packet A. A serious inefficiency lies in the fact that, if there are m loops passing through a RL interface, the interface needs to have m packet-sized buffers, one for each loop.

To address this inefficiency, we notice that an interface injects packets one at a time, so not all the loops are affected simultaneously. Based on this observation, we propose the extension buffer technique to share the packet-sized buffer among loops. As shown in Figure 2.6, each loop has only a flit-sized buffer, but the interface has a pool of extension buffers (EXBs). The size of each EXB is the size of a long packet, so when a loop is

"extended" with an EXB, it would be large enough to store a long packet. Minimally, only one EXB is needed in the pool, but having multiple EXBs may have slight performance improvement. This is because another injection might occur while the previous EXB is not entirely released (drained) due to a previous injection (e.g., clock cycle $i+3$ in Figure 2.7). However, as shown later in the evaluation, the performance difference is negligible. As a result, our proposed technique of using one shared EXB can essentially achieve the same objective of ensuring correctness as IMR but reduces the buffer requirement by m times. This is equivalent to an 8X saving in buffer resources in 8×8 networks and 16X saving in 16×16 networks.

2.2.1.2 Injection Process

The injection process with the use of EXBs is straightforward. To inject a packet p of n_f flits, the first step is to look up a small routing table to see which loop can reach p's destination. The routing table is pre-computed since all the loops are pre-determined. The packet p then waits for the loops to become available (i.e., having sufficient buffer space). Assume l is a loop that has the shortest distance to the destination among all the available loops. When the injection starts, the interface holds the output port of l for n_f cycles to inject p, and assigns a free EXB to l if $n_f > 1$ and l is not already connected to another EXB. During those n_f cycles, any incoming flit through the input port of l is enqueued in the extension buffer. The EXB is released later when its buffer slots are drained.

2.2.2 Ejection Process

The ejection process starts as soon as the head flit of a packet p reaches the RL interface of its destination node. The interface ejects p, one flit per cycle. Once p is ejected, the interface will wait for another packet to eject. There is, however, a potential issue with the ejection process. While unlikely, a RL interface with m loops may receive up to m head flits simultaneously in a given cycle that are all destined to this node. Because any incoming packets need to be consumed immediately and the packets are already at the destination, the interface needs to have m ejection links in order to eject all the packets in that cycle. As each eject link has the same width as the loop (i.e., 128-bit), this incurs substantial hardware overhead.

To reduce this overhead, we utilize the fact that the actual probability of having k packets ($1 < k \leq m$) arriving at the same destination in the same cycle is low, and this probability decreases drastically as k increases. Based on this observation, we propose to optimize for the common case where only e ejection links are provided ($e \ll m$). If more than e packets arrive at the same cycle, $(k-e)$ packets are forwarded to the next hop. Those deflected packets will continue on their respective loops and will circle back to the destination later. As shown in the evaluation, having two ejection links can reduce the percentage of circling packets to be below 1% on average (1.6% max) across the benchmarks. This demonstrates that this is a viable and effective technique to reduce overhead.

2.2.3 Avoiding Network Abnormalities

As network abnormalities are theoretically possible but practically unlikely scenarios, our design philosophy is to place very relaxed conditions to trigger the handling procedures, so as to minimize performance impact while guaranteeing correctness.

2.2.3.1 Livelock Avoidance

A livelock may occur if a packet circles indefinitely and never gets a chance to eject. We address this issue by having a byte-long circling counter at each head flit with an initial value of zero. Every time a packet reaches its destination interface and is forced to be deflected, the counter is incremented by 1. If the circling counter of a packet p reaches 254 but none of the ejection link is available, the interface marks one of its ejection links as reserved and then deflects p for the last time. The marked ejection link will not eject any more packets after finishing the current one, until p circles back to the ejection link (by then the marked ejection link will be available; otherwise there is a possible protocol-level deadlock, discussed shortly). Once p is ejected, the interface will unmark the ejection link for it to function normally. Due to the extremely low circling percentage (maximum 3 times of circling for any packet in our simulations), this livelock avoidance scheme has minimal performance impact.

2.3 Deadlock Avoidance

With no protocol-level dependence at injection/ejection endpoints, routing-induced deadlock is not possible in routerless NoCs as packets arriving at each hop are either ejected or forwarded immediately. Hence, a packet can always reach its destination interface without being blocked by other packets. The above livelock avoidance ensures that the packet can be ejected within a limited number of circlings.

With more than one dependent packet types (or message classes), the marked ejection link in the above livelock avoidance scheme may not be able to eject the current packet (say a request packet) in the ejection queue, because the associated cache controller cannot accept new packets from the ejection queue (i.e., input of the controller). This may happen when the controller itself is waiting for packets (say a reply packet) in the injection queue (i.e., output of the controller) to be injected into the network. A potential protocol-level deadlock may occur if that reply packet cannot be injected, such as the loop is full of request packets that are waiting to be ejected. If no dependency between packets type (or message classes) exists or the system has only one message class, then there is no possibilities for protocol-level deadlock to occur.

To avoid such protocol-level deadlock, the conventional approach is to have a separate physical or virtual network for each dependent packet type. For example, in Mesh network the NoC is duplicated for each message class for the purpose of avoiding protocol-level deadlock. The overhead of this solution is extremely heavy on area and power budgets and it becomes intolerable as the NoC scale to higher number of nodes. While similar approach can be used for routerless NoCs, here we propose a less resource

demanding solution which a separate ejection link for each message class to each cache controller. These ejection links work independently and simultaneously at any time for each controller. In addition, packets belonging to different message classes will no longer be mixed in the same ejection queue and according to the nature of RL design packets constantly circulating in loop until it gets a chance for ejection, hence, the resource dependency between requests and reply packets is broken.

2.3.0.1 Starvation Avoidance

The last corner case we address is starvation. With the previous livelock and deadlock handling, if a packet is consumed at its destination RL interface, the interface can use the free output to inject a new packet. However, it is possible that a particular interface X is not the destination of any packets and there is always a flit passing through X every single cycle. This never occurred in any of our experiments as it is practically impossible that a cache bank is not accessed by any other cores. However, it is theoretically possible and, when occurred, prevents X from injecting new packets. We propose the following technique to avoid starvation for the completeness of the routerless NoC design. If X cannot inject a packet after a certain number of clock cycles (a very long period, e.g., hundreds of thousand of cycles or long enough to have negligible impact on performance), X piggybacks the next passing head flit f with the ID of X. When f is ejected at its destination interface Y, instead of injecting a new packet, Y injects a single-flit no-payload dummy packet that is destined to X. When the dummy packet arrives at X, X can now inject a new packet by using the free slot created by the dummy

packet. This breaks the starvation configuration.

2.3.1 Interface Hardware Implementation

Figure 2.6 depicts the main components of a RL interface. We have explained the extension buffers (EXBs), single-flit buffers, routing table, and multiple ejection links in the previous subsections. The arbitrator receives flits from input buffers and selects up to *e* input loops for ejection based on the oldest first policy. The arbitrator contains a small register that holds the arbitration results. The link status selector is a simple state machine associated with the loops. It monitors the input loops and arbitration results, and changes the state of the loops (e.g., ejection, stall in extension buffers, etc.) in the state machine. There are several other minor logic blocks that are not shown in Figure 2.6 for better clarity. Note that the RL interface does not use the information of neighboring nodes, which differs from most conventional router-based NoCs that need credits or on/off signals for handshaking.

To ensure the correctness of the proposed interface hardware, we implement the design in RTL Verilog that includes all the detailed components. The Verilog implementation is verified in Modelsim, synthesized in Synopsys Design Compiler, and placed and routed using Cadence Encounter tool. We use the latest 15nm process NanoGate FreePDK 15 Cell Library [34] for more accurate evaluation. As a key result, the RL interface is able to operate at up to 4.3GHz frequency while keeping the packet forwarding process in one clock cycle. This is fast enough to match up with most commercial many-core processors. Injecting packets may take an additional cycle for table look-up.

In the main evaluation below, both the interfaces and cores are operating at 2GHz.

2.4 Evaluation methodology

We evaluate the proposed routerless NoC (RL) extensively against Mesh, EVC, and IMR in Booksim [29]. For synthetic traffic workloads, we use uniform, transpose, bit reverse, and hotspot (with 8 hotspots nodes). Booksim is warmed up for $10,000$ clock cycles and then collects performance statistics for another $100,000$ cycles at various injection rates. The injection rate starts at 0.005 flit/node/cycle and is incremented by 0.005 flit/node/cycle until the throughput is reached. Moreover, we integrate Booksim with Synfull [9] for performance study of PARSEC [15] and SPLASH-2 [45] benchmarks. Power and area studies are based on Verilog post-synthesis simulations, as described in Section 2.3.1.

In the synthetic study, each router in Mesh is configured with relatively small buffer resources, having 2 VCs per link and 3 flits per VC. The link width is set to 256-bit. Also, the router is optimized with lookahead routing and speculative switch allocation to reduce pipeline stages to 2 cycles per router and 1 cycle per link. EVC has the same configuration as Mesh except for one extra VC that is required to enable express channels. For IMR, the ring set is generated by the evolutionary approach described in [33]. To allow a fair comparison with RL, the maximum number of overlapping cap, for both RL and IMR, is set to n for $n \times n$ NoC. We also follow the original paper to faithfully implement IMR's network interface. Each input link in an IMR's interface is attached with a buffer of 5 flits and the link width is set to 128-bit (the same as the original paper). In RL, loops are generated by RLrec algorithm and accordingly the routing table for each node is calculated. Each interface is configured with two ejection links and each input

link has a flit-size buffer. Also, an EXB of 5 flits is implemented in each interface. The link width is 128-bit (the same as IMR). In all the designs, packets are categorized into data and control packets where each control packet has 8 bytes and each data packet has 72 bytes. Accordingly, data packets in Mesh, EVC, IMR, and RL are of 3, 3, 5 and 5 flits, respectively, and the control packets are of a single flit.

For benchmark performance study, we also add 2D Mesh with various configurations as well as a 3D Cube design into the comparison. RL has the same configuration as the synthetic study. For 2D Mesh, we use 9 configurations, each having the configuration M(x, y) where $x \in \{1, 2, 3\}$ is the router delay and $y \in \{1, 2, 3\}$ is the buffer size, i.e., routers with 1-cycle, 2-cycle and 3-cycle delay, and with 1-flit, 2-flit and 3-flit buffer size. 3D Cube is configured with 2 VCs per link, 3 flits per VC, and 2-cycle per hop latency.

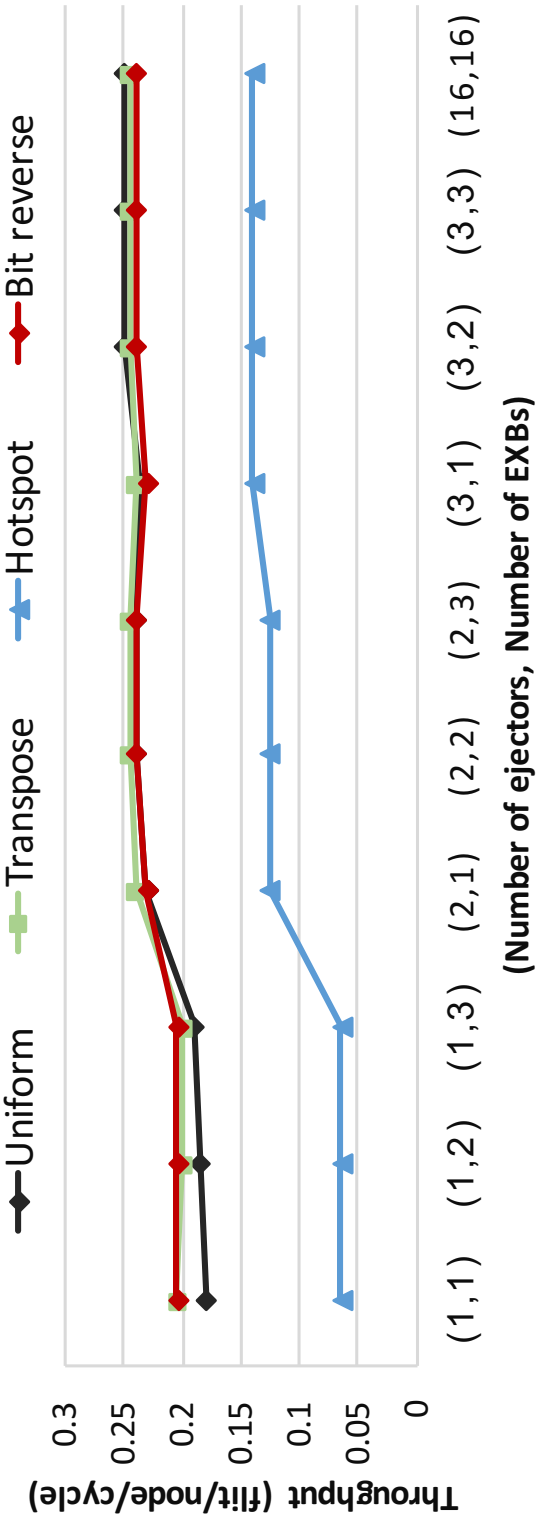


Figure 2.8: Throughput of routerless NoC under different number of ejection links and extension buffers (EXBs).

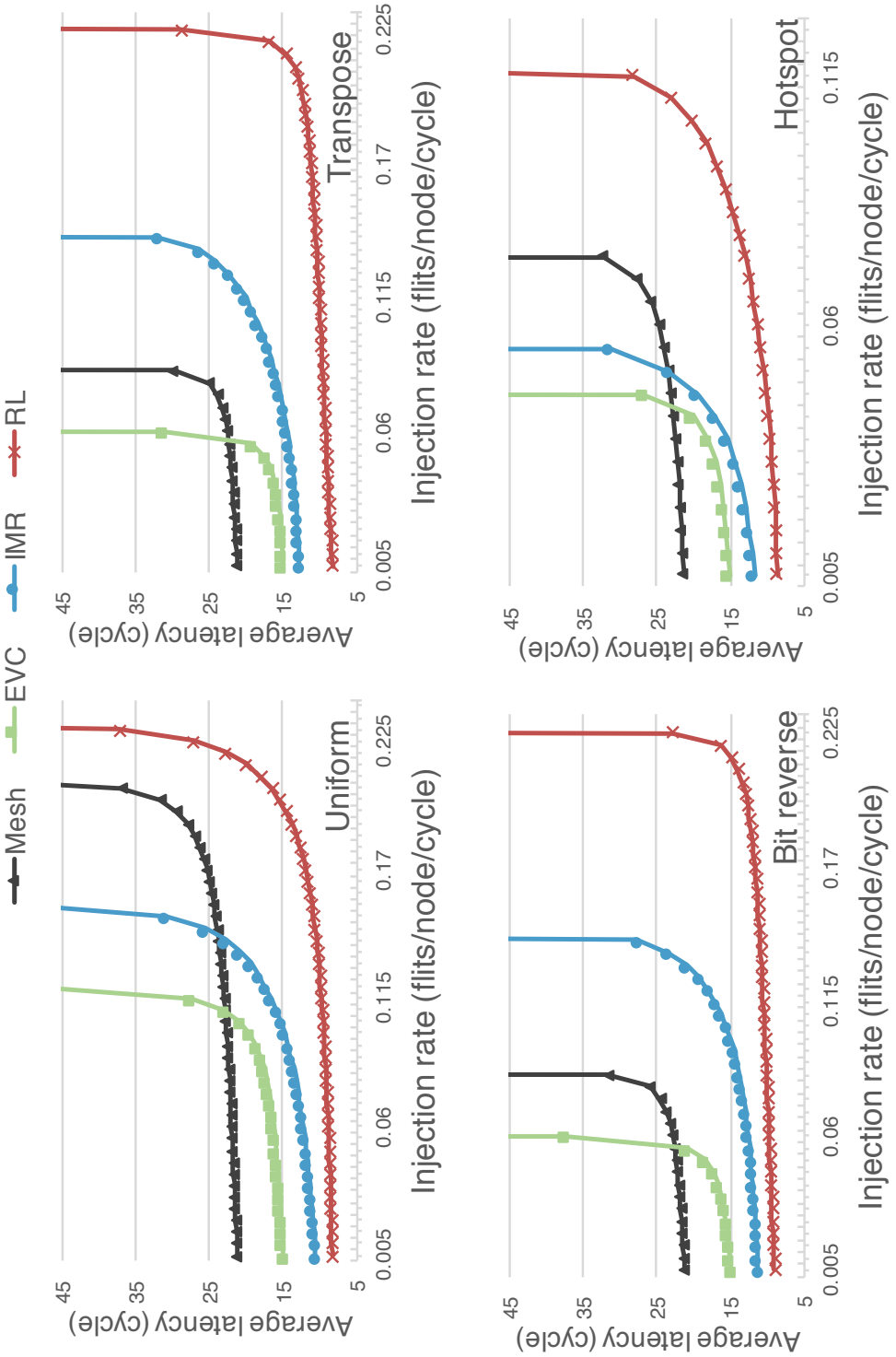


Figure 2.9: Performance comparison for synthetic traffic patterns.

2.5 Results and Analysis

2.5.1 Ejection Links and Extension Buffers

The proposed RL scheme is flexible to use any number of ejection links and EXBs. On the ejection side, the advantages of having more ejection links are higher chance for packet ejection and lower chance for packet circling in a loop. However, adding more ejection links complicates the design of the interface and leads to additional power and area overhead in the interface and the receiving node. On the injection side, EXBs have a direct effect on the injection latency of long packets. Recall that, a loop must be already attached with an EXB or a free EXB is available to be able to inject a long packet. Similar to ejection links, having more EXBs can lower injection latency but incur larger area and power overhead.

We studied the throughput of RL with different configurations of ejection links and EXBs on various synthetic traffic patterns. The NoC size for this study is 8×8. The results are shown in Figure 2.8. In the figure, each configuration is denoted by (x,y) where x is the number of ejection links and y is the number of EXBs. The basic and best in terms of area and power overhead is $(1,1)$ configuration but it has the worst performance. By adding up to three EXBs with a single ejection link, the throughput is only slightly changed (less than 5%). This indicates that the number of EXBs is not very critical to performance, and it is possible to use only one EXB for injecting long packets while saving buffer space.

For $(2,1)$ configuration, it doubles the chance for packet ejection when compared to $(1,x)$ configurations. The throughput is notably improved by an average of 38% for

all the patterns when compared to $(1,1)$ configurations. For instance, hotspot traffic pattern has 0.125 throughput in $(2,1)$ configuration but only 0.065 in $(1,1)$, a 92.5% improvement). However, on top of $(2,1)$ configuration, adding up-to three EXBs (i.e., $(2,3)$) improves throughput only by 5% on average.

Given all the results, we choose the $(2,1)$ configuration as the best trade-off point, and use it for the remainder of this section. We also plot the $(16,16)$ configuration which is the ideal case (no blocking in injection or ejection may happen). As can be seen, $(2,1)$ is very close to the ideal case. Section 2.5.3 provides a detailed study for the number of times packet circling in loops for the $(2,1)$ configuration.

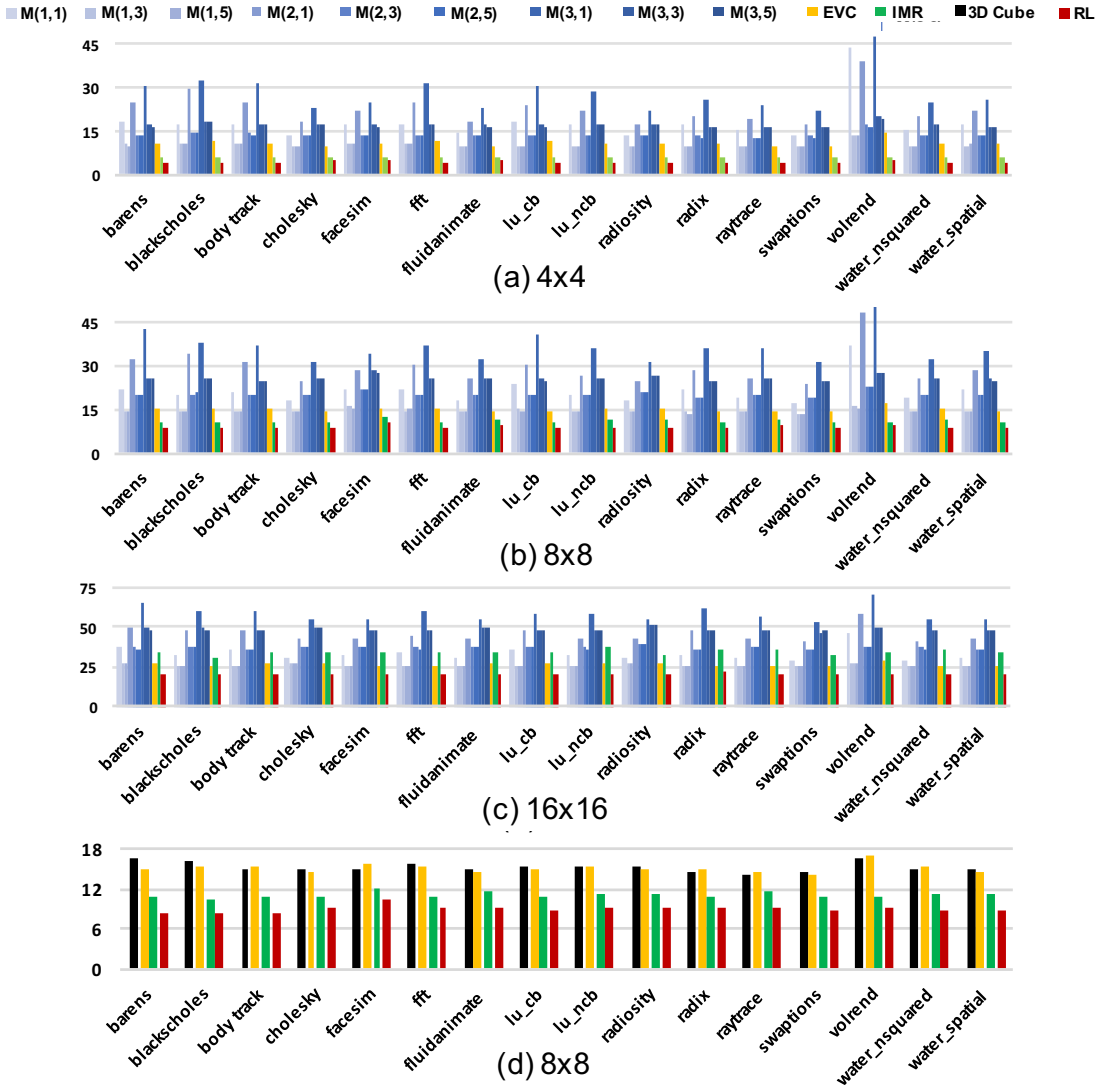


Figure 2.10: PARSEC and SPLASH-2 benchmark performance results (y-axis represents average pack latency in cycles.) RL is compared with different Mesh configurations, EVC, and IMR in (a), (b) and (c). In (d), RL is also compared with a 3D Cube.

2.5.2 Synthetic Workloads

Figure 2.9 plots the performance results of four synthetic traffic patterns for an 8×8 NoC. RL has the lowest zero-load packet latency in all four traffic patterns. For example, in uniform random, the zero-load packet latency is 21.2, 14.9, 10.5, and 8.3 cycles for Mesh, EVC, IMR, and RL, respectively. When averaged over the four patterns, RL has an improvement of 1.59x, 1.43x, and 1.25x over Mesh, EVC, and IMR, respectively. RL achieves this due to low per hop latency (one cycle) and low hop count.

In terms of throughput, the proposed RL also has advantage over other schemes. For example, the throughput for hotspot is 0.08, 0.05, 0.06, and 0.125 (per flit/node/cycle) for Mesh, EVC, IMR, and RL, respectively. In fact, RL has the highest throughput for all the traffic patterns. When averaged over the four patterns, RL improves throughput by 1.73x, 2.70x, and 1.61x over Mesh, EVC, and IMR, respectively. This is mainly owing to the better utilization of wiring resources in RL. Note that, EVC has a lower throughput than Mesh as EVC is essentially a scheme that trades off throughput for lower latency at low traffic load.

2.5.3 PARSEC and SPLASH-2 Workloads

We utilize Synfull and Booksim to study the performance of RL, 2D Mesh with different configurations, EVC, IMR, and a 3D Cube under 16 PARSEC and SPLASH-2 benchmarks. The NoC sizes under evaluation are 4×4, 8×8 and 16×16 for RL, 2D Mesh, EVC and IMR, and $4 \times 4 \times 4$ for 3D cube. Figure 2.10 shows the results.

In Figure 2.10(a)-(c), RL is compared against 2D Mesh, EVC and IMR. From the

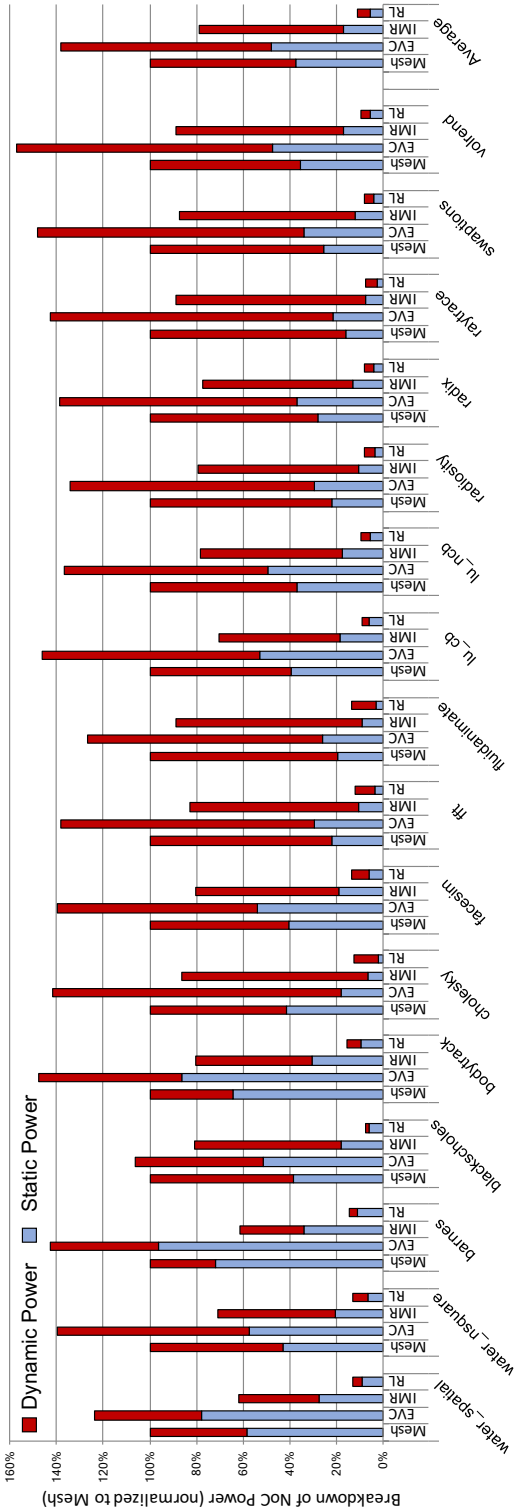


Figure 2.11: Breakdown of power consumption for different PARSEC and SPLASH-2 workloads (normalized to Mesh).

figures, the best configuration for Mesh is M(1,5) (i.e. per hop latency of 1 and buffer size of 5) and the worst is M(3,1). Lowering per hop latency in Mesh helps to improve overall latency, and reducing buffer sizes may cause packets to wait longer for credits and available buffers. The average packet latency of RL in 4×4, 8×8, and 16×16 are 4.3, 8.9 and 20.1 cycles, respectively. This translates into an average latency reduction of RL over M(1,5) by 57.8%, 38.4% and 22.2% in 4×4, 8×8 and 16×16, respectively. The IMR rings in a 16×16 network are very long and seriously affect the latency. RL reduces the average latency by 23.3% over EVC and 41.2% over IMR.

As shown in Figure 2.10(d), the performance of 3D cube is clearly better than all the Mesh configurations in (b) mainly due to lower hop count and larger bisection bandwidth. Despite this, RL still offers better performance than 3D cube. The average latency of RL is 8.9 cycles, which is 41% lower than the 15.2 cycles of 3D cube.

2.5.4 Power

Figure 2.11 compares the power consumption of Mesh (i.e. M(2,3)), EVC, IMR and RL for different benchmarks, normalized to the Mesh. All the power consumption shown in this Figure are reported after P&R in NanGate FreePDK 15 Cell Library [34] by Cadence Encounter. The activity factors for the power measurement are obtained from Booksim, and the power consumption includes that of all the wires.

The average dynamic power consumption for RL is only 0.26mW, and for Mesh, EVC and IMR the average is 2.88mW, 4.27mW and 2.91mW, respectively. Because RL has no crossbar, it requires only 9%, 6.1% and 8.9% of the dynamic power consumed

by Mesh, EVC and IMR, respectively. Meanwhile, static power is mostly consumed by buffers. Unlike Mesh, EVC and IMR, RL has a much lower buffer requirement. As a result, RL consumes very low static power of 0.18mW on average, while Mesh, EVC and IMR consume 1.39mW, 1.64mW and 0.58mW, respectively. Adding dynamic and static power together, on average, RL reduces the total NoC power consumption by 9.48X, 13.1X and 7.75X over Mesh, EVC and IMR, respectively.

2.5.5 Area

Figure 2.12 compares the router or interface area of the different schemes we are studying. The results are obtained from Cadence Encounter after P&R[2]. We also add a bufferless design to the comparison. The largest area is 60731μm^2 for EVC (not shown in the figure) followed by 45281μm^2, 28516μm^2, 20930μm^2 and 6286μm^2 for Mesh, Bufferless, IMR and RL, respectively. The EXB and ejection link sharing techniques as well as the simplicity of the RL interface are the main contributors for the significant reduction of area overhead. Overall, RL has an area saving of 89.6%, 86.1%, 77.9% and 69.9% compared with EVC, Mesh, Bufferless[3] and IMR, respectively.

The wiring area is not included as wires are spread throughout the metal layers and cannot be compared directly. We do acknowledge that IMR and RL use more wiring resources than other designs. RL uses a small percentage of middle metal layers for wires and, as a result, more repeaters are needed. The total area for all the link repeaters is

[2]Our CAD tools limit P&R for processing cores.

[3]In addition to area reduction, RL also has 2.8X higher throughput (under UR) and 64.3% lower latency than bufferless NoC.

$0.127mm^2$ which is 4.3% of the mesh router area. However, as middle layers are above the logic area, RL is unlikely to increase the chip size.

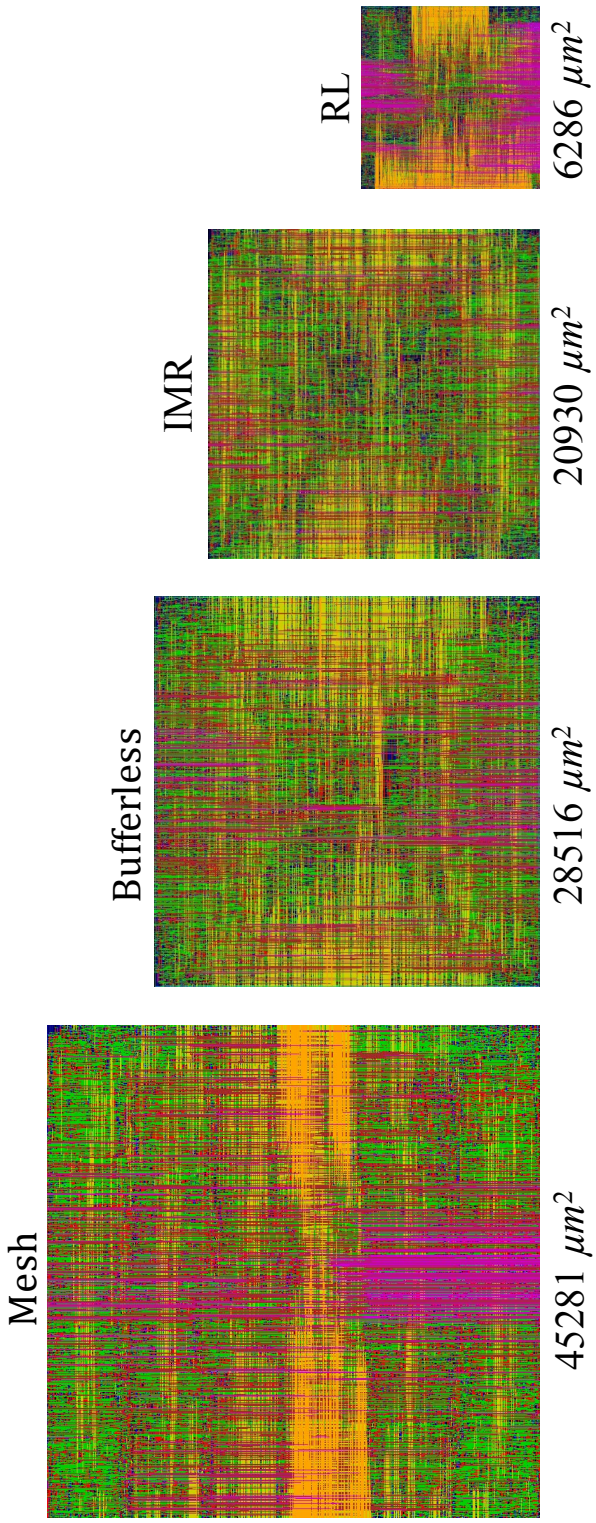


Figure 2.12: Area comparison under 15nm technology.

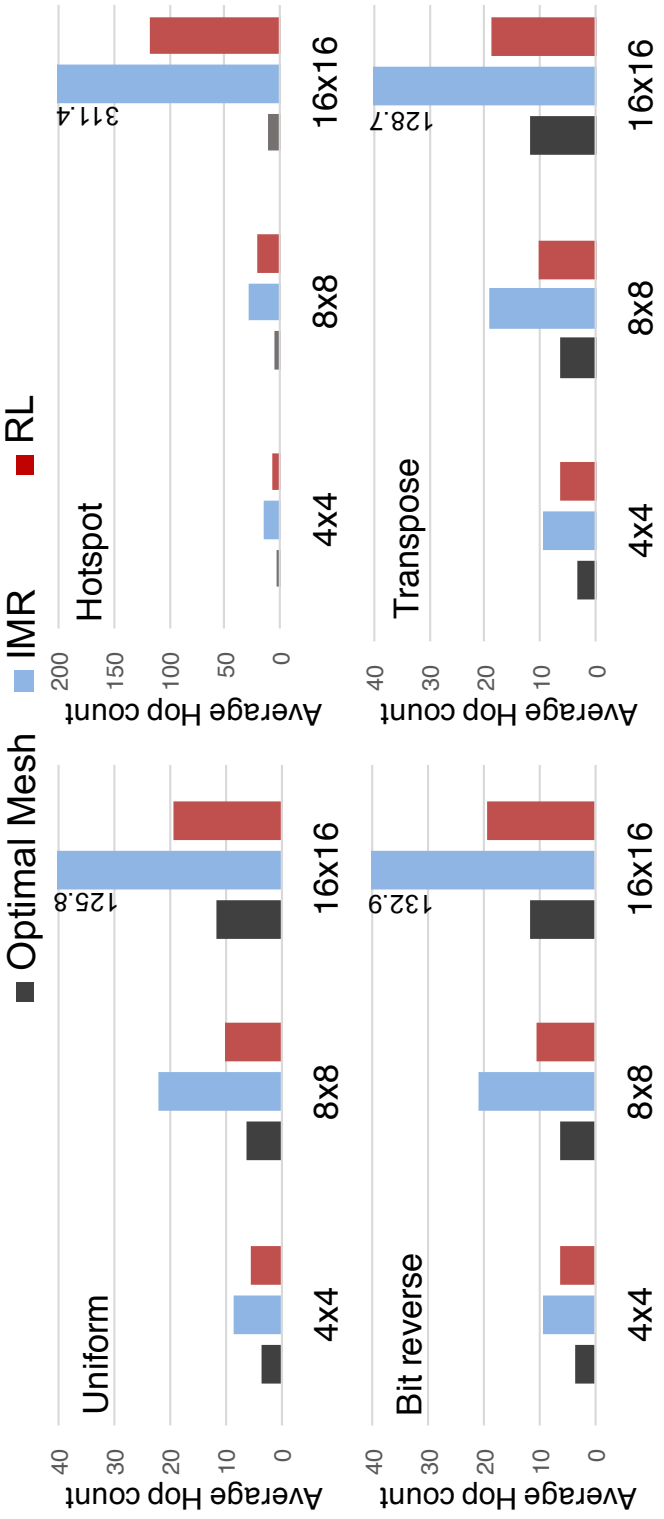


Figure 2.13: Average hop count for synthetic workloads.

2.6 Discussion

2.6.1 Scalability and Regularity

Figures 2.9 and 2.10 already showed the advantage of RL in terms of latency and throughput for large networks. Figure 2.13 further compares the average hop count (zero-load hop count) of RL, IMR, and optimal Mesh. As can be seen, IMR has very high average hop count because of its lengthy rings. In contrast, the average hop count of RL is only slightly higher than optimal Mesh. Note that RL achieves this low hop count without having the switch capability of conventional routers.

Routerless NoC is not as irregular as it appears in the figures. In our actual design and evaluation, all the RL interfaces use the same design (some ports are left unused if no loops are connected), so the main irregularity is the way that links form loops. One way to quantify the degree of link irregularity is how many different possible lengths of links, which is $n-1$ for $n \times n$ NoC. This degree is similar to that of Flattened Butterfly [30] and MECS [23].

2.6.2 Average Overlapping

We discussed before that as long as the overlapping cap is met, it is beneficial to approach this cap for as many neighboring node pairs as possible to increase resource utilization and improve performance. Table 2.1 presents this statistics for RL and IMR. It can be seen that the average overlapping between adjacent nodes in RL is at least 20% more than that of IMR. Also, the longest loop in RL is always shorter than the longest ring in

Table 2.1: Average overlapping and loops/rings in RL/IMR

Network	Overlap cap	Avg overlap(%) of links	Max loops/ rings in node	Avg loops/ rings(%) in node	Longest loop/ring
RL 4x4	4	3.33 (83.3%)	6	5 (62%)	12
IMR 4x4	4	2.33 (58.3%)	4	3.5 (43%)	14
RL 8x8	8	6 (75%)	14	10.5 (65%)	28
IMR 8x8	8	4.71 (58.9%)	10	8.2 (54%)	48
RL 16x16	16	11.33 (70.8%)	30	21.2 (66%)	60
IMR 16x16	16	8.13 (50.8%)	18	15.2 (47%)	240

IMR, and the difference increases as the NoC gets bigger. Shorter loops reduce average hop count and offer a lower latency. For example, in 16×16 the longest loop in RL is of 60 nodes while in IMR it is of 240 nodes.

2.6.3 Impact on Latency distribution

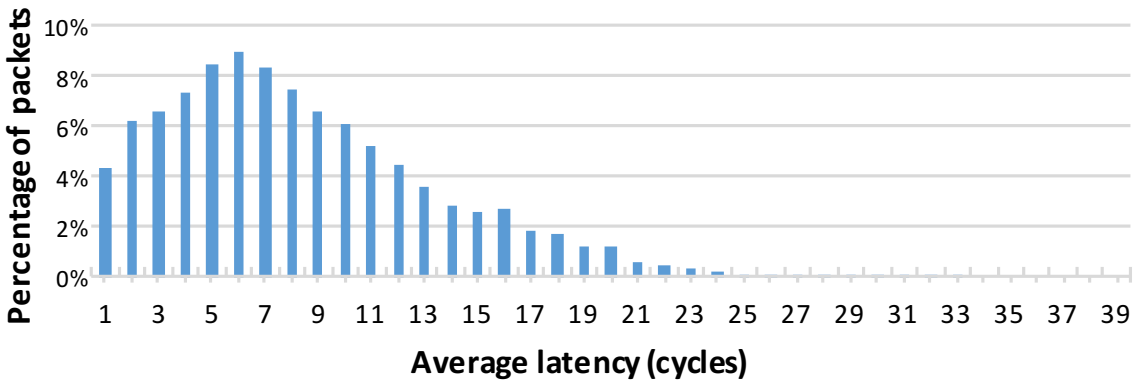


Figure 2.14: Latency distribution of benchmarks for RL 8×8 NoC.

The extension buffer technique and the reduced ejection link technique save buffer

resources at the risk of increasing packet latency. Figure 2.14 shows distribution of average packet latency, averaged over different benchmarks. The RL interface is configured the same as previous sections with one EXB and two ejectors. The take away message from the figure is that the two techniques has minimal impact on latency under tight resource allocation. For example, the average packet latency is only 8.3 cycles for RL, and only 0.71% of the packets having latency larger than 20 cycles, with the largest being 39 cycles. The tail in the latency distribution is thin and short.

2.6.4 RL for $n \times m$ Chip

The RL design can be easily extended to any $n \times m$ network sizes. The RL interface design and functionalities remain unchanged. The RLrec algorithm needs to be modified slightly. With rectangular shapes instead of squares, N_L and N_H are not sufficient to denote the four corners of a layer. Two more variables are needed to specify the corners of a layer correctly. For Instance, N_Lr and N_Hr for low and high rows, and N_Lc and N_Hc for low and high columns. Once a layer is correctly specified, the four groups of loops can be generated in similar fashion. The rotation step is skipped as this is not possible for rectangular networks, but the reversing direction step remains. The overlapping calculation needs to reflect the orientation of the rectangular loops as well.

2.7 Highlights on implementing Routerless in Booksim

2.7.1 How Booksim works

Booksim [29] is a cycle-accurate simulator developed at Standford and has been validated for accuracy for that RTL of Network-on-chip routers. Booksim comes with a broad set of typologies, such as Butterfly, flattened butterfly, torus, mesh, etc. and a broad set of routing algorithms such as XY-routing and dimension order routing. Booksim is very easy to customize for any topology or router microarchitecture. The structure of Booksim's code is fairly simple and easy to grasp. The main configuration parameters are all defined in `booksim_config.cc` file including the total number of clock cycles the simulator is expected to run, the configuration of the routers, the type of topology, injection rate, and traffic pattern. One can set up all configurations and put them in a single file and pass this file to Booksim to customize the simulator parameters. Example of such files can be found in textttsrc/examples folder. Booksim also allow you to define your own ad hoc topology. It includes a simple example file (`networks/anynet.cpp`) that can be walked of through the steps to add new nodes and new links, for any custom topology to the simulator process.

There are two main branches in Booksim *traffic manager* and *network*. The traffic manager includes the traffic pattern module and the injection to the network and ejection from the network modules. On the other side is the network branch which includes all routers and links between routers. The traffic manager executes the function `_Step()` every time the global clock cycle counter increases to feeds the network with packets generated by a traffic pattern. The network will deliver such packets to their destinations

by going through the pipeline of every router on the path from the source router to the destination router. Once the packet reached its destination, it will be ejected and sent to the traffic manager. Moreover, in each `_Step()` the traffic manager checks if there is any received packet from the network. If there is, the necessary statistics are recorded (mainly latency and hopcount) and the packet is deleted.

The network, all routers, and all links in Booksim implement the virtual functions

1. `ReadInputs()`
2. `Evaluate()`
3. `WriteOutputs()`

inherited from `TimedModule` class. These functions must be invoked by all implementers (i.e. network, routers, and links) every clock cycle and are called by network, routers, and links. The traffic manager, which holds pointer to the network invokes the network's `ReadInputs, Evaluate, WriteOutputs`. The network implements each of `ReadInputs, Evaluate, WriteOutputs` by looping through all the pointers it holds for routers and links to invoke `ReadInputs, Evaluate, WriteOutputs`, respectively, of the related component (router or channel). Figure 2.15 depict the execution hierarchy in every clock cycle.

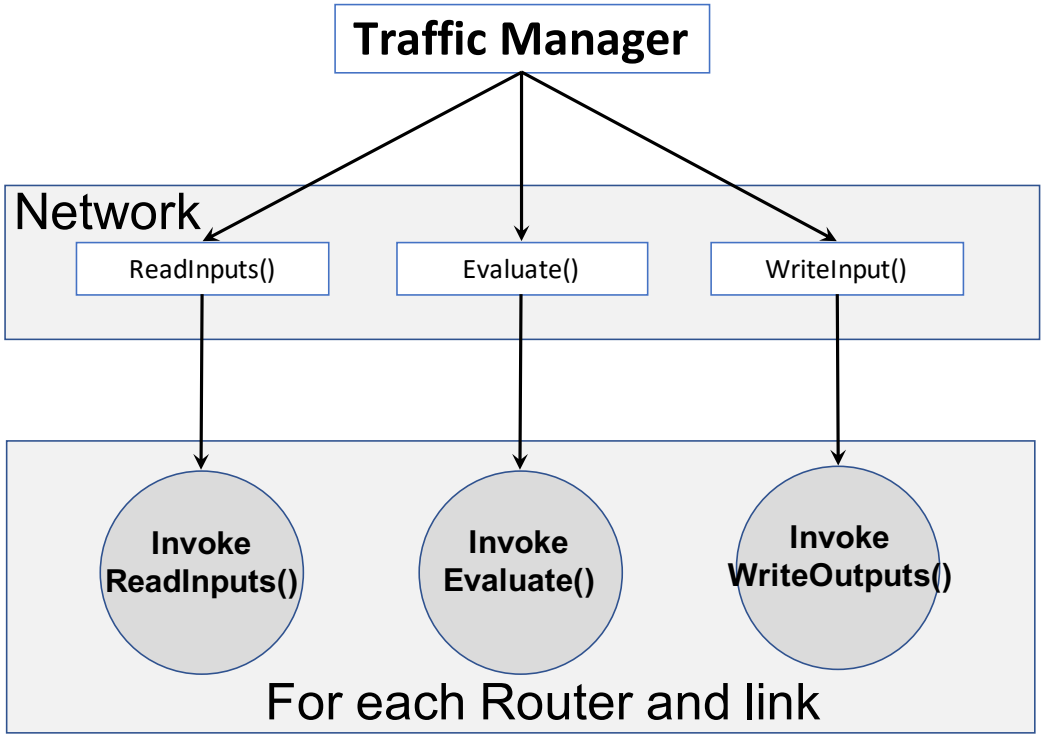


Figure 2.15: Call hierarchy for `TimedModule` objects in every clock cycle

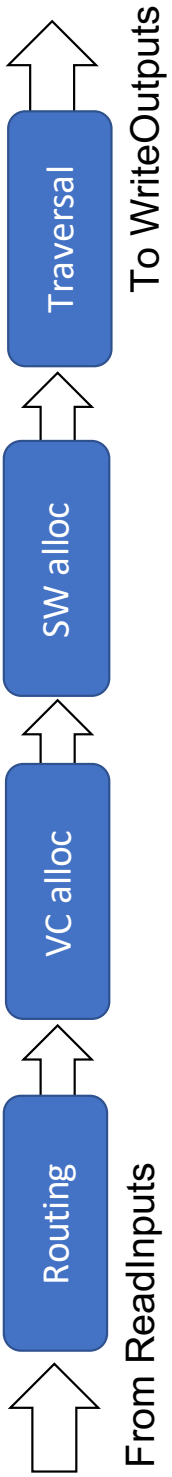


Figure 2.16: Router's pipline implemented by `Evaluate()` in Booksim

Routers and links are the basic building blocks of the simulator. Routers implement the `TimedModule` virtual functions as follows. The `ReadInput` function will check all input links attached to a Router object and read any available flit or credit. The `Evaluate` function process flits in buffers by moving them through the router's pipeline as illustrated in Figure 2.16. The last function `WriteOutput` sends any processed flit/credit to the output queue of the attached link. The link objects implements only `ReadInputs()` and `WriteOutputs()`. `ReadInputs()` will read any read flit and will deliver it to the other side of the links in the following clock cycle and `WriteOutputs()` will send flits/credit to the attached router or ejection process.

Booksim supports a wide range of synthetic traffic patterns, namely, uniform, bit complement, bit reverse, tornado, traverse, diagonal, asymmetric, taper64, neighbor, shuffle, bad dragon, and random. The traffic patterns are implemented in `traffic.cpp` file. Moreover, one can easily feed Booksim with real workload traffic models by feeding injection process of Booksim (current one uses synthatic traffic patterns) with another one. Synfull [9] is an example for modeling workload traffic models that can easily interact with Booksim to run under workload models.

2.7.2 Routerless implementation

Recall that, the routerless design replaces routers with simple interfaces interconnected by a set of loops. The interface has a basic pipeline process where each loop, passing through an interface, acts independently from all other loops in the interface. Moreover, routerless is not a credit-based design.

We code the RL interface in Booksim as a `TimedModule` and implement each loop as a stand alone ad hoc topology. The routerless interface pipeline is implemented by executing a set of functions sequentially. The functions are shown in pseudocode 2.

Routerless Pseudo-code 2: Evaluation

```
1 input queuing().
2 Ejection Module().
3 Input Module().
4 Injection Module().
5 Route Traffic().
6 Reset Status().
```

The above pseudocode 2 for every clock cycle during simulation. The *input queuing* ,pseudocode 3, goes through all the loops in an interface and read and enqueue any incoming flit (from upstream interface or injection link) in the respective flit buffer of the loop.

Routerless Pseudo-code 3: Input Queuing

```
1 for each input channel i do
2     if there is an incoming Flit then
3         Enqueue flit to buffer i
```

The *ejection module*,pseudocode 4, checks if any flit, in any buffer of the loops in the interface, is at its destination. If the number of flits found is more than the ejection links, then we choose the oldest flits among all the flits and mark them ready for ejection.

Routerless Pseudo-code 4: Ejection Module

```
1  begin
2      if Ejection link is busy ejecting a long packet then
3          Return. //use previously used loop
4      oldestFlit = NIL
5      L = −1 //loop id of the oldest flit
6      for each head flit F in the buffer of loop X do
7          if destination of F is the current node then
8              if F is older than oldestFlit OR oldestFlit == NIL then
9                  oldestFlit = F
10                 L = X
11     if oldestFlit is not NIL then
12         Flag the interface that a flit is ready to eject at loop
           L
```

The *input module*, pseudocode 5, forwards flits in buffers that are not marked for ejection.

Routerless Pseudo-code 5: Input Module

```
1 begin
2     for each head flit F in the buffer of loop L do
3         if destination of F is the not the current node then
4             if L is flagged for long packet injection then
5                 Stall AND Continue.
6             Flag the interface that loop L has a flit to forward.
```

The *injection module*, pseudocode 6, tries to inject a flit into a loop that is not marked as busy by input module. In other words, the injection module goes through all loops that include the destination node of the injected flit and pick a loop that is not marked as busy by the input module.

Routerless Pseudo-code 6: Injection Module

```
1  if Injection links is busy injecting a long packet then
2      Inject the head flit at injection links to previously used
       loop.
3  if head of injection link has a flit F then
4      if flit F is part of long packet AND EXB is not available
       then
5          return.
6      loopSet = all loop which has destination node of F.
7      for each loop L in loopSet do
8          if L is free (i.e. not flagged then
9              Flag L to inject flit F.
10             if flit F is part of long packet then
11                 Attached an EXB to the tail of L's buffor to stall
                   incoming flits.
12                 Mark the used EXB as busy.
13             Break AND Return
```

All the previous functions do not take any action but only mark respective flit/loop with a tag. The action to either send/eject/forward/inject is executed by *route traffic*, pseudocode 7. The last line in calls *reset status* function, pseudocode 8, which clears all tags and flags of loops and links (except for the case of injecting/forwarding/ejecting

long packets.) Following the order of execution above is very important to assure no conflict between the interface's modules.

Routerless Pseudo-code 7: Route Traffic Module

```
1 for each loop L do
2     if L is flagged for ejection then
3         Eject the head flit of L buffer
4     else if L is flagged for Injection then
5         Inject the head flit at injection buffer to output link
           of loop L
6     else if L is flagged for Forward then
7         Forward the head flit of L buffer to output links of loop
           L
```

Routerless Pseudo-code 8: Reset Flags Module

```
1  for each loop L do
2      if L forwarded a tail flit of a packet then
3          reset the flag of L.
4  for each Injection link I do
5      if I injected a tail flit of a packet then
6          reset the flag of I.
7  for each Ejection link E do
8      if E injected a tail flit of a packet then
9          reset the flag of E.
10 for each EXB exb do
11     if exb empty then
12         detached exb from any loop.
13         mark the exb as available.
```

The loops are implemented in Booksim by a set of links where those links connect interfaces together. Booksim implements one loop after the other. It reads loops from a file that is structured as shown in Table 2.2 for a 4×4 network (for 8×8 and 16×16, check the appendix.) Moreover, routerless has one clock cycle delay per hop. One possible solution to implement one clock cycle per hop delay is to set the link delay to zero. However, Booksim will raise an error if we set delay by zero. We work around this by updating `channel.cpp` file to send any received flit in the same clock cycle without

Table 2.2: Loops for 4×4 NoC

Loop ID	Nodes
1	[0, 1, 5, 9, 13, 12, 8, 4]
2	[3, 7, 11, 15, 14, 10, 6, 2]
3	[0, 1, 2, 6, 10, 14, 13, 12, 8, 4]
4	[3, 7, 11, 15, 14, 13, 9, 5, 1, 2]
5	[0, 4, 8, 12, 13, 14, 15, 11, 7, 3, 2, 1]
6	[0, 1, 2, 3, 7, 6, 5, 4]
7	[4, 5, 6, 7, 11, 10, 9, 8]
8	[8, 9, 10, 11, 15, 14, 13, 12]
9	[5, 6, 10, 9]
10	[5, 9, 10, 6]

any delay.

The above implementation will allow executing simulations using synthatic workloads only. For application workloads, we used Synfull [9] as explained in the evaluation section. Synfull is capable to generate workload traffic running on at most 16 cores. In order to cater for more than 16 cores, we duplicated Synfull processes and randomly mapped the cores of each Synfull process to Booksim nodes. Each Synfull process uses a unique socket id on the host system. Synfull package provides `fes2_interface` that lunch a Synfull instance and communicate with it to send/receive flits. The `fes2_interface` acts like an interface between Synfull and Booksim. In order to duplicate Synfull processes, we need Booksim to decide how many Synfull instances it needs (this is implemented in `fes2_interface.cpp` file.) On the Synfull side, each instance created must use a unique socket id or else a run time error occurred (`NetworkInterface.cpp` in Synfull packege.)

2.8 Conclusion

Current and future many-core processors demand highly efficient on-chip networks to connect hundreds or even thousands of processing cores. In this paper, we analyze on-chip wiring resources in detail, and propose a novel routerless NoC design to remove the costly routers in conventional NoCs while still achieving scalable performance. We also propose an efficient interface hardware implementation, and evaluate the proposed scheme extensively. Simulation results show that the proposed routerless NoC design offers significant advantage in latency, throughput, power and area, compared with other designs. These results demonstrate the viability and potential benefits of the routerless approach, and also call for future works that continue to improve various aspects of routerless NoCs such as performance, reliability, and power efficiency.

Chapter 3: Reliability of Routerless NoC

3.1 Faults in Networks-on-Chip

Networks-on-Chip have been historically regarded as fault-free; however, as manufacturing technologies continue to push physical limits, and feature sizes on chips shrink into the tens of nanometers, faults are increasingly expected during the operating life of a chip, and represent a significant design challenge to overcome. Due to the rarity of fault events, chip designers must carefully consider resource costs devoted to tolerating faults.

Faults may be categorized as either soft or hard faults. Soft faults are transient effects, commonly caused by events such as crosstalk, radiation, and electron tunneling. Hard faults are persistent failures, often as a result of manufacturing defects or due to aging of the chips due to heat and environmental exposure. As silicon feature size continues to decrease towards the practical limit of 5nm [19], faults will become more prevalent, as the above effects are more pronounced at smaller scales [39]. A link fault severs communication between two or more cores, preventing them from sharing cache data, rendering the chip nonfunctional if a workaround is not present. Thus, fault tolerance is necessary for many NoC applications.

While macro-scale computer networks may be designed to overcome faulty links through techniques such as retransmission, rerouting, and physical repair of faulty network hardware, the availability of such solutions for NoC is limited due to the tight

constraints on size, speed, and power usage. While soft faults are more prevalent than hard faults, they are fairly easy to overcome through retransmission which is enforced by timeouts and cache coherence protocol. Hard faults, on the other hand, cannot be resolved without bypassing the fault through leveraging of existing redundancy in the network. Redundancy can be integrated into chip design with a combination of features, such as duplicated links, modified routing algorithms that re-route around faults, and forward error correcting (FEC). Although FEC can be implemented on the circuit level, independent of overall network design, the most commonly used FEC techniques, such as SEC/DED, are not equipped to overcome the link faults. Thus, redundancy in the network design itself is required for effective fault tolerance.

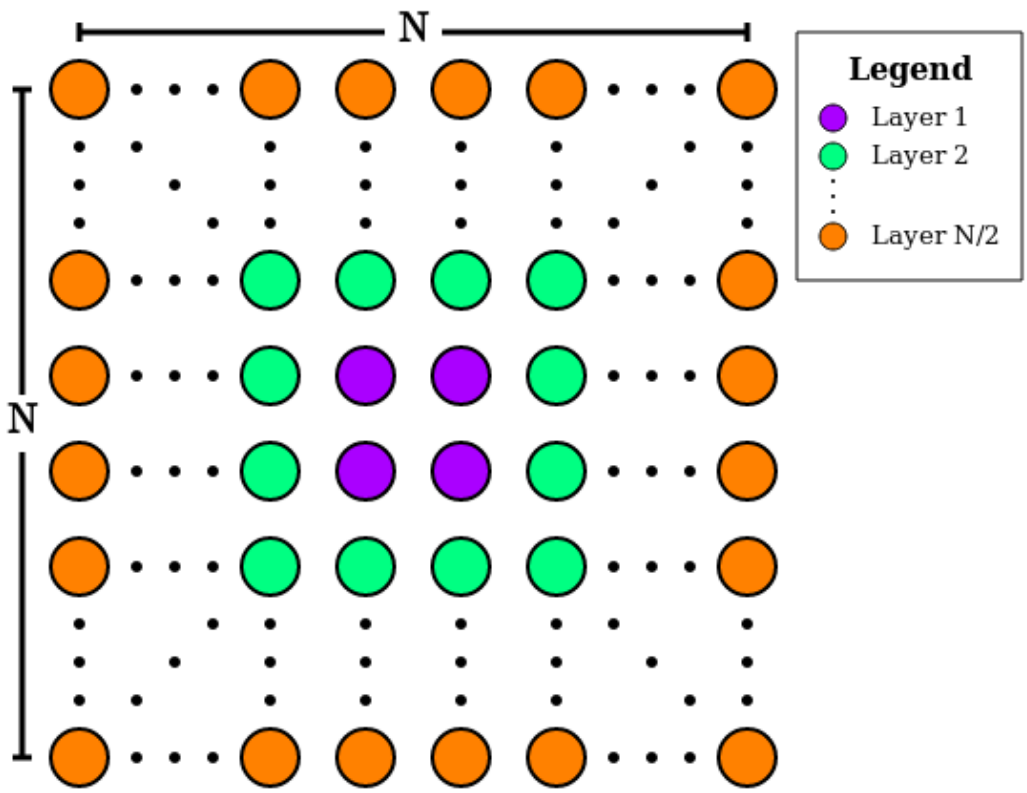


Figure 3.1: RL Algorithm grows the network in concentric layers

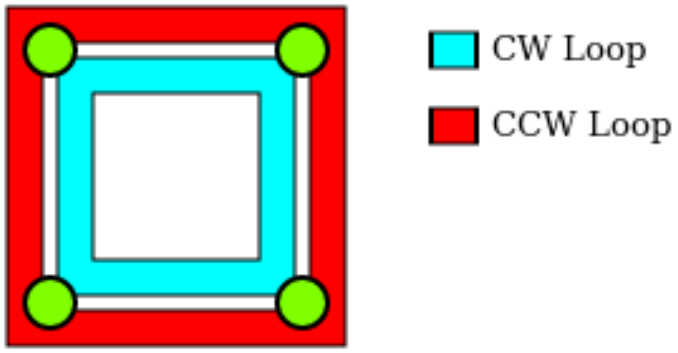


Figure 3.2: Layer 1 (2x2) contains two counter-rotating loops

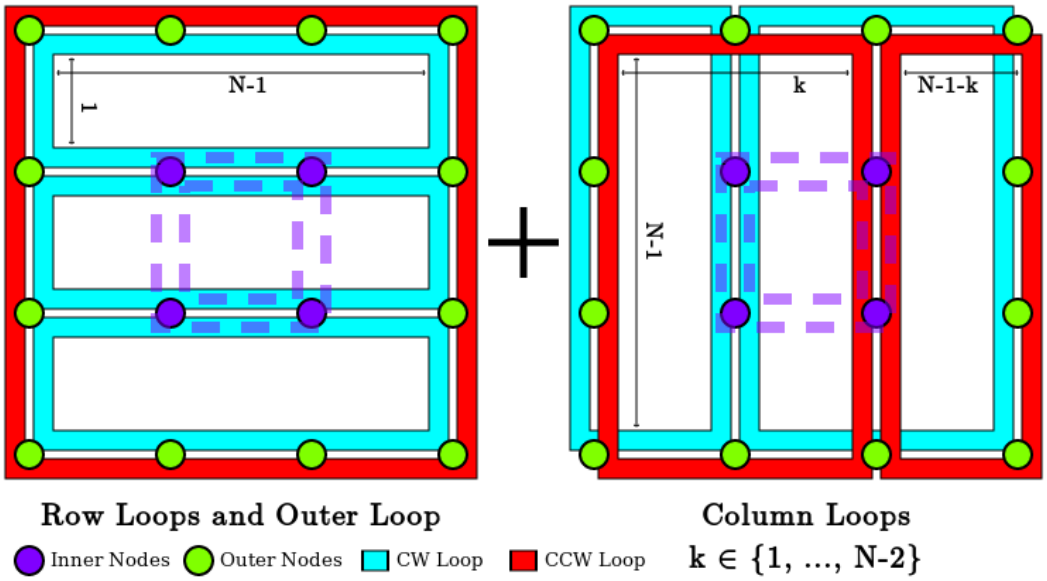


Figure 3.3: Layer 2 (4x4) loops provide connectivity to new outer nodes

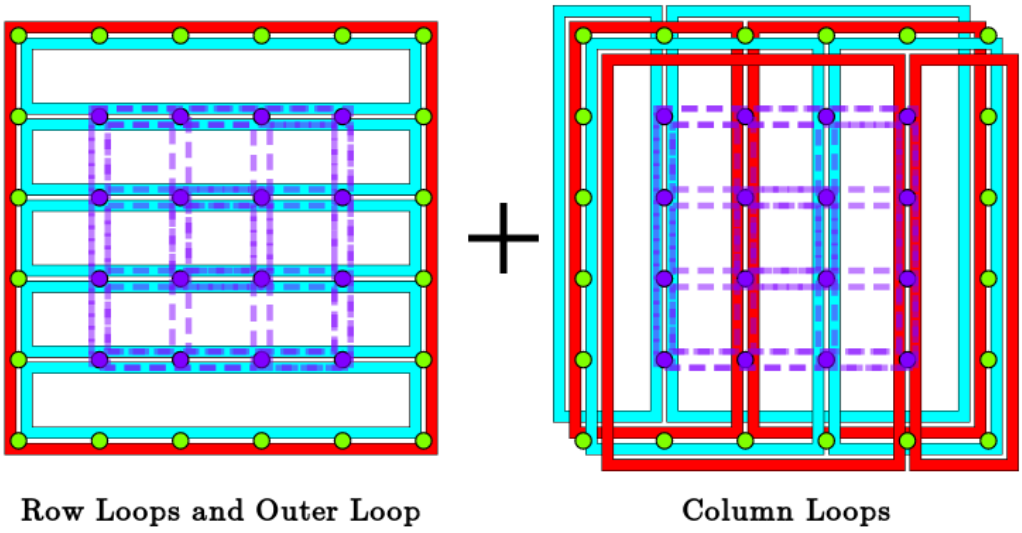


Figure 3.4: Layer 3 (6x6) loops showing alternating direction of column loops

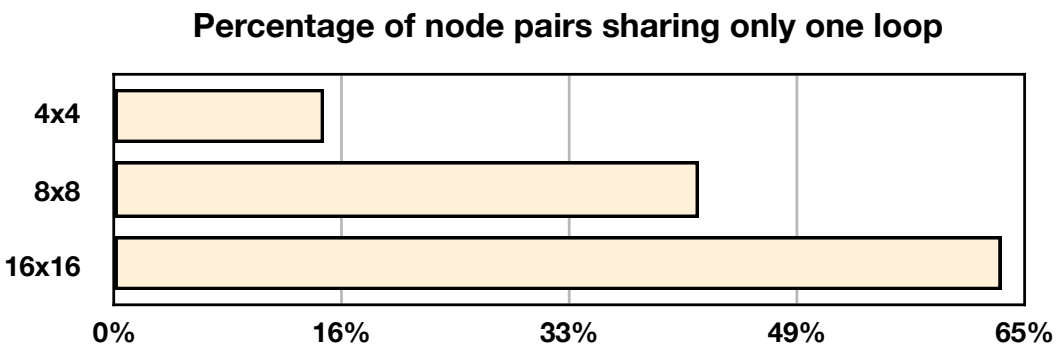


Figure 3.5: Percentage of node pairs that share a single loop

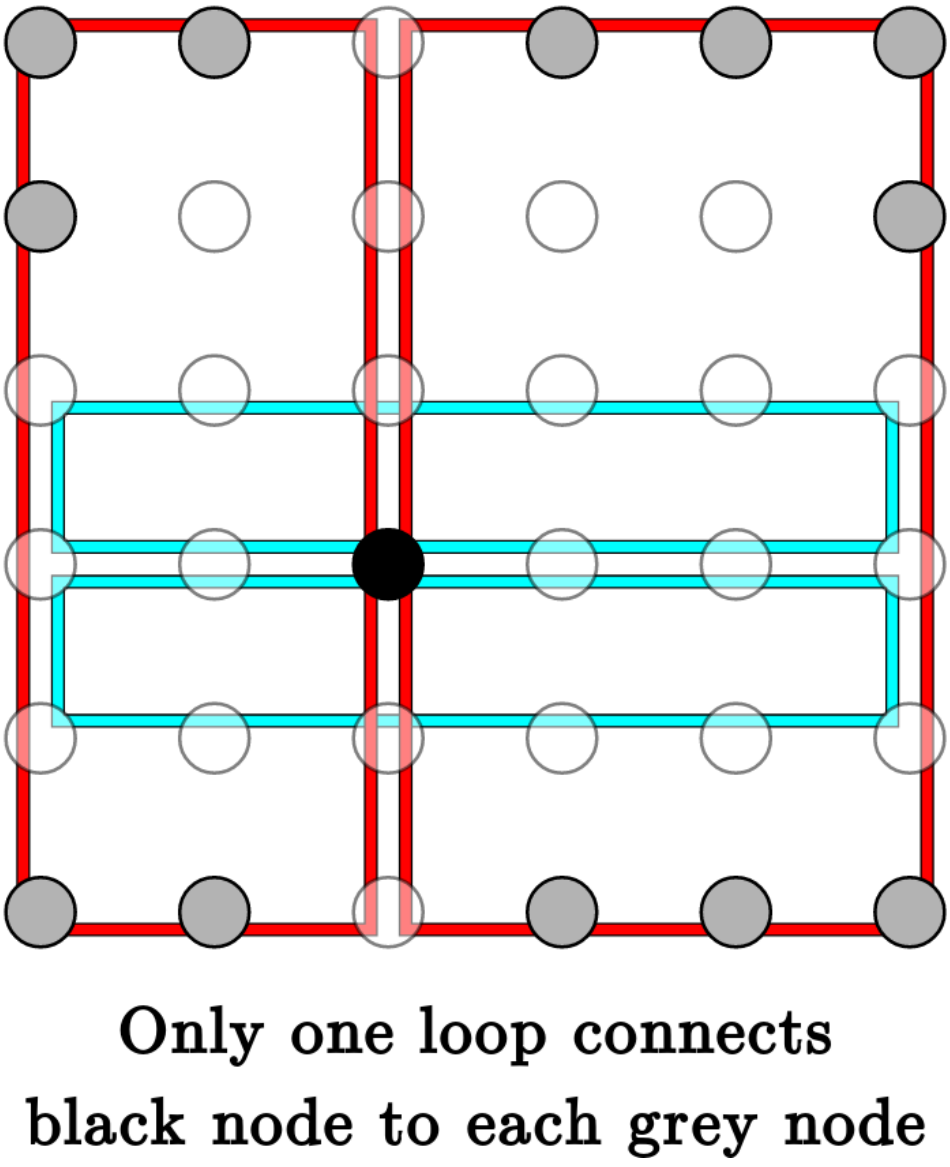


Figure 3.6: Only one loop shared between gray nodes and the black node

3.1.1 Motivation

Although RL is a promising approach to NoC design, it is severely limited compared to router-based networks in its ability to adapt to changes in network state. Although many router-based networks are implemented using simple routing algorithms – such as xy-ordered routing, where a flit travels along the x-axis until reaching the same x-coordinate as its destination, and then travels along the y-axis until reaching its destination – expanding the decision-making capabilities of routers to handle network congestion and other abnormalities is relatively straightforward, as much of the required hardware (crossbar, buffers, etc.) is already present. Router-based networks also have the advantage of information availability, in that each router along a transmission path has the capability of sensing its neighborhood and altering the path of packets traveling through it in a distributed manner.

In contrast, RL's advantages over router-based networks come from making single-point routing decisions based on hardwired knowledge of the entire network topology. Adding the ability of the sending node to sense the state of candidate loops and select the optimal path to inject a packet on is untenable as it would require each node to have complete state information about the network, which would complicate the design and add network congestion. The elapsed time between a state information broadcast and the expected arrival of a packet to the broadcaster may also be quite long, reducing the usefulness of such an approach. Furthermore, while a router-based network has a high degree of available path diversity to support more complex routing algorithms if desired, an RL network may provide only one or two paths, as shown in figures 3.5 and 3.10,

for an injecting node to select from in order to reach the destination node, limiting the usefulness of adaptable routing algorithms at the injection stage without modifications to the RL design to increase path diversity. An alternative approach that avoids these obstacles is to add resources that enable nodes to resolve adjacent faults with redundant links, and/or limited re-routing capabilities.

3.2 Addressing reliability in RL NoC

While this is the first paper addressing fault tolerance in the recently-developed RL design, prior research has addressed several techniques in mesh networks, generally leveraging path diversity and existing router hardware to bypass faulty links through mis-routing. Mesh networks are well suited to this task by design as the fault and routing hardware that handles the bypass operation are already co-located. In contrast, RL requires significant changes to support such techniques due to the inability to reroute packets mid-flight. Routing decisions in an RL network are made by the packet injector, which may be separated by some distance from the fault. Thus, a solution requires network-wide fault broadcasts and updating of routing tables, or fault-adjacent reliability hardware.

One straightforward approach to tackling the RL reliability is to add extra passive links to act as redundant connections between nodes, as shown in Figure 3.7. In the event of link failure, the nodes on each end of the broken link would simply switch to using an available passive link as needed. This strategy is also flexible, in that extra passive links can always be added to further increase the fault tolerance of the network. However, passive links have a major downside in that they occupy physical space that could be used

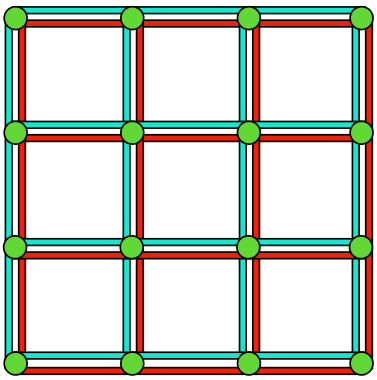


Figure 3.7: Passive links offer redundancy in the event of link failure

for active wiring resources, such as extra loops, and they require switching architecture on each node to bypass any link that fails.

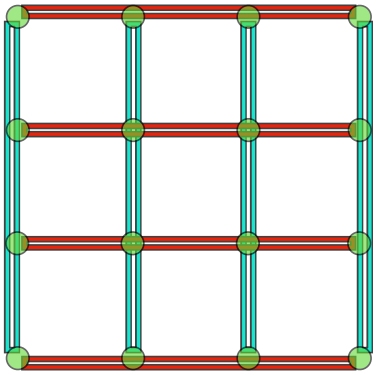


Figure 3.8: Supplemental short loops deactivate and provide donor links to bypass failed links in RL loops

Another strategy is similar to adding passive links, except, in this design, the additional links are connected to create several auxillary short loops (an example of a possible short loop configuration is shown in figure 3.8). These short loops provide increased throughput while active; when a link on a main RL loop fails, the short loop on that row

or column donates its own link to the main loop, and becomes inactive. Although this provides some benefits over passive links, it still requires more than $4N^2$ buffers and links to implement. All traffic in a faulty loop and short loop may be dropped in the event of a link failure, as well, and the short loop links from a donor loop becomes inactive, reducing the utilization of available wiring resources.

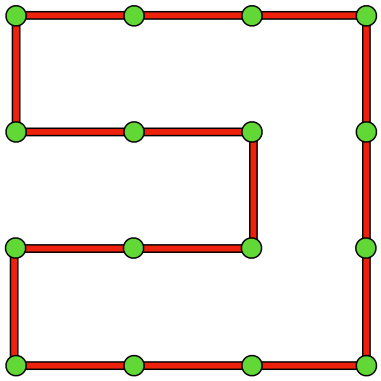


Figure 3.9: Proposed auxiliary's Hamiltonian loop to provide redundancy in the event of loop failure

RL can also be made more reliable by the addition of a final loop that follows a Hamiltonian cycle through the network, depicted as in Figure 3.9, providing redundant connectivity in the event that any two nodes are disconnected due to a single loop failure. A Hamiltonian cycle has an advantage over short loops in that it uses fewer wiring resources for the additional loop, and also requires fewer buffers. However, this strategy is still quite costly as it requires the addition of N^2 buffers and links. The Hamiltonian cycle also dramatically increases latency between nodes that may have been closely connected in the failed loop, as nodes in a Hamiltonian cycle may be directly connected, at most, to two of four possible neighbors. Also, as communication between nodes often

entails a request message and a reply message, the total length of the loop, being N^2 hops, incurs a tremendous performance penalty in the event that it is used to bypass a faulty link. This strategy also entails disabling a faulty loop once detected, which might cause all traffic currently on that loop to be dropped. Furthermore, in the event of multiple link failures across the network, the Hamiltonian path would quickly saturate trying to handle the simultaneous traffic load of multiple deactivated loops. Finally, the otherwise functional links of a deactivated loop would remain unused in the event of a single link failure on that loop, which is very wasteful of wiring resources.

Although the above solutions could improve RL reliability, we did not find them to be workable techniques due to their costs and unacceptable performance degradation. In this paper, we instead present a method of achieving reliability in RL network without the addition of passive links, active loops, or buffers, with minimal hardware overhead to achieve the switching capabilities necessary to implement a bypass scheme. In contrast to the above solutions, we will provide the necessary fault tolerance by leveraging existing structures within the RL network design to overcome link failure, and provide seamless redundancy.

3.2.1 A fault example in RL

In RL networks, the probability of link failure is expected to be higher than in a router-based network due to the abundance of links. Although one might assume that the excess connectivity of RL would provide fault tolerance due to path diversity, we will show that this is not the case; in fact, a single fault can completely sever communication between

pairs of nodes, as shown in the following example. Consider the section of a 6×6 network shown in Figure 3.10. After the entire network has been generated using the RL algorithm, the blue loop in Figure 3.10 remains the only loop that connects the black colored node in position $(3,4)$ to the gray colored nodes in positions $(1,1)$, $(1,2)$, $(1,3)$, $(5,1)$, $(6,1)$, $(6,2)$, and $(6,3)$.

As no other loop will connect these pairs in the final network design, a single fault in one of the 16 links comprising this critical loop will prevent the black node either from sending or receiving messages to or from each of the gray nodes, preventing normal communication for 7 pairs of nodes. In RL, each node considers only the state of its network interface to determine which action (inject, eject, forward, or stall) to perform. This means that, even if link fault detection were available, only the two nodes connected by the faulty link would be aware of the problem. The remaining nodes would be unable to respond in any way to the fault, instead treating the broken loop as if it were fully operational. The consequence of this is that flits will be constantly dropped or corrupted when traversing the faulty link, causing abnormal operation of the whole chip due to a single-point failure. In a 6×6 NoC, RL produces a total of 280 links, while a mesh network of the same size, with one link between each node requires only 60 links. Thus, it is reasonable to assume that the probability of single-point failure is higher in RL, and a robust fault-tolerance mechanism is necessary.

In this section, we present a design for handling link failure locally, without requiring broadcast of link state to affected nodes, or additional links beyond the original RL design. We fuse the faulty loop with a non-faulty loop, to bypass the fault and restore connectivity. This fused loop visits all of the nodes previously visited by its two constituent loops,

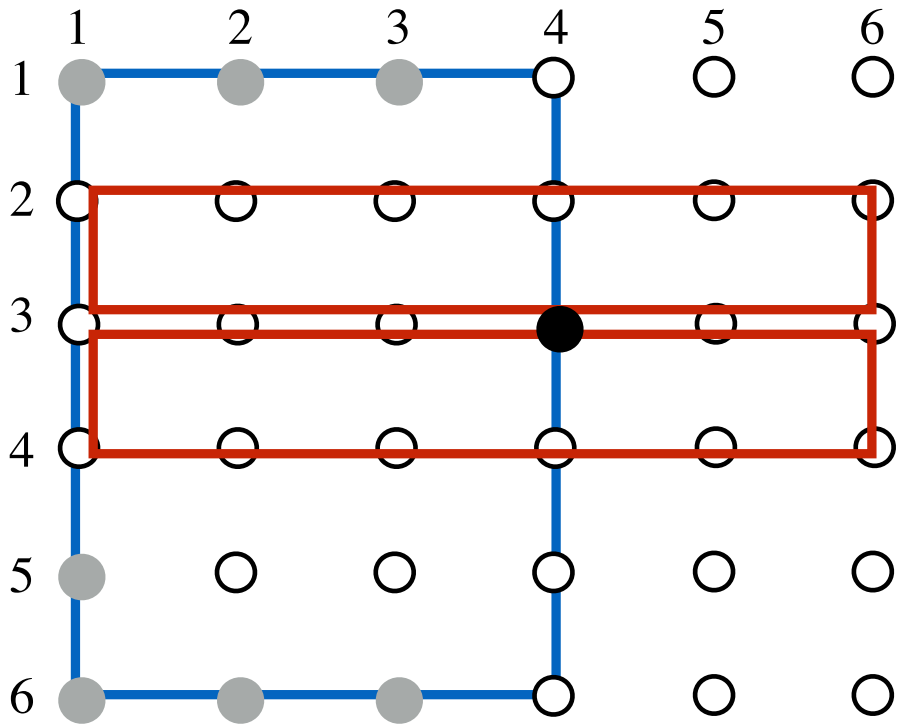


Figure 3.10: Only one loop shared between gray nodes and the black node

ensuring that nodes can still communicate as if both loops were functioning normally, with a tolerable increase in average latency for nodes utilizing the fused loop. If the faulty link provided a connection from node x to node y then the only requirement for the donor loop is that it provides a connection in the reverse direction, from node y to node x; when this condition is met, the two loops may fuse, isolating the fault from the rest of the network. This process is carried out by the following steps, depicted in figure 3.11.

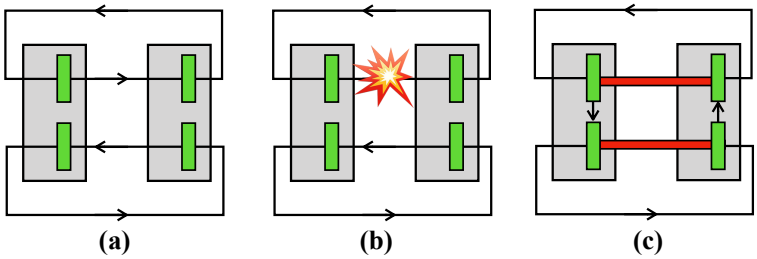


Figure 3.11: Two loops are joined into one after detecting a faulty link

1. Detection of the faulty link f, connecting $x \rightarrow y$.
2. Selection of donor loop with link f', connecting $y \rightarrow x$.
3. At x switch the output of the f_x buffer from f, to the input of the f'_x buffer.
4. At y switch the output of the f'_y buffer from f', to the input

Following the completion of the above steps, the loop previously containing the faulty link, f, and the donor loop previously containing f' are joined into a single larger loop, as shown in Figure 3.11. Notice that, in step (c) of the figure, the new loop visits two buffers when passing through x or y, rather than one, as it has combined the resources of both component loops together.

Thus far, we have demonstrated a technique for fusing a faulty loop and donor loop to bypass a faulty link. However, we have not yet discussed the second step of the fusion process described above, where the donor loop is selected. For this technique to work across the entire RL network, we must guarantee that a donor loop can be selected for every fault that might occur.

Here, we demonstrate a technique to select a donor loop for every potential fault on a generic RL layer, which can be applied to each layer independently, thus providing a donor loop for every link in the network. If we look at only the row loops and the outer loop for a moment, as shown on the left portion of Figures 3.3 and 3.4, we can see that each link only ever overlaps with one other loop, and that the direction of travel of the overlapping loop is always opposite. Thus, for each link on any of these loops, the donor loop is simply whichever of these loops overlaps that link. Next, we turn our attention to the set of column loops on the layer in question. On the right half of Figure 3.3, we can see that there are a total of four column loops. For each of the links shown, there is, as before, only one overlapping loop, which also happens to travel in the opposite direction of the link in question. As before, for each link, the donor loop is whichever loop overlaps that link. For layers with more columns, we simply group the columns into pairs (1 with 2, 3 with 4, 5 with 6, etc.) and apply the same process for each pair of columns.

3.2.2 Fault detection and switches

Fault detection in integrated circuits is very challenging and heavily depends on the underlying physical properties of the fabrication technology and design decisions such as manufacturing process, space between wires, clock rate, and when to perform testing. For detecting faulty links in RL, we consider a popular technique, Built-In-Self-Test (BIST), which allows a circuit to test itself without interruptions. BIST consist of the *test data generator* (TDG) and the *test error detector* (TED). TDG generates different patterns on the transmission end of a link and TED checks on the receiving end of the link if the unaltered pattern is received. If an alteration is detected, then the link is no longer considered reliable, and the fault tolerance procedure is triggered.

To fuse the loops together, we use a simple multiplexer at the input of each buffer, allowing it to take input either from its link, or from the output of the proper buffer, once loop fusion is triggered.

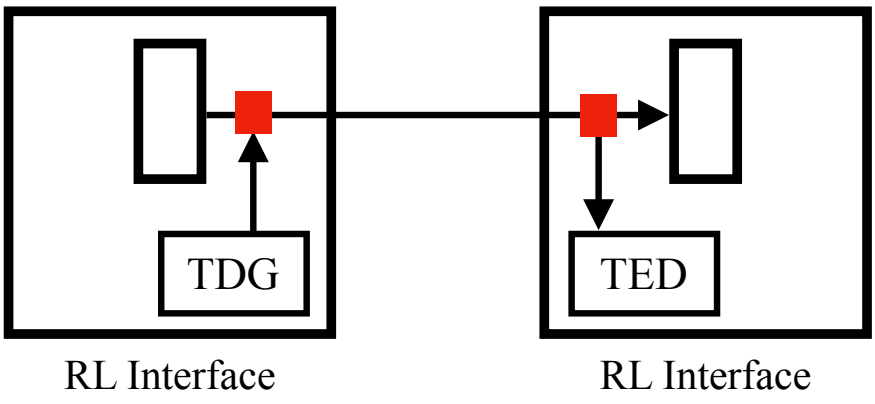


Figure 3.12: High level BIST design to detect faults in links.

Table 3.1: Key configuration parameters of simulation

Coherence Protocol	MOESI
Virtual networks	3
Memory size	1GB
l1i size	32kB
l2 cache	64
l1d associate	4
l1d size	32kB
l2 size	128kB
l2 associate	8
Memory Controllers	4
Link width	128bit
Control packet	1 flits
Data packet	5 flit
Extension Buffer	5 flits

3.3 Evaluation and results

3.3.1 Evaluation methodology

To evaluate the effectiveness of our proposed fault tolerance strategy, we simulated an 8×8 RL NoC with the reliability features implemented using Garnet2.0 [5] in the GEM5 [12] platform with the configuration parameters shown in Table 3.1. RL loops were generated following the RL algorithm in [7], except that the algorithm was modified to alternate the orientation of successive pairs of column loops between clockwise and counter-clockwise, to support the proposed fault tolerance scheme.

Synthetic traffic patterns and PARSEC [15] workloads were simulated to capture performance statistics. The synthetic traffic statistics collected after completion of 100,00 cycles at various injection rates starting from 0.005 flit/node/cycle and incremented by

0.005 flit/node/cycle until the maximum throughput was reached. Moreover, performance statistics for seven PARSEC workloads were collected using GEM5 full system simulation. Traffic traces were also collected to evaluate and compare hop counts to Mesh topology.

Synthetic benchmarks were run repeatedly for one-at-a-time failure of each of the 672 links in the 8×8 RL network. For the full system simulations running PARSEC benchmarks, only one run was performed with a randomly selected link failure, due to computational constraints. For each of the synthetic and full system simulation workloads, one control run without any faults was also performed.

Power and area were estimated after place and route, following a similar implementation to [7] based on Verilog post-synthesis simulation. In the power calculations, we assumed a high 0.5 link utilization ratio for switching activity input and 1 GHz clock frequency.

3.4 Results and Analysis

3.4.1 Synthetic workloads

Figure 3.13 presents the average latency results for 8 synthetic traffic patterns at injection rate 0.005, for fault-free operation, and the overall average latency across all 672 link-fault runs. On average, latency increased by 5.2% due to a fault, with the minimum average latency increase of 4.01% with the transpose traffic pattern, and maximum average latency increase of 6.92% with the tornado traffic pattern.

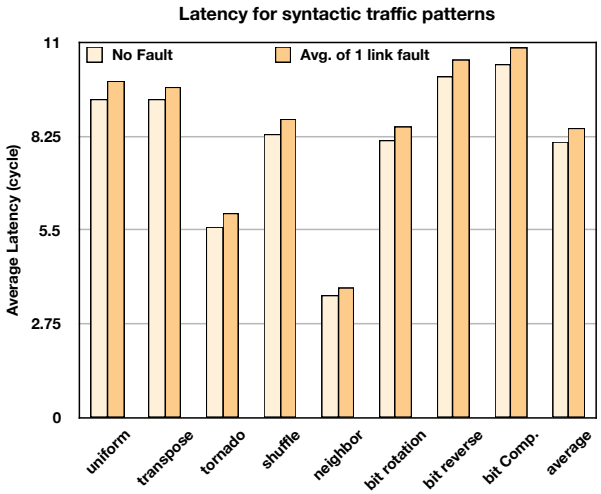


Figure 3.13: Average latency of traffic patterns at injection rate 0.005.

When a fault occurs in RL, only two loops fuse together to form a longer loop; as a result, all the other loops remain in normal operation, which limits the impact that the new fused loop can have on the network latency, as it comprises a small fraction of the overall network. For example, an 8×8 network has 44 loops; after a fault, 42 loops remain intact, and a 43rd loop is created from the faulty and donor loops. Because of this, the effect on hop count is low. Moreover, Figure 3.14 investigates the effect of link faults on hop count (based on uniform random traffic pattern) in RL NoCs. The effect of a fault on the average hop count is low, and the relative percentage change decreases as the network size increases, because the percentage of nodes and loops affected by a fault decreases as the network size grows. For example, the hop count of the 4×4 network with a fault is increased by 8% in the average case and 16% in the worst case. However, for the 16×16 network the hop counts increase by 1% and 4% for the average and worst cases, respectively.

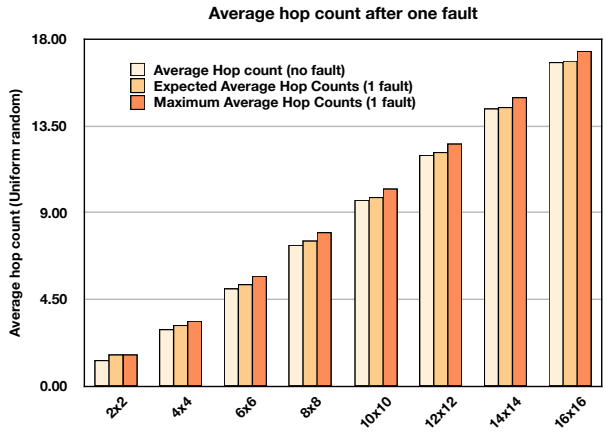


Figure 3.14: Average hop count prior to fault vs. after a fault. Average hop count depends on fault location expected and worst case average hop count are shown.

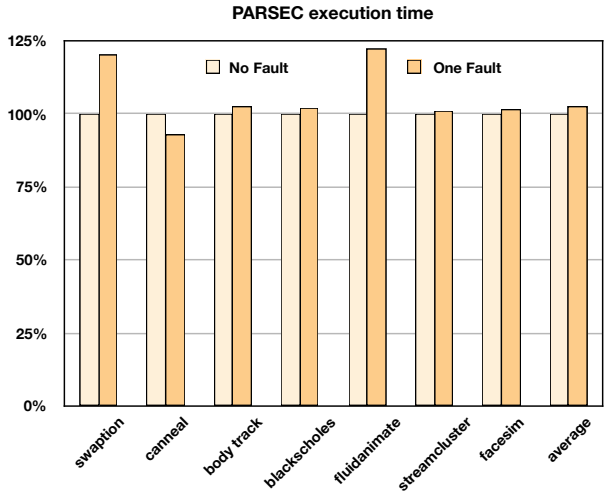


Figure 3.15: Percentage of execution time change with and without a fault

3.4.2 PARSEC Workloads

Figure 3.17 presents the average latency for the 7 PARSEC workloads with no fault and with a randomly selected fault. Similar to the synthetic results, the latency increase after a fault is still very low. The benchmark with the largest increases in latency is blackscholes,

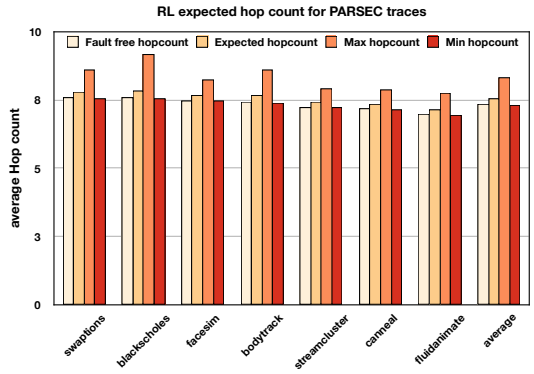


Figure 3.16: Expected average hop count for 8×8 RL

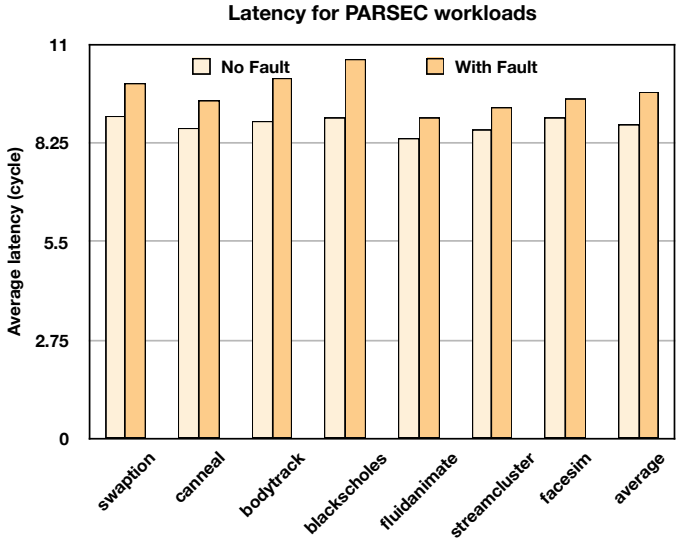


Figure 3.17: Average latency of PARSEC workloads with and without a fault

with an increase of 18.4%, while the lowest increase in latency was found with facesim, at 5.8%. Overall, the average increase in latency across all the PARSEC benchmarks was 10%.

Moreover, Figure 3.15 compares the total execution time for each workload to the baseline, following a random fault. Among all the workloads, fluidanimate execution

time increased by 22.15% while the execution time for canneal reduced by 7.17% after the fault, meaning that the modified loop structure following the fault offered better performance for canneal. The average increase in execution time is 2.63%

3.4.3 PARSEC traces

We extensively study the hop count metric in a fault free and single fault instances in RL and optimal Mesh. Due to long execution time of a full system simulation in GEM5, it would be hard to run the simulator to capture hop count for each link failure. Therefore, PARSEC traces are used and feed into another program to calculate only hop count for the traces.

Using PARSEC traces, we computed average hop count 672 times for every link fault in the 8×8 RL. Figure 3.16 depicted the expected average hop count for all workloads beside maximum and minimum average hop count after a single fault. The average hop count for fault free case is 7.35 and the expected average in a single fault scenario increases to 7.56 hops which is 2.8% where as the maximum is 3.5% (blackscholes) and the minimum is 2.29% (fluidanimate).

3.4.4 Power and area analysis

The unmodified RL interface offers significant reductions in power and area usage compared to router-based NoCs – 86% less area and 91% less power than a typical mesh router [7]. The additional logic required to implement the reliability scheme comprises

of switches to fuse the loops, and BIST units to detect faults. This logic was added to the base routerless design, and was placed and routed using Cadence Encounter. The number of loops visiting a node is non-uniform across the network, increasing towards the central nodes. To calculate

To calculate the number of loops overlapping a node in RL is non-uniform – the overlapping increases towards the center of the network. For an 8×8 RL network, the average overlap is 10.5 loops overlapping a node, so we calculated the area and power usage for a 10-loop overlapping node. The 10-loop node requires $5853\mu m^2$ of area for the interface, including the reliability components.

RL interface design offers a significant reduction in power and area usage compared to router-based NoCs [7] (86% less than typical mesh Router.) We implement additional switches and BIST in Cadence Encounter after P&R. An 8 RL interface require $5853\mu m^2$ area assuming the number of loops in this interface is 10. The additional basic components to implement the reliability features require less than 4% of the total RL interface area and this brings the total to $6083.125\mu m^2$.

All switching activities for power calculations were gathered during running PARSEC workloads. The additional BIST and switches components elevated static power usage by 13% and dynamic power by 38%. In total, an extra 20.3% increase in total power. The results for each workload is reported in Figure 3.18. The design of RL is very basic compared to on-chip routers. In [7], RL is reported a power reduction of 9.48X over Mesh. Therefore, any additional component added to the design will notably elevate the power requirement. This is why the dynamic power of RL is increased by 38% after the addition of BIST and link switches.

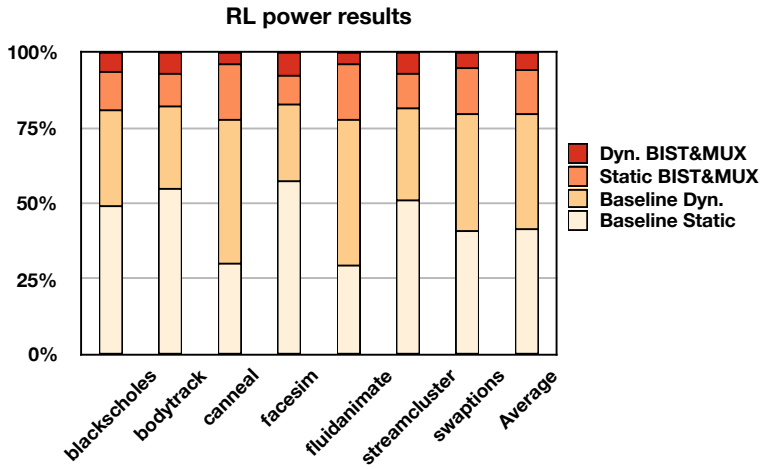


Figure 3.18: Breakdown of power consumption for PARSEC workloads

3.5 Discussion

3.5.1 Multiple faults in NoCs

We presented above an approach to tolerate a single fault link in RL NoC using the original loop set. However, if two links go faulty the NoC may get disconnected if and only if two loops were fused twice. For example, assume links x, y are on loop A and the donor loop for both x, y is the same loop B. After a link fails, say x, the NoC would follow the steps to fuse loops **A** and **B** into one big loop **AB**. Now the donor loop for link y is the new loop **AB** and what happens if two opposite sides of a donut are squeezed? It gets disconnected into two smaller donuts. See Figure 3.19 for illustrations. But how likely two or more faults happen. For two faults case in an 8×8 NoC, we wrote a small program to count the number of link pairs that will fuse two loops twice. Among all possible 225456 pairs, only 2216 pairs are considered bad and will disconnect the NoC if any

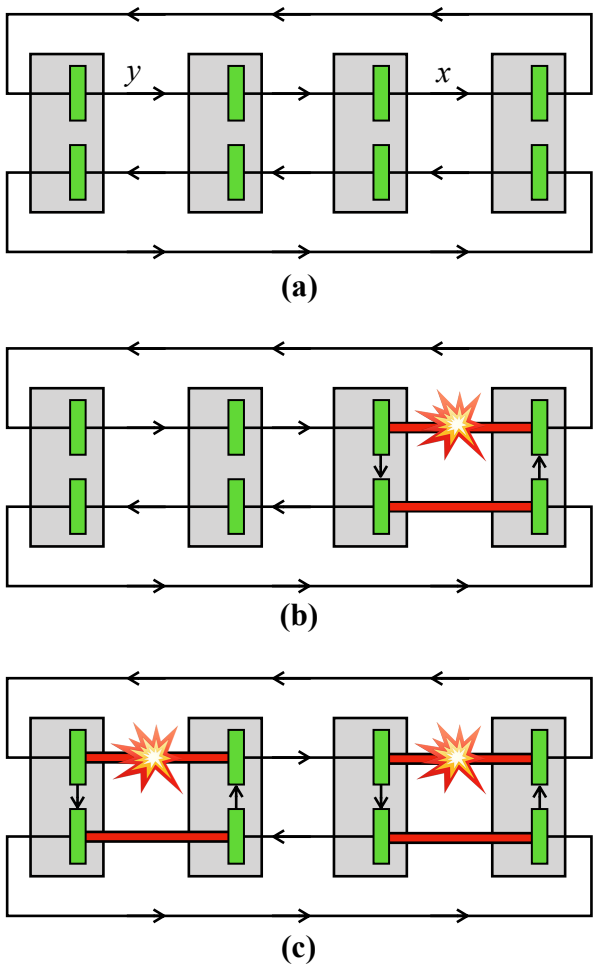


Figure 3.19: Fusing two loops more than once will result into disconnected loops.

of those pairs are used for fault toleration. The percentage of bad pairs is only %0.982. For three faults case, the total number of triple links is 50351840 and only 672412 are triples may disconnect the network if two or three links failed. The percentage of the bad triples is %1.335. In the extreme case, a loop set of **N** cordiality can theoretically tolerate at most $\mathbf{N}-\mathbf{1}$ if and only if each link failure fuse two disjoint loops. In Figure 3.20, we demonstrate an example of how a 4×4 RL NoC can achieve the maximum number of

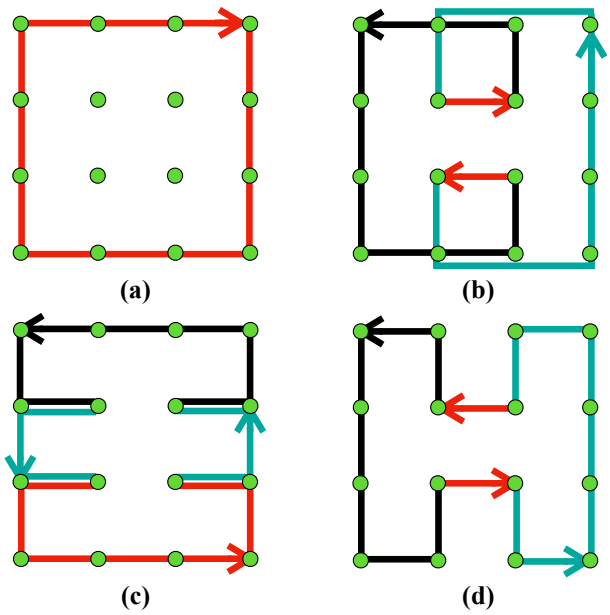


Figure 3.20: Example of ten faults in 4x4 NoC

fault which is ten. The sub-figures (a), (b), and (c) of Figure **??** are results of fusing three loops after two link failures in each sub-figure. Then, (d) can clearly fuse with (a),(b),(c) because those loops overlap on the outer layer and (d) is the only clockwise direction loop.

In router-based NoCs tolerating faults is more complex compared to RL NoC. Upon detecting a link is malfunctioning, the NoC behavior and function may change drastically especially the routing protocol. Unlike RL, updating or changing the routing protocol is straightforward because the NoC may no longer look regular (e.g. $\mathbf{N} \times \mathbf{N}$ Mesh). The adaptation on the routing protocol must assure the NoC is deadlock-free.

3.6 Highlight on implementing RL in Gem5

The Gem5 is a big piece of software that is structured into multiple subsystems. GARNET, or the NoC simulator, is part of the Ruby subsystem. Gem5 has one global counter for the system clock cycle and employ a global event queue that holds pointers to all components that needs to be waked up in the future (upcoming clock cycles.) Unlike Booksim, where it goes through all components and execute the code regardless if the component is idle or has nothing to process, Gem5 saving computation cycles of the host system by running components that are not idle.

GARNET mainly contain two kinds of components, links and routers. Each router/link is a `Consumer` object that can be enqueued in global event queue and waked up in by simulation process. GARNET, by default, implement a standard Mesh routers which is very similar to the pipeline implementation of Booksim. Therefore, we follow a similr implementation of RL as given by Section 2.7.2. Moreover, we joined routers along with links in order to implement the one clock cycle per hop. Therefore, the flow of execution for each RL interface is to read inputs from links, process the flits, and then send them. We include our implementation code in the appendix in addition to topology implementation.

3.7 Conclusion

In this chapter, we presented a solution to let the routerless tolerate one fault. The technique does not require any additional links and no changes to the functionality of any of the interfaces. Instead, we fuse two loops into one in case a fault occur to make a

bigger loop to avoid passing through the faulty link. This solution requires 4% and 20.3 % of area and power budges, respectively. Latency, after a fault occur, is increased by 5.2% and 5.8% for synthetic patterns and PARSEC workloads, respectively.

Chapter 4: Conclusion and future work

In this thesis, we presented a solution for the network-on-chip (NoC) part of upcoming era of many-core processors. These many-core processors usually employ lightweight processing elements running on reduced instruction sets to stay within area and power constraints. A major performance bottleneck of these many-core processors is the on-chip interconnect. Conventional router-based designs highly depend on the on-chip router component. Despite the flexibility that routers brings to the design, they are area and power heavy components. When the number of routers increase on-chip, they will occupy a large amount of the die's area and consume a great amount of power and, hence, devote most of the chip resources to the NoC instead of the processing elements. The routerless (RL) design is a novel NoC design that do not employ any routers and brings a large saving in area and power budgets without any loss in performance when compared to the conventional NoC designs. A key aspect in this design, when compared to the router-based NoC, is the amount of wires utilized to create the loops that interconnect all processing elements. The number of wires has been underutilized in all router-based NoC mainly due to crossbar in routers. Each loop in RL connects a subset of nodes and acts independently among all other loops. The loop isolation incredibility reduces the design complexity and offered a notable saving in area and power budgets with no performance trade-off. In addition, RL design is inherently deadlock free and all possible network abnormalities are resolved.

Shrinking features sizes due to technology innovation increased failure rates of on-chip components. The routerless design uses a large number of wires which increases their failure rate. We presented an approach to tolerate permanent link failures that require no additional links between RL components. The solution simply is to fuse two loops into one big loop to detour around the faulty link. This fault tolerance approach requires no changes on the functionally of RL design after a link goes faulty.

4.1 Future work

The RL has a huge design space where we only scratch its surface. In the following subsections, we present several possible future directions.

4.1.1 Routerless tiles

Adjacent nodes in large RL networks share a lot number of loops. However, deflection of flits may notable increase latency and hence effect performance. A possible solution might be to use small $n \times n$ RL network and place them as tiles on the die. We can either use loops or router to connect those tiles.

4.1.2 Optimal width of a link

RL will perform better if each packet is of a single flit. In such scenario, there will be no need for extension buffers and the possibilities for stalling a packet is reduced. In evaluation part of this thesis, we used a width of 128bit for each link. So a question worth

asking here is, what is the lowest width of a link that can be employed in order to keep the performance in a good range.

4.1.3 Use AI to find better set of loops

The total number of loops grows exponential with the network size. The presented design follows a specific pattern that is applicable for any $n \times n$ network. Using machine learning or deep learning is a great tool to explore design spaces and find hidden pattern in the loops with a lower hop count.

4.1.4 Application mapping for RL networks

Unlike router-based network, RL employs loops where each loop act independently. Applications perform better when they are connected in their own network. So, is it possible to map an application on a loop for the purpose of enhancing its performance?

4.1.5 Multicast and broadcast

Something we have not discussed in this thesis is how to handle broadcast and multicast packets. Loops in RL are all isolated and act independently. The naive way to address broadcast/multicast packets is to send each broadcast/multicast packet on all loops. So, can we use a limited number of loops in order broadcast/multicast a packet.

Bibliography

[1] http://ark.intel.com/products/95830/intel-xeon-phi-processor-7290-16gb-1_50-ghz-72-core/.

[2] http://gem5.org.

[3] https://oeis.org/A140517.

[4] http://wccftech.com/intel-sc15-knights-landing-14nm-wafer-specification/.

[5] Niket Agarwal, Tushar Krishna, Li-Shiuan Peh, and Niraj Kumar Jha. Garnet: A detailed on-chip network model inside a full-system simulator. *2009 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 33–42, 2009.

[6] Thomas William Ainsworth and Timothy Mark Pinkston. On characterizing performance of the cell broadband engine element interconnect bus. In *International Symposium on Networks-on-Chip (NOCS)*, 2007.

[7] F. Alazemi, A. AziziMazreah, B. Bose, and L. Chen. Routerless network-on-chip. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 492–503, Feb 2018.

[8] R. Arunachalam, E. Acar, and S. R. Nassif. Optimal shielding/spacing metrics for low power design. In *IEEE Annual Symposium on VLSI*, 2003.

[9] Mario Badr and Natalie Enright Jerger. Synfull: synthetic traffic models capturing cache coherent behaviour. In *ISCA*, 2014.

[10] L. A. Barroso and M. Dubois. The performance of cache-coherent ring-based multiprocessors. In *ISCA*, 1993.

[11] Luiz Andre Barroso and Michel Dubois. Cache coherence on a slotted ring. In *ICPP*, 1991.

[12] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011.

[13] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, Bin Liu, A. Tran, E. Adeagbo, and B. Baas. A 5.8 pj/op 115 billion ops/sec, to 1.78 trillion ops/sec 32nm 1000-processor array. In *Symposium on VLSI Circuits*, 2016.

[14] L. Chen and T. M. Pinkston. Nord: Node-router decoupling for effective power-gating of on-chip routers. In *MICRO*, 2012.

[15] Bienia Christian. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[16] Ian Cutress. Supercomputing 15: Intel's knights landing xeon phi silicon on display. November 2015.

[17] W. J. Dally and B. Towles. Route packets, not wires: on-chip interconnection networks. In *DAC*, 2001.

[18] G. S. Delp, D. J. Farber, R. G. Minnich, J. M. Smith, and M. C. Tam. Memory as a network abstraction. *IEEE Network*, 5(4), 1991.

[19] S. B. Desai, S. R. Madhvapathy, A. B. Sachid, J. P. Llinas, Q. Wang, G. H. Ahn, G. Pitner, M. J. Kim, J. Bokor, C. Hu, H.-S. P. Wong, and A. Javey. MoS2 transistors with 1-nanometer gate lengths. *Science*, 354(6308):99–102, oct 2016.

[20] Chris Fallin, Chris Craik, and Onur Mutlu. Chipper: A low-complexity bufferless deflection router. In *HPCA*, 2011.

[21] Chris Fallin, Xiangyao Yu, Greg Nazario, and Onur Mutlu. A high-performance hierarchical ring on-chip interconnect with low-cost routers. 2011.

[22] Paul Gratz, Changkyu Kim, Robert McDonald, Stephen W Keckler, and Doug Burger. Implementation and evaluation of on-chip network architectures. In *International Conference on Computer Design*. IEEE, 2006.

[23] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu. Express cube topologies for on-chip interconnects. In *HPCA*, 2009.

[24] D. Harris and N. Weste. *CMOS VLSI Design: A Circuits and Systems Perspective*. Pearson/Addison-Wesley, 2005.

[25] Yatin Hoskote, Sriram Vangal, Arvind Singh, Nitin Borkar, and Shekhar Borkar. A 5-ghz mesh interconnect for a teraflops processor. *IEEE Micro*, 2007.

[26] Jason Howard, S. Dighe, Y. Hoskote, Sriram Vangal, et al. A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling. *IEEE Journal of Solid-State Circuits*, 2011.

[27] C-H Jan, Uddalak Bhattacharya, R Brain, S-J Choi, G Curello, G Gupta, W Hafez, M Jang, M Kang, K Komeyli, et al. A 22nm soc platform technology featuring 3-d tri-gate and high-k/metal gate, optimized for ultra low power, high performance and high density soc applications. In *Electron Devices Meeting (IEDM)*. IEEE, 2012.

[28] N. E. Jerger, L. S. Peh, and M. Lipasti. Virtual circuit tree multicasting: A case for on-chip hardware multicast support. In *ISCA*, 2008.

[29] Nan Jiang, James Balfour, Daniel U Becker, Brian Towles, William J Dally, George Michelogiannakis, and John Kim. A detailed and flexible cycle-accurate network-on-chip simulator. In *ISPASS*. IEEE, 2013.

[30] John Kim, William J. Dally, and Dennis Abts. Flattened butterfly: A cost-efficient topology for high-radix networks. In *ISCA*, 2007.

[31] A. K. Kodi, A. Sarathy, and A. Louri. ideal: Inter-router dual-function energy and area-efficient links for network-on-chip (noc) architectures. In *ISCA*, 2008.

[32] J. Liu, L. R. Zheng, D. Pamunuwa, and H. Tenhunen. A global wire planning scheme for network-on-chip. In *International Symposium on Circuits and Systems (ISCAS)*, 2003.

[33] S. Liu, T. Chen, L. Li, X. Feng, Z. Xu, H. Chen, F. Chong, and Y. Chen. Imr: High-performance low-cost multi-ring nocs. *IEEE Transactions on Parallel and Distributed Systems*, 27(6), 2016.

[34] NanGate, Inc. Nangate freePDK15 open cell library.

[35] S Natarajan, M Agostinelli, S Akbar, M Bost, A Bowonder, V Chikarmane, S Chouksey, A Dasgupta, K Fischer, Q Fu, et al. A 14nm logic technology featuring 2 nd-generation finfet, air-gapped interconnects, self-aligned double patterning and a 0.0588 μm 2 sram cell size. In *IEEE International Electron Devices Meeting*, 2014.

[36] C. A. Nicopoulos, D. Park, J. Kim, N. Vijaykrishnan, M. S. Yousif, and C. R. Das. Vichar: A dynamic virtual channel regulator for network-on-chip routers. In *MICRO*, 2006.

[37] D. Pamunuwa, J. Oberg, L. R. Zheng, M. Millberg, A. Jantsch, and H. Tenhunen. Layout, performance and power trade-offs in mesh-based network-on-chip architectures. In *International Conference on Very Large Scale Integration*, 2003.

[38] M. K. Papamichael and J. C. Hoe. The connect network-on-chip generator. *Computer*, 48(12), 2015.

[39] Martin Radetzki, Chaochao Feng, Xueqian Zhao, and Axel Jantsch. Methods for fault tolerance in networks-on-chip. *ACM Comput. Surv.*, 46(1):8:1–8:38, July 2013.

[40] A. N. Udipi, N. Muralimanohar, and R. Balasubramonian. Towards scalable, energy-efficient, bus-based on-chip networks. In *HPCA*, 2010.

[41] Sriram Vangal, Jason Howard, Gregory Ruhl, Saurabh Dighe, Howard Wilson, James Tschanz, David Finan, Priya Iyer, Arvind Singh, Tiju Jacob, et al. An 80-tile 1.28 tflops network-on-chip in 65nm cmos. In *ISSCC*, 2007.

[42] Laung-Terng Wang, Yao-Wen Chang, and Kwang-Ting (Tim) Cheng, editors. *Electronic Design Automation: Synthesis, Verification, and Test*. Morgan Kaufmann Publishers Inc., 2009.

[43] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F Brown III, and Anant Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 2007.

[44] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *Computing frontiers*. ACM, 2006.

[45] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *ISCA*, 1995.

APPENDICES

US 20170250926A1

(19) **United States**

(12) **Patent Application Publication** (10) **Pub. No.: US 2017/0250926 A1**

Chen et al. (43) **Pub. Date: Aug. 31, 2017**

111

(54) **ROUTERLESS NETWORKS-ON-CHIP**

(71) Applicant: **Oregon State University**, Corvallis, OR (US)

(72) Inventors: **Lizhong Chen**, Portland, OR (US); **Fawaz M. Alazemi**, Corvallis, OR (US); **Bella Bose**, Corvallis, OR (US)

(73) Assignee: **Oregon State University**, Corvallis, OR (US)

(21) Appl. No.: **15/445,736**

(22) Filed: **Feb. 28, 2017**

Related U.S. Application Data

(60) Provisional application No. 62/301,451, filed on Feb. 29, 2016.

Publication Classification

(51) **Int. Cl.**

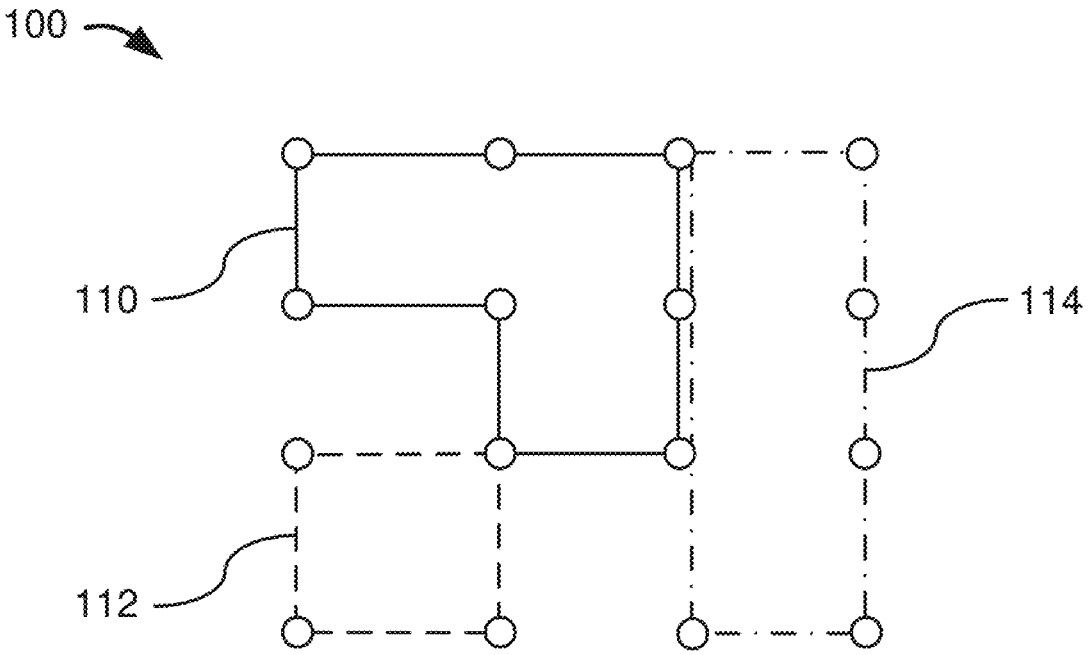
H04L 12/933	(2006.01)
H04L 12/931	(2006.01)
G06F 17/50	(2006.01)

(52) **U.S. Cl.**

CPC ***H04L 49/109*** (2013.01); ***G06F 17/5077*** (2013.01); ***H04L 49/40*** (2013.01)

(57) **ABSTRACT**

The disclosed technology concerns methods, apparatus, and systems for designing and generating networks-on-chip ("NoCs"), as well as to hardware architectures for implementing such NoCs. The disclosed NoCs can be used, for instance, to interconnect cores of a chip multiprocessor (aka a "multi-core processor"). In one example implementation, a wire-based routerless NoC design is disclosed that uses deterministically specified wire loops to connect the cores of the chip multiprocessor. The disclosed technology also comprises network interface architectures for use in an NoC. For example, a core can be equipped with a low-area-cost interface that is deadlock-free, uses buffering sharing, and provides low latency.



112

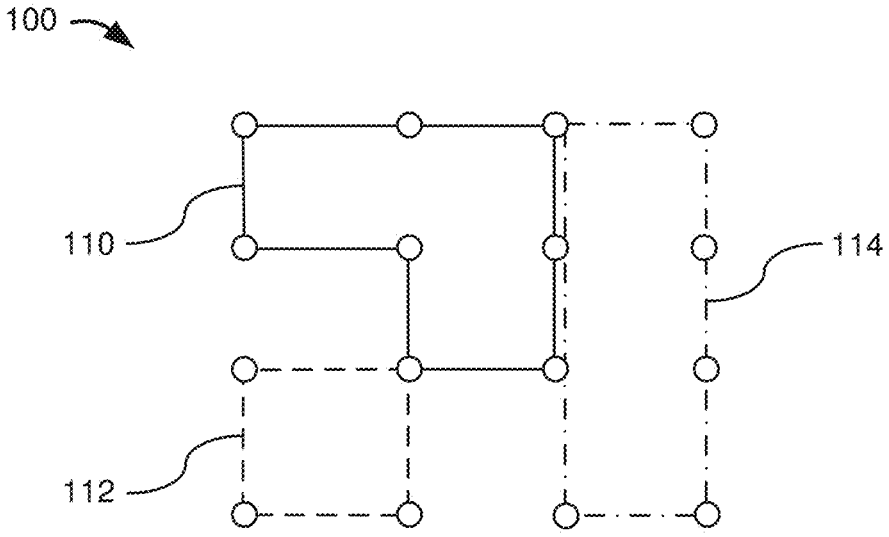


FIG. 1A

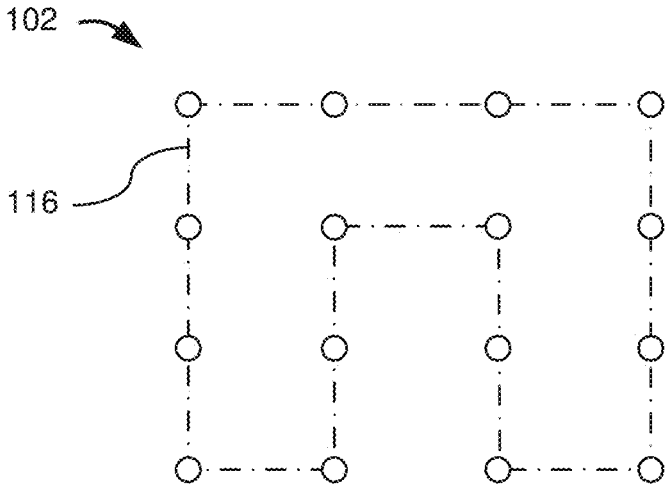


FIG. 1B

113

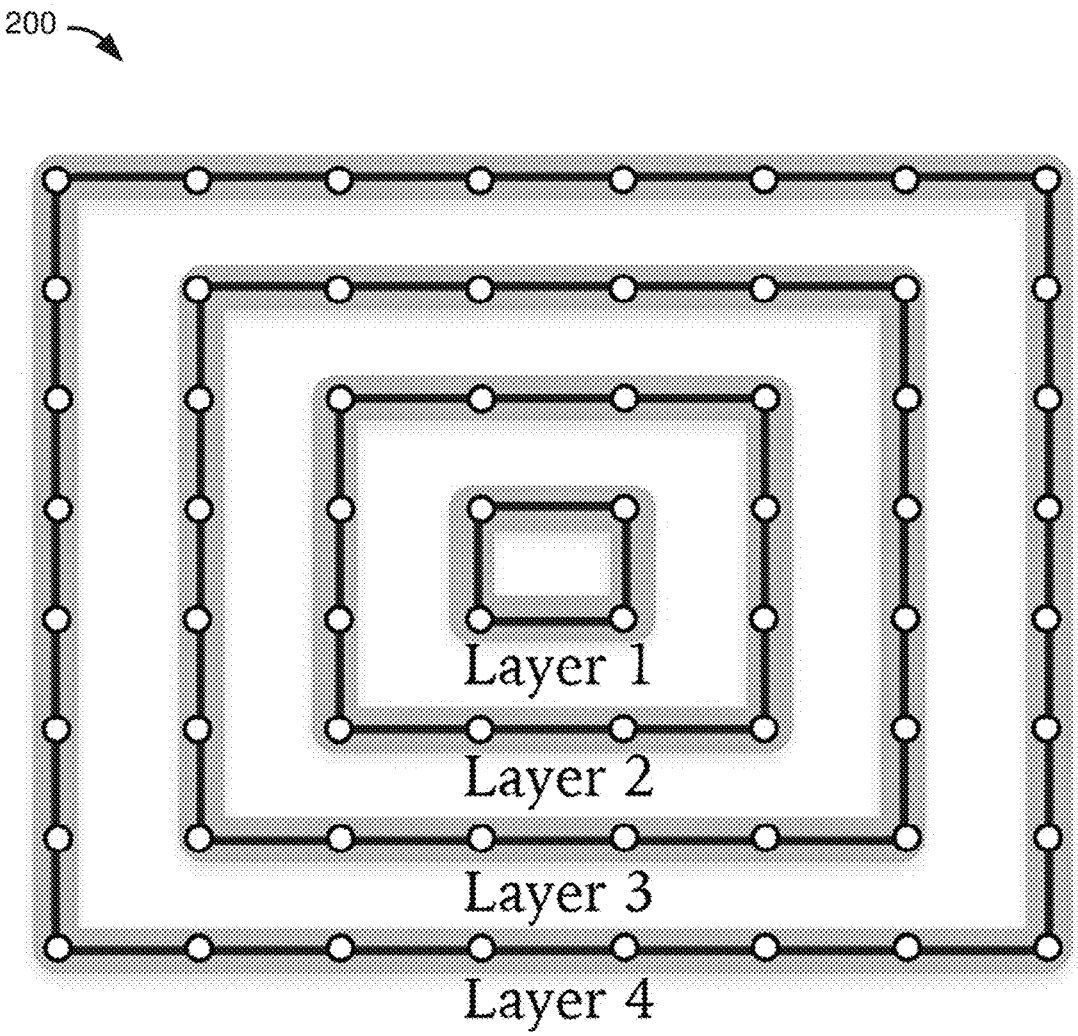


FIG. 2

114

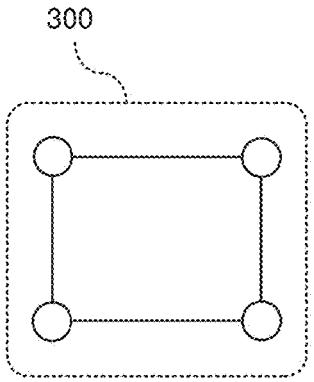


FIG. 3A

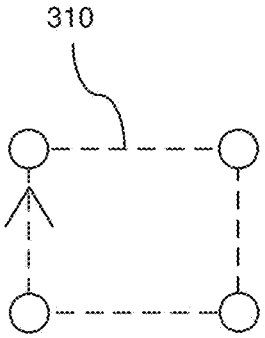


FIG. 3B

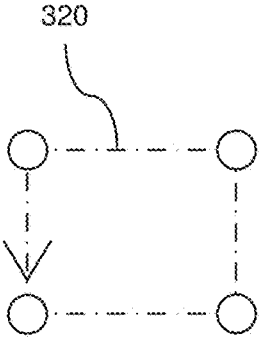


FIG. 3C

115

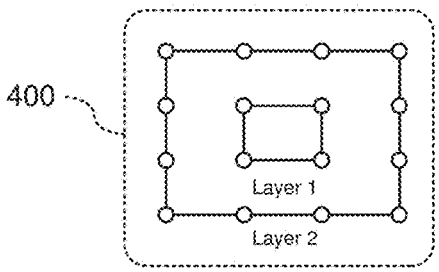


FIG. 4A

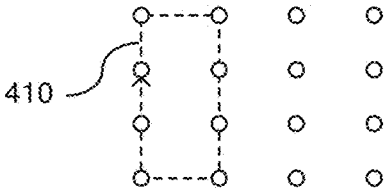


FIG. 4B

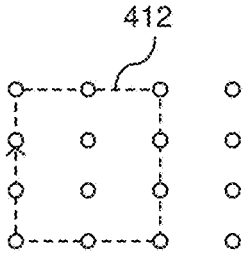


FIG. 4C

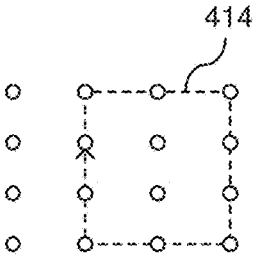


FIG. 4D

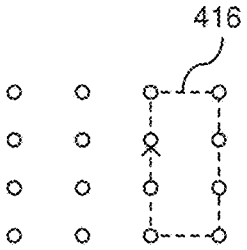


FIG. 4E

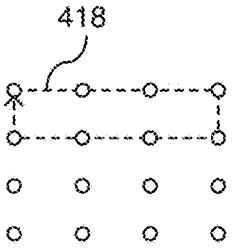


FIG. 4F

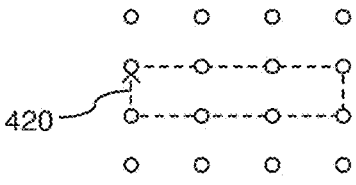


FIG. 4G

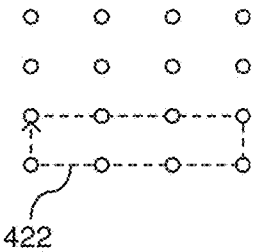


FIG. 4H

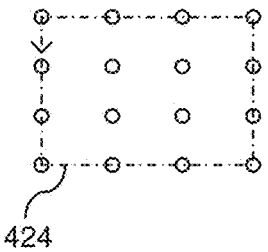


FIG. 4I

116

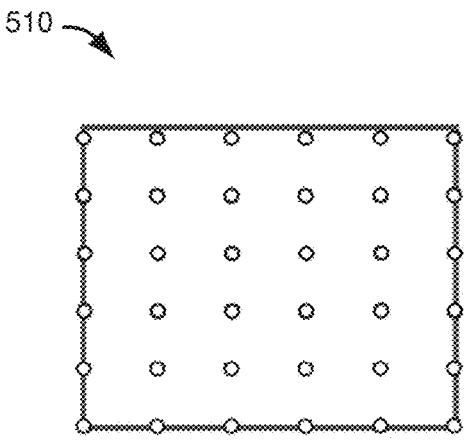


FIG. 5A

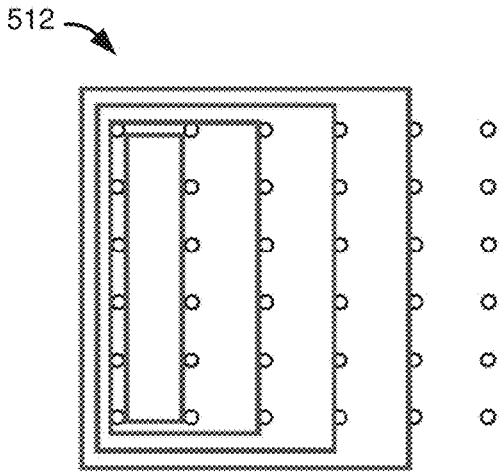


FIG. 5B

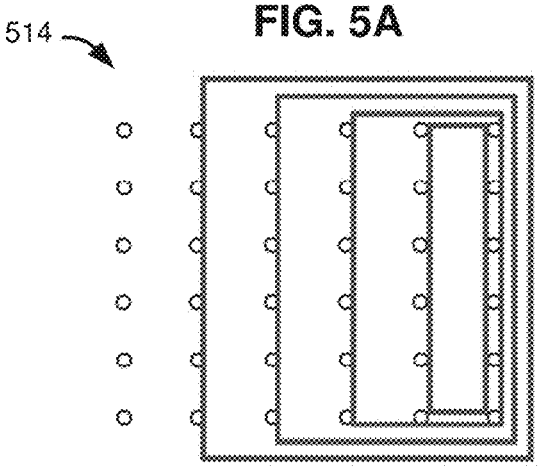


FIG. 5C

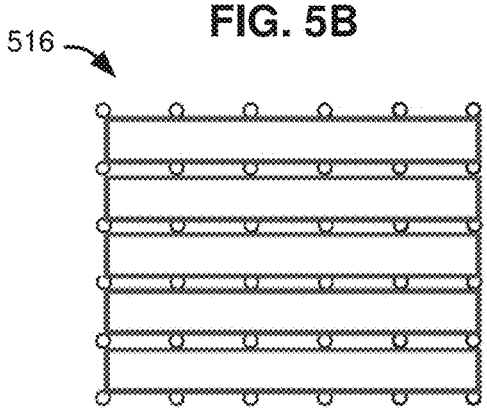


FIG. 5D

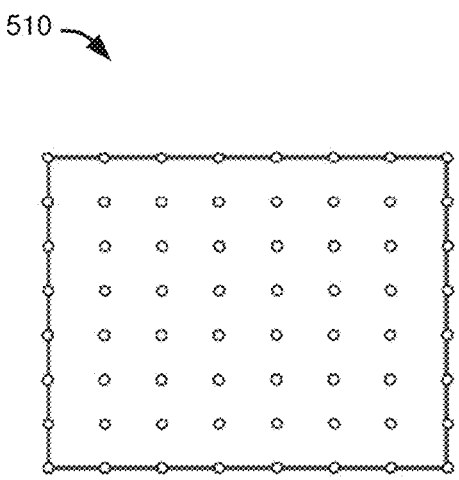


FIG. 6A

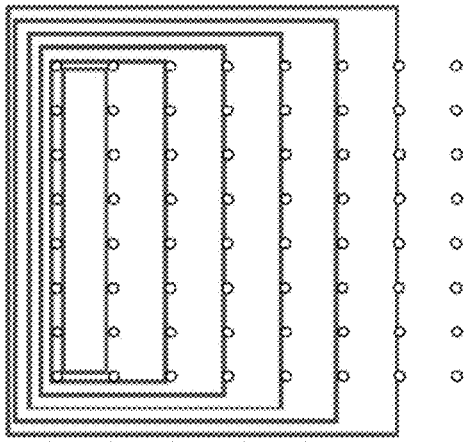


FIG. 6B

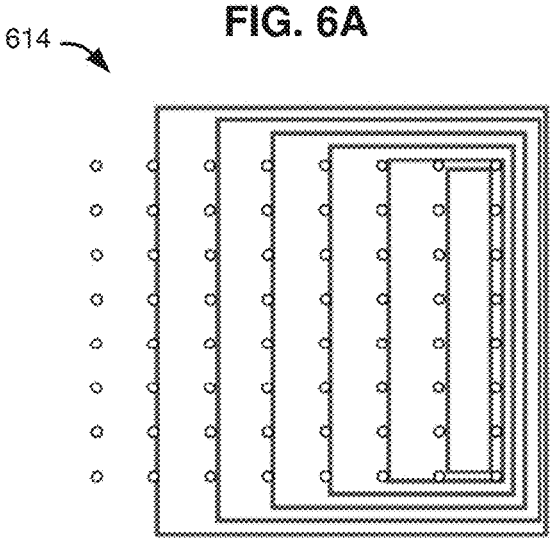


FIG. 6C

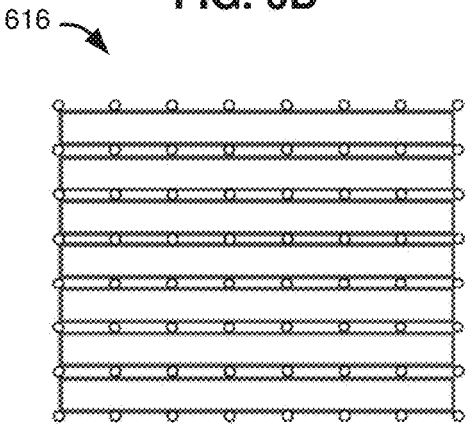


FIG. 6D

700

Algorithm 1: RLrec

Input : L, H: the low and high numbers
Output: M; set of circles

```
1  begin
2      if L = H then
3          return {}
4      Let M = {}
5      if H − L = 1 then
6          M = M ∪ C(L, H, L, H, clockwise)
7          M = M ∪ C(L, H, L, H, anticlockwise)
8          return M
9      M = M ∪ C(L, H, L, H, anticlockwise)
10     for i = L + 1 → H − 1 do
11         M = M ∪ C(L, H, L, i, clockwise)
12         M = M ∪ C(L, H, i, H, clockwise)
13     for i = L → H − 1 do
14         M = M ∪ C (i, i + 1, L, H, clockwise)
15     M′ = RLrec(L+1, H-1)
16     Reverse and rotate for 90° every circle in M′ and add
       them to M″
17     return M ∪ M″
```

FIG. 7

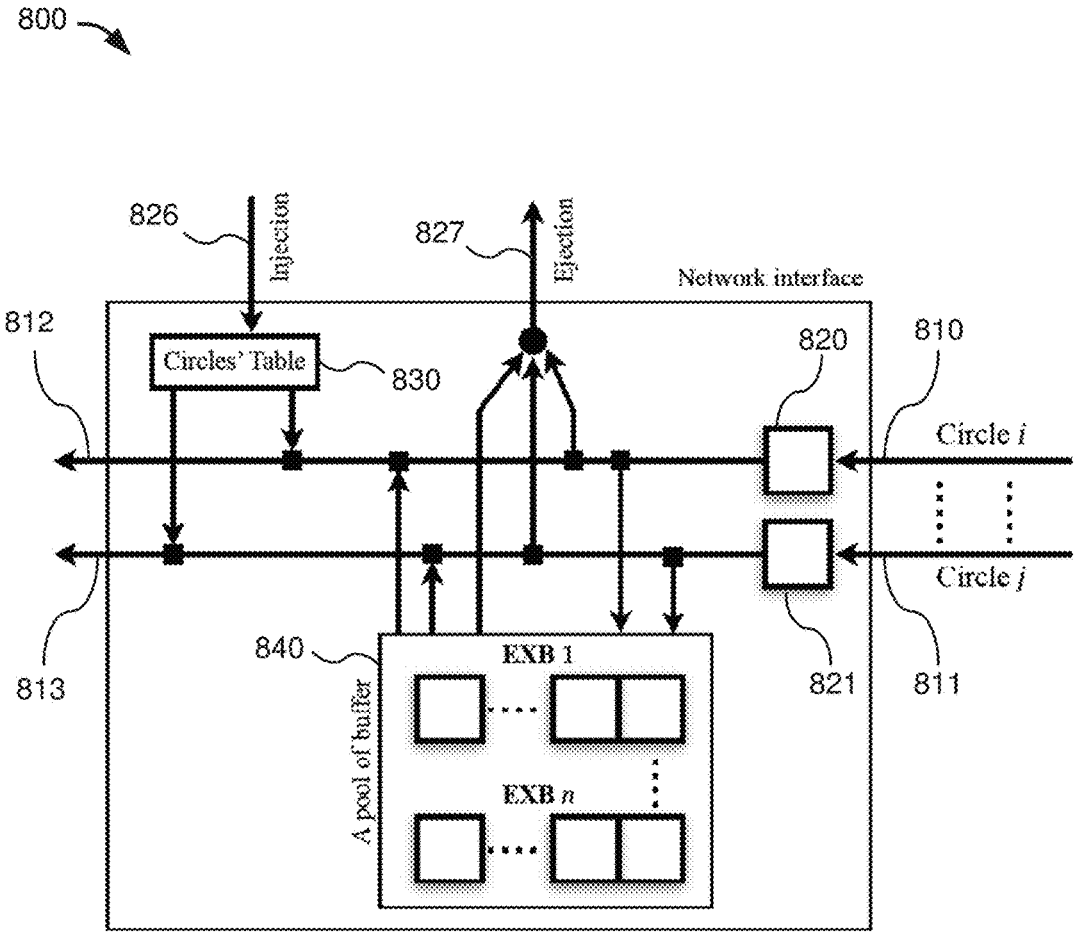

800
826
Injection
827
Ejection
Network interface
812
Circles' Table
830
820
810
Circle i
Circle j
840
EXB 1
EXB n
A pool of buffer
813
821
811

FIG. 8

120

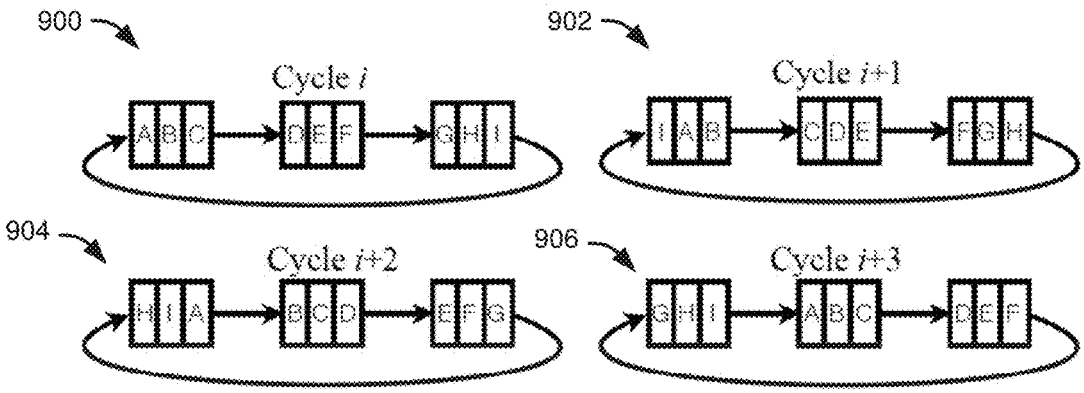


FIG. 9

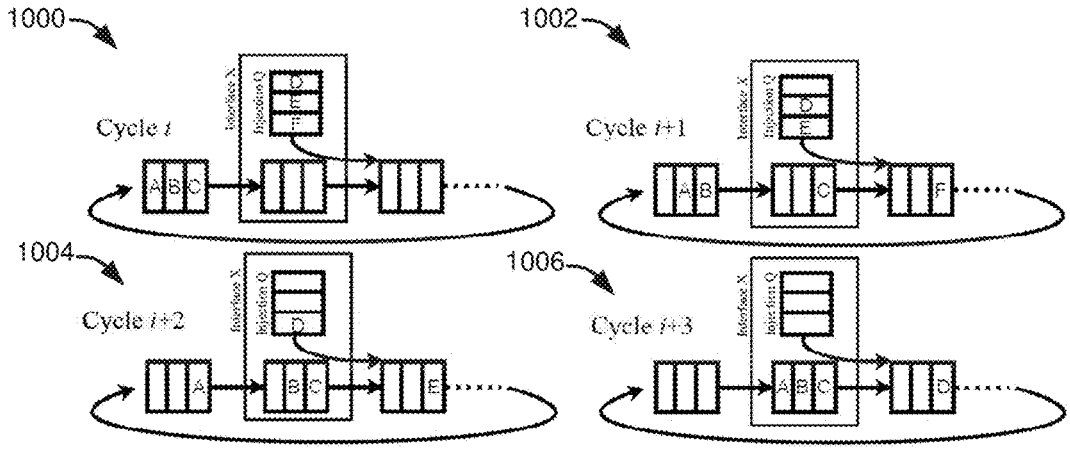


FIG. 10

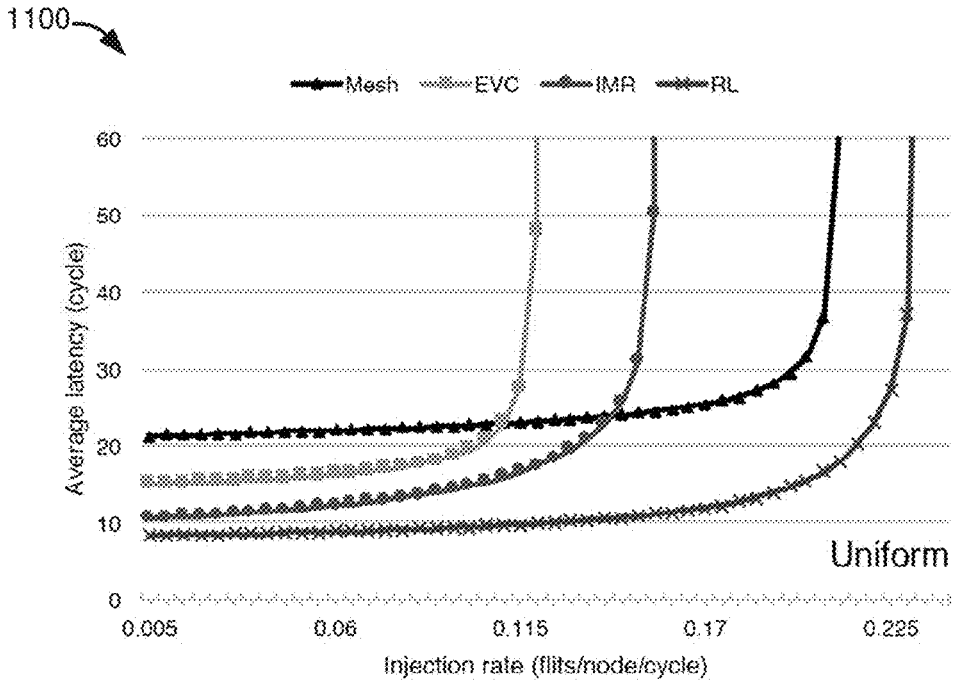


FIG. 11A

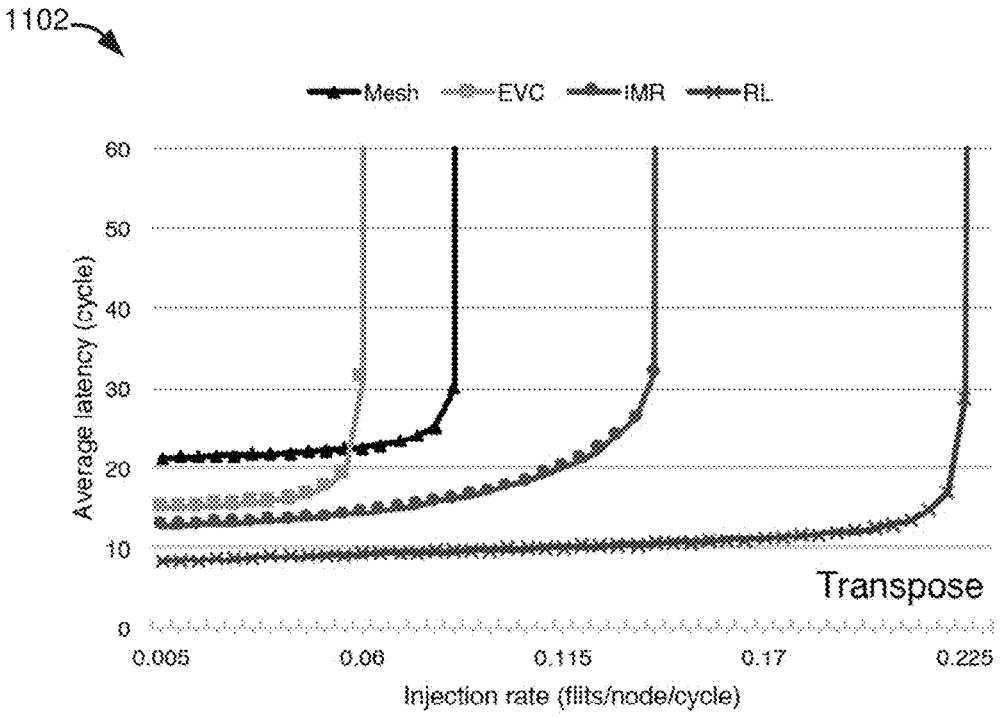


FIG. 11B

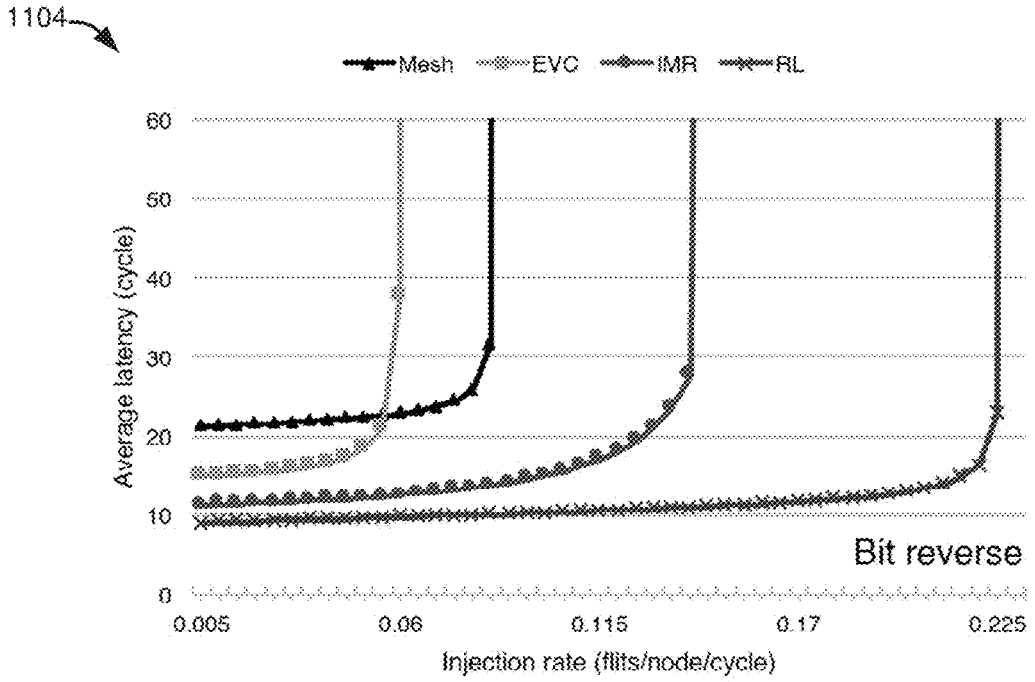


FIG. 11C

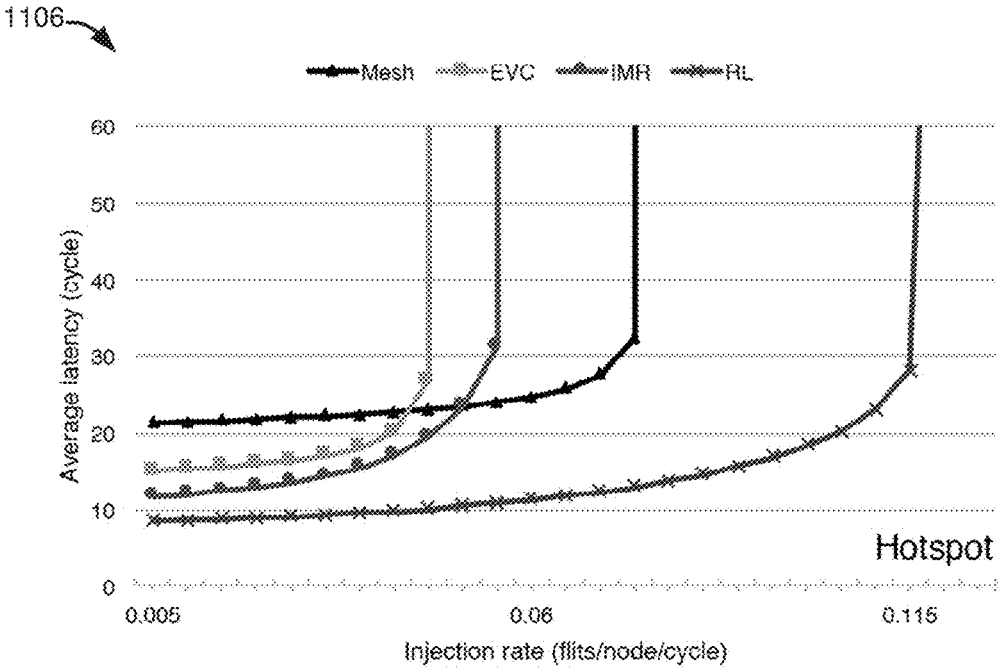


FIG. 11D

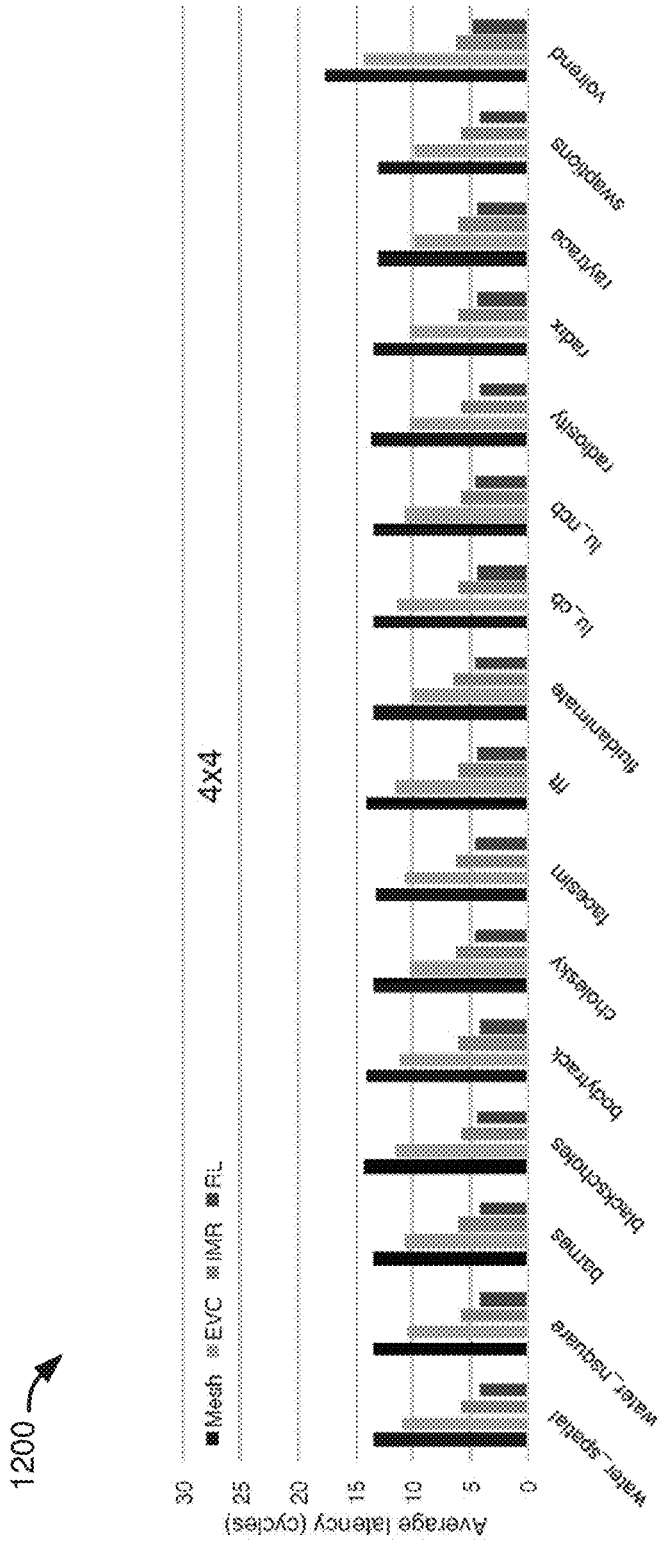

1200
Mesh
EVC
IMR
RL
4x4
Average latency (cycles)
0
5
10
15
20
25
30
water_spatial
water_nsquared
barnes
blackscholes
bodytrack
cholesky
facesim
ft
fluidanimate
lu_cb
lu_ncb
radiosity
radix
raytrace
swaptions
volrend

FIG. 12A

124

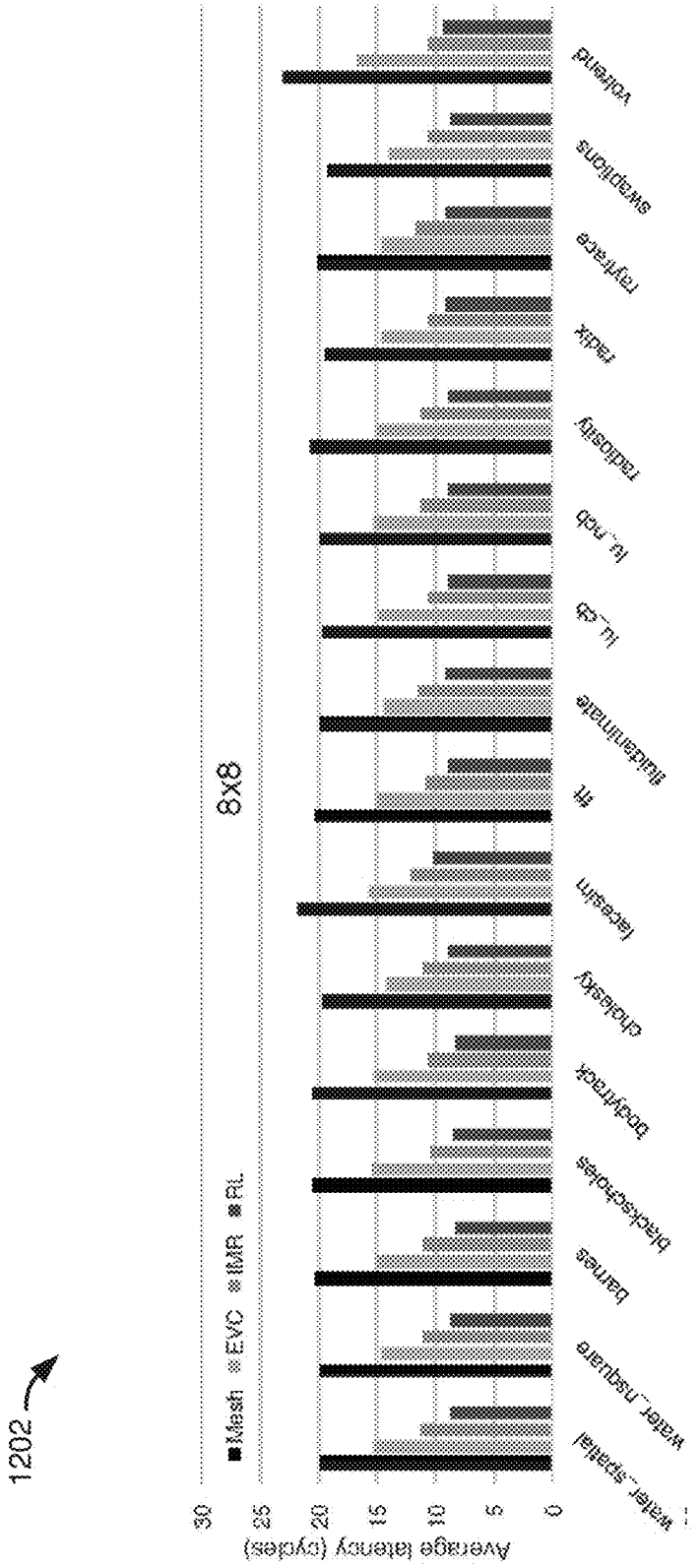

1202
8x8
Mesh
EVC
iMR
RL
Average latency (cycles)
0
5
10
15
20
25
30
water_spatial
water_nsquare
barnes
blackscholes
bodytrack
cholesky
facesim
ft
fluidanimate
lu_cb
lu_ncb
radiosity
radix
raytrace
swaptions
volrend

FIG. 12B

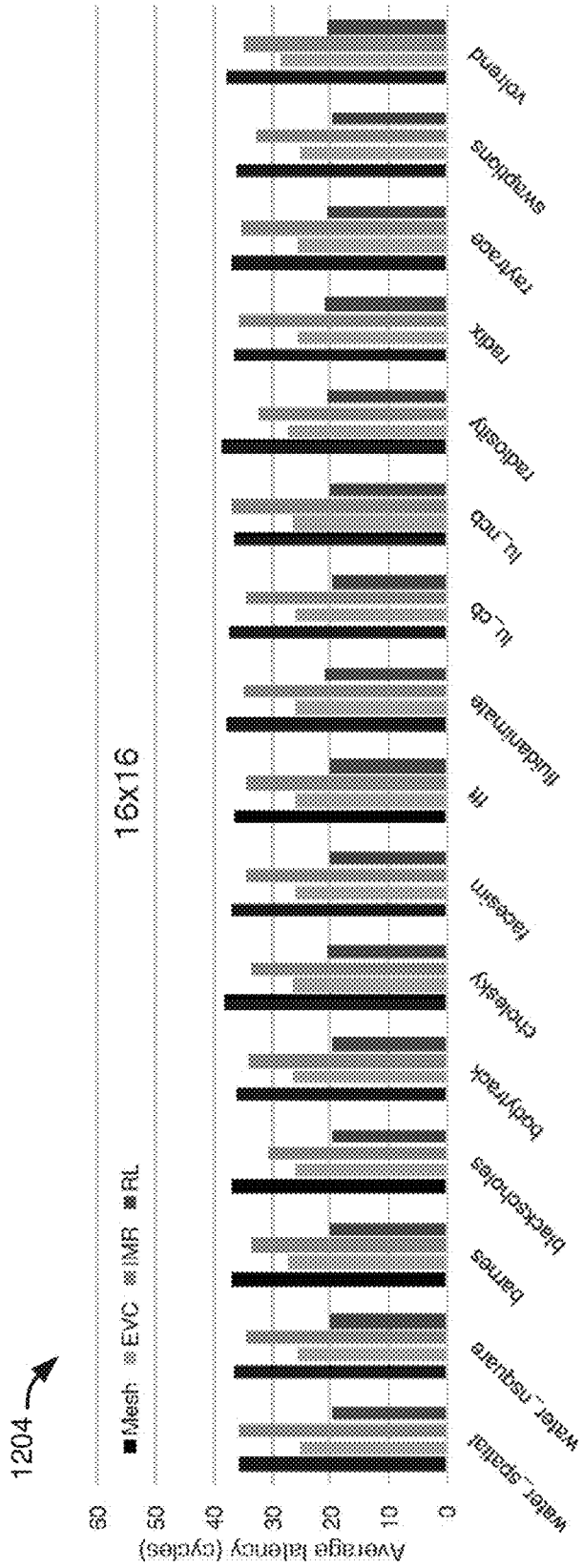
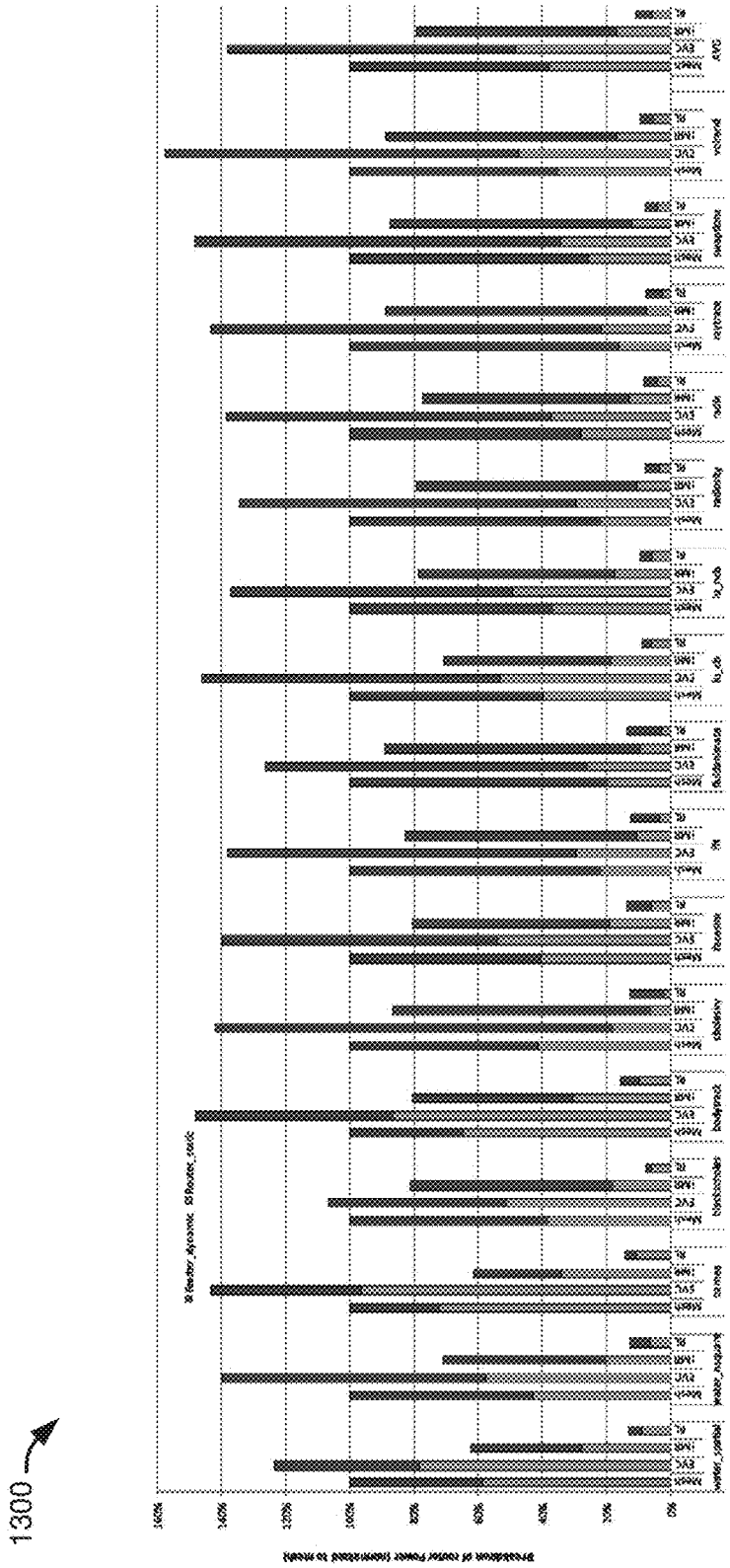

125
1204
16x16
Mesh
EVC
IMR
RL
Average latency (cycles)
0
10
20
30
40
50
60
water_spatial
water_nsquare
barnes
blackscholes
bodytrack
cholesky
facesim
ft
fluidanimate
lu_cb
lu_ncb
radiosity
radix
raytrace
swaptions
volrend

FIG. 12C

126



1300

FIG. 13

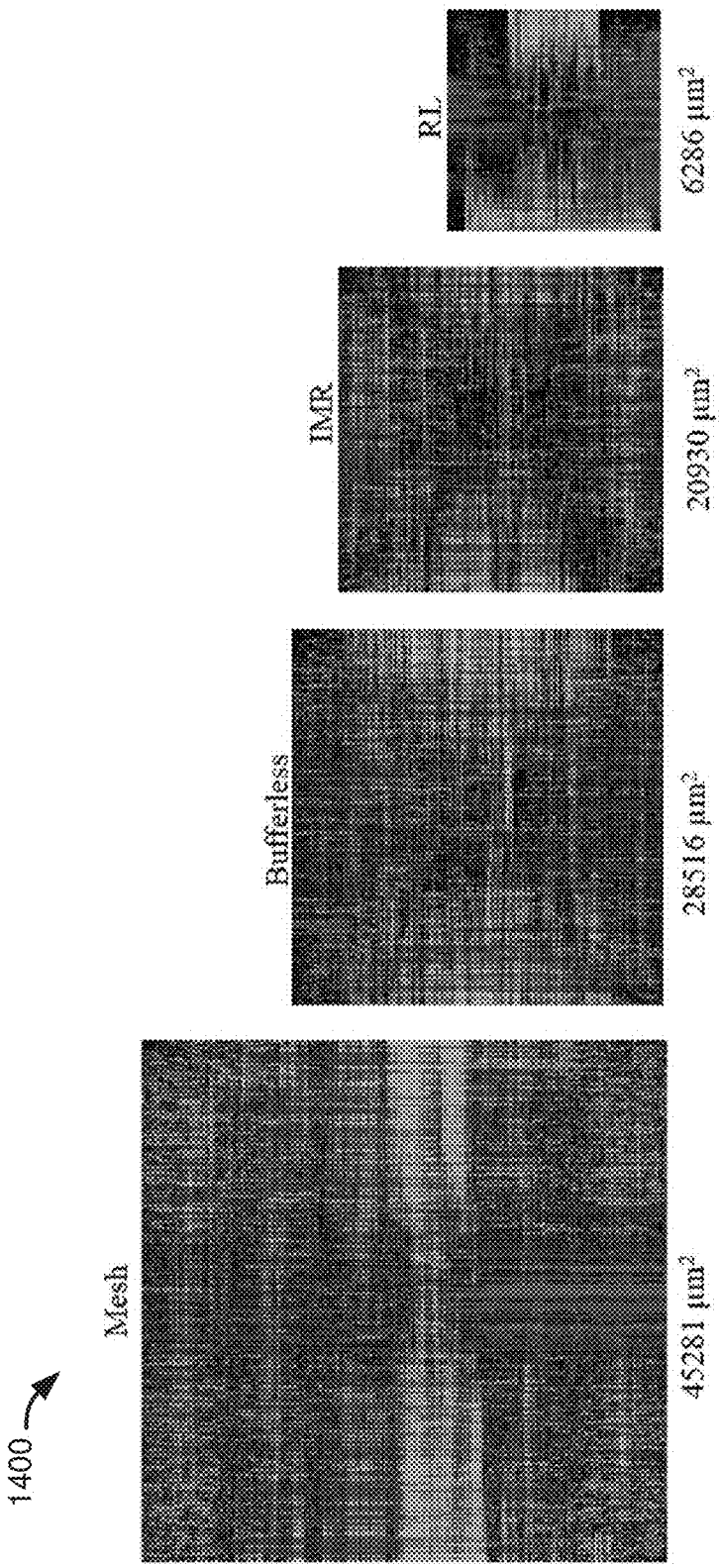

1400
Mesh
45281 μm²
Bufferless
28516 μm²
IMR
20930 μm²
RL
6286 μm²

FIG. 14

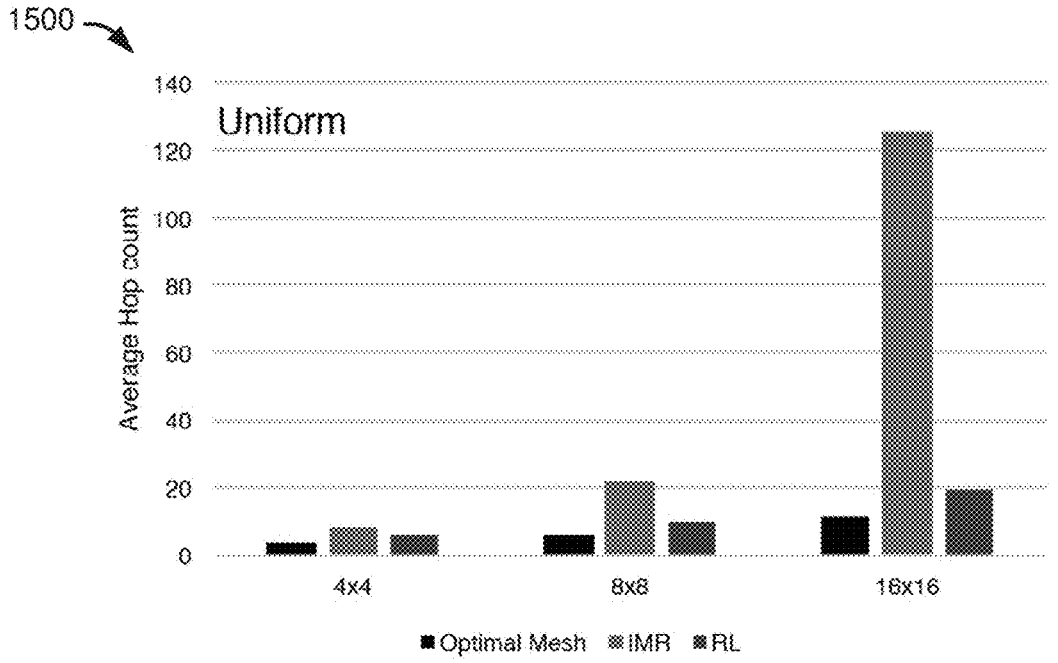


FIG. 15A

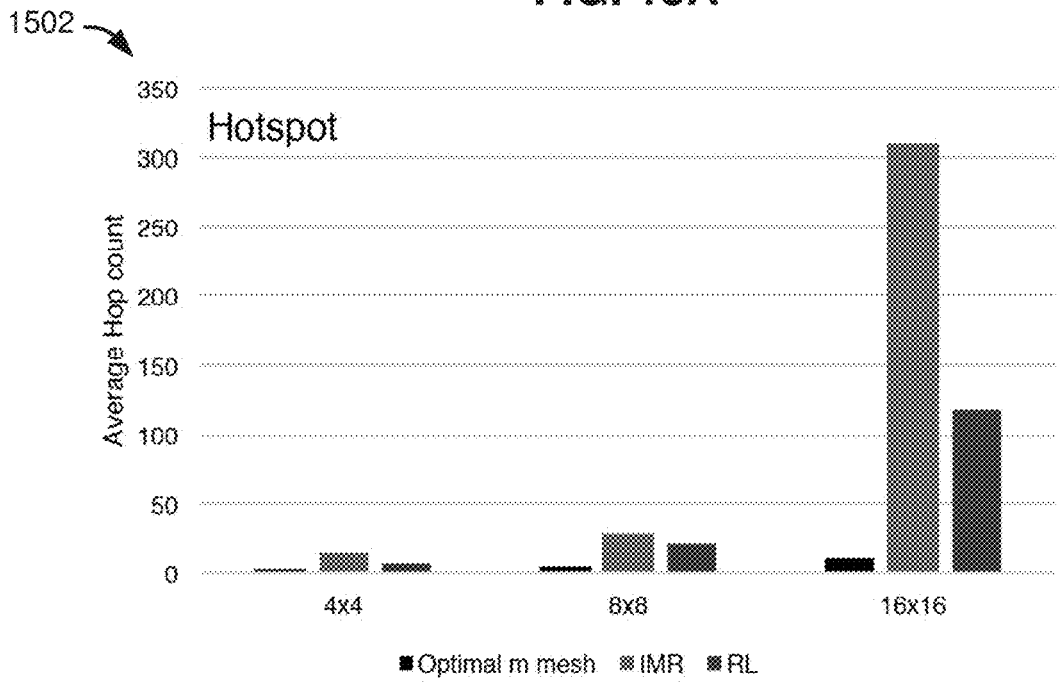


FIG. 15B

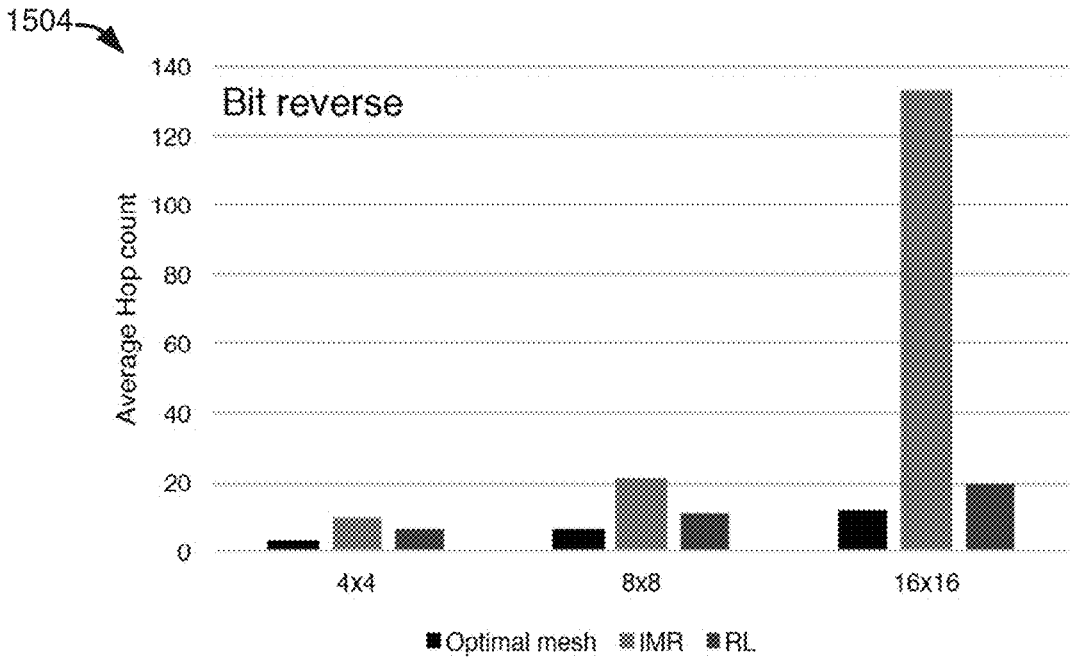


FIG. 15C

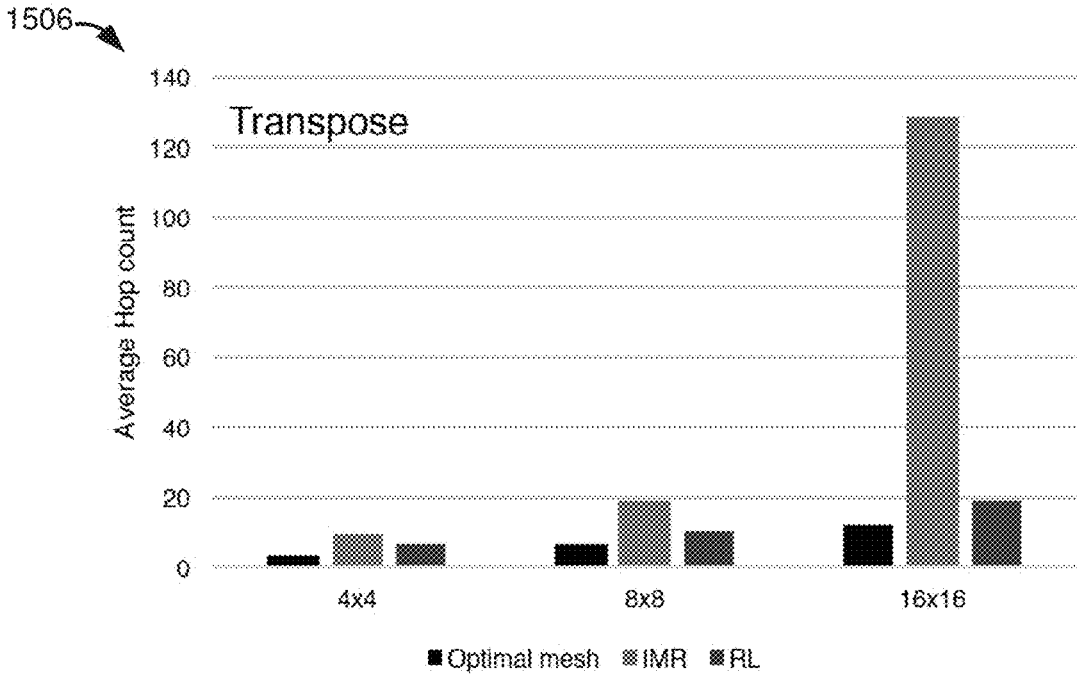


FIG. 15D

130

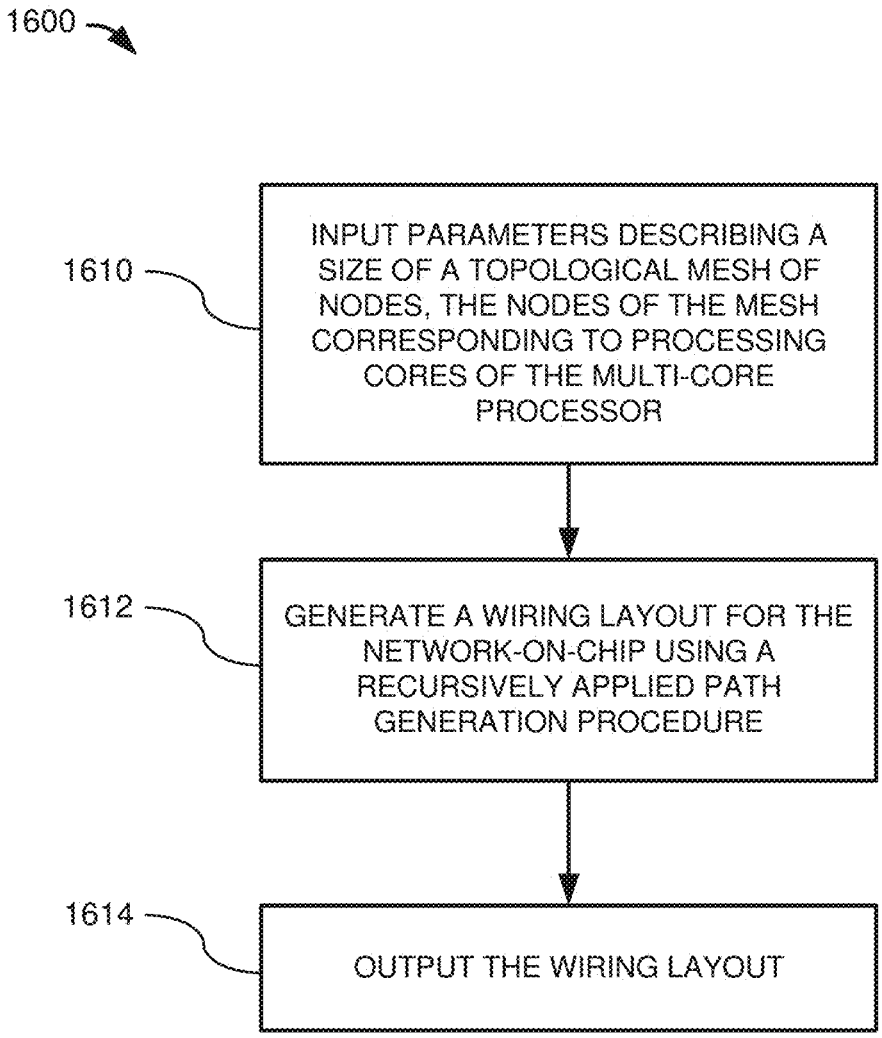

1600
1610
INPUT PARAMETERS DESCRIBING A SIZE OF A TOPOLOGICAL MESH OF NODES, THE NODES OF THE MESH CORRESPONDING TO PROCESSING CORES OF THE MULTI-CORE PROCESSOR
1612
GENERATE A WIRING LAYOUT FOR THE NETWORK-ON-CHIP USING A RECURSIVELY APPLIED PATH GENERATION PROCEDURE
1614
OUTPUT THE WIRING LAYOUT

FIG. 16

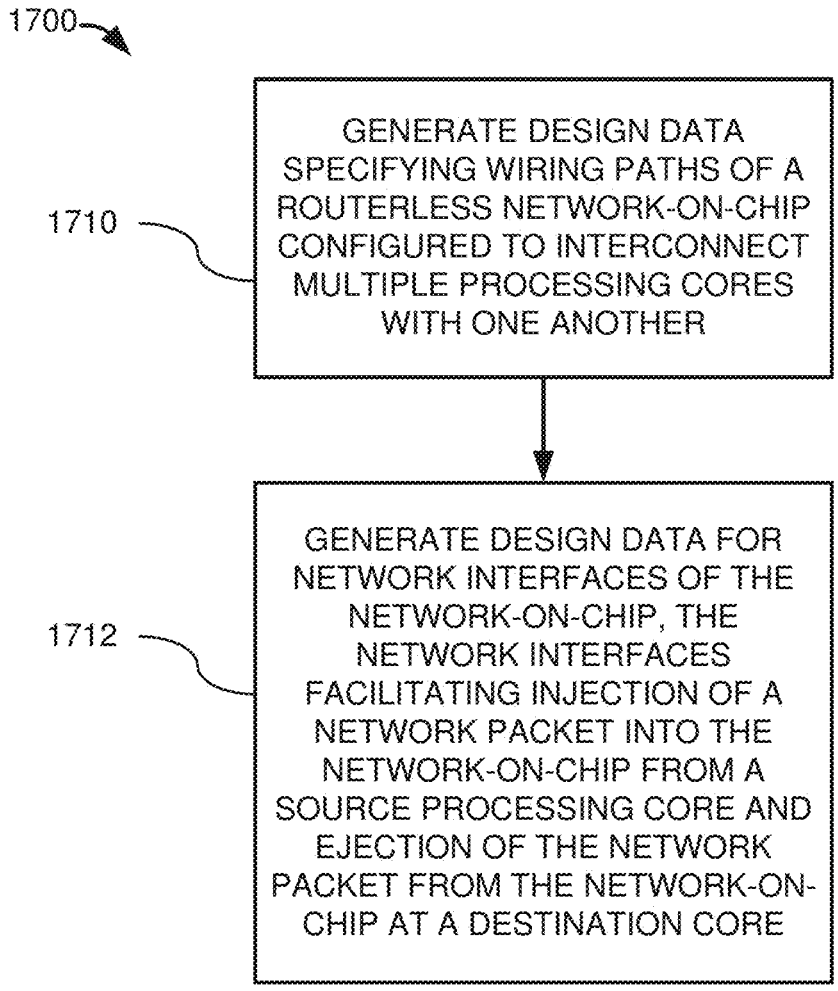

1700
1710
GENERATE DESIGN DATA SPECIFYING WIRING PATHS OF A ROUTERLESS NETWORK-ON-CHIP CONFIGURED TO INTERCONNECT MULTIPLE PROCESSING CORES WITH ONE ANOTHER
1712
GENERATE DESIGN DATA FOR NETWORK INTERFACES OF THE NETWORK-ON-CHIP, THE NETWORK INTERFACES FACILITATING INJECTION OF A NETWORK PACKET INTO THE NETWORK-ON-CHIP FROM A SOURCE PROCESSING CORE AND EJECTION OF THE NETWORK PACKET FROM THE NETWORK-ON-CHIP AT A DESTINATION CORE

FIG. 17

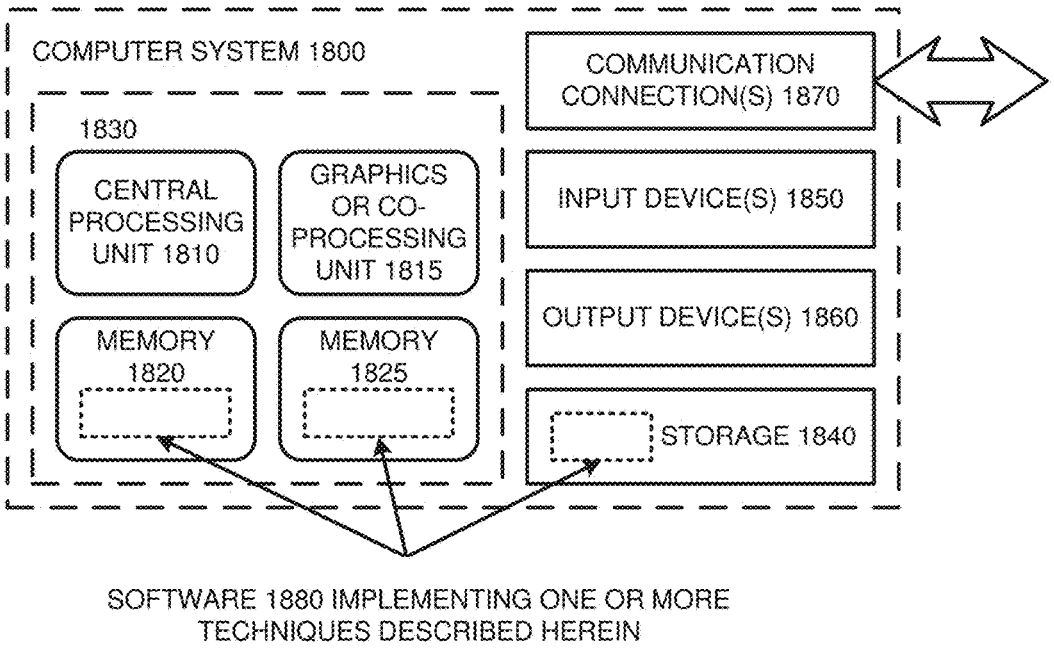

COMPUTER SYSTEM 1800
1830
CENTRAL PROCESSING UNIT 1810
GRAPHICS OR CO-PROCESSING UNIT 1815
MEMORY 1820
MEMORY 1825
COMMUNICATION CONNECTION(S) 1870
INPUT DEVICE(S) 1850
OUTPUT DEVICE(S) 1860
STORAGE 1840
SOFTWARE 1880 IMPLEMENTING ONE OR MORE TECHNIQUES DESCRIBED HEREIN

FIG. 18

133

ROUTERLESS NETWORKS-ON-CHIP

CROSS-REFERENCE TO RELATED APPLICATIONS

[0001] This application claims the benefit of U.S. Provisional Application No. 62/301,451, entitled "LOW-LATENCY ROUTERLESS NETWORK-ON-CHIP DESIGN" and filed on Feb. 29, 2016, which is hereby incorporated herein by reference.

FIELD

[0002] This application concerns networks-on-chip ("NoCs") that are used to interconnect cores of a chip multiprocessor.

SUMMARY

[0003] The disclosed technology concerns methods, apparatus, and systems for designing and generating networks-on-chip ("NoCs"), as well as to hardware architectures for implementing such NoCs. The disclosed NoCs can be used, for instance, to interconnect cores of a chip multiprocessor (also referred to as a "multi-core processor"). The disclosed methods, apparatus, and systems should not be construed as limiting in any way. Instead, the present disclosure is directed toward all novel and nonobvious features and aspects of the various disclosed embodiments, alone or in various combinations and subcombinations with one another.

[0004] In general, an NoC connects multiple cores of a chip-multiprocessor ("CMP") (also referred to as a "multi-core processor"). The efficiency of NoCs can greatly affect the performance and cost of a CMP. An NoC may use on-chip routers, but such routers demand high power consumption and area.

[0005] In this disclosure, architectures that, among other things, reduce or eliminate routers from the design are disclosed. For instance, in one example implementation, a routerless ("RL") NoC design is disclosed that is wire-based (e.g., solely wire-based). For instance, by utilizing wiring resources of an NoC effectively, one can form multiple circular paths (e.g., wiring paths that form a loop and that typically have a rectangular shape) to interconnect cores (e.g., all cores) on a CMP without routers. Also disclosed herein are memory- and resource-efficient techniques for identifying and/or specifying the circular paths. For instance, one such technique is a fast, efficient recursive algorithm where every pair of cores shares at least one circular path.

[0006] The disclosed technology also comprises network interface architectures for use in an NoC. For example, a core can be equipped with a low-area-cost interface that provides low latency (e.g., 1 cycle per hop latency). Further, in some cases, the circular paths (e.g., all circular paths) related to a respective core pass through its interface. Moreover, in certain embodiments, and in the favor of reducing the area and power, deadlock-free buffer sharing techniques are provided such that each input port has a small packet size buffer and/or a set of available long packet size buffers to be shared among input ports.

[0007] The innovations can be implemented as part of an NoC hardware architecture (e.g., on a CMP or multi-core processor). The innovations can also be implemented as part of an electronic design automation ("EDA") tool used to generate design data for the NoC for a particular CMP or multi-core processor design (e.g., as a part of a behavioral synthesis tool that generates HDL data, a logic synthesis and/or place-and-route tool that generates gate-level netlists, a physical synthesis tool that generates physical layouts (e.g., GDSII data), or any other suitable EDA tool). In this regard, the innovations can be implemented as a method (e.g., a NoC design method, a multicore processor chips, a many-core processor), as part of a computing system configured to perform the method, or as part of computer-readable media storing computer-executable instructions for causing a processing device (e.g., a circuit, such as a microprocessor or microcontroller), when programmed thereby, to perform the method. Using the design data generated from an EDA tool implementing the disclosed techniques, mask-level models can ultimately be produced, masks can be printed, and the final integrated circuit can be fabricated (e.g., using suitable lithography techniques).

[0008] The foregoing and other objects, features, and advantages of the disclosed technology will become more apparent from the following detailed description, which proceeds with reference to the accompanying figures.

BRIEF DESCRIPTION OF THE DRAWINGS

[0009] FIGS. **1**A-**1**B are schematic block diagrams of a 4×4 topology of nodes in which several example circular paths (circles), in accordance with the disclosed technology, are depicted.

[0010] FIG. **2** is a schematic block diagram illustrating the layers for an 8×8 mesh, where each layer is labeled.

[0011] FIGS. **3**A-C are schematic block diagrams showing a 2×2 mesh topology as well as a clockwise and counter-clockwise circle for the topology.

[0012] FIGS. **4**A-I are schematic block diagrams showing a 4×4 mesh topology as well as various circles for connecting nodes of the topology to one another in accordance with the disclosed technology.

[0013] FIGS. **5**A-D are schematic block diagrams showing a 6×6 mesh topology as well as various circles for connecting nodes of the topology to one another in accordance with the disclosed technology.

[0014] FIGS. **6**A-D are schematic block diagrams showing a 8×8 mesh topology as well as various circles for connecting nodes of the topology to one another in accordance with the disclosed technology.

[0015] FIG. **7** shows example pseudocode for a technique that generates node-connecting circles for an arbitrary NoC design.

[0016] FIG. **8** is a schematic block diagram of an example network interface with exemplary components.

[0017] FIG. **9** shows schematic block diagrams illustrating how flow control units can loop in a circle for four incremental clock cycles in accordance with one example embodiment.

[0018] FIG. **10** shows schematic block diagrams illustrating how a packet of three flow control units can be injected into an example embodiment of the interface.

[0019] FIGS. **11**A-D show plots illustrating the latency and throughput details for various experiments performed comparing the disclosed design techniques with other techniques.

[0020] FIGS. **12**A-C show plots illustrating the latency performance of the disclosed design techniques relative to other techniques using various benchmarks.

134

[0021] FIGS. **13** show plots illustrating the power consumption of the disclosed design techniques relative to other techniques using various benchmarks.

[0022] FIG. **14** is an image showing the layout and area differences of the routers and interfaces for the various tested designs.

[0023] FIGS. **15**A-D show plots illustrating the average hop count of the disclosed design techniques relative to other techniques for various mesh topologies.

[0024] FIG. **16** is a flow chart showing a generalized example embodiment for implementing an NoC generation technique according to the disclosed technology.

[0025] FIG. **17** is a flow chart showing another generalized example embodiment for implementing an NoC generation technique according to the disclosed technology.

[0026] FIG. **18** illustrates a generalized example of a suitable computer system in which the described innovations may be implemented.

DETAILED DESCRIPTION

I. General Considerations

[0027] Disclosed below are representative embodiments of methods, apparatus, and systems for generating routerless networks-on-chip ("NoCs") as well as to hardware architectures for implementing such NoCs. The disclosed methods, apparatus, and systems should not be construed as limiting in any way. Instead, the present disclosure is directed toward all novel and nonobvious features and aspects of the various disclosed embodiments, alone or in various combinations and subcombinations with one another. Furthermore, any features or aspects of the disclosed embodiments can be used in various combinations and subcombinations with one another. For example, one or more method acts from one embodiment can be used with one or more method acts from another embodiment and vice versa. Further, the various innovations can be used in combination or separately. The disclosed methods, apparatus, and systems are not limited to any specific aspect or feature or combination thereof, nor do the disclosed embodiments require that any one or more specific advantages be present or problems be solved.

[0028] Although the operations of some of the disclosed methods are described in a particular, sequential order for convenient presentation, it should be understood that this manner of description encompasses rearrangement, unless a particular ordering is required by specific language set forth below. For example, operations described sequentially may in some cases be rearranged or performed concurrently. Moreover, for the sake of simplicity, the figures may not show the various ways in which the disclosed methods can be used in conjunction with other methods.

[0029] Various alternatives to the examples described herein are possible. For example, some of the methods described herein can be altered by changing the ordering of the method acts described, by splitting, repeating, or omitting certain method acts, etc. The various aspects of the disclosed technology can be used in combination or separately. Different embodiments use one or more of the described innovations. Some of the innovations described herein address one or more of the problems noted in the background. Typically, a given technique/tool does not solve all such problems.

[0030] As used in this application and in the claims, the singular forms "a," "an," and "the" include the plural forms unless the context clearly dictates otherwise. Additionally, the term "includes" means "comprises." Further, as used herein, the term "and/or" means any one item or combination of any items in the phrase.

II. Introduction to the Disclosed Technology

[0031] Networks-on-Chip ("NoC") are becoming an increasingly significant component of chip multiprocessor ("CMP") or multi-core processor designs. Generally speaking, the NoC is the backbone that facilitates communication among multiple cores. With NoCs, processing cores can be effectively interconnected on a single chip. Similar to computer networks, NoCs directly affect many performance and cost factors of a CMP. In fact, an NoC can have a great impact on electrical and physical properties (such as power and area) of a CMP design. For instance, the power consumption of an NoC on a CMP can be 10%~36%—a highly undesirable proportion. See Hoskote et al., "A 5-GHz Mesh Interconnect for a Teraflops Processor," *IEEE Micro*, vol. 27, no. 5, pp. 51-61 (September-October 2007); Howard et al., "A 48-Core IA-32 Processor in 45 nm CMOS Using On-Die Message-Passing and DVFS for Performance and Power Scaling," *IEEE Journal of Solid-State Circuits*, vol. 46, no. 1, pp. 173-183 (January 2011). Further, with continuous advancements in processing technologies and higher clock frequencies, CMP power is increasingly becoming a major concern. Consequently, new NoC design approaches that work efficiently and effectively to achieve both high performance and with lower area cost are becoming ever more desirable.

[0032] Traditionally, there have been two types of NoCs: bus-based and router-based. A bus-based NoC has a very simple design. In this type of system, cores are attached to a bus that facilitates communications among the cores. This system functions well for few cores; however, as one attaches more cores to the bus, the system's performance degrades significantly. One of the main reasons is the distance between cores: the more cores attached to the bus, the longer the length of the bus. Hence, bus-based NoCs do not scale well with the number of nodes.

[0033] Router-based NoCs use routers, which are attached to each core. A router is a complex and relatively large component that must be carefully designed. With routers, a desired hop count is achieved by directing traffic appropriately through their shortest paths. This is typically attained by complex router designs and routing algorithms. For example, a flow control unit ("flit") has to pass through four stages on a conventional router (specifically, routing computation, VC allocation, switch allocation, and switch traversal) to correctly determine its output port. Those stages require a flit to traverse a router for several clock cycles which, as a result, affects the flit's latency. Moreover, a router comprises several components, such as buffers and crossbars, that contribute to the router's area and power consumption. For example, 28% of the total power and 17% of the die area are devoted to routers on the Intel Terascale chip. See Hoskote et al., "A 5-GHz Mesh Interconnect for a Teraflops Processor," *IEEE Micro*, vol. 27, no. 5, pp. 51-61 (September-October 2007). Notably, such area and power requirements will only be higher as more cores are added to the CMP.

135

[0034] In this disclosure, example embodiments of a routerless NoC design are described. In certain examples, a new NoC design approach is disclosed that intelligently uses available wiring resources that have previously been under-utilized in router-based design due to their commitment to routers. Embodiments of the disclosed technology are driven by the appreciation that the data on an NoC is ultimately transferred by wires. A wire is the elemental part of any digital circuit, and its function is to transport a signal from one point to another. With wires, components (such as transistors, gates, flip-flops, and the like) can communicate and exchange data and, as a result, be integrated into more complex components, such as multiplexer, buffers, and even routers.

[0035] In certain example approaches of the disclosed technology, wires are used to connect cores in predefined circular paths. In particular implementations, each circular path is isolated from the others and all paths are arranged such that every pair of cores has a path. This predefinition and isolation of paths reduces or eliminates the need for routers and, hence, produces a savings in power consumption and area. Furthermore, in certain examples, the circular paths are intelligently placed on the NoC in order to achieve a desirable average hop count. Example of a recursive techniques for generating such circles are also disclosed herein.

[0036] Additionally, in certain embodiments of the disclosed technology, the cores (e.g., each core) are attached to the circular paths using a network interface that allows the sharing of buffer resources among circles passing through the interface. Unlike router-based NoCs, the network interface designs disclosed herein dramatically reduce power and area requirements and allow die areas to accommodate more processing and storage units.

III. Technical Observations and Challenges

[0037] As process technologies scale down to smaller dimensions, more and more features and devices can be fit onto a silicon surface. With this increasing trend in the number of available features and devices on the silicon surface, each technology node comes with more and more metal layers to meet the growing demand for integration. For example, typical many-core processor chips, such as Xeon Phi, Knights Landing or KiloCore, are fabricated using a process technology with 11 to 13 metal layers. Further, each metal layer has a pitch size that defines a minimum wire width and space between two adjacent wires. The pitch size is one of the principle factors in determining the number of available wires in each layer. In modern technology nodes, several metal layers with different thicknesses and minimum pitches are available. These physical differences between metal layers also result in different electrical characteristics (such as resistance and capacitance) and give designers an avenue for meeting their design constraints (such as delay on critical nets) by switching between different layers.

[0038] Designers are also confronted with other challenges when designing integrated circuits, such as CMPs. For instance, one of the challenging issues in interconnect design in modern technologies is crosstalk noise. In general, there are two main techniques to cope with crosstalk noise: (a) spacing; and (b) shielding. For the spacing technique, the interconnect designer tries to keep the coupling noise at a level which is tolerable by the target process and applies a desired space between wires for each layer. See, e.g., Arunachalam et al., "Optimal shielding/spacing metrics for low power design," *IEEE Computer Society Annual Symposium on VLSI*, pp. 167-172 (2003). For the shielding technique, the designer typically reduces the crosstalk noise between two adjacent wires by inserting another wire (which is usually connected to the ground or supply voltage) between them. See id. In comparison with the spacing technique, the shielding technique has more area overhead and it reduces the number of wires in each layer, but it can almost entirely suppress crosstalk noise.

[0039] Table 1 below shows statistical information for a set of example many-core processors, including the wiring resources available in two respective metal layers.

TABLE 1

ManyCore Processor	Intel Teraflop			Intel IA-32 Message-Passing Processor (SCC)			KiloCore			Xeon Phi, Knights Landing		
Number of Cores	80			48			1000			72		
Die area	(21.72 mm × 12.64 mm) 275 mm^2			(26.5 mm × 21.4 mm) 567.1 mm^2			(8 mm × 8 mm) 64.0 mm^2			(31.9 mm × 21.4 mm) 683 mm^2		
Technology	Intel65 nm			Intel45 nm			IBM 32 nm			Intel14 nm		
Interconnect	8 Metal Layers			9 Metal Layers			11 Metal Layers			13 Metal Layers		
Inter-router interconnects Layer	Metal Layer	Pitch	#Wire	Metal Layer	Pitch	#Wire	Metal Layer	Pitch	#Wire	Metal Layer	Pitch	#Wire
	M4	280 nm	22571 wires	M4	240 nm	44583 wires	2 Layers 1X Metal	100 nm	80000 wires	M4	80 nm	133750 wires
	M5	330 nm	19151 wires	M5	280 nm	38214 wires				M5	104 nm	102884 wires
Total Wires	41722 wires (40K)			82797 wires (82K)			80000 wires (80K)			236634 wires (236K)		

[0040] As mentioned, a more conservative approach to cope with the coupling noise is to use a shielding technique. The number of wires in Table 1 was calculated taking into account the area overhead of using a shielding technique to suppress the crosstalk noise Minimum metal pitches are used to estimate the number of wires for each layer.

[0041] As is revealed in Table 1, there has been a trend toward increasing the available number of wires with technology scaling and more advanced multi-core processors. Unfortunately, wires are underutilized in router-based NoCs. As the number of wires is relatively large, this opens new opportunities for new design directions that utilize more wires. An NoC that is smartly design based on wires allows routers to be removed, resulting in a routerless NoC.

[0042] A few earlier works have suggested the removal of routers. However, those works suffered from many factors,

such as scalability. For instance, a point-to-point design to connect every core with the other is infeasible. For example, each node on a 4×4 mesh NoC would have 16×15=240 input and output links. This approach clearly requires a large number of buffers and results in an extremely costly NoC. A shared bus, or conventional bus, is another approach. However, the number of nodes that can be attached to such buses is limited due to noise and collision factors. As more nodes are attached to a bus, the more noise and collisions occur, thus reducing the overall performance. Ring NoCs are another possible approach to connecting a few cores. See, e.g., Barroso et al., "The performance of cache-coherent ring-based multiprocessors." *ACM SIGARCH Computer Architecture News*, vol. 21. no. 2. ACM, (1993); Delp et al., "Memory as a network abstraction," *IEEE Network,* 5(4), pp. 34-41 (1991); and Barroso et al., "Cache Coherence on a Slotted Ring," *ICPP* (1), pp. 230-237 (1991). Like the shared bus approach, however, this approach suffers from scalability and performance issues. As the number of cores on a CMP increases, rings become very slow and no longer scalable.

[0043] Recently, a multi-ring NoC approach called integrated multiple rings (IMR) was introduced. See Liu et al., "IMR: High-Performance Low-Cost Multi-Ring NoCs," *IEEE Transactions on Parallel and Distributed Systems,* 27(6), pp. 1700-1712 (2016). IMR deploys a set of multiple rings such that each ring is to be shared by a specific set of cores. Also, packets are not allowed to switch between rings. However, the ring set is generated by an evolutionary algorithm, which takes a long time to produce a good ring set. Further, such evolutionary-generated ring sets are prone to producing large rings that affect packet latency, hop count, and power consumption. In addition, the design proposes require a large set of buffers to assure deadlock avoidance.

[0044] Buffers can also be helpful to an NoC design. Although buffers contribute negatively on area and power resources, they can help eliminate many issues such as deadlock. Recently, a bufferless technique was introduced to reduce power consumption. See, e.g., Fallin et al., "CHIPPER: A low-complexity bufferless deflection router," *IEEE 17th International Symposium on High Performance Computer Architecture*, pp. 144-155 (February 2011). Bufferless designs, however, suffer from numerous disadvantages. For example, bufferless designs suffer from livelock, deflection, and packet reassembly issues. The designs, however, do realize some gains in the savings of power consumption and area. Even though buffers consume some amounts of power, other routers components also consume significant amounts of power (e.g. 45%) regardless if buffers exist or not. See, e.g., Chen et al., "Nord: Node-router decoupling for effective power-gating of on-chip routers," *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 270-281 (December 2012).

[0045] Therefore, by eliminating routing components and reducing buffer size, an NoC design can realize a substantial reduction to the consumption of static power. Embodiments of the disclosed technology realize both of these objectives. For instance, certain embodiments of the disclosed technology reduce buffer size by allowing different components to share buffers. Further, by reducing or eliminating routers, a significant savings in power consumption and area can be gained. However, the challenge is how to link cores in the NoC without losing the performance advantages and path flexibilities of a router while still providing a scalable approach.

[0046] As noted, there is an increasing trend in the number of available wires in advanced many-core processors. Therefore, one could speed up data transfer from one component to another by utilizing more wires. However, linking a component with more wires increases the component's size and power consumption. For example, if two routers are connected by 256 bit links, then doubling the number of wires on a link would, at least, double the crossbar size for each router. Thus, adding more wires to link routers is not an efficient approach.

[0047] Further, conventional NoC designs have limited capabilities to use the large amount of wires due to the power and area requirements of routers with wider ports. For example, a 4×4 mesh with 64 bit flit size needs a die of around 2.16 mm×2.36 mm in a 45 nm technology node. See Park et al., "Approaching the theoretical limits of a mesh NoC with a 16-node chip prototype in 45 nm SOI," *Proceedings of the 49th Annual Design Automation Conference*, pp. 398-405 (June 2012). For this die size and based on the pitch size listed in Table 1 for a 45 nm technology node, two metal layers (M4 and M5) can provide 8357 wires in cross-section (4500=2.16 mm/(2×240 nm) wires in metal layer 4, pitch size is doubled in order to take into account the area overhead of shielding technique, and 3857=2.16 mm/(2×280) nm wires in metal layer 5). However, a 4×4 mesh with a 64 bit flit size can only use (4×64) 256 wires in the cross-section. This means that the mesh NoC design can only utilize 4% of all available wire resources in only two metal layers. On the other hand, increasing the flit size to enhance the wire resource utilization results in more buffers and a larger crossbar in routers, resulting in higher power consumption and area overhead. Therefore, new NoC designs that can better utilize the large amount of available wires with low power and area are highly desirable for current and future multi-core processors.

[0048] Embodiments of the disclosed technology implement a wired-based NoC by forming multiple unidirectional circular paths, also referred to as "circles". These circles are wiring loops that connect two or more nodes to one another. In particular implementations, every pair of nodes on the NoC shares at least one circle. Using wires, one can form a number of circles such that routers are no longer required, thus facilitating a routerless NoC.

[0049] It should be noted that a "circle" as discussed herein is not strictly circular in shape, but rather traverses a set of two or more nodes along a path that forms a closed loop and is therefore circular in nature. Typically, though not necessarily, the nodes traversed by a circle are arranged (or, as part of the circuit design process, considered to be arranged) in an arrangement of columns and rows. For this reason, the disclosure will sometimes refer to a particularly numbered row or column (or a "lowest" or "highest" row or column), which references a row numbered consecutively from top to bottom (or, equivalently, bottom to top) or a column numbered consecutively from left to right (or, equivalently, right to left). This arrangement is typically described as an n×n mesh topology, where n is a positive integer, or an m×n mesh topology, where m and/or n are positive integers. For ease of illustration, such arrangements are shown herein rectilinearly. The actual arrangement or final physical layout, however, need not be strictly rectilin-

ear. Instead, the references to "columns", "rows", and "mesh topologies" discussed herein encompass equivalent logical relationships between nodes (or cores). Further, it should be understood that the terms "rows" and "columns" encompass the equivalent conversion to "columns" and "rows".

[0050] Theoretically speaking, the number of circles on a n×n mesh topology is abundant and grows rapidly with n. For example, in a 4×4 topology of nodes, the total number of possible circles is 426. FIGS. **1**A and **1**B are schematic block diagrams of a 4×4 topology **100**, **102** of nodes in which several example circles (**110**, **112**, **114**, **116**) are depicted. Further, in FIGS. **1**A and **1**B as well as the other figures depicting n×n mesh topologies, each node of the mesh represents a connection point of a core of the many-core processor design to the NoC. Moreover, for bigger meshes, the number of circles are numerous. In particular, Table 2 shows the total number of circles for various mesh sizes.

TABLE 2

n	# of directed rings in n × n mesh
2	2
3	26
4	426
5	18698
6	2444726
7	974300742
8	1207683297862

[0051] For an 8×8 mesh topology, there are more than 10^{13} circles, not to mention combinations. While it is easy to connect all the nodes with lengthy circles, such as Hamiltonian circles (an example of which is shown as circle **116** in FIG. **1**B), such circles greatly increase the average hop count and, as a result, are not recommended for NoCs.

[0052] To generate a set of circles that minimizes or otherwise significantly reduces the hop count is very challenging due to large number of choices. Embodiments of the disclosed technology are directed to tools and techniques for addressing these difficulties by generating circles using deterministic techniques that also significantly reduce the average hop count set of circles. Particular implementations use a fast, recursive algorithm for determining the circles.

IV. Example Approaches to Routerless NOC Design

[0053] This section presents examples of a routerless ("RL") design for an n×n mesh topology. Embodiments of the disclosed approach utilize the abundant wiring resource by intelligently placing a set of circles that interconnects nodes on the mesh. Further, in some implementations, the set of circles is generated by a fast and recursive algorithm. In particular implementations, the circles (e.g., all circles) are unidirectional, have a rectangular shape, and have the same width of wires. As a result of the circles, the paths from a source to a destination are predefined; consequently, once a packet is pushed to a downstream circle, it remains on the same circle until the destination ejects it. Hence, the role of a router is no longer required and can therefore be eliminated.

[0054] Further, in some embodiments, a node (e.g., each node) on the mesh is connected to its corresponding core (e.g., a processing core of a multi-core processor) with a network interface to access the NoC such that all circles that contain the node pass through the interface. Although the wiring resources are large, attaching too many wires to an interface would increase the interface's buffer and, eventually, the demand for the power and area becomes undesirable. Therefore, for the purpose of controlling the power and area requirements, certain embodiments limit the maximum number of circles overlapping at any link to n, where n is the dimension size of the mesh topology.

[0055] In the next subsections, several example embodiments are presented to show how circles are generated followed by a formal description of an illustrative non-limiting algorithm that recursively generates the circles. Details of example network interfaces and hardware implementations are then introduced and discussed.

[0056] A. Circle Generation Examples

[0057] In this subsection, example techniques for generating circles for 2×2, 4×4, 6×6, and 8×8 mesh topologies are described. In general, circles generated for an n×n mesh are denoted by M_n.

[0058] In accordance with one exemplary technique, the mesh is split into layers. FIG. **2** is a schematic block diagram **200** illustrating the layers for an 8×8 mesh, where each layer is labeled. As shown in FIG. **2**, Layer **1** is a 2×2 mesh, Layers **1** & **2** form a 4×4 mesh, Layers **1** & **2** & **3** combined result in a 6×6 mesh, and finally all layers form the original 8×8 mesh. Due to the structure of layers, circles generated for an n−2×n−2 mesh can be a subset of circles generated for n×n mesh topology. Furthermore, FIG. **2** illustrates that the layers are concentric in nature. For purposes of this disclosure, let M_n denote the set of circles on an n×n mesh and L_i be the set of circles generated specifically for Layer i. More details are discussed below in the examples.

[0059] i. 2×2 Mesh Topology

[0060] This is the basic case. It has one layer and two circles, as shown in FIGS. **3**A-**3**C. More specifically, FIG. **3**A is a schematic block diagram of the overall 2×2 mesh topology **300**; FIG. **3**B illustrates clockwise circle **310** on the topology; and FIG. **3**C illustrates counterclockwise circle **320**. Both circles are included in $M_2=L_1$. Notice that, with M_2, the mesh is interconnected and the maximum number of circles overlapping at any link is 2. Also, the average hop count is 0.333.

[0061] ii. 4×4 Mesh Topology

[0062] In the case of the 4×4 mesh topology, the mesh has two layers. For example, FIG. **4**A is a schematic block diagram of the overall 4×4 mesh topology **400** and shows that the topology has 2 layers: Layer **1** and Layer **2**. For Layer **2**, the set of circles L_2 is generated and depicted by circles **410**, **412**, **414**, **416**, **418**, **420**, **422**, **424** in FIGS. **4**B-**4**I. Notice that the two circles **410**, **414** in FIGS. **4**B and **4**D connect nodes (which represent respective cores and their respective interfaces) on Layer **2** with all nodes on Column **2**. Similarly, nodes on Column **3** are connected to Layer **2** nodes by the circles **412**, **416** shown in FIGS. **4**C and **4**E. Then, the nodes on Row **1** are connected to Row **2** nodes by the circle **418**, the nodes on Row **2** are connected to Row **3** by the circle **420**, and the nodes Row **3** are connected to Row **4** by the circle **422**. Finally, the perimeter nodes are connected by the largest circle **424**, which also is oriented in the opposite direction of circles **410**, **412**, **414**, **416**, **418**, **420**, **422** (clockwise vs. counterclockwise or vice versa depending on the directionality of the circles **410**, **412**, **414**, **416**, **418**, **420**, **422**). Therefore, circles in L_2 connect nodes on Layer **2** with every other node in the mesh.

Moreover, in this illustrated embodiment, exactly four circles are overlapping at every link (which corresponds to a set of wires linking components (such as nodes) to one another) on Layer **2** and, in like manner, every other link (not on Layer **2**) is overlapped by two circles. Links on Layer **2** can no longer allow more circles to overlap as the maximum is four, whereas links on Layer **1** have room for two more circles. Therefore, M_2 can be used to interconnect all nodes on Layer **1** (as Layer **1** is a 2×2 mesh) with at most two circles overlapping at any link. The final set is $M_4=L_2 \cup M_2=L_2 \cup L_1$ where the average hop count is 2.93.

[0063] iii. 6×6 Mesh Topology

[0064] For the 6×6 topology, there are 3 layers in this mesh. In a similar fashion as above, circles in L_3 are generated for Layer **3** as illustrated in FIGS. **5**A-D. For ease of illustration, FIG. **5** depicts related circles together in composite images. FIG. **5**A, for example, shows a largest circle **510** traversing the perimeter nodes along one direction (either clockwise or counterclockwise). FIG. **5**B shows the circles **512** that extend from column **2** incrementally toward column **5**. In the illustrated embodiment, the circles **512** are oriented in the opposite direction of circle **510**. FIG. **5**C shows the circles **514** that extend from column **5** decrementally toward column **2**. In the illustrated embodiment, the circles **514** are oriented in the opposite direction of circle **510**. FIG. **5**D shows the circles **516** that form incremental circles for adjacent pairs of rows and that are also oriented in the opposite direction of circle **510**. Again, all nodes on Layer **3** are connected to every other node in the mesh by circles in L_3. Links on Layer **3** are overlapped by six circles and all other links are overlapped by two circles. Therefore, M_4 can be used to interconnect all nodes on layers **2** and **1**. In certain embodiments, for the purpose of improving the average hop count, every circle in M_4 is reversed and rotated 90°. This new set is denoted by M'_4. As a result, $M_6=L_3 \cup M'_4=L_3 \cup (L_2 \cup L_1)'$ with an average hop count of 5.07. Note that, in the previous example, it was not necessary to reverse and rotate the circles in M_2 because $M_2=M'_2$.

[0065] iv. 8×8 Mesh Topology

[0066] Similar to the earlier examples, the circles can be generated using the outer layer (Layer **4**) to generate L_4. For ease of illustration, FIG. **6** depicts related circles together in composite images. FIG. **6**A, for example, shows a largest circle **610** traversing the perimeter nodes along one direction (either clockwise or counterclockwise). FIG. **6**B shows the circles **612** that extend from column **2** incrementally toward column **7**. The circles **612** can be oriented in the opposite direction of circle **610**. FIG. **6**C shows the circles **614** that extend from column **7** decrementally toward column **2**. The circles **614** can be oriented in the opposite direction of circle **610**. FIG. **6**D shows the circles **616** that form incremental circles for adjacent pairs of rows and that are also oriented in the opposite direction of circle **610**. Then, using earlier results, M_8 can be defined as:

$$M_8=L_4 \cup M'_6=L_4 \cup (L_3 \cup M'_4)'=L_4 \cup (L_3 \cup (L_2 \cup L_1)')'$$

where the average hop count is 7.32.

[0067] B. Formal Description

[0068] For an n×n mesh topology, circles for the routerless design can be recursively generated. One example of such an algorithm (termed "RLrec") is illustrated in example pseudocode **700** shown in FIG. **7**. In the illustrated example, the algorithm begins by generating circles for the outer layer, say layer i, and then recursively generating circles for layer i−1 and so on until the basic case (Layer **1**) is reached or the layer has a single node.

[0069] The example algorithm takes two integers L and H as input parameters with initial values 1 and n, respectively. L denotes the lowest row/column and H denotes the highest row/column. Using L and H, the number of layers is

$$x=\left\lceil \frac{H-L+1}{2} \right\rceil = \lceil \frac{n}{2} \rceil.$$

In subsequent recursive calls, RLrec is called with L=L+1 and H=H−1 to work on the next layer. The first set of circles generated by RLrec is for the boarder layer (layer x). The example algorithm begins with

$$C(L, H, L, H, \text{anticlockwise}) \quad (1)$$

which overlaps with layer x. The procedure C (r_1, r_2, c_1, c_2, d) draws two lines on rows r_1, r_2 and two lines on columns c_1, c_2 and generates a circle with direction d from the resultant rectangular shape. FIG. **6**A shows an example result for C (0,7,0,7, anticlockwise). Moreover, the next generated circles are

$$C(L, H, L, i, \text{clockwise}) \,\&\, C(L, H, i, H, \text{clockwise}) \quad (2)$$

where L+1≤i≤H−1 and correspond to the circles in FIGS. **6**B and **6**C. The pair of circles C(L, H, L, i, clockwise) & C(L, H, i, H, clockwise) overlaps with layer x and column i. Notice that, every column, other than L, H, is overlapped by two circles only. The circles generated so far allow every node on layer x share at least one circle with every column. In other words, each node on layer x shares at least a circle with every other node on the mesh. Also, every link on layer x is overlapped by

$$\underbrace{1}_{from(1)} + \underbrace{H-1-(L+1)+1}_{from(2)} = H-L = n-1$$

circles and every link on columns L+1 . . . , H−1 is overlapped by exactly two circles. There are n−2 pairs of circles generated by (2). Observe that, rows L+1, L+2, . . . , H−1 are not yet utilized by any circle. The last set of circles generated is

$$C(i,i+1, L, H, \text{clockwise}) \quad (3)$$

where L≤i≤H−1. This set is similar to circles in FIG. **6**D. The two circles C(i, i+1, L, H, clockwise) & C(i+1, i+2, L, H, clockwise) overlap at row i+1 only and all circles in this set use each link on layer x only once. Therefore, links on layer x are now overlapped by n circles and every other link is overlapped by two circles. Finally, the algorithm recursively calls RLrec with L=L+1 and H=H−1 and then reverses and rotates for 90°. Calling RLrec with L=L+1 and H=H−1 will ignore layer x and generate circles for the outer layer of an n−2×n−2 mesh (layer x−1). Notice that, initially L=1 and H=n and therefore, H−1−(L+1)+1=n−2 (the dimension size of the mesh.)

[0070] The number of circles generated by RLrec can be easily calculated due to its simplicity and recursive nature by the following recurrence function:

$$F(n) = \underbrace{1}_{\text{from(1)}} + \underbrace{2\times(n-2)}_{\text{from(2)}} + \underbrace{n-1}_{\text{from(3)}} + F(n-2),$$

where F(2)=2 and F(1)=F(0)=0.

[0071] C. Example Network Interfaces

[0072] In embodiments of the disclosed technology, a node in the NoC is part of many circles where such circles

allow the node to communicate with every other node in the NoC. In other words, the circles avail the node with predefined paths to all destinations. In particular embodiments, a node communicates with the circles through a network interface.

[0073] FIG. **8** is a schematic block diagram **800** of an example network interface with exemplary components. With reference to FIG. **8**, each circle is represented by an input port (shown by representative input ports **810**, **811** for circles i and circle j, respectively), an output port (shown by representative output ports **812**, **813** for circles i and circle j, respectively), and a small buffer of $B_c \geq 1$ flits (shown by representative buffers **820**, **821** for circles i and circle j, respectively). The flit size is fixed and the smallest packet in the NoC has B_c flits (e.g., the circle's buffer can accommodate the smallest packet). The network interface also includes injection port **826** and one or more ejection ports **827** for receiving packets from the associated core and outputting packets to the associated core, respectively. In addition, in certain embodiments, the network interface comprises a pre-calculated table **830** (e.g., a look-up table) that maps destinations with the circles to reach those destinations. Moreover, a pool of one or more buffers **840** ("expansion buffers") is available to allow a circle's buffer to be expanded. Each buffer in this pool is of $B_{EXB} \geq 1$ flits and can be used by at most one circle at a time and is denoted by extension buffer (EXB). Moreover, the largest packet in the NoC has at most B_c+B_{EXB} flits. Notice that, in the illustrated embodiment, each circle is completely isolated from the others. Therefore, if a packet is injected into a circle, it will remain in the same circle until ejection.

[0074] In the following paragraphs, an example of how the interface operates is described.

[0075] When a packet p of n flits is injected into the interface, the available circles that can reach the packet's destination are identified. In particular example embodiments, a circle is considered available if one of the following conditions holds:

[0076] 1. The buffer is empty and either: (a) $n \leq B_c$; or (b) $n \geq B_c$ and an EXB is available; or

[0077] 2. The buffer is not empty and has $x \geq 0$ free flit slots and the destination of the head flit is the current node and either: (a) $n \leq x$; (b) the buffer is extended by an EXB and has at least n free slots in total; or (c) the buffer is not extend by an EXB, $n \geq B_c$, $n \leq B_{EXB}+x$, and an EXB is available.

[0078] Once the list of available circles is known, the shortest circle to the destination, say to inject p is selected. While the interface is injecting the head flit of p, it will attach an EXB to ζ if $n \geq B_c$ and ζ is not already attached to an EXB. Moreover, the output port of ζ will be busy injecting the n flits of p for n clock cycles. During those n clock cycles, any incoming flit through input port of ζ is enqueued in the buffer. Notice that ζ has sufficient space to accept n flits while p is being injected. If no circle is available, p will be blocked or stalled at injection port and will try again in the next cycle.

[0079] One of the reasons to consider a circle as not available for injection is if the destination of the head flit in its buffer is not the current node. In this case, and in particular embodiments, the circle forwards the head flit through the output port directly regardless of the interface or the NoC state. The only case to stall such a flit is if the output port of the circle is in use for injection prior to the flit's arrival to the head of the circle's buffer. Moreover, if a circle's buffer is extended by an EXB, then this EXB will be tied with the circle until the circle's buffer and the EXB are both empty. In this case, the EXB will be freed and returned to the pool.

[0080] The discussion above concerns how an interface can inject, stall, and forward a packet. The remaining action on a packet is ejection. In particular example embodiments, the ejection process can begin as soon as the head flit of a packet p becomes the head flit of a circle's buffer in the destination's interface. In the same cycle, the interface begins ejecting flits p, one flit per cycle. Once p is fully ejected, the interface will wait for another packet to eject. If multiple packets reach the head of multiple circle's buffers, all such packets will compete for the ejection link and the oldest packet will win (though other prioritizations are also possible, including the packet being associated with a flag or other indication of a highest priority packet). Packets that lost their chance for ejection will be stalled and try again in the next cycle as long as their circle's buffer is not full. In the case of a full buffer, the packet is forwarded to the next interface. If the interface has multiple ejection ports, then it can eject in parallel m packets, where m is less than or equal to the number of circles in the interface. If more than m packets are competing to eject, and in certain embodiments, the oldest m packets wins (though other prioritizations are also possible, including the packet being associated with a flag or other indication of a highest priority packet).

[0081] Notice that in the example embodiments described above, the described operational actions do not depend on the state of a neighboring interface nor the NoC. In addition, the interface is always welcoming incoming flits from its neighbors. Allowing the interface to always accept a flit helps the NoC to avoid deadlocks as described in the next subsection.

[0082] i. Deadlocks

[0083] Deadlocks occur when each member of a group is holding a resource and each member is waiting for another resource, held by another member, to complete its task. Examples of the disclosed network design are not credit based; instead, each interface works solely by itself, and each of its ports can accept an incoming flit regardless of its state or NoC state. Moreover, the isolation of circles eliminates many problems introduced by "head-of-line" blocking. FIG. **9** shows schematic block diagrams **900**, **902**, **904**, **906** showing how flits can loop in a circle for four incremental clock cycles in accordance with one example embodiment. In every cycle, flits are forwarded to the next interface, and it is the interface's responsibility to assure space availability for incoming flits without any prior information. FIG. **10** shows schematic block diagrams **1000**, **1002**, **1004**, **1006** showing how a packet of three flits can be injected into an example embodiment of the interface as described above. In FIG. **10**, notice that flits A, B, C are blocked in the middle interface **1020** until the injection process is completed.

V. Evaluation Methodology

[0084] Embodiments of the disclosed technology were extensively evaluated using Booksim, a cycle-accurate simulator, for synthetic traffic. See Jiang et al., "A detailed and flexible cycle-accurate network-on-chip simulator," 2013 *IEEE International Symposium on Performance Analysis of Systems and Software* (*ISPASS*), pp. 86-96 (April 2013). Although synthetic workloads may be practical, they do not capture the essence and actual behavior of real-world

140

applications. To help simulate real-world applications, the Synfull application was used. Synfull has a synthetic traffic generation methodology that better reflects an actual application's behaviors. See Badr et al., "SynFull: synthetic traffic models capturing cache coherent behavior," 2014 *ACM/IEEE 41st International Symposium on Computer Architecture* (*ISCA*), pp. 109-120 (June 2014). Synfull uses a variety of PARSEC and SPLASH-2 benchmarks and is based on 16-core multi-threaded applications. See Bienia, "Benchmarking modern multiprocessors," New York: Princeton University, Ph.D. Thesis (January 2011); Woo et al., "The SPLASH-2 programs: Characterization and methodological considerations," *ACM SIGARCH Computer Architecture News*, vol. 23, no. 2, pp. 24-36 (July 1995). Synfull also integrates well with Booksim. For the tests disclosed herein, Synfull and Booksim are used to evaluate power and area.

[0085] For the purpose of evaluation, an exemplary routerless ("RL") design as described herein (comprising circles generated according to the technique illustrated in FIG. **7** and having a network as in FIG. **8**) was compared against three NoC designs, including a traditional mesh design (denoted as mesh), EVC design (as described in Kumar et al., "Express virtual channels: towards the ideal interconnection fabric," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 2, pp. 150-161 (2007), and an IMR design.

[0086] The configuration details for each design are as follows. For the mesh design, the router was configured with two virtual channels (VCs) per link with three flit buffers per VC. Also, the router's latency was minimized by setting look-ahead routing with speculative switch allocation, and setting the number of cycles for each pipeline stage to one. Such a configuration allowed for the optimization of router performance and made it very competitive with the least amount of area for buffers. Furthermore, the EVC design had the same configuration as the mesh design except that one extra VC was dedicated to implement the express channels.

[0087] The IMR design was implemented in Booksim. The interface had one injection link and one ejection link and every other link was equipped with a full packet-size buffer. Additionally, the per-hop latency for every interface was one clock cycle. The maximum number of links between a pair of nodes was 16. In this regard, the limit was set to n to allow a fair comparison of the RL design with IMR.

[0088] For the RL design, the network interface was implemented such that every link had one flit-size buffer and two extension full packet-size buffers for injection purposes and two ejection links. After several experiments, it was found that having one ejector degraded performance, but having two ejectors greatly enhanced performance For more than two ejectors, the performance gain was negligible. Similarly, with one extension buffer, the injection link suffered from being blocked for a long time and hence performance was affected negatively. With two extension buffers, significant performance improvements were realized. Desirably, the design has as many extension buffers as the input ports, but this may affect interface area and static power negatively. Finally, the circles were generated by the technique illustrated in FIG. **7**.

[0089] The width of each link for the mesh and EVC designs is 256 bits, while for the RL and IMR designs it is 128 bits. From the link width, the maximum flits per packet can be determined. The control packet is of 64 bit and the data packet is of 576 bit. Therefore, the control packet was a one flit packet for all NoCs, while the data packet in the mesh and EVC designs was a three flits packet, and for the RL and IMR designs it is a five flits packet.

[0090] A. Synthetic

[0091] In this section, results are described from simulating the above NoCs in Booksim using four synthetic patterns (uniform random, transpose, bit reverse, and hotspots for 8 hotspot nodes) on an 8×8 mesh topology. In each test, the simulator ran for 100,000 cycles to collect latency and throughput statistics at different injection rates. The ratio of long and short packets was 2 to 8. The initial injection rate for all the runs is 0.005 and it was incremented by 0.005 until reaching the throughput.

[0092] The latency and throughput detail are shown in plots **1100**, **1102**, **1104**, and **1106** of FIGS. **11**A-D for a variety of injection rates for each traffic pattern. Notice that the RL design performed normally under all injection rates in all traffic patterns. This gives an indication that the RL design is a deadlock-free approach with flit size buffers at input links and few extension buffers. Moreover, the RL design outperformed all other NoCs in latency and throughput. The average packet latency in uniform traffic pattern was 21.2, 14.9, 10.5, and 8.3 cycles for the mesh, EVC, IMR, and RL designs, respectively. Other traffic patterns had similar trends. Therefore, the RL design outperformed all other NoCs in latency and provided an average improvement of 59%, 43%, and 25% over the mesh, EVC, and IMR designs, respectively.

[0093] Regarding throughput, a similar trend was observed. For example, the throughput for the hotspot pattern was 0.08, 0.05, 0.06, and 0.125 (per flit/node/cycle) for the IMR, RL, mesh, and EVC nodes, respectively. It was also observed that the RL design had the highest throughput and performed very well for the bit reverse, hotspot, and transpose patterns. The RL design, on average, enhanced the throughput by 94%, 187%, and 68% over the mesh, EVC, and IMR designs, respectively.

[0094] B. PARSEC and SPLASH-2

[0095] Using Synfull, all benchmark traffic patterns were generated and the latency was evaluated for over 500,000 cycle for 4×4, 8×8, and 16×16 mesh topologies. For 8×8 topologies, Booksim was interfaced with 4 Synfull processes where each Synfull process is mapped randomly to 16 nodes such that a node is mapped to only one Synfull process. Similarly, for 16×16 topologies, Booksim was interfaced with 16 Synfull processes and, again, each Synfull process is mapped randomly to 16 nodes.

[0096] FIGS. **12**A-C shows plots **1200**, **1202**, and **1204** of the PARSEC and SPLASH-2 benchmarks evaluated by Synfull for the RL, IMR, conventional mesh, and EVC designs. As shown by the plots **1200**, **1202**, **1204**, the RL design provided better performance over all other NoCs in all topologies and all the benchmarks. For example, for a 16×16 topology, the average latency for the RL design was 20.1 flits/node/cycle, whereas the average latency for the mesh, EVC, and IMR designs was 37.01, 26.2, 34.2 flits/node/cycle, respectively. In general, the latency of the RL design was only 52%, 35%, and 34% of the latency of the mesh, EVC, and IMR designs, respectively. Further, the average number of cycles where both extension buffers were free is more than 99% of the total number of cycles. This

gives the potential to neglect one of the extension buffer to further reduce the area of RL network interface.

[0097] C. Power and Area

[0098] To evaluate the power and area of the RL design, a Verilog version of an example RL interface as disclosed herein and shown in FIG. **8** was implemented, and the functionality of the interface was verified using extensive Modelsim simulations. For comparison, Verilog versions of other routers and interfaces were also implemented. For the mesh design, a parametrized RTL implementation provided by Hoskote et al., "A 5-GHz Mesh Interconnect for a Teraflops Processor," *IEEE Micro*, vol. 27, no. 5, pp. 51-61 (September-October 2007) was used. Additionally, Synopsys' Design Compiler and Cadence's Encounter tools were used for synthesis, and place and route was implemented using the NanGate FreePDK 15 Cell Library (see NanGate, Inc, "Nangate freePDK15 open cell library").

[0099] FIG. **13** is a plot **1300** showing the breakdown of the routers' and interfaces' power consumption across the benchmarks and normalized to the mesh design. All power consumption shown in the plots is reported after place and route using the NanGate FreePDK 15 Cell Library by Cadence's Encounter. Activity factors for power measurements were obtained from the extensive simulation results. The power measurements were decomposed into dynamic power and static power. The dynamic power includes internal power and switching power. The internal power is consumed because of short circuit power while devices are changing their state. On the other hand, the switching power is consumed due to charging and discharging capacitive output loads. Compared to the mesh design, and on the average, the EVC design consumes 38% more power, and the IMR design has 20% lower power consumption. The RL design reduces the power consumption across the benchmarks by nearly 90%. This tremendous power saving is mainly due to the structure of the interface, which removes power hungry components, such as crossbars, and reduces the number of buffers.

[0100] The reported area for the mesh design was 45281 μm^2 and for EVC design was 60731 μm^2. The reported area for the IMR design was 20930 μm^2, and the area for the RL design was 6286 μm^2. All the reported areas are core area and they are reported by Cadence's Encounter after place and route using the NanGate FreePDK 15 Cell Library.

[0101] FIG. **14** is an image **1400** showing the layout of the routers and the interfaces for the various tested designs. The layouts were generated by Cadence's Encounter after place and route using the NanGate FreePDK 15 Cell Library. As shown in FIG. **14**, the mesh design has a high area requirement when compared to the IMR interface and bufferless router. Using the IMR design, one can save about 53% while the bufferless router reduces the area by 37% in comparison with the mesh area. Furthermore, the RL interface can improve the area even more than bufferless. That is, the RL interface require less than 86% area than the mesh design. Therefore, the RL design allow to save a significant amount of silicon area for processing and storage units.

[0102] D. Additional Observations

[0103] The average hop count for synthetic traffic patterns was also evaluated. FIGS. **15**A-D are plots **1500**, **1502**, **1504**, **1506** showing the average hop count for the RL, IMR, and optimal Mesh designs for several traffic patterns. From these plots, it can be observed that the IMR design has a high average hop count because rings produced by their approach have a long length. Notably, the RL design has only a slightly higher average hop count than the optimal mesh.

[0104] NoCs will continue to accommodate more and more processing cores. Certain designs, for example, already propose 1000 cores. Therefore, scalability is vital for any NoC approach to help further support such high-core designs. As shown and described above, NoC designs in accordance with the disclosed technology are very effective in terms of latency and throughput for higher dimensional mesh topologies. Moreover, the power consumption and area occupancy exhibited by embodiments of the disclosed routerless network interfaces are extremely low when compared to conventional mesh routers or IMR designs, even for high mesh topologies.

VI. Overview of Example Embodiments

[0105] FIG. **16** is a flow chart **1600** showing a generalized example embodiment for implementing an NoC generation technique according to the disclosed technology. The particular operations and sequence of operations should not be construed as limiting, as they can be performed alone or in any combination, subcombination, and/or sequence with one another. Additionally, the illustrated operations can be performed together with one or more other operations. Still further, the identified operations need not be performed by a single software module, but can be implemented using multiple modules or software tools, which collectively perform the illustrated method. The example embodiment of FIG. **16** can be performed, for example, by one or more specialized electronic design automation (EDA) tools that are adapted to perform the disclosed circuit design techniques (e.g., an EDA design tool for generating a hardware description of a network-on-chip configured to interconnect a plurality of processing cores of a mutli-core processor). Such tool(s) can be used, for instance, as part of a circuit design and/or manufacturing process and be implemented by one or more computing devices as described above. The example embodiments described with respect to or related to FIG. **16** can be used to realize any one or more of the benefits disclosed herein.

[0106] At **1610**, parameters that describe a size of a topological mesh of nodes are input (e.g., buffered into memory or otherwise prepared for further processing). In this embodiment, the nodes of the mesh correspond to processing cores of the mutli-core processor.

[0107] At **1612**, a wiring layout is generated for the network-on-chip by applying a path generation procedure that generates circular wiring paths connecting processing cores of a first layer of the topological mesh, and that is then recursively applied to generate circular wiring paths connecting processing cores for one or more additional layers of the topological mesh.

[0108] At **1614**, the wiring layout is output (e.g., stored in an output file that can then be used by one or more downstream tools in the design and manufacturing flow of a multi-core processor). For instance, the wiring layout can be a hardware description, such as a hardware-design-language representation or a gate-level netlist.

[0109] In particular embodiments, the one or more additional layers of the topological mesh are concentric and interior to the first layer. In certain embodiments, the path generation procedure generates the circular wiring paths such that every pair of nodes of the topologic mesh share at least one circular wiring path. In some embodiments, the

path generation procedure generates the circular wiring paths for a respective layer of the topological mesh in a deterministic fashion that minimizes hop count among the circular wiring paths.

[0110] FIG. **17** is a flow chart **1700** showing a generalized example embodiment for implementing an NoC generation technique according to the disclosed technology. The particular operations and sequence of operations should not be construed as limiting, as they can be performed alone or in any combination, subcombination, and/or sequence with one another. Additionally, the illustrated operations can be performed together with one or more other operations. Still further, the identified operations need not be performed by a single software module, but can be implemented using multiple modules or software tools, which collectively perform the illustrated method. The example embodiment of FIG. **17** can be performed, for example, by one or more specialized electronic design automation (EDA) tools that are adapted to perform the disclosed circuit design techniques (e.g., an EDA design tool for generating a hardware description of a network-on-chip configured to interconnect a plurality of processing cores of a mutli-core processor). Such tool(s) can be used, for instance, as part of a circuit design and/or manufacturing process and be implemented by one or more computing devices as described above. The example embodiments described with respect to or related to FIG. **17** can be used to realize any one or more of the benefits disclosed herein.

[0111] At **1710**, design data is generated specifying wiring paths of a routerless network-on-chip configured to interconnect the multiple processing cores with one another. In some embodiments, the wiring paths are generated using a deterministic wiring path selection procedure.

[0112] At **1712**, design data is generated for network interfaces of the network-on-chip, the network interfaces facilitating injection of a network packet into the network-on-chip from a source processing core and ejection of the network packet from the network-on-chip at a destination core.

[0113] In particular embodiments, the one or more of the network interfaces comprise extension buffers that are shared among multiple input ports of the one or more network interfaces. In some embodiments, the wiring paths comprise a set of unidirectional wiring loops arranged so that every pair of processing cores shares at least one of the unidirectional wiring loops. In certain embodiments, the wiring paths have a rectangular shape.

[0114] Further disclosed embodiments comprise integrated circuits implementing any of the disclosed technologies. For instance, embodiments of the disclosed technology are chip multiprocessors (or multi-core processors). These particular embodiments should not be construed as limiting, as they can include any combination, subcombination, and/or combination of features as disclosed herein.

[0115] Particular embodiments include an integrated circuit, comprising: a plurality of processing cores; and a network-on-chip subsystem configured to interconnect the cores via a set of deterministically specified wiring circles. In certain embodiments, the network-on-chip subsystem is router-and-crossbar-free. In some embodiments, the plurality of cores are arranged in an n×n mesh topology having n columns and n rows, and the wiring circles include a first set of n−2 circles and a second set n−2 circles such that either a. every column other than the lowest and highest column is overlapped by only two circles of the first set and the second set; or b. every row other than the lowest and highest row is overlapped by only two circles of the first set and the second set. In certain embodiments, the plurality of cores are arranged in an m×n mesh topology having m columns and n rows, and the wiring circles include a first set of circles that includes: (a) all nodes of the lowest column along a first side of the circles and all nodes from respective incrementally higher columns along a second side of the circles except for the highest column; (b) all nodes of the highest column along a first side of the circles and all nodes from respective incrementally lower columns along a second side of the circles except for the lowest column; b. all nodes of the lowest row along a first side of the circles and all nodes from respective incrementally higher rows along a second side of the circles except for the highest row; or c. all nodes of the highest row along a first side of the circles and all nodes from respective incrementally lower rows along a second side of the circles except for the lowest row. In certain embodiments, the plurality of cores are arranged in an m×n mesh topology having m columns and n rows, the wiring circles include a perimeter circle the includes all perimeter nodes of the mesh topology and is configured to propagate network data in a first direction, and the wiring circles include a plurality of interior circles that at least partially include nodes that are interior of the perimeter node and that are configured to propagate network data in a second direction opposite of the first direction. In some embodiments, the wiring circles further include a sub-layer perimeter circle that is concentric to the perimeter circle and that includes all nodes along a perimeter of a layer of the mesh topology that is interior to perimeter circle, and wherein the wiring circles include a plurality of sub-layer interior circles that are interior of the perimeter circle and that at least partially include nodes that are interior of the sub-layer perimeter circle. In certain embodiments, the network-on-chip subsystem further comprises a network interface configured to implement a deadlock-free protocol for injecting and ejecting flow control units to and from the network interface. In some embodiments, the network-on-chip subsystem further comprises a network interface for a respective core of the integrated circuit, the respective core having network access to a plurality of wiring circles connected to the network interface, the network interface further comprising an extension buffer that is shared among the plurality of wiring circles. In certain embodiments, the network-on-chip subsystem further comprises a network interface comprising one or more extension buffers configured to receive network packets from a neighboring core as packets are being simultaneously received by an injection port and output from an ejection port of the network interface.

VII. Example Computing Environments

[0116] FIG. **18** illustrates a generalized example of a suitable computer system **1800** in which the described innovations may be implemented. The example computer system **1800** can be a server or computer workstation (e.g., PC, laptop, tablet computer, mobile device, or the like) used by a design engineer during the design and production of a many-core processor.

[0117] With reference to FIG. **18**, the computer system **1800** includes one or more processing devices **1810**, **1815** and memory **1820**, **1825**. The processing devices **1810**, **1815** execute computer-executable instructions. A processing

device can be a general-purpose CPU, GPU, processor in an ASIC, FPGA, or any other type of processor. In a multi-processing system, multiple processing units execute computer-executable instructions to increase processing power. For example, FIG. **18** shows a CPU **1810** as well as a GPU or co-processing unit **1815**. The tangible memory **1820**, **1825**) may be volatile memory (e.g., registers, cache, RAM), non-volatile memory (e.g., ROM, EEPROM, flash memory, NVRAM, etc.), or some combination of the two, accessible by the processing device(s). The memory **1820**, **1825** stores software **1880** implementing one or more innovations described herein, in the form of computer-executable instructions suitable for execution by the processing device (s). The software **1880** can be, for example, an electronic design automation ("EDA") design tool. For instance, the EDA tool can be a behavioral synthesis tool configured to generate an HDL description of any of the disclosed NoCs or components (e.g., a Verilog, SystemVerilog, or VHDL description), a logic synthesis and/or place-and-route tool configured to generate a gate-level netlist (e.g., from an HDL description) for any of the of the disclosed NoCs or components, a physical synthesis tool configured to generate a geometric layout (e.g., a GDSII or Oasis file) that can be used to make a mask-level model form which masks can be printed and the final integrated circuit fabricated. The software **1880** can also comprise other suitable EDA tools for implementing the disclosed technology.

[0118] The computer system **1800** may have additional features. For example, the computer system **1800** includes storage **1840**, one or more input devices **1850**, one or more output devices **1860**, and one or more communication connections **1870**. An interconnection mechanism (not shown) such as a bus, controller, or network interconnects the components of the computer system **1800**. Typically, operating system software (not shown) provides an operating environment for other software executing in the computer system **1800**, and coordinates activities of the components of the computer system **1800**.

[0119] The tangible storage **1840** may be removable or non-removable, and includes magnetic disks, magnetic tapes or cassettes, optical storage media such as CD-ROMs or DVDs, or any other medium which can be used to store information and which can be accessed within the computer system **1800**. The storage **1840** stores instructions for the software **1880** implementing one or more innovations described herein.

[0120] The input device(s) **1850** may be a touch input device such as a keyboard, mouse, pen, or trackball, a voice input device, a scanning device, or another device that provides input to the computer system **1800**. For video or image input, the input device(s) **1850** may be a camera, video card, TV tuner card, screen capture module, or similar device that accepts video input in analog or digital form, or a CD-ROM or CD-RW that reads video input into the computer system **1800**. The output device(s) **1860** include a display device. The output device(s) may also include a printer, speaker, CD-writer, or another device that provides output from the computer system **1800**.

[0121] The communication connection(s) **1870** enable communication over a communication medium to another computing entity. For example, the communication connection(s) **1870** can connect the computer system **1800** to the internet and provide the functionality described herein. The communication medium conveys information such as computer-executable instructions, audio or video input or output, image data, or other data in a modulated data signal. A modulated data signal is a signal that has one or more of its characteristics set or changed in such a manner as to encode information in the signal. By way of example, and not limitation, communication media can use an electrical, optical, RF, or other carrier.

[0122] The innovations presented herein can be described in the general context of computer-readable media. Computer-readable media are any available tangible media that can be accessed within a computing environment. By way of example, and not limitation, with the computer system **1800**, computer-readable media include memory **1820**, **1825**, storage **1840**, and combinations of any of the above. As used herein, the term computer-readable media does not cover, encompass, or otherwise include carrier waves or signals per se.

[0123] The innovations can be described in the general context of computer-executable instructions, such as those included in program modules, being executed in a computer system on a target real or virtual processor. Generally, program modules include routines, programs, libraries, objects, classes, components, data structures, etc. that perform particular tasks or implement particular abstract data types. The functionality of the program modules may be combined or split between program modules as desired in various embodiments. Computer-executable instructions for program modules may be executed within a local or distributed computer system.

[0124] The terms "system" and "device" are used interchangeably herein. Unless the context clearly indicates otherwise, neither term implies any limitation on a type of computer system or computer device. In general, a computer system or computer device can be local or distributed, and can include any combination of special-purpose hardware and/or general-purpose hardware with software implementing the functionality described herein.

[0125] The disclosed methods can also be implemented using specialized computing hardware configured to perform any of the disclosed methods. For example, the disclosed methods can be implemented by an integrated circuit (e.g., an ASIC such as an ASIC digital signal processor ("DSP"), a GPU, or a programmable logic device ("PLD") such as a field programmable gate array ("FPGA")) specially designed or configured to implement any of the disclosed methods.

VIII. Concluding Remarks

[0126] In view of the many possible embodiments to which the principles of the disclosed invention may be applied, it should be recognized that the illustrated embodiments are only preferred examples of the invention and should not be taken as limiting the scope of the invention.

What is claimed is:

1. A computer-implemented method, comprising:

by a processor implementing an electronic design automation (EDA) design tool, generating a hardware description of a network-on-chip configured to interconnect a plurality of processing cores of a multi-core processor, wherein the generating comprises:

inputting parameters describing a size of a topological mesh of nodes, the nodes of the mesh corresponding to processing cores of the multi-core processor;

generating a wiring layout for the network-on-chip by applying a path generation procedure that generates circular wiring paths connecting processing cores of a first layer of the topological mesh and recursively applying the path generation procedure to generate circular wiring paths connecting processing cores for one or more additional layers of the topological mesh; and

outputting the wiring layout.

2. The method of claim **1**, wherein the one or more additional layers of the topological mesh are concentric and interior to the first layer.

3. The method of claim **1**, wherein the path generation procedure generates the circular wiring paths such that every pair of nodes of the topologic mesh share at least one circular wiring path.

4. The method of claim **1**, wherein the path generation procedure generates the circular wiring paths for a respective layer of the topological mesh in a deterministic fashion that minimizes hop count among the circular wiring paths.

5. The method of claim **1**, wherein the hardware description is a hardware-design-language representation or a gate-level netlist.

6. One or more memory or storage devices storing computer-executable instructions which when executed cause the computer to perform the method of claim **1**.

7. A system, comprising:

a memory or storage device; and

one or more processors, the one or more processors being configured to implement an electronic design automation (EDA) tool for generating an integrated circuit design comprising multiple processing cores;

generating design data specifying wiring paths of a routerless network-on-chip configured to interconnect the multiple processing cores with one another; and

generating design data for network interfaces of the network-on-chip, the network interfaces facilitating injection of a network packet into the network-on-chip from a source processing core and ejection of the network packet from the network-on-chip at a destination core.

8. The system of claim **7**, wherein one or more of the network interfaces comprise extension buffers that are shared among multiple input ports of the one or more network interfaces.

9. The system of claim **7**, wherein the wiring paths comprise a set of unidirectional wiring loops arranged so that every pair of processing cores shares at least one of the unidirectional wiring loops.

10. The system of claim **7**, wherein the generating comprises applying a deterministic wring path selection procedure.

11. The system of claim **7**, wherein the wiring paths all have a rectangular shape.

12. An integrated circuit, comprising:

a plurality of processing cores; and

a network-on-chip subsystem configured to interconnect the cores via a set of deterministically specified wiring circles.

13. The integrated circuit of claim **12**, wherein the network-on-chip subsystem is router-and-crossbar-free.

14. The integrated circuit of claim **12**, wherein the plurality of cores are arranged in an n×n mesh topology having n columns and n rows, and wherein the wiring circles include a first set of n−2 circles and a second set n−2 circles such that either a. every column other than the lowest and highest column is overlapped by only two circles of the first set and the second set; or b. every row other than the lowest and highest row is overlapped by only two circles of the first set and the second set.

15. The integrated circuit of claim **12**, wherein the plurality of cores are arranged in an m×n mesh topology having m columns and n rows, and

wherein the wiring circles include a first set of circles that includes:

a. all nodes of the lowest column along a first side of the circles and all nodes from respective incrementally higher columns along a second side of the circles except for the highest column;

b. all nodes of the highest column along a first side of the circles and all nodes from respective incrementally lower columns along a second side of the circles except for the lowest column;

c. all nodes of the lowest row along a first side of the circles and all nodes from respective incrementally higher rows along a second side of the circles except for the highest row; or

d. all nodes of the highest row along a first side of the circles and all nodes from respective incrementally lower rows along a second side of the circles except for the lowest row.

16. The integrated circuit of claim **12**, wherein the plurality of cores are arranged in an m×n mesh topology having m columns and n rows,

wherein the wiring circles include a perimeter circle the includes all perimeter nodes of the mesh topology and is configured to propagate network data in a first direction, and

wherein the wiring circles include a plurality of interior circles that at least partially include nodes that are interior of the perimeter node and that are configured to propagate network data in a second direction opposite of the first direction.

17. The integrated circuit of claim **12**, wherein the wiring circles further include a sub-layer perimeter circle that is concentric to the perimeter circle and that includes all nodes along a perimeter of a layer of the mesh topology that is interior to perimeter circle, and

wherein the wiring circles include a plurality of sub-layer interior circles that are interior of the perimeter circle and that at least partially include nodes that are interior of the sub-layer perimeter circle.

18. The integrated circuit of claim **12**, wherein the network-on-chip subsystem further comprises a network interface configured to implement a deadlock-free protocol for injecting and ejecting flow control units to and from the network interface.

19. The integrated circuit of claim **12**, wherein the network-on-chip subsystem further comprises a network interface for a respective core of the integrated circuit, the respective core having network access to a plurality of wiring circles connected to the network interface, the network interface further comprising an extension buffer that is shared among the plurality of wiring circles.

20. The integrated circuit of claim **12**, wherein the network-on-chip subsystem further comprises a network interface comprising one or more extension buffers configured to

145

receive network packets from a neighboring core as packets are being simultaneously received by an injection port and output from an ejection port of the network interface.

* * * * *

Routerless Networks-on-Chip

Fawaz Alazemi, Arash Azizimazreah, Bella Bose, Lizhong Chen
Oregon State University, USA
{alazemif, azizimaa, bose, chenliz}@oregonstate.edu

ABSTRACT

Traditional bus-based interconnects are simple and easy to implement, but the scalability is greatly limited. While router-based networks-on-chip (NoCs) offer superior scalability, they also incur significant power and area overhead due to complex router structures. In this paper, we explore a new class of on-chip networks, referred to as *Routerless NoCs*, where routers are completely eliminated. We propose a novel design that utilizes on-chip wiring resources smartly to achieve comparable hop count and scalability as router-based NoCs. Several effective techniques are also proposed that significantly reduce the resource requirement to avoid new network abnormalities in routerless NoC designs. Evaluation results show that, compared with a conventional mesh, the proposed routerless NoC achieves 9.5X reduction in power, 7.2X reduction in area, 2.5X reduction in zero-load packet latency, and 1.7X increase in throughput. Compared with a state-of-the-art low-cost NoC design, the proposed approach achieves 7.7X reduction in power, 3.3X reduction in area, 1.3X reduction in zero-load packet latency, and 1.6X increase in throughput.

1. INTRODUCTION

As technologies continue to advance, tens of processing cores on a single chip-multiprocessor (CMP) has already been commercially offered. Intel Xeon Phi Knight Landing [12] is an example of a single CMP that has 72 cores. With hundreds of cores in a CMP around the corner, there is a pressing need to provide efficient networks-on-chip (NoCs) to connect the cores. In particular, recent chips have exhibited the trend to use many but simple cores (especially for special-purpose many-core accelerators), as opposed to a few but large cores, for better power efficiency. Thus, it is imperative to design highly scalable and ultra-low cost NoCs that can match with many simple cores.

Prior to NoCs, buses have been used to provide on-chip interconnects for multi-core chips [7, 8, 14, 17, 37, 38]. While many techniques have been proposed to improve traditional buses, it is hard for their scalability to keep up with modern many-core processors. In contrast, NoCs offer a decentralized solution by the use of routers and links. Thanks to the switching capability of routers to provide multiple paths and parallel communications, the throughput of NoCs is significantly higher than that of buses. Unfortunately, routers have been notorious for consuming a substantial percentage of chip's power and area [20, 21]. Moreover, the cost of routers increases rapidly as link width increases. Thus, except for a few ad hoc designs, most on-chip networks do not employ link width higher than 256-bit or 512-bit, even though additional wiring resources may be available. In fact, our study shows that, a 6x6 256-bit Mesh only uses 3% of the total available wiring resources (more details in Section 3).

The high overhead of routers motivates researchers to develop *routerless NoCs* that eliminate the costly routers but use wires more efficiently to achieve scalable performance. While the notion of routerless NoC has not been formally mentioned before, prior research has tried to remove routers with sophisticated use of buses and switches, although with varying success. The goal of routerless NoCs is to select a set of smartly placed loops (composed of wires) to connect cores such that the average hop count is comparable to that of conventional router-based NoCs. However, the main roadblocks are the enormous design space of loop selection and the difficulty in avoiding deadlock with little or no use of buffer resources (otherwise, large buffers would defeat the purpose of having routerless NoCs).

In this paper, we explore efficient design and implementation to materialize the promising benefits of routerless NoCs. Specifically, we propose a layered progressive method that is able to find a set of loops that meet the requirement of connectivity and the limitation of wiring resources. The method progressively constructs the design of a large routerless network from good designs of smaller networks, and is applicable to any $n \times m$ many-core chips with superior scalability. Moreover, we propose several novel techniques to address the challenges in designing routerless interface to avoid network abnormalities such as deadlock, livelock and starvation. These techniques result in markedly reduced buffer requirement and injection/ejection hardware overhead. Compared with a conventional router-based Mesh, the proposed routerless design achieves 9.48X reduction in power, 7.2X reduction in area, 2.5X reduction in zero-load packet latency, and 1.73X increase in throughput. Compared with the current state-of-the-art scheme that tries to replace routers with less costly structures (IMR [28]), the proposed scheme achieves 7.75X reduction in power, 3.32X reduction in area,1.26X reduction in zero-load packet latency, and 1.6X increase in throughput.

2. BACKGROUND AND MOTIVATION

2.1 Related Work

Prior work on on-chip interconnects can be classified into *bus-based* and *network-based*. The latter can be further categorized as *router-based NoCs* and *routerless NoCs*. The main difference between bus-based interconnects and routerless NoCs is that bus-based interconnects use buses in a direct, simple and primitive way, whereas routerless NoCs use a network of buses in a sophisticated way and typically need some sort of switching that earlier bus systems do not need. Each of the three categories is discussed in more detail below.

Bus-based Interconnects are centralized communication systems that are straightforward and cheap to implement. While buses work very well for a few cores, the overall performance degrades significantly as more cores are connected to the bus [17, 37]. The two main reasons for such

degradation are the length of the bus and its capacitive load. Rings [7,8,14] can also be considered as variants of bus-based systems where all the cores are attached to a single bus/ring. IBM Cell processor [38] is an improved bus-based system which incorporates a number of bus optimization techniques in a single chip. Despite having a better performance over conventional bus/ring implementations, IBM Cell process still suffers from serious scalability issues [4].

Router-based NoCs are decentralized communication systems. A great deal of research has gone into this (e.g., [10, 13, 18, 23, 25, 26, 31, 33], too many to cite all here). The switching capability of routers provides multiple paths and parallel communications to improve throughput, but the overhead of routers is also quite substantial. Bufferless NoC (e.g., [15]) is a recent interesting line of work. In this approach, buffer resources in a router are reduced to the minimal possible size (i.e. one flit buffer per input port). Although bufferless NoC is a clever approach to reduce area and power overhead, the router still has other expensive components that are eliminated in the routerless approach (Section 7.5 compares the hardware cost).

Routerless NoCs aim to eliminate the costly routers while having scalable performance. While the notion of routerless NoC has not been formally mentioned before, there are several works that try to remove routers with sophisticated use of buses and switches. However, as discussed below, the hardware overhead in these works is quite high, some requiring comparable buffer resources as conventional routers, thus not truly materializing the benefits of routerless NoCs. One approach is presented in [34], where the NoC is divided into segments. Each segment is a bus, and all the segments are connected by a central bus. Segments and central bus are linked by a switching element. In large NoCs, either the segments or the central bus may suffer from scalability issues due to their bus-based nature. A potential solution is to increase the number of wires in the central bus and the number of cores in a segment. However, for NoCs larger than 8×8, it would be challenging to find the best size for the segments and central bus without affecting scalability. Hierarchical rings (HR) [16] has a similar design approach to [34]. The NoC is divided into disjoint sets of cores, and each set is connected by a ring. Such rings are called local rings. Additionally, a set of global rings bring together the local rings. Packets switch between local and global rings through a low-cost switching element. Although the design has many nice features, the number of switching element is still not small. For example, for an 8×8 NoC, there are 40 switching element, which is close to the number of routers in the 8×8 network. Recently, a multi-ring-based NoC called isolated multiple rings (IMR) is proposed in [28] and has been shown to be superior than the above Hierarchical rings. To our knowledge, this is the latest and best scheme so far along the line of work on removing routers. While the proposed concept is promising, the specific IMR design has several major issues and the results are far from optimal, as discussed in the next subsection.

2.2 Need for New Routerless NoC Designs

2.2.1 Principles and Challenges

We use Figure 1 to explain the basic principles of routerless NoCs. This figure depicts an example of a 16-core chip. The 4×4 layout specifies only the positions of the cores, not

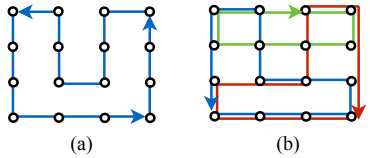


Figure 1: An example of loops in a 4×4 grid.

Table 1: Number of unidirectional loops in $n \times n$ grid [2].

n	# of loops	n	# of loops
1	0	2	2
3	26	4	426
5	18,698	6	2,444,726
7	974,300,742	8	1,207,683,297,862

any topology. A straightforward but naive way to achieve routerless NoC is to use a long loop (e.g., a Hamiltonian cycle) that connects every node on the chip as shown in Figure 1(a). Each node injects packets to the loop and receives packets from the loop through a simple interface (referred to as RL interface hereafter). Apparently, even if a flit on the loop can be forwarded to the next node at the speed of one hop per cycle, this design would still be very slow because of the average $O(n^2)$ hop count, assuming an $n \times n$ many-core chip. Scalability is poor in this case, as conventional topology as such Mesh has an average hop count of $O(n)$.

To reduce the hop count, we need to select a better set of loops to connect the nodes, while guaranteeing that every pair of nodes is connected by at least one loop (so that a node can reach another node directly in one loop). Figure 1(b) shows an example with the use of three loops, which satisfies the connectivity requirement and reduces the all-pair average hop count by 46% compared with (a). Note that, when injecting a packet, a source node chooses a loop that connects to the destination node. Once the packet is injected into a loop, it stays on this loop and travels at the speed of one hop per cycle all the way to the destination node. No changing loops is needed at RL interfaces, thus avoiding the complex switching hardware and per-hop contention that may occur in conventional router-based on-chip networks.

Several key questions can be asked immediately. Is the design in Figure 1(b) optimal? Is it possible to select loops that achieve comparable hop count as conventional NoCs such as Mesh? Is there a generalized method that we can use to find the loops for any $n \times n$ network? How can this be done without exceeding the available on-chip wiring resources? Unfortunately, answering these questions is extremely challenging due to the enormous design space. We calculated the number of possible loops for $n \times n$ chips based on the method used in [2], where a loop can be any unidirectional circular path with the length between 4 and n. Table 1 lists the results up to $n = 8$. As can be seen, the number of possible loops grows extremely rapidly. To make things more challenging, because the task is to find a set of loops, the design space that the routerless NoC approach is looking at is not the number of loops, but the combination of these loops! A large portion of the combinations would be invalid, as not all combinations can provide the connectivity where there is at least one loop between any source and destination pair.

Meanwhile, any selected final set of loops needs to comfortably fit in the available wiring resources on the chip. Specifically, when loops are superpositioned, the number of over-

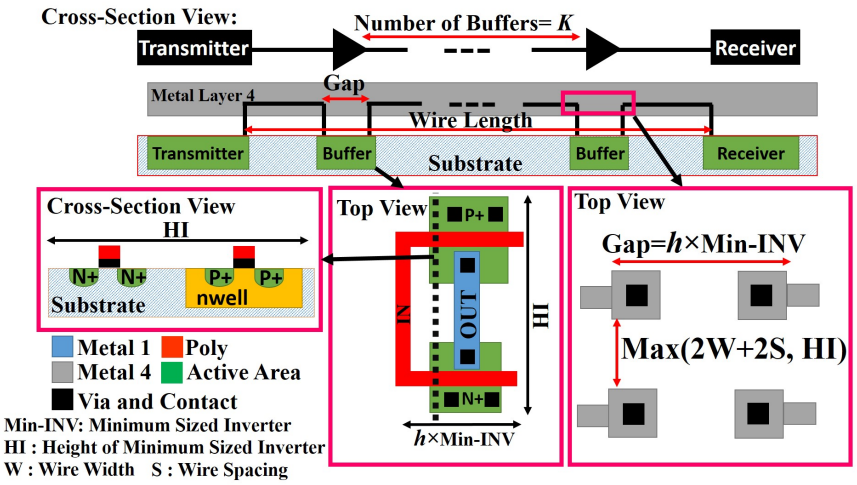


Figure 2: A long wire in NoCs with repeaters.

Table 2: Wiring resources in a many-core processor chip.

Many Core Processor	Xeon Phi, Knights Landing	
Number of Cores	72	
NoC Size	6×6	
Die Area	(31.9mm x 21.4mm) 683 mm^2 [3]	
Technology	FinFET 14nm	
Interconnect	13 Metal Layers	
Inter-core Metal Layers	Metal Layer	Pitch [22] [30]
	M4	80nm
	M5	104nm

lapped loops between any neighboring node pairs should not exceed a limit. In what follows, we use *overlapping* to refer to the number of overlapped loops between two neighboring nodes (e.g., in Figure 1(b) some neighboring nodes have two loops passing through them while others have only one loop passing), and use *overlapping cap* to refer to the limit of the overlapping. Note that the cap should be much lower than the theoretical wiring resources on chip due to various practical considerations (analyzed in Section 3). As an example, if the overlapping cap is 1, then Figure 1(a) has to be the final set. If the overlapping cap increases to 2, it provides more opportunity for improvement, e.g., the better solution in Figure 1(b). The overlapping cap is a hard limit and should not be violated. However, as long as this cap is met, it is actually beneficial to approach this cap for as many neighboring node pairs as possible. Doing this indicates more wires are being utilized to connect nodes and reduce hop count.

2.2.2 *Major Issues in Current State-of-the-Art*

There are several major issues that must be addressed in order to achieve effective routerless NoCs. We use IMR [28] as an example to highlight these issues. IMR is a state-of-the-art design that follows the above principle to deploy a set of rings such that each ring joins a subset of cores. While IMR has been shown to outperform other schemes with or without the use of routers, the fundamental issues in IMR prevent it from realizing the true potential of routerless NoCs. This calls for substantial research on this topic to develop more efficient routerless designs and implementations.

(1) *Large overlapping*. For example, IMR uses a large number of superpositioned rings (equivalent to the above-defined overlapping cap of 16) without analyzing the actual availability of wiring resources on-chip.

(2) *Extremely slow search*. A genetic algorithm is used in IMR to search the design space. This general-purpose search algorithm is very slow (taking several hours to generate results for 16×16, and is not able to produce good results in a reasonable time for larger networks). Moreover, the design generated by the algorithm is far from optimal with high hop counts, as evaluated in Section 6. Thus, efforts are much needed to utilize clever heuristics to speed up the process.

(3) *High buffer requirement*. Currently, the network interface of IMR needs one packet-sized buffer per ring to avoid deadlock. Given that up to 16 rings can pass through an IMR interface, the total number of buffers at each interface is very close to a conventional router.

The above issues are addressed in the next three sections. Section 3 analyzes the main contributing factors that determine the wiring availability in practice, and estimates reasonable overlapping caps using a contemporary many-core processor. Section 4 proposes a layered progressive approach to select a set of loops, which is able to generate highly scalable routerless NoC designs in less than a second (up to 128×128). Section 5 presents our implementation of routerless interface. This includes a technique that requires only one flit-sized buffer per loop (as opposed to one packet-sized buffer per loop). This technique alone can save buffer area by multiple times.

3. ANALYSIS ON WIRING RESOURCES

3.1 Metal Layers

As technology scales to smaller dimensions, it provides a higher level of integration. With this trend, each technology comes with an increasing number of routing metal layers to meet the growing demand for higher integration. For example, Intel Xeon Phi (Knights Landing) [1] and KiloCore [9] are fabricated in the process technology with 11 and 13 metal layers, respectively. Each metal layer has a pitch size which defines the minimum wire width and the space between two adjacent wires. The physical difference between metal layers results in various electrical characteristics. This allows designers to meet their design constraints such as delay on the critical nets by switching between different layers. Typically, lower metal layers have narrower width and are used for local interconnects (e.g., within a circuit block); higher metal layers have wider width and are used for global interconnects (e.g., power supply, clock); middle metal layers are used for semi-global interconnects (e.g., connecting neighboring cores). Table 2 lists several key physical parameters of Xeon Phi including the middle layers that can be used for on-chip networks.

3.2 Wiring in NoC

To estimate the actual wiring resources that can used for

routing, several important issues should be considered when placing wires on the metal layers.

Routing strategy: In general, two approaches can be considered for routing interconnects over cores in NoCs. In the first approach, dedicated routing channels are used to route wires in NoCs. This method of routing was widely used in earlier technology nodes where only three metal layers were typically provided [36], and it has around 20% area overhead. In the second approach, wires are routed over the cores at different metal layers [32]. In the modern technology nodes with six to thirteen metal layers, this approach of routing over logic becomes more common for higher integration. This can be done in two ways: 1) several metal layers are dedicated for routing wires, and 2) a fraction of each metal layer is used to route the wires. The first way is preferable given that many metal layers are available in advanced technology nodes [32, 36].

Repeater: Wires have parasitic resistance and capacitance which increase with the length of wires. To meet a specific target frequency, a long wire needs to be split into several segments, and repeaters (inverters) are inserted between the segments, as shown in Figure 2. The size of repeaters should be considered in estimating the available wiring resources. For a long wire in the NoC, the size of each repeater (h times of an inverter with minimum size) is usually not small, but the number of repeaters (k) needed is small [27]. In fact, it has been shown that increasing K has negligible improvement in reducing the delay [27]. For a 2GHz operating frequency, using only one repeater with the size of 40 times W/L of the minimum sized inverter can support a wire length of 2mm [32], which is longer than the typical distance between two cores in a many-core processor [35].

Coping with cross-talk: Cross-talk noises can occur either between the wires on the same metal layer or between the wires on different metal layers, both of which may affect the number of wires that can be placed. The impact of cross-talk noises on voltage can be calculated by Equation (1) as the voltage changes on a floated victim wire [19].

$$\Delta V_{victim} = \frac{C_{adj}}{C_{victim} + C_{adj}} \times \Delta V_{aggressor} \quad (1)$$

where ΔV_{victim} is the voltage variation on the victim wire, $\Delta V_{aggressor}$ is the voltage variation on the aggressor, C_{victim} is the total capacitance (including load capacitance) of the victim wire, and C_{adj} is the coupling capacitance between the aggressor and the victim. It can be observed from Equation (1) that the impact of cross-talk on the victim wire depends on the ratio of C_{adj} to C_{victim}. Hence, the cross-talk on the same layer has much larger impact on the power, performance, and functionality of the NoC since the adjacent wires which run in parallel on the same metal layer has larger coupling capacitance (C_{adj}) [19]. There are two major techniques to mitigate cross-talk noises, shielding and spacing. In the shielding approach, crosstalk noises are largely avoided between two adjacent wires by inserting another wire (which is usually connected to the ground or supply voltage) between them. In the spacing approach, adjacent wires are separated by a certain distance that would keep the coupling noise below a level tolerable by the target process and application. Compared with spacing, shielding is much more effective as it can almost remove crosstalk noises [5]. However, shielding also incurs more area overhead as the distance used in the spacing approach is usually smaller than that of inserting a wire.

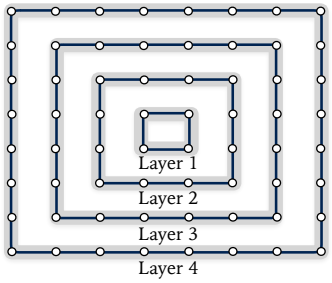


Figure 3: Layers of an 8×8 grid.

3.3 Usable Wires for NoCs

To gain more insight on how many wiring resources are usable for on-chip networks under current manufacturing technologies, we estimated the number of usable wires by taking into account the above factors. The estimation is based on using two metal layers to route wires over the cores. The area overhead of the repeater insertion including the via contacts and the area occupation of the repeaters are considered based on the layout design rules of each metal layer. We used the conservative way of shielding to reduce crosstalk noises (and the inserted wires are not counted towards usable wires), although spacing may likely offer more usable wires. In addition, in practice, 20% to 30% of each dedicated metal layer for routing wires over the cores is used for I/O signals, power, and ground connections [32]. This overhead is also accounted for. The maximum values of h and K are used for worst-case estimation. As such, the above method gives a very conservative estimation of the usable wires. Assuming that there is a chip with similar physical configuration as Table 2, the two metal layers M4 and M5 under 14nm technology can provide 101,520 wires in the cross-section. This translates into 793 unidirectional links of 128-bit, or 396 unidirectional links of 256-bit, or 198 unidirectional links of 512-bit in the cross-section. In contrast, a 6×6 mesh only uses 12 unidirectional 256-bit links in the bisection, which is about 3% of the usable wires. It is important to note that the conventional router-based NoCs do not use very wide links for good reasons. For instance, router complexity (e.g., the number of crosspoints in switches, the size of buffers) increases rapidly as the link width increases. Also, although wider links provide higher throughput, it is difficult to capitalize on wider links for lower latency. The reduction in serialization latency by using wider links quickly becomes insignificant as link width approaches the packet size. This motivates the need for designing routerless NoCs where wiring resources can be used more efficiently.

The above estimation of the number of usable wires helps to decide the overlapping cap mentioned previously. To avoid taxing too much on the usable wiring resources and to have a scalable design, we propose to use an overlapping cap of n for $n \times n$ chips. In the above 6×6 case, this translates into 4.5% of the usable wires for 128-bit loop width, or 9.1% for 256-bit loop width. This parameterized overlapping cap helps to provide the number of loops that is proportional to chip size, so the quality of the routerless designs can be consistent for larger chips.

4. DESIGNING ROUTERLESS NOCS

4.1 Basic Idea

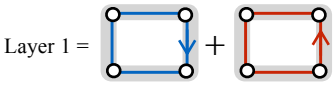


Figure 4: Loops in L_1, and $M_2 = L_1$.

Our proposed routerless NoC design is based on what we call *layered progressive* approach. The basic idea is to select the loop set in a progressive way where the design of a large routerless network is built on top of the design of smaller networks. Each time the network size increments, the newly selected loops are conceptually bundled as a layer that is reused in the next network size.

Specifically, let M_k be the final set of selected loops for $k \times k$ grid ($2 \leq k \leq n$) that meets the connectivity, overlapping and low hop count requirements. We construct M_{k+2} by combining M_k with a new set (i.e., layer) of smartly placed loops. The new layer utilizes new wiring resources that are available when expanding from $k \times k$ to $(k+2) \times (k+2)$. The resulting M_{k+2} can also meet all the requirements and deliver superior performance. For example, as shown in Figure 3, the grid is logically split into multiple layers with increasing sizes. Let L_k be the set of loops selected for Layer k. Firstly, suppose that we already find a good set of loops for 2×2 grid that connects all the nodes with a low hop count and does not exceed an overlapping of 2 between any neighboring nodes. That set of loops is M_2, which is also L_1 as this is the base case. Then we find another set of loops L_2, together with M_2, can form a good set of loops for 4×4 grid (i.e., $M_4 = L_2 \cup M_2$). The resulting M_4 can connect all the nodes with a low hop count and do not exceed an overlapping of 4 between any neighboring nodes. And so on so forth, until reaching the targeted $n \times n$ grid. In general, we have $M_n = L_{\lfloor n/2 \rfloor} \cup M_{n-2} = L_{\lfloor n/2 \rfloor} \cup L_{\lfloor n/2 \rfloor - 2} \cup M_{n-4} = \ldots = L_{\lfloor n/2 \rfloor} \cup L_{\lfloor n/2 \rfloor - 2} \cup L_{\lfloor n/2 \rfloor - 4} \cup \ldots \cup L_1$.

Apparently, the key step in the above progressive process is how to select the set of loops in Layer k, which enables the progression to the next sized grid with low hop count and overlapping. In the next subsections, we walk through several examples to illustrate how it is done to progress from 2×2 grid to 8×8 grid.

4.2 Examples

4.2.1 2×2 *Grid*

This is the base case with one layer. There are exactly two possible loops, one in each direction, in a 2×2 grid. Both of them are included in $M_2 = L_1$, as shown in Figure 4. The resulting M_2 satisfies the requirement that every source and destination pair is connected by at least one loop. The maximum number of loops overlapping between any neighboring nodes is 2, which meets the overlapping cap. This set of loops achieves a very low all-pair average hop count of 1.333, which is as good as the Mesh.

4.2.2 4×4 *Grid*

M_4 consists of loops from two layers. Based on our layered progressive approach, L_1 is from M_2. We select 8 loops to form L_2, as illustrated in Figure 5. The 8 loops fall into four groups (from this network size and forward, each new layer is constructed using four groups with the similar heuristics as discuss below). The first group, A_4 (the subscript indicates the size of the grid), has only one anti-clockwise loop. It provides connectivity among the 12 new nodes when expanding from Layer 1 to Layer 2. The loops in the second group, B_4, have the first column as the common edge of the loops, but the opposite edge of the loops moves gradually towards the right (this is more evident in group B_6 in Figure 6). Similarly, the third group, C_4, uses the last column as the common edge of the loops and gradually moves the opposite edge towards the left. It can be verified that groups B_4 and C_4 provide connectivity between the 12 new nodes in Layer 2 and the 4 nodes in Layer 1. Since the connectivity among the 4 inner nodes has already been provided by L_1, the connectivity requirement of 4×4 grid is met by having L_1, A_4, B_4 and C_4. The fourth group, D_4, offers additional "shortcuts" in the horizontal dimension.

A very nice feature of the selected M_4 is that the wiring resources are efficiently utilized, as the overlapping between many neighboring node pairs is close to the overlapping cap of 4. For example, for the first (or the last) column, each group of loops has exactly one loops passing through that column, totaling an overlapping of 4, which is the same as the cap. Thus, no overlapping "ration" is under-utilized. For the second column (or the third) column, groups A_4 and D_4 have no loop passing through, and groups B_4 and C_4 have two loops passing through in total. However, note that the final M_4 also includes L_1 which has two loops passing through the second (or the third) column. Hence, the total overlapping of the middle columns is also 4, exactly the same as the cap. Simple counting can show that the overlapping on the horizontal dimension is also 4 for each row. Owing to this efficient use of wiring resource "ration", the all-pair average hop count is 3.93 for the selected set of loops in M_4. The final set is $M_4 = L_2 \cup M_2 = L_2 \cup L_1$.

4.2.3 6×6 *Grid*

M_6 consists of loops from three layers. L_1 and L_2 are from M_4, and L_3 is formed in a similar fashion as 4×4 grid from four groups, as illustrated in Figure 6. Again, connectivity is provided by M_4 and groups A to C. Together with group D, the number of overlapping on each column and row is 6, thus fully utilizing the allocated wiring resources.

Additionally, for the purpose of reducing hop count and balancing horizontal and vertical wiring utilization, when we combine M_4 and L_3 to form M_6, every loop in M_4 is reversed and then rotated for 90° clockwise[1]. If this slightly changed M_4 is denoted as M_4', the final set can be expressed as $M_6 = L_3 \cup M_4' = L_3 \cup (L_2 \cup L_1)'$, with an all-pair average hop count of 6.07.

4.2.4 8×8 *Grid*

Similar to earlier examples, L_4 consists of loops shown in Figure 7. The final set M_8 is $M_8 = L_4 \cup M_6' = L_4 \cup \big(L_3 \cup (L_2 \cup L_1)'\big)'$ with an all-pair average hop count of 8.32.

4.3 Formal Procedure

For an $n \times n$ grid, the loops for a routerless NoC design can be recursively found by the procedure shown in Algorithm 1. The procedure is recursive and denoted as RLrec. The procedure begins by generating loops for the outer layer, say layer i, and then it recursively generates loops for layer $i-1$

[1] In 4×4 grid, reversal and rotation of M_2 is not necessary because M_2 and M_2' have the same effect on L_1.

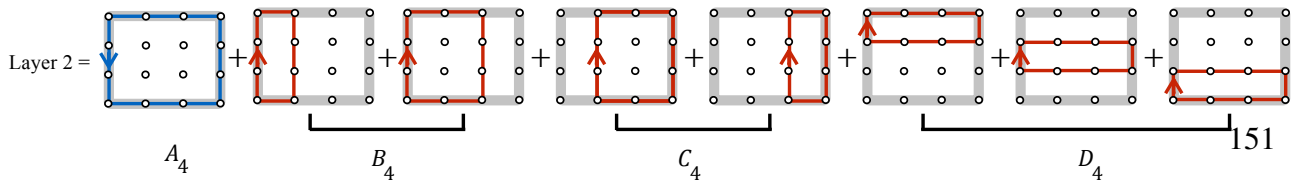


Figure 5: Loops in L_2. $M_4 = L_2 \cup L_1$.

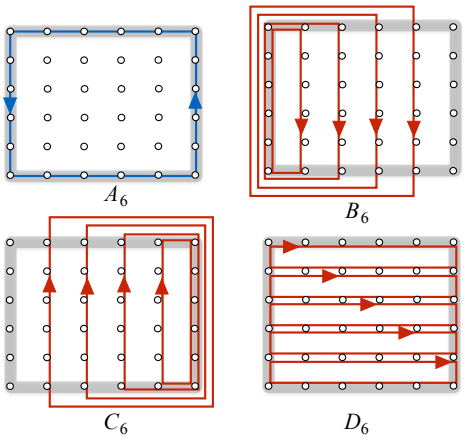


Figure 6: Loops in L_3. $M_6 = L_3 \cup L_2 \cup L_1$.

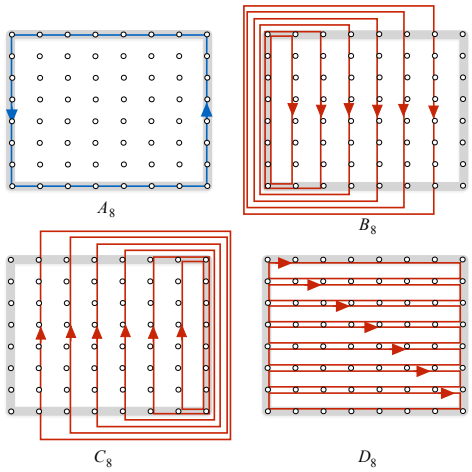


Figure 7: Loops in L_4. $M_8 = L_4 \cup L_3 \cup L_2 \cup L_1$.

Algorithm 1: RLrec

Input : N_L, N_H; the low and high numbers
begin
  if $N_L = N_H$ **then**
    return {}
  Let $M = \{\}$
  if $N_H - N_L = 1$ **then**
    $M = M \cup G(N_L, N_H, N_L, N_H$, clockwise)
    $M = M \cup G(N_L, N_H, N_L, N_H$, anticlockwise)
    return M
  $M = M \cup G(N_L, N_H, N_L, N_H$, anticlockwise) // Group A
  for $i = N_L + 1 \rightarrow N_H - 1$ **do**
    $M = M \cup G(N_L, N_H, N_L, i$, clockwise) // Group B
    $M = M \cup G(N_L, N_H, i, N_H$, clockwise) // Group C
  for $i = L \rightarrow H - 1$ **do**
    $M = M \cup G(i, i+1, N_L, N_H$, clockwise) // Group D
  $M' =$ RLrec(N_L+1, N_H-1)
  Reverse and rotate for 90° every loop in M'
  return $M \cup M'$

and so on until the base case is reached or the layer has a single node or empty. Procedure $G(r_1, r_2, c_1, c_2, d)$ is a simple function that generates a rectangular shape loop with corners (r_1, c_1), (r_1, c_2), (r_2, c_1) and (r_2, c_2) and direction d. When processing each layer in this algorithm, procedure G is called repeatedly to generate four groups of loops. Additionally, the generated loops rotate 90 degrees and reverse directions after processing each layer to balance wiring utilization and reduce hop count, respectively. The final loops generated by the RLrec algorithm have an overlapping of at most n.

While it would be ideal if an analytical expression can be derived to calculate the average hop count for this heuristic approach, this seems to be very challenging at the moment. However, it is possible to calculate the average hop count numerically. This result is presented in the evaluation, which shows that our proposed design is highly scalable.

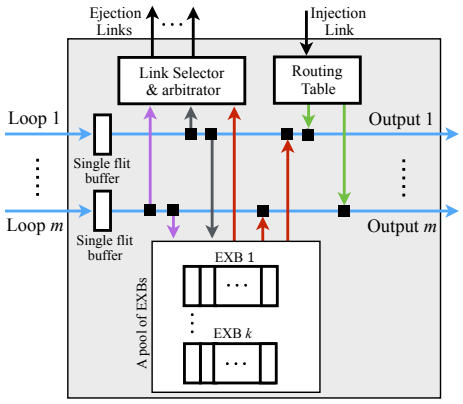


Figure 8: Routerless interface components.

5. IMPLEMENTATION DETAILS

After addressing the key issue of finding a good set of loops, the next important task is to efficiently implement the routerless NoC design in hardware. Because of the routerless nature, no complex switching or virtual channel (VC) structure is needed at each hop, so the hardware between nodes and loops has a small area footprint in general. However, due to various potential network abnormalities such as deadlock, livelock, and starvation, a certain number of resources are required to guarantee correctness. If not addressed appropriately, this may cause substantial overhead that is comparable to router-based NoCs. In this section, we propose a few

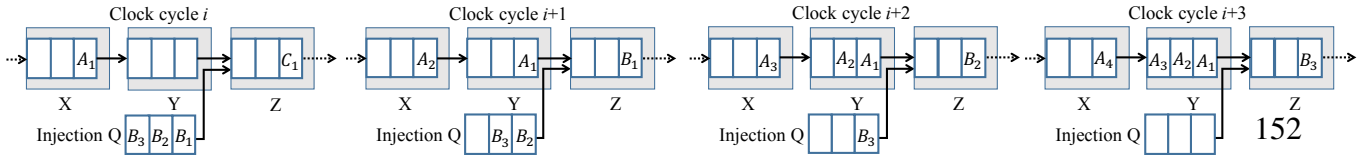


Figure 9: Injecting a long packet requires a packet-sized buffer per loop at each hop in prior implementation (X, Y and Z are interfaces).

effective techniques to minimize those overhead.

In a routerless NoC, each node uses an interface (RL interface) to interact with one or multiple loops that pass through this node. Figure 8 shows the main components of a RL interface. While details are explained in the following subsections, the essential function of the interface includes injecting packets into a matching loop based on connectivity and availability, forwarding packets to the next hop on the same loop, and ejecting packets at the destination node. Notice that packets cannot switch loops once injected. All the loops have the same width (e.g., 128-bit wires).

5.1 Injection Process

5.1.1 Extension Buffer Technique

A loop is basically a bundle of wires connected with flip-flops at each hop (Figure 8). At clock cycle i, a flit arriving at the flip-flop of loop l must be consumed immediately by either being ejected at this node or forwarded to the next hop on loop l through output l. If no flit arrives at loop l (thus not using output l), the RL interface can inject a new flit on loop l through output l. However, it is possible that an injecting packet consists of multiple flits and requires several cycles to finish the injection, during which other flits on loop l may arrive at this RL interface. Therefore, addition buffer resources are needed to hold the incoming flits temporarily.

If routerless NoC uses the scheme proposed in prior ring-based work (e.g., IMR [28]), a full packet-sized buffer per loop at each hop would be needed to ensure correctness, which is very inefficient. As illustrated in Figure 9, a long packet B with multiple flits is waiting for injection (there is no issue if it is a short single-flit packet). At clock cycle i, the injection is allowed because packet B sees that no other flit in Interface Y is competing with B for the output to Interface Z. From cycle $i+1$ to $i+3$, the flits of B are injected sequentially. However, while packet B is being injected during these cycles, another long packet A may arrive at Interface Y. Because RL interfaces do not employ flow control to stop the upstream node, Interface Y needs to provide a packet-sized buffer to temporarily store the entire packet A. A serious inefficiency lies in the fact that, if there are m loops passing through a RL interface, the interface needs to have m packet-sized buffers, one for each loop.

To address this inefficiency, we notice that an interface injects packets one at a time, so not all the loops are affected simultaneously. Based on this observation, we propose the extension buffer technique to share the packet-sized buffer among loops. As shown in Figure 8, each loop has only a flit-sized buffer, but the interface has a pool of extension buffers (EXBs). The size of each EXB is the size of a long packet, so when a loop is "extended" with an EXB, it would be large enough to store a long packet. Minimally, only one EXB is needed in the pool, but having multiple EXBs may have slight performance improvement. This is because another injection might occur while the previous EXB is not entirely released (drained) due to a previous injection (e.g., clock cycle $i+3$ in Figure 9). However, as shown later in the evaluation, the performance difference is negligible. As a result, our proposed technique of using one shared EXB can essentially achieve the same objective of ensuring correctness as IMR but reduces the buffer requirement by m times. This is equivalent to an 8X saving in buffer resources in 8×8 networks and 16X saving in 16×16 networks.

5.1.2 Injection Process

The injection process with the use of EXBs is straightforward. To inject a packet p of n_f flits, the first step is to look up a small routing table to see which loop can reach p's destination. The routing table is pre-computed since all the loops are pre-determined. The packet p then waits for the loops to become available (i.e., having sufficient buffer space). Assume l is a loop that has the shortest distance to the destination among all the available loops. When the injection starts, the interface holds the output port of l for n_f cycles to inject p, and assigns a free EXB to l if $n_f > 1$ and l is not already connected to another EXB. During those n_f cycles, any incoming flit through the input port of l is enqueued in the extension buffer. The EXB is released later when its buffer slots are drained.

5.2 Ejection Process

The ejection process starts as soon as the head flit of a packet p reaches the RL interface of its destination node. The interface ejects p, one flit per cycle. Once p is ejected, the interface will wait for another packet to eject. There is, however, a potential issue with the ejection process. While unlikely, a RL interface with m loops may receive up to m head flits simultaneously in a given cycle that are all destined to this node. Because any incoming packets need to be consumed immediately and the packets are already at the destination, the interface needs to have m ejection links in order to eject all the packets in that cycle. As each eject link has the same width as the loop (i.e., 128-bit), this incurs substantial hardware overhead.

To reduce this overhead, we utilize the fact that the actual probability of having k packets ($1 < k \leq m$) arriving at the same destination in the same cycle is low, and this probability decreases drastically as k increases. Based on this observation, we propose to optimize for the common case where only e ejection links are provided ($e \ll m$). If more than e packets arrive at the same cycle, $(k-e)$ packets are forwarded to the next hop. Those deflected packets will continue on their respective loops and will circle back to the destination later. As shown in the evaluation, having two ejection links can reduce the percentage of circling packets to be below 1% on average (1.6% max) across the benchmarks. This demonstrates that this is a viable and effective technique to reduce overhead.

5.3 Avoiding Network Abnormalities

As network abnormalities are theoretically possible but

practically unlikely scenarios, our design philosophy is to place very relaxed conditions to trigger the handling procedures, so as to minimize performance impact while guaranteeing correctness.

5.3.1 Livelock Avoidance

A livelock may occur if a packet circles indefinitely and never gets a chance to eject. We address this issue by having a byte-long circling counter at each head flit with an initial value of zero. Every time a packet reaches its destination interface and is forced to be deflected, the counter is incremented by 1. If the circling counter of a packet p reaches 254 but none of the ejection link is available, the interface marks one of its ejection links as reserved and then deflects p for the last time. The marked ejection link will not eject any more packets after finishing the current one, until p circles back to the ejection link (by then the marked ejection link will be available; otherwise there is a possible protocol-level deadlock, discussed shortly). Once p is ejected, the interface will unmark the ejection link for it to function normally. Due to the extremely low circling percentage (maximum 3 times of circling for any packet in our simulations), this livelock avoidance scheme has minimal performance impact.

5.3.2 Deadlock Avoidance

With no protocol-level dependence at injection/ejection endpoints, routing-induced deadlock is not possible in routerless NoCs as packets arriving at each hop are either ejected or forwarded immediately. Hence, a packet can always reach its destination interface without being blocked by other packets. The above livelock avoidance ensures that the packet can be ejected within a limited number of circlings.

With more than one dependent packet types, the marked ejection link in the above livelock avoidance scheme may not able to eject the current packet (say a request packet) in the ejection queue, because the associated cache controller cannot accept new packets from the ejection queue (i.e., input of the controller). This may happen when the controller itself is waiting for packets (say a reply packet) in the injection queue (i.e., output of the controller) to be injected into the network. A potential protocol-level deadlock may occur if that reply packet cannot be injected, such as the loop is full of request packets that are waiting to be ejected.

To avoid such protocol-level deadlock, the conventional approach is to have a separate physical or virtual network for each dependent packet type. While similar approach can be used for routerless NoCs, here we propose a less resource demanding solution. which is made possible by the circling property of loops. This solution only needs an extra reserved EXB, as well as a separate injection and ejection queue for each dependent packet type. The separate injection/ejection queues can come from duplicating original queues or from splitting the original queues to multiple queues. In either case, the loops and wiring resources are *not* duplicated, which is important to keep the cost low. Following the above livelock avoidance scheme, when a packet p on loop l completes the final circling (counter value of 255) and finds that the marked ejection link is still not available, p is temporarily buffered in the reserved EXB instead of forwarding to output l. Meanwhile, we allow the head packet q in the injection queue of the terminating packet type (e.g., a reply packet in the request-reply example) to inject into loop l through output l. Once q is injected, the cache controller is able to put another reply packet in its output (i.e., the injection queue) which, in turn, allows the controller to accept a new request from its input (i.e., the ejection queue). This creates space in the ejection queue to accept packet p that is previously stored in the reserved EXB. Once p moves to the ejection queue, the EXB is freed. Essentially, the reserved EXB acts as a temporary exchanging space while the separate injection/ejection queues avoid blocking of different packet types at the endpoints.

5.3.3 Starvation Avoidance

The last corner case we address is starvation. With the previous livelock and deadlock handling, if a packet is consumed at its destination RL interface, the interface can use the free output to inject a new packet. However, it is possible that a particular interface X is not the destination of any packets and there is always a flit passing through X every single cycle. This never occurred in any of our experiments as it is practically impossible that a cache bank is not accessed by any other cores. However, it is theoretically possible and, when occurred, prevents X from injecting new packets. We propose the following technique to avoid starvation for the completeness of the routerless NoC design. If X cannot inject a packet after a certain number of clock cycles (a very long period, e.g., hundreds of thousand cycles or long enough to have negligible impact on performance), X piggybacks the next passing head flit f with the ID of X. When f is ejected at its destination interface Y, instead of injecting a new packet, Y injects a single-flit no-payload dummy packet that is destined to X. When the dummy packet arrives at X, X can now inject a new packet by using the free slot created by the dummy packet. This breaks the starvation configuration.

5.4 Interface Hardware Implementation

Figure 8 depicts the main components of a RL interface. We have explained the extension buffers (EXBs), single-flit buffers, routing table, and multiple ejection links in the previous subsections. The arbitrator receives flits from input buffers and selects up to e input loops for ejection based on the oldest first policy. The arbitrator contains a small register that holds the arbitration results. The link status selector is a simple state machine associated with the loops. It monitors the input loops and arbitration results, and changes the state of the loops (e.g., ejection, stall in extension buffers, etc.) in the state machine. There are several other minor logic blocks that are not shown in Figure 8 for better clarity. Note that the RL interface does not use the information of neighboring nodes, which differs from most conventional router-based NoCs that need credits or on/off signals for handshaking.

To ensure the correctness of the proposed interface hardware, we implement the design in RTL Verilog that includes all the detailed components. The Verilog implementation is verified in Modelsim, synthesized in Synopsys Design Compiler, and placed and routed using Cadence Encounter tool. We use the latest 15nm process NanoGate FreePDK 15 Cell Library [29] for more accurate evaluation. As a key result, the RL interface is able to operate at up to 4.3GHz frequency while keeping the packet forwarding process in one clock cycle. This is fast enough to match up with most commercial many-core processors. Injecting packets may take an additional cycle for table look-up. In the main evaluation below, both the interfaces and cores are operating at 2GHz.

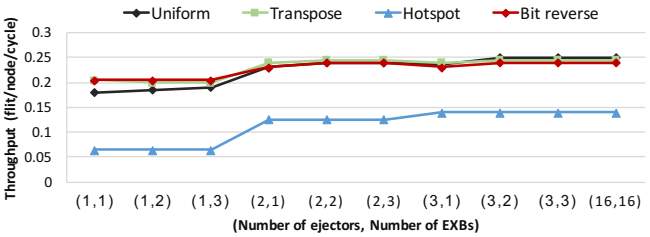


Figure 10: Throughput of routerless NoC under different number of ejection links and extension buffers (EXBs).

6. EVALUATION METHODOLOGY

We evaluate the proposed routerless NoC (RL) extensively against Mesh, EVC, and IMR in Booksim [24]. For synthetic traffic workloads, we use uniform, transpose, bit reverse, and hotspot (with 8 hotspots nodes). BookSim is warmed up for 10,000 clock cycles and then collects performance statistics for another 100,000 cycles at various injection rates. The injection rate starts at 0.005 flit/node/cycle and is incremented by 0.005 flit/node/cycle until the throughput is reached. Moreover, we integrate Booksim with Synfull [6] for performance study of PARSEC [11] and SPLASH-2 [39] benchmarks. Power and area studies are based on Verilog post-synthesis simulations, as described in Section 5.4.

In the synthetic study, each router in Mesh is configured with relatively small buffer resources, having 2 VCs per link and 3 flits per VC. The link width is set to 256-bit. Also, the router is optimized with lookahead routing and speculative switch allocation to reduce pipeline stages to 2 cycles per router and 1 cycle per link. EVC has the same configuration as Mesh except for one extra VC that is required to enable express channels. For IMR, the ring set is generated by the evolutionary approach described in [28]. To allow a fair comparison with RL, the maximum number of overlapping cap, for both RL and IMR, is set to n for $n \times n$ NoC. We also follow the original paper to faithfully implement IMR's network interface. Each input link in an IMR's interface is attached with a buffer of 5 flits and the link width is set to 128-bit (the same as the original paper). In RL, loops are generated by RLrec algorithm and accordingly the routing table for each node is calculated. Each interface is configured with two ejection links and each input link has a flit-size buffer. Also, an EXB of 5 flits is implemented in each interface. The link width is 128-bit (the same as IMR). In all the designs, packets are categorized into data and control packets where each control packet has 8 bytes and each data packet has 72 bytes. Accordingly, data packets in Mesh, EVC, IMR, and RL are of 3, 3, 5 and 5 flits, respectively, and the control packets are of a single flit.

For benchmark performance study, we also add 2D Mesh with various configurations as well as a 3D Cube design into the comparison. RL has the same configuration as the synthetic study. For 2D Mesh, we use 9 configurations, each having the configuration M(x,y) where $x \in \{1,2,3\}$ is the router delay and $y \in \{1,2,3\}$ is the buffer size, i.e., routers with 1-cycle, 2-cycle and 3-cycle delay, and with 1-flit, 2-flit and 3-flit buffer size. 3D Cube is configured with 2 VCs per link, 3 flits per VC, and 2-cycle per hop latency.

7. RESULTS AND ANALYSIS

7.1 Ejection Links and Extension Buffers

The proposed RL scheme is flexible to use any number of ejection links and EXBs. On the ejection side, the advantages of having more ejection links are higher chance for packet ejection and lower chance for packet circling in a loop. However, adding more ejection links complicates the design of the interface and leads to additional power and area overhead in the interface and the receiving node. On the injection side, EXBs have a direct effect on the injection latency of long packets. Recall that, a loop must be already attached with an EXB or a free EXB is available to be able to inject a long packet. Similar to ejection links, having more EXBs can lower injection latency but incur larger area and power overhead.

We studied the throughput of RL with different configurations of ejection links and EXBs on various synthetic traffic patterns. The NoC size for this study is 8×8. The results are shown in Figure 10. In the figure, each configuration is denoted by (x,y) where x is the number of ejection links and y is the number of EXBs. The basic and best in terms of area and power overhead is $(1,1)$ configuration but it has the worst performance. By adding up to three EXBs with a single ejection link, the throughput is only slightly changed (less than 5%). This indicates that the number of EXBs is not very critical to performance, and it is possible to use only one EXB for injecting long packets while saving buffer space.

For $(2,1)$ configuration, it doubles the chance for packet ejection when compared to $(1,x)$ configurations. The throughput is notably improved by an average of 38% for all the patterns when compared to $(1,1)$ configurations. For instance, hotspot traffic pattern has 0.125 throughput in $(2,1)$ configuration but only 0.065 in $(1,1)$, a 92.5% improvement). However, on top of $(2,1)$ configuration, adding up-to three EXBs (i.e., $(2,3)$) improves throughput only by 5% on average.

Given all the results, we choose the $(2,1)$ configuration as the best trade-off point, and use it for the remainder of this section. We also plot the $(16,16)$ configuration which is the ideal case (no blocking in injection or ejection may happen). As can be seen, $(2,1)$ is very close to the ideal case. Section 7.3 provides a detailed study for the number of times packet circling in loops for the $(2,1)$ configuration.

7.2 Synthetic Workloads

Figure 11 plots the performance results of four synthetic traffic patterns for an 8×8 NoC. RL has the lowest zero-load packet latency in all four traffic patterns. For example, in uniform random, the zero-load packet latency is 21.2, 14.9, 10.5, and 8.3 cycles for Mesh, EVC, IMR, and RL, respectively. When averaged over the four patterns, RL has an improvement of 1.59x, 1.43x, and 1.25x over Mesh, EVC, and IMR, respectively. RL achieves this due to low per hop latency (one cycle) and low hop count.

In terms of throughput, the proposed RL also has advantage over other schemes. For example, the throughput for hotspot is 0.08, 0.05, 0.06, and 0.125 (per flit/node/cycle) for Mesh, EVC, IMR, and RL, respectively. In fact, RL has the highest throughput for all the traffic patterns. When averaged over the four patterns, RL improves throughput by 1.73x, 2.70x, and 1.61x over Mesh, EVC, and IMR, respectively. This is mainly owing to the better utilization of wiring resources in RL. Note that, EVC has a lower throughput than Mesh as EVC is essentially a scheme that trades off throughput for lower latency at low traffic load.

7.3 PARSEC and SPLASH-2 Workloads

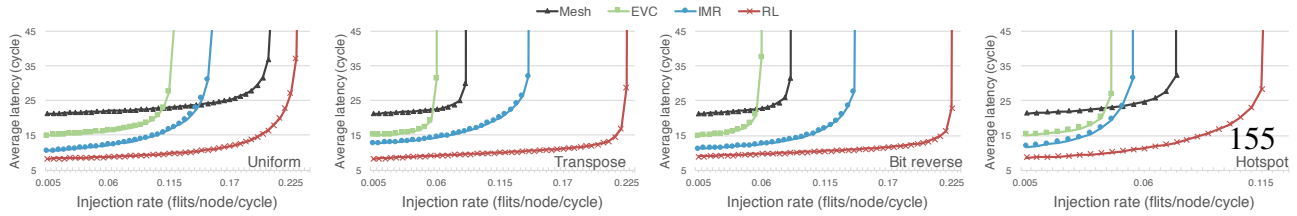


Figure 11: Performance comparison for synthetic traffic patterns.

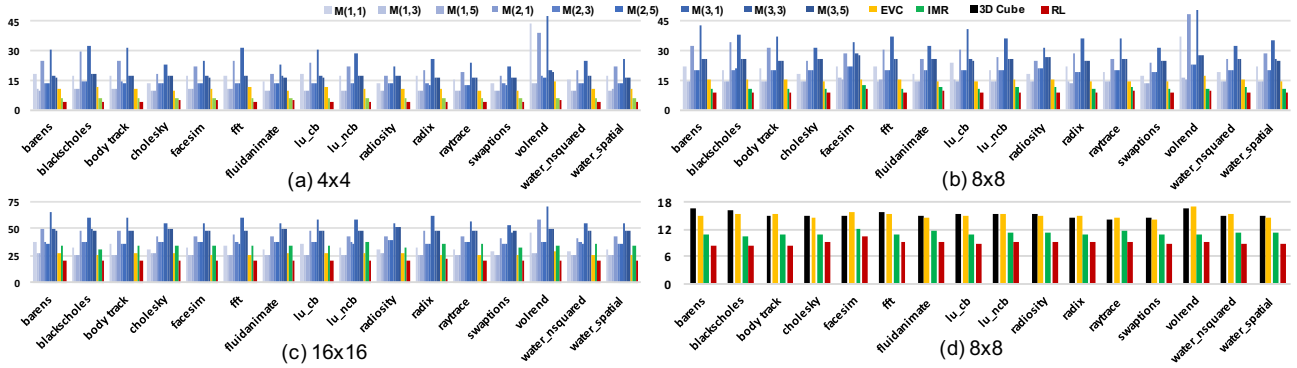


Figure 12: PARSEC and SPLASH-2 benchmark performance results (y-axis represents average pack latency in cycles.) RL is compared with different Mesh configurations, EVC, and IMR in (a), (b) and (c). In (d), RL is also compared with a 3D Cube.

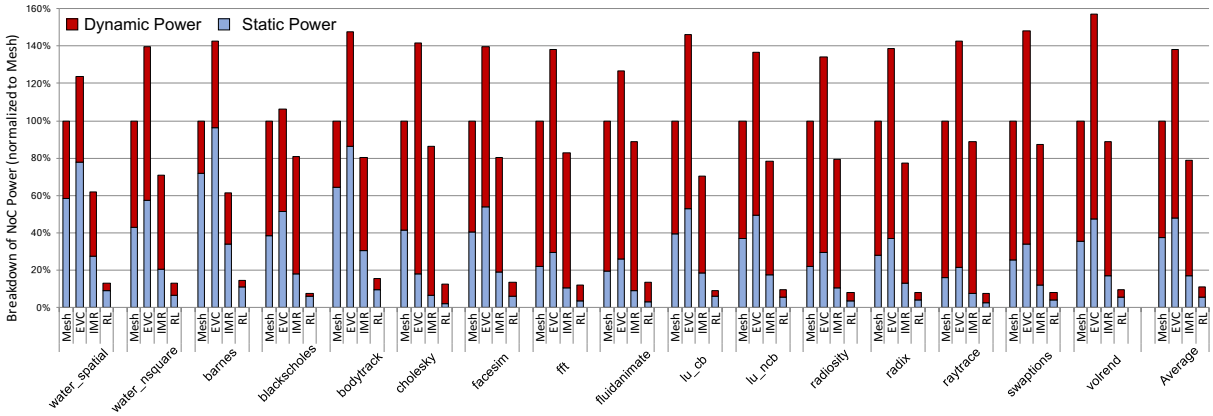


Figure 13: Breakdown of power consumption for different PARSEC and SPLASH-2 workloads (normalized to Mesh).

We utilize Synfull and Booksim to study the performance of RL, 2D Mesh with different configurations, EVC, IMR, and a 3D Cube under 16 PARSEC and SPLASH-2 benchmarks. The NoC sizes under evaluation are 4×4, 8×8 and 16×16 for RL, 2D Mesh, EVC and IMR, and $4 \times 4 \times 4$ for 3D cube. Figure 12 shows the results.

In Figure 12(a)-(c), RL is compared against 2D Mesh, EVC and IMR. From the figures, the best configuration for Mesh is M(1,5) (i.e. per hop latency of 1 and buffer size of 5) and the worst is M(3,1). Lowering per hop latency in Mesh helps to improve overall latency, and reducing buffer sizes may cause packets to wait longer for credits and available buffers. The average packet latency of RL in 4×4, 8×8, and 16×16 are 4.3, 8.9 and 20.1 cycles, respectively. This translates into an average latency reduction of RL over M(1,5) by 57.8%, 38.4% and 22.2% in 4×4, 8×8 and 16×16, respectively. The IMR rings in 16×16 are very long and seriously affects its latency. RL reduces the average latency by 23.3% over EVC and 41.2% over IMR.

In Figure 12(d), the performance of 3D cube is clearly better than all the Mesh configurations in (b) mainly due to lower hop count and larger bisection bandwidth. Despite this, RL still offers better performance than 3D cube. The average latency of RL is 8.9 cycles, which is 41% lower than the 15.2 cycles of 3D cube.

7.4 Power

Figure 13 compares the power consumption of Mesh (i.e. M(2,3)), EVC, IMR and RL for different benchmarks, normalized to the Mesh. All the power consumption shown in this Figure are reported after P&R in NanGate FreePDK 15 Cell Library [29] by Cadence Encounter. The activity factors for the power measurement are obtained from Booksim, and the power consumption includes that of all the wires.

The average dynamic power consumption for RL is only 0.26mW, and for Mesh, EVC and IMR the average is 2.88mW, 4.27mW and 2.91mW, respectively. Because RL has no crossbar, it requires only 9%, 6.1% and 8.9% of the dynamic power consumed by Mesh, EVC and IMR, respectively. Meanwhile,

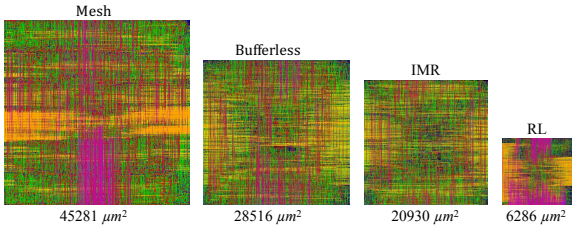


Figure 14: Area comparison under 15nm technology.

static power is mostly consumed by buffers. Unlike Mesh, EVC and IMR, RL has a much lower buffer requirement. As a result, RL consumes very low static power of 0.18mW on average, while Mesh, EVC and IMR consume 1.39mW, 1.64mW and 0.58mW, respectively. Adding dynamic and static power together, on average, RL reduces the total NoC power consumption by 9.48X, 13.1X and 7.75X over Mesh, EVC and IMR, respectively.

7.5 Area

Figure 14 compares the router or interface area of the different schemes we are studying. The results are obtained from Cadence Encounter after P&R[2]. We also add a bufferless design to the comparison. The largest area is 60731μm^2 for EVC (not shown in the figure) followed by 45281μm^2, 28516μm^2, 20930μm^2 and 6286μm^2 for Mesh, Bufferless, IMR and RL, respectively. The EXB and ejection link sharing techniques as well as the simplicity of the RL interface are the main contributors for the significant reduction of area overhead. Overall, RL has an area saving of 89.6%, 86.1%, 77.9% and 69.9% compared with EVC, Mesh, Bufferless[3] and IMR, respectively.

The wiring area is not included as wires are spread throughout the metal layers and cannot be compared directly. We do acknowledge that IMR and RL use more wiring resources than other designs. RL uses a small percentage of middle metal layers for wires and, as a result, more repeaters are needed. The total area for all the link repeaters is 0.127mm^2 which is 4.3% of the mesh router area. However, as middle layers are above the logic area, RL is unlikely to increase the chip size.

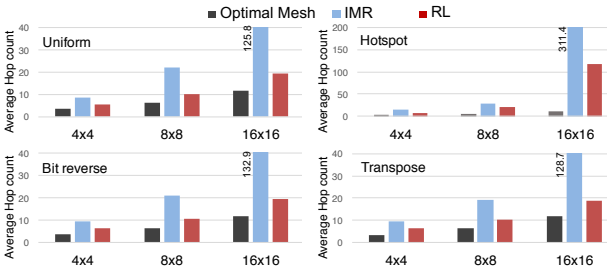


Figure 15: Average hop count for synthetic workloads.

8. DISCUSSION

8.1 Scalability and Regularity

[2]Our CAD tools limit P&R for processing cores.

[3]In addition to area reduction, RL also has 2.8X higher throughput (under UR) and 64.3% lower latency than bufferless NoC.

Table 3: Average overlapping and loops/rings in RL/IMR

Network	Overlap cap	Avg overlap(%) of links	Max loops/ rings in node	Avg loops/ rings(%) in node	Longest loop/ring
RL 4x4	4	3.33 (83.3%)	6	5 (62%)	12
IMR 4x4	4	2.33 (58.3%)	4	3.5 (43%)	14
RL 8x8	8	6 (75%)	14	10.5 (65%)	28
IMR 8x8	8	4.71 (58.9%)	10	8.2 (54%)	48
RL 16x16	16	11.33 (70.8%)	30	21.2 (66%)	60
IMR 16x16	16	8.13 (50.8%)	18	15.2 (47%)	240

Figures 11 and 12 already showed the advantage of RL in terms of latency and throughput for large networks. Figure 15 further compares the average hop count (zero-load hop count) of RL, IMR, and optimal Mesh. As can be seen, IMR has very high average hop count because of its lengthy rings. In contrast, the average hop count of RL is only slightly higher than optimal Mesh. Note that RL achieves this low hop count without having the switch capability of conventional routers.

Routerless NoC is not as irregular as it appears in the figures. In our actual design and evaluation, all the RL interfaces use the same design (some ports are left unused if no loops are connected), so the main irregularity is the way that links form loops. One way to quantify the degree of link irregularity is how many different possible lengths of links, which is $n-1$ for $n \times n$ NoC. This degree is similar to that of Flattened Butterfly [25] and MECS [18].

8.2 Average Overlapping

We discussed before that as long as the overlapping cap is met, it is beneficial to approach this cap for as many neighboring node pairs as possible to increase resource utilization and improve performance. Table 3 presents this statistics for RL and IMR. It can be seen that the average overlapping between adjacent nodes in RL is at least 20% more than that of IMR. Also, the longest loop in RL is always shorter than the longest ring in IMR, and the difference increases as the NoC gets bigger. Shorter loops reduce average hop count and offer a lower latency. For example, in 16×16 the longest loop in RL is of 60 nodes while in IMR it is of 240 nodes.

8.3 Impact on Latency distribution

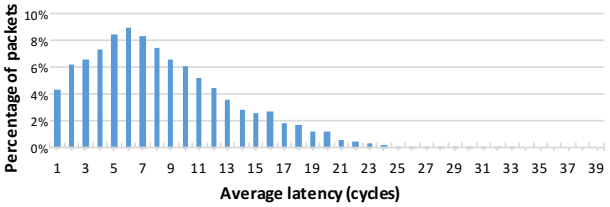


Figure 16: Latency distribution of benchmarks for RL 8×8 NoC.

The extension buffer technique and the reduced ejection link technique save buffer resources at the risk of increasing packet latency. Figure 16 shows distribution of average packet latency, averaged over different benchmarks. The RL interface is configured the same as previous sections with one EXB and two ejectors. The take away message from the figure is that the two techniques has minimal impact on latency under tight resource allocation. For example, the average packet latency is only 8.3 cycles for RL, and only 0.71% of the packets having latency larger than 20 cycles, with the

largest being 39 cycles. The tail in the latency distribution is thin and short.

8.4 RL for n x m Chip

The RL design can be easily extended to any $n \times m$ network sizes. The RL interface design and functionalities remain unchanged. The RLrec algorithm needs to be modified slightly. With rectangular shapes instead of squares, N_L and N_H are not sufficient to denote the four corners of a layer. Two more variables are needed to specify the corners of a layer correctly. For Instance, $N_L r$ and $N_H r$ for low and high rows, and $N_L c$ and $N_H c$ for low and high columns. Once a layer is correctly specified, the four groups of loops can be generated in similar fashion. The rotation step is skipped as this is not possible for rectangular networks, but the reversing direction step remains. The overlapping calculation needs to reflect the orientation of the rectangular loops as well.

9. CONCLUSION

Current and future many-core processors demand highly efficient on-chip networks to connect hundreds or even thousands of processing cores. In this paper, we analyze on-chip wiring resources in detail, and propose a novel routerless NoC design to remove the costly routers in conventional NoCs while still achieving scalable performance. We also propose an efficient interface hardware implementation, and evaluate the proposed scheme extensively. Simulation results show that the proposed routerless NoC design offers significant advantage in latency, throughput, power and area, compared with other designs. These results demonstrate the viability and potential benefits of the routerless approach, and also call for future works that continue to improve various aspects of routerless NoCs such as performance, reliability, and power efficiency.

Acknowledgments

We sincerely thank the anonymous reviewers for their helpful comments and suggestions. We appreciate the authors of IMR [28] for sharing the source code of generating IMR. We also thank Timothy M. Pinkston for providing valuable feedback to the work. This research was supported, in part, by the National Science Foundation (NSF) grants #1619456, #1566637, #1423656, #1619472 and #1321131.

10. REFERENCES

[1] http://ark.intel.com/products/95830/intel-xeon-phi-processor-7290-16gb-1_50-ghz-72-core/.
[2] https://oeis.org/A140517.
[3] http://wccftech.com/intel-sc15-knights-landing-14nm-wafer-specification/.
[4] T. W. Ainsworth and T. M. Pinkston, "On characterizing performance of the cell broadband engine element interconnect bus," in *International Symposium on Networks-on-Chip (NOCS)*, 2007.
[5] R. Arunachalam, E. Acar, and S. R. Nassif, "Optimal shielding/spacing metrics for low power design," in *IEEE Annual Symposium on VLSI*, 2003.
[6] M. Badr and N. E. Jerger, "Synfull: synthetic traffic models capturing cache coherent behaviour," in *ISCA*, 2014.
[7] L. A. Barroso and M. Dubois, "The performance of cache-coherent ring-based multiprocessors," in *ISCA*, 1993.
[8] ——, "Cache coherence on a slotted ring." in *ICPP*, 1991.
[9] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas, "A 5.8 pj/op 115 billion ops/sec, to 1.78 trillion ops/sec 32nm 1000-processor array," in *Symposium on VLSI Circuits*, 2016.
[10] L. Chen and T. M. Pinkston, "Nord: Node-router decoupling for effective power-gating of on-chip routers," in *MICRO*, 2012.
[11] B. Christian, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.
[12] I. Cutress, "Supercomputing 15: Intel's knights landing xeon phi silicon on display," November 2015.
[13] W. J. Dally and B. Towles, "Route packets, not wires: on-chip interconnection networks," in *DAC*, 2001.
[14] G. S. Delp, D. J. Farber, R. G. Minnich, J. M. Smith, and M. C. Tam, "Memory as a network abstraction," *IEEE Network*, vol. 5, no. 4, 1991.
[15] C. Fallin, C. Craik, and O. Mutlu, "Chipper: A low-complexity bufferless deflection router," in *HPCA*, 2011.
[16] C. Fallin, X. Yu, G. Nazario, and O. Mutlu, "A high-performance hierarchical ring on-chip interconnect with low-cost routers," 2011.
[17] P. Gratz, C. Kim, R. McDonald, S. W. Keckler, and D. Burger, "Implementation and evaluation of on-chip network architectures," in *International Conference on Computer Design*. IEEE, 2006.
[18] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu, "Express cube topologies for on-chip interconnects," in *HPCA*, 2009.
[19] D. Harris and N. Weste, *CMOS VLSI Design: A Circuits and Systems Perspective*. Pearson/Addison-Wesley, 2005.
[20] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar, "A 5-ghz mesh interconnect for a teraflops processor," *IEEE Micro*, 2007.
[21] J. Howard, S. Dighe, Y. Hoskote, S. Vangal *et al.*, "A 48-core ia-32 processor in 45 nm cmos using on-die message-passing and dvfs for performance and power scaling," *IEEE Journal of Solid-State Circuits*, 2011.
[22] C.-H. Jan, U. Bhattacharya, R. Brain, S.-J. Choi, G. Curello, G. Gupta, W. Hafez, M. Jang, M. Kang, K. Komeyli *et al.*, "A 22nm soc platform technology featuring 3-d tri-gate and high-k/metal gate, optimized for ultra low power, high performance and high density soc applications," in *Electron Devices Meeting (IEDM)*. IEEE, 2012.
[23] N. E. Jerger, L. S. Peh, and M. Lipasti, "Virtual circuit tree multicasting: A case for on-chip hardware multicast support," in *ISCA*, 2008.
[24] N. Jiang, J. Balfour, D. U. Becker, B. Towles, W. J. Dally, G. Michelogiannakis, and J. Kim, "A detailed and flexible cycle-accurate network-on-chip simulator," in *ISPASS*. IEEE, 2013.
[25] J. Kim, W. J. Dally, and D. Abts, "Flattened butterfly: A cost-efficient topology for high-radix networks," in *ISCA*, 2007.
[26] A. K. Kodi, A. Sarathy, and A. Louri, "ideal: Inter-router dual-function energy and area-efficient links for network-on-chip (noc) architectures," in *ISCA*, 2008.
[27] J. Liu, L. R. Zheng, D. Pamunuwa, and H. Tenhunen, "A global wire planning scheme for network-on-chip," in *International Symposium on Circuits and Systems (ISCAS)*, 2003.
[28] S. Liu, T. Chen, L. Li, X. Feng, Z. Xu, H. Chen, F. Chong, and Y. Chen, "Imr: High-performance low-cost multi-ring nocs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 6, 2016.
[29] NanGate, Inc. Nangate freePDK15 open cell library. [Online]. Available: http://www.nangate.com
[30] S. Natarajan, M. Agostinelli, S. Akbar, M. Bost, A. Bowonder, V. Chikarmane, S. Chouksey, A. Dasgupta, K. Fischer, Q. Fu *et al.*, "A 14nm logic technology featuring 2 nd-generation finfet, air-gapped interconnects, self-aligned double patterning and a 0.0588 μm 2 sram cell size," in *IEEE International Electron Devices Meeting*, 2014.
[31] C. A. Nicopoulos, D. Park, J. Kim, N. Vijaykrishnan, M. S. Yousif, and C. R. Das, "Vichar: A dynamic virtual channel regulator for network-on-chip routers," in *MICRO*, 2006.
[32] D. Pamunuwa, J. Oberg, L. R. Zheng, M. Millberg, A. Jantsch, and H. Tenhunen, "Layout, performance and power trade-offs in mesh-based network-on-chip architectures," in *International Conference on Very Large Scale Integration*, 2003.
[33] M. K. Papamichael and J. C. Hoe, "The connect network-on-chip generator," *Computer*, vol. 48, no. 12, 2015.
[34] A. N. Udipi, N. Muralimanohar, and R. Balasubramonian, "Towards scalable, energy-efficient, bus-based on-chip networks," in *HPCA*, 2010.
[35] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob *et al.*, "An 80-tile 1.28 tflops network-on-chip in 65nm cmos," in *ISSCC*, 2007.
[36] L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, Eds., *Electronic Design Automation: Synthesis, Verification, and Test*. Morgan Kaufmann Publishers Inc., 2009.
[37] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. F. Brown III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, 2007.
[38] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "The potential of the cell processor for scientific computing," in *Computing frontiers*. ACM, 2006.
[39] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: Characterization and methodological considerations," in *ISCA*, 1995.

The 4 x 4 RL network Loops

Loop ID	Nodes
1	[0, 1, 5, 9, 13, 12, 8, 4]
2	[3, 7, 11, 15, 14, 10, 6, 2]
3	[0, 1, 2, 6, 10, 14, 13, 12, 8, 4]
4	[3, 7, 11, 15, 14, 13, 9, 5, 1, 2]
5	[0, 4, 8, 12, 13, 14, 15, 11, 7, 3, 2, 1]
6	[0, 1, 2, 3, 7, 6, 5, 4]
7	[4, 5, 6, 7, 11, 10, 9, 8]
8	[8, 9, 10, 11, 15, 14, 13, 12]
9	[5, 6, 10, 9]
10	[5, 9, 10, 6]

The 8 x 8 RL network Loops

Loop ID	Nodes
1	[0, 1, 9, 17, 25, 33, 41, 49, 57, 56, 48, 40, 32, 24, 16, 8]
2	[7, 15, 23, 31, 39, 47, 55, 63, 62, 54, 46, 38, 30, 22, 14, 6]
3	[0, 1, 2, 10, 18, 26, 34, 42, 50, 58, 57, 56, 48, 40, 32, 24, 16, 8]
4	[7, 15, 23, 31, 39, 47, 55, 63, 62, 61, 53, 45, 37, 29, 21, 13, 5, 6]
5	[0, 1, 2, 3, 11, 19, 27, 35, 43, 51, 59, 58, 57, 56, 48, 40, 32, 24, 16, 8]
6	[7, 15, 23, 31, 39, 47, 55, 63, 62, 61, 60, 52, 44, 36, 28, 20, 12, 4, 5, 6]
7	[0, 1, 2, 3, 4, 12, 20, 28, 36, 44, 52, 60, 59, 58, 57, 56, 48, 40, 32, 24, 16, 8]
8	[7, 15, 23, 31, 39, 47, 55, 63, 62, 61, 60, 59, 51, 43, 35, 27, 19, 11, 3, 4, 5, 6]
9	[0, 1, 2, 3, 4, 5, 13, 21, 29, 37, 45, 53, 61, 60, 59, 58, 57, 56, 48, 40, 32, 24, 16, 8]
10	[7, 15, 23, 31, 39, 47, 55, 63, 62, 61, 60, 59, 58, 50, 42, 34, 26, 18, 10, 2, 3, 4, 5, 6]
11	[0, 1, 2, 3, 4, 5, 6, 14, 22, 30, 38, 46, 54, 62, 61, 60, 59, 58, 57, 56, 48, 40, 32, 24, 16, 8]
12	[7, 15, 23, 31, 39, 47, 55, 63, 62, 61, 60, 59, 58, 57, 49, 41, 33, 25, 17, 9, 1, 2, 3, 4, 5, 6]
13	[0, 8, 16, 24, 32, 40, 48, 56, 57, 58, 59, 60, 61, 62, 63, 55, 47, 39, 31, 23, 15, 7, 6, 5, 4, 3, 2, 1]
14	[0, 1, 2, 3, 4, 5, 6, 7, 15, 14, 13, 12, 11, 10, 9, 8]
15	[8, 9, 10, 11, 12, 13, 14, 15, 23, 22, 21, 20, 19, 18, 17, 16]
16	[16, 17, 18, 19, 20, 21, 22, 23, 31, 30, 29, 28, 27, 26, 25, 24]
17	[24, 25, 26, 27, 28, 29, 30, 31, 39, 38, 37, 36, 35, 34, 33, 32]
18	[32, 33, 34, 35, 36, 37, 38, 39, 47, 46, 45, 44, 43, 42, 41, 40]
19	[40, 41, 42, 43, 44, 45, 46, 47, 55, 54, 53, 52, 51, 50, 49, 48]
20	[48, 49, 50, 51, 52, 53, 54, 55, 63, 62, 61, 60, 59, 58, 57, 56]
21	[9, 17, 18, 19, 20, 21, 22, 14, 13, 12, 11, 10]
22	[49, 50, 51, 52, 53, 54, 46, 45, 44, 43, 42, 41]
23	[9, 17, 25, 26, 27, 28, 29, 30, 22, 14, 13, 12, 11, 10]
24	[49, 50, 51, 52, 53, 54, 46, 38, 37, 36, 35, 34, 33, 41]
25	[9, 17, 25, 33, 34, 35, 36, 37, 38, 30, 22, 14, 13, 12, 11, 10]
26	[49, 50, 51, 52, 53, 54, 46, 38, 30, 29, 28, 27, 26, 25, 33, 41]
27	[9, 17, 25, 33, 41, 42, 43, 44, 45, 46, 38, 30, 22, 14, 13, 12, 11, 10]
28	[49, 50, 51, 52, 53, 54, 46, 38, 30, 22, 21, 20, 19, 18, 17, 25, 33, 41]
29	[9, 10, 11, 12, 13, 14, 22, 30, 38, 46, 54, 53, 52, 51, 50, 49, 41, 33, 25, 17]
30	[9, 17, 25, 33, 41, 49, 50, 42, 34, 26, 18, 10]
31	[10, 18, 26, 34, 42, 50, 51, 43, 35, 27, 19, 11]

Loop ID	Nodes
32	[11, 19, 27, 35, 43, 51, 52, 44, 36, 28, 20, 12]
33	[12, 20, 28, 36, 44, 52, 53, 45, 37, 29, 21, 13]
34	[13, 21, 29, 37, 45, 53, 54, 46, 38, 30, 22, 14]
35	[18, 19, 27, 35, 43, 42, 34, 26]
36	[21, 29, 37, 45, 44, 36, 28, 20]
37	[18, 19, 20, 28, 36, 44, 43, 42, 34, 26]
38	[21, 29, 37, 45, 44, 43, 35, 27, 19, 20]
39	[18, 26, 34, 42, 43, 44, 45, 37, 29, 21, 20, 19]
40	[18, 19, 20, 21, 29, 28, 27, 26]
41	[26, 27, 28, 29, 37, 36, 35, 34]
42	[34, 35, 36, 37, 45, 44, 43, 42]
43	[27, 28, 36, 35]
44	[27, 35, 36, 28]

The 16 x 16 RL network Loops

Loop ID	Nodes
1	[0, 1, 17, 33, 49, 65, 81, 97, 113, 129, 145, 161, 177, 193, 209, 225, 241, 240, 224, 208, 192, 176, 160, 144, 128, 112, 96, 80, 64, 48, 32, 16]
2	[15, 31, 47, 63, 79, 95, 111, 127, 143, 159, 175, 191, 207, 223, 239, 255, 254, 238, 222, 206, 190, 174, 158, 142, 126, 110, 94, 78, 62, 46, 30, 14]
3	[0, 1, 2, 18, 34, 50, 66, 82, 98, 114, 130, 146, 162, 178, 194, 210, 226, 242, 241, 240, 224, 208, 192, 176, 160, 144, 128, 112, 96, 80, 64, 48, 32, 16]
4	[15, 31, 47, 63, 79, 95, 111, 127, 143, 159, 175, 191, 207, 223, 239, 255, 254, 253, 237, 221, 205, 189, 173, 157, 141, 125, 109, 93, 77, 61, 45, 29, 13, 14]
5	[0, 1, 2, 3, 19, 35, 51, 67, 83, 99, 115, 131, 147, 163, 179, 195, 211, 227, 243, 242, 241, 240, 224, 208, 192, 176, 160, 144, 128, 112, 96, 80, 64, 48, 32, 16]
6	[15, 31, 47, 63, 79, 95, 111, 127, 143, 159, 175, 191, 207, 223, 239, 255, 254, 253, 252, 236, 220, 204, 188, 172, 156, 140, 124, 108, 92, 76, 60, 44, 28, 12, 13, 14]
7	[0, 1, 2, 3, 4, 20, 36, 52, 68, 84, 100, 116, 132, 148, 164, 180, 196, 212, 228, 244, 243, 242, 241, 240, 224, 208, 192, 176, 160, 144, 128, 112, 96, 80, 64, 48, 32, 16]
8	[15, 31, 47, 63, 79, 95, 111, 127, 143, 159, 175, 191, 207, 223, 239, 255, 254, 253, 252, 251, 235, 219, 203, 187, 171, 155, 139, 123, 107, 91, 75, 59, 43, 27, 11, 12, 13, 14]
9	[0, 1, 2, 3, 4, 5, 21, 37, 53, 69, 85, 101, 117, 133, 149, 165, 181, 197, 213, 229, 245, 244, 243, 242, 241, 240, 224, 208, 192, 176, 160, 144, 128, 112, 96, 80, 64, 48, 32, 16]
10	[15, 31, 47, 63, 79, 95, 111, 127, 143, 159, 175, 191, 207, 223, 239, 255, 254, 253, 252, 251, 250, 234, 218, 202, 186, 170, 154, 138, 122, 106, 90, 74, 58, 42, 26, 10, 11, 12, 13, 14]
11	[0, 1, 2, 3, 4, 5, 6, 22, 38, 54, 70, 86, 102, 118, 134, 150, 166, 182, 198, 214, 230, 246, 245, 244, 243, 242, 241, 240, 224, 208, 192, 176, 160, 144, 128, 112, 96, 80, 64, 48, 32, 16]
12	[15, 31, 47, 63, 79, 95, 111, 127, 143, 159, 175, 191, 207, 223, 239, 255, 254, 253, 252, 251, 250, 249, 233, 217, 201, 185, 169, 153, 137, 121, 105, 89, 73, 57, 41, 25, 9, 10, 11, 12, 13, 14]
13	[0, 1, 2, 3, 4, 5, 6, 7, 23, 39, 55, 71, 87, 103, 119, 135, 151, 167, 183, 199, 215, 231, 247, 246, 245, 244, 243, 242, 241, 240, 224, 208, 192, 176, 160, 144, 128, 112, 96, 80, 64, 48, 32, 16]
14	[15, 31, 47, 63, 79, 95, 111, 127, 143, 159, 175, 191, 207, 223, 239, 255, 254, 253, 252, 251, 250, 249, 248, 232, 216, 200, 184, 168, 152, 136, 120, 104, 88, 72, 56, 40, 24, 8, 9, 10, 11, 12, 13, 14]
15	[0, 1, 2, 3, 4, 5, 6, 7, 8, 24, 40, 56, 72, 88, 104, 120, 136, 152, 168, 184, 200, 216, 232, 248, 247, 246, 245, 244, 243, 242, 241, 240, 224, 208, 192, 176, 160, 144, 128, 112, 96, 80, 64, 48, 32, 16]
16	[15, 31, 47, 63, 79, 95, 111, 127, 143, 159, 175, 191, 207, 223, 239, 255, 254, 253, 252, 251, 250, 249, 248, 247, 231, 215, 199, 183, 167, 151, 135, 119, 103, 87, 71, 55, 39, 23, 7, 8, 9, 10, 11, 12, 13, 14]
17	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 25, 41, 57, 73, 89, 105, 121, 137, 153, 169, 185, 201, 217, 233, 249, 248, 247, 246, 245, 244, 243, 242, 241, 240, 224, 208, 192, 176, 160, 144, 128, 112, 96, 80, 64, 48, 32, 16]
18	[15, 31, 47, 63, 79, 95, 111, 127, 143, 159, 175, 191, 207, 223, 239, 255, 254, 253, 252, 251, 250, 249, 248, 247, 246, 230, 214, 198, 182, 166, 150, 134, 118, 102, 86, 70, 54, 38, 22, 6, 7, 8, 9, 10, 11, 12, 13, 14]
19	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 26, 42, 58, 74, 90, 106, 122, 138, 154, 170, 186, 202, 218, 234, 250, 249, 248, 247, 246, 245, 244, 243, 242, 241, 240, 224, 208, 192, 176, 160, 144, 128, 112, 96, 80, 64, 48, 32, 16]
20	[15, 31, 47, 63, 79, 95, 111, 127, 143, 159, 175, 191, 207, 223, 239, 255, 254, 253, 252, 251, 250, 249, 248, 247, 246, 245, 229, 213, 197, 181, 165, 149, 133, 117, 101, 85, 69, 53, 37, 21, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]

Loop ID	Nodes
21	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 27, 43, 59, 75, 91, 107, 123, 139, 155, 171, 187, 203, 219, 235, 251, 250, 249, 248, 247, 246, 245, 244, 243, 242, 241, 240, 224, 208, 192, 176, 160, 144, 128, 112, 96, 80, 64, 48, 32, 16]
22	[15, 31, 47, 63, 79, 95, 111, 127, 143, 159, 175, 191, 207, 223, 239, 255, 254, 253, 252, 251, 250, 249, 248, 247, 246, 245, 244, 228, 212, 196, 180, 164, 148, 132, 116, 100, 84, 68, 52, 36, 20, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
23	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 28, 44, 60, 76, 92, 108, 124, 140, 156, 172, 188, 204, 220, 236, 252, 251, 250, 249, 248, 247, 246, 245, 244, 243, 242, 241, 240, 224, 208, 192, 176, 160, 144, 128, 112, 96, 80, 64, 48, 32, 16]
24	[15, 31, 47, 63, 79, 95, 111, 127, 143, 159, 175, 191, 207, 223, 239, 255, 254, 253, 252, 251, 250, 249, 248, 247, 246, 245, 244, 243, 227, 211, 195, 179, 163, 147, 131, 115, 99, 83, 67, 51, 35, 19, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
25	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 29, 45, 61, 77, 93, 109, 125, 141, 157, 173, 189, 205, 221, 237, 253, 252, 251, 250, 249, 248, 247, 246, 245, 244, 243, 242, 241, 240, 224, 208, 192, 176, 160, 144, 128, 112, 96, 80, 64, 48, 32, 16]
26	[15, 31, 47, 63, 79, 95, 111, 127, 143, 159, 175, 191, 207, 223, 239, 255, 254, 253, 252, 251, 250, 249, 248, 247, 246, 245, 244, 243, 242, 226, 210, 194, 178, 162, 146, 130, 114, 98, 82, 66, 50, 34, 18, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
27	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 30, 46, 62, 78, 94, 110, 126, 142, 158, 174, 190, 206, 222, 238, 254, 253, 252, 251, 250, 249, 248, 247, 246, 245, 244, 243, 242, 241, 240, 224, 208, 192, 176, 160, 144, 128, 112, 96, 80, 64, 48, 32, 16]
28	[15, 31, 47, 63, 79, 95, 111, 127, 143, 159, 175, 191, 207, 223, 239, 255, 254, 253, 252, 251, 250, 249, 248, 247, 246, 245, 244, 243, 242, 241, 225, 209, 193, 177, 161, 145, 129, 113, 97, 81, 65, 49, 33, 17, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
29	[0, 16, 32, 48, 64, 80, 96, 112, 128, 144, 160, 176, 192, 208, 224, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 239, 223, 207, 191, 175, 159, 143, 127, 111, 95, 79, 63, 47, 31, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
30	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16]
31	[16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 47, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33, 32]
32	[32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 63, 62, 61, 60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50, 49, 48]
33	[48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 79, 78, 77, 76, 75, 74, 73, 72, 71, 70, 69, 68, 67, 66, 65, 64]
34	[64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 95, 94, 93, 92, 91, 90, 89, 88, 87, 86, 85, 84, 83, 82, 81, 80]
35	[80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 111, 110, 109, 108, 107, 106, 105, 104, 103, 102, 101, 100, 99, 98, 97, 96]
36	[96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 127, 126, 125, 124, 123, 122, 121, 120, 119, 118, 117, 116, 115, 114, 113, 112]
37	[112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 143, 142, 141, 140, 139, 138, 137, 136, 135, 134, 133, 132, 131, 130, 129, 128]
38	[128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 159, 158, 157, 156, 155, 154, 153, 152, 151, 150, 149, 148, 147, 146, 145, 144]

Loop ID	Nodes
39	[144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 175, 174, 173, 172, 171, 170, 169, 168, 167, 166, 165, 164, 163, 162, 161, 160]
40	[160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 175, 191, 190, 189, 188, 187, 186, 185, 184, 183, 182, 181, 180, 179, 178, 177, 176]
41	[176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 207, 206, 205, 204, 203, 202, 201, 200, 199, 198, 197, 196, 195, 194, 193, 192]
42	[192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 223, 222, 221, 220, 219, 218, 217, 216, 215, 214, 213, 212, 211, 210, 209, 208]
43	[208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 239, 238, 237, 236, 235, 234, 233, 232, 231, 230, 229, 228, 227, 226, 225, 224]
44	[224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239, 255, 254, 253, 252, 251, 250, 249, 248, 247, 246, 245, 244, 243, 242, 241, 240]
45	[17, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18]
46	[225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 222, 221, 220, 219, 218, 217, 216, 215, 214, 213, 212, 211, 210, 209]
47	[17, 33, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 46, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18]
48	[225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 222, 206, 205, 204, 203, 202, 201, 200, 199, 198, 197, 196, 195, 194, 193, 209]
49	[17, 33, 49, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 62, 46, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18]
50	[225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 222, 206, 190, 189, 188, 187, 186, 185, 184, 183, 182, 181, 180, 179, 178, 177, 193, 209]
51	[17, 33, 49, 65, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 78, 62, 46, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18]
52	[225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 222, 206, 190, 174, 173, 172, 171, 170, 169, 168, 167, 166, 165, 164, 163, 162, 161, 177, 193, 209]
53	[17, 33, 49, 65, 81, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 94, 78, 62, 46, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18]
54	[225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 222, 206, 190, 174, 158, 157, 156, 155, 154, 153, 152, 151, 150, 149, 148, 147, 146, 145, 161, 177, 193, 209]
55	[17, 33, 49, 65, 81, 97, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 110, 94, 78, 62, 46, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18]
56	[225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 222, 206, 190, 174, 158, 142, 141, 140, 139, 138, 137, 136, 135, 134, 133, 132, 131, 130, 129, 145, 161, 177, 193, 209]
57	[17, 33, 49, 65, 81, 97, 113, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 126, 110, 94, 78, 62, 46, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18]
58	[225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 222, 206, 190, 174, 158, 142, 126, 125, 124, 123, 122, 121, 120, 119, 118, 117, 116, 115, 114, 113, 129, 145, 161, 177, 193, 209]
59	[17, 33, 49, 65, 81, 97, 113, 129, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 142, 126, 110, 94, 78, 62, 46, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18]

Loop ID	Nodes
60	[225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 222, 206, 190, 174, 158, 142, 126, 110, 109, 108, 107, 106, 105, 104, 103, 102, 101, 100, 99, 98, 97, 113, 129, 145, 161, 177, 193, 209]
61	[17, 33, 49, 65, 81, 97, 113, 129, 145, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 174, 158, 142, 126, 110, 94, 78, 62, 46, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18]
62	[225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 222, 206, 190, 174, 158, 142, 126, 110, 94, 93, 92, 91, 90, 89, 88, 87, 86, 85, 84, 83, 82, 81, 97, 113, 129, 145, 161, 177, 193, 209]
63	[17, 33, 49, 65, 81, 97, 113, 129, 145, 161, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 174, 158, 142, 126, 110, 94, 78, 62, 46, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18]
64	[225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 222, 206, 190, 174, 158, 142, 126, 110, 94, 78, 77, 76, 75, 74, 73, 72, 71, 70, 69, 68, 67, 66, 65, 81, 97, 113, 129, 145, 161, 177, 193, 209]
65	[17, 33, 49, 65, 81, 97, 113, 129, 145, 161, 177, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 190, 174, 158, 142, 126, 110, 94, 78, 62, 46, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18]
66	[225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 222, 206, 190, 174, 158, 142, 126, 110, 94, 78, 62, 61, 60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50, 49, 65, 81, 97, 113, 129, 145, 161, 177, 193, 209]
67	[17, 33, 49, 65, 81, 97, 113, 129, 145, 161, 177, 193, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 206, 190, 174, 158, 142, 126, 110, 94, 78, 62, 46, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18]
68	[225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 222, 206, 190, 174, 158, 142, 126, 110, 94, 78, 62, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33, 49, 65, 81, 97, 113, 129, 145, 161, 177, 193, 209]
69	[17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 46, 62, 78, 94, 110, 126, 142, 158, 174, 190, 206, 222, 238, 237, 236, 235, 234, 233, 232, 231, 230, 229, 228, 227, 226, 225, 209, 193, 177, 161, 145, 129, 113, 97, 81, 65, 49, 33]
70	[17, 33, 49, 65, 81, 97, 113, 129, 145, 161, 177, 193, 209, 225, 226, 210, 194, 178, 162, 146, 130, 114, 98, 82, 66, 50, 34, 18]
71	[18, 34, 50, 66, 82, 98, 114, 130, 146, 162, 178, 194, 210, 226, 227, 211, 195, 179, 163, 147, 131, 115, 99, 83, 67, 51, 35, 19]
72	[19, 35, 51, 67, 83, 99, 115, 131, 147, 163, 179, 195, 211, 227, 228, 212, 196, 180, 164, 148, 132, 116, 100, 84, 68, 52, 36, 20]
73	[20, 36, 52, 68, 84, 100, 116, 132, 148, 164, 180, 196, 212, 228, 229, 213, 197, 181, 165, 149, 133, 117, 101, 85, 69, 53, 37, 21]
74	[21, 37, 53, 69, 85, 101, 117, 133, 149, 165, 181, 197, 213, 229, 230, 214, 198, 182, 166, 150, 134, 118, 102, 86, 70, 54, 38, 22]
75	[22, 38, 54, 70, 86, 102, 118, 134, 150, 166, 182, 198, 214, 230, 231, 215, 199, 183, 167, 151, 135, 119, 103, 87, 71, 55, 39, 23]
76	[23, 39, 55, 71, 87, 103, 119, 135, 151, 167, 183, 199, 215, 231, 232, 216, 200, 184, 168, 152, 136, 120, 104, 88, 72, 56, 40, 24]
77	[24, 40, 56, 72, 88, 104, 120, 136, 152, 168, 184, 200, 216, 232, 233, 217, 201, 185, 169, 153, 137, 121, 105, 89, 73, 57, 41, 25]
78	[25, 41, 57, 73, 89, 105, 121, 137, 153, 169, 185, 201, 217, 233, 234, 218, 202, 186, 170, 154, 138, 122, 106, 90, 74, 58, 42, 26]

Loop ID	Nodes
79	[26, 42, 58, 74, 90, 106, 122, 138, 154, 170, 186, 202, 218, 234, 235, 219, 203, 187, 171, 155, 139, 123, 107, 91, 75, 59, 43, 27]
80	[27, 43, 59, 75, 91, 107, 123, 139, 155, 171, 187, 203, 219, 235, 236, 220, 204, 188, 172, 156, 140, 124, 108, 92, 76, 60, 44, 28]
81	[28, 44, 60, 76, 92, 108, 124, 140, 156, 172, 188, 204, 220, 236, 237, 221, 205, 189, 173, 157, 141, 125, 109, 93, 77, 61, 45, 29]
82	[29, 45, 61, 77, 93, 109, 125, 141, 157, 173, 189, 205, 221, 237, 238, 222, 206, 190, 174, 158, 142, 126, 110, 94, 78, 62, 46, 30]
83	[34, 35, 51, 67, 83, 99, 115, 131, 147, 163, 179, 195, 211, 210, 194, 178, 162, 146, 130, 114, 98, 82, 66, 50]
84	[45, 61, 77, 93, 109, 125, 141, 157, 173, 189, 205, 221, 220, 204, 188, 172, 156, 140, 124, 108, 92, 76, 60, 44]
85	[34, 35, 36, 52, 68, 84, 100, 116, 132, 148, 164, 180, 196, 212, 211, 210, 194, 178, 162, 146, 130, 114, 98, 82, 66, 50]
86	[45, 61, 77, 93, 109, 125, 141, 157, 173, 189, 205, 221, 220, 219, 203, 187, 171, 155, 139, 123, 107, 91, 75, 59, 43, 44]
87	[34, 35, 36, 37, 53, 69, 85, 101, 117, 133, 149, 165, 181, 197, 213, 212, 211, 210, 194, 178, 162, 146, 130, 114, 98, 82, 66, 50]
88	[45, 61, 77, 93, 109, 125, 141, 157, 173, 189, 205, 221, 220, 219, 218, 202, 186, 170, 154, 138, 122, 106, 90, 74, 58, 42, 43, 44]
89	[34, 35, 36, 37, 38, 54, 70, 86, 102, 118, 134, 150, 166, 182, 198, 214, 213, 212, 211, 210, 194, 178, 162, 146, 130, 114, 98, 82, 66, 50]
90	[45, 61, 77, 93, 109, 125, 141, 157, 173, 189, 205, 221, 220, 219, 218, 217, 201, 185, 169, 153, 137, 121, 105, 89, 73, 57, 41, 42, 43, 44]
91	[34, 35, 36, 37, 38, 39, 55, 71, 87, 103, 119, 135, 151, 167, 183, 199, 215, 214, 213, 212, 211, 210, 194, 178, 162, 146, 130, 114, 98, 82, 66, 50]
92	[45, 61, 77, 93, 109, 125, 141, 157, 173, 189, 205, 221, 220, 219, 218, 217, 216, 200, 184, 168, 152, 136, 120, 104, 88, 72, 56, 40, 41, 42, 43, 44]
93	[34, 35, 36, 37, 38, 39, 40, 56, 72, 88, 104, 120, 136, 152, 168, 184, 200, 216, 215, 214, 213, 212, 211, 210, 194, 178, 162, 146, 130, 114, 98, 82, 66, 50]
94	[45, 61, 77, 93, 109, 125, 141, 157, 173, 189, 205, 221, 220, 219, 218, 217, 216, 215, 199, 183, 167, 151, 135, 119, 103, 87, 71, 55, 39, 40, 41, 42, 43, 44]
95	[34, 35, 36, 37, 38, 39, 40, 41, 57, 73, 89, 105, 121, 137, 153, 169, 185, 201, 217, 216, 215, 214, 213, 212, 211, 210, 194, 178, 162, 146, 130, 114, 98, 82, 66, 50]
96	[45, 61, 77, 93, 109, 125, 141, 157, 173, 189, 205, 221, 220, 219, 218, 217, 216, 215, 214, 198, 182, 166, 150, 134, 118, 102, 86, 70, 54, 38, 39, 40, 41, 42, 43, 44]
97	[34, 35, 36, 37, 38, 39, 40, 41, 42, 58, 74, 90, 106, 122, 138, 154, 170, 186, 202, 218, 217, 216, 215, 214, 213, 212, 211, 210, 194, 178, 162, 146, 130, 114, 98, 82, 66, 50]
98	[45, 61, 77, 93, 109, 125, 141, 157, 173, 189, 205, 221, 220, 219, 218, 217, 216, 215, 214, 213, 197, 181, 165, 149, 133, 117, 101, 85, 69, 53, 37, 38, 39, 40, 41, 42, 43, 44]
99	[34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 59, 75, 91, 107, 123, 139, 155, 171, 187, 203, 219, 218, 217, 216, 215, 214, 213, 212, 211, 210, 194, 178, 162, 146, 130, 114, 98, 82, 66, 50]

Loop ID	Nodes
100	[45, 61, 77, 93, 109, 125, 141, 157, 173, 189, 205, 221, 220, 219, 218, 217, 216, 215, 214, 213, 212, 196, 180, 164, 148, 132, 116, 100, 84, 68, 52, 36, 37, 38, 39, 40, 41, 42, 43, 44]
101	[34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 60, 76, 92, 108, 124, 140, 156, 172, 188, 204, 220, 219, 218, 217, 216, 215, 214, 213, 212, 211, 210, 194, 178, 162, 146, 130, 114, 98, 82, 66, 50]
102	[45, 61, 77, 93, 109, 125, 141, 157, 173, 189, 205, 221, 220, 219, 218, 217, 216, 215, 214, 213, 212, 211, 195, 179, 163, 147, 131, 115, 99, 83, 67, 51, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44]
103	[34, 50, 66, 82, 98, 114, 130, 146, 162, 178, 194, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 205, 189, 173, 157, 141, 125, 109, 93, 77, 61, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35]
104	[34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 61, 60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50]
105	[50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 77, 76, 75, 74, 73, 72, 71, 70, 69, 68, 67, 66]
106	[66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 93, 92, 91, 90, 89, 88, 87, 86, 85, 84, 83, 82]
107	[82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 109, 108, 107, 106, 105, 104, 103, 102, 101, 100, 99, 98]
108	[98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 125, 124, 123, 122, 121, 120, 119, 118, 117, 116, 115, 114]
109	[114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 141, 140, 139, 138, 137, 136, 135, 134, 133, 132, 131, 130]
110	[130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 157, 156, 155, 154, 153, 152, 151, 150, 149, 148, 147, 146]
111	[146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 173, 172, 171, 170, 169, 168, 167, 166, 165, 164, 163, 162]
112	[162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 189, 188, 187, 186, 185, 184, 183, 182, 181, 180, 179, 178]
113	[178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 205, 204, 203, 202, 201, 200, 199, 198, 197, 196, 195, 194]
114	[194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 221, 220, 219, 218, 217, 216, 215, 214, 213, 212, 211, 210]
115	[51, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 60, 59, 58, 57, 56, 55, 54, 53, 52]
116	[195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 188, 187, 186, 185, 184, 183, 182, 181, 180, 179]
117	[51, 67, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 76, 60, 59, 58, 57, 56, 55, 54, 53, 52]
118	[195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 188, 172, 171, 170, 169, 168, 167, 166, 165, 164, 163, 179]
119	[51, 67, 83, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 92, 76, 60, 59, 58, 57, 56, 55, 54, 53, 52]
120	[195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 188, 172, 156, 155, 154, 153, 152, 151, 150, 149, 148, 147, 163, 179]
121	[51, 67, 83, 99, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 108, 92, 76, 60, 59, 58, 57, 56, 55, 54, 53, 52]
122	[195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 188, 172, 156, 140, 139, 138, 137, 136, 135, 134, 133, 132, 131, 147, 163, 179]
123	[51, 67, 83, 99, 115, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 124, 108, 92, 76, 60, 59, 58, 57, 56, 55, 54, 53, 52]

Loop ID	Nodes
124	[195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 188, 172, 156, 140, 124, 123, 122, 121, 120, 119, 118, 117, 116, 115, 131, 147, 163, 179]
125	[51, 67, 83, 99, 115, 131, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 140, 124, 108, 92, 76, 60, 59, 58, 57, 56, 55, 54, 53, 52]
126	[195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 188, 172, 156, 140, 124, 108, 107, 106, 105, 104, 103, 102, 101, 100, 99, 115, 131, 147, 163, 179]
127	[51, 67, 83, 99, 115, 131, 147, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 156, 140, 124, 108, 92, 76, 60, 59, 58, 57, 56, 55, 54, 53, 52]
128	[195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 188, 172, 156, 140, 124, 108, 92, 91, 90, 89, 88, 87, 86, 85, 84, 83, 99, 115, 131, 147, 163, 179]
129	[51, 67, 83, 99, 115, 131, 147, 163, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 172, 156, 140, 124, 108, 92, 76, 60, 59, 58, 57, 56, 55, 54, 53, 52]
130	[195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 188, 172, 156, 140, 124, 108, 92, 76, 75, 74, 73, 72, 71, 70, 69, 68, 67, 83, 99, 115, 131, 147, 163, 179]
131	[51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 76, 92, 108, 124, 140, 156, 172, 188, 204, 203, 202, 201, 200, 199, 198, 197, 196, 195, 179, 163, 147, 131, 115, 99, 83, 67]
132	[51, 67, 83, 99, 115, 131, 147, 163, 179, 195, 196, 180, 164, 148, 132, 116, 100, 84, 68, 52]
133	[52, 68, 84, 100, 116, 132, 148, 164, 180, 196, 197, 181, 165, 149, 133, 117, 101, 85, 69, 53]
134	[53, 69, 85, 101, 117, 133, 149, 165, 181, 197, 198, 182, 166, 150, 134, 118, 102, 86, 70, 54]
135	[54, 70, 86, 102, 118, 134, 150, 166, 182, 198, 199, 183, 167, 151, 135, 119, 103, 87, 71, 55]
136	[55, 71, 87, 103, 119, 135, 151, 167, 183, 199, 200, 184, 168, 152, 136, 120, 104, 88, 72, 56]
137	[56, 72, 88, 104, 120, 136, 152, 168, 184, 200, 201, 185, 169, 153, 137, 121, 105, 89, 73, 57]
138	[57, 73, 89, 105, 121, 137, 153, 169, 185, 201, 202, 186, 170, 154, 138, 122, 106, 90, 74, 58]
139	[58, 74, 90, 106, 122, 138, 154, 170, 186, 202, 203, 187, 171, 155, 139, 123, 107, 91, 75, 59]
140	[59, 75, 91, 107, 123, 139, 155, 171, 187, 203, 204, 188, 172, 156, 140, 124, 108, 92, 76, 60]
141	[68, 69, 85, 101, 117, 133, 149, 165, 181, 180, 164, 148, 132, 116, 100, 84]
142	[75, 91, 107, 123, 139, 155, 171, 187, 186, 170, 154, 138, 122, 106, 90, 74]
143	[68, 69, 70, 86, 102, 118, 134, 150, 166, 182, 181, 180, 164, 148, 132, 116, 100, 84]
144	[75, 91, 107, 123, 139, 155, 171, 187, 186, 185, 169, 153, 137, 121, 105, 89, 73, 74]
145	[68, 69, 70, 71, 87, 103, 119, 135, 151, 167, 183, 182, 181, 180, 164, 148, 132, 116, 100, 84]
146	[75, 91, 107, 123, 139, 155, 171, 187, 186, 185, 184, 168, 152, 136, 120, 104, 88, 72, 73, 74]
147	[68, 69, 70, 71, 72, 88, 104, 120, 136, 152, 168, 184, 183, 182, 181, 180, 164, 148, 132, 116, 100, 84]
148	[75, 91, 107, 123, 139, 155, 171, 187, 186, 185, 184, 183, 167, 151, 135, 119, 103, 87, 71, 72, 73, 74]
149	[68, 69, 70, 71, 72, 73, 89, 105, 121, 137, 153, 169, 185, 184, 183, 182, 181, 180, 164, 148, 132, 116, 100, 84]
150	[75, 91, 107, 123, 139, 155, 171, 187, 186, 185, 184, 183, 182, 166, 150, 134, 118, 102, 86, 70, 71, 72, 73, 74]
151	[68, 69, 70, 71, 72, 73, 74, 90, 106, 122, 138, 154, 170, 186, 185, 184, 183, 182, 181, 180, 164, 148, 132, 116, 100, 84]

Loop ID	Nodes
152	[75, 91, 107, 123, 139, 155, 171, 187, 186, 185, 184, 183, 182, 181, 165, 149, 133, 117, 101, 85, 69, 70, 71, 72, 73, 74]
153	[68, 84, 100, 116, 132, 148, 164, 180, 181, 182, 183, 184, 185, 186, 187, 171, 155, 139, 123, 107, 91, 75, 74, 73, 72, 71, 70, 69]
154	[68, 69, 70, 71, 72, 73, 74, 75, 91, 90, 89, 88, 87, 86, 85, 84]
155	[84, 85, 86, 87, 88, 89, 90, 91, 107, 106, 105, 104, 103, 102, 101, 100]
156	[100, 101, 102, 103, 104, 105, 106, 107, 123, 122, 121, 120, 119, 118, 117, 116]
157	[116, 117, 118, 119, 120, 121, 122, 123, 139, 138, 137, 136, 135, 134, 133, 132]
158	[132, 133, 134, 135, 136, 137, 138, 139, 155, 154, 153, 152, 151, 150, 149, 148]
159	[148, 149, 150, 151, 152, 153, 154, 155, 171, 170, 169, 168, 167, 166, 165, 164]
160	[164, 165, 166, 167, 168, 169, 170, 171, 187, 186, 185, 184, 183, 182, 181, 180]
161	[85, 101, 102, 103, 104, 105, 106, 90, 89, 88, 87, 86]
162	[165, 166, 167, 168, 169, 170, 154, 153, 152, 151, 150, 149]
163	[85, 101, 117, 118, 119, 120, 121, 122, 106, 90, 89, 88, 87, 86]
164	[165, 166, 167, 168, 169, 170, 154, 138, 137, 136, 135, 134, 133, 149]
165	[85, 101, 117, 133, 134, 135, 136, 137, 138, 122, 106, 90, 89, 88, 87, 86]
166	[165, 166, 167, 168, 169, 170, 154, 138, 122, 121, 120, 119, 118, 117, 133, 149]
167	[85, 101, 117, 133, 149, 150, 151, 152, 153, 154, 138, 122, 106, 90, 89, 88, 87, 86]
168	[165, 166, 167, 168, 169, 170, 154, 138, 122, 106, 105, 104, 103, 102, 101, 117, 133, 149]
169	[85, 86, 87, 88, 89, 90, 106, 122, 138, 154, 170, 169, 168, 167, 166, 165, 149, 133, 117, 101]
170	[85, 101, 117, 133, 149, 165, 166, 150, 134, 118, 102, 86]
171	[86, 102, 118, 134, 150, 166, 167, 151, 135, 119, 103, 87]
172	[87, 103, 119, 135, 151, 167, 168, 152, 136, 120, 104, 88]
173	[88, 104, 120, 136, 152, 168, 169, 153, 137, 121, 105, 89]
174	[89, 105, 121, 137, 153, 169, 170, 154, 138, 122, 106, 90]
175	[102, 103, 119, 135, 151, 150, 134, 118]
176	[105, 121, 137, 153, 152, 136, 120, 104]
177	[102, 103, 104, 120, 136, 152, 151, 150, 134, 118]
178	[105, 121, 137, 153, 152, 151, 135, 119, 103, 104]
179	[102, 118, 134, 150, 151, 152, 153, 137, 121, 105, 104, 103]
180	[102, 103, 104, 105, 121, 120, 119, 118]
181	[118, 119, 120, 121, 137, 136, 135, 134]
182	[134, 135, 136, 137, 153, 152, 151, 150]
183	[119, 120, 136, 135]

Loop ID	Nodes
184	[119, 135, 136, 120]

```
from m5.params import *
from m5.objects import *

from BaseTopology import SimpleTopology

class RouterLess(SimpleTopology):
    description='RouterLess'

    def __init__(self, controllers):
        self.nodes = controllers

    # Makes an equal number of cache and directory cntrls
    def makeTopology(self, options, network, IntLink, ExtLink, Router):
        nodes = self.nodes

        num_routers = options.num_cpus
        num_rows = options.num_rows

        # There must be an evenly divisible number of cntrls to routers
        # Also, obviously the number or rows must be <= the number of routers
        cntrls_per_router, remainder = divmod(len(nodes), num_routers)
        assert(num_rows <= num_routers)
        num_columns = int(num_routers / num_rows)
        assert(num_columns * num_rows == num_routers)

        # Create the routers in the mesh
        routers = [Router(router_id=i) for i in range(num_routers)]
        network.routers = routers

        # link counter to set unique link ids
        link_count = 0

        # Add all but the remainder nodes to the list of nodes to be uniformly
        # distributed across the network.
        network_nodes = []
        remainder_nodes = []
        for node_index in xrange(len(nodes)):
            if node_index < (len(nodes) - remainder):
                network_nodes.append(nodes[node_index])
            else:
                remainder_nodes.append(nodes[node_index])
        # Connect each node to the appropriate router
        ext_links = []
        for (i, n) in enumerate(network_nodes):
            cntrl_level, router_id = divmod(i, num_routers)
            assert(cntrl_level < cntrls_per_router)
            ext_links.append(ExtLink(link_id=link_count, ext_node=n,
                                     int_node=routers[router_id]))
            link_count += 1

        # Connect the remainding nodes to router 0.  These should only be
        # DMA nodes.
        for (i, node) in enumerate(remainder_nodes):
            assert(node.type == 'DMA_Controller')
            assert(i < remainder)
            ext_links.append(ExtLink(link_id=link_count, ext_node=node,
                                     int_node=routers[0]))
            link_count += 1

        network.ext_links = ext_links

        # Create the mesh links.  First row (east-west) links then column
        # (north-south) links
        int_links = []

        from RouterLessReader import RL_readLoops
        import math
        L = RL_readLoops(int(math.sqrt(num_routers)))


        # Create the RouterLess links.
        int_links = []
        for i in xrange(len(L)):
            loop = L[i]
            l_len = len(loop)
            for j in xrange(l_len):
                src_r = loop[j]
                dest_r = loop[(j+1)%l_len]
```

```
                int_links.append(IntLink(link_id=link_count,
                                         node_a=routers[src_r],
                                         node_b=routers[dest_r],
                                         node_a_port=3, # east port
                                         node_b_port=1, # west port
                                         weight=1,
                                         src_router_id=src_r,
                                         dest_router_id=dest_r,
                                         loop_id=i))
                            link_count += 1
                        network.int_links = int_links
```