International Journal of Parallel Programming (2020) 48:730–749
https://doi.org/10.1007/s10766-019-00648-9

# DancerFly: An Order-Aware Network-on-Chip Router On-the-Fly Mitigating Multi-path Packet Reordering

**Kang Jin**[1] · **Dezun Dong**[1] · **Cunlu Li**[1] · **Libo Huang**[1] · **Sheng Ma**[1] · **Binzhang Fu**[2]

Received: 2 August 2019 / Accepted: 4 November 2019 / Published online: 24 December 2019

**Abstract**
Adaptive routing algorithms can improve performance by balancing load across network channels in the presence of non-uniform traffic patterns. However, out-of-order packets can be introduced due to multi-path transmission of adaptive routing. With out-of-order transmission in the network, packets need to be reordered at the destination before being absorbed. Increasing network size with adaptive routing makes the time when a packet arrives at the destination extremely uncertain, which requires a large buffer to reorder the packets and this can exceed design space. Therefore, the challenge is to balance the trade-off between multi-path transmission and packet reordering. In this paper, we propose a novel packet reordering metric-OOD to quantify the degree of out-of-order. To minimize the OOD of packets, we propose DancerFly, an order-aware network-on-chip router that mitigates out-of-order packets caused by adaptive routing. DancerFly achieves this goal by providing two-level reordering. First, it performs in-buffer reordering by reordering packets queuing in the input buffer. Second, packets from different input ports are reordered before traversing through the router. We evaluate our design and the results show that the OOD can be reduced by 36.3% with comparable performance to the baseline.

**Keywords** Network-on-chip · Adaptive routing · Router architecture · Multi-path · Packet reordering

## 1 Introduction

As the number of cores keeps increasing on multi-core chips, the design of an efficient on-chip interconnection network is becoming more and more essential. Given

Kang Jin and Dezun Dong have contributed equally to this work.

✉ Dezun Dong
dong@nudt.edu.cn

Extended author information available on the last page of the article

the topology of the network, its performance is significantly determined by the routing algorithm. Adaptive routing algorithms balance load across the network channels to improve network performance and provide fault tolerance. However, parallel transmission in multiple paths can cause severe out-of-order packet delivery. The point is that the packets should be absorbed in order at the destination node. Therefore, faced with out-of-order packets, a reorder buffer is needed to reorder packets at the destination. Considering the uncertain arrival time of out-of-order packets, the reorder buffer usually needs to be designed very large, which brings a significant amount of hardware overhead. In fact, if packets arrived at the destination in order, the large reorder buffer can be omitted. Therefore, this paper is dedicated to reordering the out-of-order packets generated by adaptive routing.

Out-of-order packets are mainly caused by multi-path transmission in the adaptive routing. Routing algorithm can be classified as deterministic, oblivious and adaptive routing algorithms [3]. Deterministic routing algorithms route packets along the same path, regardless of the network status. Oblivious routing algorithms route packets without regard for the state of the network. In contrast, adaptive routing selects an appropriate path for the packet based on the network state. Adaptive routing algorithms facilitate load balancing and thus improve the network performance even in non-uniform traffic. What's more, the property of path diversity of adaptive routing algorithm enhances the robustness of the network and allows the network to tolerate faulty channels and nodes. Routing algorithms can also be classified as either minimal or non-minimal routing algorithms. Minimal routing algorithms always select the shortest path from source to destination, while non-minimal routing algorithms choose paths from the set of all minimal and non-minimal routes. Non-minimal routing algorithms improve load balance to some extent theoretically, but are hard to be applied in practice due to significant increase in delay and vulnerable to livelock and deadlock. Therefore, in this paper, we focus on minimal adaptive routing.

Recently, a large number of adaptive routing algorithms have been proposed to improve network performance [5,6,10,12,14,15]. However, adaptive routing causes severe packets disorder due to multi-path transmission, especially for non-uniform traffic patterns. With unbalanced traffic pattern, a packet at the front of a flow can arrive at the destination node after other packets in this flow, because a congested path is selected for the front packet. Despite out-of-order packets, in-order packet delivery is a widely assumed basis for a wide range of applications such as multimedia, file transfer protocols and cache coherence protocols [7,8]. Implementations of direct-communication computation models such as stream computing also require that packets be delivered in the order they were sent, as do explicit message-passing applications.

Adaptive routing algorithms whose granularity of adaptivity is packet cannot achieve in-order packet delivery due to multi-path transmission. To ensure in-order delivery, previous works choose deterministic or oblivious routing [4,13,16] rather than adaptive routing. To enable adaptive routing, we must abandon in-order delivery. If the order is required, we can only perform reordering at the destination dynamically. As far as we know, there is no other solution except for setting a large reorder buffer at the destination. What we can do is to mitigate packet reordering as much as possible to reduce the buffer and latency overhead of reordering.

In this paper, we innovatively view the out-of-order problem from the perspective of router architecture, rather than routing strategy or flow control. This ensures that there is no performance loss while reordering. We first propose an elaborate packet reordering metric: Out-of-Order Degree (OOD). It is defined as the maximum reorder buffer occupancy among all nodes. With the goal of minimizing OOD, an or**D**er-**a**ware **n**etwork-on-**c**hip rout**er** on-the-**Fly** mitigating multi-path packet reordering (DancerFly) is proposed. With DancerFly, packets routed through the router are reordered, and the reordering is performed in two aspects. First, DancerFly reorders packets queuing in the input buffer through the proposed reordering logic once a packet is inserted. Second, DancerFly reorders packets from different input port in switch allocation stage, which is accomplished by the proposed order-aware switch allocator. Through the two-level reordering, packets can maintain order when ejected from the router. More remarkable, DancerFly is highly effective and does not degrade network performance. This is because DancerFly perceives the order of packets passing through and reorders them, rather than restricting their transmission. That is to say, we can retain the advantages (e.g., performance improvement, load balancing and fault tolerance) of adaptive routing algorithms and minimize the OOD of packets meanwhile.

In particular, the contributions of this work are as follows.

- We propose a new packet reordering metric which quantifies the degree of packets disorder.
- To the best of our knowledge, this is the first work to address packets disorder while truly retaining adaptive routing.
- A novel router design, DancerFly, is proposed, which minimizes packets disorder without performance loss.

The rest of this paper is organized as follows. Section 2 describes the out-of-order delivery problem with adaptive routing and discusses the prior solutions to this problem. Section 3 presents a new packet reordering metric. In Sect. 4, the design of DancerFly is introduced. For evaluating the performance of DancerFly, we present extensive experimental results in Sect. 5 to compare DancerFly with the canonical router. The implementation overheads are discussed in Sect. 6. More in-depth studies relating to the implementation and scalability of DancerFly are presented in Sect. 7. We conclude in Sect. 8.

## 2 Motivation

Adaptive routing, which allows packets follow multiple path from the same source and destination, leads to severe packets disorder. To highlight this problem, a scenario of packets disorder caused by adaptive routing is shown in Fig. 1, for a $4 \times 4$ 2D mesh with 3 VCs per physical channel. Packet A and packet B are two sequential packets of a flow from source $s$=01 to destination $d$=11. Packet A is transmitted along the path represented by red solid arrows. The latter packet B is routed along the path denoted by red dashed arrows. Since adaptive routing always selects the smoothest path, packet B passes node 07 before A and reaches the destination $d$=11 earlier.

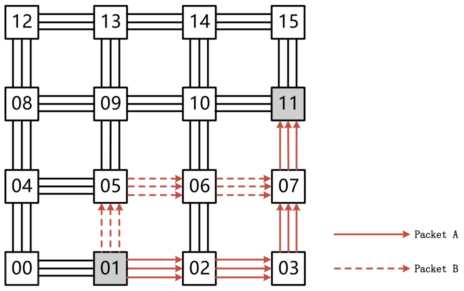


**Fig. 1** A scenario of packets disorder caused by adaptive routing

In many cases, we are required to maintain ordering among certain packets traveling through the network. Parallel memory accesses require in-order delivery to improve network performance [11]. Some protocols, such as file transfer protocols and cache coherence protocols, depend on message order being preserved for correctness. Stream computing (e.g., StreamIt [18]) and explicit message-passing applications also require in-order packet delivery.

Although basic dimension-order routing (DOR) without virtual channels (VCs) [2] naturally deliver packets in order, it offers limited efficiency. Murali et al. [16] use flow control to ensure in-order delivery for oblivious multi-path routing. The basic idea is to only allow non-intersecting[1] paths between each source destination pair and force out-of-order packets to wait at the input port of the destination router. The destination router is forced to pick up the packets in order from different input port buffers before delivering them to the core. This scheme suffers from three important problems. First, it increases network latency and decreases network throughput because of restrictions on the transmission of packets. Second, it strongly affects the path diversity of the underling routing function as all intersecting routing paths are not allowed. Third, it is not deadlock free due to the condition for ejecting packets from the network.

EDVCA [4] targets the ordering problems incurred by dynamic virtual channel allocation. It proposes a technique in which a packet is only restricted to a certain next-hop VC if it contains some packet of the same flow. However, this scheme only applicable to deterministic routing algorithms. Based on the idea of EDVCA, POIOR [13] ensures that at any snapshot packets flow through only one path even though the path may be different at different times. This way, the flow only travels along one path at any given instant. Apparently, this scheme suffers from two problems. First, it limits the parallel transmission of packets which degrades the network performance. Second, it is only applicable to oblivious routing.

The scheme proposed in [17] aims to ensure in-order packet delivery while retaining the performance advantages of adaptive routing. The basic idea is to force the packets of a flow to use the same path which is adaptively determined by the first packet. That is to say, the adaptive of the routing algorithm is allowed in terms of flows and not in terms of packets. There is a major problem with this scheme. It does not respond to the state of the network in a timely manner. In other words, the load cannot be accurately balanced, which will decrease the network performance.

[1] A set of paths are defined to be non-intersecting if the paths originate from the same source vertex but do not intersect each other in the network, except at the destination vertex.

Prior works that provide in-order guarantee are achieved by modifying or abandoning the adaptive routing algorithms. Limiting the adaptivity or parallel transmission of adaptive rouging algorithms permits in-order packet delivery, but at the expense of performance loss. Conversely, it is impossible to realize in-order delivery without limiting adaptivity and parallel transmission. This is due to the fact that, for instance, some packets which started earlier in the sequence get routed through heavily congested paths and reach the destination later than packets which started after them. Therefore, we can only strive to reduce the degree of out-of-order to reduce the required reorder buffer in the destination. The absence of effective and practical techniques able to mitigate packet reordering in the context of NoC design is the main argument which motivates this work.

## 3 Packet Reordering Metric

In this section, we propose a novel packet reordering metric called Out-of-Order Degree (OOD). Then we present a methodology for OOD measurement at the receiver, including the theory and its implementation algorithm.

### 3.1 The Definition of OOD

Before to explain OOD, we first give the definition of some terms. Consider a sequence of packets (0, 1, 2, . . ., N) transmitted from a source node to a destination node.

**Expected packet (E)** Expected packet '$E$' is the largest sequence number such that all the packets with sequence number less than $E$ have already arrived.

**Buffer-occupancy (B)** An arrived packet with a sequence number greater than that of expected packet is considered to be stored in a hypothetical buffer long enough to recover from reordering. At any instance of packet arrival, the buffer occupancy is equal to the number of such out-of-order packets in the buffer including the arrived packet (assuming one buffer for each packet).

The buffer occupancy '$B$' is recorded following each arrival. For a set of packets, we define the maximum buffer occupancy as OOD of the packets. This is because the maximum buffer occupancy determines the lower bound of the required reorder buffer in the receiver. Similarly, the OOD of a flow is the maximum buffer occupancy in the receiving process and the OOD of the whole network is maximum OOD of all flows.

To explain OOD explicitly, we give a sample example. For sequence of arrivals (4, 3, 2, 0, 1) shown in Table 1, the corresponding expected packet values are (0, 0, 0, 0, 1). The buffer occupancy value immediately following the arrival of packet with sequence number 4 is 1; it has to be buffered as an earlier packet is still being expected. Similarly, following the arrivals of packet 3 and 2, the buffer occupancy becomes 3. Then, the packet with sequence number 0 arrives, while buffer still cannot be released since packet 1 is expected. Finally, when packet 1 arrives, packets 4, 3 and 2 can be recovered from reordering buffer and the buffer-occupancy changes to zero.

**Algorithm 1** OOD Measurement Algorithm Description

```
1: // STEP 1: Convert arrival sequence to natural number sequence
2: $seq_{con} = Convert(seq)$
3: // STEP 2: OOD Measurement
4: // $exp$ = expected value; $size$ = maximum sequence number; $deg$ = temporary degree; $beg$ = position
   of the first packet buffered
5: $exp = 0$; $beg = 0$; $deg_{max} = 0$
6: while $exp < size$ do
7:     $deg = 0$
8:     if $seq_{con}[beg] == exp$ then
9:         $beg++$
10:        $exp++$
11:    else
12:        $deg++$
13:        for $i = beg + 1$ to $size$ do
14:            if $seq_{con}[i] == exp$ then
15:                if $deg > deg_{max}$ then
16:                    $deg_{max} = deg$
17:                end if
18:                $exp++$
19:                $break$
20:            else if $seq_{con}[i] > exp$ then
21:                $deg++$
22:            end if
23:        end for
24:        if $i == size$ then
25:            // expected packet is lost, break the process
26:            if $deg > deg_{max}$ then
27:                $deg_{max} = deg$
28:            end if
29:            $break$
30:        end if
31:    end if
32: end while
33: $OOD = deg_{max}$
```

**Table 1** Buffer-occupancy computation for arrival sequence

| Cycle | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Arrival sequence | 4 | 3 | 2 | 0 | 1 |
| Expected packet | 0 | 0 | 0 | 0 | 1 |
| Buffer-occupancy | 1 | 2 | 3 | 3 | 0 |

So, the OOD of those packets is 3 in this case. OOD is applicable not only for natural number identified sequence of packets, but also for cases where the sequence number is incremented discontinuously. All we have to do is to convert the irregular sequence into natural number sequence.

## 3.2 The Implementation of OOD Measurement

To implement the measurement of OOD at the receiver, we provide an efficient methodology based on the observation as follows:

$$OOD = \max((Pos_{small} - Pos_{large}) - Num_{less})$$

$Pos_{small}$ and $Pos_{large}$ are the positions of two inverse numbers of an arrival sequence. $Pos_{small}$ is the position of the packet that has a small sequence number but arrives late. $Pos_{large}$ is the position of the packet that has a large sequence number but arrives early. So, $Pos_{small} - Pos_{large}$ is the interval between the two inverse numbers. Since the packets with a sequence number less than the smaller one have already recovered from reorder buffer, we exclude $Num_{less}$ as shown in the equation. The maximum $((Pos_{small} - Pos_{large}) - Num_{less})$ for all inversions is the maximum buffer-occupancy, i.e., OOD.

For the example described in the previous subsection (i.e., sequence of arrivals (4, 3, 2, 0, 1)), the buffer-occupancy for the inversion between 4 and 3 is 1. Similarly, the buffer-occupancy for the inversion between 4 and 0 is 3 according to the formula above. In particular, for the inversion between 4 and 1, packet 0 should be excluded and the buffer-occupancy is 3. Overall, the OOD of the arrival sequence is 3 which is equal to the theoretical result analyzed in the previous subsection.

The algorithm of OOD measurement is described in Algorithm 1. The algorithm first converts the arrival sequence to natural number sequence. For each expected value, we traverse from *beg* to the end of the sequence. The variable *beg* is the position of the first packet buffered. All the packets before it in the sequence become in order and are recovered from reorder buffer. If the expected packet is found, the expected packet value *exp* is increased and the temporary out-of-order degree *deg* is updated. Else, the program is interrupted since the expected packet is lost and thus the following packets in the sequence cannot be processed.

## 4 Design Concept of DancerFly

### 4.1 The Basic Idea

There are two kinds of parallelism leading to packets disorder. The first is multi-VC transmission which can cause disorder even if the packets take the same path. The second is multi-path transmission which causes more serious disorder. In order to mitigate the degree of disorder as much as possible without limiting the path diversity and parallel transmission, we start with the router architecture. In the first case, we propose to reorder the packets queuing in the input buffer. In the second case, we propose to reorder packets through switch allocation strategy at the path intersection node. What's more, all the routers passed by a traffic flow mitigate the packets disorder collectively. This scheme is deadlock-free since it does not introduce new dependency circles. As long as the routing algorithm is deadlock-free, the proposed scheme is deadlock-free.

### 4.2 Baseline Router Microarchitecture

The proposed DancerFly is based on a canonical virtual-channel router. A typical virtual-channel router has multiple VCs per input port and virtual channel flow control

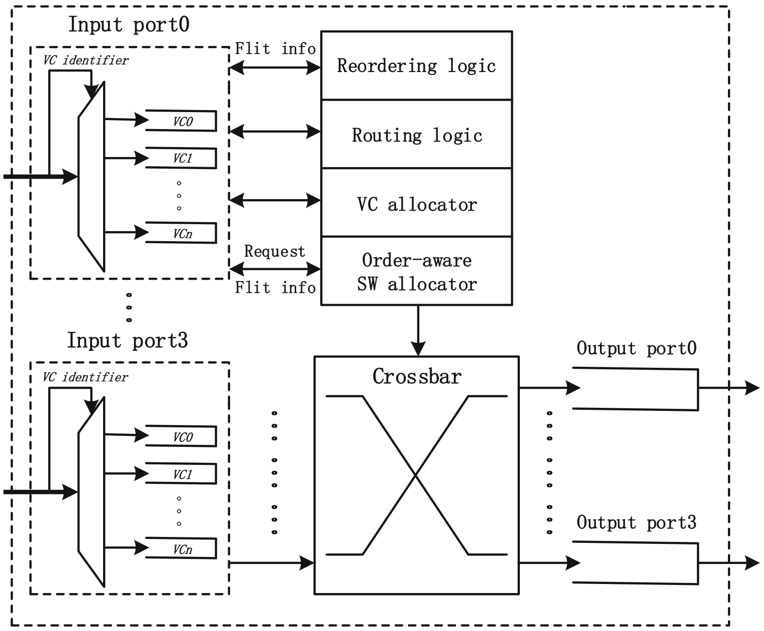


**Fig. 2** DancerFly architecture

**Algorithm 2** Reordering Algorithm Description

```
1: // $cur\_flit$ = pointer to the incoming flit; $tmp$ = pointer to linked list node
2: $tmp = head{-}{>}next$
3: while $tmp \neq$ NULL do
4:     // traverse the linked list
5:     if $tmp{-}{>}flit.fid == cur\_flit.fid$ then
6:         if $tmp{-}{>}flit.pid > cur\_flit.pid$ then
7:             // if out-of-order, swap position
8:             $tmp\_flit = cur\_flit$
9:             $cur\_flit = tmp{-}{>}flit$
10:            $tmp{-}{>}flit = tmp\_flit$
11:        end if
12:    end if
13:    $tmp = tmp{-}{>}next$
14: end while
15: $tmp$ = new node
16: $tmp{-}{>}flit = cur\_flit$
17: $tmp{-}{>}next =$ NULL
18: $tail{-}{>}next = tmp$
```

is applied [1]. The components of a virtual-channel router can be partitioned into two groups based on functionality: the datapath and control plane. The datapath of the router handles the storage and movement of a packet and consists of a set of input buffers, a switch, and a set of output buffers. The remaining blocks implement the control plane of the router and are responsible for coordinating the movement of packets through the resources of the datapath. For the virtual-channel router, the control blocks perform route computation, VC allocation, and switch allocation.

Input buffers hold flits while they are waiting for VCs, switch bandwidth, and channel bandwidth. Input buffers are evenly partitioned across VCs, which enables input speedup in the switch and thus increases the throughput of the router. Linked list [3] data structure is utilized for each VC buffer, which gives flexibility in buffer allocation and management. The switch is the heart of a router. General crossbar switch which makes efficient use of internal bandwidth is exploited in the canonical router. Since there is no need to partition the output buffer storage across VCs, a single FIFO buffer is used for each output.

The pipeline of the canonical router is composed of four stages: *Routing Computation* (RC), *VC Allocation* (VA), *Switch Allocation* (SA) and *Switch Traversal* (ST) [3]. We assume each pipeline stage takes one clock cycle. Prior works on improving the throughput of router are orthogonal to the proposed packet reordering router design, so our idea can be integrated into most modern router architectures.

### 4.3 Order-Aware Input Buffer

To reorder packets in the input buffer, we expand the flit insertion logic into more powerful logic called reordering logic as shown in Fig. 2. To help understand the operation of reordering logic, single-flit packet is assumed. Compared with original flit insertion logic, which just adds flit to the tail of the linked list simply, the reordering logic inserts the flit into an appropriate position and keeps packets in order. This is achieved by perceiving the order of packets by utilizing the flit information. Generally, a flit contains $fid$ and $pid$ fields. $fid$ is flow identifier to which it belongs. $pid$ is the packet sequence number in the flow. When a flit is about to enter a VC, the reordering logic will find all the flits in this VC buffer that belong to the same flow as the incoming flit, by comparing the $fid$ field. Then, these flits, including the incoming flit, are reordered by comparing the $pid$ field. Note that the reordering logic just swaps position among these flits. Other flits do not move. For multi-flit packet, packets are reordered in the similar manner. Since the reordering only triggers when a tail flit arrives, the integrity of packets is guaranteed.

Algorithm 2 illustrates the pseudo-code of reordering logic. For ease of understanding, the reordering operation is described in an iterative way. Since reordering is triggered every time a packet enters a VC, the packets are always in order for each VC buffer. Thus, we only need compare the incoming flit with the flits in the linked list once. $cur\_flit$ is a pointer to the incoming flit. Each flit whose fid is equal to $cur\_flit.fid$ is compared and swapped with $cur\_flit$ if its $pid$ is greater than $cur\_flit.pid$. Finally, a new node needs to be created and added to the tail of the list. The flit stored in the new node can be either the incoming flit or the flit swapped out. In fact, the reordering process can be implemented in parallel. $fid$ and $pid$ of incoming flit can be compared with those of flits in the linked list simultaneously. Then, all the flits, which belong to the same flow as the incoming flit and have a greater $pid$, are moved backward at the same time. The incoming flit is inserted in the vacancy

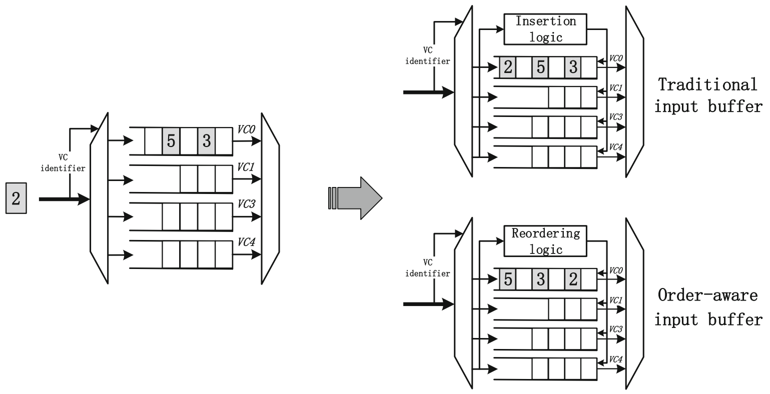


**Fig. 3** Example of reordering in order-aware input buffer

meanwhile. Overall, the reordering operation can be executed in two cycles. Note that only the non-head packets in a VC are reordered. The additional cycle can be hidden.

Figure 3 shows an example that illustrates the difference between traditional input buffer and the proposed order-aware input buffer. We assume there are 4 VCs for an input. The grey cells are packets from the same flow. The white cells are packets from other flows. A packet with $pid$ equal to 2 is about to enter VC0 as shown in the figure. For traditional input buffer, this flit is added to the tail of the queue simply. The input buffer cannot realize these packets are out-of-order. In contrast, the flit is inserted into the right place and kept in order with order-aware input buffer. This is accomplished by the lightweight and powerful reordering logic.

## 4.4 Order-Aware Switch Allocator

In this subsection, we present a novel order-aware switch allocator which can reorder packets from different input port. Compared with traditional switch allocator, the order-aware switch allocator can sense the order of packets by utilizing the flit information, as shown in Fig. 2. Before to display the operation of order-aware switch allocator, we first give a brief introduction about switch allocation and its representation.

**Allocation** In a switch, each input may request one or more outputs, and each output may be requested by one or more inputs. An allocator performs a matching between the input ports and output ports. The allocator considers the requests and performs allocation subject to three rules:

- A grant can be asserted only if the corresponding request is asserted.
- At most one grant for each input may be asserted.
- At most one grant for each output can be asserted.

For a group of requests, there may be many valid allocation results. A solution that contains the maximum possible number of assignments is called a maximum matching. A solution in which no additional requests can be serviced without removing one of the existing grants, is called a maximal matching. Apparently, a maximum matching is

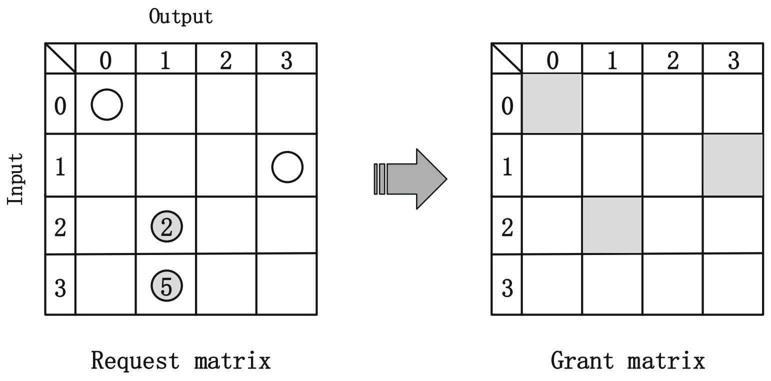


**Fig. 4** Example of order-aware switch allocation

certainly a maximal matching, but a maximal matching is not necessarily a maximum matching conversely.

**Representation** The requests can be represented by a request matrix $R$, and the corresponding grants can be represented by a grant matrix $G$. Each cell of the matrix represents an input-output pair. $R$ is an arbitrary binary-valued matrix. $G$ is also a binary-valued matrix that only contains ones in entries corresponding to non-zero entries in R. In addition, $G$ has at most one '1' in each row, and at most one '1' in each column.

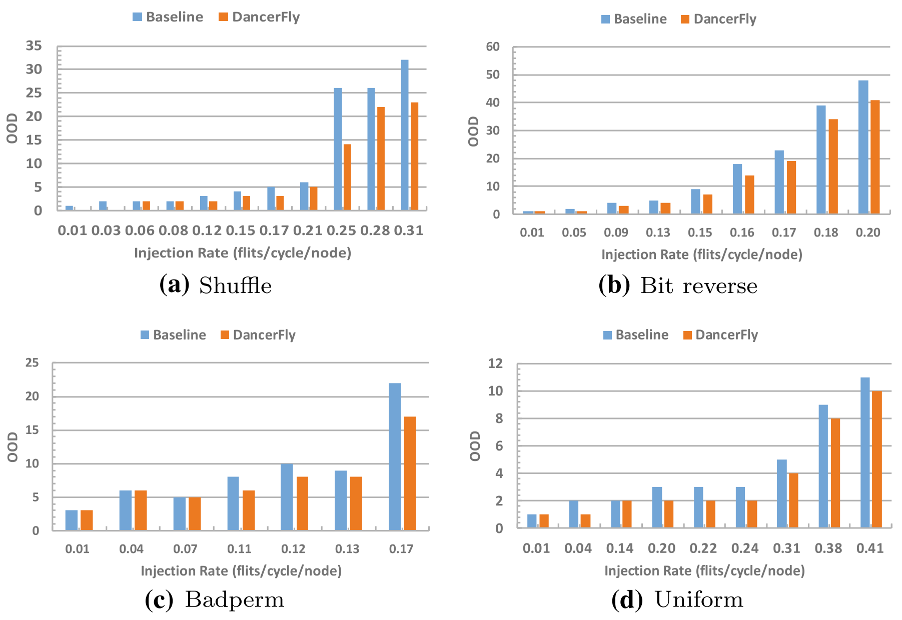
It is common that packets, which belong to the same flow, from different input ports contend for an output port due to adaptive routing. This is what happens at node 07 shown in Fig. 1. The proposed order-aware switch allocator resolves this conflict by prioritizing the packet with lower $pid$. In contrast, the traditional switch allocator selects packet blindly when facing conflict.

To have an intuitive understanding, we give an example as shown in Fig. 4. We assume a 4-port router and the number of VC per physical channel is set to 2. In the figure, the matrix on the left is the request matrix and the matrix on the right is the corresponding grant matrix generated by order-aware switch allocator. The circles stand for the requests. The grey circles represent that the corresponding requesting packets are from the same flow and the number inside is the $pid$. The grey cells in the grant matrix are granted requests. There are four requests in the request matrix, i.e., (0, 0), (1, 3), (2, 1), (3, 1), of which request (2, 1) conflicts with the request (3,1). With order-aware switch allocator, the request (2, 1) is prioritized as the corresponding requesting packet has smaller $pid$. In contrast, traditional switch allocators cannot perceive the order of packets and select request blindly. The resulting grant matrix with order-aware switch allocator is shown at the right part of the figure. Note that the proposed switch allocator achieves maximum matching. Actually order-aware switch allocator provides better selection when output conflicts occur and does not affect the number of assignment.

**Table 2** Network simulation parameters

| Parameters | Values |
|---|---|
| Routing algorithm | **Footprint** [5], Dyxy [12] |
| Traffic pattern | **Shuffle**, Bit reverse, Badperm, Uniform |
| Flow control mechanism | Virtual Channel, Credit-Based |
| Packet size | **single-flit packet**, 2-flit packet, 4-flit packet |
| Flow size | **100 packets**, 300 packets, 500 packets, 700 packets |
| VC number | **2**, 4, 8 VCs per physical channel |
| Network size | $4 \times 4$, $\mathbf{8\times8}$, $16 \times 16$ |

The default values are marked in bold



**(a)** Shuffle **(b)** Bit reverse **(c)** Badperm **(d)** Uniform

**Fig. 5** OOD comparison with different traffic pattern for Footprint

## 5 Evaluation

In this section, we compare the performance of the proposed DancerFly with the canonical virtual channel router. The traditional flit insertion logic and iSLIP switch allocator are used in the baseline router. Other configurations of baseline router are the same as those of DancerFly. The evaluation is twofold. First, we focus on the OOD of the network and we synthesize several conventional traffic patterns to evaluate the effect of different router design. The OODs of network with different routing algorithms and flow sizes are also measured to evaluate the reliability of DancerFly. Second, we focus on the network performance (latency and throughput). Similarly, the impact of different traffic patterns and routing algorithms are evaluated.

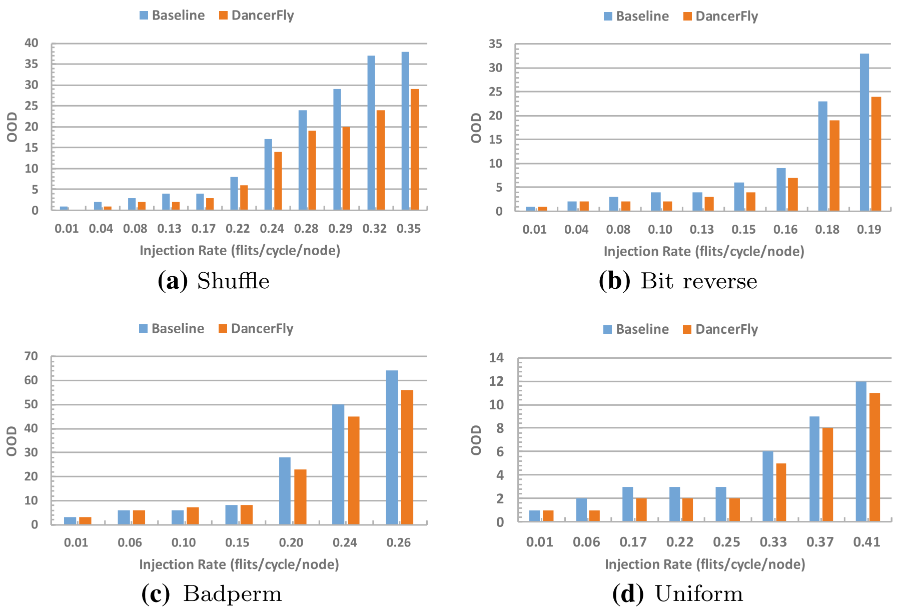


**Fig. 6** OOD comparison with different traffic pattern for Dyxy

## 5.1 Methodology

We modified the Booksim simulator [9] to model the router architecture and interconnection network. The router pipeline is three cycles plus one cycle for link traversal. The detailed configurations are listed in Table 2. If the configurations are modified in a specific experiment, they will be mentioned for avoiding confusion. The baseline network topology is a 8 × 8 2D mesh. Footprint [5] and Dyxy [12] are chosen for adaptive routing algorithms. The default VC number per physical channel is 2 and VC buffer size is 20 flits. Different VC number and VC buffer size are also evaluated. To understand the impact of the router architecture and isolate the impact of the flow control, single-flit packets are used as default in the evaluation. The impact of multi-flit packets is also evaluated. Flow size defaults to 100 packets. Different synthetic traffic patterns are used in the evaluation such as uniform, shuffle, bit reverse and badperm. The simulator is warmed up for 15,000 cycles and then the OOD and network performance are measured over another 30,000 cycles.

## 5.2 Evaluation for OOD

### 5.2.1 Different Adaptive Routing Algorithm

Figures 5 and 6 show the OOD comparison of Baseline and DancerFly as injection rate increases in different synthetic traffic patterns with Footprint and Dyxy routing algorithm. Considering practical significance, only the results within saturation throughput are given. As shown in Fig. 5a, the OOD is very small at low load but increases rapidly as injection rate increases. In this case, the OOD can be up to 32% of the flow size at saturation throughput. This is because at low load the network is unimpeded and thus it is rare for later packets to arrive before the earlier packets. However, at high load, congestion occurs in some areas due to load imbalance and it is very likely that the earlier packets are blocked and the later packets choose other unblocked path, leading to faster arrival. Based on this observation, the maximum OOD for a specific network configuration is determined by the OOD at saturation throughput.

Apparently, DancerFly always achieves smaller OOD than Baseline under different traffic patterns for Footprint and Dyxy. With Footprint routing algorithm, DancerFly shows 28.1%, 14.6%, 22.7% and 9.1% OOD reduction over Baseline in saturation throughput for shuffle, bit reverse, badperm and uniform traffic pattern. With Dyxy routing algorithm, DancerFly improves the OOD in saturation throughput for shuffle, bit reverse, badperm and uniform versus Baseline by 23.7%, 27.2%, 12.5% and 8.3%. These traffic patterns cause regional congestion in high load, and adaptive routing algorithms which enable multi-path transmission to avoid congestion, lead to severe out-of-order. Previous router designs cannot be aware of the disorder of packets. With DancerFly, the packets are reordered as soon as they enter the input buffer. In addition, packets in different input ports are reordered through order-aware switch allocator.

Note that the OOD reductions with uniform traffic are less than that with non-uniform traffic. That is because the load is already balanced across the network with uniform traffic pattern. It rarely happens that the latter packets route around congestion area, selecting the less congested path, and arrive at the destination earlier finally. Therefore, the OOD in uniform traffic is relatively low and there is little room for DancerFly to improve. Nevertheless, DancerFly still achieves about 10% improvement.

### 5.2.2 Impact of Packet Size

The effect of packet size is shown in Fig. 7a. The maximum OOD is the OOD in saturation throughput which determines the reorder buffer size in the destination. DancerFly always achieves OOD reductions, but the reductions decrease when increasing packet size. Specifically, DancerFly achieves 28.1%, 18.8% and 19.1% OOD reduction over Baseline for 1, 2, 4-flit packet respectively. The OOD reduction decreases from 1-flit packet to 2-flit packet. The reason is that there are fewer packets that can be accommodated for the same VC buffer. Thus, the effect of reordering is weakened in the order-aware input buffer. However, the effect of order-aware switch allocator is not affected by packet length because the switch allocator operates on flits.

### 5.2.3 Impact of Flow Size

In addition to 100 packets flow, we also evaluate the effect of DancerFly with different flow size. Figure 7b shows the OOD comparison between Baseline and DancerFly with 300 packets flow, 500 packets flow and 700 packets flow. As shown in the figure, DancerFly outperforms Baseline for different flow sizes. In particular, the OOD reductions of DancerFly against Baseline in saturation throughput are 36.3%, 14.8% and

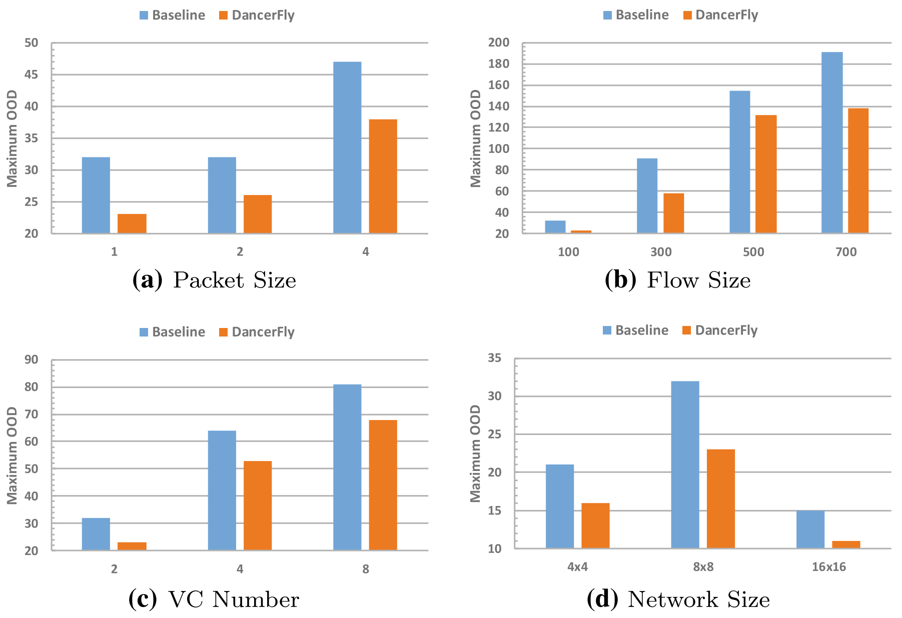
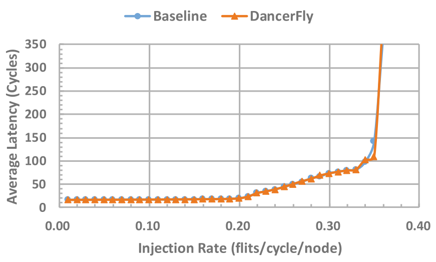


**Fig. 7** Impact of packet size (**a**), flow size (**b**), VC number (**c**) and network size (**d**)

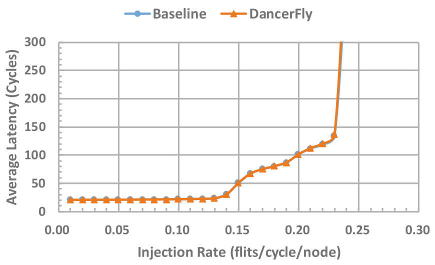
27.8% for 300 packets flow, 500 packets flow and 700 packets flow respectively. This is achieved because DancerFly provides order awareness and keeps packets transmitting in order compared to Baseline. Furthermore, all DancerFlys in the network work collectively and reduce OOD of the network dramatically.

### 5.2.4 Impact of Number of VCs

In this evaluation, we explore the impact of VC number on DancerFly. Figure 7c shows the OOD comparison of Baseline and DancerFly with 2, 4 and 8 VCs per physical channel. Since the input buffer size remains the same, the corresponding VC buffer sizes are 20, 10 and 5 flits. It is expected that the number of VC can have some impact on the effect of DancerFly. The reason is that the proposed lightweight order-aware buffer reorders packets in each VC independently. Considering that reordering packets across VCs will cause a lot of area, power and latency overhead, and get few or even no improvement in return, the method in this paper is more cost-effective. In addition, the VC buffer size decreases as VC number increases. Thus, the effect of reordering is weakened due to fewer packets in the VC buffer. Nevertheless, DancerFly always outperforms Baseline for any VC number. Specifically, DancerFly achieves 28.1%, 17.2% and 16.0% OOD reduction compared to Baseline for 2, 4 and 8 VCs respectively. This shows that DancerFly can significantly reduce the OOD with different number of VCs. Similarly, this is the result of two-level reordering of DancerFly.



**(a)** Shuffle



**(b)** Bit reverse

**Fig. 8** Latency-throughput comparison with different traffic pattern for Dyxy

### 5.2.5 Impact of Network Size

We analyze the scalability of DancerFly by evaluating the OOD on a $4 \times 4$ and a $16 \times 16$ mesh topology. The results are shown in Fig. 7d. In general, DancerFly outperforms Baseline regardless of the network size. Specifically, the OOD reductions of DancerFly over Baseline are 23.8% and 26.7% for $4 \times 4$ and $16 \times 16$ mesh respectively. Note that the OOD reductions are comparable to that of $8 \times 8$ mesh. This demonstrates that the proposed scheme, DancerFly, is scalable. This is achieved because each DancerFly performs reordering independently.

## 5.3 Evaluation for Network Performance

In this section, we evaluate the network performance with DancerFly and Baseline. Theoretically, the network performance with DancerFly is identical to that with Baseline under any network configurations. We prove it from two aspects. First, for order-aware buffer, we does not change the routing path (port and VC) of the packets. Since the reordering logic just swaps the position of packets of a flow, it does not affect the flow latency and does not affect the packets of other flows. Second, for order-aware switch allocator, it does not change the maximum matching, since it just provides better choice when output conflicts occur compared to previous switch allocators which select input port blindly.

Figures 8 and 9 give the latency results for Dyxy and Footprint routing algorithms using shuffle and bit reverse traffic pattern. The blue curve shows the Baseline and the orange curve represents the DancerFly. As shown in the figures, the two curves are almost completely coincident. This means that the average latency and throughput are the same for Baseline and DancerFly. There is a little difference in average latency at 0.35 for the first figure in Fig. 8. Similarly, the difference exists at 0.32 for the first figure in Fig. 9. This is due to different selection strategy when output conflicts occur, as described in Sect. 4.4. Since only one output can be granted to each input, whichever input you choose will cause other requests from this input to be unsatisfied. It is entirely a matter of probability. Sometimes DancerFly gets more matches and sometimes Baseline. Overall, the experiment results verify the theoretical result stated above. According to the evaluation, the proposed DancerFly not only retains

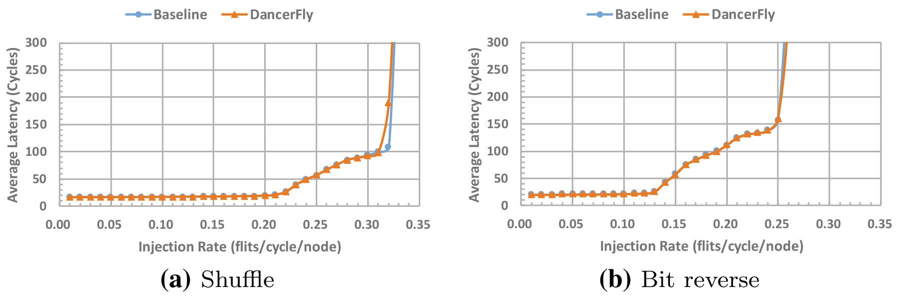


**Fig. 9** Latency-throughput comparison with different traffic pattern for Footprint

the advantages of adaptive routing (e.g., lower latency, higher saturation throughput, load balancing and fault tolerance), but also reduces the OOD dramatically.

## 6 Overhead

For order-aware input buffer, the flit insertion logic is substituted by a lightweight reordering logic which ensures that packets are ordered after insertion. This process does not introduce additional registers since we only need change several pointers which already exist in pointer memory [3]. The reordering logic can be executed in two cycles. Since only non-head packets in the VC buffer are reordered, the additional cycle can be hidden. Thus, no additional cycles are introduced to the critical path. For order-aware switch allocator, we use flit information to select the front packet of a flow when output conflict occurs. This only introduces sample compare logic and does not lengthen the clock of switch allocation stage. In summary, the proposed DancerFly just adds some simple logic in control plane, but does not introduce additional cycles and registers in critical path.

## 7 Discussion

### 7.1 Scalability

The DancerFly is topology agnostic. It can be applied to other topologies without alteration. In addition, the reordering system composed of all DancerFlys in the network performs reordering in a fully local manner, allowing it to scale well to large networks. DancerFly only involves router architecture design and does not change routing strategy. That makes DancerFly applied for any routing algorithms, including adaptive and deterministic routing algorithms. In addition to on-chip network, the idea of DancerFly can be applied to off-chip network as long as packets reordering is needed.

### 7.2 Reordering Between VCs

Reordering between VCs needs complex logic unit, additional resisters, a lot of wiring and power overhead, but gets little or no OOD reduction in return. Since reordering between VCs involves adjustment of elements in multiple linked lists, additional registers are required to store temporary data. The reordering logic will be more complex and may need multiple cycles. Additional cycles may be added to the critical path accordingly. Actually, the disorder between VCs has little impact on OOD of the whole flow which is dominated by multi-path (port) transmission. The reason is that the disorder between VCs is local, while OOD reflects the maximum degree of disorder. In addition, the number of VCs is de facto small. More VCs can increase the circuit complexity and power consuming considerably.

### 7.3 Other Instances in Switch Allocation

We considered the case that different inputs request different outputs for packets from the same flow. However, this is almost impossible to happen. Route computing for a packet is based on the local information and the destination address. Since these packets perform route computing almost at the same time, the local information and their destination are the same and thus the resulting outputs are the same. Even if this happens, we should not just choose the packet with small $pid$ and discard the packet with large $pid$. It is because this will decrease the number of assignment and result in performance loss. Thus we just leave it as it is.

## 8 Conclusion

Packet reordering is a thorny problem that mainly caused by adaptive routing. Setting a huge reorder buffer which is rare in NoC for each receiver node is infeasible in practice. To minimize packets disorder, we have to abandon adaptive routing and back to deterministic routing previously. In this work, we address this problem from the perspective of router architecture innovatively. We first propose a packet reordering metric and its implementation. Then, we propose DancerFly, a novel order-aware and packet reordering router. To the best of our knowledge, it is the first work minimizing packet reordering for adaptive routing. We achieve this by utilizing the flit information and sensing the order of packets in the router. Then, the packets are reordered in the router pipeline. This is implemented by adding some simple logic to the control plane of the conical router and there is no additional buffer overhead in datapath. Experiments show that DancerFly achieves significant OOD reduction for various adaptive routing algorithms and traffic patterns. DancerFly is topology-agnostic; future work will extend DancerFly to additional topologies beyond mesh networks. We also plan to apply DancerFly to off-chip networks.

**Acknowledgements** We thank the anonymous reviewers for their valuable feedback. We also appreciate members of Tianhe interconnect group at NUDT for many inspiring conversations early in the project. This project was partially supported by the National Science and Technology Major Projects on Core

Electronic Devices, High-End Generic Chips and Basic Software under grants No.2018ZX01028101 and No. 2017ZX01038104-002.

## References

1. Dally, W.J.: Virtual-channel flow control. IEEE Trans. Parallel Distrib. Syst. **3**(2), 194–205 (1992)
2. Dally, W.J., Aoki, H.: Deadlock-free adaptive routing in multicomputer networks using virtual channels. IEEE Trans. Parallel Distrib. Syst. **4**(4), 466–475 (1993). https://doi.org/10.1109/71.219761
3. Dally, W., Towles, B.: Principles and Practices of Interconnection Networks. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2003)
4. Devadas, S., Cho, M.H., Shim, K.S., Lis, M.: Guaranteed in-order packet delivery using exclusive dynamic virtual channel allocation (2009)
5. Fu, B., Kim, J.: Footprint: Regulating routing adaptiveness in networks-on-chip. In: Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA '17, pp. 691–702. ACM, New York (2017). https://doi.org/10.1145/3079856.3080249
6. Gratz, P., Grot, B., Keckler, S.W.: Regional congestion awareness for load balance in networks-on-chip. In: 2008 IEEE 14th International Symposium on High Performance Computer Architecture, pp. 203–214 (2008). https://doi.org/10.1109/HPCA.2008.4658640
7. Hennessy, J.L., Patterson, D.A.: Computer Architecture, Fourth Edition: A Quantitative Approach. Morgan Kaufmann Publishers Inc., San Francisco (2006)
8. Jerger, N.D.E., Peh, L.S., Lipasti, M.H.: Virtual tree coherence: leveraging regions and in-network multicast trees for scalable cache coherence. In: Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 41, pp. 35–46. IEEE Computer Society, Washington, DC, USA (2008). https://doi.org/10.1109/MICRO.2008.4771777
9. Jiang, N., Becker, D.U., Michelogiannakis, G., Balfour, J., Towles, B., Shaw, D.E., Kim, J., Dally, W.J.: A detailed and flexible cycle-accurate network-on-chip simulator. In: 2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 86–96 (2013). https://doi.org/10.1109/ISPASS.2013.6557149
10. Jin, K., Li, C., Dong, D., Fu, B.: HARE: history-aware adaptive routing algorithm for endpoint congestion in networks-on-chip. Int. J. Parallel Program. (2018). https://doi.org/10.1007/s10766-018-0614-6
11. Kwon, W., Yoo, S., Um, J., Jeong, S.: In-network reorder buffer to improve overall noc performance while resolving the in-order requirement problem. In: 2009 Design, Automation Test in Europe Conference Exhibition, pp. 1058–1063 (2009). https://doi.org/10.1109/DATE.2009.5090821
12. Li, M., Zeng, Q.A., Jone, W.B.: DyXY: a proximity congestion-aware deadlock-free dynamic routing method for network on chip. In: Proceedings of the 43rd annual Design Automation Conference, pp. 849–852. ACM (2006)
13. Lis, M., Cho, M.H., Shim, K.S., Devadas, S.: Path-diverse in-order routing. In: The 2010 International Conference on Green Circuits and Systems, pp. 311–316 (2010). https://doi.org/10.1109/ICGCS.2010.5543048
14. Lotfi-Kamran, P., Rahmani, A.M., Daneshtalab, M., Afzali-Kusha, A., Navabi, Z.: EDXY: a low cost congestion-aware routing algorithm for network-on-chips. J. Syst. Archit. **56**(7), 256–264 (2010). https://doi.org/10.1016/j.sysarc.2010.05.002
15. Ma, S., Jerger, N.E., Wang, Z.: DBAR: an efficient routing algorithm to support multiple concurrent applications in networks-on-chip. In: 2011 38th Annual International Symposium on Computer Architecture (ISCA), pp. 413–424 (2011)
16. Murali, S., Atienza, D., Benini, L., De Micheli, G.: A multi-path routing strategy with guaranteed in-order packet delivery and fault tolerance for networks on chips. In: Proceedings of Design Automation Conference (DAC) pp. 845–848 (2006). http://infoscience.epfl.ch/record/89541
17. Palesi, M., Holsmark, R., Wang, X., Kumar, S., Yang, M., Jiang, Y., Catania, V.: An efficient technique for in-order packet delivery with adaptive routing algorithms in networks on chip. In: 2010 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools, pp. 37–44 (2010). https://doi.org/10.1109/DSD.2010.53
18. Thies, W., Karczmarek, M., Amarasinghe, S.: Streamit: a language for streaming applications. In: Horspool, R.N. (ed.) Compiler Construction, pp. 179–196. Springer, Berlin (2002)

## Affiliations

**Kang Jin[1] · Dezun Dong[1] · Cunlu Li[1] · Libo Huang[1] · Sheng Ma[1] · Binzhang Fu[2]**

Kang Jin
jinkang17@nudt.edu.cn

Cunlu Li
cunluli@nudt.edu.cn

Libo Huang
libohuang@nudt.edu.cn

Sheng Ma
masheng@nudt.edu.cn

Binzhang Fu
fubinzhang@huawei.com

1 National University of Defense Technology, Changsha, China

2 HuaWei, Shenzhen, China