2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

High Frequency Response Tracking System Micro-architecture

Gopalakrishnan Sridhar, Senior Logic Design Engineer/Micro-architect, Intel Technology India Pvt Ltd, Bengaluru, India (*gopalakrishnan.sridhar@intel.com*)

Vadlamuri Venkata Sateesh, Logic Design Engineer, Intel Technology India Pvt Ltd, Bengaluru, India (*vadlamuri.v.sateesh@intel.com*)

Abstract—In order to meet higher performance targets and to keep track of ordering every data transfer protocol used in the SOC has its own rules and features. The implementation of these features differ extensively among the different protocols. There is thus a need to develop various bridges to convert from one protocol to the other. In order to reformat responses in a way required by the requester some of the attributes of the incoming transaction have to be stored by these bridges. Hence there is a need to have a response tracker in the design for this purpose. This paper discusses the complexities involved in the high frequency design of such a tracker.

Keywords— Ordering rules, Protocols, Master/Slave Interface IP, SoC.

I. INTRODUCTION

Different IP's in the SOC follow different protocols due to various reasons. Various protocols has ordering rules governed by tags or IDs. Transactions initiated with different tag/ID can be returned out of order by the completer. Hence there is a need to content access this completion tracker using the response tag as key, to get the attributes of an outgoing transaction.

With growing SOC performance needs, the completion tracker design has to work at very high frequencies and must be able to process transaction without bubbles. This paper discusses the architecture and design of a Response tracking micro-architecture and the various design challenges that were faced and how they were solved.

The Response tracking system discussed in this paper has been used in the IPP (Intel Proprietary Protocol)/PCI to AXI (Advanced Extensible Interface) Bridge, it has been designed to meet clock frequency of 1GHz with 256 outstanding transactions. IPP and AXI are very different in the way they handle data transfer. IPP is a save and forward - credit based protocol. On the other hand AXI works purely based on a valid ready approach. AXI supports narrow and wrap transaction which are not supported by IPP. This requires the protocol converter to split various transactions and manage the completions accordingly. The completion tracker used in the bridge has to hold many attributes (address alignment, number of splits etc.) so that it can reformat the completion accordingly.

The paper discusses the various complexities involved in the design of the completion tracker and how these were handled.

A. IPP/PCI2AXI Bridge Introduction

IPP2AXI Bridge helps an IP with an AXI master Interface talk to an IPP based Intel Interconnect system. The bridge receives requests on its AXI slave interface and converts them to IPP request transaction.

The maximum size of an AXI transaction is 4K bytes but IPP has a 1K byte request data length limitation. Hence there is a need to split transactions received from the AXI IP to split into multiple IPP requests in order to take care of the above scenarios. The split transactions are sent out with the same tag, so that the completions of each split are not reordered.

For the split responses received from IPP Interconnect System, the bridge has to take care of formatting the completion before sending it to the AXI IP. The bridge has to take care of the following before sending the completions to the AXI IP.

I. It needs to check whether all split responses have arrived or not.

II. It has to realign read data as AXI has byte aligned address and IPP has 32bit aligned address.

III. It has to take care of AXI ordering requirement of sending a response for buffarable transaction only after sending the Response for non-buffarable transaction.

The bridge cannot serialize the request transactions to achieve above functional requirements, as it would lead to heavy performance degradation. So, IPP2AXI Bridge needs some tracking system to manage this. The diagram shown in Figure 1, gives an overview of where the Response tracking system sits in the bridge.

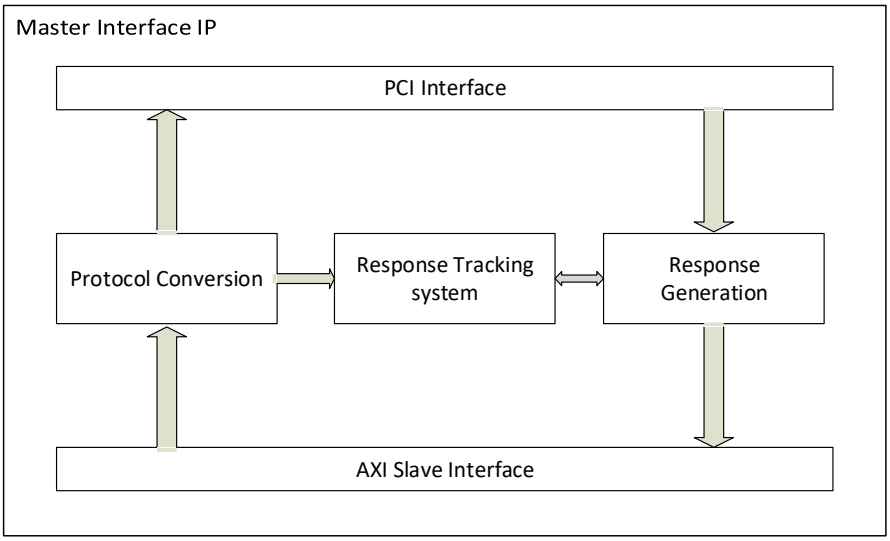

Figure 1: Block Diagram of IPP/PCI-AXI Bridge with Response Tracking System

II. RESPONSE TRACKING SYSTEM

The completion of the requests can be returned out of order based on the tag. Completions of same tag request are returned in order. Hence the response tracker has to be content accessible i.e. it should be able to access the attributes of a request from the tag of the completion as a key.

This section discusses the microarchitecture of a response tracking system. The following are some of the requirements.

1) The attributes of an incoming transaction have to be pushed into the Response tracking System (RTS). These are preserved until the bridge receive the completions for the same. On arrival of the response, RTS provides these attributes to frame the response to the AXI IP.
2) Responses for all Requests with same tag will return in order. Hence the RTS has to provide response attributes for these transactions in the same order as it was pushed into it. A linked list of the attributes has to be maintained by the RTL.
3) Every tag has a link from its first request transaction (head_ptr) that came with this tag to the last request transaction (tail_ptr). RTS could have also been implemented with static queues, one queue has to be allocated for one tag, so that on arrival of response a Queue, which corresponds to the Tag can be popped. This approach is area prone, as some of the tags may not come, so the buffer allocated for that tag will be wasted. To avoid this buffer is allocated dynamically for every request irrespective of the tag and maintained a link between the transactions using linked list. This linked list approach is area efficient.
4) One Head pointer array register, to get the current head pointer for a tag. RTS provides the request attributes which are indexed at that current head pointer.
5) One Tail Pointer array register, to maintain the link between previous transactions with a tag, which went on IPP Interconnect System to the current request of same tag.
6) One valid Register, for every outstanding request, one pointer (index) is used to preserve the request attributes. On a new request valid bit corresponding to that pointer will be set and will be cleared on arrival of its response.
7) The RTS should be able to store the attributes of all the outstanding transactions.

8) Response Tracking System (RTS) block diagram is as shown in the Figure 2.

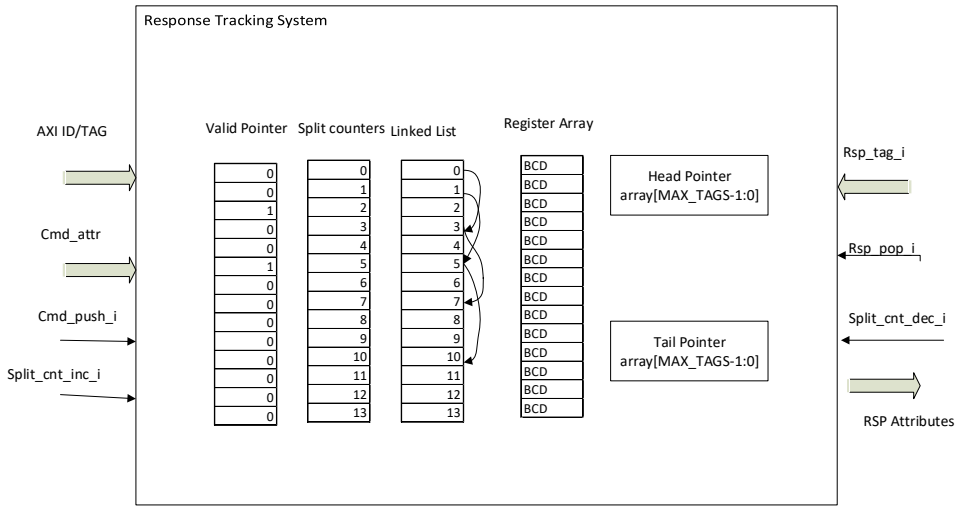

Figure 2: Response Tracking System block diagram

If the AXI request transaction has to split on IPP interface, “split_cnt_inc_i” is asserted for every split. This split count increment pulse increments split counter, which corresponds to that transaction pointer.

On arrival of responses, provide the tag to RTS, so that RTS can give the attributes to frame the response command on rsp interface. For each split response, generate “split_dec_i pulse, so that RTS can decrement split counter, which corresponds to that “rsp_tag”.

The RTS implementation has a few challenges in meeting Higher Frequencies. Bottlenecks in meeting higher frequencies are

1) Responses can come back to back. Response attributes need to be fetched from the RTS and invalidating transaction pointer for the last response has to be done in the same cycle.
2) On Command Interface, using “cam_full” indication to push a command in to RTS.

Pipelining above two paths straight away can corrupt the intended functionality, detailed hypothesis over these bottlenecks are presented along with architectural changes for meeting higher frequencies in the section II.A and section II.B.

A. Back to Back response handling:

As mentioned in the above section II, the RTS should handle split responses. So, all the split responses of a request has to be aggregated and required to send on slave interface. In the response command, some of the request attributes needs to be echoed back in a different format. Slave response interface can get those attributes from RTS by providing “rsp_tag”. The number of splits for a slave request are tracked through split counters for each outstanding transaction. If split count attribute in the RTS reaches a value of “one” for a transaction, then it means that the split response is the last pending response for that slave request. In this case, the transaction pointer has to be invalidated.

Refer to the section II for Response tracking and formation of Response command using RTS. RTS tracks the responses for its outstanding requests. On arrival of response, RTS will provide the required attributes to form a response command and to send on the slave interface. RTS also invalidates transaction pointer on arrival of response, so that it can be used to other request transaction.

Transaction pointers in RTS has to be invalidated, during the response availability of last split command of last request. Response tag is provided to RTS on arrival of split response. RTS gets Head pointer for that tag, and with this head pointer, it gets the corresponding split count. This split count is used to invalidate the transaction pointer or not. If split count reaches one, transaction pointer is invalidated by asserting “rsp_pop_i” port. The assertion of rsp_pop updates the head pointer register, which corresponds to the “rsp_tag”.

Now, critical path in the response path is a kind of a ping pong path into RTS as shown in the Figure 3.

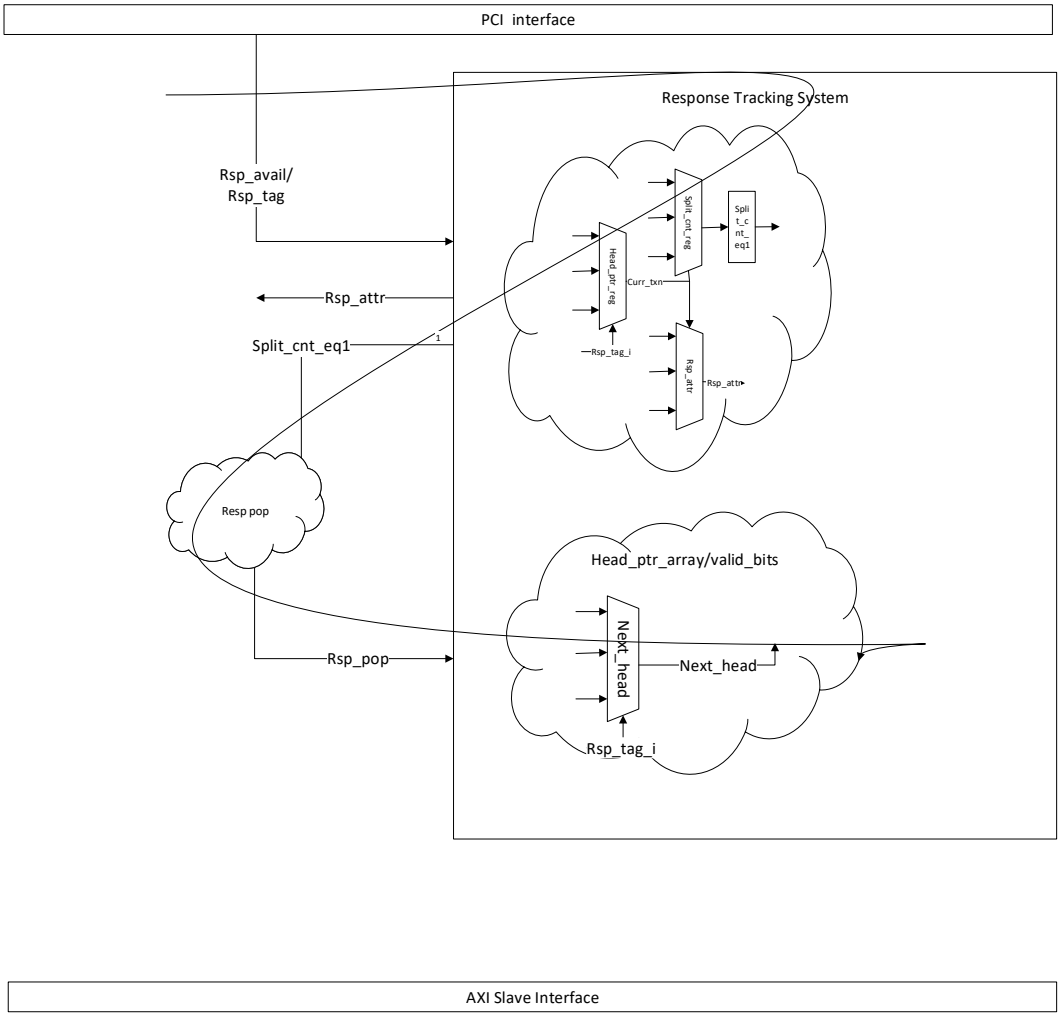

Figure 3: Ping pong critical path on Response side of Response Tracking System

Critical path is “rsp_tag_i” → head_pointer → split_cnt → slit_cnteq1→ rsp_pop_i→ next_head_ptr→ head_pointer_reg.

By seeing the critical path, immediate idea would be to deploy a pipe stage on “rsp_pop” port. The pipestage on “rsp_pop” will not work in this scenario. Responses of a same “rsp_tag” from IPP interface for different requests are arriving one after the other. The first response in this scenario is the last split response and second response is first split response of a different request but the “rsp_tag” is same. As the responses are back to back, the transaction pointer (current head pointer) has to change on the first response. But as the “rsp_pop” is pipelined, it cannot update the transaction pointer for the second response. The solution used to break this timing path without impact on functionality and latency is as shown in the Figure 4.

In response path, If RTS is able to give proper request attributes and “rsp_pop” happens whether in same cycle or in later cycles, it is irrelevant. So, one side pipe is deployed to take of care of “rsp_pop” in the next cycle and multiple pointers are used. One set of pointers is for providing the request attributes and other set of pointers for invalidating the transaction pointer. So, two “rsp_tag”s are given to RTS, one rsp_tag is for response attributes and other “rsp_tag_to_pop” is for invalidating the transaction pointer. Now let us go back to the scenario of not updating the current head pointer and hence given the previous response attributes instead of present response attributes. For first response, it will get the proper attributes for the current response, for 2nd response ideally it has to get the 1st response attributes as the response pop is coming through side pipe. For this case, one signal is send to the RTS to provide the next pointer attributes. So, RTS will provide next pointer attributes instead of current pointer attributes by seeing the assertion of “enable_nxt_ptr_attr”. With this multiple pointers, the functional problem is resolved.

Now the next question is whether next pointer is it not a critical path through “rsp_tag” → curr_head_ptr→ next_head_ptr→split_cnt→ split_cnteq1→ rsp_pop.

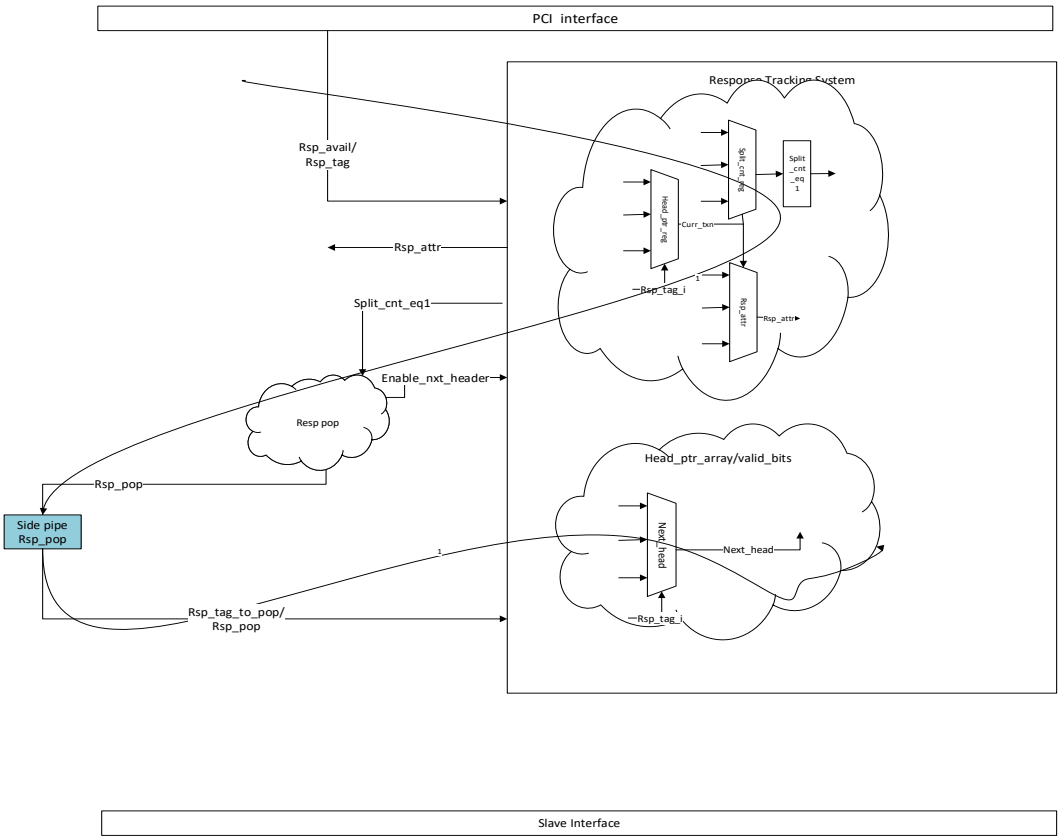

Figure 4: Pipelining critical path A, with multiple response tag for fetching attributes and invalidating entry using a pipeline

The above critical path is not there, as the path is broken by registering “next_head_ptr “. Then the path becomes “rsp_tag” → “curr_head_ptr”→ “next_head_ptr_ff”.

The approach of using multiple pointers and one side pipe for invalidating transaction has resolved the critical timing paths without adding latency in response path. This is the bottle neck in Response Tracking system to operate at higher frequencies.

B. Command push during buffer full:

In the Response tracking system, which is shown in section II has a critical path, which goes into the RTS internal registers from AXI Slave interface is as shown in the Figure 5. Total combinational logic delay in this critical path is sum of inside logic delay (between command interface of RTS and the split_cnt registers/tail_ptr_regiters inside RTS) and outside logic delay (between AXI slave interface and command interface of RTS). This critical path can be short down by inserting a pipe on input command interface of RTS. But it is not done due to functional limitations. The scenario for functional failure is as follows.

RTS generates “cam_full” indication to the AXI salve request interface, so that it can decide whether the transaction can be pushed into RTS or not. If AXI slave request interface pushes a command into RTS through input pipe by seeing the “cam_full” indication, the transaction will be lost. Because there is one cycle delay between cmd_push to and “cam_full” indication. The scenario of losing a transaction happens during last outstanding transaction. So, the pipe on input command interface of the RTL solves timing issues, but it is creating functional issues.

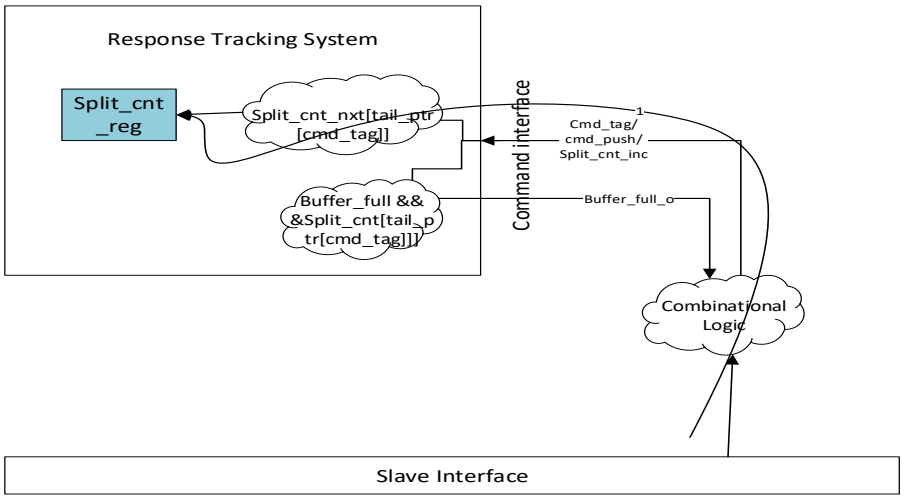

Figure 5: critical path on command interface of Response Tracking System

To Resolve Timing issues on command interface inputs of the RTS, input pipe is deployed. To take care of the functional issues raised by this input pipe is resolved by having one outstanding counter before RTS, to check whether the transaction can be pushed into RTS or not. So after deploying this outstanding counter, "cam_full" indication given by the RTS is not required to check. This approach gives better timing inside the RTS without breaking the functionality. Command interface pipe and outstanding counter is as shown in the Figure 6.

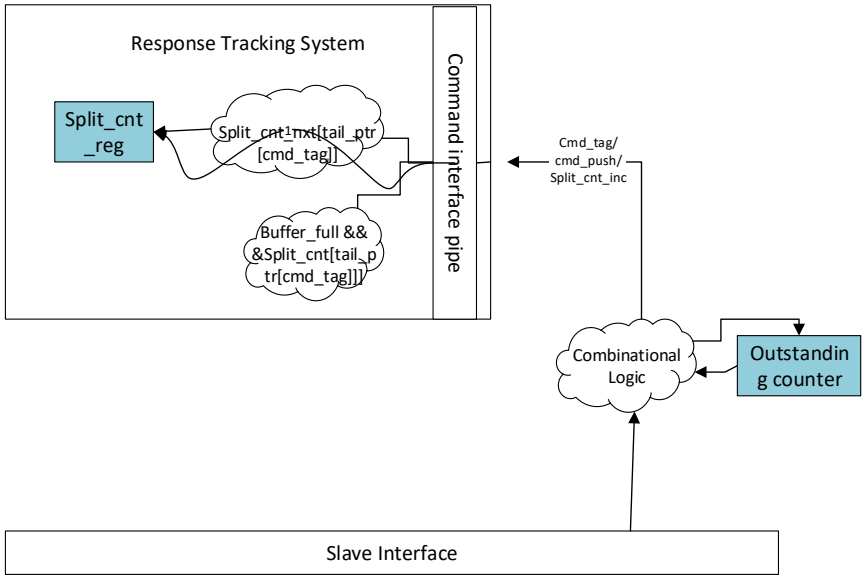

Figure 6: Pipelining critical path on command interface with outstanding counters

This approach doesn't add any additional latency even though there is an input pipe on command interface of RTS. Because, the commands pushed into the RTS are popped during arrival of response for that request. In any SoC or NoC, it takes definitely more than one cycle latency will be there, as there will be multiple pipe stages from Initiator to the Responder and from Responder to the Initiator.

III. SUMMARY

Response tracking system with modified micro-architectural changes mentioned in section II.A and Section II.B integrated in Protocol Bridges doesn't have ping-pong paths, this enables Response tracking system to operate at higher frequencies. So that response tracking system can be deployed across protocol bridges. Timing analysis is performed with a frequency of 1GHz with 256 outstanding transactions.

Table I. WNS for the Critical Paths

Timing path	With High Frequency RTS(WNS)	With RTS(WNS)
Path A	-800ps	-211ps

Path B	-797ps	0ps

ACKNOWLEDGMENT

We thank Sudeep P, for helping in micro-architecting the Skelton of Response Tracking system.

REFERENCES

[1] W. J. Dally and B. Towles, “Bufferd flow control,” in Principles and Practices of Interconnection Networks. San Francisco, CA, USA: Morgan Kaufmann, 2003.

[2] H. Zhang, K. Wang, Y. Dai, and L. Liu, “A multi-VC dynamically shared buffer with prefetch for network on chip,” in Proc. IEEE 7th Int. Conf. Netw., Archit., Storage, Xiamen, China, Jun. 2012, pp. 320–327.

[3] M. Lai, Z. Wang, L. Gao, H. Lu, and K. Dai, “A dynamically-allocated virtual channel architecture with congestion awareness for on-chip routers,” in Proc. 45th ACM/IEEE DAC, Anaheim, CA, USA, Jun. 2008, pp. 630–633