Efficient Support of AXI4 Transaction Ordering Requirements in Many-Core Architecture

BOQIAN WANG 1,2 AND ZHONGHAI LU 1 (Senior Member, IEEE)
1 School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology, 114 28 Stockholm, Sweden
2 School of Computer Science, National University of Defense Technology, Changsha 410073, China
Corresponding author: Zhonghai Lu (zhonghai@kth.se)

ABSTRACT The Advanced Microcontroller Bus Architecture (AMBA) Advanced eXtensible Interface 4 (AXI4) protocol was initially a bus oriented interface designed for on-chip communication. To offer the possibility of utilizing the AXI4 based processors and peripherals in the on-chip network based system, we propose a whole system architecture solution to make the AXI4 protocol compatible with the Network-on-Chip (NoC) based communication interconnect in the many-core architecture. Due to the out-of-order transaction in the NoC interconnect, which conflicts with the ordering requirements specified by the AXI4 protocol, we especially focus on the design of the transaction ordering units, realizing a high-performance and low cost (area) solution to the ordering requirements by the sequence ID (seq_ID) reuse mechanism and a simple but smart seq_ID synchronization process. Besides, the micro-architectures and the functionalities of the transaction ordering units are described and explained in detail for ease of implementation. The experimental results in a C++ based system simulator show that, compared with the state-of-the-art works, our solution can maximally increase the system throughput by 66.0% and decrease the transaction queueing delay in the master-side ordering unit by 91.2%.

INDEX TERMS AMBA AXI4, many-core architecture, Network-on-Chip (NoC), ordering model.

I. INTRODUCTION

Advanced eXtensible Interface 4 (AXI4) is the fourth generation of the Advanced Microcontroller Bus Architecture (AMBA) interface specification from ARM [1]. AXI4 protocol contributes to high-performance, high-frequency system designs. Many processors such as ARM Cortex-A35 and peripherals such as USB and LAN devices follow the AXI4 protocol for communication. The AXI4 was initially a bus-based protocol, utilizing five individual channels to transfer read address and control, read data, write address and control, write data and write response signals. Besides, with the development of semiconductor technology, more and more IP cores can be integrated on chip. Among them, many IP cores follow the AXI4 protocol, and at the same time, their communication demands, such as the latency and the throughput, get higher as the number of IP cores on chip gets larger. However, these communication demands cannot be satisfied by the traditional bus-based interconnect design anymore.

Comparatively, Network-on-Chip (NoC) provides a potential alternative which can enhance the communication among multiple cores. To make the IP cores, such as processors and peripherals following the AXI4 protocol, communicate with each other in the NoC based high-throughput and low-latency interconnect, the NoC system must support the AXI4 protocol.

For example, in 5G infrastructure [2] and 5G access network [3], many processors and peripherals follow the AXI4 protocol and at the same time, they require a high-throughput and low-latency communication service to satisfy the 5G standards, which can be well supported by the NoC based interconnect system instead of the bus based interconnect. In this scenario, we need to design a NoC based communication system architecture to support the AXI4 protocol.

When making the NoC system compatible with the AXI4 protocol, the biggest problem is that, for certain transactions, the AXI4 protocol has ordering requirements on them during their delivery in the interconnect, which will not cause problems in the bus architecture. However, as to the NoC interconnect, many routing algorithms, such as oblivious routing and adaptive routing, and QoS related schemes, such as the
priority service for the latency critical packet, may result in out-of-order transmission in the NoC interconnect, leading to an immense challenge of maintaining the in-order delivery of transactions specified by the AXI4 ordering requirements. Especially for the NoC with high-performance design, the delivery of transactions in NoC will be likely out-of-order so that the NoC performance, such as the latency and throughput, can be opportunistically improved. Even in a NoC with deterministic routing, say X-Y routing in a VC based wormhole network with round-robin VC allocation and switch arbitration, the out-of-order problem may occur. For instance, the whole flits of a shorter packet sent out into the NoC later could still reach the destination ahead of the whole flits of a longer packet sent out into the NoC earlier. To solve the problem caused by the out-of-order communication in many NoC systems, a dedicated ordering mechanism should be designed to support AXI4 protocol under NoC interconnect condition.

In the state-of-the-art works, several many-core architectures and ordering solutions have been proposed to support AXI (AXI3 or AXI4) protocol [4]–[6]. However, these mechanisms are still insufficient. Firstly, they fail to show the whole system architectures with detailed information related to the support of the AXI4 protocol, without which, the designs are hard to implement. Secondly, they do not fully explore the AXI4 transaction ordering requirements and thus, these designs will result in a tremendous decrease of communication performance in throughput and latency. Thirdly, no previous work has considered the seq_ID synchronization issue. That is how to synchronize the sequence records between the master and slave when a series of transactions with delivery sequence have completed. The realization will greatly influence the AXI4 protocol correctness and the hardware consumption. We will detail the deficiency in Section III-B.

In this paper, overcoming the deficiency discussed above, we propose a NoC based many-core whole system architecture with a high-performance low cost (area) solution for the AXI4 ordering requirements to support the AXI4 protocol. The contributions of this paper are listed below.

- We show a whole system design to satisfy the AXI4 protocol in the NoC based many-core architecture. In particular, we detail the micro-architectures and functionalities designed for the transaction ordering requirements so that the related implementation can be easily realized.
- We propose a high-performance AXI4 transaction ordering solution for the ordering units in the network interface NI) by supporting seq_ID reuse mechanism (Section IV-B), which compared with the state-of-the-art designs, can greatly increase the whole system throughput and decrease the transaction’s transfer delay.
- We show and explain the synchronization behavior among the transaction ordering units, which can ensure the correctness of ordering process and simplify the hardware structure of the transaction ordering units.
- We build up a cycle-accurate whole system simulator in C++ to follow the AXI4 protocol and verify our proposed system architecture and test its performance. Besides, we also show the hardware synthesis results of the ordering units for comparison.

We structure the rest of the paper as follows. Related work is discussed in Section II. In Section III, we explain the AXI4 transaction ordering requirements and detail the deficiency in the state-of-the-art works as the background and motivation. Then, in Section IV, we show the whole system design and the micro-architectures and functionalities in the ordering units including the seq_ID reuse and the seq_ID synchronization mechanisms. Experimental results are shown in Section V. Finally, we conclude in Section VI.

II. RELATED WORK

A. AXI COMPATIBLE SYSTEM DESIGN

There are several related works discussing the realization of AXI bus protocol. In [7], [8], based on the bus interconnect, the authors show the FPGA design results of AXI protocol, which mainly focus on the AXI master port and slave port design and implementation. In their consideration, the interconnect is the bus which can maintain the ordering of the transferred signals. Therefore, no ordering related design or unit is utilized in their works.

In [14], the authors discuss the interleaving method in the NoC system employing AXI protocol, which is one aspect of the ordering problem in the AXI protocol. While, the interleaving is only supported in the AXI3 protocol, which has been removed from the support of the AXI4 protocol. So we do not consider the interleaving problem in the state-of-the-art AXI version.

In [9]–[11], the authors take the NoC interconnect into consideration to design an AXI based system in FPGA. In these related works, they show the design results of the NoC-AXI interface and the performance of the whole system. In [12], the authors propose an on-chip network interface, offering backward compatibility with existing bus protocols. In [13], the authors implement the AXI protocol on the 3D System-on-Chips (SoCs) to allow high-performance communication and scalability of the design. While, in their designs, they still do not discuss the most important ordering requirements in the AXI protocol. In reality, in most scenarios, the NoC interconnect cannot offer an ordering service for signals, which is related to many details such as the router micro-architecture, the routing algorithm, the flow control mechanism, etc. Also, in most cases, the ordering unit is required to apply AXI protocol in the NoC system.

B. GENERAL TRANSACTION ORDERING

Flow control and reorder buffers are two typical ways to solve the out-of-order problem. In [17], Murali et al. use flow control to ensure in-order packet delivery. In their design, they utilize a look-up table in each router for ordering check. Once the packets with ordering requirement that traverse
through different paths have arrived at the convergence router, the arbiter in the router will conduct the ordering function according to the look-up table.

Masoud et al. [18] utilize a reorder buffer with a dynamic buffer allocation architecture in the NI to increase the utilization and the overall NoC performance. The proposed NI exploits a streamlined reordering mechanism to handle the in-order delivery and utilizes the advance extensible interface transaction-based protocol to maintain compatibility with existing IP cores. In [19], Du et al. propose an analysis framework for the worst-case reorder buffer size, which is usually needed in multi-path minimal routing NoCs to guarantee in-order packet delivery. The framework can not only model the upper size bound of the reorder buffer for a given application-specific NoC, but also resize the reorder buffer by varying the traffic splitting proportion on sub-flows. Kwon et al. [15] propose an in-network reorder buffer to realize the packets in-order requirements as well as increase the utilization of the reorder buffer resource by moving the reorder buffer resource and related functions from the network interface to the network router. In [16], the authors use transaction ID renaming mechanism to make the original in-order memory accesses independent from each other. They also utilize distributed arbitration to decrease the read delay due to out-of-order data arrivals when dynamic reordering memory read accesses the reorder buffer.

However, all these works consider the transaction ordering problem in a quite general scenario, which can not be directly utilized to support the ordering requirement specified by the AXI4 protocol.

C. AXI TRANSACTION ORDERING

Xu et al. [4] propose an AXI compliant on-chip network interface micro-architecture to offer the transaction reordering processing. It focuses on parts of the AXI ordering requirements by offering the global ordering to the response packet. In [5], Ebrahimi et al. propose a network interface micro-architecture for NoCs using reorder buffer sharing. In the design, a hybrid network interface is presented in both the memory side and the processor side. By dynamic buffer allocation, the resource utilization and performance of the network interface can be improved. In [6], the authors propose a debug aware network interface which is compatible with the AXI standard. This work focuses on the mechanism for cross-trigger debugging, and the ordering micro-architecture in this work is similar to the one in [5].

Focusing on the transaction ordering, the authors in [4]–[6] show a similar architecture of the reorder unit in the NI to realize the data transaction ordering. This is a basic support of the AXI4 ordering requirement under the NoC scenario. They utilize a sequence ID to record the sequence of a series of transactions, which will be referred to accordingly in the NI for the reordering purpose. However, their proposed mechanism is still insufficient to fully adapt to the AXI4 ordering requirements in the multi-processor and NoC-interconnect scenario, which will be detailed in Section III-B. In this paper, we will address the deficiency and take the ordering proposal in these state-of-the-art works as a baseline for comparison.

III. BACKGROUND AND MOTIVATION

A. AXI4 ORDERING REQUIREMENTS

In this part, as the background, we describe the transaction ordering requirements in the AXI4 protocol. The AXI4 protocol uses transaction ID (AXI4_ID) to indicate its ordering requirements. Except for the write data channel, other four channels (the read address channel, the read data channel, the write address channel and the write response channel) have their own AXI4_ID signals which are separately named as the Read Address ID (ARID), Read ID tag (RID), Write Address ID (AWID) and Response ID tag (BID). The AXI4 transaction ordering requirements can be explained below [1]:

- Transactions from different masters have no ordering restrictions. They can complete in any order.
- Transactions from the same master, but with different AXI4_ID values, have no ordering restrictions. They can complete in any order.
- There are no ordering restrictions between read and write transactions using a common value for AWID and ARID.
- The data transfers for a sequence of read transactions with the same ARID value must be returned in the order in which the master issued the addresses.
- The data transfers for a sequence of write transactions with the same AWID value must complete in the order in which the master issued the addresses.

In summary, we can simplify the transaction ordering requirements below:

- Transactions with the same AXI4_ID, the same read/write type and from the same master node must be completed in the order in which the master issued the transactions.
- Transactions with different AXI4_IDs or different read/write types or from different master nodes have no ordering requirement for completion.

The IDs shown or implied in the AXI4 transaction ordering requirements are: The AXI4_ID specified in the AXI4 signals, the master ID (M_ID) indicating the ID of the transaction generator node in the many-core system, and the read/write ID (r/w_ID) determined by the read/write type of the transaction.

B. DEFICIENCY IN THE STATE-OF-THE-ART WORKS

To support the AXI4 ordering requirements, in the state-of-the-art works [4]–[6], they utilize an additional sequence ID (seq_ID) to record the sequence of a series of transactions to ensure in-order delivery in case of possible out-of-order communication in the interconnect. However, when this mechanism comes to the multi-core scenario, it will greatly decrease the system performance, such as the throughput and latency, increase hardware consumption, and even cause...
execution error due to its insufficient support for the AXI4 ordering requirements.

The first problem is caused by the seq_ID conflict. As shown in Figure 1, a master could send out two transactions with the same AXI4_ID, M_ID and r/w_ID to different slaves. This is because the AXI4_ID is decided by the processor who has no idea which slave the requested memory address will be mapped to. In this case, the first transaction to the slave 0 will be assigned with the seq_ID 0 and the second transaction to the slave 1 with the seq_ID 1. This process will cause system error in the slave 1. The slave 1 has to block the second transaction (permanently) and wait for the first transaction in this request series (the next deliverable seq_ID in this node is still 0), which has already been sent to the slave 0 and will never arrive at the slave 1. To avoid system error, in the state-of-the-art works, the second transaction has to be blocked in the master ordering until the finish of the first transaction’s round-trip transfer and then, the second transaction can be sent out with seq_ID 0 (because there is no more transaction of this series in NoC right now). This settling will greatly increase the queueing delay of the second transaction in the master and decrease the system throughput. The scenario gets worse when the injection rate of a master node gets higher, which will usually cause more seq_ID conflict scenarios.

To overcome this drawback, we involve the slave ID (S_ID) to record the slave node that the transaction targets at. All the seq_ID values will be recounted from 0 for the transactions that target different slaves even if they have the same AXI4_ID, M_ID and r/w_ID. In this way, the seq_ID can be reused among transactions and therefore, we call it the seq_ID reuse mechanism. By the seq_ID reuse mechanism, we can avoid the conflict among transactions, increasing the system throughput and decreasing the transaction’s queueing delay in the master.

The second problem is seq_ID mismatch caused by an inappropriate seq_ID synchronization mechanism. The seq_ID synchronization is a mechanism to ensure that the seq_ID record in the slave-side ordering unit can always match the assigned sequence of the seq_ID record in the master-side ordering unit. Only by an appropriate seq_ID synchronization mechanism can the next deliverable seq_ID recorded in the slave-side ordering unit maintain the correct value.

In Figure 2, we use an example to explain the seq_ID mismatch problem. In the ordering mechanism, the slave-side ordering unit needs to record the information of the next deliverable seq_ID, which will be updated after each acceptance of transaction. In AXI4, the maximal value of the seq_ID is 15 (from 0 to 15) to support 16 in-order transactions at the same time. The process causing the seq_ID mismatch problem is listed below.

1) Three previous transactions from the master 0 to the slave 0, which belong to an ordering series, have completed. The record of the next deliverable seq_ID will be 3 in the slave 0.
2) Then, a new transaction in this series is generated by the master 0, which has the same AXI4_ID, M_ID, r/w_ID, and target (slave 0) with the previous three transactions.
3) Because all the previous transactions have completed and the previous seq_ID records in the master-side ordering unit have been removed, the seq_ID of this new transaction will be assigned 0 (from the beginning).
4) But at this moment, the slave 0 is still waiting for the next deliverable transaction with seq_ID 3 and thus, will block this new transaction with the seq_ID 0.

This seq_ID mismatch problem is caused by the missing of an appropriate seq_ID synchronization mechanism between the master-side and the slave-side ordering units when all previous transactions in a series have completed. However, in all the state-of-the-art solutions, such as [4]–[6], they did not address this problem, which could result in a system error. Besides, the solution to this problem will also influence the hardware consumption.

To overcome this drawback, we propose a high-efficiency seq_ID synchronization method to ensure the correctness of the ordering mechanism and at the same time, decrease the hardware consumption.

Motivated by the problems mentioned above, in the following section, after introducing the overall system design, we show the seq_ID reuse mechanism and detail the microarchitectures and functionalities of the ordering units that support it. Then, we discuss and show the seq_ID synchronization method.

IV. SYSTEM DESIGN
A. OVERALL SYSTEM DESIGN
We show the overall design of the AXI4 based many-core system in Figure 3. The master and the slave (processor/peripheral) follow the AXI4 transaction protocol. In our
design, each master node has a master-side AXI4 transaction ordering unit connected to the AXI4 master port and a slave-side AXI4 ordering unit connected to the AXI4 slave port. In this way, a master (processor) can act as the master to send out requests and at the same time, as the slave to response requests. Comparatively, the slave node (peripheral) only has a slave-side AXI4 ordering unit connected to the AXI4 slave port so that it can only act as the slave to response requests.

In the AXI4 master/slave port, five individual AXI4 channels, which are the read address channel (RA), the read data channel (RD), the write address channel (WA), the write data channel (WD) and the write response channel (WR), are utilized to transfer different types of AXI4 signals. The RA carries the read request transaction and the RD carries the read response transaction. The WA and WD hold the write request transaction, the signals in which will be combined together before they reach the master-side ordering unit. Correspondingly, the write request transaction will be split into the write address signals and the write response signals after it leaves the slave-side ordering unit. The WR holds the write response transaction.

Both the master-side and slave-side ordering units are connected to the interconnect via the network interface (NI) where the AXI4 transaction will be packed or depacked. For the transaction to the interconnect, we will pack the routing information into the AXI4 transaction to form a packet, and for the packet to the interconnect, we will depack the routing information to from the packet to extract the AXI4 transaction.

In this system design, we apply the ordering and message format process in the node side, offering great independence for the high-performance NoC interconnect design. However, due to the out-of-order transaction in NoC, the biggest challenge for the system is the design of the high-performance ordering units.

The rest part of this section will focus on the ordering units, which includes the description of the seq_ID reuse mechanism, the micro-architectures and functionalities of the ordering units, and the seq_ID synchronization method.

### B. SEQ_ID REUSE MECHANISM

In Figure 4, for a better understanding of the seq_ID reuse mechanism, we firstly show and explain the utilization of the different IDs in the different transfer stages. We can see that, except for the AXI4_ID, M_ID, r/w_ID and seq_ID, we additionally utilize a slave ID (S_ID) to support the seq_ID reuse mechanism. The S_ID shows the destination of a transaction and in our design, the seq_ID is utilized to record the sequence number of a transaction among a series of transactions which have the same AXI4_ID, M_ID, S_ID and r/w_ID.

All the IDs will be transferred in the interconnect. The master adds and removes the M_ID, r/w_ID, S_ID and seq_ID and the slave the r/w_ID, S_ID and seq_ID. In this slave side, the AXI4_ID and the M_ID will be treated together as a new AXI4_ID utilized only in the slave.

In each master-side ordering unit (the same M_ID), the transactions with the same AXI4_ID and r/w_ID will form
a master ordering list (the master ordering list 0 and the master ordering list 1 in this case). In each master ordering list, the seq_ID will be calculated independently for each slave ordering list where all transactions also need have the same S_ID. In this case:

1) The master ordering list 0 is divided into two slave ordering lists separately recorded in the slave-side ordering units of the slave 0 and the slave 1.
2) In the slave 0, the seq_IDs of these transactions vary from 0 to 2.
3) In the slave 1, the seq_IDs of these transactions vary from 0 to 1.
4) Since all the transactions in the master ordering list 1 have the same target (slave 1), it has only one corresponding slave ordering list, which is recorded in the slave 1.

In the seq_ID reuse mechanism, the master (processor) restricts the acceptance sequence of the transactions from the master-side ordering unit according to the master ordering list and the slave (memory or peripheral) restricts the acceptance sequence of the transactions from the slave-side ordering unit according to the slave ordering list.

In this way, in the master ordering list 0, the transactions to the slave 0 will not block the transactions to the slave 1. They can transfer in the interconnect at the same time. However, as we explained in Section III-B, in the state-of-the-art mechanisms, the transaction to the slave 1 will be blocked in the master-side ordering unit all transactions to the slave 0 have finished their round-trip transfers. By solving this problem, our proposed seq_ID reuse mechanism can greatly decrease the transactions’ queuing delay in the master side and thus, immensely increase the system throughput. Besides, because no cyclic dependence will be generated by our proposed seq_ID reuse mechanism, our system is deadlock free.

The AXI4 transaction ordering requirements and the seq_ID reuse mechanism are realized by the cooperation of the master-side and slave-side ordering units. Next, we will show the micro-architectures and functionalities of the master-side and slave-side ordering units in details.

C. MASTER-SIDE ORDERING UNIT

The main job of the master-side ordering unit is to maintain the master ordering list and reorder all transactions according to it. Besides, it also needs to decide the seq_ID for the establishment of the slave ordering list that will be utilized in the slave-side ordering unit.

The block diagram of the master-side ordering unit is shown in Figure 6, which contains the Sequence Number Reference Table, Head Address Register, Sequence Status Holding Register and Re-order Buffer. The Sequence Number Reference Table is utilized to decide appropriate seq_ID for the transaction to the interconnect which will be utilized by the slave ordering list in the slave-side ordering unit. The Head Address Register holds the address of the first record in each master ordering list and the Sequence Status Holding Register holds all the records of each master ordering list. The Re-order Buffer stores transactions from the interconnect and reorders their delivery sequences according to the master ordering list.

By the cooperation of these four units, the AXI4 master-side ordering requirements and the seq_ID reuse mechanism are realized. In detail, their functions are listed below:

- Sequence Number Reference Table: It is utilized to determine the expecting seq_ID of the transaction from the AXI4 master port. The seq_ID is calculated independently for each slave ordering list and thus, the seq_ID mechanism is realized by the Sequence Number Reference Table. The corresponding master ordering list information will be recorded in the Head Address Register and Sequence Status Holding Register and then, the transaction is delivered to the interconnect.
- Sequence Status Holding Register: It records the master ordering lists of unfinished round-trip transactions. For transactions with the same ID (r/w_ID plus AXI4_ID), a master ordering list records their delivery sequences to the interconnect which will finally be utilized by the Re-order Buffer for reordering control.
- Head Address Register: It is utilized to record the head address of each master ordering list in the Sequence Status Holding Register. One record points to one master ordering list. The Head Address Register will be referred to by the Sequence Status Holding Register to record an item in the corresponding master ordering list and by the Re-order Buffer to find out the corresponding master ordering list.
- Re-order Buffer: It stores all transactions from interconnect and reorder their delivery sequences via the master ordering list information recorded in the Sequence Status Holding Register. The AXI4 transaction ordering requirement in the master side is realized by the sequence records in the master ordering list.

The micro-architecture of master-side ordering unit is shown in Figure 7. For all transactions from the AXI4 master port, on the one hand, the master-side ordering unit is responsible for appending the appropriate r/w_ID (1 bit),
FIGURE 7. The ordering unit in the master side.

M_ID, S_ID and seq_ID to each transaction, which will then, be utilized in the slave-side ordering model to determine their delivery sequences to the slave. On the other hand, the master-side ordering unit will reorder the response transactions according to the **master ordering lists**. The steps to process the transaction from the AXI4 master port are shown below:

1. **Determine r/w_ID, M_ID, and S_ID:**
   - Determine the r/w_ID by the AXI4 channel the signal comes from and determine S_ID by mapping the read/write address signal to certain slave ID. The M_ID is decided by the master ID. Each master has a unique M_ID.

2. **Determine seq_ID:**
   - Search the valid items in Sequence Number Reference Table by the index of the r/w_ID, ARID/AWID (AXI4_ID, 4 bits, specified by the AXI4 protocol) and S_ID to determine an appropriate seq_ID. If a previous record is found, update this item by adding 1 to the seq_ID. Otherwise, establish an item with the corresponding index and initial seq_ID to 0 and at the same time, set the valid tag to 1 to validate this item.
   - Then, append the seq_ID to the transaction. The seq_ID records the sequence of a transaction in a **slave ordering list**.

3. **Update record in the Head Address Register and Sequence Status Holding Register:**
   - Search the Head Address Register by the ID (r/w_ID plus ARID/AWID). If the valid tag of the corresponding item is 0 (invalid), create a new record, which will be the first record in a **master ordering list** in the Sequence Status Holding Register and point the Addr of this item to the new record to record the head address of the master ordering list. For the new record in the Sequence Status Holding Register, its valid tag will be set to 1 (valid), the S_ID and the seq_ID will be set to the corresponding values in the Sequence Number Reference Table. The record pointer will point to itself to implicitly show the end of the master ordering list and the data pointer validation (DP_v) tag will be set to 0 to represent that there has not been a response, which is related to this record, received by the Re-order Buffer. If the valid tag of the corresponding item in Head Address Register is 1 (valid), then just create a new item in the Sequence Status Holding Register in the same way and make the original last record in this master ordering list point to it.

   For example, in the case shown in Figure 7, the transaction is the second one in the master ordering list and its r/w_ID is 0 and AXI4_ID is 0b0001. It will firstly refer to the Sequence Number Reference Table to look up the corresponding record. We can find out that the transaction is the second one in the corresponding slave ordering list. The seq_ID in this record increases one and the result (1) is appended to the transaction as its seq_ID. Then, according to its ID (r/w_ID plus AXI4_ID), which equals to 0b00001, look up the Head Address Register to find out the first record address of the master ordering list in the Sequence Status Holding Register, which, in this case, equals to 0xf. Finally, create a new item in an available address 0 × 1 in the Sequence Status Holding Register and append it to the previous last record of the master ordering list. In this case, the transaction is the third one in the corresponding master ordering list. The S_ID and seq_ID are both set to 1, which are the same as the records in the Sequence Number Reference Table. The item’s pointer points to itself to show the end of the master ordering list and the data pointer validation is set as 0 (invalid).

For the response transaction from the interconnect, the processes in the master-side ordering unit are described below:

1. **Store the response transaction in the Re-order Buffer.**
   - The transaction is divided into small parts with identical length and each part of the transaction are stored in the Re-order Buffer via the link structure.

2. **Find record in the Sequence Status Holding Register:**
   - Via the Head Address Register, find the corresponding master ordering list according to the transaction’s ID, which equals to the r/w_ID plus the RID/BID (AXI4_ID, 4 bits, the
same as the corresponding ARID/AWID). Then, according to the S_ID and seq_ID, find its corresponding record in this **master ordering list** in the **Sequence Status Holding Register**. Pass the head-part address of the transaction in the **Re-order Buffer** to the data pointer (DP) of the record and set its data pointer validation (DP_v) tag to 1 (valid).

(3) Deliver the first record:

For each **master ordering list** in the **Sequence Status Holding Register**, check the first record’s data pointer validation tag. If it is 1 (valid), deliver its corresponding response transaction in the **Re-order Buffer**, which is pointed by the data pointer, to the AXI4 master port. Then, invalidate the record. If there is no record in the **master ordering list** anymore, invalid the head-address record in the **Head Address Register**. Otherwise (at least one record exists), update the Addr value in the **Head Address Register** to the address of the next record in the **master ordering list**.

For example, in the case shown in Figure 7, the ID (r/w_ID plus RID/BID) of the response transaction equals to 1. Firstly, according to the ID, look up the **Head Address Register** to find out the first record address (0xf) of the **master ordering list** in the **Sequence Status Holding Register**. Then, according to its S_ID (1) and seq_ID (0), find its position in the **master ordering list** and set its data pointer to 0 × 00 to point to the head-part of the response transaction and set the data pointer validation to 1 (valid). We can find out that the response transaction is the first one in the **master ordering list**, so the response transaction will be delivered to the AXI4 master port and the record in the **Sequence Status Holding Register** will be removed. Finally, the Addr in the **Head Address Register** points to the new head (0 × 0) of the **master ordering list**.

### D. SLAVE-SIDE ORDERING UNIT

The main job of the slave-side ordering unit is to maintain the **slave ordering list** and reorder all transactions according to it. Besides, it also needs to assign the seq_ID to each response transaction, which should be the same as the value in its corresponding request transaction.

The block diagram of the slave-side ordering unit is shown in Figure 8, which also contains the same four substructures. The **Sequence Number Reference Table** is utilized to decide appropriate seq_ID for the response transaction to the interconnect which must be the same as its request transaction. The **Head Address Register** holds the address of the first record and the next deliverable seq_ID in each **slave ordering list**. The **Sequence Status Holding Register** holds all the records of each **slave ordering list**. The **Re-order Buffer** stores transactions from the interconnect and reorder their delivery sequences according to the **slave ordering list**.

In details, their functions are described below:

- **Sequence Status Holding Register**: It records the non-delivered (to-slave) transactions’ **slave ordering lists**. For transactions with the same r/w_ID, AXI4_ID and M_ID, the **slave ordering list** is linked according to their **arrival sequence** in the slave-side ordering unit and the last record will point to itself to implicitly show the end of the list.

- **Head Address Register**: It is utilized to record the first record address of each **slave ordering list** in the **Sequence Status Holding Register**. It also records the seq_ID value that should be delivered next for each **slave ordering list**. The AXI4 ordering requirement in the slave side is maintained by the cooperation of the **Head Address Register** and **Sequence Status Holding Register**.

- **Re-order Buffer**: It is utilized to store all transactions from the interconnect, which are waiting to reorder their delivery sequences via the **slave ordering list** information in the **Head Address Register** and **Sequence Status Holding Register**.

- **Sequence Number Reference Table**: It is utilized to determine the expecting seq_ID of certain response transaction from the AXI4 slave port, which, for each response transaction, should be the same as the seq_ID in its corresponding request transaction from the interconnect.

We show the micro-architecture of slave-side ordering unit in Figure 9. For all transactions to the slave, on the one hand, the slave-side ordering unit is responsible for reordering the transactions from the interconnect via the **slave ordering list** information. The transactions with the same AXI_ID, r/w_ID and M_ID, that is in a same **slave ordering list** will be delivered to the AXI4 slave port according to their seq_ID. On the other hand, the slave-side ordering unit is also responsible for appending the r/w_ID, M_ID, S_ID and seq_ID to the response transaction. In this way, the response transactions delivery sequences can be correctly maintained in the master-side ordering unit via the **master ordering list**.

The steps to process the transactions from the interconnect are described below:

1. Store the request transaction in the **Re-order Buffer**.
2. Find the **slave ordering list** and record information in the **Head Address Register** and **Sequence Status Holding Register**.

Look up the **Head Address Register** according to the transaction’s ID (AXI4_ID plus r/w_ID plus M_ID). If no record exists, create a new record in the **Head Address Register** and
the Sequence Status Holding Register. For the record in the Sequence Status Holding Register, set it valid tag to 1 and the seq_ID to the value in the transaction. Make its record pointer point to itself and its data pointer point to the address of its head-part or head/tail-part (H/T) in the Re-order Buffer. For the record in the Head Address Register, set its valid tag to 1 and the AXI4_ID, r/w_ID and M_ID to the values in the transaction. Initialize the seq_ID as 0, which shows the next deliverable transaction’s seq_ID value in this slave ordering list and point Addr to the record address in the Sequence Status Holding Register. If a record exists, just create a record in the Sequence Status Holding Register in the same way and append it to the end of the corresponding existing slave ordering list.

(3) Check deliverable transaction:

For each slave ordering list recorded in the Sequence Status Holding Register, if a record’s seq_ID equals to the seq_ID recorded in the Head Address Register, which shows the next deliverable transaction’s seq_ID value in this slave ordering list, deliver the corresponding transaction and remove the record in the Sequence Status Holding Register. The seq_ID in the Head Address Register increases one to show the next deliverable transaction’s seq_ID.

For example, in the case shown in Figure 9, the ID (AXI_ID plus r/w_ID plus M_ID) and seq_ID of the request transaction from the interconnect are separately 1 and 0. And a transaction with the same ID but different seq_ID (1) has come before it. Firstly, according to its ID, find the first record’s address in its corresponding slave ordering list, the result of which is 0 × 0. Then, create a new item in the available address 0xf in the Sequence Status Holding Register and append it to the previous last record of the slave ordering list. The seq_ID is the same as the one in the request transaction. The record pointer points to itself to show the end of the slave ordering list and the data pointer points to the head-part of the transaction in the Re-order Buffer. Finally, check each slave ordering list. We can find that the slave ordering list with ID 1 has a record, the seq_ID (0) of which equals to the next deliverable transaction’s seq_ID (0) recorded in the Head Address Register. So, the corresponding transaction in the Re-order buffer, which is pointed by the record, will be delivered from the Re-order Buffer to the AXI4 slave port and the record in the Sequence Status Holding Register will be removed from the slave ordering list. The seq_ID in Head Address Register increases one and thus, the transaction with seq_ID 1 will be delivered next.

In the many-core system, the AXI4 ID utilized in the slave port is wider than the one in the master port, which additionally includes the M_ID. The slave node follows the AXI4 protocol so that its response transactions’ delivery sequences will be the same as their arrival sequences of the corresponding request transactions. For each response transaction from the slave port, it will inherit the AXI4_ID and M_ID from its request transaction. Then, the processes in the slave-side ordering unit are described below:

(1) Determine r/w_ID and S_ID:

Determine the r/w_ID by the AXI4 channel the signal comes from and the S_ID by the current slave ID. Each slave has a unique S_ID

(2) Determine the seq_ID in the transaction

Look up the Sequence Number Reference Table according the ID (AXI4_ID plus r/w_ID plus M_ID). If no record exists, create a new one with the same ID and set its seq_ID value to 0. If a record exists, the seq_ID value increases one. Then, tag the seq_ID into the response transaction, which will make the seq_ID in the response transaction the same as the seq_ID in its corresponding request transaction from the interconnect. This is based on the fact that, in the AXI4 based slave, the
response transactions will be sent out in the same sequences as the input of their corresponding request transactions.

(3) Append the seq_ID to the response transaction and send it out to the interconnect.

E. SEQ_ID SYNCHRONIZATION

In this part, we first detail the necessity of seq_ID synchronization between the master-side and slave-side ordering units. Then, we explain and compare two solutions separately, which are transaction-based synchronization and loop-based synchronization.

As discussed above, three seq_IDs are utilized throughout the transaction’s transfer process in the master-side and slave-side ordering units. For the seq_ID in the Sequence Number Reference Table in the master-side ordering unit, it records the last delivered-to-interconnect request transaction’s seq_ID. The seq_ID in the Head Address Register in the slave-side ordering unit records the next deliverable-to-slave request transaction’s seq_ID. For the seq_ID in the Sequence Number Reference Table in the slave-ordering unit, it records the seq_ID value that has been tagged into the last delivered-to-interconnect response transaction’s seq_ID. In our mechanism described above and the mechanisms proposed by the state-of-the-art works, before a slave ordering list completes, the seq_IDs can perfectly cooperate with each other to maintain the correct value in each unit, ensuring the correctness of the ordering process.

But, when all transactions in a slave ordering list complete, we must provide a way to synchronize the three seq_ID values recorded in the three different units so that the correctness of the ordering process can be still maintained. This problem will also happen in the state-of-the-art works when a series of transactions with ordering restriction have completed in the master. However, this important synchronization process has not been mentioned in any related work.

To solve the synchronization problem in the many-core system, we could utilize transaction to transfer the synchronization information, which we call the transaction-based synchronization. In the transaction-based synchronization, when all transactions in a slave ordering list have completed, a special transaction will be generated by the Sequence Number Reference Table in the master-side ordering unit and sent out to the target slave to inform it of the end of the transaction sequence. The slave-side ordering unit will reset its seq_ID records in the Head Address Register and the Sequence Number Reference Table to 0. In this way, the seq_ID records in the three units can match each other to serve a new series of transactions, the seq_IDs of which begin from 0.

In the transaction-based synchronization, additional synchronization transaction generator and reset controller should be separately involved in the master-side ordering unit and the slave-side ordering unit, which will increase the hardware consumption. Besides, the additional synchronization transaction will influence the other transactions in the NoC interconnect, which will decrease the NoC performance.

FIGURE 10. Loop-based synchronization.

To overcome the drawbacks caused by the transaction-based synchronization, we propose another method utilizing loop-based synchronization. As shown in Figure 10, three seq_ID values in three different units follow the same update equation. They will firstly be initialized as 0 when the system starts. For each arrival transaction, the seq_ID increases one, like the process we explained in Section IV-C and Section IV-D. When the seq_ID reaches its maximal value, the seq_ID returns to 0. The continual seq_ID values will form a local loop and the three seq_IDs in the three different units will automatically synchronize with each other in this way. In the loop-based synchronization mechanism, the master-side and slave-side ordering units do not care whether all transactions in a slave ordering list are completed or not. Even if all the previous transactions in a series have completed in the master-side ordering unit, it will still record the last seq_ID value and for a new coming transaction in this series, it will assign the transaction the seq_ID based on the last seq_ID in record, which is different from the mechanism in the state-of-the-art works where the seq_ID will start from 0 again for the new coming transaction and finally makes the seq_ID records in the master-side and the slave side ordering units mismatch with each other. In our loop-based synchronization, for each transaction, it just finds its associated slave ordering list and continues to utilize the seq_ID of the slave ordering list and finally, the values will form a local loop from 0 to the maximal value (15) and then back to 0.

The advantage of the loop-based synchronization mechanism is its independence from additional synchronization information in the transaction, which can decrease the transaction length or increase the valid information in the transaction. Besides, as the loop-based synchronization is based on the original seq_ID control logic, no more hardware unit should be involved, which can decrease the hardware complexity in the master-side and slave-side ordering units compared with the transaction-based synchronization.

V. EXPERIMENTS

A. METHODOLOGY

We utilize a behavior-level cycle-accurate simulator in C++, which supports the AXI4 transaction generation and ordering, to verify the functionality and analyze the performance of our proposed many-core system. To verify the correctness of the simulator, we will first show its validation in Section V-B.
Our proposed ordering solution is independent from NoC interconnect topologies, such as the mesh, torus, hybrid, 3D mesh, etc. Since different topologies can only influence the baseline results of the packet throughput and latency in NoC and we mainly focus on the network interface not the NoC architecture, we take the mesh topology as an implementation example in our experiments. We draw the architecture of the simulator in Figure 11. The interconnect is an $8 \times 8$ 2D mesh architecture with each master node (M) connected to a router. Four slave nodes are connected to four routers in the borders. The master node simulates the processor node and the slave node simulates the peripheral in real system. The detailed structures of the master node and the slave node are shown in Figure 12. The master node has an AXI4 traffic generator, which is connected to the master-side ordering unit via the AXI4 master port, and an AXI4 traffic receiver, which is connected to the slave-side ordering unit via the AXI4 slave port. In this way, the master node can act as either a master or a slave. The slave node consists of an AXI4 traffic receiver and a slave-side ordering unit, which are connected via the AXI4 slave port.

Since there is no available traffic trace following AXI4 protocol from the real application, in our design, the behavior of the AXI4 traffic generator is controlled by the Markov Modulated Process (MMP) (Chapter 24.2.2 Synthetic Workloads in [20]) to generate synthetic traffic that follows AXI4 protocol. The related parameters are shown in Table 1. The data size in the transaction is 128 bytes by default, which equals to the widest data bus supported in the AXI4 protocol. For ease of implementation, the ratio of read and write transactions (R-W ratio) generated by the AXI4 traffic generator is set as 50%-50% and the target of the transaction is set randomly. In real system, the average response delay (without queueing delay) in L1 cache in the master side is around 6 cycles and in L2 cache in the slave side is around 20 cycles. Based on this, the AXI4 traffic receiver simulated by a FIFO queue in the master node has a response delay of 6 cycles and in the slave node has a response delay of 20 cycles. The ordering units are structured according to Figure 7 and Figure 9. The transaction’s transfer latencies in the ordering units for the master-side request transaction (M_req), the master-side response transaction (M_resp), the slave-side request transaction (S_req) and the slave-side response transaction (S_resp) are separately 3 cycles, 3 cycles, 3 cycles and 2 cycles, which equals to their corresponding numbers of pipeline stage specified in the synthesis process. In the interconnect NI, a transaction packing unit is utilized to pack routing information into the transaction to the interconnect and a transaction depacking unit is utilized to depack routing information from the transaction sent out by the interconnect.

For the interconnect system, we utilize the mesh NoC with the virtual-channel (VC) based wormhole router as an example to implement the interconnect. The interconnect is built according to the architecture and functionality of Garnet in Gem5 [21]. But instead of the event triggered model that Garnet utilizes in the full system simulator, we use the time triggered model, like BookSim [22], which is more suitable for the cycle-accurate simulation control and result collection. In the interconnect, we utilize the router with a two-stage pipeline (as in many state-of-the-art works such as [23]–[25]) as an example, and the transaction’s transfer latencies in a router and a link are both 2 cycles. Each output port contains 2 virtual networks (VNs), separately utilized by the response and request transactions to avoid protocol-level deadlock in the real system. Each VN has 4 virtual channels (VCs) and each VC can maximally store 5 flits. The unidirectional link width between each router (a flit)
is 128 bits (16 bytes). We utilize round-robin method for the arbitration of VC and switch and the input-first control logic for the switch allocation among the input ports and the output ports.

**B. EXPERIMENTAL RESULTS**

1) **CORRECTNESS VERIFICATION**

The ordering units should follow the AXI4 transaction ordering requirements explained in Section III. Transactions with the same AXI4_ID, r/w_ID and M_ID must complete in the order in which the master issued them and transactions with different AXI4_IDs or r/w_IDs or M_IDs have no ordering requirement for completion.

To test the correctness of our proposed ordering units, we design 6 AXI4 transactions, by which we can verify each scenario specified in the AXI4 transaction ordering requirements. As shown in Figure 13, the transaction_1, the transaction_4 and the transaction_5 have different AXI4_IDs, r/w_IDs and M_IDs from their “opponents”, which are separately the transaction_0, the transaction_2 and the transaction_4. So there is no ordering requirement between them and their “opponents” both in the master-side and slave-side ordering units. The transaction_0 and the transaction_3 have the same AXI4_ID, r/w_ID and M_ID and target the same slave, so there are ordering requirements for them both in the master-side and slave-side ordering units. The transaction_0 and the transaction_3 have the same AXI4_ID, r/w_ID and M_ID and target the same slave, so there are ordering requirements for them both in the master-side and slave-side ordering units. As for the transaction_0 and the transaction_2, although they have the same AXI4_ID, r/w_ID and M_ID, they target the different masters (M4 and M9). So they have the ordering requirement only in the master-side ordering unit. Obviously, the transaction_0 together with the transaction_2 shows an example of our proposed seq_ID reuse mechanism where transactions with the same r/w_ID and M_ID, but targeting different slaves, can be assigned with the same AXI4_ID.

In details, during the test, the transaction_1, transaction_4 and transaction_5 are sent out by the master-side NI later than their “opponents”. Although they arrive at the slave-side NI NoC and return to the master-side NoC NI before their “opponents”, they are not blocked, for the reason that there is no ordering requirement between them and their “opponents” both in the master-side and slave-side ordering units.

The transaction_0 is sent out by the master-side NoC NI before the transaction_3. But it arrives at the slave-side NoC NI later than the transaction_3 due to its longer data length in the transaction. We can observe that the transaction_3 is blocked by the transaction_0 in the slave side unit the arrival of the transaction_0 and thus, the transaction_0 can firstly access the memory.

Finally, for the transaction_2, which targets the different slave from the transaction_0, it is not blocked by the transaction_0 in the slave side, although the transaction_2 is sent out by the master-side NoC NI later than the transaction_0. But in the master side, it has been blocked by the ordering unit until the arrival of the transaction_0 and thus, the transaction_0 can firstly reach the master.

In this way, we verify different scenarios in the AXI4 transaction ordering requirements. The results show that our mechanism can perfectly support the AXI4 transaction ordering requirements as well as our proposed seq_ID reuse mechanism.

2) **PERFORMANCE TEST**

When applying the state-of-the-art AXI4 transaction ordering solutions, some transactions will be blocked due to the restriction in the state-of-the-art solutions, as we explained in Section III-B. In reality, the processor does not know which slave the requested memory will be mapped to and thus, could generate a series of transactions with the same AXI4_ID and r/w_ID but targeting different slaves. In this context, the following transactions will be blocked by the first transaction in the master-side ordering unit.

By allowing seq_ID reuse mechanism, the problems can be solved and the system throughput can be enhanced and the transactions’ queueing delay in the master-side ordering unit can be greatly reduced.
To compare the performance results with the state-of-the-art solutions [4]–[6], we test the system throughput and the transaction’s queuing delay in the master-side ordering unit under three different scenarios where different percentages of transactions will be blocked due to the drawback of the state-of-the-art solutions but will not be blocked in our proposed mechanism with the seq_ID reuse mechanism. In reality, the percentage will possibly increase when the injection rate of the master node increases. So we utilize different scenarios with various percentages of blocked transactions for a comprehensive comparison. In the scenario one, averagely for each four transactions, one of them will be blocked by its previous transaction without the seq_ID reuse mechanism. So, in the first scenario, 25% of transactions will be blocked in the state-of-the-art solution on average. In the second scenario, averagely for each four transactions, two of them will be blocked by their previous transaction without the seq_ID reuse mechanism. The average percentage increases to 50% in this scenario. In the third scenario, averagely for each four transactions, three of them will be blocked by their previous transaction without the seq_ID reuse mechanism, that is the average percentage of 75%.

We show the throughput comparison between the state-of-the-art and the seq_ID reuse mechanism separately under 25%, 50%, and 75% conflict for four different master nodes (M0, M9, M18 and M27) in Figure 14, Figure 15, and Figure 16. The percentage of conflict represents proportions for blocked transactions in the state-of-the-art solution without the seq_ID reuse mechanism. The average is 0.2 transaction/cycle per node, indicating average 1000 transactions generated by a node in a simulation time interval.

We can see that the improvement of the throughput by the seq_ID reuse mechanism gets higher when the percentage of transaction conflict gets larger. Especially, after the NoC gets saturated, the throughput in the state-of-the-art solution will greatly decrease due to the serious blocking of transactions. In our mechanism, although the throughput will be influenced by the saturation, the value can still be at a relatively high level.

For a better comparison and clearly showing the performance improvement tendency, we plot the improvement of the system throughput by the seq_ID reuse mechanism in Figure 17. The throughput improvement under different scenarios is calculated by Equation 1 where the $\Theta_{improvement}$ represents the throughput improvement, the $\Theta_{seq\_ID-reuse}$ represents the throughput generated by our proposed seq_ID reuse mechanism and the $\Theta_{state-of-the-art}$ represents the throughput generated by the mechanism in the state-of-the-art works.

$$\Theta_{improvement} = \Theta_{seq\_ID-reuse} - \Theta_{state-of-the-art} \quad (1)$$

We can see that the seq_ID reuse mechanism can significantly contribute to more system throughput improvement when the proportions are getting higher for each master node. Under the same proportion, the throughput improvement gets higher if the master node gets further away from the network center. Our mechanism can maximally increase the system
throughput by 0.528 flit/cycle when the system injection rate is 0.8 flit/cycle, which is 66.0% of the system injection rate. The throughput improvements decrease after the system injection rate is higher than 0.8 flit/cycle. This is because the network is already saturated at that time, and the system overall throughput will decrease after saturation, and so as the throughput improvement results.

In terms of latency, we show the comparison of transaction’s queueing delay in the NI between the state-of-the-art and the seq_ID reuse mechanism separately under 25%, 50%, and 75% conflict for four different master nodes (M0, M9, M18 and M27) in Figure 18, Figure 19, and Figure 20. They clearly show that, under each condition with different conflict percentages, our proposed seq_ID reuse mechanism can successfully decrease the transaction’s queueing delay to a great extent.

For a better comparison and clearly showing the performance improvement tendency, we show the decrease of queueing delay of the transaction in the master-side ordering unit by the seq_ID reuse mechanism in Figure 21. The latency decrease under different scenarios is calculated by the Equation 2 where the $L_{decrease}$ represents the latency decrease, the $L_{seq\_ID\_reuse}$ represents the latency generated by our proposed seq_ID reuse mechanism and the $L_{state\_of\_the\_art}$ represents the latency generated by the mechanism in the state-of-the-art works.

\[
L_{decrease} = L_{state\_of\_the\_art} - L_{seq\_ID\_reuse} \tag{2}
\]

We can conclude that our mechanism can significantly decrease the queueing delay of transactions before the master-side ordering unit with the proportion of AXI4 transaction ID conflict getting higher. In this case, the position of the master node and the injection rate will also slightly influence the results. Our mechanism can maximally decrease the queueing delay of the transaction in the master-side ordering unit as well as its round-trip latency by 145.2 cycles, which takes 91.2% of the queueing delay (not shown in the figure) with the state-of-the-art solutions.

Besides, the packet and flit sizes will also have strong impact on the performance of the NoC system. In this section, we have shown the throughput and injection rate in the unit of flit/cycle. Under a given packet injection rate (packet/cycle), the increase of the packet size (with fixed flit size) or the decrease of the flit size (with fixed packet size) will both increase the injection rate in the unit of flit/cycle. When changing the packet and flits sizes, it will have the same influence on the performance as purely changing the injection rate as we have shown in all these figures in this section.

3) HARDWARE SYNTHESIS

We show the hardware block diagram and report synthesis results of the master-side and slave-side ordering units. It is worth noting that no previous works have reported hardware-related details and synthesis results.

In Figure 22, we show the block diagram of master-side and slave-side ordering units. We utilize single-port RAM to realize the Sequence Number Reference Table (SNRT), Head...
Address Register (HAR), Sequence Status Holding Register (SSHR) and Re-order Buffer (RoB), for the single-port RAM can satisfy the performance requirement and at the same time, greatly reduce power and area consumption. The sizes of each register buffer and data width are shown in Table 2. The Sequence Number Reference Table has a private finite-state machine (FSM) to deal with the AXI4 signal from the AXI4 master/slave port. The Head Address Register, Sequence Status Holding Register and Re-order Buffer are controlled by a global FSM to restrict their access sequences, which will also interact with the SNRT FSM and deal with the signal/packet to/from the interconnect NI for packing or depacking target. For the reason that the Head Address Register, Sequence Status Holding Register and Re-order Buffer will be read and written frequently, we utilize three status vectors to decrease the access rate to the three register buffers and thus, their access efficiency can be improved. Each number in a vector is corresponding to a record item in a register buffer, which follows the status (valid/invalid) of the record. Since the operation of the Sequence Number Reference Table is relatively simple, we do not design the status vector for it, as it will not influence the performance. In reality, considering the NI for co-design, the message buffer in the NI can be modified and utilized as the Re-order Buffer, which can decrease the area and power consumption of the ordering unit.

| Table 2. Parameters for each functional part in the ordering unit. |
|---------------------------------------------------------------|
| Master-side SNRT HAR SXHR RoB |
| Width (bits) 16 5 20 133 |
| Depth 16 32 16 12 |
| Slave-side SNRT HAR SXHR RoB |
| Width (bits) 16 20 14 133 |
| Depth 16 32 16 12 |

The RTL hardware synthesis results using a 40 nm library for the master-side and slave-side ordering units are reported in Table 3. The synthesized frequency can reach up to 0.6 GHz under the voltage of 1.1 V. The ordering unit in the master side consumes 9.0 K equivalent NAND gates, which is less than its counterpart in the slave side that consumes 13.4 K equivalent NAND gates. This is because the width of the Head Address Register unit in the master side is much shorter than the one in the slave side. For comparison, we also show the size of the router with the same configurations as the parameters shown in Table 1, which consumes 22.0 K equivalent NAND gates. In [5], the synthesized areas of master-side and slave-side NIs are separately 75559 $\mu$m² and 42848 $\mu$m² under UMC 90 nm technology. The dynamic power consumptions of the three units are close to each other, which are separately 1.89 mW (master), 1.95 mW (slave), and 1.80 mW (router).

| Table 3. Hardware synthesis results. |
|--------------------------------------|
| Master | Slave | Router |
| Frequency | 0.6 GHz | 0.6 GHz | 2 GHz |
| Gate count | 9.0 K | 13.4 K | 22.0 K |
| Dynamic power | 1.89 mW | 1.95 mW | 1.80 mW |

VI. CONCLUSION

In this paper, we proposed a many-core system architecture design to satisfy the AXI4 protocol, and especially focused on the design of the AXI4 transaction ordering units. By involving the seq_ID reuse mechanism, we can greatly improve the system throughput and decrease the transaction’s queueing delay in the master-side ordering unit. For ease of implementation, we described the micro-architectures and functionalities of the master-side and slave-side ordering units in detail. Besides, our proposed seq_ID synchronization method can simplify the design of ordering units as well as the information carried by the transaction, which can decrease the hardware consumption. In the experimental part, we showed the correctness verification and performance test results by the software simulator, and the hardware synthesis results for the master-side and slave-side ordering units.

REFERENCES

[1] AXI and ACE Protocol Specification, ARM AMBA, 2011.
[2] T. Kazaz, C. Van Praet, M. Kulin, P. Willemsen, and I. Moerman, “Hardware accelerated SDR platform for adaptive air interfaces,” 2017, arXiv:1705.00115 [Online]. Available: http://arxiv.org/abs/1705.00115
[3] C. Liss, M. Ulbricht, U. F. Zia, and H. Müller, “Architecture of a synchronized low-latency network node targeted to research and education,” in Proc. IEEE 18th Int. Conf. High Perform. Switching Routing (HPSR), Jun. 2017, pp. 1–7.
[4] X. Yang, Z. Qing-Li, F. Fang-Fa, Y. Ming-Yan, and L. Cheng, “NISAR: An AXI compliant on-chip NI architecture offering transaction reordering processing,” in Proc. 7th Int. Conf. ASIC, Oct.2007, pp. 890–893.
[5] M. Ebrahimi, M. Daneshchalab, P. Liljeberg, J. Plosila, and H. Tenhunen, “A high-performance network interface architecture for NoCs using reorder buffer sharing,” in Proc. 18th Euromicro Conf. Parallel, Distrib. Netw.-Based Process., Feb. 2010, pp. 546–550.
[6] M. H. Neishaburi and Z. Zilic, “Debug aware AXI-based network interface,” in Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst., Oct. 2011, pp. 399–407.
[7] F.-M. Xiao, D.-S. Li, G.-M. Du, Y.-K. Song, D.-L. Zhang, and M.-L. Gao, “Design of AXI bus based MPSoc on FPGA,” in Proc. 3rd Int. Conf. Anti-Counterfeiting, Secur., Identificat. Commum., Aug. 2009, pp. 560–564.
[8] R. Bhaktavatchalu, B. S. Rekha, G. A. Divya, and V. U. S. Jyothi, “Design of AXI bus interface modules on FPGA,” in Proc. Int. Conf. Adv. Commun. Control Comput. Technol. (ICACCCT), May 2016, pp. 141–146.
[9] M. Ramirez, M. Daneshchalab, J. Plosila, and P. Liljeberg, “NoC-AXI interface for FPGA-based MPSoC platforms,” in Proc. 22nd Int. Conf. Field Program. Log. Appl. (FPL), Aug. 2012, pp. 479–480.
[10] N. Tidala, “High performance network on chip using AXI4 protocol interface on an FPGA,” in Proc. 2nd Int. Conf. Electron., Commun. Aerosp. Technol. (ICECA), Mar. 2018, pp. 1647–1651.
[11] T. Nguyen, D.-H. Bui, H.-P. Phan, T.-T. Dang, and X.-T. Tran, “High-performance adaption of ARM processors into network-on-chip architectures,” in Proc. IEEE Int. SOC Conf., Sep. 2013, pp. 222–227.
[12] A. Radulescu, J. Dielissen, S. G. Pestana, O. P. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens, “An efficient on-chip NI offering guaranteed services, shared-memory abstraction, and flexible network configuration,” IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., vol. 24, no. 1, pp. 4–17, Jan. 2005.
[13] X. Liao, J. Zhou, and X. Liu, “Exploring AMBA AXI on-chip interconnection for TSV-based 3D SoCs,” in Proc. IEEE IEEE Int. 3D Syst. Integr. Conf. (3DIC) Int., Jan. 2012, pp. 1–4.
[14] E. Kim, S. Rhim, and B. Lee, “NoC system employing AXI protocol and interleaving method thereof,” U.S. Patent 11 513 021, May 24, 2007.
[15] W.-C. Kwon, S. Yoo, J. Um, and S.-W. Jeong, “In-network reorder buffer to improve overall NoC performance while resolving the in-order requirement problem,” in Proc. Design, Autom. Test Eur. Conf. Exhibit., Apr. 2009, pp. 1058–1063.
BOQIAN WANG received the B.S. and M.S. degrees in computer science from the National University of Defense Technology, Changsha, Hunan, China, in 2013 and 2016, respectively. He is currently pursuing the Ph.D. degree with the School of Electrical Engineering and Computer Science, KTH Royal Institute of Technology (KTH), Stockholm, Sweden. His research interests include interconnection network and computer architecture.

ZHONGHAI LU (Senior Member, IEEE) received the B.S. degree in radio and electronics from Beijing Normal University, Beijing, China, in 1989, and the M.S. degree in system-on-chip design and the Ph.D. degree in electronic and computer system design from the KTH Royal Institute of Technology (KTH), Stockholm, Sweden, in 2002 and 2007, respectively. From 1989 to 2000, he was an Engineer in the areas of electronic and embedded systems. He is currently a Professor with the School of Electrical Engineering and Computer Science, KTH. He has authored over 180 peer-reviewed papers. His research interests include interconnection networks, computer architecture, embedded system, and the Internet of Things.

* * *

[16] W.-C. Kwon, S. Yoo, S.-M. Hong, B. Min, K.-M. Choi, and S.-K. Eo, “A practical approach of memory access parallelization to exploit multiple off-chip DDR memories,” in Proc. 45th Annu. Conf. Design Autom. (DAC), 2008, pp. 447–452.

[17] S. Murali, D. Atienza, L. Benini, and G. De Micheli, “A method for routing packets across multiple paths in NoCs with in-order delivery and fault-tolerance guarantees,” VLSI Des., vol. 2007, pp. 1–11, Apr. 2007.

[18] M. Daneshtalab, M. Ebrahimi, P. Liljeberg, J. Plosila, and H. Tenhunen, “Memory-efficient on-chip network with adaptive interfaces,” IEEE Trans. Comput.-Aided Design Integr. Circuits Syst., vol. 31, no. 1, pp. 146–159, Jan. 2012.

[19] G. Du, M. Li, Z. Lu, M. Gao, and C. Wang, “An analytical model for worst-case reorder buffer size of multi-path minimal routing NoCs,” in Proc. 8th IEEE/ACM Int. Symp. Netw.-Chip (NoCS), Sep. 2014, pp. 49–56.

[20] W. J. Dally and B. P. Towles, Principles and Practices of Interconnection Networks. Amsterdam, The Netherlands: Elsevier, 2003, pp. 477–478.

[21] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sen, K. Sewell, M. Shoail, N. Vaish, M. D. Hill, and D. A. Wood, “The gem5 simulator,” ACM SIGARCH Comput. Archit. News, vol. 39, no. 2, pp. 1–7, May 2011.

[22] N. Jiang, D. Becker, G. Michieliannakis, J. Balfour, B. Towles, J. Kim, and W. Dally, “A detailed and flexible cycle-accurate network-on-chip simulator,” in Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS), Apr. 2013, pp. 86–96.

[23] J. Yin, Z. Lin, O. Kayiran, M. Poremba, M. S. B. Altaf, N. E. Jerger, and G. H. Loh, “Modular routing design for chiplet-based systems,” in Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA), Jun. 2018, pp. 726–738.

[24] Y. Yao and Z. Lu, “iNPG: Accelerating critical section access with in-network packet generation for NoC based many-cores,” in Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA), Feb. 2018, pp. 15–26.

[25] J. Jose and A. Das, “An adaptive deflection router with dual injection and ejection units for mesh NoCs,” in Proc. 31st Int. Conf. VLSI Design 17th Int. Conf. Embedded Syst. (VLSID), Jan. 2018, pp. 374–379.