A Low-Latency and Memory-Efficient On-chip Network

Masoud Daneshtalab, Masoumeh Ebrahimi, Pasi Liljeberg, Juha Plosila, Hannu Tenhunen
Department of Information Technology, University of Turku, Finland
{ masdan, masebr, pakrli, juplos, hanten }@utu.fi

Abstract- ***Using multiple SDRAMs in MPSoCs and NoCs to increase memory parallelism is very common nowadays. In-order delivery, resource utilization, and latency are the most critical issues in such architectures. In this paper, we present a novel network interface architecture to cope with these issues efficiently. The proposed network interface exploits a resourceful reordering mechanism to handle the in-order delivery and to increase the resource utilization. A brilliant memory controller is efficiently integrated into this network interface to improve the memory utilization and reduce both memory and network latencies. In addition, to bring compatibility with existing IP cores the proposed network interface utilizes AXI transaction based protocol. Experimental results with synthetic test cases demonstrate that the proposed architecture gives significant improvements in average network latency (12%), average memory access latency (19%), and average memory utilization (22%).***

1. INTRODUCTION

Integrating a large number of functional and storage modules onto a single die in the deep sub-micron regime and beyond is becoming a major performance issue in System-on-Chip (SoC) architectures [1][2][3][4]. Network-on-Chip (NoC) has emerged as a solution to address the communication demands of many/multi core architectures due to its reusability, scalability, and parallelism in communication infrastructure [3][4]. The fundamental function of network interfaces (NI) is to provide communication between Processing Elements (PE) and the network infrastructure [5][6][7]. That is, one of the practical approaches of network interfaces is to translate the language between the PE and router based on a standard communication protocol such as AXI [6] and OCP [7]. On top of that, in-order delivery is another practical approach of network interfaces [8][9][10]. In-order delivery should be handled when exploiting an adaptive routing algorithm for distributing packets through the network [8], when obtaining memory access parallelization by sending requests from a master IP core to multiple slave memories [9][10], or when exploiting a modern memory access scheduling in memory controller to reorder memory requests [11]. In this paper, we introduce an efficient network interface architecture where the key ideas are threefold. The first idea is to deal with out-of-order handling in such a way that when a master sends requests to different memories, the responses might be required to return in the same order in which the master issued the addresses, and therefore a reordering mechanism in NoC should be provided by the network interface. The second idea is to improve resource utilization because in on-chip networks we have limitation of hardware resources (e.g. buffers), power consumption, and network latency. According to our observation, utilization of reorder buffers in network interfaces is significantly inefficient, due to the fact that the traditional buffer management is not efficient enough for network interfaces. Therefore, an advantageous reordering mechanism via resourceful management of buffers in the network interface is presented. The last idea is to present a brilliant memory controller that is efficiently integrated into the proposed network interface, which helps to improve memory utilization and reduce both memory and network latencies. Also, the proposed network interface architecture exploits the AMBA AXI protocol, to allow backward compatibility with existing IP cores [6]. We also present micro-architectures of the proposed ideas, particularly the memory access reordering mechanism. The paper is organized as follows. In Section 2, the preliminaries are discussed. In Section 3, a brief review of related works is presented while the proposed network interface architectures and the brilliant memory controller are presented in Section 4 and 5, respectively. The experimental results are discussed in Section 6 with the summary and conclusion given in the last section.

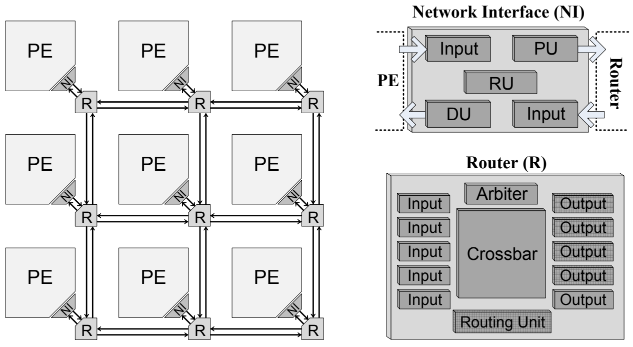

Fig. 1. Tile-based 2D-Mesh topology.

2. PRELIMINARIES

2.1. NETWORK-ON-CHIP ARCHITECTURE

A 2D-mesh NoC based system is shown in Fig. 1. NoC consists of Routers (R), Processing Elements (PE), and Network Interfaces (NI). PEs may be intellectual property (IP) blocks or embedded memories. Each core is connected to the corresponding router port using the network interface. To be compatible with existing transaction-based IP-cores, we use the AMBA AXI protocol. AMBA AXI is an interfacing protocol, having advanced functions such as a multiple outstanding address function and data interleaving function [6]. AXI, providing such advanced functions, can be implemented on NoCs as an interface protocol between each PE and router to avoid the structural limitations in SoCs due to the bus architecture. The protocol can achieve very high speed of data transmission between PEs [6]. In the AXI transaction-based model [6][9], IP cores can be classified as master (active) and slave (passive) IP cores [10][18]. Master IP cores initiate transactions by issuing read and write requests and one or more slaves (memories) receive and execute each request. Subsequently, a response issued by a slave can be either an acknowledgment (corresponding to the write request) or data (corresponding to the read request) [10]. The AXI protocol provides a "transaction ID" field assigned to each transaction. Transactions from the same master IP core, but with different IDs have no ordering restriction while transactions with the same ID must be completed in order. Thus, a reordering mechanism in the network interface is needed to afford this ordering requirement [6][7][14]. The network interface lies between a PE and the corresponding attached router. This unit forms the foundation of the generic nature of the architecture as it prevents the PEs from directly interacting with the rest of the network components in the NoC. A generic network interface architecture is shown in Fig. 1.

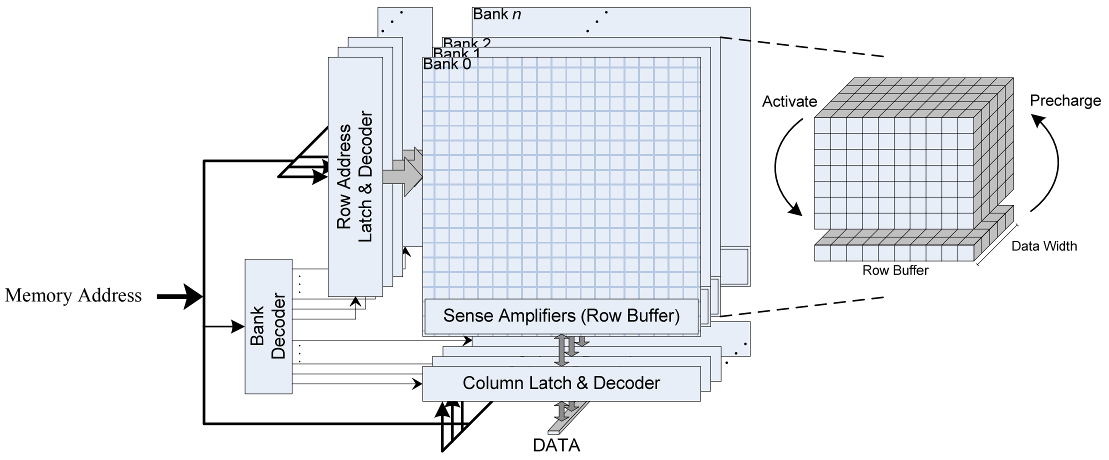

Fig. 2. High-level structure of an SDRAM.

The network interface consists of input buffers (forward and reverse directions), a Packetizer Unit (PU), a Depacketizer Unit (DU), and a Reorder Unit (RU). A data burst coming from a PE is latched into the input buffer of the corresponding network interface. PU is configured to packetize the burst data stored in the input buffer and transfer the packet to the router. Similarly, data packets coming from the router are latched into the input buffer located in the reverse path. DU is configured to restore original data format, required for the PE, from the packet provided by the router. The RU performs a packet reordering to meet the in-order requirement of each PE.

2.2. SDRAM STRUCTURE

SDRAM is designed to provide high memory depth and bandwidth. Fig. 2 shows a simplified three dimensional architecture of an SDRAM memory chip with the dimensions of bank, row, and column [11][19][20]. An SDRAM chip is composed of multiple independent memory banks such that memory requests to different banks can be serviced in parallel. That is, a benefit of a multibank architecture is that commands to different banks can be pipelined. Each bank is formed as a two dimensional array of DRAM cells that are accessed an entire row at a time. Thus, a location in the DRAM is identified by an address consisting of bank, row, and column fields. A complete SDRAM access may require three commands (transactions) in addition to the data transfer: bank precharge, row activation, and column access (read/write). A bank precharge charges and prepares the bank, while a row-activation command (with the bank and row address) is used to copy all data in the selected row into the row buffer, i.e. sense amplifier. The row buffer serves as a cache to reduce the latency of subsequent accesses to that row. Once a row is in the row buffer, then column commands (read/write) can be issued to read/write data from/into the memory addresses (columns) contained in the row. To prepare the bank for a next row activation after completing the column accesses, the cached row must be written back to the bank memory array by the precharge command [11]. Also, the timing constraints associated with bank precharge, row activation, and column access are t_{RP}, t_{RCD}, and t_{CL} respectively [11][19][21]. Since the latency of a memory request depends on whether the requested row is in the row buffer of the bank or not, a memory request could be a *row hit*, *row conflict* or *row empty* with different latencies [22]. A *row hit* occurs when a request is accessing the row currently in the row buffer and only a read or a write command is needed. It has the lowest bank access latency (t_{CL}) as only a column access is required. A *row conflict* occurs when the access is to a row different from the one currently in the row buffer. The contents of the row buffer first need to be written back into the memory array using the precharge command. Afterward, the required row should to be opened and accessed using the activation and read/write commands. The *row conflict* has the highest bank access latency (t_{RP} +t_{RCD} +t_{CL}). If the bank is closed (precharged) or there is no row in the row buffer then a *row empty* occurs. An activation command should be issued to open the row followed by read or write command(s). The bank access latency of this case is t_{RCD} +t_{CL}.

2.2.1. MEMORY ACCESS SCHEDULING

The memory controller lies between processors and the SDRAM to generate the required commands for each request and schedules them on the SDRAM buses. The memory controller consists of a request table, request buffers, and a memory access scheduler. A request table is used to store the state of each memory request, e.g. valid, address, read/write, header pointer to the data buffer and any additional state necessary for memory scheduling. The data of outstanding requests are stored in read and write buffers. The read and write buffers (request buffers) are implemented as linked list [12][13]. Each memory request (read and write) allocates an entry in its respective buffer until the request is completely serviced. Among all pending memory requests, based on the state of the DRAM banks and the timing constraints of the DRAM, the memory scheduler decides which DRAM command should be issued. The average memory access latency and memory bandwidth utilization can be reduced and improved, respectively if an efficient memory scheduler is employed [11][19][20]. Fig. 3 reveals how the memory access scheduling affects the performance. As shown in the figure, the sequence of four memory requests is considered. Request 1 and 3 are row empties, and request 2 and 4 are row conflicts. Timing constraints of a DDR2-512MB used as example throughout this paper is 2-2-2 (t_{RP}-t_{RCD}-t_{CL}) [21]. As depicted in Fig. 3(a), if the controller schedules the memory requests in order, it will take 22 memory cycles to complete them. In Fig. 3(b) the same four requests are scheduled out of order. As can be seen, request 4 is scheduled before request 2 and 3 to turn request 4 from a row conflict to a row hit. In addition, request 3 is pipelined after request 1, called *bank interleaving*, since it has the different bank address from the bank address of request 1. As a result, only 14 memory cycles are needed to complete the four requests. In sum, how the memory scheduler can improve the memory performance has been shown by this example where the memory utilization of the in order scheduler and the out of order are 4(data)/22(cycle) = 18% and 4/14= 29%, respectively. In this work, we presented a brilliant memory controller that is efficiently integrated into the proposed network interface to improve the memory utilization and reduce both memory and network latencies. The idea and the implementation details of the proposed architecture are described in Section 5.

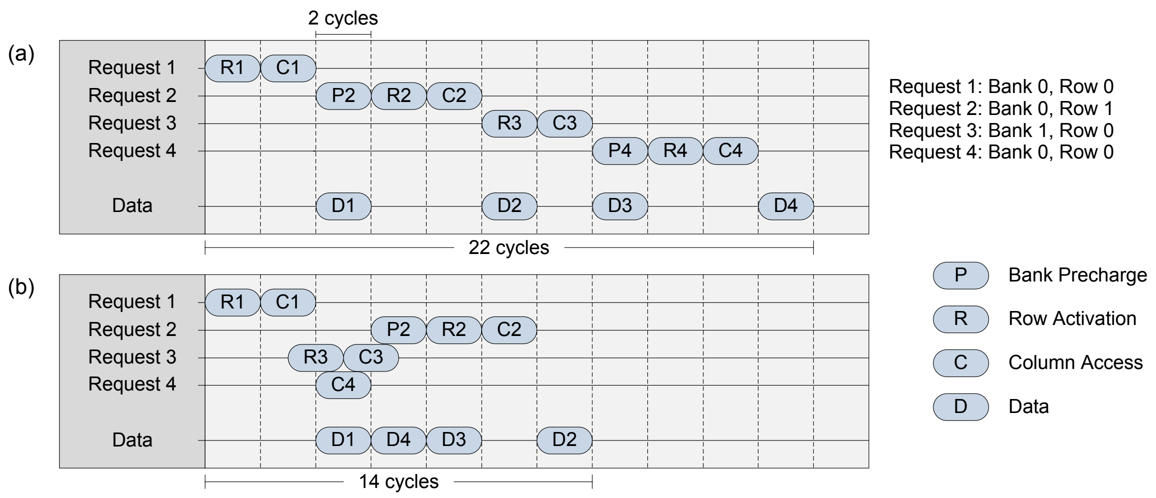

Fig. 3. Memory access scheduling of four memory requests with (a) in order and (b) with out of order access scheduling.

3. Related Work

Due to the fact that most of the recent published researches have focused on the design and description of NoC architectures, there has been relatively little attention to network interface designs particularly when supporting out-of-order mechanism. The authors in [9] present ideas of transaction ID renaming and distributed soft arbitration in the context of distributed shared memories. In such a system, because of using a global synchronization in the on-chip network, the performance might be degraded and the cost of hardware overhead for the on-chip network is too high. In addition, the implementation of ID renaming and reorder buffer can suffer from low resource utilization. This idea has been improved in [14] by moving reorder buffer resources from the network interface into network routers. In spite of increasing the resource utilization, the delay of release packets recalling data from distributed reordering buffer can significantly degrade the performance when the size of the network increases [14]. Moreover, the proposed architecture is restricted to deterministic routing algorithms and thus, it is not a suitable method for an adaptive routing. However, neither [9] nor [14] has presented a micro-architecture of the network interface. An efficient on-chip network interface supporting shared memory abstraction and flexible network configuration is presented by Radulescu et al [10]. The proposed architecture has the advantage of improving reuse of IP cores, and offers ordering messages via channel implementation. Nevertheless, the performance is penalized because of increasing latency, and besides, the packets are routed on the same path in NoC, which forces the routers to use the deterministic routing. Yang et al proposed NISAR [8], a network interface architecture using the AXI protocol capable of packet reordering based on a look up table; but NISAR uses a statically partitioned reorder buffer, thereby it has a simple control logic but suffers from low buffer utilization in different traffic patterns. In addition, NISAR does not support burst transactions, whereas burst type should be handled by the network interface.

Regarding the memory scheduler, several memory scheduling mechanisms were presented to improve the memory utilization and to reduce the memory latency. The key idea of these mechanisms is on the scheduler for reordering memory accesses. The memory access scheduler proposed in [11] reorders memory accesses to achieve high bandwidth and low average latency. In this scheme, called bank-first scheduling, memory accesses to different banks are issued before those to the same bank. Shao et al. [23] proposed the burst scheduling mechanism based on the row-first scheduling scheme. In this scheme, memory requests that might access the same row within a bank are formed as a group to be issued sequentially, i.e. as a burst. Increasing the row hit rate and maximizing the memory data bus utilization are the major design goals of burst scheduling. The core-aware memory scheduler revealed that it is reasonable to schedule the requests by taking into consideration the source of the requests because the requests from the same source exhibit better locality [20]. In [19] the authors introduced an SDRAM-aware router to send one of the competing packets toward an SDRAM using a priority-based arbitration. An adaptive history-based memory scheduler which tracks the access patterns of recently scheduled accesses and selects memory accesses matching the pattern of requests is proposed in [24] and [25]. As network-on-chips are strongly emerging as a communication platform for chip-multiprocessors, the major limitation of presented memory scheduling mechanisms is that none of them did take the order of the memory requests into consideration. As discussed earlier, requests with the same transaction ID from the same master must be completed (turn back) in order. While the requests would be issued out-of-order in memories (slave-sides), the average network latency might be increased significantly due to the out-of-order mechanism in master sides. Therefore, it is necessary to consider the order of memory requests for making an optimal memory scheduling.

The major contribution of this paper is to propose a novel network interface architecture within a dynamic buffer allocation mechanism for the reorder buffer to increase the utilization and overall performance. That is, using the dynamic buffer allocation to get more free slots in the reorder buffer may lead more messages to be entered to the network. On top of that, an efficient memory scheduler mechanism based on the order of requests is introduced and integrated in our network interface to diminish both the memory and network latencies.

4. Proposed Network Interface Architecture

Since IP cores are classified into masters and slaves, the network interface is also divided into the master network interface (Fig. 4) and slave network interface (Fig. 5). Both network interfaces are partitioned into two paths: forward and reverse. The forward path transmits the AXI transactions received from an IP core to a router; and the reverse path receives the packets from the router and converts them back to AXI transactions. The proposed network interfaces for both master and slave sides are described in detail as follows.

4.1. Master-side Network Interface

As shown in Fig. 4, the forward path of the master network interface transferring requests to the network is composed of an AXI-

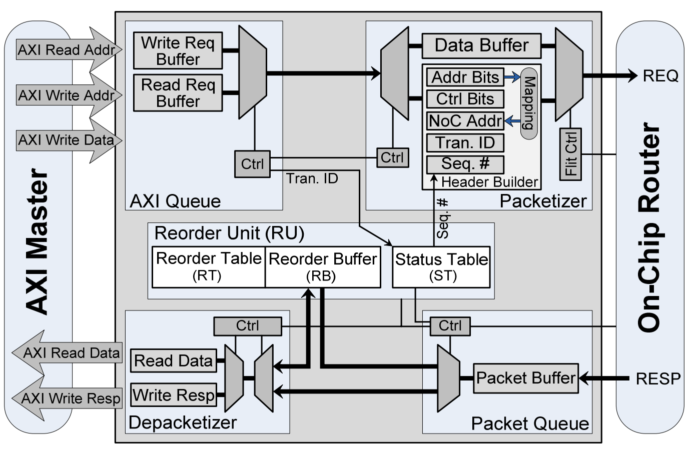

Fig. 4. Master-side network interface architecture.

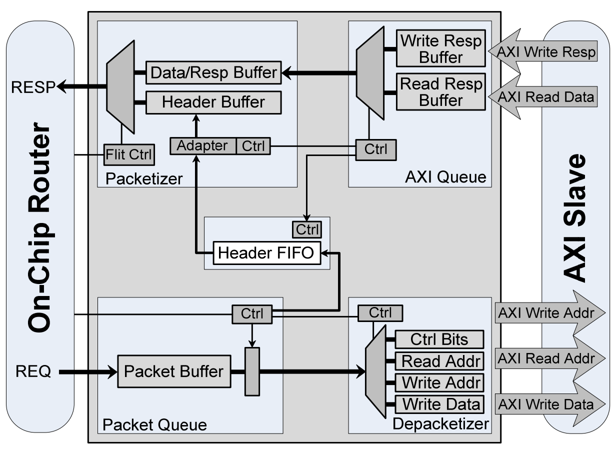

Fig. 5. Slave-side network interface architecture.

Queue, a Packetizer unit, and a Reorder unit, while the reverse path, receiving the responses from the network, is composed by a Packet-Queue, a Depacketizer unit, and the Reorder unit. The Reorder unit is a shared module between the forward and reverse paths.

AXI-Queue: the AXI master transmits write address, write data, or read address to the network interface through channels. The AXI-Queue unit performs the arbitration between write and read transaction channels and stores requests in either write or read request buffer. The request messages will be sent to the packetizer unit if admitted by the reorder unit, and on top of that a sequence number for each request should be prepared by the reorder unit after the admittance.

Packetizer: it is configured to convert incoming messages from the AXI-Queue unit into header and data flits, and delivers the produced flits to the router. Since a message is composed of several parts, the data is stored in the data buffer and the rest of the message is loaded in corresponding registers of the header builder unit. After the mapping unit converts the AXI address into a network address by using an address decoder, based on the request information loaded on relative registers and the sequence number provided by the reorder buffer, the header of the packet can be assembled. Afterward, the flit controller wraps up the packet for convenient transmission.

Packet-Queue: this unit receives packets from the router; and according to the decision of the reorder unit a packet is delivered to the depacketizer unit or reorder buffer. In fact, when a new packet arrives, the sequence number and transaction ID of the packet will be sent to the reorder unit. Based on the decision of the reorder unit, if the packet is out of order, it is transmitted to the reorder buffer, and otherwise it will be delivered to the depacketizer unit directly.

Depacketizer: the main functionality of the Depacketizer unit is to restore packets coming from either the packet queue unit or reorder buffer into the original data format of the AXI master core.

Reorder Unit: it is the most influential part of the network interface including a Status-Register, a Status-Table, a Reorder Buffer, and a Reorder-Table. In the forward path, preparing the sequence number for corresponding transaction ID, and avoiding overflow of the reorder buffer by the admittance mechanism are provided by this unit. On the other side, in the reverse path, this unit determines where the outstanding packets from the packet queue should be transmitted (reorder buffer or depacketizer), and when the packets in the reorder buffer could be released to the depacketizer unit.

Status-Register and Status-Table: Status-Register (S_Reg) is an n-bit register where each bit corresponds to one of the AXI transaction IDs. As depicted in Fig. 6, this register records whether there are one or more messages with the same transaction ID being issued or not. To record the state of the outstanding messages, Status-Table (S_Table) is adopted. Each entry of this table is considered for messages with the same transaction ID, and includes valid tag (v), Transaction ID (T-ID), Number of outstanding Message (N-M) as well as the Expecting Sequence number (E-S). The register and table might be updated in both forward and reverse paths described as follows. In the forward path, when the first message of each transaction ID requests for an admittance from the reorder unit to enter the network, the corresponding bit in the status register goes high (Procedure A: line 1, Fig. 6(a)). The sequence number (Seq-Num) is produced by the reorder unit, if the admittance is given. This value, indicating the order of the messages within the transaction ID, is equal to zero for the first message of each transaction ID (Procedure A: line 2). “ReservedSize” keeps the required space of all outstanding transactions in the network. Indeed, this register reserves the number of buffer slots required by outstanding messages of different transaction IDs. In order to prevent overflow of the reorder buffer, the reorder unit compares the new message size with the free space of the reorder buffer. If the required space is available, the message will be admitted and the required space in the reorder buffer must be reserved (Procedure A, line 3). An available (free) row in the status table will be initiated by procedure B, when the second request of a transaction ID is admitted. For the rest of the admitted requests of the transaction ID, the procedure C should be executed as the sequence number is obtained by adding N-M and E-S values. Also, the number of outstanding message (N-M) is increased by +1, and the required space in the reorder buffer must be reserved by procedure C. Note that E-S indicates the next response sequence number of the corresponding transaction ID that should be delivered to the depacketizer unit.

```
Procedure A:
1  S_Reg(T_ID)          <= ‘1’;
2  SeqNum               <= (others =>’0’);
3  ReservedSize         <= ReservedSize + NewMsgSize;

Procedure B:
1  S_Table(FreeRow)(v)    <= ’1’;
2  S_Table(FreeRow)(T_ID) <= Tran_ID;
3  S_Table(FreeRow)(N_M)  <= “0010”;
4  S_Table(FreeRow)(E_S)  <= (others =>’0’);
5  SeqNum                 <= “001”;
6  ReservedSize           <= ReservedSize +
                             NewMsgSize;

Procedure C:
1  SeqNum                 <= S_Table(FindRow)(N_M) +
                             S_Table(FindRow)(E_S);
2  S_Table(FindRow)(N_M)<= S_Table(FindRow)(N_M) + 1;
3  ReservedSize           <= ReservedSize + NewMsgSize;
```

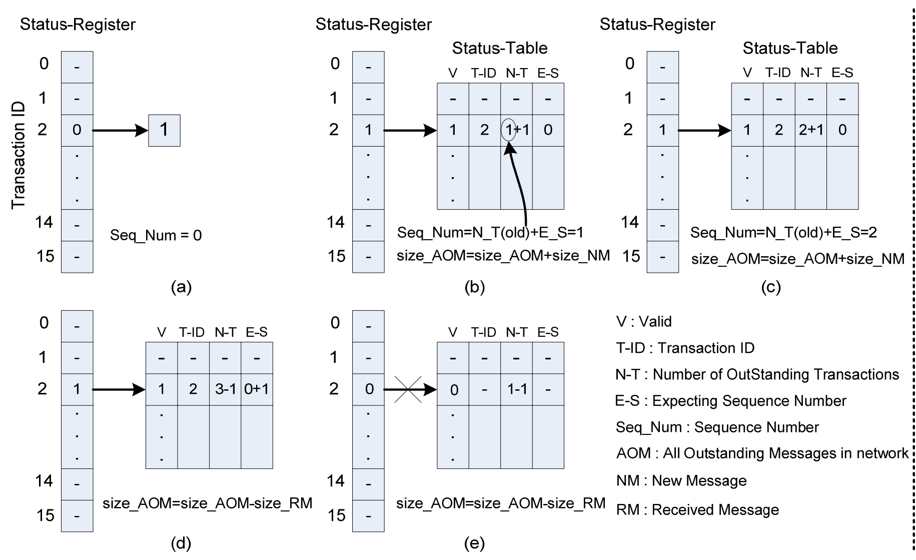

Fig. 6. Status-Register and StatusTable of Reorder Unit.

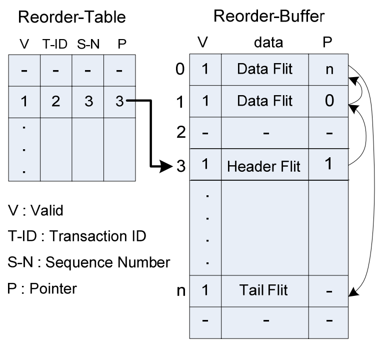

Fig. 7. Dynamic buffer allocation

In the reverse path, the transaction ID and sequence number of the arriving response packet (message) are sent to the reorder unit to find the related row in the status table according to the transaction ID (T-ID). Ifthe sequence number of incoming packet is equal to E-S value, the packet is an expected packet (in-order) and should be delivered to the depacketizer unit which releases the occupied buffer space; thereafter, E-S and N-T values will be increased by +1 and -1, respectively (Procedure D). If N-M value reaches zero, the transaction will be terminated by resetting the valid bit for both status register and status table. However, the packet is out-of-order and should be delivered to the reorder buffer, if the sequence number of the packet is not equal to E-S. Additionally, only one message with the given transaction ID should have been sent to the network, if the given transaction ID is not matched in the status table, thereby only the corresponding bit in the status register will be reset.

```
Procedure D:
1 S_Table(FindRow)(N_M) <= S_Table(FindRow)(N_M)-1;
2 S_Table(FindRow)(E_S) <= S_Table(FindRow)(E_S)+1;
3 ReservedSize          <= ReservedSize-
                           ReceivedMsgSize;
```

The N-M value can be used as a ranking metric of packets. It represents the number of packets with the same transaction ID as the current packet which traveling inside the network once the packet is injected. In fact, the N-M value indicates the order of packet to return back. Therefore, a packet with a greater N-M value has lower priority to be sent back to the corresponding master core and vice versa.

Reorder-Table and Reorder-Buffer: As shown in Fig. 7, each row of the reorder table corresponds to an out-of-order packet stored in the reorder buffer. This table includes the valid tag (v), the transaction ID (T-ID), the sequence number (S-N) as well as the head pointer (P). In the reorder buffer, the flits of each packet are maintained by a linked list structure providing high resource efficiency with little hardware overhead. On top of that, the goal of using the shared reorder buffer is to support variable packet size and improve the buffer utilization which can also increase the performance by feeding more packets into the network. Fig. 7 exhibits a pointer field adopted to indicate the next flit position in the reorder buffer. Using the proposed structure in Fig. 7, each out-of-order packet updates the reorder table and reorder buffer according to the procedure E, and F. The first three operations in the procedure E, stores the transaction ID and sequence number from the header flit of the out-of-order packet to the available slot indicated by FreeRow in the reorder table; and the last operation in E updates the pointer to point to the available slot in the reorder buffer.

```
Procedure E:
ReorderTable(FreeRow)(V)     <= '1';
ReorderTable(FreeRow)(T-ID) <= HeaderFlit(TranID);
ReorderTable(FreeRow)(S-N)  <= HeaderFlit(SeqNum);
ReorderTable(FreeRow)(P)    <= Current_Free_Slot;
```

```
Procedure F:
ReorderBuf(Current_Free_Slot)(V)      <= '1';
ReorderBuf(Current_Free_Slot)(Data)   <= flit;
ReorderBuf(Current_Free_Slot)(P)      <= Next_Free_Slot;
Current_Free_Slot                     <= Next_Free_Slot;
```

The procedure F is intended to store the incoming flits into the reorder buffer. While Current_Free_Slot shows the current free location in the reorder buffer in order to store the current flit, Next_Free_Slot returns an available slot for the next flit. By repeating the operations in the procedure F, whole of the payload flits will be stored in the reorder buffer. The tail flit can be determined by extracting header flit information. Whenever an in-order packet delivered to the depacketizer unit, the depacketizer controller checks the reorder table for the validity of any stored packet with the same transaction ID and next sequence number. If so, the stored packet will be released from the reorder unit to the depacketizer unit.

4.2. SLAVE-SIDE NETWORK INTERFACE

A slave IP core cannot operate independently. It receives requests from master cores and responds to them. Hence, using reordering mechanism in the slave network interface is completely meaningless. But to avoid losing the order of header information (transaction ID, sequence number, and etc) carried by arriving requests, a FIFO has been considered. After processing a request in a slave core, the response packet should be created by the packetizer. As can be seen from Fig. 5, to generate the response packet, after the header content of the corresponding request is invoked from the FIFO, and some parameters of the header (destination address, and packet size, and etc) are modified by the adapter, the response packet will be formed. However, the components of the slave-side interface in both forward and reverse paths are almost similar to the master-side interface components, except the reorder unit.

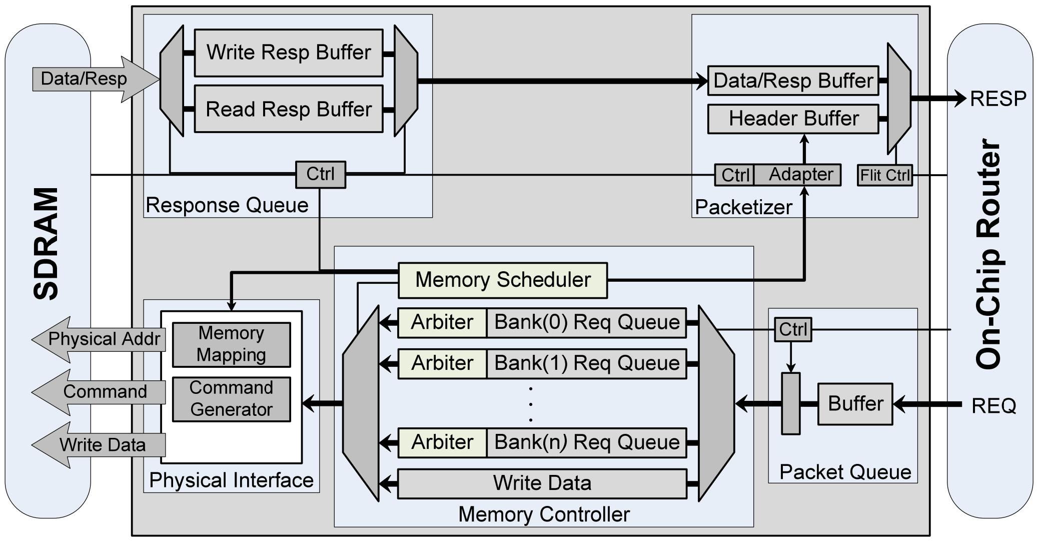

Fig. 8. The proposed memory controller integrated in the slave-side network interface.

```
P(i) : priority of the i(th) request in the queue.
SN   : sequence number of a new request.
RA(i): row address of i(th) request.
CRA  : current row address issued prior.
----------------
Process(input_queue)
Begin
For ‘i:1 to number of requests in input queue’ loop
    If Req is a new packet then
       P(i) <= SN;
    Else
       P(i) <= P(i)+1;
    End if;
 End loop;
End process;
----------------
Process(arbiter)
Begin
 MaxValue1 <=0; select1 <=0;
 MaxValue2 <=0; select2 <=0;
   For ‘i:1 to all requests in input queue’ loop
     If RA(i)= CRA then
       If p(i)>= MaxValue1 then
         select1    <= i;
         MaxValue1  <= p(i);
       End if;
     Else
       If p(i)>= MaxValue2 then
         select2    <= i;
         MaxValue2  <= p(i);
       End if;
    End if;
   End loop;
   If select1 /= 0 then
       select <= select1;
   Else
       select <= select2;
End process;
```

Fig. 9. Pseudo VHDL code of the arbiter in the memory controller.

5. ORDER SENSITIVE MEMORY SCHEDULER

The architecture of the proposed memory controller, dubbed OS from Order Sensitive, is depicted in Fig. 8. As illustrated in the figure, the proposed memory controller is efficiently integrated in the slave-side network interface. Requests after arriving to the network interface on the edge of the network are stored in the respective queues based on their target banks. The data associated with write requests are stored in the write queue. The queues are implemented as the linked list structure which has been described in subsection 4.1. Depending on the sequence number, new received requests in each bank queue obtain a priority value to access the memory. Once a new request enters a queue, the process *input_queue*, shown in Fig. 9, updates the priority value of each request in the queue. The packet's sequence number of received request is assigned as a priority value for this request. In addition, to prevent starvation, the priority values of existing requests in the queue at every *input_queue* event will be increased. As mentioned earlier, each bank arbiter selects a request from the queue with the highest priority value based on the bank timing constraints as the first level of scheduling procedure. Since the *row-first* policy has better memory utilization in comparison with the other bank arbitration policies, the bank arbiters of the presented memory controller also takes advantage of the row-first policy. The bank arbitration policy in our memory controller is shown in Fig. 9. Whenever the *arbiter* process is activated, it tries to find a request which is a row hit and has a higher priority value. If there are not any row hits, the bank arbiter selects the highest priority request which is a row conflict from the queue and issues the SDRAM commands to service the selected request. In the second level of the scheduling procedure, at each memory cycle the memory scheduler decides which request from all bank arbiters should be issued. To simplify the hardware implementation and provide the *bank interleaving*, round robin mechanism is utilized by the memory scheduler.

6. EXPERIMENTAL RESULTS

In this section, we compare the proposed network interface architecture with the baseline architecture by measuring the average network latency under different traffic patterns. Hence, a 2D NoC simulator is implemented with VHDL to assess the efficiency of the proposed method. The simulator models all major components of the NoC such as network interface, routers, and wires.

6.1. SYSTEM CONFIGURATION

In this work, we use a 25-node (5×5) 2D mesh on-chip network configuration for the entire architecture. In this configuration, illustrated in Fig. 10, out of 25 nodes, ten nodes are assumed to be processors (master cores, connected by master NIs) and other fifteen nodes are memories (slave cores, connected by slave NIs). For convenience, we used MS (Master/Slave NI) and MS-OS (Master/Slave NI where slave NIs uses Order Sensitive structure) to represent the proposed architectures. The processors are 32b AXI and the memories, specified in subsection 2.2.1, are DDR2-512MB (t_{RP}-t_{RCD}-t_{CL}=2-2-2, 32b, 4 banks) [21]. We adopt a commercial memory controller and memory physical interface, DDR2SPA module from gaisler ip-cores [26]. In addition, the on-chip network considered for experiment is formed by a typical state-of-the-art router structure including input buffers, a VC (Virtual Channel) allocator, a routing unit, a switch allocator and a crossbar. Each router has 5 input/output ports, and each input port of the router has 2 VCs. Packets of different message types (request and response) are assigned to corresponding VCs to avoid message dependency deadlock [15]. The arbitration scheme of the switch allocator in the typical router structure is round-robin.

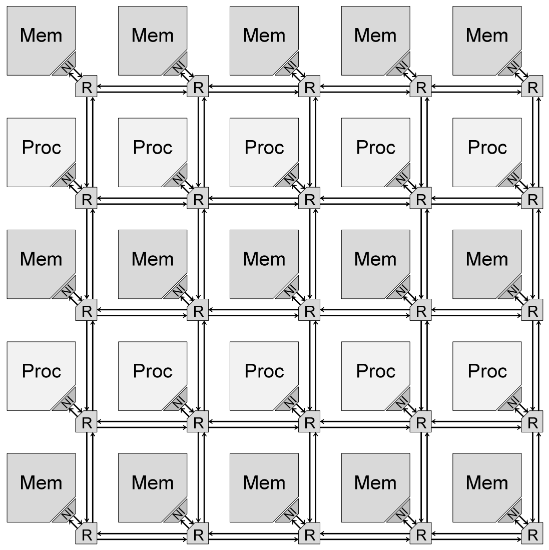

Fig. 10. 5x5 NoC layout.

The array size, routing algorithm, link width, number of VCs, buffer depth of each VC, and traffic type are the other parameters which must be specified for the simulator. The routers adopt the XY [16][17] routing and utilize wormhole switching. For all routers, the data width (flit) was set to 32 bits, and the buffer depth of each VC is 5 flits. The baseline architecture [8][9] (with fixed packet length) uses 1 flit for messages related to read requests and write responses, and 5 flits for data messages, representative of read responses and write requests; the size of read request messages typically depends on the network size and memory capacity of the configured system. As discussed in the Section 4, the message size of the proposed mechanism is variable and depends on the request/response length produced by a master/slave core. As the performance metric, we use latency defined as the number of cycles between the initiation of a request operation issued by a master (processor) and the time when the response is completely delivered to the master from a slave (memory). The request rate is defined as the ratio of the successful read/write request injections into the network interface over the total number of injection attempts. All the cores and routers are assumed to operate at 2GHz. For fair comparison, we keep the bisection bandwidth constant in all configurations. We also set the size of the reorder buffer to 48 words, able to embed 6 outstanding requests with burst size of 8. All memories (slave cores) can be accessed simultaneously by each master core with continuously generating memory requests. Furthermore, the size of each queue (and FIFO) is set to 8×32 bits.

6.2. Performance Evaluation

To evaluate the performance of the proposed schemes, the uniform and non-uniform synthetic traffic patterns have been considered separately for the specified configuration. These workloads provide insight into the strengths and weakness of the different buffer management mechanisms in the interconnection networks, and we expect applications stand between these two synthetic traffic patterns. The random traffic represents the most generic case, where each processor sends in-order read/write requests to memories with the uniform probability. Hence, the memories and request type (read or write) are selected randomly. Eight burst sizes, among 1 to 8, are stochastically chosen regarding the data length of the request. In the non-uniform mode, the traffic consists of 70% local requests, where the destination memory is one hop away from the master core, and the rest 30% traffic is uniformly distributed to the non-local memories. Fig. 11 and Fig. 12 show the simulation results under uniform and non-uniform traffic models, respectively. The proposed network interface architecture has been compared with the baseline. As demonstrated in both figures, compared with the baseline architecture the presented architecture reduces the average latency when the request rate increases under uniform and non-uniform traffic models. One of the foremost reasons of such an improvement is that because the size of packets is not fixed and depends on the request and response lengths, the resource utilization is high and thus, the latency is reduced. Another subtle reason for improving the performance is that getting more free slots in the reorder buffer allows more messages to enter the network.

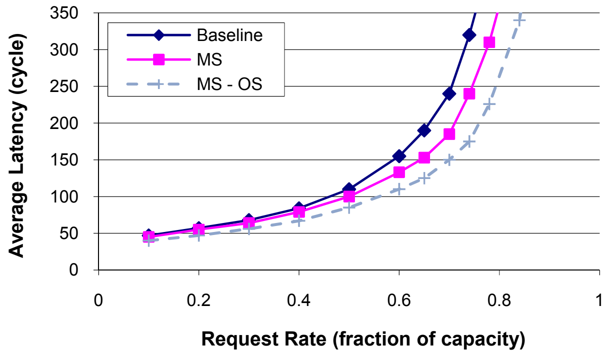

Fig. 11. Performance evaluation of both configurations under uniform traffic model.

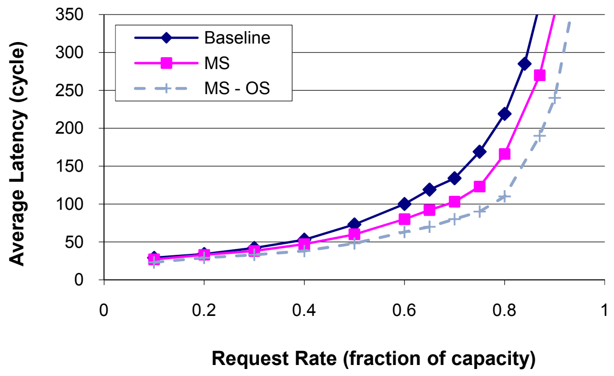

Fig. 12. Performance evaluation of both configurations under non-uniform traffic model.

The average utilization of memories and average latency of the network and memories have been computed near saturation point (0.6) under the uniform traffic profile. As a result of using OS technique, compared with the baseline architecture, the average utilization of memories is improved by 22%. The average memory latency and average network latency are reduced by 19%, and 12%, respectively.

6.3. HARDWARE OVERHEAD

The network interfaces were synthesized by Synopsys Design Compiler using the UMC 0.09μm technology. In addition to the aforementioned configuration of the network interface, the tran_id and seq_id were set to 4-bit and 3-bit respectively. The layout areas and power consumptions of the master-side, slave-side, and OS interfaces are listed in Table 1. Since all queues (and FIFOs) are equal in the size, it will not affect the comparison. Also, comparing the area cost of the baseline model for each proposed network interface indicates that the hardware overheads of implementing the proposed schemes are less than 0.5%. Furthermore, for the slave-side interface within memory controller, since each of the memories utilized in this work, has 4 banks, four bank queues have been implemented in the memory controller.

Table 1. Hardware implementation details.

NI	Area		Power (mW)
	Gates (#)	(μm^2)	
Slave-side	18218	42848	0.274
Master-side	32125	75559	0.441
Slave-side with OS	33218	80848	0.486

7. SUMMARY AND CONCLUSION

To increase the memory bandwidth, accessing multiple memories simultaneously is necessitated, but it requires a reordering mechanism to cope with the deadline. In this work, we presented a high performance dynamic reordering mechanism integrated in the network interface to improve the resource utilization, and overall on-chip network performance. In addition to the resource utilization of the network interface and on-chip network, the utilization of memories considerably affects the network latency. Therefore, we have developed a novel scheduling method for the modern SDRAM memories and integrated in the network interface such that the network and memory latencies reduced effectively in comparison with the baseline architecture. The micro-architectures of the proposed network interfaces which are compatible with the AMBA AXI protocol have been presented. A cycle-accurate simulator was used to evaluate the efficiency of the proposed architecture. Under both uniform and non-uniform traffic models, in high traffic load, the proposed architecture had lower average communication delay in comparison with the baseline architecture.

REFERENCES

[1] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar. A 5-GHz mesh interconnect for a teraflops processor. IEEE Micro, 27:51–61, September-October 2007.

[2] A. Jantsch and H. Tenhunen, "Networks on Chip," Kluwer Academic, 2003.

[3] B. Towles and W. Dally, "Route packets, not wires: on-chip interconnection networks", Proc. DAC 2001.

[4] L.Benini and G.De Micheli, "Networks on chips: a new SoC paradigm", IEEE Computer, January 2002.

[5] C. A. Zeferino, M. E. Kreutz, and A. A. Susin, "RASoC: A Router Soft-Core for Networks-on-Chip", Proceedings of DATE'04, pp. 1530-1591, 2004.

[6] ARM, AMBA AXI Protocol Specification, Mar. 2004.

[7] OCP International Partnership, Open Core Protocol Specification. 2.0 Release Candidate, 2003.

[8] X. Yang, Z. Qing-li, F. Fang-fa, Y. Ming-yan, L. Cheng, "NISAR: An AXI compliant on-chip NI architecture offering transaction reordering processing", in Proc. ASICON, pp. 890-893, 2007, Greece.

[9] W. Kwon, et al., "A Practical Approach of Memory Access Parallelization to Exploit Multiple Off-chip DDR Memories", Proc. DAC, 2008.

[10] A. Radulescu, and et al., "An Efficient On-Chip NI Offering Guaranteed Services, Shared-Memory Abstraction, and Flexible Network Configuration", in Proc IEEE TCAD, 24(1), January 2005.

[11] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," In Proc. of ISCA'00, pp. 128-138, US, 2000.

[12] M. Lai, Z. Wang, L. Gao, H. Lu, K. Dai, "A Dynamically-Allocated Virtual Channel Architecture with Congestion Awareness for On-Chip Routers," in Proceedings of the 46th Design Automation Conference (DAC), pp. 630-633, 2008.

[13] G. L. Frazier and Y. Tamir, "The Design and Implementation of a Multi-Queue Buffer for VLSI Communication Switches," In Proceedings of the IEEE Conference on Computer Design (ICCD), pp. 466-471, 1989.

[14] W. Kwon, S. Yoo, J. Um, and S. Jeong, "In-network reorder buffer to improve overall NoC performance while resolving the in-order requirement problem", In proc. DATE'09, pp. 1058 – 1063, France, 2009.

[15] S. Murali, and et al. "Designing message-dependent deadlock free networks on chips for application-specific systems on chips," In Proc. VLSI-SoC, pages 158-163, 2006.

[16] G. Chiu, "The Odd-Even Turn Model for Adaptive Routing," IEEE Tran. On Parallel and Distributed System, pp 729-738, July 2000.

[17] J. Duato, S. Yalamanchili, and L. Ni, "Interconnection Networks: An Engineering Approach." Morgan Kaufmann, 2002.

[18] S. E. Lee, J. H. Bahn, Y. S. Yang, and N. Bagherzadeh, "A Generic Network Interface Architecture for a Networked Processor Array (NePA)", In proc. ARCS'08, pp. 247-260, 2008.

[19] W. Jang and D. Z. Pan, "An SDRAM-Aware Router for Networks-on-Chip," in proc. of DAC'09, pp. 800-805, US, 2009.

[20] Z. Fang, X. H. Sun, Y. Chen, S. Byna, "Core-aware memory access scheduling schemes," In Proc. of IEEE International Symposium on Parallel & Distributed Processing (IPDPS'09), pp. 1-12, Italy, 2009.

[21] Micron Technology, Inc. Micron 512Mb: x4, x8, x16 DDR2 SDRAM Datasheet, 2006.

[22] H. G. Rotithor, R. B. Osborne, and N. Aboulenein, "Method and Apparatus for Out of Order Memory Scheduling," United States Patent 7127574, Intel Corporation, October 2006.

[23] Jun Shao and Brian T. Davis, "A Burst Scheduling Access Reordering Mechanism", Proceedings of the 13th International Symposium on High-Performance Computer Architecture, 2007.

[24] David E. Culler, Jaswinder Pal Singh, and Anoop Gupta, Parallel Computer Architecture: A Hardware/Software Approach, Morgan Kaufmann Press, 1998.

[25] I. Hur and C. Lin, "Memory scheduling for modern microprocessors," ACM Trans. on Computer Systems, vol. 25, no. 4, Dec. 2007.

[26] Gaisler IP Cores, http://www.gaisler.com/products/grlib/, 2009.