DMNI: A Specialized Network Interface for NoC-based MPSoCs

Marcelo Ruaro, Felipe B. Lazzarotto, César A. Marcon, Fernando G. Moraes
PUCRS University, Computer Science Department, Porto Alegre, Brazil
{marcelo.ruaro, felipe.lazzarotto}@acad.pucrs.br, {cesar.marcon, fernando.moraes}@pucrs.br

Abstract*—Current proposals of NoC-based MPSoC adopt an NI (Network Interface) interconnected to a DMA (Direct Memory Access) module to enable the communication between processors through the NoC. The adoption of both modules decouples computation from communication, and a standard interface at the NI provides an abstract way for designers to connect new cores. However, this architecture is inherited from bus-based architectures and can be optimized, by removing unnecessary interfaces, signals, and registers. This paper presents a specialized communication interface for NoC-based MPSoCs, called DMNI (Direct Memory Network Interface). The DMNI merges the functionalities of the DMA and the NI into a single component, directly connecting the NoC router with the processor memory. To avoid stalls in the communication, the design of the DMNI supports simultaneous packet reception and transmission. A simplified and generic programming interface exposes the DMNI services to the software layer. Results show a reduction in the silicon area and performance improvement in the packet transmission.

***Keywords* —NoC; MPSoC; Network Interface; DMA.**

I. Introduction

The scalability of NoCs enabled to design systems with a large number of PEs into a single integrated circuit. NoCs proposals started at the beginning of the last decade and became the most adopted communication infrastructure for large scale MPSoCs. Besides scalability, NoCs enable parallel communication between PEs, essential features for systems with a high density of processors.

The design of a NoC-based MPSoC requires an NI. The goal of the NI is to decouple the computation from the communication, achieved by an interface between the processor and the NoC that implements the communication protocol to send and receive packets. Fig. 1(a) shows an MPSoC architecture. The MPSoC may be homogenous or heterogeneous, according to the processors' ISA. Fig. 1(b) presents a typical PE architecture [1]. Each PE contains a processor, local memory, NI, DMA, and the NoC router. The router is instantiated within the PE module to simplify the floorplanning and hence the physical synthesis.

A DMA module was already present in the early processing systems. The processor programs the DMA writing in memory mapped registers the initial memory address and the block size to transfer. After programming the DMA module, the processor resumes the execution while the DMA transfers data to/from the memory. For performance reasons, DMA is broadly used at systems that support real-time applications [2].

The architecture of Fig. 1(b) is inherited from bus-based architectures and commonplace in NoC-based MPSoCs designs. The processor has interfaces with different modules (NI, DMA), and the software has APIs to control each one.

The *goal* of this work is to merge both modules into a new one, named DMNI (*Direct Memory Network Interface*), as shown in Fig. 1(c). The main *contribution* is to provide a specialized interface for NoC-based MPSoCs that directly connects the NoC router to the internal memory using a single module. The DMNI supports simultaneous packet reception and transmission, managed by a memory access arbiter. A simplified programming interface exposes the DMNI services to the software layer.

An important feature of the DMNI is the access to two distinct memory blocks to transfer a packet. This feature is important for NoC-based systems because the packet header and the payload are distinct data structures.

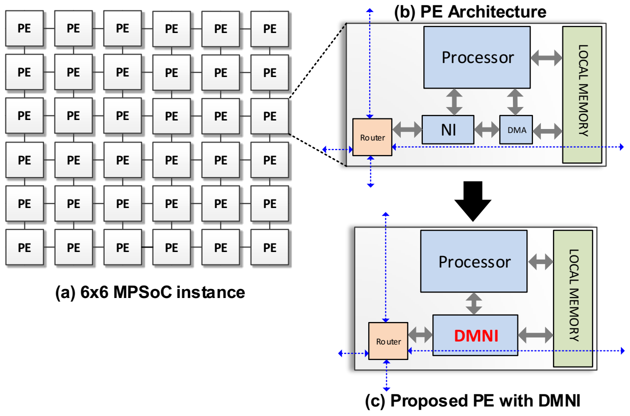

Fig. 1 – MPSoC and PE (Processing Element) organizations.

II. Related Work

Several works in the literature mention a design including a DMA and a NI [3][4]. Derin et al. [5] briefly mention an NI with DMA capabilities, but the work only groups the two modules, without an effective integration. Molnos et al. [6] mention the use of a DMA to send and receive data between two memories (local and shared) through the NoC, without design details.

Attia et al. [7] present a pipelined NI architecture for NoCs. The work presents a modular design, separating the injection and extraction path between the IP and the network sides. The proposed design outperforms other works in terms of latency and power, but the analysis is restricted to the NoC and the NI, without evaluation a complete system. Chouchene et al. [8] add a low-power technique in the NI design of [7], using a stoppable clock technique. The NI is turned off when there is not data to be handled.

Designs proposed in [9] and [10] target heterogeneous MPSoCs by using asynchronous communication architectures. Das et al. [11] propose a fault-tolerant NI to be used in SDM NoCs, with serializers and deserializers to support the spatial division concept.

Kariniemi et al. [12] propose an NI aiming to reduce the interruption frequency in the Micronmesh MPSoC by an interrupt batch mechanism. The results demonstrated a throughput improvement with longer messages. The work assumes the use of a DMA to improve communication latency but without specifying implementation details.

Fanfga et al. [13] propose an NI design combining a Lookup Table (LUT) mechanism and DMA features. The proposal is focused on the packet reception process. The tag segment of the LUT (programmed by the CPU) is compared with the tag information in the packet, and if matches, the address stored in the LUT can be used to start the DMA transfer directly. The goal, as in [12], is to reduce the interruption handler overhead. The Authors evaluate the latency to receive packets, with performance gains in larger packets.

Chen et al. [14] and Ma et al. [15] employ an NI named DMC (*Dual Microcoded Controller*), targeting architectures with distributed shared memory organization. The DMC is a programmable hardware module that connects the memory, processor and NoC. The DMC programming is eased using a microcode approach within two mini-processors. One mini-processor is used to handle local memory requests and the other to handle remote memory requests, by accessing the virtual shared memory space. A synchronizer ensures atomic memory access between the two mini-processors.

Some proposals for NoC-based MPSoCs do not assume a DMA implementation [7][8][9][10][11], focusing only on the NI design. Other works separate the DMA from the NI [3][4] or lack implementation details [5][6][12]. Works focusing on the integration of both modules, either lack validation data [13] or cover specific implementations [14][15]. As our proposal, works [12][13][14] explore an NI design including a system perspective, identifying bottlenecks not addressed in previous works, as the cost to handle interruptions by the processor attached to the router NoC.

This paper has two main *contributions*. The first one is to present a *unified* design interfacing the NoC router with the memory. The second contribution is a *simple and generic API* to program this module, simplifying the software development.

III. Communication Model

Each processor of the MPSoC executes a simple operating system (*μkernel*) with multitasking support and a unicast communication API. This API is used by the μkernel to send and receive packets. Packets may be related to the user inter-task communication model (MPI or shared-memory), or to the system management. Fig. 2 presents the flow to send and to receive a packet between two different PEs.

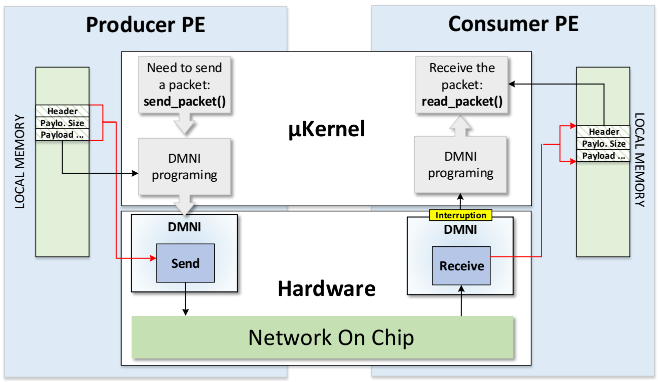

Fig. 2 – Inter-PE communication flow.

When the μkernel of the producer PE needs to send a packet, it calls a *send_packet()* function that programs the DMNI to start to send the packet from the memory. At the consumer side, when the DMNI receives a packet it interrupts the processor. The interruption handler calls the *read_packet()* that programs the DMNI to read the packet. Once the packet is completely received, the μkernel executes the actions related to the contents of the packet. For example, if the packet contains data to a user task t_i, the packet (message in the user task context) is written in the t_i memory space, and t_i is scheduled to execute. The next section details the API functions, *send_packet()* and *read_packet()*.

Fig. 3 details the packet and message structures. From the NoC point of view, the packet has a header and a payload. The packet header contains the target router address and the payload size. From the task point of view, a message contains:

- *message header*: encapsulates the packet and service header (e.g. message reception, task mapping, request for a message);
- *message payload*: optional field. It may contain for example user data or an object code of a task.

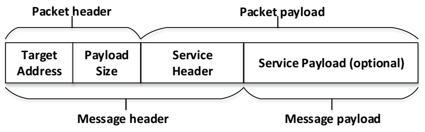

Fig. 3 – Packet and message structures.

IV. Proposed DMNI Design

Fig. 4 details the DMNI architecture. The DMNI contains 3 main modules: *send*, *receive*, and *arbiter*. The arbiter manages the memory accesses for both modules, enabling simultaneous send and receive operations. The μkernel controls the DMNI through memory-mapped registers (MMRs). The DMNI design is *generic* because it enables to send and receive any type of data, not necessarily related to the message structure presented in Fig. 3.

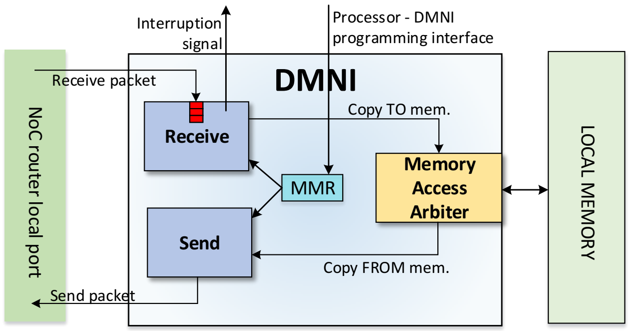

Fig. 4 – DMNI architecture.

A. Send Module

This section details the process related to the producer PE (left side of Fig. 2). The role of the send module is to inject a packet into the NoC. The *particular feature of this module is the possibility to transfer two memory blocks* (message header and payload) as a single transfer. This feature is important because the message header and the payload are distinct data structures, mapped to different memory regions.

Fig. 5 presents the *send_packet()* function provided in the DMNI API. It receives, respectively, the first (message header) and second (message payload) memory sizes and addresses. If the DMNI is transmitting a packet (**DMNI_SEND_ACTIVE=1**), the procedure stays at line 2 until the release of the DMNI module. At lines 3 and 4, the first memory block is configured. If the message has a payload, at lines 6 and 7, the second memory block is configured. At line 8, it is written the operation type, i.e., read from the memory. Finally, at line 9, the DMNI is allowed to start the packet transmission.

```
1.   void send_packet(mem_size_1, mem_addr_1, mem_size_2,
     mem_addr_2){
2.     while (MemoryRead(DMNI_SEND_ACTIVE));
3.     MemoryWrite(DMNI_SIZE, mem_size_1);
4.     MemoryWrite(DMNI_ADDRESS, mem_addr_1);
5.     if (mem_size_2 > 0){
6.       MemoryWrite(DMNI_SIZE_2, mem_size_2);
7.       MemoryWrite(DMNI_ADDRESS_2, mem_addr_2);
8.     MemoryWrite(DMNI_OP, READ);
9.     MemoryWrite(DMNI_START, 1);
10.  }
```

Fig. 5 – *Send_packet()* function, executed in the μkernel of the processor.

Fig. 6 presents the FSM (Finite State Machine) controlling the Send module. Initially, the FSM waits the configuration of the MMRs (**WAIT** state) by the *send_packet()* function. When lines 8-9 of the function are executed, the FSM goes to the **LOAD** state, and the FSM assert a *send_active* signal to the arbiter to request access to the memory. The **LOAD** verifies if the local port of the router may receive data (*credit=1*) and if the arbiter allows a read operation (*read_enable=1*). If both conditions are satisfied, the data is read from the memory and injected into the router local port (state

COPY_FROM_MEM). Whenever the arbiter or the local port disables the transmission, the FSM returns to the **LOAD** state. The FSM sends the first memory block and then changes the address pointer to the second memory block (if configured) to transmit the remaining data.

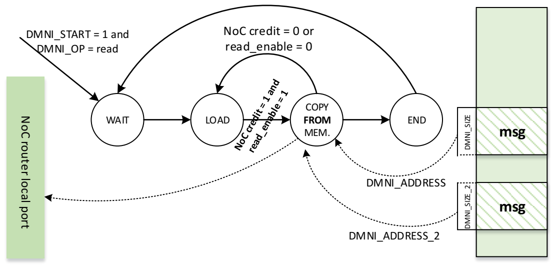

Fig. 6 – FSM controlling the send module.

Fig. 7 presents the transmission of a packet configured with two memory regions. Two memory regions are configured: one starting at address 0x910 with contents {1, 7, A1, A2, A3}, and the second one starting at address 0x8c8 with contents {B1, B2, B3, B4}. Between clock cycles 3 to 10, the first memory block is transmitted (signal *data_out*). At clock cycle 11, the *send_size* becomes zero, changing the *mem_addr* signal to the second memory region, and the second part of the packet is transmitted. *The gap to change the memory region is only two clock cycles*. In a standard implementation (DMA+NI), which requires programming the DMA twice, the minimal gap is 22 clock cycles, penalizing the transmission of packets with a small payload.

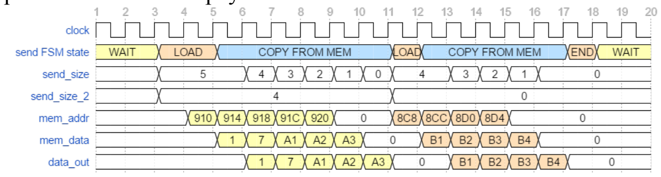

Fig. 7 – Packet transmission by accessing two memory blocks.

B. Receive Module

This section details the process related to the consumer PE (right side of Fig. 2). Fig. 8 details the receive module. It contains two FSMs and a 16-flit buffer. The buffer depth is parameterizable at design-time.

When a packet arrives at the local port of the NoC router, the **HEADER** state reads the first flit of the packet, interrupting the processor. Next, the **PAYLOAD_SIZE** state reads the payload size and advances to the **DATA** state, which reads the remaining flits of the packet. The buffer receives all incoming flits. The NoC stalls when the buffer becomes full.

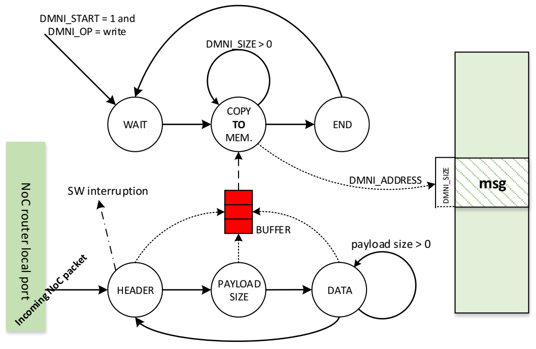

Fig. 8 – FSM controlling the receive module.

Fig. 9 presents the interruption handler process (*read_packet()* function). The *read_packet()* function writes into MMRs the amount of data to receive (line 2), the memory address to store the packet (line 3), the DMNI operation (line 4), and a *start* command (line 5). The µkernel waits the complete reception of the packet (line 6) to safely read the packet content from memory, and executing the actions related to the packet service.

```
1.  void read_packet (init_addr, packet_size)
2.     MemoryWrite(DMNI_SIZE, packet_size);
3.     MemoryWrite(DMNI_ADDRESS, init_addr);
4.     MemoryWrite(DMNI_OP, WRITE);
5.     MemoryWrite(DMNI_START, 1);
6.     while (MemoryRead(DMNI_RECEIVE_ACTIVE));
7.  }
```
Fig. 9 – *Read_packet()* function, executed in the µkernel of the processor.

The *write* and *start* conditions start the FSM at the top of Fig. 8 (lines 4 and 5 of the *read_packet()*). This FSM transfer the data stored in the buffer to the local memory (state **COPY_TO_MEM**). To write into the memory, this second FSM assert the *receive_active* signal to the arbiter to request access to the memory. The arbiter can grant access to the memory by asserting the signal *write_enable*. If the arbiter does not grant access to the memory, the FSM stays blocked in the **COPY_TO_MEM** state.

Note that both FSMs of the receive module work in parallel. The first one receives data from the NoC storing the flits into the buffer, and the second one reads the buffer storing the data into the memory.

C. Memory Access Arbiter

The arbiter enables concurrent memory accesses to receive and to send packets. With such feature, the PE may receive new data and concurrently inject new packets into the NoC, interleaving the memory accesses. A round-robin (RR) arbiter enables this feature, by controlling two signals: *read_enable* (send) and *write_enable* (receive). A timer (DMNI_TIMER) controls the amount of time each module may access the memory.

Fig. 10 presents the FSM controlling the arbiter. A signal named *round* selects the module to grant access. The receive and send FSMs assert the signals *send_active* and *receive_active*, respectively. When the arbiter goes to **SEND** state, the *read_enable* signal is asserted, enabling the send module to access the memory. The FSM stays in this state while *send_active* is asserted, or the timer expired and the other module requested access to the memory. Note that the arbiter may stay in the **SEND** state for periods larger than the timer limit if the other module does not request access to the memory. The **RECEIVE** state has the same behavior of the **SEND** state. When the FSM returns to the **ROUND** state, the *round* signal inverts, changing the order to verify which module must be served.

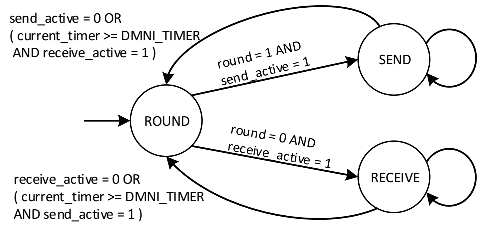

Fig. 10 – FSM controlling the arbiter module.

Fig. 11 presents the arbiter operation. For the sake of clarity, the DMNI_TIMER was configured to 5 clock cycles (*cc*). At the cc=2, the *receive_active* becomes true, signalizing to the arbiter that the send module needs to be stopped, and the RR scheduling executed. Note the timer value at this cc (0xF) is larger than the DMNI_TIMER (5) because only the send module is active. The RR execution occurs at cc=3, the timer returns to zero, the *round* signal is inverted, the *read_enable* becomes false, and the arbiter releases the receive module by activating the *write_enable* signal. The RR executes again at cc=10, and now the send module is released (*receive_active*←*1*). This interleaved operation continues while both *receive_active* and *send_active* signals remain asserted.

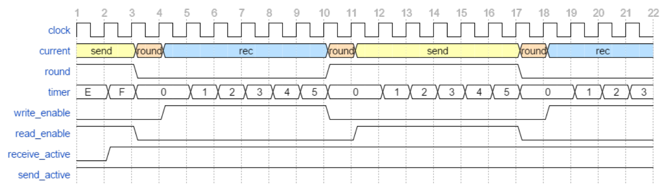

Fig. 11 - Memory access scheduling.

V. RESULTS

The DMNI was implemented using synthesizable VHDL, integrated into a public-available MPSoC [1]. The baseline design, with separated modules, is named DMA+NI.

A. *Latency to transmit packets*

Fig. 12 presents the latency to transmit packets with different sizes. The latency is measured from the moment when *send_packet()* is invoked up to the end of the execution of the *read_packet()*. Note that this latency includes the network latency, the interruption handling, and the context saving. The network latency represents a small fraction in the total latency, corresponding to 5 clock cycles per hop (in non-congested scenarios).

It is possible to observe that in both scenarios the latency grows linearly with the packet size. DMNI had a latency decrease of 116 clock cycles per 128 flits compared to DMA+NI. This reduction comes from two main reasons. The first one is related to software. For the DMA+NI implementation, the processor has to wait the transmission of the message header and then program the DMA to transmit the message payload. Using the DMNI, the processor programs once the memory regions, without waiting the transmission of the message header. The second reason is related to hardware. The unified DMNI design can transmit 1 flit per clock cycle, while in the DMA+NI it is necessary 2 clock cycles to inject one flit into the NoC due to the interface protocol between the two modules.

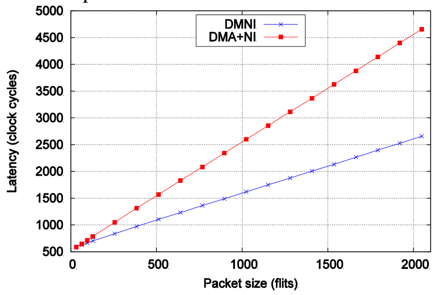

Fig. 12 – DMNI and DMA+NI latency comparison.

A second latency evaluation concerns the latency in a real application, an MPEG decoder. The latency to decode one frame with the DMNI presented a reduction of 12.3% compared with DMA+NI. Further, the impact in the application execution time of MPEG is 15% lower with the DMNI design. Such results highlight the performance improvement offered by the DMNI, which specializes and simplifies the PE design.

B. *Area and State-of-the-Art comparison*

Both designs, DMA+NI and DNMI, were synthesized using the Cadence ASIC design flow for a 65nm CMOS technology, and prototyped in FPGA (Xilinx XC5VLX330). Table 1 presents the area for the proposed DMNI, DMA+NI, and related works (those that have area report).

Comparing the baseline design (DMA+NI) with the proposed DMNI, there is a small area reduction (3.47%) when targeting an ASIC implementation. On the other side, for FPGAs, an important reduction in the number of flip-flops is observed – 48%, with an increased number of LUTs – 11.5%. The reduction observed in the number of flip-flops comes from the smaller number of registers required by the DMNI implementation.

Comparison to related works is difficult due to different specific objectives and use different technologies. Observing the table, works from Derin [5] and Ma [15] (that use a DMA and a NI) have similar FPGA area results compared with DMNI.

Table 1 - Area comparison.

Author	**FPGA/ASIC**	**Work Goal**	**LUTs**	**FFs**	**Area**
Derin et al. [5]	FPGA (Xilinx XC6VLX240T)	Network Adapter (DMA + NI)	879	577	N/A
Chouchene et al. [8]	FPGA (Xilinx XC5VLX30)	Power-efficient NI (Credit Based)	420	590	N/A
Matos et al. [9]	ASIC (0.18um)	Asynchronous NI	N/A	N/A	18735 um^2
Ma et al. [15]	FPGA (Zynq7000)	Programmable NI (DME)	1163	313	N/A
Baseline design	FPGA (Xilinx XC5VLX330) / ASIC (65 nm)	Standard PE arch.	682	787	22141 um^2
This proposal		Unified design - DMNI	761	409	21371 um^2

VI. CONCLUSION

This paper presented the integration of DMA and NI modules into a single hardware component named DMNI. This integration removed unnecessary interfaces, registers, and signals, being specialized for NoC-based MPSoCs. The adoption of the DMNI reduced the latency to transmit packets, execution time, and area (FPGA and ASIC technology) when compared to the baseline implementation (DMA+NI). This paper showed the need to adopt specialized architectures for many-core systems, as NoC-based MPSoC, targeting simplified software programming together with an efficient hardware implementation.

ACKNOWLEDGMENTS

The Author Fernando Moraes is supported by CNPq - projects 472126/2013-0 and 302625/2012-7, and FAPERGS - project 2242-2551/14-8.

REFERENCES

[1] Carara E.; Oliveira, R; Calazans, N.; Moraes, F. "HeMPS - A Framework for NoC-Based MPSoC Generation". In: ISCAS 2009, pp. 1345-1348l.

[2] Laplante, P. A.; Ovaska, S. J. "Real-time systems Design and Analysis". Wiley-IEEE Press, 4 edition, 2012.

[3] Palumbo, F.; Pani, D.; Congiu, A.; Raffo, L. "Concurrent hybrid switching for massively parallel systems-on-chip: the CYBER architecture". In: Computing Frontiers, 2012 , pp. 173-182.

[4] Arnold, O.; Fettweis, G. "Adaptive runtime management of heterogeneous MPSoCs: Analysis, acceleration and silicon prototype". In: SoC, 2014, 4p.

[5] Derin, O.; et al. "A system-level approach to adaptivity and fault-tolerance in NoC-based MPSoCs: The MADNESS project". Journal Microprocessors & Microsystems, 2013. v. 37(6-7).

[6] Molnos, A.; et al. "Decoupled inter- and intra-application scheduling for composable and robust embedded MPSoC platforms". In: SCOPES, 2012, pp. 13-21.

[7] Attia, B.; et al. "A new pipelined network interface for Network on Chip with latency and jitter optimization". In: ICM, 2011, pp.1-6.

[8] Chouchene, W.; Attia, B.; Zitouni, A.; Abid, N.; Tourki, R. "A low power network interface for network on chip". In: SSD, 2011 pp.1-6.

[9] Matos, D.; Carro, L.; Susin, A. "Associating packets of heterogeneous cores using a synchronizer wrapper for NoCs". In: ISCAS, 2010.

[10] Swaminathan, K.; Lakshminarayanan, G.; Seok-Bum Ko. "High Speed Generic Network Interface for Network on Chip Using Ping Pong Buffers". In: ISED, 2012, pp.72-76.

[11] Das, A.; Kumar, A.; Veeravalli, B. "Fault-tolerant network interface for spatial division multiplexing based Network-on-Chip". In: ReCoSoC 2012, pp. 1-8.

[12] Kariniemi, H.; Nurmi, J. "High-Performance NoC Interface with Interrupt Batching for Micronmesh MPSoC Prototype Platform on FPGA". In: NORCHIP, 2010, pp.1-6.

[13] Fangfa, F.; Xin'na, H.; Jinxiang, W.; Mingyan, Y. "A novel communication strategy between PE and NI in NoC-based MPSoC". In: RCSLPLT, 2010, pp.374-377.

[14] Chen, X.; Lu, Z.; Jantsch, A.; Chen, S. "Supporting Distributed Shared Memory on multi-core Network-on-Chips using a dual microcoded controller". In: DATE, 2010, pp. 34-44.

[15] Ma, R.; Hui, Z.; Jantsch, A. "A packet-switched interconnect for many-core systems with BE and RT service". In: DATE, 2015, pp.980-983.