26th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

RVNoC: A Framework for Generating RISC-V NoC-based MPSoC

Mahmoud A. Elmohr*, Ahmed S. Eissa*, Moamen Ibrahim†, Mostafa Khamis‡, Sameh El-Ashry§, Ahmed Shalaby¶, Mohamed AbdElsalam‡, M. Watheq El-Kharashi§
*Department of Communications and Electronics, Faculty of Engineering, Alexandria University, Alexandria, Egypt,
† Center for Ubiquitous Computing, University of Oulu, Oulu, Finland,
‡Mentor Graphics, Cairo, Egypt,
§Department of Computer and Systems Engineering, Ain Shams University, Cairo, Egypt,
¶Department of Computer Science, Faculty of Computers and Informatics, Benha University, Egypt
Email: mahmoud.a.elmohr@ieee.org, es-ahmed.mahmoud1217@alexu.edu.eg, moamen.ibrahim@student.oulu.fi, mostafa_abdelaziz@mentor.com, samehelashry@ieee.org, ahmed.shalaby@ejust.edu.eg, mohamed_abdelsalam@mentor.com, watheq.elkharashi@eng.asu.edu.eg

***Abstract*—With the increase in the number of cores embedded on a chip; The main challenge for Multiprocessor System-on-Chip (MPSoC) platforms is the interconnection between that massive number of cores. Networks-on-Chip (NoC) was introduced to solve that challenge, by providing a scalable and modular solution for communication between the cores. In this paper, we introduce a configurable MPSoC framework called RVNoC that generates synthesizable RTL that can be used in both ASIC and FPGA implementations. The proposed framework is based on the open source RISC-V Instruction Set Architecture (ISA) and an open source configurable flit-based router for interconnection between cores, with a core network interface of our design to connect each core with its designated router. A benchmarking environment is developed to evaluate variant parameters of the generated MPSoC. Synthesis of a single building block containing a single core without any peripherals, a router, and a core network interface, using 45nm technology, shows an area of 102.34 kilo Gate Equivalents (kGE), a maximum frequency of 250 MHz, and a 9.9 μW/MHz power consumption.**

***Keywords*—Multiprocessor System-on-Chip (MPSoC), Network-on-Chip (NoC), RISC-V, System-on-Chip (SoC)**

I. Introduction

Multiprocessor System-on-Chip (MPSoC) platforms have been introduced to satisfy the complexity of modern applications. Such systems offer more modularity, scalability, and processing power than ever before. One of the major challenges facing the performance of MPSoC architectures is the communication between different Intellecutal Property cores (IPs) [1]. Traditionally, communication was handled via bus or crossbar structures. However, with Moore's law forcing the integration of more and more IPs on a single chip, these structures began to fail, especially in terms of scalability, throughput, bandwidth, and power consumption [2]. Networks-on-Chip (NoC) has emerged as a solution to interconnection challenges in modern digital systems [3]. NoC platforms provide significantly high on-chip communication bandwidth, which most MPSoC systems need.

RISC-V is a new extensible open-source Instruction Set Architecture (ISA) [4] developed to support computer architecture research and education and can also provide a reliable cost-free open ISA for industrial applications. The ISA itself is highly extensible with numerous open extensions such as atomic instructions, bit-manipulation, vector operations, and floating-point operations.

In this work, we introduce using RISC-V as the main Processing Element (PE) for lightweight MPSoC systems. Multiple NoC-based MPSoC systems have been introduced in literature, but none of them used any RISC-V implementation. In [5], a NoC-Based MPSoC is introduced to use clusters of a configurable 32-bit Very Large Instruction Word (VLIW) processor architecture called CoreVA, which is used mainly for multimedia and streaming applications. That MPSoC uses two hierarchical levels of communication with the inter-cluster communication implemented via a bus structure and the cluster-to-cluster communication handled using a packet-based NoC system. HeMPS is a framework for generating NoC-based MPSoC systems [6]. This framework uses a small 32-bit RISC implementation of a MIPS-I like-ISA and a simple 2-D mesh NoC, with deterministic, distributed XY routing algorithm.

II. Proposed Architecture and implementation

To emphasize scalability, the architecture of RVNoC is based on a replicated building block that is divided into three main parts, processing tile, router, and the Core Network Interface (CNI). RVNoC uses two levels of communication; the RISC-V core communicates with its dedicated peripherals within the tile using an Advanced eXtensible Interface (AXI) bus, and with other cores via a network of routers. The detailed architecture of a single building block is illustrated in Figure 1.

A. RISC-V Processing Tile

Several hardware implementations of the RISC-V ISA exist [7] [8]. This work mainly uses RI5CY as the core Central Processing Unit (CPU) for the proposed MPSoC [9]. RI5CY is a 32-bit 4-stage pipelined RISC-V CPU. It mainly uses the RV32I base integer instruction set [4], with a single-cycle multiplication and other DSP extensions. To program a single

0-7695-6380-5/18/$31.00 ©2018 IEEE
DOI 10.1109/PDP2018.2018.00103

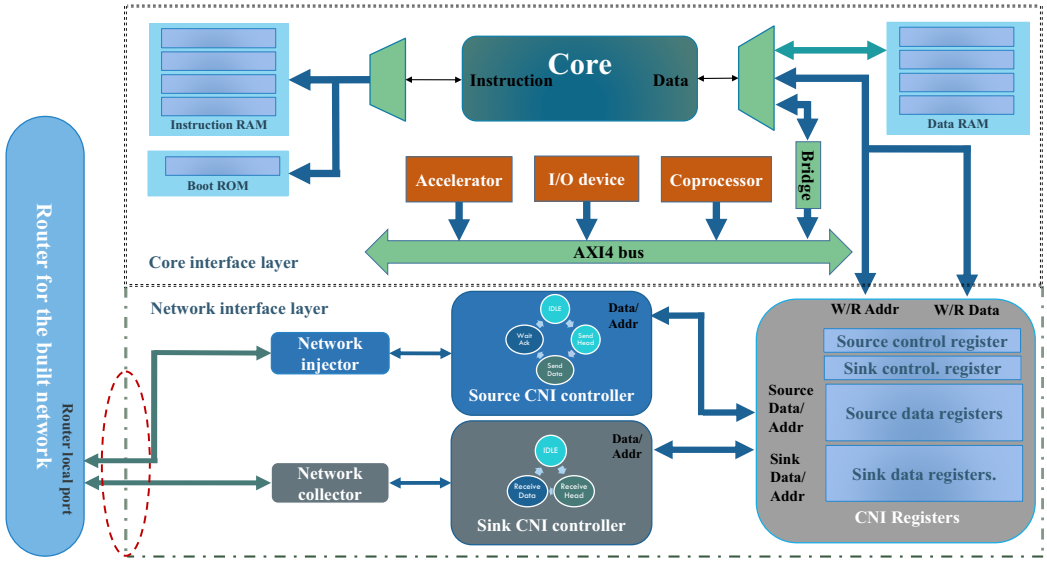


Figure 1: Detailed architecture of RVNoC building block.

RI5CY core, a custom GCC RISC-V toolchain is used. The core also uses 32 vectored interrupts. As depicted in Figure 1, other PEs as coprocessors, accelerators, I/O devices and, IP cores can be connected to the RI5CY core via the AXI4 bus. Although this work mainly focuses on the RI5CY core, other RISC-V implementations can be used and evaluated using the same methodology and architecture.

B. Configurable Router

RVNoC uses the open source router discussed in [10] which is considered a configurable, complex router with extensive options to adjust network parameters. These parameters not only control the network flow but also generate a wide variety of different NoC architectures. This router is flit-based, meaning that packets in the network are divided into flits; head, body, and tail flits. Router nodes in the network communicate with each other via two buses; channel bus, containing the traversed flits and flow control bus, containing the signals that manage the transfer of packets inside the network. Each router node is connected by its local port's channel and flow control buses to a network interface which is linked to a RI5CY core. This router was also verified and tested using the environment in [11][12]. The 3D networks routers [13] could be studied and implemented in future work.

C. RISC-V CNI Architecture

The core network interface is the bridge connecting the core to the NoC. Many CNI architectures already exist with some extra functionalities [14][15]. The CNI used in our work is a memory-mapped I/O interface, which is given an address space within the memory to facilitate accessing the core using the standard load and store instructions. The main advantage of that kind of interfaces is that they need no further modifications to the architecture of the attached core or the ISA.

As shown in Figure 1, the interface can be divided into four main elements: CNI data registers, CNI control registers, CNI controllers and the network injector and collector.

1) CNI Data Registers

CNI Registers generally are memory-mapped registers that can be directly accessed by the core and CNI controllers. CNI registers are directly connected to the core via multiplexers/demultiplexers and not via the AXI bus to accelerate communication.

CNI data registers contain: (i) the source bank in which the core injects packets to transmit, (ii) the sink bank in which the router injects packets it received for the attached core. Both source and sink bank sizes are set to the maximum payload a packet of the underlying network can hold.

2) CNI Control Registers

CNI control registers consist of two registers: source and sink control registers. The source control register contains: (i) Sending control bits (S-bits), which are two control bits accessed by the core and the CNI source controller to synchronize between them and indicate the transmission state. (ii) Packet size, written by the core to indicate the number of flits in the injected packet. (iii) Destination address, written by the core to indicate the destination core of the injected packet.

Similar to the source control register, the sink control register contains the fields: receiving control bits (R-bits), accessed by the core and the sink CNI controller, and packet size, which is the same as the corresponding one in the source control register, but is set by the sink CNI controller to indicate number of flits of the received packet.

3) CNI Controllers

CNI controllers are the abstraction layer between CNI registers and the network injector, and collector which directly interact with the router. Two versions are implemented: source and sink CNI controllers. The source CNI controller is responsible for handling the source control register, loading data from CNI data registers, formatting the packet in the appropriate number of flits, and sending it to the network injector, which in turn injects the data as flits in the network with the required flow control. At the beginning the source CNI checks if the S-bits are "11" which indicates a packet sent by the core, then it sets the S-bits to "01" in the source control register to indicate that it is currently processing the packet in the source control registers and then resets them to "00" when it is done, allowing the core to send new packet. While the sink CNI controller is responsible for interfacing with the network collector to get the received packets, extracts the data from the packets and sends them to the core through the CNI data registers while handling the CNI sink control register. The sink CNI controller sets the R-bits to "01" in the sink control register while writing a received packet to the sink control registers and then resets them to "00" when it is done to inform the core of the received packet.

4) Network Injector and Collector

The network injector and collector are responsible for injecting and collecting packets from the router, respectively. They act as a normal I/O port of a router to supply necessary mechanisms to the attached router, like flow control and credits. The separation between CNI controllers and the network injector and collector is meant to isolate the core control from the router control, and thus, facilitate connecting other network infrastructures to the framework by only replacing the network injector and collector.

III. Simulation Platform

To allow fast prototyping and performance analysis, a complete simulation environment was built as shown in Figure 2.

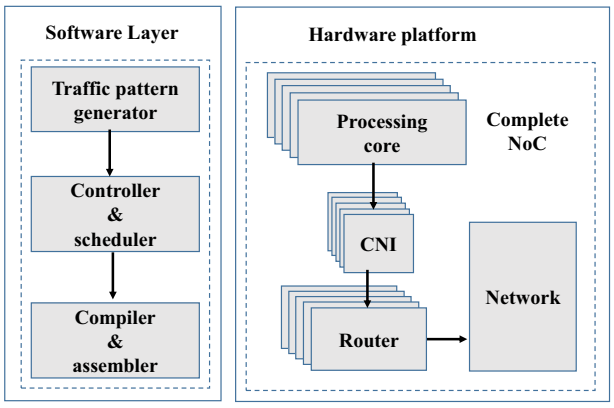


Figure 2: Simulation environment

A. Software Layer for Simulation Environment Setup

The software layer of the simulation environment consists of:

1) Traffic Generator

The traffic generator generates packets according to the adopted traffic pattern with a specified packet length distribution at each specified inter-arrival time distribution. Packets along with their destination addresses and sizes are stored in the data memory of each core to be extracted and injected by the core. The test layer supports uniform, hot-spot, transpose, bit-shuffle, bit-rotation, bit-reversal, tornado, and neighbor traffic patterns [3].

2) Scheduler and Software Controller

The scheduler is responsible for ensuring that the traffic pattern timings are met by generating an interrupt to the core to indicate that a packet should be injected in this cycle and maintaining an external register in the data memory that contains the total number of packets the core should have injected in this cycle. On the other hand, the software controller consists of two low-level control functions namely: Inject and Collect, which are developed to facilitate data transmission between cores in the higher level. These functions are implemented in assembly using both polling and interrupt-based approaches. In the following paragraphs, the implementation of the two functions are depicted.

a) Inject Function: The pseudocode shown in Algorithm 1 describes the polling-based Inject function. While for the interrupt-based approach, after checking the S-bits, if the last packet is already injected, the function will be the same as in the polling-based approach. But, if the previous packet is not yet injected, the function will put the function's arguments in a queue and enable the injection interrupt. Eventually when CNI controller finishes sending the previous packet, it will trigger the injection interrupt, which in turn loads the packet into the CNI data registers.

Algorithm 1 Inject function pseudocode

```
1:  function Inject(Packet[], Size, Addr)
2:      {S_bits, OldSize, OldAddr} ← SRC_Ctrl_Reg
3:      if S_bits ≠ "00" then
4:          return 0
5:      end if
6:      S_bits ← "10"
7:      SRC_Ctrl_Reg ← {S_bits, Size, Addr}
8:      for i from 0 to Size do
9:          SRC_Regs[i] ← Packet[i]
10:     end for
11:     S_bits ← "11"
12:     SRC_Ctrl_Reg ← {S_bits, Size, Addr}
13:     return 1
14: end function
```

b) Collect Function: The polling-based approach of the Collect function is described in Algorithm 2. Similarly for the interrupt-based approach, after checking the R-bits, if there is a packet ready for loading, the function will act normally as in the polling-approach. But, if no packet exists, the function will put the packet's array pointer in a queue and enable the collection interrupt. Eventually, when the CNI controller

receives a packet, it will trigger the collection interrupt, which in turn loads the packet into the packet's array using the pointer in the queue.

Algorithm 2 Collect function pseudocode

```
1: function Collect(Packet[])
2:     {R_bits, Size} ← SINK_Ctrl_Reg
3:     if R_bits ≠ "00" then
4:         return 0
5:     end if
6:     R_bits ← "10"
7:     SINK_Ctrl_Reg ← {R_bits, Size}
8:     for i from 0 to Size do
9:         Packet[i] ← SINK_Regs[i]
10:    end for
11:    R_bits ← "11"
12:    SINK_Ctrl_Reg ← {R_bits, Size}
13:    return Size
14: end function
```

Both functions are integerated in each core's program code and loaded into its instruction memory.

B. Hardware Platform

The hardware platform is responsible for generating the MPSoC system according to the specification and configurations given in Table I. The hardware platform also generates a synthesizable Verilog RTL that can be later used for both ASIC and FPGA implementations. The generated RTL along with the code of the desired application can be simulated using any HDL simulator.

Table I: The router's parameters and configurations.

Parameters	**Configurations**
Topology	Mesh, torus, ring, and tree.
Flow control	store and forward, virtual channel and wormhole flow controls.
Number of Virtual Channels (VCs)	Variable number of virtual channels per port.
Buffers size	Variable number for total input buffer size per port in flits.
Buffer management	Static and dynamic buffer management.
Flit size	Variable number of bits for each flit.
Routing algorithm	Oblivious, deterministic, and adaptive routing.
Arbiter	Round robin, Tree, Matrix and Priority arbiters.
Maximum payload length	Variable number of body flits per packet.
VC allocation type	Separable input-first, Separable output-first, and Wave front-based VC allocation.

IV. Results

In this section, synthesis results of the proposed architecture and performance analysis of different configurations are illustrated. The synthesis results are obtained using a 64-bit NXN mesh topology credit-based 2-virtual-channel router using deterministic dimension order routing with static buffer management and Round robin separable virtual channel and switch allocators.

A. Physical Implementation Results

A single MPSoC building block containing a single RI5CY core without any peripherals, CNI and a single router is synthesized using 45nm technology. The obtained physical implementation results depicted in Table II show relatively low area and low power implementation, which is suitable for high performance low power parallel applications. The area is reported in μm^2 and kilo Gate Equivalent (kGE).

Table II: Physical implementation results.

Total area (μm^2)	81670.77
Total area (kGE)	102.34
Core	55.75 (54.48%)
Router	32.67 (31.92%)
CNI	13.91 (13.60%)
Maximum frequency (MHz)	250
Power consumption (μW/MHz)	9.9
Operating Voltage (V)	1.1

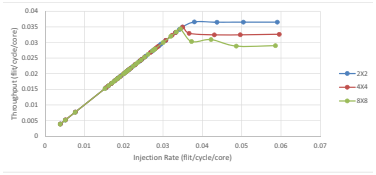
The CNI shows relatively low area, representing only 14% of the total area. Also, considering that the processing tile should also contain other peripherals attached to the AXI bus, it should result in even lower area percentage for the interconnection as a whole compared to the PEs.

B. Performance Results

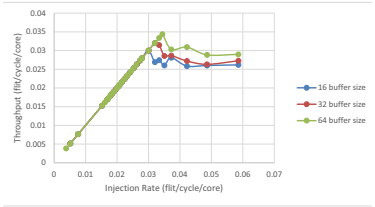
Different configurations of the MPSoC have been evaluated by the proposed benchmarking environment. Throughput and latency results versus different injection rates using some of the available configurations are depicted in this section. Injection rates and throughput are measured in flits per cycle per router to ensure fair results representation for different network sizes. Latency in cycles measures the time spent for a flit to traverse from the core's data memory until it is received by the addressed core and stored in its data memory.

Measuring the performance results from core to core results in higher latency and lower throughput compared to measuring router to router only. This happens due to the fact that the core has a software cost for transferring the packet from data memory to CNI registers and vice versa, and also for the handshake with CNI control registers. This greatly affects the performance, resulting in a real life practical impacts, rather than theoretical ones.

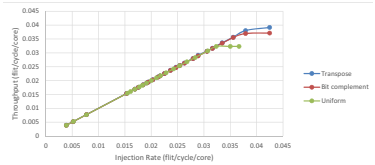
Figures 3a and 3d show the throughput and latency results respectively for 4-ary, 16-ary and 64-ary MPSoC using 1 VC and 64-bit buffer size with uniform distribution traffic pattern. The results clearly show that lower mesh sizes have higher throughput and lower latency, as expected in mesh networks. Figures 3b and 3e show the throughput and latency results respectively for 16-bit, 32-bit and 64-bit buffer sizes using 64-ary MPSoC with uniform distribution traffic pattern. While larger buffer sizes obviously have positive effects on network performance, however, it greatly affects the physical implementation performance, especially increasing area and power consumption. Real life applications tend to generate



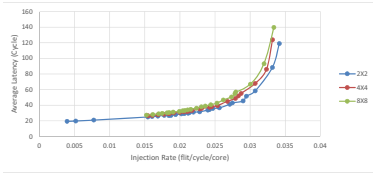
(a) Throughput results for different MPSoC mesh sizes.



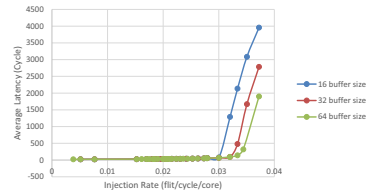
(b) Throughput results for different buffer sizes.



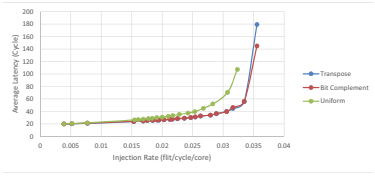
(c) Throughput results for different traffic patterns.



(d) Latency results for different MPSoC mesh sizes.



(e) Latency results for different buffer sizes.



(f) Latency results for different traffic patterns.

Figure 3: Throughput and latency results for different RVNoC configurations.

nonuniform traffic patterns, thus evaluation using other traffic patterns is essential. Figure 3c and 3f show the throughput and latency results for uniform, transpose, bit complement traffic patterns, respectively, using 16-ary MPSoC with 16-bit buffer size.

V. Conclusions

In this work, a framework for generating NoC-based MPSoC is introduced. The framework uses an open source RISC-V implementation having two levels of communication using an AXI4 interconnection bus and a highly configurable NoC system. A single building block of the MPSoC was synthesized for ASIC showing relatively low area and low power results. The performance results of various configurations were discussed.

References

[1] W. Wolf, A. A. Jerraya, and G. Martin, "Multiprocessor System-on-Chip (MPSoC) Technology," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 10, pp. 1701–1713, Oct 2008.

[2] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-Vencentelli, "Addressing the System-on-a-chip Interconnect Woes Through Communication-based Design," in *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC '01. New York, NY, USA: ACM, 2001, pp. 667–672.

[3] W. J. Dally and B. P. Towles, *Principles and practices of interconnection networks*. Morgan Kaufmann Publishers, 2004.

[4] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2011-62, May 2011. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2011/EECS-2011-62.html

[5] G. Sievers, J. Ax, N. Kucza, M. Flaßkamp, T. Jungeblut, W. Kelly, M. Porrmann, and U. Rückert, "Evaluation of interconnect fabrics for an embedded MPSoC in 28 nm FD-SOI," in *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2015, pp. 1925–1928.

[6] E. A. Carara, R. P. de Oliveira, N. L. V. Calazans, and F. G. Moraes, "HeMPS - a framework for NoC-based MPSoC generation," in *2009 IEEE International Symposium on Circuits and Systems*, May 2009, pp. 1345–1348.

[7] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip Generator," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17, Apr 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html

[8] "V-Scale," Web page. [Online]. Available: http://github.com/ucb-bar/vscale

[9] M. Gautschi, P. D. Schiavone, A. Traber, I. Loi, A. Pullini, D. Rossi, E. Flamand, F. K. Gürkaynak, and L. Benini, "Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 10, pp. 2700–2713, Oct 2017.

[10] D. U. Becker, "Efficient Microarchitecture for Network-on-Chip Routers," Ph.D. dissertation, Stanford University, August 2012. [Online]. Available: http://purl.stanford.edu/wr368td5072

[11] S. El-Ashry, H. Ibrahim, M. A. Ibrahem, M. Khamis, A. Shalaby, M. AbdElsalam, and M. W. El-Kharashi, "On Error Injection for NoC Platforms: A UVM-based Practical Case Study," in *Proceedings of the 10th International Workshop on Network on Chip Architectures*, ser. NoCArc'17. New York, NY, USA: ACM, 2017, pp. 2:1–2:6.

[12] A. S. Eissa, M. A. Ibrahem, M. A. Elmohr, Y. Zamzam, A. El-Yamany, S. El-Ashry, M. Khamis, and A. Shalaby, "A reusable verification environment for NoC platforms using UVM," in *IEEE EUROCON 2017 -17th International Conference on Smart Technologies*, July 2017, pp. 239–242.

[13] M. Khamis, M. Said, and Shalaby, "A Flexible Router Architecture for Three-Dimensional Network-on-Chips," in *Real-Time Systems Symposium (RTSS), 2017 IEEE*. RTSS'17, January 2018.

[14] B. Attia, W. Chouchene, A. Zitouni, and R. Tourki, "Network interface sharing for SoCs based NoC," in *2011 International Conference on Communications, Computing and Control Applications (CCCA)*, March 2011, pp. 1–6.

[15] A. Ferrante, S. Medardoni, and D. Bertozzi, "Network Interface Sharing Techniques for Area Optimized NoC Architectures," in *2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, Sept 2008, pp. 10–17.