NoC Emulation on FPGA: HW/SW Synergy for NoC Features Exploration

Nicolas Genko*, David Atienza*†, Giovanni De Micheli*
* LSI/EPFL, EPFL-IC-ISIM-LSI Station 14, 1015 Lausanne, Switzerland.
E-mail: {nicolas.genko, david.atienza, giovanni.demicheli}@epfl.ch
†DACYA/UCM, Avda. Complutense s/n, 28040 Madrid, Spain.

Abstract. Current Systems-On-Chip (SoC) execute applications that demand extensive parallel processing. Networks-On-Chip (NoC) provide a structured way of realizing interconnections on silicon, and obviate the limitations of bus-based solution. NoCs can have regular or ad hoc topologies, and functional validation is essential to assess their correctness and performance. In this paper, we present a flexible emulation environment implemented on an FPGA that is suitable to explore, evaluate and compare a wide range of NoC solutions with a very limited effort. We also present an automated way to perform NoC features exploration using the interaction HW/SW on an FPGA.

Our experimental results show a speed-up of four orders of magnitude with respect to cycle-accurate HDL simulation, while retaining cycle accuracy. With our emulation framework, designers can explore and optimize a various range of solutions, as well as characterize quickly performance figures.

1. Keywords

HW/SW Co-design, Interconnection mechanisms, embedded systems, Networks-on-Chip.

2. Introduction

With the growing complexity in consumer embedded products, new tendencies envisage heterogeneous *System-On-Chip* (SoC) architectures consisting of complex integrated components communicating with each other at very high-speed rates. Intercommunication requirements of SoCs made of hundreds of cores will not be feasible using a single shared bus or a hierarchy of buses due to their poor scalability with system size and their shared bandwidth between all the cores attached to them.

To overcome these problems of scalability and complexity, *Network-On-Chip* (NoC) has been proposed as a promising replacement for buses and dedicated interconnections [3,10]. NoCs involve the design of network interfaces to access the on-chip network and switches to provide the physical interconnection mechanisms to transport the data of the cores. Therefore, the definition and implementation of NoCs involve a complex design process, for instance, the selection of suitable protocols or topologies of switches to use.

Concrete options for NoC topologies and interfaces have been proposed at different levels of abstraction [15,11,12,4] and some even implemented onto FPGAs for functional validation. Nevertheless, these different physical implementations onto FPGAs are limited in flexibility and do not allow a full test of different actual realizations of NoC on silicon.

In this paper, we present some applications of a complete mixed HW-SW NoC emulation platform [1,2] where a wide range of NoC features can be easily instantiated and compared at the physical level. Such emulation is possible since our emulator takes a NoC without modifying the device under test. As a result, this emulation framework provides a consistent way to test the performance achieved by actual physical realizations of NoCs on silicon at a very high speed (16000 times faster than an HDL simulator). To this end, it is implemented onto an FPGA platform and supplies a wide

range of statistics for the different traffic patterns that can be generated in NoCs. In addition, our framework implementation is very modular and the statistics reports are easily extensible for further testing of concrete effects (e.g. saturations effects in parts of NoCs) on a particular NoC instantiation. In addition to the speed improvement, we implemented a framework, which instantiation flow is very easy and fast.

The remainder of the paper is organized as follows. In Section 3, we describe some related work. In Section 4, we draw an overview of the architecture of our emulation framework and flow seen in [1,2]. In Section 5, we show for the first time how the concept of the Emulation framework which we developed can be apply at different level of the NoC Design. In the Section 6 we present an algorithm which can run with our emulator and we provide an example of use. Finally, in Section 7 we draw our conclusions.

3. Related Work

In the last years, significant research has been done to evaluate the design and implementation features of NoC at its different levels of abstraction. To provide accurate functional validation (i.e. circuit level), several approaches have been implemented in FPGAs. In [12] and [4], NoCs with a mesh-based topology and packet-switching as communication mechanism have shown the effectiveness of NoC. Also, other NoC architectures (e.g. torus) and designs of switches/routers have been ported to FPGAs in order to validate their NoC features (e.g. packet sizes, switching-mode) based on additional HDL simulations [13,20,21,6]. These previous approaches can validate several NoC implementations features, but none of them is designed to exhaustively test the details of NoC topologies and traffics as ours.

To evaluate in detail different architectural alternatives reducing the cost of synthesizable NoC design, several cycle-accurate simulation environments have appeared. In [17], VHDL is employed to evaluate several features of virtual channels in mesh-based and hierarchical NoC topologies. In [15], XML and SystemC are used to specify the NoC components (e.g. routers, network interfaces) and to test mesh-based NoC design alternatives. The main difference with our approach is that their simulations have a much larger execution time compared to our physical NoC emulation environment.

To increase the simulation speed of cycle-accurate VHDL, several approaches have been proposed. [9] and [11] describe modeling environments for custom NoC topologies based on SystemC. [5] presents a mixed VHDL/SystemC implementation and simulation methodology using a template router to support several interconnection networks. In [7], a C++-based library of communication APIs is built on top of SystemC to explore NoCs topologies. Finally, [14] presents a fast transaction level modeling approach to explore bus-based communication architectures. While the previous approaches enable the fast exploration of the main features of NoC designs as our proposed emulation platform, their level of accuracy in the estimations and their simulation speed is more limited compared to our complete physical emulation of parameterizable NoCs.

Other proposed approaches improve the speed of cycle-accurate NoC simulations by using high-level abstraction languages, e.g. C or C++. [8] presents a C-based interconnection network simulator. Similarly, [19] proposes an event-driven C++ simulator. Also, [10] presents a NoC design methodology that uses a parameterizable NoC architecture executed in a high-level event simulator. At a higher-level of abstraction, several algorithms, analytical models and heuristics have been proposed to achieve very fast rough estimations of the cost of NoC topologies based on graphs representions [18,16]. Although these approaches attain high simulation speeds (sometimes close to real hardware), they cannot obtain detailed statistics of final physical implementation systems as our emulation framework does.

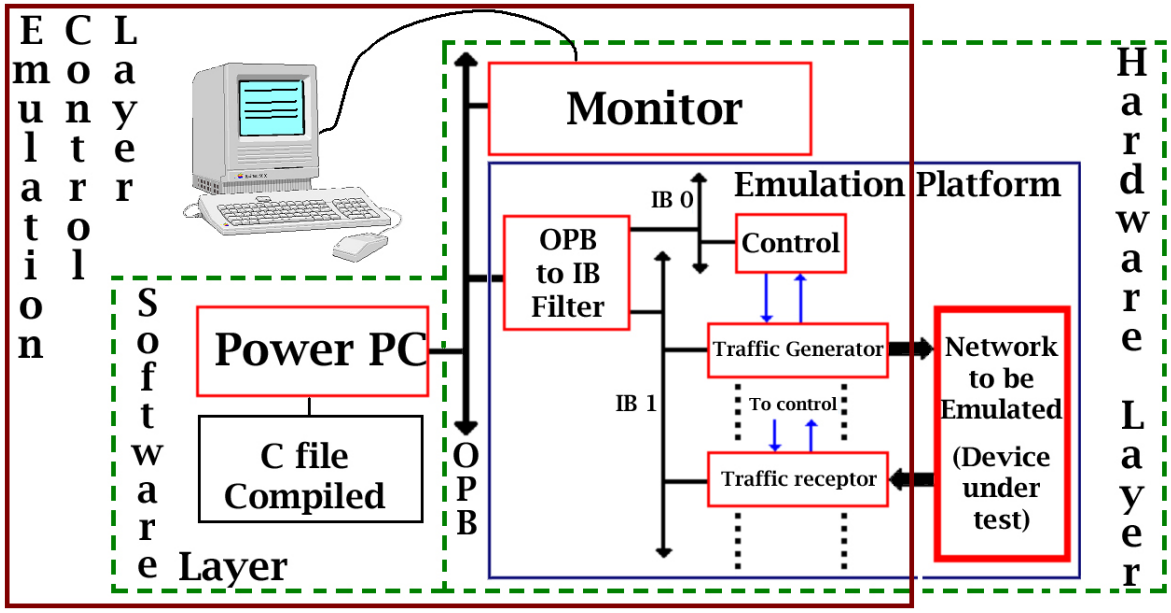

Figure 1. NoC Emulation Framework

4. Overall Emulation Framework

As previously mentioned in the introduction, our emulation approach has been designed in a modular way to easily implement various custom NoC topologies and architectures. An overview of the architecture of our framework is depicted in Figure 1. It consists of three main elements, which are mapped onto an FPGA board with a hard-core processor. In this case, we have used a Xilinx Virtex 2 Pro v20 and a Power PC. The hard-core processor of the FPGA is used to orchestrate the emulation process in a flexible way. Then, the monitor module provides the interface to communicate with the host PC and to show the produced statistics onto its screen through the serial port. Finally, the main element of our NoC emulation framework is the NoC programmable emulation platform. It is a module that consists of the necessary elements to emulate the device under test. Currently, the synthesizable NoC components are generated using the Xpipes compiler [9], but our proposed framework is directly applicable to any other type of NoC architecture. The previous modules communicate using a common bus available in our FPGA board called On-chip Peripheral Bus (i.e. OPB in Figure 1).

Our emulation platform (see Figure 1) consists of four types of components: a control module, several types of *Traffic Generators* (TGs) or *Traffic Receptors* (TRs) and a device under test. The control module and the TGs/TRs are fully addressable by the processor for configuration and statistics acquisition purposes. Also, the control component can communicate with all the components of the platform by sending broadcast control signals to all of them. Each TG generates different traffic (see Subsection 4.1) and injects the packets into the *Device Under Test* (DUT) through its dedicated connections. Then, after passing through the network of switches, the traffic is received and analyzed by a set of TRs (see Subsection 4.2). Finally, to enable an efficient scalability in the amount of TGs/TRs, we have included in the platform a set of independent busses to connect them. Hence, using our architecture it is possible to plug up to 1024 TR/TG, assuming that a larger number of traffic devices would not be fit on actual FPGAs. As a result, this emulation platform enables to

instantiate and to emulate real-life NoCs on current FPGAs.

In the following subsections we describe in detail the functionality of the main available components in our emulation platform (i.e. TGs/TRs and control module). We then detail the emulation flow of our system.

4.1. Traffic Generators

We define a Traffic Generator as a module, which is programmable by a processor and controllable by a control module. In order to make it programmable, a bench of registers is addressable by the processor in each TG. Some control signals are used to communicate with its control module. Finally, an interface is available to inject traffic into the DUT. As this component must be able to explore/analyze many characteristics of NoCs implementations, the traffic generated by each TG is a function of the content of its registers. This feature gives us the possibility to generate different types of traffic with a single type of TG, and as the configuration of the registers of the TGs is done by the processor, we do not need to resynthesize the platform to perform different emulations of NoC traffic.

In order to validate our approach, we developed two type of TG. First, it can generate stochastic traffic. Second, it can use input traffic traces generated by real-life applications, thus emulating the behavior of real workloads for the NoCs. Their implementation is well detailed in [1].

Note that both types of TG can be used at the same time by including a controller for each used type

4.2. Traffic Receptors

Similarly to the TGs, we have included two different implementations of TRs in our emulation platform. On the one hand, both have as common functionality the acknowledgment of the received traffic. In addition, both types enable two debug modes. In the first mode, it can perform an automatic check of the flits received via CRC check to guarantee that they are the correct ones sent by the TGs. In the second one, for manual checking, the content of the flits can be shown on the screen of the host PC to verify their content. The use of two different TRs support an efficient implementation according to the required type of reports to generate and a suitable debug tool for the network.

Also note that the two types of TRs provide different kind of statistics to the user. The first type generates a histogram about the number of acknowledged flits. The second type generates a trace report for each received packet. The trace has the same format as the one used by the TGs. By this way the processor can compute a detailed analysis (e.g. latency, arrival time) for each delivered packet.

4.3. Control Module

The control module is addressable by the processor and takes care of the synchronization of all traffic devices in the platform (i.e. TGs and TRs). For instance, it makes sure that all devices start the emulation at the same time. Also, the controller has the ability to reset the whole platform or even stop it. This is useful if the emulation platform needs to be programmed to execute several consecutive emulations.

4.4. Emulation Flow

The main feature of our emulation framework and its flow is the simple initialization and statistics acquisition of any circuit level emulation without re-synthesizing and remapping the whole system. This is possible thanks to its mixed HW-SW structure, which allows the processor to initialize some parameters in the hardware part of the platform. An overview of this emulation flows is shown in Figure 2.

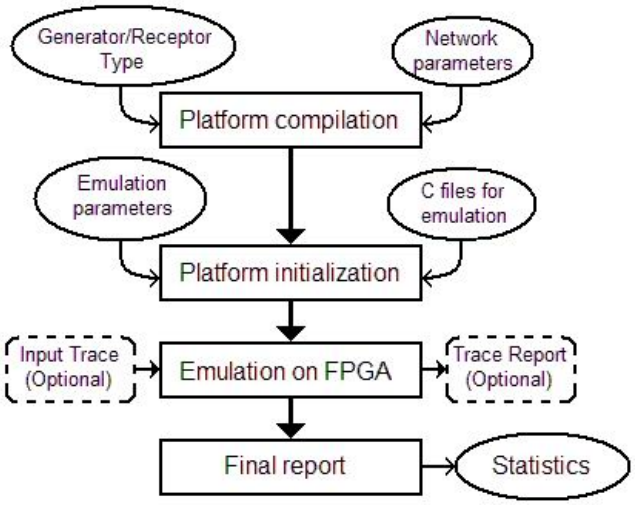

Figure 2. Our NoC Emulation Flow

As Figure 2 indicates, from the hardware point of view, thanks to the components explained in Section 4 we can emulate at the circuit level various switching configurations of a NoC. The precise configuration to use in our emulations is defined in the first phase (first square in Figure 2) by configuring the Verilog code of our platform, which is a matter of defining several parameters.

After that, in our flow the initialization traffic to generate is performed (second box in Figure 2) using the processor of the FPGA board (i.e. Power PC). It executes a file with C code that contains the software code to configure the system. This file contains information about the total emulation time, the sampling period for statistics generation, routing information, latencies to use between packets, flits per packet and stochastic traffic distribution. Thus, this enables a high flexibility because no time-consuming recompilation of the HW involved is needed to emulate and study a wide range of these parameters. Then, during the initialization phase, by reading and writing in the TGs/TRs, the processor transmits data to TGs/TRs and is configured to receive results of the specified analysis at the end of the simulation.

After that, the emulation works autonomously and the TGs/TRs acquire the information necessary to generate the statistics demanded by the user in its C configuration file.

Finally, at the end of the emulation, the stored statistics are sent back to the processor which displays a summary report about the behavior and congestion of the network on the screen of the user using the monitor module (see Section 4) and its serial interface to the host PC. Instead of displaying the statistics on the user PC, the user can also program the processor to analyze it and perform a succession of emulation. We will see in Section 6 how this analysis can be perform.

5. Applications

In this section, we show two implementations of our emulation platform. A first implementation were shown in [1]. In this first implementation, the device under test was a network of switches. We then wanted to extend it by an implementation of the emulation platform by an emulation of a full Network-on-Chip, which include in addition to switches the network interfaces (NIs). In the next subsection, we describe both implementations.

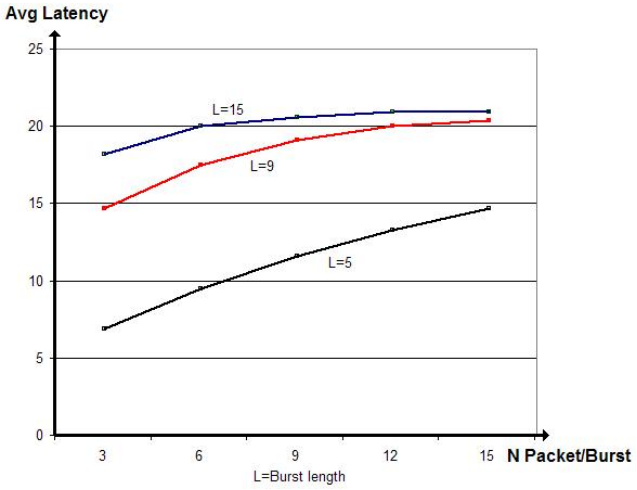

Figure 3. Emulation of a Network of Switches

5.1. Application 1: Emulation of a Network of Switches

The first implementation of our emulation framework emulates a Network of Switches. Those switches were generated by X-pipes compiler [9]. In the current version of the X-pipes compiler, the network protocol (i.e. the protocol used to communicate between switches and from NIs to switches) can be configured. There are currently 3 possibilities. The protocol, which we used, is called Ack/Nack protocol. This protocol includes a data link associated to a request wire, which indicates that the data link is valid. There are 3 additional wires, which acknowledge or not the request, and which indicate a repeated data in case of former non-acknowledgement. Having as a device under test a Network of Switches means that our TGs/TRs must have an interface compliant to this protocol.

In this implementation of the Emulation platform, we coded two kinds of TGs and two types of TRs. We can then describe those devices by showing two set of TG/TR. The first set is a stochastic TG associated to a TR, which measure the latency of packets and generate some histograms relative to the amount of traffic versus time. With a second set of TG/TR, instead of generating stochastic traffic, we generate traffic according to a trace. This implementation is more challenging since the trace (i.e. a collection of packet descriptors) must be sent to TGs in a continuous flow and must be decoded at run time by the TGs. Then, in this implementation, in addition to measuring the packet latency, we implemented a congestion counter, which measures the non-acknowledgement of flits thru the network of switches. The TRs also generate a trace according to the received traffic.

We show here an example of statistics collected with this emulation platform on Figure 3. We can see the average latency versus the number of packets per burst, with different burst length. This experiment shows how the latency of packets varies on a 2x3 mesh topology with a particular routing policy. The interesting result of this experiment is the speed to obtain such a plot. We have 15 points on this plot. Using traditional simulator such a result is much longer to obtain (see [1] for more detail).

5.2. Application 2: Emulation of a Network-on-Chip

In this paper, we introduce for the first time a second implementation of our emulation platform. After emulating a network of switches, we wanted to emulate a full NoC. Adding some NIs associated to the network of switches implies a major change in the design of TGs/TRs. Instead of having

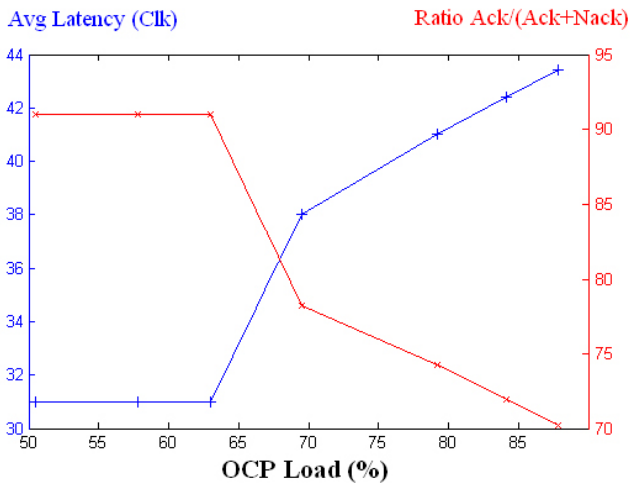

Figure 4. Emulation of a Network-on-Chip

modules, which generate or receive traffic, we must have devices, which emulate master or slave devices of a System on Chip. Those devices are controllable by the processor of the system like it was done in the previous design.

Unlike the previous design, we implemented in this case only a trace driven master programmable device. On the slave programmable side, we implemented a two state Markov chain, which turns on or off the device. We implemented this feature in order to be able to emulate the busyness of a slave device with a given probability. Note also that the protocol used is not the same as in the first application of our emulation platform since our Master and Slave modules communicates with NIs, which are OCP [22] compliant.

About generated statistics, the master device measures the average execution time of read and write operations. In the slave devices, we measure the average packet latency. We also added on the NoC links some sniffers, which measure the link activity. Note that our design is able to monitor the link activity without adding traffic on the network or without degrading the NoC performances. We insist on the fact that our emulation must not be intrusive in the design under test.

We are now showing now an experiment obtained with this emulation platform. It shows on Figure 4 a rising curve, which is the average latency and a decreasing plot, which is the ratio Ack/(Ack+Nack) on links of the NoC, versus the OCP load. To perform this experiment, we used a 2x2 mesh topology with attached to each switch, a couple of NIs, one with a master core, and one with a slave core. We then generated traffic over the NoC, avoiding deadlocks and using all the links of the NoC. the average presented on this plot shows the results for all the links together and all the traffic. The interesting point of this experiment is, in addition to the speed of the emulation, the fact that we can see here the percentage of the OCP traffic generated by each master core, which is acceptable by the NoC taken in consideration without altering its performances. In this example, we could measure that the topology under test can accept up to 63 % of OCP load. This result is not surprising since we are using in our NoC links between switches of 16 bits width since the number of wire in FPGA is limited. Thus, we could expect that this topology cannot ensure communication bandwidth at 100 %.

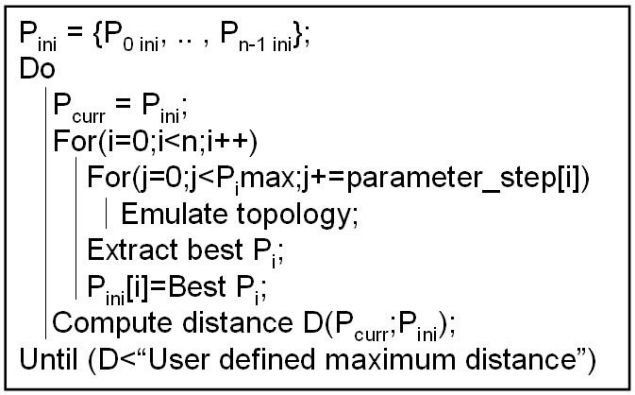

Figure 5. Algorithm for NoC Features Exploration

6. NoC features Exploration

For the first time, we introduce in this paper an application at the software level which extracts a best topology for a given NoC application. Our algorithm is presented on Figure 5. In this algorithm, we take a set of parameters of a NoC. All those parameters can be tuned by software interaction in our emulator. For each parameter, we define a set of values that it can take. If we want to emulate all topologies with all the parameters possibility, we reach a complexity which is not affordable, even with an emulator, which runs at 50 MHz. In order to reduce the complexity of the algorithm, we instantiated a 3 loops algorithm, which makes vary one parameter at one time. Once a parameter were tested under its range, we extract the best value.

6.1. Extract best value

To extract the best value of a parameter, we first must define what is the target of the tuning of parameters. Depending on what are the expected performances of the tested design, we are able to extract what is the best value for a given parameter. Note that in case several values are possible, the program will choose the parameter value, which is closer to the parameter value of the previous occurrence.

Once we tried to evaluate all the parameter, we obtain a new set of parameter. We compute the distance between the new set of parameters and the former set of parameters. If this distance is small enough, we can consider that we reach the best set of parameters possible.

6.2. Compute distance

To compute the distance between two set of parameters, we can use the typical distance definition by computing the sum of the squared differences. However, some parameters are not measurable like distances or weight. For example, if we want to evaluate the impact of the decision policy into switches, the distance between two policies is not measurable. In this case, the user must define a distance which can make sense.

6.3. Complexity

The complexity of this algorithm can be defined as follow: Number of Parameters x Parameters granularity x Number of loops to reach the user max distance. Even if this complexity is not as high as Parameter granularity to the Number of Parameters, this is still quite important. In order to execute that kind of algorithm, one must use an emulator like the one we developed. In the following

Simulation mode	Speed (cycles /sec)	Simulation time For 16 Mpackets	Simulation time For 1000 Mpackets
Verilog (ModelSim)	3.2K	13h53'	36 days 4h
SystemC (MPARM)	20K	2h13'	5 days 19h
Our Emulation	50M	3.2 sec	3'20''

Figure 6. Emulation vs. Simulation Speed

subsections, we show an example of use of our algorithm and we provide some key figures, which show the gain in time provided by our emulation platform.

6.4. Example

We can take as an example the definition and the tuning of NoC dynamic routing policies. We define a dynamic routing policy as follow: in NIs, if a path is congested, dynamically select another path. Then, as we have seen previously, we must also define a goal to achieve. Here, we decided to achieve an effective utilization of NoC links without performance degradation.

In this example, the number of parameter is in the order of magnitude of 10 and the number of configuration to test with our algorithm is in the order of magnitude of 1000. Using a reasonable benchmark of 1 million cycles, we reach the need to emulate one billion cycles.

In the next subsection, we investigate the speed of our emulator compared to traditional cycle accurate simulators.

6.5. Speed of Emulation

On the Figure 6, we compare the speed of traditional cycle accurate simulators with our emulation. First, we note that simulators are running in the two example that we show, at 3.2 to 20 kilo-cycles per seconds. In the case of our emulator, we are running at 50 MHz. It means that we are running at up to 4 order of magnitude faster than traditional simulators. Then, we have seen in the previous subsection that using complex algorithm to tune ad-hoc NoC; we need to run simulations/emulation for billions of cycles. Our emulator runs for one billion cycles in 3 minutes and 20 seconds while a fast cycle accurate simulator will take more than 5 days.

7. Conclusions and Future Work

New consumer products have increasingly higher demands and complex SoCs are used to implement such systems under the tight time-to-market constraints. NoCs solutions have been proposed to reduce the complexity of integrating tens of cores on-chip, but none of them allows complete architectural studies of different NoC realizations on silicon. In this paper, we have presented a flexible HW-SW emulation environment implemented on an FPGA that is suitable to explore, evaluate and compare at the physical level various custom NoC solutions for these new consumer systems with a very limited implementation effort. Moreover, as we have shown, a large set of important

implementation and design parameters for actual NoCs can be evaluated on this proposed emulation platform in a very short interval, thanks to its HW-SW framework design to avoid multiple hardware synthesis on the FPGA and its fast emulation speed.

Acknowledgements

This work is partially supported by the Spanish Government Research Grant TIC2002/0750 and a Mobility Post-Doc Grant from UCM for David Atienza.

References

[1] N. Genko, D. Atienza, et al. A Complete Network-on-Chip Emulation Framework. In *Proc. DATE*, March, 2005.
[2] N. Genko, D. Atienza, et al. A Novel Approach for Network-on-Chip Emulation. In *Proc. ISCAS*, June, 2005.
[3] L. Benini and G. De Micheli. Networks on chip: a new soc paradigm. *IEEE Computer*, January, 2002.
[4] G. Brebner and D. Levi. Networking on chip with platform fpgas. In *Proc. FPT*, 2003.
[5] J. Chan et al. Nocgen:a template based reuse methodology for NoC architecture. In *Proc. ICVLSI*, 2004.
[6] D. Ching, P. Schaumont, et al. Integrated modeling and generation of a reconfigurable NoC. In *Proc. IPDPS*, 2004.
[7] M. Coppola, et al. Occn: a NoC modeling and simulation framework. In *Proc. DATE*, 2004.
[8] W. Hang-Sheng, et al. Orion: a power-performance simulator for interconnect. networks. In *Proc. MICRO*, 2002.
[9] A. Jalabert, S. Murali, et al. xpipescompiler: A tool for instantiating application specific NoC. In *Proc. DATE*, 2004.
[10] S. Kolson, A. Jantsch, et al. A NoC architecture and design methodology. In *Proc. Annual Symp. VLSI*, 2002.
[11] J. Madsen, S. Mahadevan, et al. NoC modeling for system-level multiprocessor simulation. In *Proc. RTSS*, 2003.
[12] T. Marescaux, J.I. Mignolet, et al. NoC as hw components of an os for reconfigurable systems. In *Proc. FPL*, 2003.
[13] F. Moraes, et al. Hermes: an infrastructure for low area overhead packet-switch. NoC. *Integration-VLSI Journal*, 2004.
[14] S. Pasricha, et al. Fast exploration of bus-based communication architectures at the ccatb abstraction. In *DAC*, 2004.
[15] S. Pestana, E. Rijpkema, et al. Cost-performance trade-offs in NoC: a simulation-based approach. In *Proc. DATE*, 2004.
[16] A. Pinto, et al. Efficient synth. NoC. In *Proc. ICCD*, 2003.
[17] D. Siguenza-Tortosa et al. Vhdl-based simulation environment for proteo noc. In *Proc. HLDVT Workshop*, 2002.
[18] L. Tang and S. Kumar. Algorithms and tools for NoC based system design. In *Proc. SBCCI*, 2003.
[19] D. Wiklund, S. Sathe, et al. NoC simulations for benchmarking. In *Proc. IWSoC for Real-Time Apps.*, 2004.
[20] C. Zeferino, M. Kreutz, et al. Rasoc: a router soft-core for NoC. In *Proc. DATE*, 2004.
[21] C. Zeferino and A. Susin. Socin: a parametric and scalable NoC. In *Proc. SBCCI*, 2003.
[22] OCP International Partnership (OCP-IP). Open Core Protocol Standard. 2003. http://www.ocpip.org/home