NTGEN: a Network-on-Chip Traffic Generator toolkit for latency analysis

Ermis Papastefanakis[†‡], Laurent George[‡], Xiaoting Li[*], Ken Defossez[†]
[*]ECE Paris, 75015 Paris, France
[†]Thales Communications and Security, 92622 Gennevilliers, France
[‡]Université Paris-Est, LIGM / ESIEE, Champs sur Marne, France
Email: ermis.papastefanakis@thalesgroup.com, laurent.george@univ-mlv.fr, xiaoting.li@ece.fr, ken.defossez@thalesgroup.com

***Abstract*—Characterizing Networks-on-Chip (NoCs)-based Systems-on-Chip (SoCs) involves running many tests in software simulated as well as in hardware emulated environments. Tests help characterizing a platform and give metrics that can concern many different aspects. Each metric provides useful information for qualitative or quantitative conclusions. In this paper, we present a new tool called NTGEN that covers all the chain of actions for characterising latency on a Field Programmable Gate Array (FPGA) NoC-based platform. The toolkit, can be used for generating traffic scenarios that can be automatically launched. It helps manipulating as well as analysing the results in order to represent them into meaningful information.**

Keywords*—*Network-on-chip, toolkit, traffic generator, NTGEN, latency.

I. Introduction

Advances in semiconductor fabrication technologies allow chip manufacturers to include more processing cores in each new generation of products. Scalability has been an issue that needed to be addressed and the NoC paradigm was proposed as a solution to this problem [1]. It is now adopted in more and more designs [2],[3].

The effort to continue increasing performance in computer systems is leading to the adoption of highly parallel and heterogeneous platforms. Through parallelism we can distribute calculations to different Processing Elements (PEs) inside a manycore SoC and expect to obtain a high data flow rate. In addition, heterogeneous accelerators, adapted for handling specific workloads contribute to better exploit the parallel nature of such architectures.

The role of the interconnect in parallel platforms is vital in respect to their performance. As the number of PEs is steadily increasing the traditional interconnect technologies were not able to maintain scalability and started becoming a bottleneck. The NoC approach is based in the concept of taking mechanisms from computer networks and bringing them inside a SoC. NoCs suggest using a modular architecture of routers to handle the communication of PEs. This implies that in a NoC we find familiar concepts like routers and network interfaces transformed in respect to the constraints found inside a SoC ecosystem. We also find concepts such as routing algorithms, arbitration, flow control and buffering.

Initially NoCs were proposed as an alternative that would solve the scalability limits, however this quickly evolved. Researchers have been exploring NoC architectures to evaluate their properties and capabilities in other aspects that are important to SoCs. Some examples are reconfiguration, security, fault tolerance and determinism.

Since future generations of platforms for time-critical or safety critical tasks will be most likely NoCs-based, research on NoC architecture has produced concepts that are oriented to these domains. Through this exploration emerges the need for tools to validate models at a high level of detail and precision. Since such thorough validation is time consuming there is an interest for these tools to be optimised in order to perform those tasks as efficiently as possible and at the same time demand minimum human intervention and supervision.

In what concerns manycore architectures with real-time capabilities, sufficient testing needs to be performed in order to correlate theoretical results with the system model. Exhaustive tests might be necessary to cover all the potential scenarios of a use case while multiple use cases need to be considered in order to validate the system's timeliness.

Transaction-Level Modeling (TLM) allows to validate a model from a functionality perspective very fast and also provides insights in performance and timeliness as shown in [4]. However, Register Transfer Level (RTL) modeling provides more accurate results and is synthesizable which allows to also obtain realistic information on power consumption, chip surface, clock distribution etc. These advantages of RTL simulation come with the heavy cost of a rather high simulation time and high development effort to describe the hardware model. The first can be solved through emulation which allows to run a model at speeds that are three to four orders of magnitude higher than simulation. In this case the RTL model is created using a Hardware Description Language (HDL) and since emulation works on real hardware (FPGAs) it allows us to obtain a more realistic model. The downside with HDLs is the complexity in describing and debugging a model which in simulation can be more intuitive and flexible if there is a necessity to make changes.

Concerning benchmark tools, we can cite [5] where the authors present a Generic Mixed Criticality Benchmark (GMCB) providing task execution times, task criticality levels, communication patterns, and message sizes. In Mälardalen benchmarks [6], a benchmark is proposed to characterize the Worst-Case Execution Time (WCET) of applications. For timing analysis, the TACLebench [7] is a benchmark that can be used.

In the context of NoC, the MCSL benchmarks [8] can also be used to experiment classical signal processing ap-

plications including a H264 decoder, Fourier transforms, and Reed-Solomon encoders/decoders. With this benchmark, it is possible to define flow parameters (message size, paths) and the execution time of flows in the NoC. These applications define execution times. It is also possible to generate statistical traffic and use recorded traffic patterns. We are not aware of an open source toolkit such as the one presented in this paper

Our contribution: We propose a Noc Traffic GENerator (NTGEN) Verilog module that allows to produce flows (source-sink) in a NoC and a toolkit that handles this procedure by creating random test scenarios that cover the space we want to explore. The toolkit also allows us to automate the execution of all the tests, store and post-process the results. It finally also provides visualization of the results designed specifically to allow the user to add customized views for aspects of interest.

In section II, we describe the platform we use and provide information on its flow profiles. In section III we express the requirement for the toolkit and afterwards in section IV we proceed to detail the implementation. In section V we present the challenges we faced. Finally, we reach our conclusion in section VI and finish the paper by proposing our future work in section VII

II. NoC architecture

A. Platform

We consider a system based on a 4x4 2D mesh NoC where each node consists of a router, a Network Interface (NI) and an Intellectual Property (IP) element. Even though this is the platform that the toolkit was conceived for we considered that the effort to make it compatible with other platforms is small and was not be a limiting factor.

Each **router** R_{xy} possesses five links, four located at the edges **N**orth, **E**ast, **W**est, **S**outh (NEWS), used to connect with neighbor routers and the fifth is used to connect with the **L**ocal (L) IP$_{xy}$. At every input there is a buffer with the capacity of containing four **fl**ow control dig**its** (flits). A crossbar along with an arbiter are handling packet transmission (XY dimension routing) and flow control (Stop&Go). Virtual Channels (VCs) are not implemented and this means that a packet cannot bypass another packet that is already in an input buffer.

An illustration of a router R_{xy} is given in Figure 1.

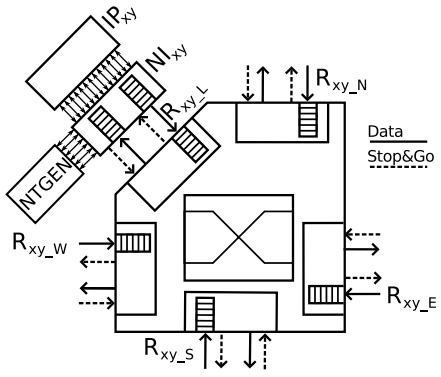

Fig. 1. Architecture of a NoC router R_{xy}

The **NI** is in charge of serializing and de-serializing packets. Packets are then split into smaller size flits in order to travel in the NoC. When an IP element makes a request for a memory location, the NI will encapsulate that into a packet, split it into flits and send them one by one to the appropriate router. When they reach their destination, the local NI will reassemble the packet, de-serialize it and forward it to the IP element that will handle the request. The same applies for the response.

A packet containing a memory response can take up 64 bytes of data and is split in 8 flits of 8 bytes each. The NI will add a header flit containing routing information making a total of 9 flits. A packet containing a request can be as small as one flit.

The **IP** element is the end point in the NoC. It can be a PE, memory, General Purpose Processor (GPP) or a peripheral (slave or a master). Consequently not all nodes in a NoC can initiate traffic.

This platform is implemented in Verilog and is able to synthesize on a FPGA (Xilinx Virtex-7). Measurements can be taken through a cycle accurate simulator (Xilinx Vivado) or through traces of the FPGA output stream passing through gigabit ethernet.

B. Network model

We consider n periodic flows transmitted in the NoC. A periodic flow τ_i sends packets respecting two parameters: 1) the period T_i which is the temporal interval between the arrival of two consecutive packets, and 2) the maximum transmission time C_i which is the maximum time to transmit all the flits of a packet on a router. In addition, the Average-Case Traversal Time (ACTT) and Worst-Case Traversal Time (WCTT) represent the average and worst-case time it took packets to traverse their path.

Due to the dimension-order X-Y routing, each packet of flow τ_i follows a static path denoted $\mathcal{P}_i$ which is composed of the source and destination IPs as well as the input ports of routers along this path. The first buffer of the source IP is denoted $first_i$, while the last buffer of the destination IP is denoted $last_i$. Then the path of flow τ_i is represented by $\mathcal{P}_i = \{first_i, ..., last_i\}$.

We consider one diffusion path in the network which means that when packets of different flows join one path, they do not leave this path until they are transmitted to the same destination (source-sink model). A real use case that illustrates this concept can be found in memory hierarchies where the last level, a common bottleneck in Multi-Processor Systems-on-Chip (MPSoCs), is the Random Access Memory (RAM). This assumption also comes from Avionics Full DupleX switched Ethernet (AFDX) network flows and is related to our previous work in [9].

III. Toolkit requirements

In order to be able to characterize the platform mentioned above and thoroughly validate new features we proceed by defining the requirements of the toolkit.

We took as input two use cases: a comparison between two arbitration schemes (First-in First-out (FIFO) and Round-Robin) as well as in validating our past work in [9]. Although both use cases were used to define requirements, in this paper we present how the toolkit was used to develop measurements

and visualization concerning the first use case of arbitration comparison. The context of this paper being to present the toolkit, we do not present the results of the use case here.

In Figure 2, we consider the NoC described in section II with 7 flows $\tau_1 \dots \tau_7$ reaching one destination and where each IP is indexed with the coordinates of its router. The solid lines represent the paths that join to reach the destination node. We focus on flow τ_1 following the path $\mathcal{P}_1 = \{\text{IP}_{00}, R_{00\_L}, R_{01\_W}, R_{11\_N}, R_{21\_N}, \text{IP}_{21}\}$. The paths of the other flows are:

$$\begin{aligned}
\mathcal{P}_2 &= \{\text{IP}_{03}, R_{03_L}, R_{02_E}, R_{01_E}, R_{11_N}, R_{21_N}, \text{IP}_{21}\} \\
\mathcal{P}_3 &= \{\text{IP}_{20}, R_{20_L}, R_{21_W}, \text{IP}_{21}\} \\
\mathcal{P}_4 &= \{\text{IP}_{22}, R_{22_L}, R_{21_E}, \text{IP}_{21}\} \\
\mathcal{P}_5 &= \{\text{IP}_{23}, R_{23_L}, R_{22_E}, R_{21_E}, \text{IP}_{21}\} \\
\mathcal{P}_6 &= \{\text{IP}_{30}, R_{30_L}, R_{31_W}, R_{21_S}, \text{IP}_{21}\} \\
\mathcal{P}_7 &= \{\text{IP}_{32}, R_{32_L}, R_{31_E}, R_{21_S}, \text{IP}_{21}\}
\end{aligned}$$

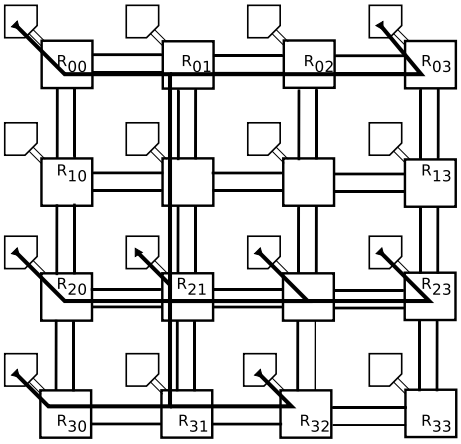

Fig. 2. NoC example of the case study

The FPGA used for the measurements already has an interface that allows us to send commands and recover traces using the gigabit ethernet port. Basic error detection is also implemented allowing to react when frames are lost. An NTGEN instance at each IP element should be addressable and modeled in a way that would make it possible to receive commands at runtime. In addition all NTGEN modules can be synchronized and start their transmission simultaneously.

Concerning the scenario generation, we wanted to be able to specify the desired average link utilization of the output node (here the link R_{21} to IP_{21}) as well as the number of traffic sources and obtain random periods to create such a pattern. Automated deployment is available based on the list of scenarios generated with minimum supervision. This means that reinitialising the NoC at the beginning of each test is also necessary. Assuming results are obtained we are able to process them and filter the information that is necessary. Eventually for the visualisation phase we exploit a framework allowing to plot the results.

During all this process we also have non-functional requirements such as performance and resource usage that should be contained in order to make this toolkit compatible with an average computer. In addition the development languages should be chosen based on their capacity to provide reusability, readability and ease of maintenance. In the same context, standard data formats should be chosen avoiding customized structures that would pose a limit to compatibility. Finally the toolkit should have a modular architecture that would make it portable to other NoC models and would allow anyone to replace or add modules to improve its functionality.

IV. Toolkit implementation

Following the requirements specified we propose a toolkit described in Figure 3. On the left we have the FPGA which is connected through an Ethernet cable to a host computer on the right. On the FPGA we have the synthesized bitstream of the NoC architecture along with NTGEN which is present in each node. On the host side we have all the software elements that the toolkit consists of.

A. Traffic Generator Implementation

NTGEN, as illustrated in Figure 1 is connected to the NI similarly to an IP element. Once configured, NTGEN can inject 9-flit long packets to the NI destined to any of the other nodes in the network. The corresponding NTGEN in the destination node will receive the packet and send a message through the Ethernet interface to the host detailing the packet's transmission. The length of the packet was intentionally chosen to represent a cache line transfer. However, if supported by the NoC model longer packets can be generated in order to represent for example Direct Memory Access (DMA) modules. This would reduce the number of messages sent to the host as we would have a lower packet per flit ratio. When synthesized, NTGEN takes around 200 Look Up Tables (LUTs) which makes it light and capable to scale as the number of nodes might increase.

B. Scenario generation tools

For the host side of the toolkit, the starting point is the random generation of periods [10] for each of the source nodes in a scenario. The input information is the number of source nodes, the desired utilization and the total number of random scenarios that are going to be generated. The algorithm in [10] is called to generate a set of periods and also calculate the necessary amount of cycles (one hyperperiod equal to the least common multiple of flow periods) that each scenario needs to run. This file is then taken by a script that aggregates multiple lists of different utilizations, source and destination nodes based on user input provided to the script. The final file generated is passed to the command interface that can interpret it and launch each of the scenarios sequentially. It will configure NTGEN in each of the source nodes using a library that transforms each line of the list to commands for the FPGA. This tool also manages the initialization of the FPGA before each scenario, the launch of each scenario as well as its termination when the hyperperiod is reached.

At the same time, the response interface (the counter-part of the control interface) receives traces from the FPGA and performs a preliminary post-processing that allows to filter a big part of the information and keep the necessary parts that are then saved in a file. This file is taken by the visualisation tool whose goal is to transform the raw information into objects. Then we can easily exploit them to produce visualization

graphs that can then be used to obtain an insight on the temporal behavior of the model.

We can understand that the user is required to interact with the toolkit mainly at the early stages as we tried to keep the configuration parameters simple. If necessary, we can obtain a finer configuration granularity by intervening inside the scripts.

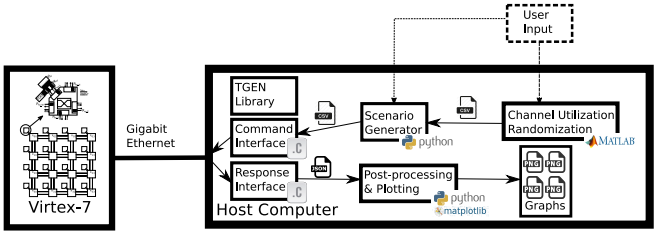

Fig. 3. Toolkit software architecture

C. FPGA interfacing

Concerning the command and response mechanisms that manage the tests on the FPGA and store the results, we can see a top-level state diagram in Figure 4. Both interfaces are launched and reach an idle state. The user launches the command to execute test scenarios and the command interface resets and configures NTGEN. The NoC model possesses a reset mechanism that enables the command interface to put it in an initial state that is consistent and allows us to perform each test with fixed initial conditions. After NTGEN is configured for the first scenario the command interface passes at a "wait state" during which it is polling a pipe that is established for communication between the two interfaces. When the scenario is over, the response interface will send a message in that pipe and the command interface will reset the FPGA and configure NTGEN to continue with the next scenario. In this current version of the toolkit, managing errors is not supported, in the case that a packet is lost, the response interface will send a message in the pipe and both programs will exit.

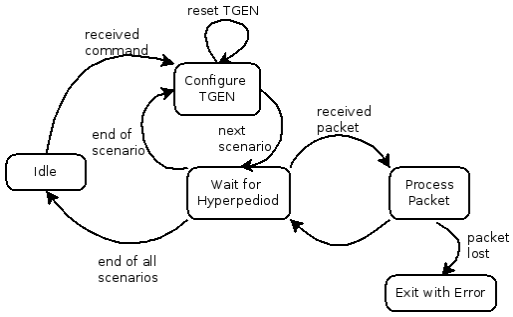

Fig. 4. Toolkit state machine

D. Visualisation

When results have been stored, we can use the visualisation script in order to plot the information into graphics (using matplotlib library). That way, we can get a better understanding on the behavior of the model in addition to having raw information.

The plotter of the toolkit provides mainly pre-coded graphics that: will automatically be plotted based on the results (see below) and allow the verification of the precision of the tested scenarios. The capability for the user to code his/her own graphs that can visualize additional aspects of the results.

In figures 5, 6, 7, 8, 9, 10, we present some of the graphs that are provided by default in the toolkit.

In figure 5 we can see all the scenarios executed and the worst case latency in each. In figures 6, 7 we have all the scenarios grouped in relation to the link utilization at the destination node. In figure 6 we see the average time it takes packets to traverse the NoC. In figure 7 we have the worst case of the worst traversal time of the scenarios with the same link utilization. This will allow us to perform side to side comparison between two features, being arbitration, routing etc. In figure 8 we obtain the histogram for a specific node so that we can observe its behavior. Finally in figures 9, 10 we have information on the precision of the tests we performed. This can be used to easily to verify that the traffic generation does indeed produce correct utilization patterns.

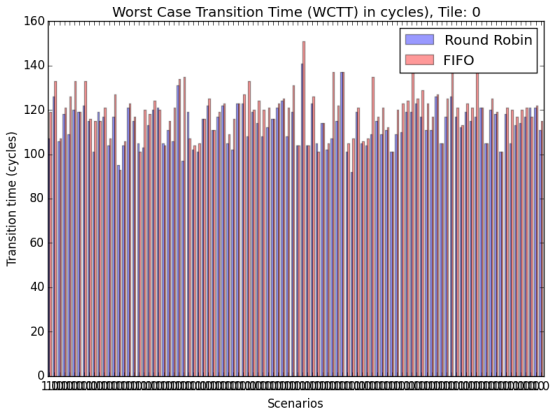

Fig. 5. Overview of all scenarios

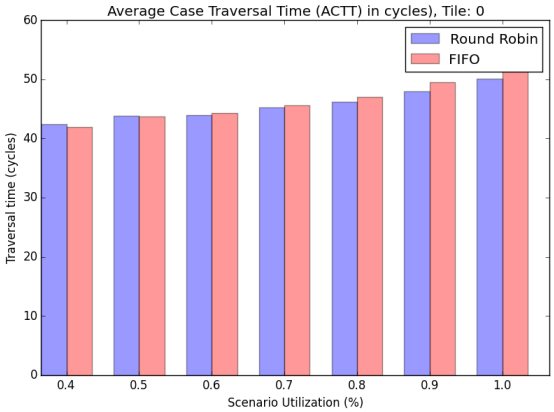

Fig. 6. Bar graph for the ACTT

V. Challenges

A. Implementation of HDL traffic generator model

HDL is quite different from software development as it requires to keep the model synthesizable and to always take into account the way the synthesis tool generates the

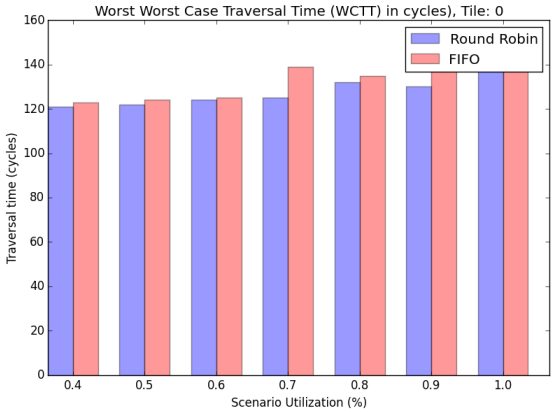

Fig. 7. Bar graph for the WCTT

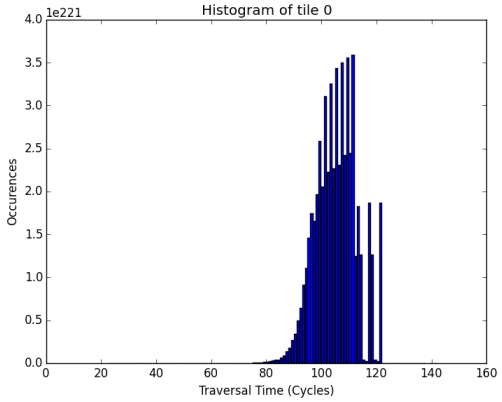

Fig. 8. Distribution of Traversal Time of node IP_{00}

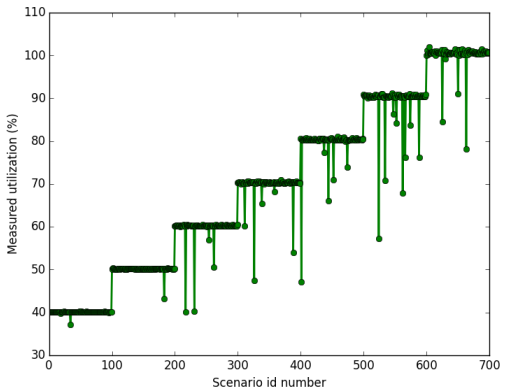

Fig. 9. Precision validation between theoretical and measured values for each scenario

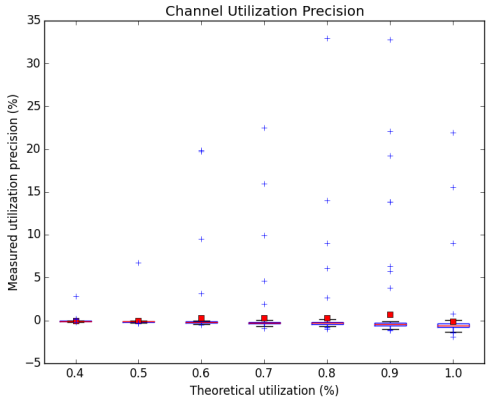

Fig. 10. Relative precision validation for each utilization set of scenarios

bitstream based on the code. For example, a modulo counter implemented using the % symbol (usual in languages such as C) will use much more resources compared to implementing the module through a register. Validation can be challenging as we can choose between two worlds, cycle accurate simulation or emulation on FPGA, each having its own advantages. With simulation we can have a very high degree of visibility of the model's behavior, monitor or alter any part of it and pause the simulation at any point. The downside is that simulation is very slow and can become inefficient in long scenarios. Emulation on FPGA can be significantly faster as the model is running in real-time. However, synthesizing the bit-stream can take long (30-120 minutes) and needs to be repeated at every modification. As a result emulation is more suitable for long scenario validation while simulation is very efficient in short fine grained validation as well as for the development/debugging phase.

B. *Implementation of scenario generator*

The challenges in scenario generation relate to producing enough scenarios to cover all the channel utilization needed for testing during a sufficient duration (at least one hyperperiod). At first we needed to produce random scenarios with a specific bandwidth output. This was achieved by using Randfixedsum algorithm [10] to generate each source's transmission period. The goal was to obtain periods that would cause the link at the destination node to have a desired average utilization rate. In addition, the total duration of each scenario, defined by its hyperperiod should not be too large to test it. As a result from all the scenarios generated, we only keep the ones that do not exceed 1 billion cycles.

C. *Implementation of communication software with the FPGA*

The main challenge here relates to performance and more specifically to the reception of packets from the FPGA. The objective is to be able to process and store packets fast enough to avoid dropping them due to congestion. Depending on the channel utilization, the FPGA would output packets to the host computer at different rates. For higher utilization scenarios the bandwidth would reach as much as 600 MBit/s and storing this information would generate files that as much as 8GBytes per scenario. Considering that we would need to execute multiple

tests of hundreds of scenarios, archiving the information would limit the capacities in testing. As a result there would have to be some processing at the reception of each packet in order to keep the useful parts of the information and reduce storage requirements.

A new challenge emerges here as the amount of processing resources is limited by the packet reception rate. The processing time should not exceed the time between packets as it will end up filling the input buffer to a point that packets will eventually be dropped. To resolve this issue various mechanisms were implemented. Firstly the CPU/memory trade-off was shifted and the memory used for the input buffer was maximised in order to be able to absorb more packet pressure giving the possibility to the CPU to complete the processing. Secondly, the processing task was attached to a specific core; avoiding migrations to other cores would optimize the resource usage. Finally, to ensure that there would be no other tasks sharing the core with the processing task, creating context switching and preemption overhead, we masked the core making it solely available to the toolkit.

Through this challenge we are able to identify that the major factor of scalability for the toolkit is the destination node. Since messages to the host computer are sent each time a packet is received, it is obvious that scaling to more cores or adding virtual channels will not affect the volume of data we receive at each scenario. However adding more destination nodes will have an impact, but again only in high bandwidth scenarios. At this case slowing the operational frequency of the FPGA is a rather feasible solution that will add more emulation time for intense scenarios but remain more efficient than simulation and keep the tool scalable. Another possibility is to offload part of the post-processing to the FPGA and receive information that, requiring little or no processing, is ready to be stored and exploited. This solution needs to be evaluating its complexity in implementation and maintenance. However, given the gain of processing in FPGAs it can have a very high potential.

Secondary challenges involve the FPGA communication protocol stack that we implemented in order to provide a layers of abstraction, making the toolkit able to be adapted to other platforms.

D. Implementation of data analysis and representation

The JSON library in Python opens data files and parses all the content into memory in a single operation. This results in the creation of multiple objects holding information in RAM that is not immediately necessary but still occupies memory space. In fact for a rather small amount of results the memory occupied by object was so big that made us realize that the analysis would be impossible for the full scenarios list. This was another reason that made processing necessary before storing the data to disk. Currently the scenario file size after partially processing results, allows Python's JSON parser to cope without problems. However, in order to anticipate for future scenarios yielding more voluminous results a more solid solution needs to be implemented.

VI. Conclusion

In this paper we propose a new toolkit called NTGEN for a FPGA NoCs platform performance analysis. This toolkit can be used to perform tests to validate platform features and functionalities and characterize the latency of flows sent through the NoC. NTGEN can automatically generate traffic scenarios and perform analysis to present them in useful valuable information.

VII. Future work

At first in order to take an orientation for a more mature simple set of tools, we envision to converge the different programming languages and file formats. Secondly, we plan to enable support for traffic generation and visualization for more than one destinations. This way the toolkit will be able to handle more use cases and take a more general character. Thirdly, there is an interest to be able to generate realistic traffic patterns in addition to random ones. We intend to look into this subject so that for example, we can record real applications and replay them to simulate traffic.

Another interesting aspect would be to support results acquired from simulation in addition to emulation. The effort is minimal and it will allow to use the toolkit for small scenarios that still need visualization.

Finally, in the long term we would like to provide a Graphical User Interface (GUI) making the toolkit easier to use. Scenario generation will be easier, as well as following the progress of testing scenarios. In addition, by being able to chose subsets of data through a user interface will make managing the visualisation of the results much faster.

References

[1] W. J. Dally and B. Towles, "Route packets, not wires: On-chip interconnection networks," in *Design Automation Conference, 2001. Proceedings*. IEEE, 2001, pp. 684–689.

[2] B. D. de Dinechin, R. Ayrignac, P.-E. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss *et al.*, "A clustered manycore processor architecture for embedded and accelerated applications." in *HPEC*, 2013, pp. 1–6.

[3] A. Varghese, B. Edwards, G. Mitra, and A. P. Rendell, "Programming the adapteva epiphany 64-core network-on-chip coprocessor," *International Journal of High Performance Computing Applications*, p. 1094342015599238, 2015.

[4] L. S. Indrusiak, J. Harbin, and O. M. Dos Santos, "Fast simulation of networks-on-chip with priority-preemptive arbitration," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 20, no. 4, p. 56, 2015.

[5] J. Harbin, T. Fleming, L. S. Indrusiak, and A. Burns, "Gmcb: An industrial benchmark for use in real-time mixed-criticality networks-on-chip," in *Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[6] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The mälardalen wcet benchmarks: Past, present and future," in *OASIcs-OpenAccess Series in Informatics*, vol. 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010.

[7] "Taclebench website," http://www.tacle.eu/index.php/activities/taclebench, 2016, [Online; accessed 26-May-2016].

[8] W. Liu, J. Xu, X. Wu, Y. Ye, X. Wang, W. Zhang, M. Nikdast, and Z. Wang, "A noc traffic suite based on real applications," in *VLSI (ISVLSI), 2011 IEEE Computer Society Annual Symposium on*. IEEE, 2011, pp. 66–71.

[9] E. Papastefanakis, X. Li, and L. George, "Deterministic scheduling in network-on-chip using the trajectory approach," in *Real-Time Distributed Computing (ISORC), 2015 IEEE 18th International Symposium on*. IEEE, 2015, pp. 60–65.

[10] P. Emberson, R. Stafford, and R. I. Davis, "Techniques for the synthesis of multiprocessor tasksets," in *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, 2010, pp. 6–11.