

# Performance Analysis of Networks on Chip Using Coloured Petri Nets

J. Schleifer, H. Blume, T. G. Noll

Chair for Electrical Engineering and Computer Systems

RWTH Aachen University

Schinkelstraße 2, 52062 Aachen, Germany

Phone: +49 (0)214 80 97614, Fax: +49 (0)241 80 92282

Email: [schleifer@eecs.rwth-aachen.de](mailto:schleifer@eecs.rwth-aachen.de), [blume@ims.uni-hannover.de](mailto:blume@ims.uni-hannover.de),  
[tgn@eecs.rwth-aachen.de](mailto:tgn@eecs.rwth-aachen.de)

**Abstract**—Large multi-core systems require complex networks on chip (NoC) as communication infrastructure. For designing those NoCs efficient performance analysis techniques are necessary to reduce the design space as early as possible.

An attractive choice balancing model complexity and quality of the accomplished results is Petri Net based modelling, facilitating the description of concurrent processes in a fashion derived from automata theory. Coloured Petri Net (CPN) modelling integrates data structures and allows for reducing model complexity and thus simulation time. Due to the exponential increase of simulation time for large NoCs associated with other Petri Net based methods this enables simulation of significantly larger NoCs.

To support a modular approach to NoC modelling a component library was devised in course of this work. This library consists of models of NoC components, such as routing switches, and traffic generators which support both statistical traffic models and scenarios described by task graphs. Traffic generation can be controlled at each individual network node.

Easy access to information regarding latency and load generated at the network nodes enables designers to monitor traffic parameters and identify hotspots. Thus, it is possible to compare NoC performance regarding different traffic situations as well as changes in the network architecture such as employing different topologies and/or routing algorithms.

Focussing on evaluating network traffic with a very abstract model, only basic parameters are specified. Contrary to other techniques like SystemC or VHDL models our approach allows designers to concentrate on analyzing network traffic without having to implement a detailed model of the architecture. A comparison of CPN-based simulation and emulation of a detailed VHDL model of the same system results in estimation errors as small as only a few percent.

CPN-based modelling of NoCs therefore shows to be an efficient performance analysis technique, combining sufficiently high accuracy with small modelling and simulation effort.

**Index Terms**—Network on Chip, Coloured Petri Nets, Performance Modelling

## I. INTRODUCTION

The progress of modern microelectronics leads to a steady increase of chip complexity. This in turn enables designers to integrate complete systems on a single die as Systems on Chip (SoC). Those SoCs consist of a multitude of functional units such as processor cores, DSPs, memories or embedded FPGAs. With an increasing demand for computing power and continuous shrinking of the feature size the number of functional units contained in a single SoC grows steadily.

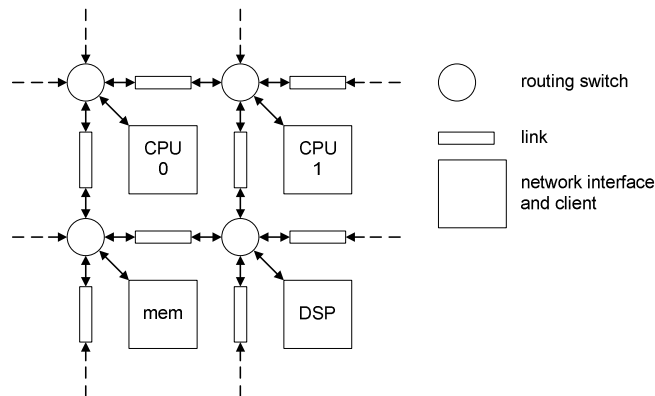


Fig. 1. Partial Network on Chip with attached clients (mesh topology)

As the number of cores increases communication between the different functional units becomes a major problem in realizing large SoCs. Conventional busses or point-to-point connections are not scalable to meet the demands of large SoCs due to problems like increasing load capacitance and lack of parallelism. A widely discussed communication architecture addressing these problems is Network-on-Chip (NoC). NoCs are derived from concepts of computer networking and supply a dedicated modular parallel communication infrastructure for SoCs (Fig. 1).

Generally NoCs consist of three components:

- network interfaces connecting clients to the

network,

- routing switches that route traffic through the network and
- links, by which the data is transported.

Due to this modularity NoCs are both flexible and scalable, thus eliminating the drawbacks of conventional communication concepts. The flexibility of NoCs leads to a vast set of parameters describing each individual NoC. These include but are not limited to network topology, switching method, routing algorithm and arbitration scheme. The consequence of this is a huge design space that needs to be explored in the course of implementing a NoC [1]. Several methods to model on-chip communication have been proposed in the past, including SystemC based simulation [2], FPGA based emulation [3], combined analytical and simulative approaches [4] and stochastic models, for example employing Markov Models [5] or Deterministic and Stochastic Petri Nets [6].

All of these approaches feature individual advantages and disadvantages. The FPGA- and SystemC-based methods for example require an elaborate description of the NoC which on one hand yields high accuracy but on the other makes it less suited for early design stages. The goal of the CPN based method proposed in this article is to provide an abstract NoC model including only the most important design parameters and thus being well suited for performance analysis in an early stage of the design process.

The article organized as follows. The following section is an introduction to Coloured Petri Nets, the third deals with the NoC component models while results are presented in the next one. The last section contains a summary and short discussion of the achieved results.

## II. COLOURED PETRI NETS

### A. Petri Nets

Petri Nets are a modelling formalism extending automata graphs to facilitate modelling of concurrent processes.

Petri Nets are comprised of several basic components:

- places (depicted as circles or ellipses),
- transitions (shown as bars or rectangles) and
- arcs.

Places usually represent states of a system component, for example the current state of a data source could be represented by places called *idle* or *active*. Places can be occupied by tokens (shown as dots), in the aforementioned example a token in the place *idle* would mark the source as being in the corresponding state. Places are connected by arcs and transitions with each arc connecting a place and a transition. Arcs can either be input (marked by an arrowhead towards the transition) or output arcs (arrowhead towards place). Depending on the arc direction transitions either remove tokens from a place or generate tokens in a place upon activation – called firing. A transition can only fire if each of the places connected by input arcs contains an appropriate

number of tokens denoted by a number on the corresponding arc. In the example shown in Fig. 2 the transition T1 removes two tokens from place P1 and a single token from P2 and then generates a token in P3. If there were no tokens in P2 or less than two in P1, the transition would not fire.

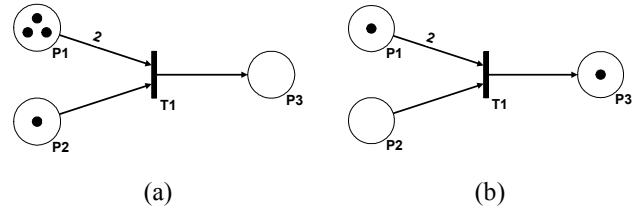


Fig. 2. Petri Net before (a) and after (b) firing of transition T1

Each place can be connected to an arbitrary number of transitions and vice versa. Furthermore the total number of tokens contained in a Petri Net is not limited. Petri Nets thus facilitate modelling of concurrent processes.

### B. Coloured Petri Nets

So-called Coloured Petri Nets (CPN) [7] expand these modelling features further: While tokens in a (classical) Petri Net do not possess any property and thus can be understood as representing a single bit, tokens in a CPN can contain data. Each place is assigned a data structure called *colourset* and can only contain tokens with data matching the constraints of this colourset. These coloursets range from simple data types such as integers to compound data types containing several less complex data types. Examples are

```
colset control: with data | request | ack | fail;
colset packet: int, control;
```

The colourset *control* defines a designator to identify different types of packets (*data*, *request*, *ack* and *fail*). The second colourset *packet* consists of two parts, an *integer* type data part and a *control* part containing the packet type designator.

Token data can be accessed and modified by transitions to define complex behavioural models. In the context of NoC modelling a transition could represent reception of a routing request at a client (Fig. 3): Transition T1 removes an incoming token ( $x$ ) containing the request designator and destination address from place *in*. The transfer function (below T1) then compares the address contained in the incoming token with the local address stored in place *add*. If the addresses match (the request reached the correct client) an *ack* message ( $y$ ) is generated at place *out* or a *fail* message otherwise, the *content*

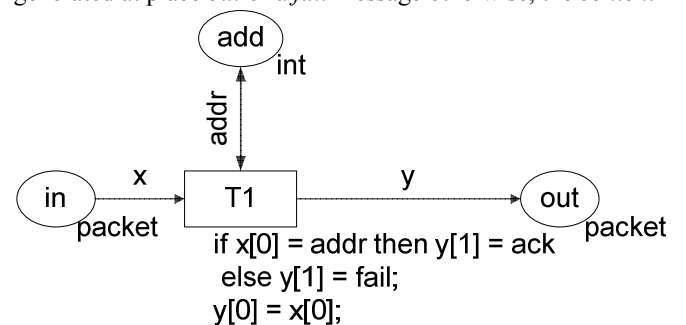


Fig. 3. CPN modelling reception of a routing request

remains unchanged. Since the messages to be handled are tokens of the colourset *packet* (see example above) this colourset is assigned to places *in* and *out* while the address is a simple integer. The different parts of the *packet* colourset are chosen by their respective position in the colourset,  $x[1]$  defining the *control* portion of  $x$  for example.

It is furthermore possible to create timed coloursets – tokens with such coloursets assigned to them include time stamps. Transitions accordingly can be assigned a processing time. Upon firing a transition the processing time is added to the time stamp of the input tokens to calculate the time stamp for the output tokens. If multiple transitions are ready to fire at the same time, they fire in the order of the time stamps.

Evaluation of CPN models is typically done by simulation. It is theoretically possible to convert a CPN into an equivalent (classical) Petri Net. This would provide the advantage of being able to find an analytical solution to the set of equations describing this underlying Petri Net. Due to the complex data structures this set of equations would be very large and thus require considerable memory and computing power to solve.

Simulation of CPN models usually yields statistical characteristics of several parameters. These parameters are

- transition throughput
- place occupation (type and number of tokens in a place) and
- token data.

By recording simulation steps it is also possible to monitor the model at any given moment. Since the relevant measures to be regarded in the NoC model can be calculated with just the statistics this possibility is not regarded any further.

The software used here (CPN Tools - Version 2.2.0, [8]) further provides means of conditional data collection. By monitoring CPN parameters only if a certain condition is met (for example a place having to be occupied), allowing complex measurements. Furthermore it features hierarchical CPN modelling.

### III. NOC MODELLING LIBRARY

In order to facilitate modelling of large NoCs a modular approach was taken making use of hierarchical CPN models. The basic NoC components are modelled as clients representing both the network interfaces and attached functional units and routing switches. On the top level of the NoC model clients and routing switches are shown as transitions. Links are represented as places connecting these transitions. The lower hierarchy levels show the components themselves to be split up into smaller units as described below. This allows easy reconfiguration by changing parameters in only a single module instead of needing to step through the model on the topmost hierarchy level. The same is true for changes by replacing a single module by another for example exchanging the routing algorithm.

Instead of modelling the NoC in great detail the library components are kept as abstract as possible without sacrificing too much accuracy. Therefore some design parameters are

omitted. These are most prominently clock frequency, which is replaced by abstract clock cycles, and wordlength – each message is modelled as a single entity defined only by its length measured in clock cycles.

#### A. Communication Protocol

The switching scheme used in this model is line switching. Communication between two clients can thus be divided into three stages:

- establishing a route from source to destination
- data transmission
- releasing the route.

Since routing is only blocked by occupation of partial routes and does not depend on actual data transmission, data is not explicitly modelled. Instead only the routing and releasing stages are included. This reduces the total number of tokens in the model at any given time and thus speeds up simulation.

Packets in the network are represented by tokens of the colourset *word* defined in Def. 1. Each message consists of a header (colourset *control*) defining the type of packet and a *content* part containing the destination address in case of a routing request or being empty.

```
colset coordinate = int with 0..15; // x and y coordinates of nodes
colset address = coordinate, coordinate; // node addresses (x, y)
colset control = with req | rel | relb | kill | ack; // control sequence to identify message type
colset content = address | empty; // contains either an address or is empty
colset word = control, content; // messages to be sent through the network
```

Def. 1. Coloursets describing messages

Each message sent through the net is preceded by a routing request (*req*), which is processed by the routing switches to establish a route. Upon reaching the destination an acknowledge signal (*ack*) is generated and sent to the source that then keeps idle for a number of clock cycles to represent data transmission. Afterwards the signal to release the route (*rel*) is sent, which is in turn answered by an acknowledge (*relb*) that initiates freeing of the route. If a routing request is blocked, a *kill* signal is generated by the appropriate routing switch to release the route and signal the source to retry.

For modelling traffic described by task graphs, the messages also contain an identifier that is unique to the originating data source.

The *word* colourset furthermore is a timed colourset to allow cycle-true modelling of the communication process.

#### B. Client

The library contains two types of clients, one generating randomly distributed traffic and the other supporting traffic models described by task graphs.

The client for randomly distributed traffic consists of two parts: data source and sink. Both parts are connected to the network by places that effectively represent the network interface. The source switches between three states represented by places: *idle*, *wait* and *active*. When not sending any data the source stays *idle*. After a configurable time the source generates a routing request – the corresponding *req* token is generated at the output place connecting the source to the

network. The destination of the route can be configured to be chosen randomly from a list of possible destinations. Waiting time before a request is sent can be configured to be any number of clock cycles either fix or random according to an arbitrary probability distribution function. Upon sending a request the source state switches to *wait* – waiting for the route to be established. If the route is blocked (a *kill* token is received), a new request is generated after an optional configurable pause. If the route is established successfully (*ack* token), the source switches from *wait* to *active*. By monitoring occupation of the place *wait* thus latency can be measured. The source stays in the *active* state for a time corresponding to the message length before generating a *release* signal and switching to *idle* again. Like the waiting time, message length can be configured to be fix as well as randomly distributed. The source load, defined as *active* time divided by total time can be measured similar to latency by monitoring occupation of the place *active*.

The only function of the sink contained in this client is to receive incoming *requests* and *acknowledge* them.

The client to support task graphs is configurable only to begin transmission after receiving a given number of messages from a list of sources according to the task graph. This conditional behaviour is enabled by the inclusion of unique identifiers into the messages. To register incoming data the sink generates tokens accordingly which are then removed by the source upon sending a routing request.

Timing of the client can furthermore be configured by setting processing times for several transitions modelling the answer of the client to incoming messages.

### C. Routing Switch

The routing switch is split up into several parts each processing packets of a single type. Each of these parts is connected to the input and output places that represent links connecting the routing switch to its neighbours. In case of a mesh or torus topology there are 4 links to the neighbouring nodes and two places to connect the routing switch to the clients as shown in Fig. 4.

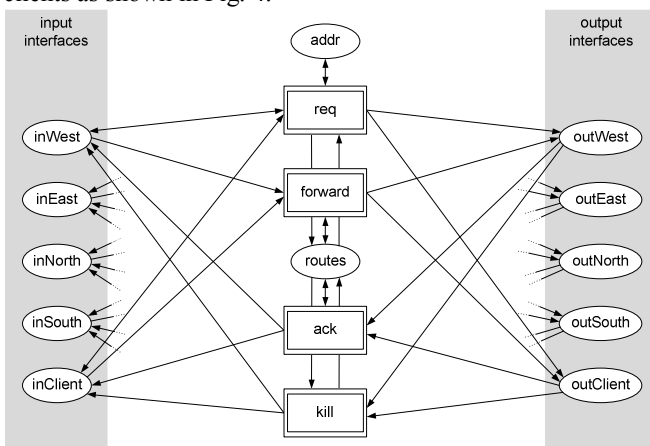


Fig. 4. CPN model of a routing switch for a mesh or torus topology

The places marked as interface places are mapped to links

when modelling a complete NoC, the place *routes* contains a list of routes currently occupied in this routing switch.

The module named *req* processes routing requests and consists of the router and arbiter modules. The router module evaluates the *content* part of an incoming *request*. The network address contained in the *content* is then checked against the local address of the routing switch in place *addr* by use of a transfer function similar to the one in Fig. 3. According to the routing algorithm (which is again realized as a transfer function) the request to reserve a port is sent to the arbiter module. If the requested port is free, the arbiter adds a token representing both the incoming and outgoing ports to the place *routes* and forwards the routing request appropriately. If the port is blocked and thus part of the *routes* list, the arbiter decides which route to grant priority according to the arbitration scheme resulting in a *kill* token to be generated. To change routing algorithms or arbitration schemes these modules have to be exchanged or modified.

The *forward* part handles *release* signals while the *ack* module processes acknowledge signals. Both of these models only forward the corresponding signals according to the routes stored in the *routes* place.

The *kill* section handles *kill* and *relb* signals. These are forwarded to the appropriate ports and the corresponding routes removed from the list.

By monitoring the occupation of place *routes* the load at each routing switch is observed.

As with the client processing times for any part of the routing switch are freely configurable.

### D. NoC Model

When a complete NoC is composed of library components the input and output places of clients and routing switches are mapped to places representing the links. As already mentioned above, the routing switches and clients themselves are represented by transitions. After assembling a complete NoC model, parameters can be adjusted at each individual component, for example to configure different traffic distributions at each client.

## IV. RESULTS

Several experiments with different NoC parameters were conducted using the model library described above. The average latency and load at each client, and the average load at each routing switch were measured and compared to the results of an FPGA based emulation [3] of the same scenario. The results of two exemplary experiments are presented below.

In the first scenario different routing algorithms were included in the model of a NoC with 5-by-5 mesh topology. Fig. 5 shows the achieved load at the sources considering an two variants of the xy-routing algorithm – a non adaptive and an adaptive one. The requested load only contains the time a source is sending and the pause between two messages without considering latency generated by establishing a route. The achieved load however does include latency and thus is always

lower than the requested load. This effect becomes more visible for high requested loads as routing requests are blocked more often due to the increased traffic on the network. As was to be expected the adaptive variant of the routing algorithm shows slightly better performance because routing conflicts can sometimes be resolved at the network node where they occur thus reducing latency and thereby increasing the achieved load.

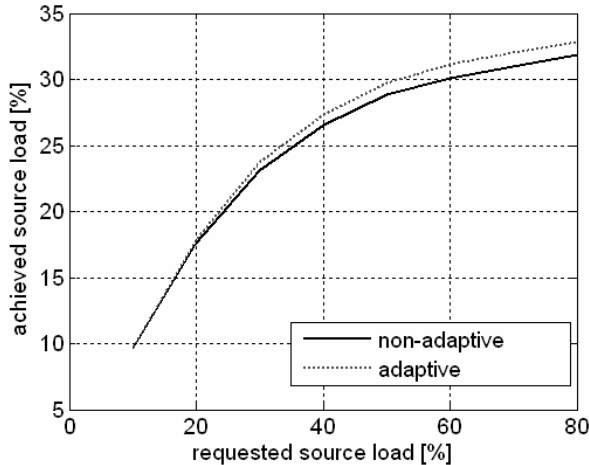


Fig. 5. Average achieved load using static and adaptive xy-routing

It is furthermore possible to analyze traffic conditions on the network with the CPN model, for example identifying hotspots in different traffic situations. A basic example of this is shown in Fig. 6. Fig. 6(a) shows the load of the routing switches in a 5-by-5 mesh NoC with uniformly random distributed traffic. In the second step two nodes at the edge of the NoC (addresses [1,0] and [3,0]) as well as the central one were configured to be accessed with higher probability than the rest of the nodes. This results in considerably higher loads along the paths leading to these nodes as shown in Fig. 6(b).

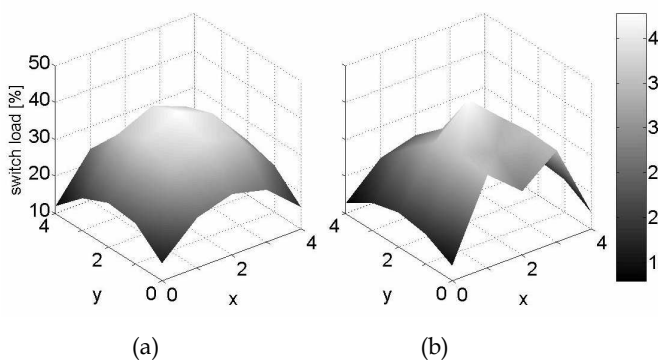


Fig. 6. Switch load for uniform and irregular traffic distribution leading to hotspots

For verification the results are compared to those of an FPGA based emulation. This reference is chosen because the VHDL-description of the NoC needed for synthesis on an FPGA is detailed enough also to be synthesizable on silicon. The results of the FPGA based emulation are therefore

considered to be very close to a real NoC implementation.

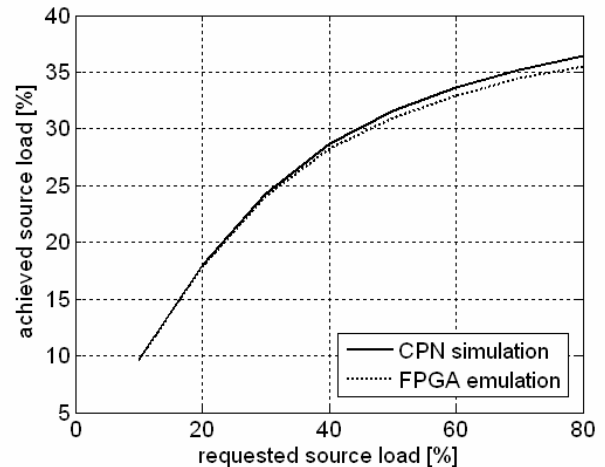


Fig. 7. Comparison of CPN modelling an FPGA based emulation

Comparison with the results of an FPGA based emulation of the same scenarios show the error to be well within 5% for all experiments conducted. An example is presented in Fig. 7 showing results of an experiment similar to that in Fig. 5 but with different message length settings. It is clearly visible that the CPN based model provides results with high accuracy. Considering that the FPGA based emulation is not exact either the achieved accuracy is approximately the same.

In case of the NoC used in the experiments presented, this accuracy is reached within 5 minutes of simulation time. For more complex NoCs simulation time increases like shown in Fig. 8.

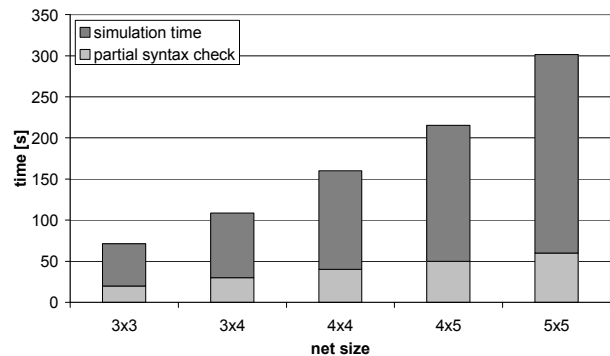


Fig. 8. simulation of 500,000 clock cycles and partial syntax check

Before simulation a complete syntax check of the model has to be done. Depending on the NoC size and complexity this takes several minutes (Fig. 9). Exchanging components such as the router or modifying parameters for example adjusting traffic conditions does not necessitate another complete syntax check but only a partial one (see Fig. 8).

Since the CPN model is kept very abstract the modelling effort is moderate. Based on library components assembly of a complete NoC model takes approximately one hour. The effort for creating said library components is dependent on the complexity of each component and ranges from several

minutes to hours.

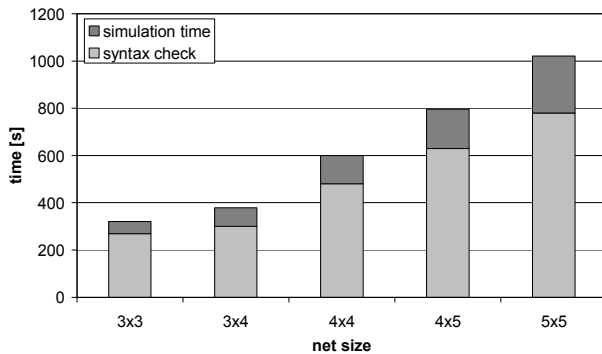


Fig. 9. Simulation of 500,000 clock cycles and complete syntax check

With exception of changes to the network topology or the size all parameter changes only require a partial syntax check to ready the model for simulation. This facilitates fast simulation of different scenarios and reduces the obvious drawback of the long complete syntax check.

Combining moderate effort and high accuracy CPN based modelling thus is well suited for performance analysis in an early design stage.

## V. SUMMARY

Due to its complexity and flexibility Networks-on-Chip feature a wide range of design parameters. This makes efficient design space exploration a primary task in the design process. To this end Coloured Petri Nets (CPN) can be employed as presented here.

A library of NoC component models was compiled and used to analyze several NoC scenarios. These experiments show CPN models to be an appropriate means of NoC performance analysis. With small changes to the abstract CPN model designers are able to evaluate the impact of parameters such as routing algorithms or traffic scenarios on NoC performance early on. Verification of the CPN based method by comparison with an FPGA based NoC emulator furthermore proved the CPN model to yield results of sufficient accuracy with only moderate effort.

CPN based performance analysis thus appears to be an attractive approach to explore the design space of a NoC early in the design process.

## REFERENCES

- [1] Bjerregaard, T. & Mahadevan, S. (2006). A Survey of Research and Practices of Network-on-Chip, *ACM Computing Surveys*, Vol. 38, article 1, March 2006
- [2] Sonntag, S.; Gries, M.; Sauer, C. (2005). SystemQ: A Queuing-Based Approach to Architecture Performance Evaluation with SystemC, *Proceedings of the SAMOS V Workshop*, Samos, Greece, July 18-20 2005, LNCS 3553, pp. 434-444
- [3] Neuenhahn, M.; Blume, H.; Noll, T. G. (2006). Quantitative analysis of network topologies for NoC-architectures on an FPGA-based emulator, *Proceedings of the URSI Advances in Radio Science - Kleinheubacher Berichte*, Miltenberg, September 2006

- [4] Lahiri, K.; Raghunathan, A.; Dey, S. (2001). System-level performance analysis for designing On-Chip communication architectures, *IEEE Transactions on CAD of Integrated Circuits and Systems*, June 2001.
- [5] Mickle, M. H. (1998). Transient and steady-state performance modelling of parallel processors, *Applied Mathematical Modelling* 22 (7) (1998) 533-543
- [6] Blume, H.; von Sydow, T.; Becker, D. Noll, T. G. (2007). Application of Deterministic and Stochastic Petri Nets for Performance Modelling of NoC Architectures, *Journal of Systems Architecture*, Vol. 53, Issue 8, 2007, pp. 466-476
- [7] Jensen, K. (1980). Net Models in System Development, PhD thesis, Aarhus University
- [8] Ratzler, A.V. et al., "CPN Tools for Editing, Simulating and Analysing Coloured Petri Nets", *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets (ICATPN) 2003*, 2003, p. 13