High-Level NoC Model for MPSoC Compilers

Christian Menard, Andrés Goens, and Jeronimo Castrillon
Center for Advancing Electronics Dresden (cfaed)
Technische Universität Dresden, Dresden, Germany
Email: firstname.lastname@tu-dresden.de

***Abstract*—Programming modern Multi-Processor Systems-on-Chip (MPSoCs) is a complex problem. To address it, academic compilers and programming flows exist that use internal hardware models and simulations to guide the automatic search for an efficient implementation. For the search to be effective, models need to be *accurate* to guide the search in the right direction and *fast* to allow for exploration of the large design space that comes with modern architectures. However, academic flows rarely evaluate the accuracy of their models against real hardware. In this paper, we introduce a high-level analytical Network-on-Chip (NoC) model, amenable for integration in MPSoC compilation flows, and validate it against actual hardware. On a trace-driven discrete event simulation, we show potential for a simulation speedup of an order of magnitude compared to a state-of-the-art SystemC model. We show that the model is accurate in low network congestion regimes, using $2,500$ randomly-generated applications. In the case study, our model is within 1% relative error more than 88% of the time, and within 5% more than 99% of the time. Finally, we also stress the model to determine a suitable operational range.**

I. Introduction

Data access speed plays a crucial role in the efficiency of execution in most computing systems. In the same manner, with the multicore era, inter-core communication has been increasingly important for ensuring performance and efficiency. As Multi-Processor Systems-on-Chip (MPSoCs) keep increasing in size and diversity, Network-on-Chip (NoC) technologies become imperative if the chip performance is to scale.

As new developments constantly push the boundaries of modern hardware, the difficulty of programming this hardware increases accordingly. In order to harness the power of NoC-based SoCs, a developer not only has to partition the applications but also explore where in the chip which part of the code will be executed. The complexity of this problem grows exponentially with the size of applications and architectures. Therefore, we need compilers that automate the deployment of applications on MPSoC platforms.

Several academic works are addressing the problem of efficient MPSoC programming, like MAPS [1], Sesame [2], Daedalus [3], DOL [4], or Turnus [5]. In order to decide how to partition and deploy applications, and to evaluate the decisions, most academic flows rely on simulations, which in turn rely on hardware models. The simulations should be fast in order to evaluate a large portion of the design-space in a short development cycle. They should also be accurate, in order to make right decisions and produce good results. However, most works assume fast and accurate simulations are given and focus on methods for using them. In particular, most academic flows for programming MPSoCs stay at the model level and do not validate their results running on actual silicon.

In this paper, we focus on the hardware models used in MPSoC compilers, and study their accuracy with respect to execution on actual hardware. For this, we introduce an analytical NoC hardware model for trace-driven discrete event simulation. Our model differs from traditional NoC-simulators like HNOCS [6], Booksim [7], or Noxim [8] which provide a cycle-accurate simulation of the NoC itself. This level of detail is too complex in order to quickly simulate applications executing on systems. Our model, instead, abstracts hardware details and focuses on efficient system-level simulation of entire applications for performance estimation in MPSoC compilers.

While the approach of trace-driven simulation on a high abstraction level warrants speed, we show it can also be accurate. In order to validate our model, we integrate it into the MAPS MPSoC programming flow [9] and compare simulations to measurements on a real MPSoC platform, the Tomahawk2 [10]. We do this with a pipeline application similar to those common in the multimedia and signal processing domains as well as with randomly-generated applications.

The rest of this paper is structured as follows. Section II gives background information. We introduce our NoC model in Section III and compare it to real hardware in Section IV. Section V presents related work, and Section VI concludes the paper.

II. Background

This section briefly introduces the Tomahawk MPSoC platform, that we use for validation, as well as the MAPS compiler framework, in which we integrated our NoC model.

A. Tomahawk

Tomahawk is a family of research MPSoCs targeting systems with high performance and energy-efficiency requirements [11]. For this paper, we use the second generation Tomahawk, the Tomahawk2 [10]. However, we only use a subset of the available components.

Figure 1 shows the simplified top-level view of the Tomahawk2, as used in this paper. It consists of eight Xtensa LX4 processing elements (PEs) with 64 kiB of local scratchpad memory and an SDRAM-Interface. A star-mesh NoC connects all components with each other. The NoC is packet switched with a packet size of 8 bytes, implements xy-routing, and is deadlock free. On the Tomahakw2, all network nodes and

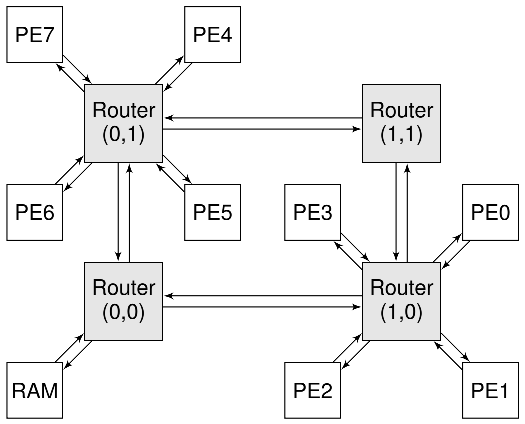


Fig. 1: A simplified top view of the Tomahawk2 MPSoC

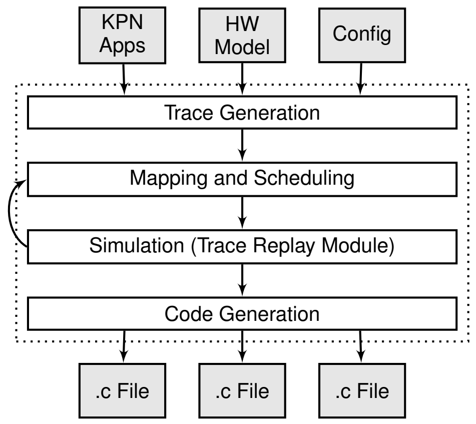


Fig. 2: The MAPS compiler framework

routers operate asynchronously in their own clock domain. Depending on the configuration, this may lead to a heterogeneous distribution of bandwidth among all the network links. In this paper, we use a homogeneous configuration where the routers operate at 500 MHz and the PEs operate at 200 MHz.

B. MAPS

MAPS is a retargetable compiler framework for programming heterogeneous MPSoC platforms [1]. It uses Kahn process networks (KPN) [12] as its underlying programming model and provides a complete tool-flow from KPN applications to implementations for various target platforms. Details on the programming model and compiler framework follow.

1) KPN: A KPN is a directed graph in which nodes correspond to processes and edges correspond to communication channels [12]. KPN processes may have an arbitrary control flow, but may only communicate by reading/writing atomic data elements (tokens) from/to channels. The channels are unbounded FIFO queues and allow a producing process to send tokens to a consuming process.

2) Compiler Framework: MAPS implements heuristics for deriving mappings of KPN applications. Here, a mapping is an assignment of KPN processes and channels to computation and communication resources on the target platform. In order to get information on the runtime behavior of KPN applications, MAPS derives execution traces for all KPN processes. Based on the process traces and a hardware model of the target platform, MAPS searches for an optimized mapping. Figure 2 illustrates the structure of the MAPS compiler framework.

A process trace is a sequence of segments, where a segment is a particular kind of path through a process' control flow graph (CFG) [1]. It starts and ends with a synchronization event, which, in this paper, is a read access from an input channel or a write access to an output channel.

In order to estimate the performance of KPN applications on a target platform, MAPS provides a discrete event simulator called TRM (trace replay module). Guided by the event traces, the TRM simulates the execution of a given application for a given mapping based on a hardware model of the target platform.

The hardware model itself comprises a processor model and a communication model. The processor model provides cost estimations for the execution of a trace segment on a given processor architecture. The costs for performing read and write accesses on KPN channels are captured by the communication model. In this paper, we focus on the communication model and replace the processor model by measured values.

3) Communication Model: The communication model of MAPS includes an interconnect model and so-called communication primitives. The interconnect model defines the available communication resources in the target platform, e.g., memories and buses. Communication primitives are an abstraction for communication via KPN channels. They describe and model the software API that implements the KPN semantics on the target platform. In this way, different APIs using the same hardware can be modeled and effectively used.

A communication primitive is defined as a four-tuple CP [9].

$$CP = \left(PE_i, PE_j, S \subseteq \mathcal{CR}, \mathcal{CM}^{CP}\right) \tag{1}$$

This tuple expresses how a process running on PE_i can communicate with a process running on PE_j via a set of communication resources S, a subset of $\mathcal{CR}$, the set of all communication resources that a platform provides. The cost model $\mathcal{CM}^{CP}$ is composed of three functions that map communication volume to a cost value in different phases [13]. When communicating x bytes, $\mathcal{P}^{CP}(x)$ represents data production costs on the producer side (PE_i), $\mathcal{T}^{CP}(x)$ transport costs in the interconnect (e.g., a DMA unit in S), and $\mathcal{C}^{CP}(x)$ consumption costs on the consumer side (PE_j).

$$\mathcal{CM}^{CP} = \left\{\mathcal{C}^{CP}(x), \mathcal{T}^{CP}(x), \mathcal{P}^{CP}(x)\right\} \tag{2}$$

We consider costs in terms of time. However, the costs can also represent other parameters, e.g., energy consumption.

III. NoC-Aware Communication Model

Since the MAPS hardware model was designed for modeling bus-based architectures, it is not well suited for NoC architectures. Firstly, the interconnect model is not aware of routers and links along the path of a data transport. Secondly, cost estimation only depends on the data volume but does not respect the network route. In this section, we describe an extended model for NoC-based architectures and derive a concrete model for the Tomahawk2.

A. Model Extension

In our model, a NoC is a list of routers, unidirectional links, and endpoints. An endpoint may be a global memory or a PE with local memory. Each link has a known constant bandwidth and the routers operate according to a known oblivious deterministic routing algorithm. The links either connect a PE and a router or two routers with each other. Given two endpoints A and B, this model can be used to derive the route from A to B. Knowing the route, one can determine the number of hops as well as the bandwidths of links along the route.

The NoC-aware cost model is based on the split cost model as shown in Equation 2, but extends the cost functions by additional arguments. This extension allows modeling the dependence of communication costs and the network route. The cost functions for consuming, transport, and producing map the data volume x as well as the number of hops h and the bandwidth of the route's slowest link b to a cost value.

$$\mathcal{CM}_{NoC}^{CP} = \left\{\mathcal{C}^{CP}(x,b,h), \mathcal{T}^{CP}(x,b,h), \mathcal{P}^{CP}(x,b,h)\right\} \quad (3)$$

By separating interconnect and cost model, it is possible to define communication costs independent of the definition of the network architecture. This allows for quick exploration of various network architectures while maintaining the network's characteristics. Additionally, the influence of various network configurations on the application performance can be explored for a given architecture.

B. Model Derivation

On the Tomahawk2 we distinguish three different ways of implementing KPN channels depending on the placement of the underlying FIFO buffer. This buffer is either located in the global RAM, in the scratchpad that is local to the producing process, or in the scratchpad that is local to the consuming process. In this paper, we focus on NoC-communication and, therefore, we do not consider the case of communicating via global RAM. This would require a model of the memory controller.

In order to create a complete communication model for the Tomahawk2, we need to define communication primitives for all possible ways of implementing KPN channels on the Tomahawk2. The communication primitives $P_{i,j}$ model communication from PE_i to PE_j via the producer's scratchpad S_i while the primitives $C_{i,j}$ model communication via the consumer's scratchpad S_j.

$$P_{i,j} = \left(PE_i, PE_j, S_i, \mathcal{CM}_{NoC}^{P}\right), \quad i \neq j \quad (4)$$

$$C_{i,j} = \left(PE_i, PE_j, S_j, \mathcal{CM}_{NoC}^{C}\right), \quad i \neq j \quad (5)$$

The communication primitives $P_{i,j}$ and $C_{i,j}$ require cost models for communication via the producer's scratchpad ($\mathcal{CM}_{NoC}^{P}$) and via the consumer's scratchpad ($\mathcal{CM}_{NoC}^{C}$), respectively. In order to define the cost models, we need to derive the cost functions for consuming, transporting, and producing. We do this by measuring the costs on the target device.

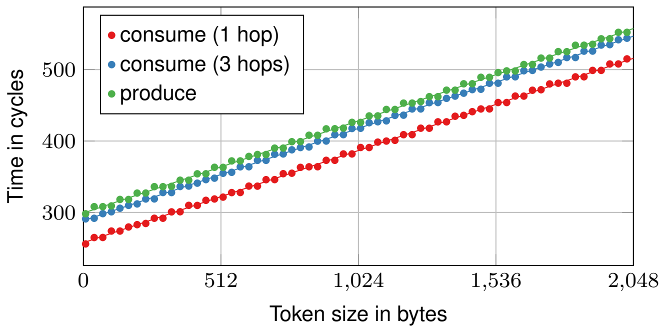


Fig. 3: Time measurement for remote buffer accesses.

$$\mathcal{CM}_{NoC}^{P} = \left\{\mathcal{C}^{P}(x,b,h), \mathcal{T}^{P}(x,b,h), \mathcal{P}^{P}(x,b,h)\right\} \quad (6)$$

$$\mathcal{CM}_{NoC}^{C} = \left\{\mathcal{C}^{C}(x,b,h), \mathcal{T}^{C}(x,b,h), \mathcal{P}^{C}(x,b,h)\right\} \quad (7)$$

In the case of the Tomahawk2, we assume that there is no transport delay as the implementation does not transfer data in parallel to processor operation, for which $\mathcal{T}^{P}(x,b,h) = \mathcal{T}^{C}(x,b,h) = 0$. Consuming or producing a token in a local buffer does not require any data transfer and, therefore, the costs for performing local channel accesses do not depend on data volume, bandwidth, or the number of hops. The costs only need to represent the execution of operations required to implement the access, which is a constant value. We measure the costs for local channel accesses on the Tomahawk2 to be 205 cycles for producing and 164 cycles for consuming, i.e., $\mathcal{C}^{C}(x,b,h) = 164$ and $\mathcal{P}^{P}(x,b,h) = 205$.

In contrast to local channel accesses, remote channel accesses require data transfers, and, therefore, the access costs depend on data volume, bandwidth, and the number of hops. In order to derive the cost functions for the Tomahawk2, we performed a series of measurements with varying data volumes and number of hops. Figure 3 shows the results of this measurement. More details on how we measured communication costs on the Tomahawk2 can be found in [14].

On the Tomahawk2, two PEs can only communicate over a distance of one hop or a distance of three hops. Therefore, we only need to consider these two cases. As the NoC on the Tomahawk2 does not send an acknowledgment when writing data, a process that produces a token in a remote buffer has no notion of the travel time of a single packet. Therefore, the time required for a remote produce operation is independent of the number of hops. Only the time required for a remote consume operation depends on the number of hops, as the process performing the access has to send a read request and waits until the reply arrives.

The plots in Figure 3 clearly have a linear trend. Therefore, we use linear regression analysis to derive the following functions:

$$\mathcal{C}^P_{h=1}(x) = 258 + 0.1256 \cdot x \tag{8}$$

$$\mathcal{C}^P_{h=3}(x) = 289 + 0.1256 \cdot x \tag{9}$$

$$\mathcal{P}^C(x) = 299 + 0.1256 \cdot x \tag{10}$$

We generalize $\mathcal{C}^P$ using linear interpolation and get:

$$\mathcal{C}^P(x, h) = 242.5 + 15.5 \cdot h + 0.1256 \cdot x$$

The slope in Equations 10 and III-B is the inverse data rate of the transfer. Therefore, we can generalize the equations for an arbitrary bandwidth by setting the slope to the inverse bandwidth.

$$\mathcal{P}^C(x, b, h) = 299 + \frac{x}{b} \tag{11}$$

$$\mathcal{C}^P(x, b, h) = 242.5 + 15.5 \cdot h + \frac{x}{b} \tag{12}$$

Apart from the communication model, we also need to derive an interconnect model for the Tomahawk2. The model simply resembles the Tomahawk2 NoC by using the same arrangement of links, routers, and endpoints (PEs). In order to define the bandwidth of all links in the model, we measured the link bandwidth on the Tomahawk2. We found that links connecting a PE and a router have a bandwidth of 8.0 byte/cycle. Links that connect two routers have a bandwidth of 10.2 byte/cycle.

IV. Evaluation

In this section we present two experiments that compare the TRM predictions using the proposed Tomahawk2 communication model with measured times on real hardware. The first experiment evaluates the model's accuracy by analyzing randomly generated KPN applications. In the second experiment, we use a synthetic benchmark to stress the model and thereby determine its operational range. We also report on the speed of the proposed model to show its applicability in MPSoC programming flows.

A. Random KPNs

In this experiment, we analyze a total of $2,500$ randomly generated KPN traces, 50 different graphs and 50 traces per graph. For graph generation we use the SDF-for-free (SDF3) tool [15] and then modify the traces in order to derive a KPN application, using the process described in [16]. In order to execute the generated application on the Tomahawk2, we created a script that generates source code from process traces. Computational segments are simply implemented by calling a function that waits for a given number of cycles.

Each one of the generated KPN applications consists of up to eight processes. Each channel has a mean token size of 512 bytes and each process segment has a mean length of 5,000 cycles. The Load Balancer [1] algorithm of MAPS is used to derive a mapping for each application. Then the TRM estimates the total execution time of this application and the application is executed on the Tomahawk2 in order to measure the actual execution time.

For each pair of measured and predicted execution times we calculate the relative error of the TRM prediction. Figure 4 shows characteristics of the distribution of the relative error in all $2,500$ measurements. The results show that most predictions are close to the actual measured value. The relative error is less than 1% for 88% of all analyzed KPNs and is less than 5% for 99% of all analyzed KPNs. However, there are cases where the TRM underestimates the total execution time significantly. We observed a maximum relative error of 15.3%. In order to examine the cause of significant mispredictions and to analyze the operating range of our model, we examine a synthetic benchmark in the following section.

Minimum	-0.983%
1st Quartile	-0.338%
Median	-0.191%
Mean	0.186%
3rd Quartile	0.258%
Maximum	15.331%

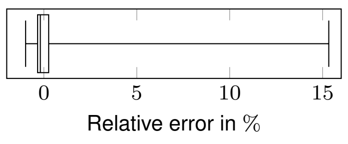


Fig. 4: Distribution of the relative error of TRM predictions.

B. Pipeline Application

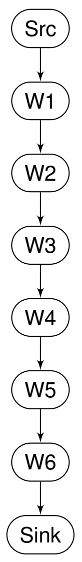
The second experiment analyzes performance predictions for a pipelined KPN application as shown in Figure 5a. The application consists of a source process, a sink process, and six worker processes. The source and sink processes permanently produce and consume tokens. The worker processes consume a token on the input channel, perform computations, and then produce a token on the output channel. For the sake of simplicity, the processes do not perform actual computations but wait for 1,000 cycles. Each process consumes and/or produces a total of 1,000 tokens. For the experiment, we vary the token size from 1 to 4,096 bytes.

The experiment considers two possible ways of mapping the pipeline application to the Tomahawk2. The best case mapping (Figure 5b) minimizes the distance between processes that communicate with each other and ensures that each link hast to serve at most one KPN channel. The worst case mapping (Figure 5b) maximizes the distance between processes and the links have to serve up to four KPN channels.

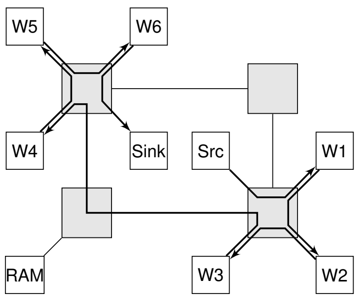
In addition to the process mapping, the placement of channel buffers has to be considered. For each channel, the buffer can be placed in the scratchpad of the producing process or in the scratchpad of the consuming process. In this experiment, we consider the case where all buffers are stored on the consumer side and the case where all buffers are stored on the producer side.

Figures 5c and 5e visualize the measured results. The plots show the measured total execution time of the KPN application for the best case and the worst case mapping as well as the TRM prediction. As the TRM prediction for best case and worst case mapping only differ by a few hundred cycles, the diagram only shows the TRM values for the best case mapping.

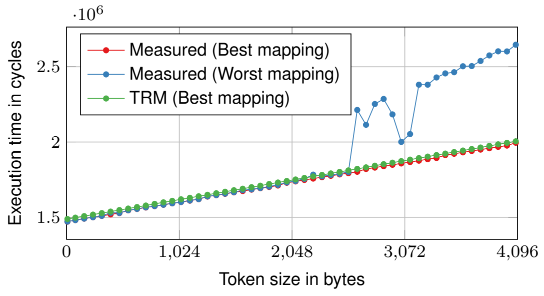
In the case that all channel buffers are mapped to the consumer scratchpad (Figure 5c), the measured best case execution time matches the TRM simulation. However, the measured execution time for the worst case mapping only matches the TRM prediction for relatively small token sizes



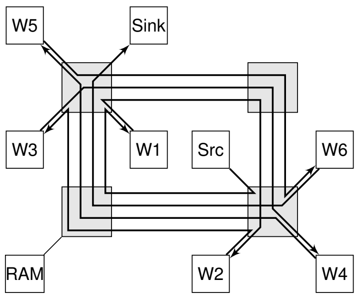
(a) KPN



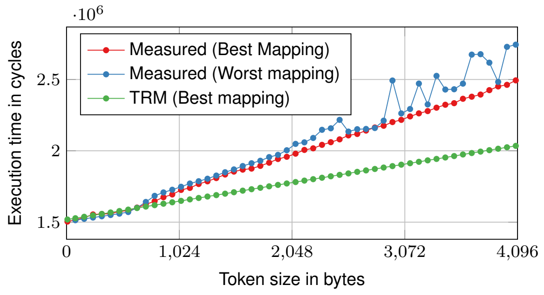
(b) Best Mapping



(c) Transfer via consumer scratchpad



(d) Worst Mapping



(e) Transfer via producer scratchpad

Fig. 5: Comparison of measured execution times and TRM predictions of a synthetic pipeline application for various cases.

and diverges significantly for token sizes larger than 2,048 bytes. This divergence is caused by network congestion as multiple KPN channels share the same network links.

Our model does not consider network congestion and, therefore, produces poor estimations for highly congested networks. However, we can detect congestion during simulation by comparing the bandwidth for all links to the combined average throughput of all KPN channels using the links. This is valuable information for an MPSoC compiler as it will try to avoid implementations that lead to network congestion. If cost predictions for congested networks are required, the simulator could try to estimate the costs based on total throughput of active channels and the link bandwidth, or it could fall back to a slower but more detailed model.

In the case that all channel buffers are mapped to the producer scratchpad (Figure 5e), the measured values for both the best case mapping and the worst case mapping are significantly higher than the predicted values. This behavior is caused by the interface between network and scratchpad memory. The interface cannot handle read and write requests from the network simultaneously, which causes additional delays.

In the pipeline application, a worker process W_i that finishes producing a token immediately consumes a token. At the same time, the following worker process W_{i+1} consumes the token produced by W_i. However, the network interface is already busy handling the read request of W_i and the read request of W_{i+1} is delayed. This leads to the discrepancy between the model predictions and measured execution times.

The prediction errors in Figure 5e, show that there are hardware effects that cannot be easily covered on the high abstraction level of our model. Analysis of the randomly generated KPNs showed, that the mispredictions are caused by the effect described above. However, as we demonstrated in the first experiment, we achieve accurate results for most of the cases despite this limitation.

C. Simulation Speed

To evaluate the speed of the proposed model, we integrated it on a simple TRM implementation that is based on the *simpy* Python module for discrete event simulation [17]. We simulate a pipeline, similar to the application described in the previous section, with 16 processes on a 4×4 NoC. We found that our TRM module simulates the 15,000 transactions (produce, transfer, and consume) of the pipeline application in about a second on an Intel® Core™ i7-6500U based system. As we do not consider dynamic behavior of the NoC, this value does neither depend on the simulated NoC architecture nor on the token size of the simulated channels.

For comparison purposes, we measured the time required for simulating a similar workload (15,000 packets, 0.001 $^{\text{packets}}\!/_{\text{cycle}}$ injection rate) on a 4×4 NoC in the Systemc-based Noxim [8]. For a payload of 512 bytes per transaction/packet and a flit size of 64 bits, we found that Noxim requires around 10 seconds per simulation. This is one order of magnitude slower than our unoptimized TRM implementation. However, the speedup depends on the average payload per transaction or packet. As Noxim simulates every single FLIT, the simulation effort and, therefore, also the speedup, increases with the packet size. For packets with only one data flit (8 byte payload) Noxim still requires about four seconds.

V. Related Work

In the literature, there are various tools for high-level evaluation of MPSoC designs that use trace-driven simulation [18]–[21]. SPADE is an early example of an efficient methodology for exploration of signal processing architectures [18]. Similar to our approach, SPADE maps KPN applications to architecture models and uses trace-driven simulation for performance evaluation. Lahiri, Raghunathan, and Dey described a trace-driven approach for accurate modeling of dynamic behavior in bus communication for various protocols [19]. Plyaskin, Masrur, Geier, *et al.* presented a trace-driven SystemC TLM SoC simulator that considers application as well as OS workload and precisely models memory accesses [20]. RAPITIMATE is a timing estimation framework that targets pipelined MPSoCs [21]. All these approaches focus on bus-based systems and do not consider communication in a NoC-based architecture.

There are also several simulation tools for exploration of NoC architectures [6]–[8]. HNOCS is an open-source NoC simulator that targets heterogeneous NoC designs [6]. Booksim is cycle-accurate NoC-simulator that uses a modular structure and provides detailed and configurable implementations of various network components [7]. Noxim is a cycle-accurate SystemC NoC simulator that considers not only traditional NoC architectures but also models wireless NoCs [8]. All of these tools are designed for exploration of NoC designs and provide very detailed simulations. However, this level of detail leads to relatively slow simulations and is not required for evaluation of application mappings. These simulators also are often limited to certain topologies and switching techniques.

VI. Conclusion

We presented an abstract analytical NoC-model for performance estimation of applications mapped to MPSoC platforms and integrated it with a trace-driven simulator. We compared simulation results with measured execution times on the Tomahawk2 and found that our high-level approach provides accurate prediction without the need for a detailed hardware model in most of the cases. In contrast to related work, our approach combines trace-based simulation with an abstract NoC-model. We showed that this allows us to provide fast and accurate evaluation of application mappings.

Future work includes the integration of a model for network congestion. We want to analyze, whether it is possible to accurately predict the overhead in communication costs for congested networks on a high abstraction level. Furthermore, we want to evaluate our model on larger systems and analyze its applicability for multi-application scenarios.

Acknowledgment

This work was partly supported by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden" (cfaed). The authors would also like to thank Silexica (www.silexica.com) for making their embedded multicore software development tool available to us.

References

[1] J. Castrillon, R. Leupers, and G. Ascheid, "MAPS: Mapping concurrent dataflow applications to heterogeneous MPSoCs," *IEEE Trans. Ind. Inform.*, vol. 9, no. 1, pp. 527–545, Feb. 2013.

[2] C. Erbas, A. D. Pimentel, M. Thompson, and S. Polstra, "A framework for fystem-level modeling and simulation of embedded systems architectures," *EURASIP J. Embed. Syst.*, vol. 2007, pp. 1–11, 2007.

[3] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere, "Daedalus: Toward composable multimedia MP-SoC design," *Proc. 45th ACMIEEE Des. Autom. Conf. DAC08*, pp. 574–579, Jun. 2008.

[4] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping applications to tiled multiprocessor embedded systems," *Proc. Th 7th Int. Conf. Appl. Concurr. Syst. Des. ACSD07*, pp. 29–40, Jul. 2007.

[5] S. Casale-Brunet, C. Alberti, M. Mattavelli, and J. W. Janneck, "Turnus: A unified dataflow design space exploration framework for heterogeneous parallel systems," *Proc. 2013 Conf. Des. Archtictures Signal Image Process. DASIP13*, pp. 47–54, Oct. 2013.

[6] Y. Ben-Itzhak, E. Zahavi, I. Cidon, and A. Kolodny, "HNOCS: Modular open-source simulator for heterogeneous NoCs," *2012 Int. Conf. Embed. Comput. Syst. SAMOS*, pp. 51–57, Jul. 2012.

[7] N. Jiang, J. Balfour, D. U. Becker, B. Towles, W. J. Dally, G. Michelogiannakis, and J. Kim, "A detailed and flexible cycle-accurate network-on-chip simulator," *2013 IEEE Int. Symp. Perform. Anal. Syst. Softw. ISPASS2013*, pp. 86–96, Apr. 2013.

[8] V. Catania, A. Mineo, S. Monteleone, M. Palesi, and D. Patti, "Noxim: An open, extensible and cycle-accurate network on chip simulator," *2015 IEEE 26th Int. Conf. Appl.-Specif. Syst. Archit. Process. ASAP13*, pp. 162–163, Jul. 2015.

[9] J. Castrillon and R. Leupers, *Programming heterogeneous MPSoCs*. Cham: Springer International Publishing, 2014.

[10] B. Noethen, O. Arnold, E. P. Adeva, *et al.*, "10.7 A 105GOPS 36mm2 heterogeneous SDR MPSoC with energy-aware dynamic scheduling and iterative detection-decoding for 4G in 65nm CMOS," *2014 IEEE Int. Solid-State Circuits Conf. Dig. Tech. Pap. ISSCC14*, pp. 188–189, Feb. 2014.

[11] T. Limberg, M. Winter, M. Bimberg, *et al.*, "A heterogeneous MPSoC with hardware supported dynamic task scheduling for software defined radio," *Proc. 46th ACMIEEE Des. Autom. Conf. DAC09*, 2009.

[12] G. Kahn, "The semantics of a simple language for parallel programming," pp. 471–475, 1974.

[13] M. Odendahl, J. Castrillon, V. Volevach, R. Leupers, and G. Ascheid, "Split-cost communication model for improved MPSoC application mapping," *2013 Int. Symp. Syst. Chip SoC*, pp. 1–8, Oct. 2013.

[14] C. Menard, "Mapping KPN-based applications to the NoC-based Tomahawk architecture," Master's Thesis, TU Dresden, Mar. 24, 2016. [Online]. Available: https://cfaed.tu-dresden.de/files/user/jcastrillon/theses/1603_Menard_DA.pdf (visited on 10/10/2016).

[15] S. Stuijk, M. Geilen, and T. Basten, "SDF3: SDF for free," *Appl. Concurr. Syst. Des. ACSD06*, pp. 276–278, 2006. (visited on 08/22/2016).

[16] J. Castrillon, A. Tretter, R. Leupers, and G. Ascheid, "Communication-aware mapping of KPN applications onto heterogeneous MPSoCs," *Proc. 49th Des. Autom. Conf. DAC12*, pp. 1262–1267, Jun. 2012.

[17] K. Müller and T. Vignaux, *SimPy: Simulating systems in Python*, Feb. 27, 2003. [Online]. Available: http://www.onlamp.com/pub/a/python/2003/02/27/simpy.html (visited on 08/22/2016).

[18] P. Lieverse, P. van der Wolf, E. Deprettere, and K. Vissers, "A methodology for architecture exploration of heterogeneous signal processing systems," *1999 IEEE Workshop Signal Process. Syst. SiPS99*, pp. 181–190, 1999.

[19] K. Lahiri, A. Raghunathan, and S. Dey, "System-level performance analysis for designing on-chip communication architectures," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 20, no. 6, pp. 768–783, Jun. 2001.

[20] R. Plyaskin, A. Masrur, M. Geier, S. Chakraborty, and A. Herkersdorf, "High-level timing analysis of concurrent applications on MPSoC platforms using memory-aware trace-driven simulations," *2010 18th IEEEIFIP Int. Conf. VLSI Syst.–Chip*, pp. 229–234, Sep. 2010.

[21] S. M. M. Shwe, K. Batra, Y. Yachide, J. Peddersen, and S. Parameswaran, "RAPITIMATE: Rapid performance estimation of pipelined processing systems containing shared memory," *2015 33rd IEEE Int. Conf. Comput. Des. ICCD*, pp. 635–642, Oct. 2015.