Mapping Real-life Applications on Run-time Reconfigurable NoC-based MPSoC on FPGA

Amit Kumar Singh [#1], Akash Kumar [*2], Thambipillai Srikanthan [#3] and Yajun Ha [*4]

[#] *School of Computer Engineering, Nanyang Technological University, Singapore*
[1] amit0011@ntu.edu.sg, [3] astsrikan@ntu.edu.sg

[*] *Department of Electrical and Computer Engineering, National University of Singapore, Singapore*
[2] akash@nus.edu.sg, [4] elehy@nus.edu.sg

(Demonstration Paper)

***Abstract*—Multiprocessor systems-on-chip (MPSoC) are required to fulfill the performance demand of modern real-life embedded applications. These MPSoCs are employing Network-on-Chip (NoC) for reasons of efficiency and scalability. Additionally, these systems need to support run-time reconfiguration of their components to cater to dynamically changing demands of the system. Designing and programming such systems for real-life applications prove to be a major challenge. This paper demonstrates the designing of reconfigurable NoC-based MPSoC and programming it for real-life applications. The NoC is reconfigured at run-time to support different combinations of multiple applications at different times.**

The platform is verified with a case study executing the parallelized C-codes of a simple producer-consumer and JPEG decoder applications on a NoC-based MPSoC on a Xilinx FPGA. Based on our investigations to map the applications on a 3×3 platform, we show that the NoC reconfiguration overhead is kept at a minimum and the platform utilizes 85% of the total available slices of Virtex-5 FPGA. Moreover, we show that the proposed approach is highly scalable when targeting for large number of applications.

I. Introduction

The advancements in nanometer technology have allowed to integrate several embedded processors on a single chip creating multiprocessor systems-on-chip (MPSoCs). The MPSoCs are proved as a promising solution to meet the increasing performance requirement of real world complex applications [1]. The communication demand of processors is fulfilled by Networks-on-Chip (NoC) [2] that are efficient and a scalable alternative over shared buses.

MPSoC on Field Programmable Gate Arrays (FPGAs) is a new and increasingly important trend. These days several FPGA-based MPSoC are appearing [3]. These facilitate rapid prototyping and allow for research in new architectures without the worries of their ASIC production. However, these have reduced performance compared to their ASIC counterpart but offer several advantages like flexibility, reconfiguration, less time-to-market and less cost, to compensate for the same. The latest FPGAs can accommodate 80-100 soft-core processors in a single chip to create an MPSoC, and NoC is the best solution to manage such large number of cores [4].

Modern embedded systems such as smart phones, PDAs etc. need to support multiple applications to fulfill the end user requirements, but all the applications are not active simultaneously. The combination of simultaneously active applications from all possible applications is defined as a *use-case*. A use-case requires some specific configuration for the FPGA to meet the performance requirement and there might be large number of use-cases to be supported in the system, requiring their specific configurations. The appropriate configuration needs to be loaded into the reconfigurable platform at run-time, as and when required.

In this paper, we demonstrate a flow for designing a low area overhead NoC-based MPSoC for FPGA platforms and then programming it for real-life applications. The applications are specified in the form of parallelized real C codes with a corresponding Synchronous Data Flow (SDF) graph model [5]. SDF graphs are often used to model DSP and concurrent multimedia applications [5]. Mapping and evaluating SDF models on the platform is relatively easy and SDF graphs of many applications are already available [6]. Our flow allows mapping of parallelized real C codes of real-life applications as well as SDF models.

The design flow to generate NoC-based MPSoC takes a low area overhead NoC [7], which supports run-time reconfiguration and also provides throughput guarantees. This run-time reconfiguration feature enables us to support different use-cases at different times. The applications' program codes are loaded and compiled onto the platform processors in advance after finding suitable placement for them using efficient mapping techniques reported in literature [8]. This obviously costs more memory but avoids the overhead of loading and compiling them at run-time when required, and we have assigned sufficient memory to all the processors. It should be mentioned that it is also possible to load and compile the program codes at run-time, but at the cost of a high overhead. Whenever a particular use-case (set of active applications) needs to be supported in the platform, the corresponding already compiled codes are enabled at the mapped processors after configuring the NoC by the required configuration to support the use-case. The NoC reconfiguration overhead is kept at a minimum so that overall execution time for the use-case is minimized.

There have been some quite recent works to generate

978-1-4244-8983-1/10/$26.00 ©2010 IEEE

NoC-based MPSoC for FPGA platforms [9] [10] [11], but, these are not area efficient and don't support for efficient run-time reconfiguration. Thus, they can not cater for the scenarios when different use-cases need to be supported at different times. Our approach is area efficient and has very low reconfiguration overhead. We have targeted Vixtex-5 ML510 board [12]. We present case studies using SDF graphs of JPEG and H263 decoders, and parallelized real C codes of producer-consumer and JPEG decoder applications, whereas existing works don't target for the real C codes.

The rest of the paper is organized as follows. Section II introduces SDF graphs. Section III gives an overview of our mapping flow including the experimentation to demonstrate the case study for different use-cases. Section IV concludes the paper and provides direction for future work.

II. Synchronous Data Flow Graphs

An example of an SDF graph is shown in Fig. 1. There are three tasks A, B and C in the graph. The values inside the circles represent execution times of tasks. A directed edge represents the dependency between the tasks, just like any typical data flow graph. In order to start the execution of a task, it needs some input data and produces some output data after finishing the execution. These data are referred as *tokens*. Tasks execution is also called *firing*. A task becomes *ready* when there are sufficient input tokens on all of its input edges and sufficient buffer space on all of its output channels. Task can only fire when it is ready.

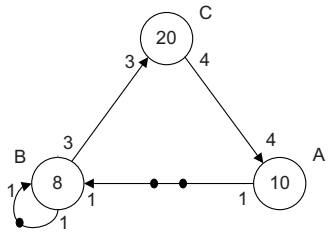

Fig. 1. Example of an SDF graph

The edges might have *initial tokens*, indicated by bullets on the edges, as shown on the edge from task *A* to *B* in Fig. 1. Buffers can be modeled as an edge with initial tokens, where the number of tokens on the edge indicates the available buffer size. The available buffer size reduces when a task writes data to such channels and the available buffer i.e. the token count increases when the receiving task consumes the written data.

In the example provided in Fig. 1, only task B can start its execution from the initial state as it has the required number of tokens on all of its incoming edges. Once task B finishes execution, it will produce 3 tokens on the edge to C. Now, C can proceed as it has sufficient tokens and will produce 4 tokens on the edge to A upon completion.

III. Demonstration Overview

In this section, we present an overview of our demonstration to map multiple applications on NoC-based MPSoC. First, we demonstrate how a generic NoC-based MPSoC platform is generated, in the Section III-A, then programming of the platform to support multiple use-cases of multiple applications, in Section III-B and finally, experiments performed for different case studies and results obtained, in Section III-C.

A. NoC-based MPSoC Generation

Fig. 2 shows an overview of NoC-based MPSoC generation. First, a desired NoC dimension is provided to generate the NoC using the NoC Generator tool [7]. This tool generates a run-time reconfigurable spatial-division-multiplexing (SDM) based NoC with throughput guarantee. The guaranteed throughput is obtained by reserving the links of the NoC in advance and avoiding the resource contention. In contrast, in a packet switched NoC [13] [14], providing throughput guarantee is difficult since there are no dedicated links.

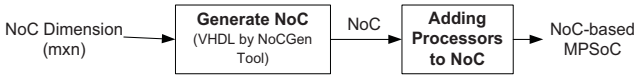

Fig. 2. MPSoC Generation Flow

After generating the NoC, required number of microblaze processors (mxn) [12] are added to the network interfaces of the NoC to generate NoC-based MPSoC platforms. The network interface ports of the NoC are connected to the Fast Simplex Link (FSL) ports of the processors through FSL communication buses available in the EDK IP catalog, for providing hight speed communication [12].

In order to transfer data from one processor to another, first of all, the NoC is configured with the required connection requirements and then data is written on the master FSL port of the sender processor to go through the network and finally, it is read at the slave FSL port of the receiver processor. The different configurations for different connection requirements are also generated by the NoC Generator tool [7]. These configurations are required to be loaded into the platform at run-time to fulfill the communication needs of processors.

Thus, we are able to generate run-time reconfigurable NoC-based MPSoC platforms. These platforms have low area overhead and the processors present in the platform can communicate faster through the dedicated links.

B. Programming MPSoC

Fig. 3 shows the process of programming the NoC-based MPSoC platform for supporting multiple applications. First, the platform is synthesized. Then, program codes of applications are loaded and compiled onto the platform processors after finding suitable mapping for each application (Load Codes onto Processors). Now, if a particular use-case (set of active applications) needs to be executed then, first, the NoC is configured with the appropriate configuration (fulfilling the communications requirements of the use-case) and then the required program codes corresponding to the use-case are enabled. The NoC configurations for the possible use-cases are stored in a controlling processor in advance to avoid the

overhead of creating the configuration data at run-time. Thus, we just need to worry about the reconfiguration of NoC with the already present configurations.

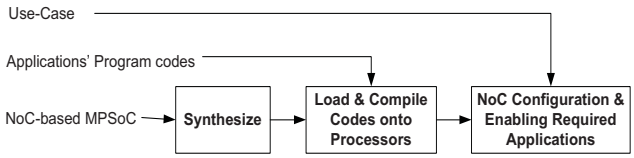

Fig. 3. MPSoC Programming Flow

The program codes representation is important in our flow. If we consider SDF graph models of the applications then the program codes will implement just some delays as indicated in the circles in Fig. 1. However, if we take the parallelized real C codes of the applications then these codes need to be placed onto the processors instead of some delay based code as in SDF. Further, when using the real C codes, special care has to be taken for sending and storing application data.

This procedure to execute a use-case facilitates faster execution as we do not need to bother about the overhead in loading and compiling the required program codes at run-time. Further, it should be noted that NoC reconfiguration overhead is kept at a minimum. When another use-case needs to be executed then the same procedure (NoC Configuration & Enabling Required Applications) is adopted. Thus, we are able to support multiple use-cases of multiple applications without re-synthesizing the platform. The synthesis is the biggest bottleneck in designing FPGA-based MPSoC due to large turn-around time.

C. Experiments

In this subsection, we demonstrate some observations that were obtained by implementing SDF graphs and parallelized real C codes using our mapping flow described in Sections III-A and III-B. The main objective of this experiment is to show that we are able to support multiple use-cases of multiple applications. We show that the reconfiguration overhead to configure the NoC is low. Separate case studies are done for SDF graphs and parallelized real C codes of applications. Our implementation platform is Xilinx Virtex-5 ML510 development board. Xilinx EDK 12.1 and ISE 12.1 were used for synthesis and implementation. All the microblaze processors in the platform are run at 100 MHz.

1) SDF Graphs Case Study: In this case study, we have taken SDF models of JPEG and H263 decoders. The JPEG and H263 decoders have been partitioned into 6 and 4 tasks respectively. The SDF model of H263 decoder is shown in Fig. 4. The JPEG decoder application is mapped on 6 processors in a 3×3 architecture on the basis of one task per processor. The H263 application needs 4 processors and it is mapped on top of the JPEG decoder application and thus sharing four processors with JPEG decoder. So, finally, we utilized total 6 processors out of the 3×3 platform, where 4 processors share tasks from JPEG and H263 decoder and 2 processors execute tasks of JPEG decoder only. As there are more available resources, so more applications can be mapped, making the platform scalable for large number of applications. Further, a bigger platform can be created depending upon the availability of logic resources in the FPGA.

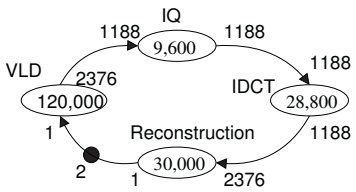

Fig. 4. SDF graph of H263 decoder

In order to evaluate multiple use cases from these two applications, we selected to run the following use-cases:

- Only JPEG decoder active
- Both JPEG and H263 decoder active
- Only H263 decoder active

TABLE I
RECONFIGURATION OVERHEAD FOR DIFFERENT USE-CASES

Use-cases	Reconfiguration Time (ms)
Only H263	0.05259
Only JPEG	0.07846
JPEG and H263	0.07849

We were able to successfully run all the above use cases one by one after configuring the NoC for them. The NoC configuration overhead (in milliseconds) for different use-cases is reported in Table I. We can see that the reconfiguration overhead is maintained at very low. This is because we are using a low area overhead NoC. A couple of observations can be made from Table I. First, configuration overhead for 'Only H263' is minimum. This is because the connection is required to connect 4 processors only. Second, configuration overhead for 'JPEG and H263' is maximum as 6 connections are required for JPEG and only 4 for H263 decoder.

2) Parallelized C codes Case Study: In this case study, we have taken parallelized real C codes of producer-consumer and JPEG decoder applications. The producer-consumer and JPEG decoder applications are partitioned into 2 and 5 tasks respectively. Here, we have mapped all the tasks on two processors only - consumer task & 2 tasks of JPEG decoder on first processor and producer task & 3 tasks of JPEG decoder on the second processor. We performed a different type of mapping to differentiate it from the one-to-one mapping as done for the SDF graphs and to also see if we can observe some different behaviors. As we can see that we have utilized only 2 processors of the 3×3 platform, so we can use rest of the processors for many other applications. Thus, the approach is highly scalable when targeting for large number of applications. We intend to test with real C codes of other real-life applications as and when they are available.

Here, we have selected following use-cases:

- Only Producer-Consumer active

- Both Producer-Consumer and JPEG decoder active
- Only JPEG decoder active

TABLE II
RECONFIGURATION OVERHEAD FOR DIFFERENT USE-CASES

Use-cases	Reconfiguration Time (ms)
Only Producer-Consumer	0.02885
Only JPEG	0.02886
Producer-Consumer and JPEG	0.02891

We were also able to successfully run all these use cases one by one after configuring the NoC for them. The NoC configuration overhead (in milliseconds) for different use-cases is reported in Table II. We can see that the reconfiguration time required for evaluated use-cases is very low (in milliseconds). A few observations can be made from Table II. First, the reconfiguration time in all the use cases in this case study is low as compared to the previous case study. The reason behind this lies in the fact that here, we have mapped all the tasks only on two processors so the connection is required between two processors only. This requires relatively less NoC configuration data as compared to the cases where connection is required to connect more than two processors like the previous case study. Second, we can see that the reconfiguration time for the 'Producer-Consumer and JPEG' use-case is more than other two use cases. This happens because more connections are required to fulfill the communication needs of two simultaneously active applications and thus more NoC configuration data, requiring more time for the configuration.

IV. CONCLUSION

This paper describes the steps required to generate a low-area overhead NoC-based MPSoC from a NoC that is run-time reconfigurable and provides throughput guarantee. Additionally, a methodology to program the NoC-based MPSoC for multiple use-cases of multiple applications has been presented. We have avoided the run-time loading and compilation overhead of program codes as well as of NoC configuration data by placing them on the platform processors in advance. While this uses more memory but the execution is very fast. However, the memory cost can be reduced at the cost of overhead in loading and compiling the codes at run-time. By keeping the program codes and NoC configurations on the platform processors in advance, applications can be executed just by configuring the NoC with the required already present configuration and enabling the program codes of applications at different processors. The applications are evaluated on a 3×3 platform synthesized on a Xilinx Virtex 5 FPGA.

For the future work, we are planning to evaluate more parallelized real C codes of real-life applications as and when we get any of them. Further, there are free processors on the platform so that they can cater for the support of new applications making the platform scalable. We also plan to evaluate different possible mapping combinations like mapping an application onto different processor counts and then evaluating them. To overcome the memory limitation issues on the processors, we would like to devise efficient techniques to load and compile the program codes as well as NoC configuration data, at run-time. We also wish to include support for generating heterogeneous platforms and then programming them as required. For all the above mentioned future work, we would like to target parallelized real C codes.

ACKNOWLEDGMENT

We thank Mr. Zhiyao Joseph Yang for providing us the source of the NoC [7] used in our work.

REFERENCES

[1] A. Jerraya et al., "Guest editors' introduction: Multiprocessor systems-on-chips," *Computer*, vol. 38, no. 7, pp. 36–40, 2005.
[2] L. Benini et al., "Networks on chips: a new soc paradigm," *Computer*, vol. 35, no. 1, pp. 70–78, 2002.
[3] T. Dorta et al., "Overview of fpga-based multiprocessor systems," in *ReConFig '09.*, Dec 2009, pp. 273 –278.
[4] G.-G. Mplemenos et al., "Mplem: An 80-processor fpga based multi-processor system," in *FCCM '08.*, 2008, pp. 273–274.
[5] E. A. Lee et al., "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. Comput.*, vol. 36, no. 1, pp. 24–35, 1987.
[6] S. Stuijk et al., "SDF^3: SDF For Free," in *ACSD '06.*, June 2006, pp. 276–278.
[7] Z. J. Yang et al., "An area-efficient dynamically reconfigurable spatial division multiplexing network-on-chip with static throughput guarentee," in *FPT'10*.
[8] A. K. Singh et al., "Communication-aware heuristics for run-time task mapping on noc-based mpsoc platforms," *Journal of Systems Architecture*, vol. 56, no. 7, pp. 242 – 255, 2010.
[9] S. Lukovic et al., "An automated design flow for noc-based mpsocs on fpga," in *RSP '08*, 2008, pp. 58–64.
[10] S. V. Tota et al., "A case study for noc-based homogeneous mpsoc architectures," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 17, no. 3, pp. 384–388, 2009.
[11] A. Kumar et al., "An fpga design flow for reconfigurable network-based multi-processor systems on chip," in *DATE '07*, 2007, pp. 117–122.
[12] "Xilinx," 2008, http://www.xilinx.com/.
[13] E. Bolotin et al., "Qnoc: Qos architecture and design process for network on chip," *J. Syst. Archit.*, vol. 50, no. 2-3, pp. 105–128, 2004.
[14] D. Bertozzi et al., "Noc synthesis flow for customized domain specific multiprocessor systems-on-chip," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 2, pp. 113–129, 2005.