A Framework for Heterogeneous Many-core SoCs Generation

Marcelo Ruaro*, Luciano L. Caimi*†, Vinicius Fochi*, Fernando G. Moraes*
*PUCRS – Av. Ipiranga 6681, Porto Alegre, Brazil – {marcelo.ruaro, vinicius.fochi}@acad.pucrs.br, fernando.moraes@pucrs.br
†UFFS – Av. Fernando Machado 108E, Chapecó, Brazil – lcaimi@uffs.edu.br

***Abstract*—This work presents a framework for heterogeneous many-core SoCs generation, which comprises a flexible EDA (Electronic Design Automation) framework and a many-core model for heterogeneous SoCs. The framework together with the many-core model supports the integration of processors, network interfaces, routers, and peripherals. The hardware model is cycle-accurate, with a SystemC model to speed up simulation time and a VHDL model enabling prototyping in FPGAs devices. The framework provides a rich set of graphical debugging tools enabling an easy and intuitive understanding of computation and communication events happening at runtime. The coupled integration of the platform model to the EDA framework makes the many-core well suited to be employed in research and teaching. As case-study, we provide an evaluations addressing the many-core generation, simulation, and debugging.**

***Index Terms*—Heterogeneous Many-core; NoC (Network-on-Chip); Architecture Model; Debug Framework.**

I. Introduction

The design of many-core SoCs became predominant for high-performance circuits. Such systems increase computing power through parallel computation due to thread-level parallelism (at the system level) by splitting an application into tasks that can run in parallel over several Processing Elements (PEs).

The many-core design can be divided into logical and physical design phases. Logical phase is concerned with functional requirements. In this phase, the circuit is described in a hardware description language. The modules of each PE are developed and integrated. The system is simulated using an RTL simulator, and the results are used to validate the design according to the specifications. The physical phase is concerned with the synthesis of the circuit according to the target technology.

While the physical steps have a well-defined design flow provided by CAD tools, the many-core logical design is an open field to frameworks aiming design space exploration, automatic system generation, and validation.

Table I presents related works related to many-core frameworks. Features of our proposal include: (1) a scalable many-core SoC with an hierarchical organization, allowing the evaluation of large systems (e.g. 16x6); (2) support to the connection with peripherals, as hardware accelerators; (3) an RTL model enabling to capture detailed performance figures, as frequency and energy consumption; (4) set of graphical debugging tools providing views as packet paths in the NoC, tasks' mapping, tasks' scheduling, tasks' messages.

978-1-7281-0453-9/19/$31.00 ©2019 IEEE

TABLE I: Related works Many-Core/MPSoC frameworks.

Work	Peripheral Support	RTL Validation	Online Support	GUI
Monemi et al.[1]-2017	As an IP	Verilog	OpenCores	Generation
Elmohr et al. [2]-2018	Inside PE	Verilog	No	No
Busseuil et al.[3]-2011	No	VHDL	No	No
Zhang et al. [4]-2015	No	No	No	No
Balkind et al. [5]-2016	No	Verilog	Own site	No
Skalicky et al. [6]-2015	No	VHDL	No	No
This Work	Chip borders	VHDL, SystemC	Git, Own site	Debugging

The *goal* of this work is to present the many-core model (hardware, management and software) and the corresponding debugging framework.

The *original* contribution of the paper is twofold:

- An architectural model, which comprises a homogeneous many-core, surrounded by input/output peripherals;
- An integration with a rich set of graphical debugging tools, aiming both the hardware (mapping, task scheduling, NoC traffic) and application debugging (individual trace messages for each executing task).

These features are coupled integrated, enabling to trace hardware and software events simultaneously during the system simulation.

II. Many-core Model

The first three subsections describe the hardware, the management, and the application models. The last subsection details the protocol for the admission of new applications into the system.

A. Hardware Model

Figure 1(a) overviews the hardware components. The system is *heterogeneous* because it contains two regions: the General Purpose Processing Cores (GPPC), and the Peripherals. The GPPC includes a set of identical PEs that execute general purpose applications. Peripheral are specialized cores, which provide I/O interface and hardware acceleration for tasks running on GPPC. Peripherals are connected to the boundaries of the GPPC. The connection of peripherals in a NoC-based SoC may occur at any location of the NoC, at external routers, or at unused ports of the mesh NoC (e.g., South ports of bottom routers). We adopted the last option, resulting in a regular floorplan for PEs, with peripherals distributed along the GPPC boundary.

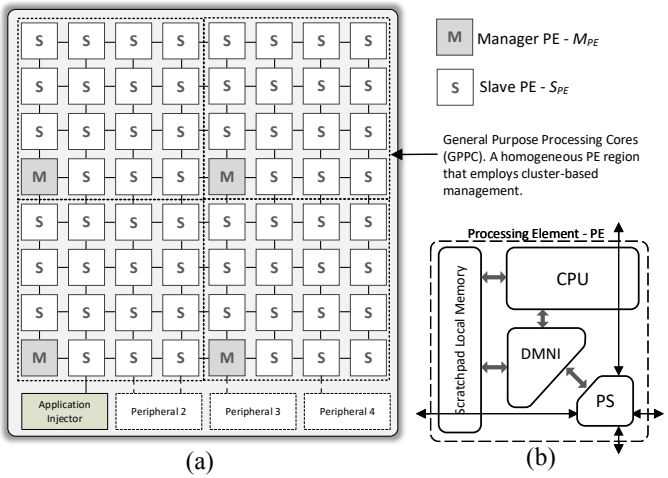

Fig. 1: Overview of hardware model.

Each GPPC PE (Figure 1(b)) contains a CPU, local memory, a NoC router (PS) and a Direct Memory Network Interface (DMNI) [7]. The CPU adopted in this work is the Plasma one (MIPS-like). The DMNI module is a network interface with DMA capabilities [7]. The local memory is a true dual-port scratchpad, storing code and instructions. The goal of using this memory model is to reduce the power consumption related to cache controllers and NoC traffic (transfer of cache lines) [8]. If some application requires a larger memory space than the one available in the local memory, it is possible to have shared memories connected to the system.

Peripherals are connected at the mesh NoC boundary ports. Examples of peripherals include shared memories, accelerators for image processing, communication protocols (e.g., Ethernet, USB), and Application Injectors ($AppInj$). The system requires at least one peripheral, the $AppInj$. This peripheral is responsible for transmitting applications to be executed in the GPPC.

The hardware architecture herein presented is well suited to design systems as Field-Programmable System-On-Chip (FPSoC), which combine the high processing power of many-cores and the reconfigurable logic flexibility of FPGAs, and is in accordance with the new demands of flexibility and computing power of the IoT market[9].

B. Management Model

Scalability at the hardware level comes from PEs executing several tasks in parallel, using the NoC to transmit concurrently multiple flows. However, large systems require high-level management for controlling the deployment of new applications, monitoring resources usage, manage task mapping and migration, and execute self-adaptive actions according to systems constraints (as power cap [10]). Thus, to achieve a scalable design, the architecture adopts cluster-based decentralized management. Clusters are virtual regions in the GPPC, with a set of slave processors (S_{PE}) and one manager PE (M_{PE}). S_{PE}s execute applications' tasks, while M_{PE}s manage the clusters.

The management occurs at the M_{PE} and S_{PE} levels, executed by the operating systems (kernels) running in those PEs. At the M_{PE} level the local memory is reserved to the kernel, without executing user's tasks. The M_{PE} executes heuristics as task mapping, task migration, monitoring, and reclustering. At the S_{PE} level, a multi-task kernel acts as an operating system. This work adopts a paged memory scheme to simplify the kernel design. Examples of actions executed by the kernel include task scheduling, inter-task communication (message passing), deadlines monitoring.

Both kernels are written in C language, easing the portability to other architectures. Only a small part of the code is written in assembly language, responsible for executing context saving and handling hardware and software interruptions.

C. Application Model

Acyclic communication task graphs model the applications, where edges represent communication between tasks, and vertices represent the computation of each task. Tasks use non-blocking $Send()$ and blocking $Receive()$ MPI-like primitives to communicate. The S_{PE} task scheduler supports real-time (RT) and best-effort tasks (BE). RT tasks have constraints: deadline, execution time and period. The adopted task scheduler is the Least Slack Time algorithm [11], which gives higher priority to the task closest to its deadline. BE tasks use the slack time of RT tasks to execute.

D. Dynamic Application Injection Protocol

Applications may start at any moment in the system, characterizing a dynamic workload behavior. To support the dynamic injection of new applications, it is necessary to deploy a protocol enabling the admission of new applications into the system. This subsection details this protocol, which is executed between the $AppInj$ and an M_{PE}. This protocol is generic, and may be deployed by other entities other than the $AppInj$, as an Ethernet core.

Figure 2 depicts the sequence diagram of the protocol. The process begins with $AppInj$ requesting the execution of a new application, by sending a "NEW APP REQUEST" message to an M_{PE} with the application's task number - step **1**. This message is addressed to the cluster zero M_{PE}, which handles this message. Only one M_{PE} handles those requests because it is necessary to have a global knowledge of the resources' usage to select where to execute the new application.

- step **2**. The M_{PE} selects the cluster according to some criterium, sending an "APP ACK" message to $AppInj$, with the M_{PE} address selected to receive the application.
- step **3**. The $AppInj$ sends an "APP DESCRIPTOR" message, with the application task graph in its payload. Upon the reception of this message, the M_{PE} executes the application task mapping.
- step **4**. After task mapping, the M_{PE} sends an "APP ALLOCATION REQUEST" message to the $AppInj$, with the tuples {task ID, address}.
- step **5**. The $AppInj$ transfers the tasks' object code to the S_{PE}s, "TASK ALLOCATION" message, with the task object code in its payload. When a given S_{PE} receive a "TASK ALLOCATION" message, it configures the DMNI to copy the task object code to a selected memory page.

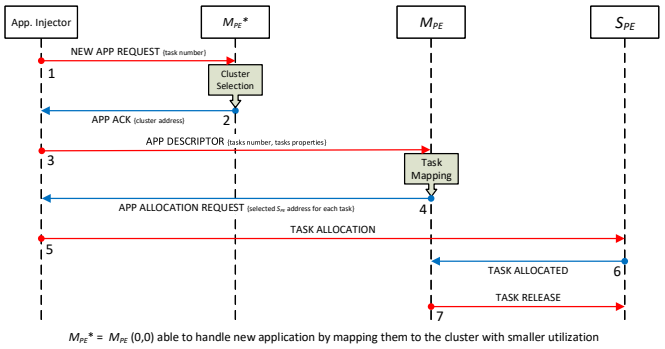

Fig. 2: Sequence diagram of the Dynamic Application Injection Protocol.

- step 6. Once received the task object code, the S_{PE} sends a "TASK ALLOCATED" message to its M_{PE}. Such message is used by the M_{PE} to control when all application tasks were loaded.
- step 7. After receiving an amount of "TASK ALLOCATED" messages equal to the application task number, the M_{PE} releases the application to execute by sending a "TASK RELEASE" message to each S_{PE}.

III. Debugging Framework

Figure 3 overviews the debugging flow. Traditional debugging can be done using GDB, waveforms and log files generated by RTL SystemC or VHDL simulators. Increasing the number of the many-core components, low-level debugging with waveform and logs becomes unfeasible. The framework supports the integration of simulators with an intuitive debugging framework [12], with a set of graphical tools enabling developers to trace high-level system events during simulation.

The many-core description receives a module named Data Extraction Layer (DEL) responsible for capturing communication and computation events. It is important to mention that the DEL is non-intrusive, i.e., it only captures data, storing them in a database and not affecting the applications' performance. DEL captures all packets arriving at any PE local port, storing the packet information in a database, corresponding to the communication events. DEL also captures software events, by sniffing the CPU buses, enabling to trace specific OS and task functions.

While DEL extracts and writes simulation data, the graphical tool reads such information at runtime, converting the raw data into meaningful information represented graphically to the user. In this sense, the debugging tool acts as a graphical interface of the simulated many-core.

The platform developer uses the graphical tools to validate heuristics such as task mapping, routing algorithms, operating system functions (as send and receive primitives). For whom is developing applications, assuming a given platform instance, a specific tool enables to filter the messages per application, allowing to validate parallels applications that use message passing.

This debugging framework is not coupled to this specific platform. The framework requires three configuration files:

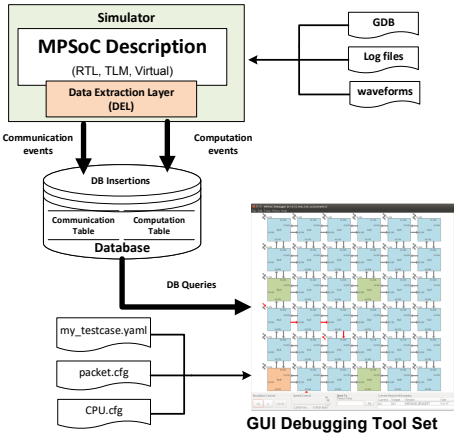

Fig. 3: Debugging flow using a graphical tool with several windows for computation and communication event debugging.

the platform description (*my_testcase.yaml*); the *packet* configuration file with the information related to the packets' services; the *CPU* configuration file with the CPU addresses to monitor and extract the computation events. Each packet has in its payload a service identifier, which corresponds to the action executed by the packet. With this service identifier, it is possible to monitor the operations executed by the messages exchanged between PEs and display them as high-level events.

IV. Experimental Results

This section presents a case study using a many-core with 36 PEs (6x6), with 4 3x3 clusters. Figure 4(a) details the applications' task graphs. The *communication* application is a parallel sort, and *MPEG* implements a pipeline MPEG decoder. Figure 4(b) presents the platform configuration file (hardware). The platform uses the SystemC model, is configured to execute one task per PE, and the M_{PE} is placed at the LB (left-bottom) position of each cluster. Observe that the $AppInj$ (peripheral) is connected at PE 0x1 at the west port. Executing the command *platform-gen 6x6_3x3_sc.yaml* the directory 6x6_3x3_sc is created, with the hardware and kernels compiled.

Figure 4(c) presents the *scenario* file, which lists the applications to execute. Application communication is injected into the system at 1 ms, being statically mapped and application MPEG is injected at 2 ms, being dynamically mapped. These applications are compiled and saved at the 6x6_3x3_sc/application directory by executing *platform-app*

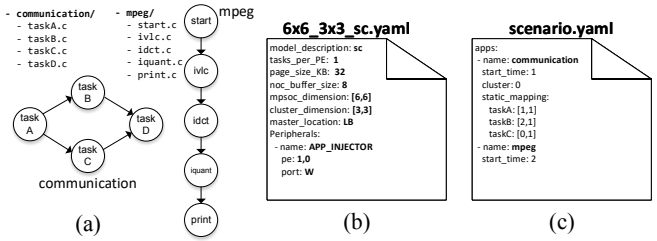

Fig. 4: Applications' task graphs, and configuration files for the platform and the software.

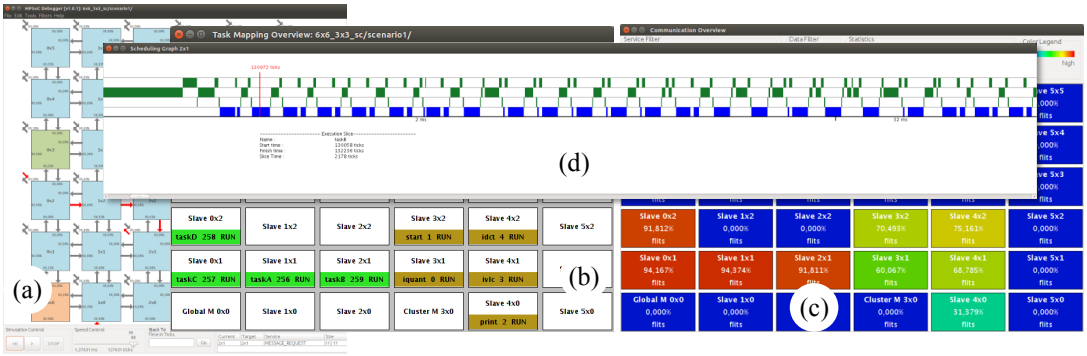

Fig. 5: Windows of the debug framework for testcase 6x6_3x3_sc execution applications of *scenario*. (a) Main view of many-core. (b) Task mapping view. (c) Communication load map view. (d) CPU utilization view.

6x6_3x3_sc.yaml communication mpeg. Finally, the execution of the command *platform-run 6x6_3x3_sc.yaml scenario.yaml 50* starts the platform simulation for 50 ms.

Figure 5 shows a set of windows of the debug framework, where it is possible to observe the system and applications behavior. Figure 5(a) depicts an overview of the many-core, allowing to trace the packets (red arrows), and see the link utilization (inside each router's box), during the simulation. Figure 5(b) shows the mapping of the two tasks of *scenario*, communication in green (statically mapped) and MPEG in brown (dynamically mapped). Each S_{PE} executes one task as specified. The number after the task name is its unique ID assigned by the M_{PE} during application admission. Figure 5(c) shows the communication load map view, where is possible to analyze the communication load distribution due to a color spectrum representation. As expected, the routers where application communication was mapped present a higher communication load. The S_{PE}s where MPEG tasks execute present a lower communication load due to the mixed profile of MPEG related to its time in computation and communication. Finally, Figure 5(d) shows the CPU of S_{PE} 2x1 executing task B. The green slices are related to the kernel execution, and blue slices are related to the task execution. It is possible to observe that taskB have periods of execution and idle periods. The idle periods are due the task is waiting for a message from taskA.

This framework allows to simultaneously debug hardware and software during system simulation using high-level events. This framework is an *original* feature of the work, which simplifies the hardware and software debugging for designers.

V. Conclusion

This work proposed an open-source framework and a many-core model suitable for researchers and parallel applications' developers. This work showed how to build a heterogeneous multi-core, using a homogeneous processing core with distributed management for processing user applications, with a set of peripherals connected at the boundaries of this core. The architecture model considers physical constraints (floorplanning). Architecture models where peripherals can be connected anywhere in the NoC are not feasible to be physically implemented. It is possible to generate only the hardware and keep the same software, allowing the design space exploration of different optimizations at the hardware level. Likewise, it is possible to maintain the same hardware and develop different applications' sets for this particular platform. Thus, the final system is flexible and easily customized by designers.

Future work include: (*i*) developing a peripherals library; (*ii*) prototype the system in FPGAs; (*iii*) make available other processor models (such as RISC-V), adapting the kernels.

Acknowledgement

Fernando Gehm Moraes is supported by FAPERGS (17/2551-0001196-1) and CNPq (302531/2016-5), Brazilian funding agencies. Luciano L. Caimi and Vinicius Fochi are supported by CAPES (184993/2018-00, 184851/2018-00). Marcelo Ruaro is supported by FAPERGS and CAPES (88887.196173/2018-00).

References

[1] A. Monemi, J. W. Tang, M. Palesi, and M. N. Marsono, "ProNoC: A low latency network-on-chip based many-core system-on-chip prototyping platform," *Microprocessors and Microsystems*, vol. 54, pp. 60–74, 2017.

[2] M. A. Elmohr, A. S. Eissa, M. Ibrahim, M. Khamis, S. El-Ashry, A. Shalaby, M. AbdElsalam, and M. W. El-Kharashi, "RVNoC: A Framework for Generating RISC-V NoC-Based MPSoC," in *PDP*, 2018, pp. 617–621.

[3] R. Busseuil, L. Barthe, G. M. Almeida, L. Ost, F. Bruguier, G. Sassatelli, P. Benoit, M. Robert, and L. Torres, "Open-Scale: A Scalable, Open-Source NOC-based MPSoC for Design Space Exploration," in *RECONFIG*, 2011, pp. 357–362.

[4] Q. Zhang, M. Zhou, J. Chen, and H. Yang, "A Homogeneous Many-core x86 Processor Full System Framework Based on NoC," in *ICCSNT*, 2015, pp. 794–797.

[5] J. Balkind *et al.*, "OpenPiton: An Open Source Manycore Research Framework," in *ASPLOS*, 2016, pp. 217–232.

[6] S. Skalicky, A. G. Schmidt, S. Lopez, and M. French, "A unified hardware/software MPSoC system construction and run-time framework," in *DATE*, 2015, pp. 301–304.

[7] M. Ruaro, F. B. Lazzarotto, C. A. Marcon, and F. G. Moraes, "DMNI: A specialized network interface for NoC-based MPSoCs," in *ISCAS*, 2016, pp. 1202–1205.

[8] B. Anuradha and C. Vivekanandan, "Usage of scratchpad memory in embedded systems — State of art," in *ICCCNT*, 2013, pp. 1–5.

[9] R. F. Molanes, K. Amarasinghe, J. Rodriguez-Andina, and M. Manic, "Deep Learning and Reconfigurable Platforms in the Internet of Things: Challenges and Opportunities in Algorithms and Hardware," *IEEE Industrial Electronics Magazine*, vol. 12, no. 2, pp. 36–49, June 2018.

[10] A. M. Rahmani, M.-H. Haghbayan, A. Miele, P. Liljeberg, A. Jantsch, and H. Tenhunen, "Reliability-Aware Runtime Power Management for Many-Core Systems in the Dark Silicon Era," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 2, pp. 427–440, 2017.

[11] J. Liu, *Real-Time System*. Prentice Hall, New Jersey, 2007.

[12] M. Ruaro, H. Chamorra, F. Rubin, A. Amory, and F. G. Moraes, "A data extraction and debugging framework for large-scale MPSoCs," in *ICECS*, 2016, pp. 616–619.