Design Automation for Embedded Systems (2019) 23:103–122
https://doi.org/10.1007/s10617-019-09223-4

Memphis: a framework for heterogeneous many-core SoCs generation and validation

Marcelo Ruaro[1] · **Luciano L. Caimi**[2] · **Vinicius Fochi**[1] · **Fernando G. Moraes**[1]

Received: 9 October 2018 / Accepted: 8 August 2019 / Published online: 20 August 2019

Abstract
This work presents Memphis, which comprises a flexible EDA 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. A set of tools enable a decoupled generation and compilation of the hardware, operating systems, and applications. 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 Memphis well suited to be employed in research and teaching. As case studies, we provide a set of evaluations addressing the many-core generation, simulation, and debugging. Different applications sets were employed, enabling to characterize the computation and communication performance of the many-core, as well as, evaluate an AES encryption application performance according to different levels of parallelism.

Keywords Heterogeneous many-core · NoC · Architecture model · Debug framework

1 Introduction

The design of many-core SoCs became predominant for high-performance circuits, serving as the base hardware platform to meet the heterogeneous computing profile of Internet-

✉ Fernando G. Moraes
fernando.moraes@pucrs.br

Marcelo Ruaro
marcelo.ruaro@acad.pucrs.br

Luciano L. Caimi
lcaimi@uffs.edu.br

Vinicius Fochi
vinicius.fochi@acad.pucrs.br

[1] Pontifical Catholic University of Rio Grande do Sul, Av. Ipiranga, 6681, Porto Alegre, Brazil

[2] UFFS, Av. Fernando Machado 108E, Chapecó, Brazil

of-Things (IoT) and Cyber-Physical Systems [9]. Many-cores 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 PEs.

This design methodology had a positive impact on the chip development process. The efforts are centered on designing and testing a single processing element (PE) and then replicate this consolidated design. The many-core design increasingly became widespread over the semiconductor industry. Such design process can be divided into logical and physical design phases [10]. The 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. Non-functional requirements related to frequency, power, energy, temperature are carefully measured and evaluated according to the design kit provided by the foundry.

While the physical steps have a well-defined design flow provided by CAD tools [10], the many-core logical design is an open field to frameworks aiming design space exploration, automatic system generation, and validation.

The *goal* of this work is to present the Memphis many-core model and framework for automatic generation and validation of many-cores at the logical level. Memphis stands for **M**any-cor**E** **M**odeling **P**latform for **H**eterogenous **S**oCs.

The *original* Memphis' contribution is twofold. The first contribution is a framework for NoC-based MPSoC generation that allows designing a platform model, which comprises a homogeneous many-core region, surrounded by peripherals. The second contribution is the coupled integration of the platform model with debugging tools, enabling to trace hardware and software events simultaneously. This framework, coupled with the debugging tools, enables the design space exploration of many-core architectures. Relevant features of the proposal include:

- A modular logical design flow, allowing to generate the hardware, and compile the operating systems and applications;
- 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 in Memphis).

Memphis is an open-source framework derived from the HeMPS many-core [4]. Both HeMPS and Memphis are available for download at https://www.inf.pucrs.br/hemps/index.html.

This paper is organized as follows. Section 2 reviews frameworks for many-core generation. Section 3 details the hardware, the management, and the application models, as well as the protocol for the admission of new applications. Section 4 describes the Memphis' logical design flow. Section 5 complete the contributions of the paper, with the framework to debug the many-core at runtime. Section 6 presents experimental results using the Memphis framework and the many-core model. Section 7 concludes this paper and point-out direction for future works.

2 Related work

Early works focused on the design space exploration of specific many-core system components. For instance, Xu et al. [29] provide a comparison of communication models, evaluating

Table 1 Related works on many-core/MPSoC design frameworks

Work	Architectural detail level	Peripheral support	RTL validation	Online support	GUI
Xu et al. [29]	NoC	No	Yes	No	No
Cheung et al. [7]	Processor	No	Software	No	No
Monemi et al. [17]	All (NoC, processor, kernel)	As an IP	Verilog	OpenCores	Generation
Elmohr et al. [8]	All (NoC, processor, kernel)	Inside PE	Verilog	No	No
Busseuil et al.[2]	All (NoC, processor, kernel)	No	VHDL	No	No
Zhang et al. [30]	All (NoC, processor, kernel)	No	No	No	No
Balkind et al. [1]	All (NoC, processor, kernel)	Off chip	Verilog	Git, Own site	No
Skalicky et al. [28]	All (NoC, processor, kernel)	No	VHDL	No	No
This work	All (NoC, processor, kernel)	Chip borders	VHDL, SystemC	Git, Own site	Debugging

bus and crossbar infrastructures in an SoC architecture. The work adopts a telecom system simulator named OPNET, which simulates the system at a cycle accuracy level. This work targets only the communication infrastructure. Cheung et al. [7] proposes the INSIDE system with the goal of design space exploration of extensible processors. The work assumes as constraints the area and performance of embedded applications. The framework allows designers to rapidly select the right combination of a processor core and the extensible instruction set. Results show that the design space exploration is equivalent of on average 2% o the design space exploration time in a full simulation of benchmarks, with an average application execution time reduced 2.03 × compared to the base processor core. The work is focused only on the processor context.

Table 1 summarizes related works for generating many-core systems, including the above discussed early works, positioning our work in the last table row.

Monemi et al. [17] propose an automated system integration tool for NoC-based MPSoCs aided by a graphical interface for rapid system configuration and FPGA prototyping. A graphical interface allows easy integration of components as IPs (including peripherals), and the connection between a PE and the NoC. The peripherals can be inserted as a given IP connected to any router of the NoC.

Elmohr et al. [8] present an MPSoC framework based on the RISC-V Instruction Set Architecture and a configurable flit-based router for interconnecting cores. The framework enables a set of configurations at the NoC level (topology, buffer size, routing algorithm), with limited information about the software and application environment control. The environment enables the peripheral connection using the AXI bus within a PE. The system support RTL level debugging. Results present the PE area without peripherals and latency results for distinct NoC configurations.

Busseuil et al. [2] describe a complete Open-Source framework for academic research and development of NoC-based MPSoC called OpenScale. OpenScale also provides simulation and synthesis scripts, which can be used for design space exploration.

Zhang et al. [30] propose a NoC-based homogeneous many-core framework with x86 processor architecture and distributed shared memory. The framework uses the GEMS simulator [14] to memory and processor simulation and the Booksim [12] for NoC simulation. The environment does not run at the RTL level and does not have debugging tools available.

Balkind et al. [1] present an open source, general-purpose, multithreaded, many-core processor and framework, named OpenPiton. The hardware can be synthesized to FPGAs and run an OS and applications. The OpenPiton project is configurable, including core count, cache sizes, and NoC topology, enabling it to adapt to different use cases. The support in OpenPiton for peripherals is only enabled through off-chip connections, using an external NoC (chip bridge). The chip bridge connects the tile array to the chipset. Multiple many-core chips are connected together with chipset logic and networks to build large scalable many-core systems. Thus, OpenPiton assumes a homogeneous on-chip architecture.

Skalicky et al. [28] propose a framework for hardware and software co-design of MPSoCs whit focus on FPGA prototyping. The framework automatically compiles, synthesize and generate heterogeneous systems. The framework is divided into stages to model the processor and hardware cores, connect them to the NoC and peripherals, and set up the application and scheduling. These phases are written using an API where the developer specifies system properties. In the end, a configuration information file is generated as input to scripts that automatically generate the bitstream and binaries files according to the FPGA vendor.

Features of our proposal include: (1) a scalable many-core SoC with a hierarchical organization (i.e., decentralized resource management), allowing the evaluation of large systems (e.g., 16x16); (2) support to the connection with peripherals at the chip borders; (3) an SystemC and 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.

As can be observed, our proposal stands out from other works in two aspects, a framework assuming on-chip peripherals, and a debugging environment that allows simultaneous evaluation of the architecture layers (hardware, operating system, and applications). The proposed organization of the hardware architecture is well suited to design systems as Field-Programmable System-On-Chip (FPSoC), which combines the high processing power of many-cores and the reconfigurable logic flexibility of FPGAs, following the new demands of flexibility and computing power of the IoT market [16].

3 Memphis many-core model

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

3.1 Hardware model

Many-core systems may be classified as: (i) homogeneous and symmetric, where all PEs have the same architecture and organization; (ii) homogeneous and asymmetric [19], where all PEs share the same ISA (instruction-set architecture) but can have different power/performance characteristics (e.g., big.LITTLE); (iii) heterogeneous, with different PEs, as processors and hardware accelerators. The Memphis system falls in the third class. It contains two regions:

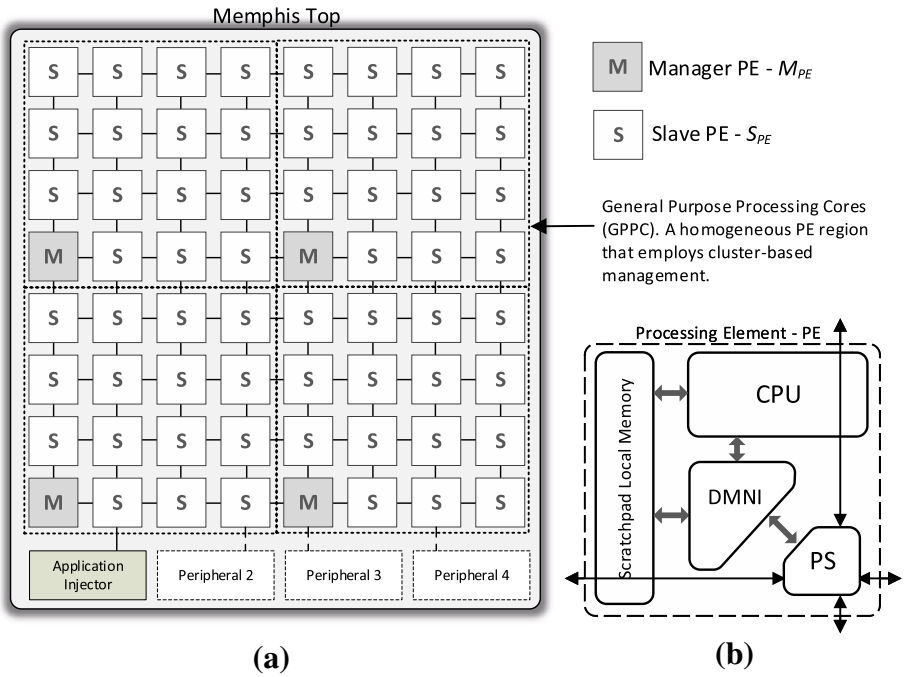

Fig. 1 Overview of the Memphis hardware model, with peripherals connected at the unused south ports of the NoC mesh

the General Purpose Processing Core (GPPC), and peripherals. Figure 1a overviews the hardware components.

The GPPC contains a set of homogeneous and symmetric PEs, which execute general purpose applications. Each PE of the GPPC region (Fig. 1b) contains:

- *Processor* This work adopts the Plasma processor (MIPS-like architecture) [22]. A low integration effort is required to replace this processor with a different architecture. Examples of architectures include RISC-V, ARM, MB-Lite. Besides the connection to the local memory, the processor has a connection to a Direct Memory Network Interface (DMNI), enabling the management of the data transfers using the NoC and memory.
- *Local Memory* The local memory is a true dual-port scratchpad memory, 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). 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, as peripherals.
- *DMNI* (*Direct Memory Network Interface*) The DMNI [24] merge two hardware modules: NI (Network Interface) and DMA (Direct Memory Access). The advantage of the DMNI compared to the traditional PE implementation (NI+DMA) is a specialized module that directly connects the NoC router to the internal memory. The DMNI supports simultaneous packet reception and transmission, managed by a memory access arbiter, which interleaves memory accesses. A programming interface exposes the DMNI services to the software layer.

Table 2 HeMPS and MEMPHIS features

Feature	Hemps [4]	Memphis
CPU	RISC, 32 bits	
Memory	Local scratchpad memory	
NI	Dedicated NI	Direct Memory Network Interface (DMNI) [24]
DMA	Dedicated DMA	
Router	HERMES packet-switching router [18]	
Support to peripherals	No support	At the chip borders, proposed in this work (Sect. 3.1.1)
Hardware models	Clock-cycle accurate RTL (SystemC and VHDL)	
Application Admission Protocol	Ad-hoc, with an external memory with all applications' codes	Well defined protocol, proposed in this work (Sect. 3.4)
Debugging options	Waveforms, log files	Waveforms, log files, and a set of graphical debugging tools
Operating system	Ad-hoc, without a standardization of the services provided by the kernel	Modular, standardization for all services provided by the kernel, task migration [26], hierarchical management [6]

– *NoC router* This work adopts the Hermes NoC [18] as the communication infrastructure—PS (Packet Switching) router in Fig. 1. The main features of the wormhole PS router are: XY routing, round-robin arbitration, input buffering, credit-based flow control.

As stated in the Introduction, Memphis is derived from the HeMPS many-core [4]. Memphis differentiates from HeMPS in several features. Table 2 presents the main features of both platforms.

3.1.1 Peripherals

Peripherals provide I/O interface or hardware acceleration for tasks running on the GPPC region. 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 connection of peripherals in a NoC-based SoC may occur at any location of the system, at buses within PEs (e.g., [8]), as an IP (e.g., [17]), at external routers (as Intel [11]), or at unused ports of the mesh NoC (as Tile-Gx [15]), e.g., south ports of bottom routers. We adopted the last option, peripherals connected at the mesh NoC boundary ports, due to the benefits of regular floorplanning for the GPPC region, easing the physical synthesis, with peripherals distributed along the GPPC boundary.

The NoC router was modified in such a way to enable the communication with boundary ports. Our NoC differentiates *data* packets from *peripheral* packets, as depicted in Fig. 2.

Data packets are those exchanged by tasks running in PEs, and peripheral packets are those transferred between a task and a peripheral. A peripheral packet arriving in a boundary PE

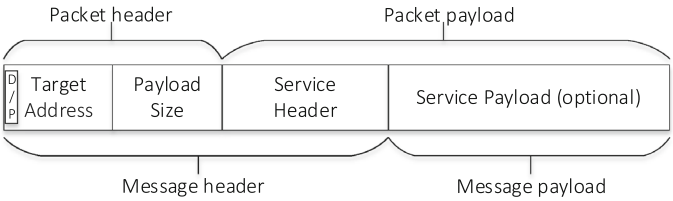

Fig. 2 Packet and message structures—a flag (D/P) in the target address field differentiates *data* packets from *peripheral* packets

goes to the peripheral, and not to the DMNI. A data packet, from the NoC point of view, has a header and a payload. The packet header contains the target router address and the payload size. From the task point of view, a message contains: (i) message header: encapsulates the packet and service header (e.g., message reception, task mapping, request for a message); (ii) message payload: optional field. It may contain, for example, user data or object code of a task.

3.2 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 [27] (as power cap [21]). Thus, to achieve a scalable design, Memphis adopts cluster-based decentralized management [6]. Clusters are virtual regions in the GPPC (depicted by the dotted lines of Fig. 1), with a set of slave processors (S_{PE}) and one manager PE (M_{PE}) [6,20]. 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, as depicted in Fig. 3.

At the M_{PE} level, Fig. 3a, 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. Reclustering is a protocol enabling to modify the cluster shape at runtime. An M_{PE} can borrow resources from other clusters when a given application needs a PE to execute a task, and the cluster S_{PE}s of the application is already full.

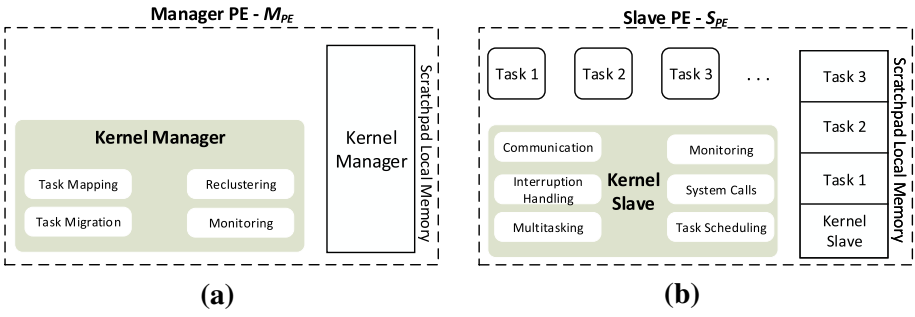

Fig. 3 Overview of the kernels running on Memphis: **a** M_{PE} kernel manages the system and do not execute users' tasks; **b** S_{PE} kernel manage users' tasks

At the S_{PE} level, Fig. 3b, a multi-task kernel acts as a operating system. This work adopts a paged memory scheme to simplify the kernel design. Examples of actions executed by the kernel include multitasking task scheduling, inter-task communication (message passing), deadlines monitoring.

Both manager 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.

3.3 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 [13], which gives higher priority to the task closest to its deadline. BE tasks use the slack time of RT tasks to execute. If a given RT task misses deadlines, the task scheduler generates deadline miss messages to the cluster M_{PE}, which executes a heuristic that can migrate the affected task to an S_{PE} with enough slack time to meet the constraints. Such behavior corresponds to a hybrid scheduler [25], mixing local and global scheduling.

Figure 4 presents the flow to send and to receive a packet between two different PEs. The $Send()$ primitive generates a system call, $send\_packet()$, that programs the DMNI to send the packet, copying the data from memory and transmitting it to the NoC. At the consumer side, when the DMNI receives a packet it interrupts the processor. The interruption handler calls the $read\_packet()$, which programs the DMNI to read the packet copying it from the NoC to memory. Once the packet is completely received, the kernel executes functions related to the contents of the packet. For example, if the packet has data to a user task t, the packet is written in the t memory space, the $Receive()$ call is unlocked, and t is scheduled to execute.

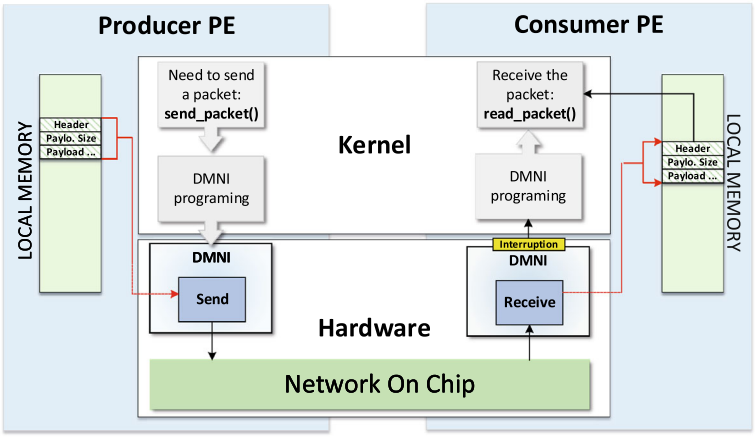

Fig. 4 Inter-PE communication flow [24]

3.4 Dynamic application injection protocol

Applications may start at any moment in Memphis, 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 an $AppInj$ and an M_{PE}. Note that this protocol is generic, and may be implemented by other entities other than an $AppInj$, as an Ethernet core. Figure 5 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 tasks number. The following steps come in the sequence:

- *Step 1* the "NEW APP REQUEST" message is addressed to the cluster zero M_{PE}, which handles this message and selects a cluster to execute the incoming application (this M_{PE} also manages cluster zero). Only one M_{PE} handle those requests because it is necessary to have a global knowledge of the resources' usage to select where to execute the application requesting execution.
- *Step 2* the M_{PE} selects the cluster according to some criterium (e.g., number of available resources or temperature), 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. It may be necessary to execute the reclustering protocol before task mapping if the number of the application tasks is greater than the available free pages in the cluster.
- *Step 4* after task mapping, the M_{PE} sends an "APP ALLOCATION REQUEST" message to the $AppInj$, with the tuples {task ID, S_{PE} address}.
- *Step 5* the $AppInj$ transfers the tasks' object code to the S_{PE}s, sending a "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.

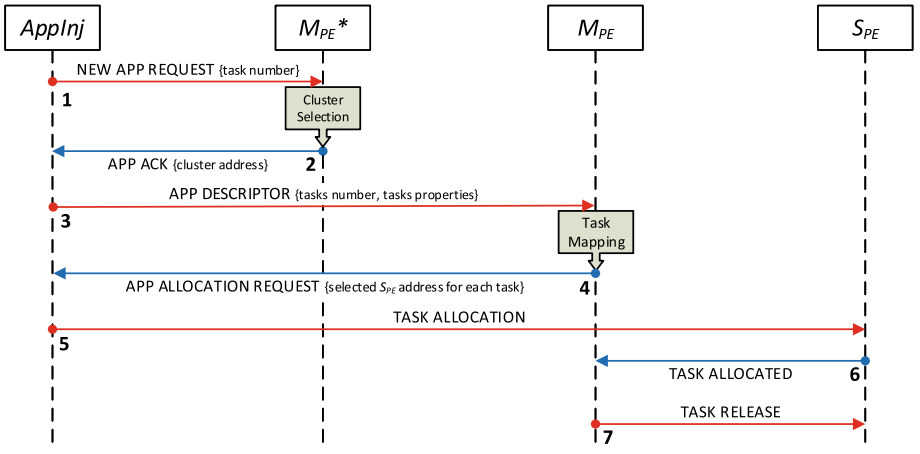

Fig. 5 Sequence diagram of the Dynamic Application Injection Protocol

– *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 RELEAS" message to each S_{PE}.

To illustrate the typical cost of this protocol, consider the admission of the MPEG application with five tasks. This protocol took approximately 40,000 clock cycles from the application request up to its execution release (40 µs in a system running at 1 GHz). Thus, this protocol presents a low overhead to admit new applications.

4 Memphis logical design flow

This Section describes the Memphis' logical design flow, detailed in Fig. 6. This section demonstrates the flexibility of the proposed framework by automatically generating hardware and software for different many-core instances. The logical design flow is divided into steps that comprise the many-core platform modeling (steps 1–5), application modeling (steps 6–8), system execution (step 9), and debugging (step 10), providing a complete design flow to model a many-core design at the logical level.

The Memphis distribution (stored in *MEMPHIS_PATH* location) contains five directories (Fig. 6-2): (i) *applications*, with a standard set of applications previously developed for the system, including benchmarks as MPEG, DTW, DIJKSTRA, AES; (ii) *build_env*, with the scripts and tools required for the many-core generation; (iii) *docs*, with Memphis documentation; (iv) hardware, with the VHDL and SystemC models of each Memphis component; (v) *kernel*, with the operating systems' code.

The framework provides three tools to automatically generate the many-core: *memphis-gen*, to generate the platform; *memphis-app*, to generate applications; *memphis-run*, to call the simulator, to run the many-core, and to open the graphical debugger. These tools read configuration files described in YAML markup language: *testcase* and *scenario* files.

The *testcase* file defines features of the platform (Fig. 6-3). Table 3 details the *testcase* file parameters. The definition of the memory size is a function of the page size and the number of tasks per PE. For instance, for a page size equal to 16 KB, and three tasks per PE, the local memory size is 64 KB (one page for the kernel and three pages for tasks). The *testcase* file also defines the set of peripherals connected to the GPPC, by specifying the peripheral name and the address and port of the border router where the peripheral is connected.

Once defined the platform parameters, the tool *memphis-gen* creates the platform (Fig. 6-4). The *memphis-gen* creates a directory with the *testcase* name, using the location *MEMPHIS_HOME* defined by the user. The *testcase* directory (Fig. 6-5) contains five directories: (i) *applications*, where applications will be stored. This directory is created empty waiting for the insertion of new applications by the user; (ii) *hardware*, where the source code of the hardware directory from *MEMPHIS_PATH* is copied and compiled; (iii) *kernel*, where the source code of the kernel directory is copied and compiled; (iv) *include*, where the include files of hardware and software (based on the testcase parameters) are inserted and linked during the compilation phase; (v) *base_scenario*, where the executable of the compiled platform is placed (for GCC SystemC hardware description), or the simulation scripts (for VHDL hardware description).

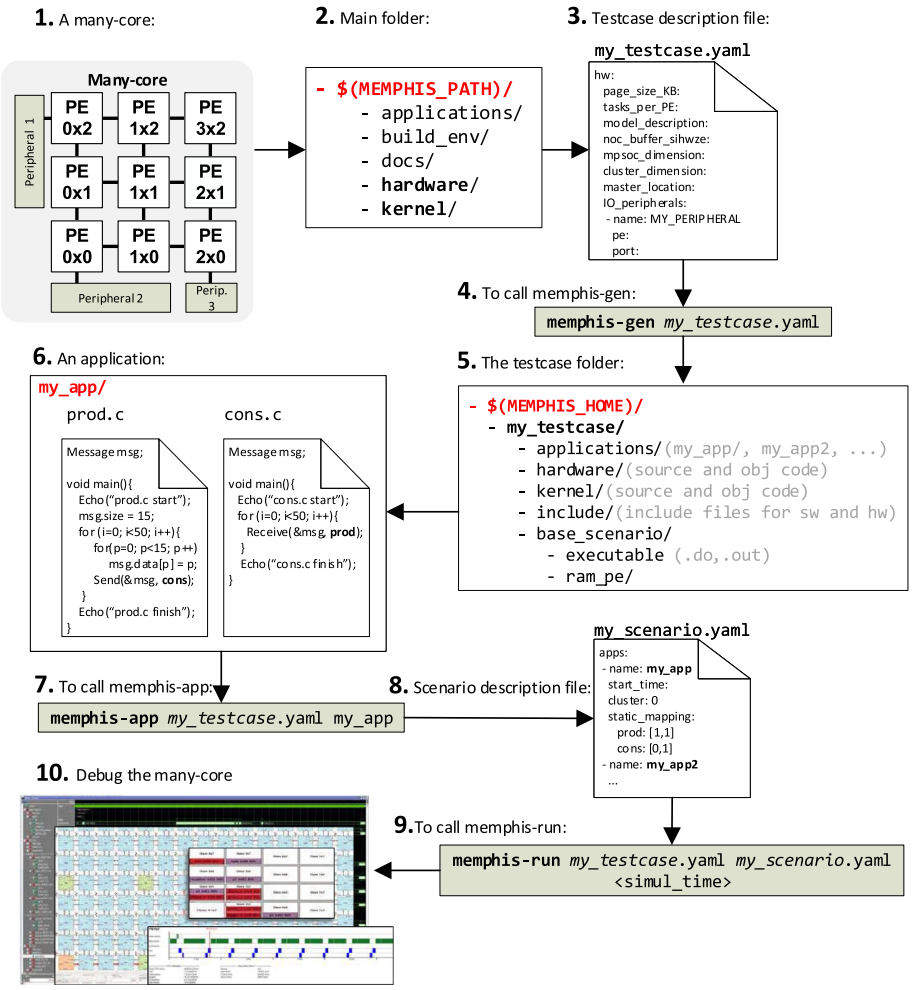

Fig. 6 Memphis' Logical Design Flow

After generating the many-core, the user can compile applications on it. Each application task is implemented as a single C file and must be inserted inside a folder with the name of the application. Figure 6-6 presents the application *my_app*, with two tasks, producer (*prod.c*) and consumer (*cons.c*).

The tool *memphis-app* compiles a given application for a given testcase (Fig. 6-7). The first action of this tool is to search the application passed as the argument in the testcase directory. If the application does not exist, the tool searches it in the *MEMPHIS_PATH* application directory, coping it to the testcase. This mechanism enables to use existing applications as a start point for the development of new ones. Finding the application, the next action is to compile the application and generate an application *descriptor* file. This file contains information related to the application, as the number of tasks, the size of each task object code, and dependences between tasks. The *AppInj* transmits the contents of this file at step 4 of the application injection protocol (Fig. 5). The *memphis-app* tool is invoked for all applications used during the execution of a given *scenario*.

Table 3 Testcase file parameters

Parameter	Description
model_description:	Description language used during generation (sc, scmod, vhdl)
tasks_per_PE:	Maximum number of tasks supported by a PE
page_size_KB:	Size of each S_{PE} page
noc_buffer_size:	Size of each input port buffer on the PS router
mpsoc_dimension:	Size in XY dimension of the many-core
cluster_dimension:	Size in XY dimension of each cluster
manager_location:	Location of the M_{PE} relative to the cluster shape (LB, CC, TR, TL, LB, RB)
Peripherals:	Peripheral parameters
–name:	name identifier (for macro reference)
–pe:	PE address where peripheral is attached
–port:	port of PE where peripheral is attached (N,S,W,E)

Table 4 Scenario file parameters

Parameter	Description
name:	Application name, the same as its directory name
start_time_ms:	Application start time in milliseconds
cluster:	(Optional) Cluster where the application will be statically mapped
static_mapping:	(Optional) Field used to store static mapping information of tasks passing the X and Y addresses where it must be mapped
task1:	
task2:	

An *scenario* is defined as file which contains the set of applications that will execute in Memphis during a simulation. Figure 6-8 shows an example of a *scenario* file. For each application there are a set of optional parameters, as: (i) *start time*, defines when $AppInj$ will make the "NEW APP REQUEST"; (ii) *mapping type*, static or dynamic [5]. Table 4 presents the scenario file parameters and its description.

With the platform and applications already compiled, it is possible to simulate the platform (Fig. 6-9), with the tool *memphis-run*. *Memphis-run* invokes a simulator and the graphical debugger. The execution of the many-core by *memphis-run* impacts in the creation of a scenario directory inside the testcase directory, where are placed all log and debugging files generated during execution. The *memphis-run* tool calls different simulators depending upon the system model description. If the model is GCC SystemC, the tools just run the executable file resulted from the platform compilation. Otherwise, the tool will open the VHDL simulator (e.g., ModelSim). In parallel to the system execution, the tool also calls the GUI debugger. The debugger provides a set of tools (Fig. 6-10) that read logs generated during the simulation,

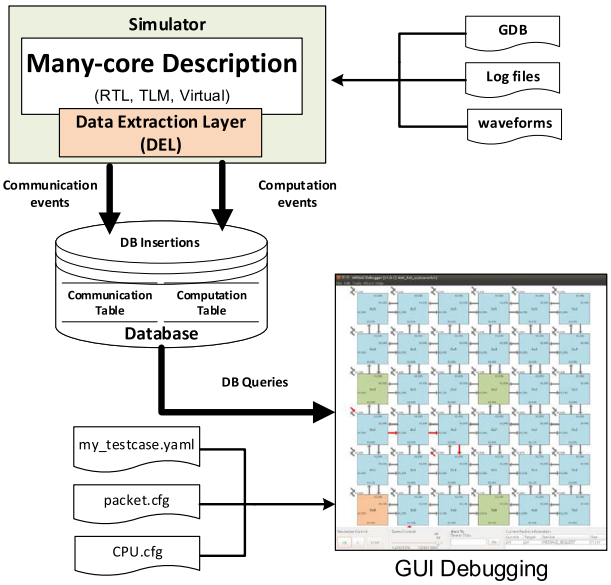

Fig. 7 Memphis' Debugging Flow using a graphical tool with several windows for computation and communication event debugging

process the data, and show graphical information, providing high-level debug facilities. The debugging tools are further explained in Sect. 5.

An important feature of the presented flow is the separation between the platform and the application generation. Different scenarios may run on the same platform, enabling, for example, evaluate the performance of different applications for the same platform configuration, or evaluate the effectiveness of mapping heuristics assuming a different applications set.

5 Memphis debugging

Figure 7 overviews the Memphis debugging flow. Traditional debugging can be done through GDB, waveforms, or/and manually observing raw data of log files generated by simulators. Increasing the number of the many-core components, low-level debugging with waveforms and logs becomes unfeasible. Memphis supports the integration of simulators with an intuitive debugging framework [23], composed by a Data Extraction Layer (DEL) and 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 that is graphically represented

to the user. In this sense, the debugging tool acts as a graphical interface, with several windows, which details the behavior of the simulated many-core.

At the communication level, the main window provides an overview of each router and interconnection, displaying the percentage of the links' utilization, corresponding to the consumed bandwidth. When a packet travels from a source PE to a target PE, the path taken by the packet is highlighted, allowing to validate, e.g., routing algorithms and to detected deadlocks [3]. Another window shows in a color spectrum the communication load on each router since the beginning of the simulation. The communication load can be filtered according to the packet service. Another window shows the task mapping, displaying applications with different colors, and for each task its name. It is also possible to filter the running and finished applications.

At the computation level, the tool provides a window detailing the CPU scheduling for each task. Thus it is possible to verify when the OS and tasks are running, or when an interruption is handled. This set of tools enables concurrent hardware and software debugging. For example, inserting breakpoints at the interruption handler, it is possible to trace the packets that generate the interruption event.

The platform developer can use 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 with this specific platform. The framework requires three configuration files: 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.

6 Experimental results

This Section presents experimental results using the Memphis framework and the many-core model previously presented.

6.1 Generic case-study

This subsection presents a case-study showing a scenario with two applications (communication and MPEG), and a many-core with 36 PEs (6 × 6), with 4 clusters (3 × 3 each one). Figure 8a details the applications' task graphs. The communication application is a parallel sort, and MPEG implements a pipeline MPEG decoder. Figure 8b presents the scenario file. The $AppInj$ is connected at PE 0 × 1 at the west port. The platform is configured to execute one task per PE, and the M_{PE} is placed at the LB (left-bottom) position of each cluster.

Executing the command *memphis-gen* 6 × 6_3 × 3_*sc.yaml* the directory 6 × 6_3 × 3_sc is created, with file structure presented in Fig. 6. After generating the platform, applications are compiled and saved at the 6 × 6_3 × 3_sc/application directory, by executing *memphis-app* 6 × 6_3 × 3_*sc.yaml communication mpeg*.

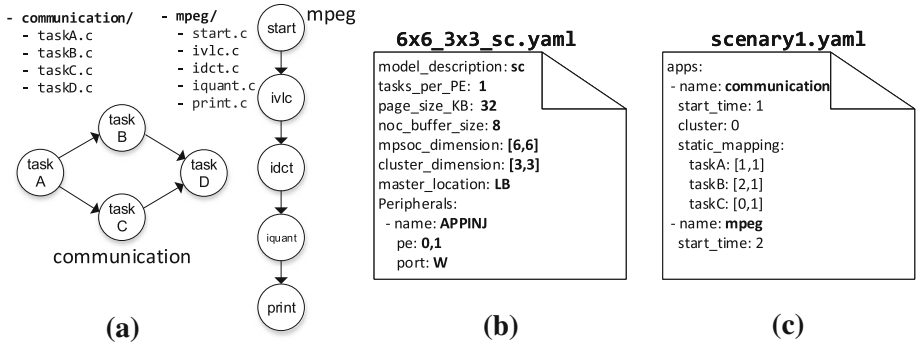

Fig. 8 Applications' task graphs, and configuration files for the platform and the software

Figure 8c presents *scenario1*, which lists the applications and its properties. Application communication is injected into the system at 1 ms, being statically mapped at cluster 0 (each cluster is identified with a unique number), with tasks A, B and C statically mapped on S_{PE} 1×1, 2×1, and 0×1, respectively. Application MPEG will be injected at 2 ms, being dynamically mapped.

Finally, the execution of the command *memphis-run* $6 \times 6\_3 \times 3\_sc.yaml$ *scenario1. yaml 50* starts the platform simulation, wich will execute for 50 ms.

Figure 9 shows a set of windows of the debug framework, where it is possible to observe the system and applications behavior. Figure 9a depicts an overview of the many-core, allowing to trace the packets and see the link utilization during the simulation. Figure 9b 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 9c 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 9d shows the CPU of S_{PE} 2×1 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.

6.2 Computation and communication performance

The goal of this subsection is to evaluate communication and computation performance in such a way to determine the offered bandwidth by the NoC and the execution time of simple benchmarks.

The NoC adopts 32-bit flits, running at 100 MHz, resulting in a bandwidth per link equal to 3.2 Gbps. To evaluate the amount of bandwidth a given application may consume, we run a producer-consumer application (with two tasks, *prod* and *cons*), with each task mapped at different processors. The *prod* task sends messages to the *cons* task, without any other computation. Since this application has a communication-intensive profile, it is possible to

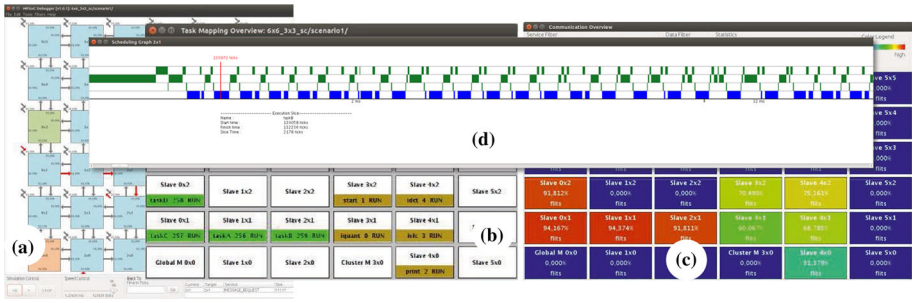

Fig. 9 Windows extracted from Debugging Tool for testcase 6 × 6_3 × 3_sc in scenario1. **a** Main view of many-core, **b** task mapping view, **c** communication load map view, **d** CPU utilization view

measure the injection rate (percentage of the link bandwidth utilization) and the bandwidth at the application level, considering the software stack.

Each message exchanged between tasks corresponds to a packet injected into the NoC. We run different scenarios changing the message size at the application level, which results in different packet sizes. For packet sizes varying from 40 to 140 flits the average injection rate was 17.4% (14.62% up to 19.4%) of the NoC bandwidth, corresponding in an average throughput of 556.64 Mbps. The reason explaining this throughput comes from the fact that the software must treat the packet at both the producer and consumer sides. The producer kernel copies the message to a memory region before sending the message, enabling the producer task to continues its execution without waiting for a transmission request (*non-blocking send*). At the consumer kernel side, the task first sends a request asking data, and when the packet arrives at the consumer task S_{PE}, the DMNI generates an interruption for the kernel to handle the packet (the kernel must stop a given task execution, save its context, and program the DMNI to read the packet). The kernel programs the DMNI twice, one to read the packet service to determine the packet function, and the other one to read the packet payload (as depicted in Fig. 2), transferring the packet data to the corresponding memory region. If the packet data carries an application task message, the kernel copies the payload to the corresponding consumer task page. The time for the kernel to handle a packet is, on average, 200 clock cycles for packet carrying application task messages.

It is important to differentiate the network throughput from the application throughput. At the network level, the throughput ranges from 467 to 620 Mbps, for packets from 40 to 120 flits, respectively. At the network level, the transmission is blocked for some periods because the kernel needs to treat the incoming packet. Thus, the network throughput is not proportional to the packet size because it is necessary to add idle periods during the packet transmission. At the application level, the throughput ranges from 72.56 (40-flit packets) to 213.23 Mbps (140-flit packets). The smaller application throughput is due to the protocol stack related to the communication protocol. With small packets, the overhead of the communication protocol is more pronounced than for large packets.

This average network bandwidth for one application, 17.4%, cannot be considered as a NoC sub-utilization. In a many-core system, parallel flows are frequent, and such bandwidth utilization allows avoiding congestion effects.

The computation evaluation used five sorting algorithms (selection, insertion, bubble, quick, and merge), implemented each one as one task. Each task was mapped alone in a free PE. The execution time for each sorting algorithm assuming an array of 1000 elements sorted in reverse order and without repeated values. Figure 10 presents the execution time in

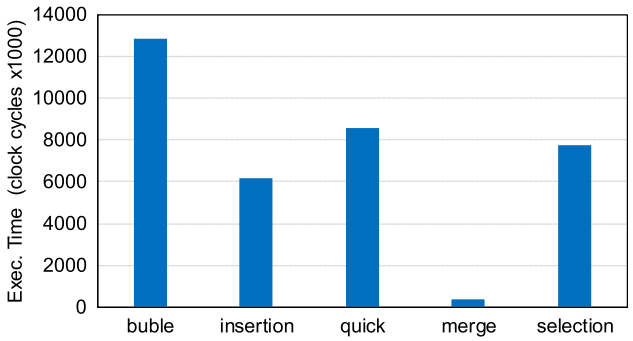

Fig. 10 Performance results of sort algorithms executing in Memphis

thousands of clock cycles. The *merge* sort is faster due to its complexity – $\mathcal{O}(n \log n)$, while the other algorithms have a complexity $\mathcal{O}(n^2)$. These values reflect the performance of the current CPU adopted in Memphis, the Plasma CPU [22].

Both communication and computation results can be used in future works as benchmarks to evaluate changes in many-core architecture.

6.3 Hardware and software overheads

This subsection evaluates the time spent by the software (kernel) and the hardware (DMNI and NoC) to exchange messages, varying the message size and the number of hops between tasks. The goal is to evaluate the behavior of the architecture, considering the components' stack involved in the transaction.

Figure 11 shows the time spent by each component. The largest overhead is in the operating system (kernel). The *send_packet* function invokes a system call that: (i) transfers the message contents from the task memory space to the kernel area; (ii) creates the packet structure (packetization process); and (iii) programs the DMNI. Next, the packet is injected into the network, with a small overhead due to a dedicated memory port that enables the injection of one flit per clock cycle (this overhead may increase in scenarios with congestion in the NoC). Finally, at the receiver side occurs the reverse process, with the message transferred to the task memory space.

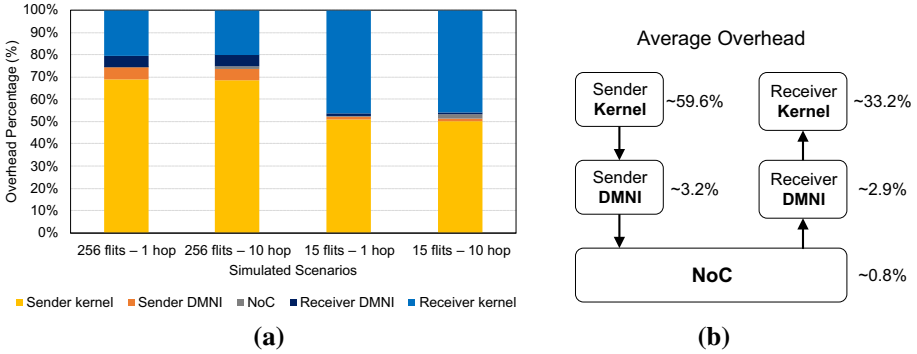

Fig. 11 **a** Hardware and software overheads of the simulated scenarios, **b** average overheads

Although the software is responsible for the most significant overhead in the communication process, the total time spent to transfer the packets varies between 2082 (1 hop, 15 flits) and 5256 clock cycles (10 hops, 256 flits). These values show the high-performance of the message exchanging mechanism. The NoC should present a low overhead in such a way to minimize the variation in the latency due to congestion effects.

6.4 Multitasking performance: AES encryption application

This subsection evaluates a real application, AES encryption, by partitioning it into a variable number of tasks distributed across multiple processors. The primary goal of this session is to

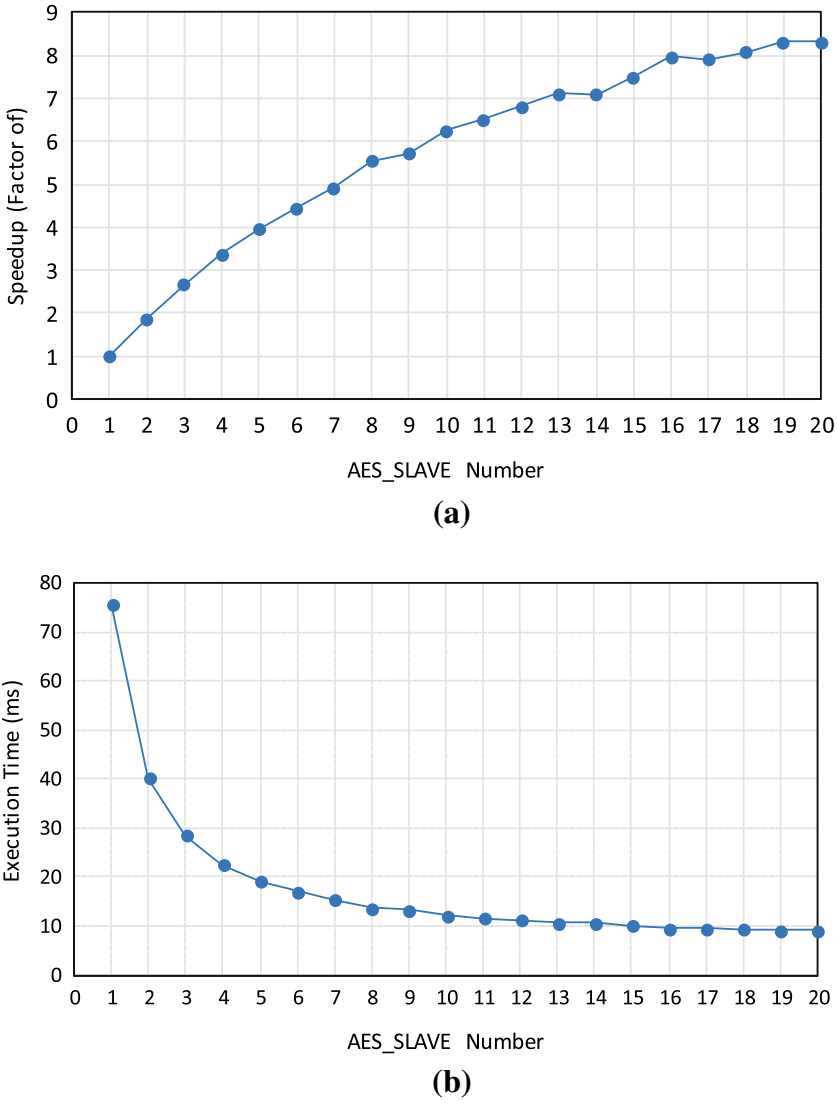

Fig. 12 AES Decryption Application Performance. **a** Speedup over a single-core execution, **b** execution time assuming different parallelism levels

demonstrate the platform's ability to run parallel applications using the message exchange paradigm.

The AES application contains one master task and a set of slave tasks (the number of slaves is parameterizable at design time). Each slave task is in charge to encrypt a data array block. The master splits the data array equally according to the number of slaves sending one block for each slave. Figure 12a shows the speedup results w.r.t the number of slaves (from 1 up to 20 slaves). The speedup follows a logarithm curve. The speedup difference is higher in executions with a lower number of slaves since the amount of the problem to be divided is higher at the beginning of the parallelization and decreases according to more slaves are used to work at the same problem. Figure 12b shows the execution time reduction according to the number of slaves increases. As well as in the speedup graph, the gains in execution time are higher for a lower number of slaves, execution time start to stabilize after eight slaves showing a trade-off between execution time improvement and resource utilization.

7 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 divided into two regions: a homogeneous processing core with distributed management for processing user applications, and a set of peripherals connected at the boundaries of this core. Such architecture model considers physical constraints (floorplanning). Architecture models where peripherals can be connected anywhere in the NoC are not feasible to be physically implemented. Associated with this model, this work also presented the hardware and software generation flow. 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).

Acknowledgements This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. 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 CAPES/FAPERGS (88887.196173/2018-00).

References

1. Balkind J, McKeown M, Fu Y, Nguyen T, Zhou Y, Lavrov A, Shahrad M, Fuchs A, Payne S, Liang X, Matl M, Wentzlaff D (2016) OpenPiton: An open source manycore research framework. In: ASPLOS, pp 217–232. https://doi.org/10.1145/2954679.2872414
2. Busseuil R, Barthe L, Almeida GM, Ost L, Bruguier F, Sassatelli G, Benoit P, Robert M, Torres L (2011) Open-scale: a scalable, open-source NOC-based MPSoC for design space exploration. In: RECONFIG, pp 357–362. https://doi.org/10.1109/ReConFig.2011.66
3. Carara E, Moraes FG (2008) Deadlock-free multicast routing algorithm for wormhole-switched mesh networks-on-chip. In: ISVLSI, pp 341–346. https://doi.org/10.1109/ISVLSI.2008.18
4. Carara EA, de Oliveira RP, Calazans NLV, Moraes FG (2009) HeMPS—a framework for NoC-based MPSoC generation. In: ISCAS, pp 1345–1348. https://doi.org/10.1109/ISCAS.2009.5118013
5. Carvalho E, Marcon C, Calazans N, Moraes F (2009) Evaluation of static and dynamic task mapping algorithms in NoC-based MPSoCs. In: SOC, pp 87–90. https://doi.org/10.1109/SOCC.2009.5335672

6. Castilhos G, Mandelli M, Madalozzo G, Moraes F (2013) Distributed resource management in NoC-based MPSoCs with dynamic cluster sizes. In: ISVLSI, pp 153–158. https://doi.org/10.1109/ISVLSI.2013.6654651
7. Cheung N, Parameswaran S, Henkel J (2003) INSIDE: instruction selection/identification & design exploration for extensible processors. In: ICCAD, pp 291–297. https://doi.org/10.1109/ICCAD.2003.159703
8. Elmohr MA, Eissa AS, Ibrahim M, Khamis M, El-Ashry S, Shalaby A, AbdElsalam M, El-Kharashi MW (2018) RVNoC: a framework for generating RISC-V NoC-based MPSoC. In: PDP, pp 617–621. https://doi.org/10.1109/PDP2018.2018.00103
9. Hassan M (2018) Heterogeneous MPSoCs for mixed-criticality systems: challenges and opportunities. IEEE Des Test 35(4):47–55. https://doi.org/10.1109/MDAT.2017.2771447
10. Iniewski K (2012) Embedded systems: hardware, design, and implementation, 1st edn. Wiley, Hoboken
11. Intel: Intel's New Mesh Architecture: The 'Superhighway' of the Data Center (2018). https://itpeernetwork.intel.com/intel-mesh-architecture-data-center/
12. Jiang N, Becker DU, Michelogiannakis G, Balfour J, Towles B, Shaw DE, Kim J, Dally WJ (2013) A detailed and flexible cycle-accurate Network-on-Chip simulator. In: ISPASS, pp 86–96. https://doi.org/10.1109/ISPASS.2013.6557149
13. Liu J (2000) Real-time system. Prentice Hall, Englewood Cliffs
14. Martin MMK, Sorin DJ, Beckmann BM, Marty MR, Xu M, Alameldeen AR, Moore KE, Hill MD, Wood DA (2005) Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset. SIGARCH Comput Arch News 33(4):92–99. https://doi.org/10.1145/1105734.1105747
15. Mellanox: TILE-Gx Processors (2018) http://www.mellanox.com/repository/solutions/tile-scm/docs/UG130-ArchOverview-TILE-Gx.pdf
16. Molanes RF, Amarasinghe K, Rodriguez-Andina J, Manic M (2018) Deep learning and reconfigurable platforms in the internet of things: challenges and opportunities in algorithms and hardware. IEEE Ind Electron Mag 12(2):36–49. https://doi.org/10.1109/MIE.2018.2824843
17. Monemi A, Tang J, Palesi M, Marsono M (2017) ProNoC: a low latency network-on-chip based many-core system-on-chip prototyping platform. Microprocess Microsyst 54:60–74. https://doi.org/10.1016/j.micpro.2017.08.007
18. Moraes FG, Calazans N, Mello A, Möller L, Ost L (2004) HERMES: an infrastructure for low area overhead packet-switching networks on chip. Integration 38(1):69–93. https://doi.org/10.1016/j.vlsi.2004.03.003
19. Muthukaruppan TS, Pricopi M, Venkataramani V, Mitra T, Vishin S (2013) Hierarchical power management for asymmetric multi-core in dark silicon era. In: DAC, pp 1–9. https://doi.org/10.1145/2463209.2488949
20. Quan W, Pimentel AD (2016) A hierarchical run-time adaptive resource allocation framework for large-scale MPSoC systems. Des Autom Embed Syst 20(4):311–339. https://doi.org/10.1007/s10617-016-9179-z
21. Rahmani AM, Haghbayan MH, Miele A, Liljeberg P, Jantsch A, Tenhunen H (2017) Reliability-aware runtime power management for many-core systems in the dark silicon era. IEEE Trans Very Large Scale Integr (VLSI) Syst 25(2):427–440. https://doi.org/10.1109/TVLSI.2016.2591798
22. Rhoads S (2016) Plasma CPU Core. https://opencores.org/project/plasma
23. Ruaro M, Chamorra H, Rubin F, Amory A, Moraes FG (2016) A data extraction and debugging framework for large-scale MPSoCs. In: ICECS, pp 616–619. https://doi.org/10.1109/ICECS.2016.7841277
24. Ruaro M, Lazzarotto FB, Marcon CA, Moraes FG (2016) DMNI: a specialized network interface for NoC-based MPSoCs. In: ISCAS, pp 1202–1205. https://doi.org/10.1109/ISCAS.2016.7527462
25. Ruaro M, Moraes FG (2016) Dynamic real-time scheduler for large-scale MPSoCs. In: GLSVLSI, pp 341–346. https://doi.org/10.1145/2902961.2903027
26. Ruaro M, Moraes FG (2017) Demystifying the cost of task migration in distributed memory many-core systems. In: ISCAS, pp 1–4. https://doi.org/10.1109/ISCAS.2017.8050257
27. Singh AK, Shafique M, Kumar A, Henkel J (2013) Mapping on multi/many-core systems: survey of current and emerging trends. In: DAC, pp 1–10. https://doi.org/10.1145/2463209.2488734
28. Skalicky S, Schmidt AG, Lopez S, French M (2015) A unified hardware/software MPSoC system construction and run-time framework. In: DATE, pp 301–304. https://doi.org/10.7873/DATE.2015.0097
29. Xu J, Wolf W, Henkel J, Chakradhar S, Lv T (2004) A case study in networks-on-chip design for embedded video. In: DATE, pp 770–775. https://doi.org/10.1109/DATE.2004.1268973
30. Zhang Q, Zhou M, Chen J, Yang H (2015) A homogeneous many-core x86 processor full system framework based on NoC. In: ICCSNT, pp 794–797. https://doi.org/10.1109/ICCSNT.2015.7490861