An Open-source Framework for Heterogeneous MPSoC Generation

Eduardo W. Wächter, Carlo Lucas, Everton A. Carara, Fernando G. Moraes
FACIN - PUCRS - Av. Ipiranga 6681- Porto Alegre - 90619-900 – Brazil
eduardo.wachter@acad.pucrs.br, carlo.lucas@acad.pucrs.br, everton.carara@pucrs.br, fernando.moraes@pucrs.br

Abstract — The design of a Multiprocessor System-on-Chip (MPSoC) is a complex task, including steps as application development, platform configuration, code generation, task mapping onto the platform and debugging. An integrated environment covering most of these steps is a gap in the literature. The present work first details an MPSoC architecture, which supports the execution of distributed applications, including an operating system enabling multitask execution at each processing element. The MPSoC is heterogeneous, due to the support to different processor architectures. Then, a framework able to cover the design steps previously mentioned is presented. The framework enables the design space exploration for applications to be executed in the MPSoC, varying for example the number and type of processors, the memory size, the task mapping. Results demonstrate the correct operation for different MPSoC configurations, generated from the proposed framework. Such open-source framework enables the research community to investigate new subjects related to MPSoC and Network on Chip (NoC) design, as well as evaluate distributed applications in a multiprocessor environment. *(Abstract)*

Key words: *MPSoC, NoC, CAD tools, prototyping*.

I. INTRODUCTION

The use of NoC-based MPSoCs is a clear trend in the semiconductor industry, since these systems enable complex products design, coping with performance and power budgets, as well as with tight time-to-market constraints. The inherent complexity of MPSoCs requires dedicate frameworks to assist engineers to develop such systems. Examples of commercial bus-based MPSoC frameworks include Xilix EDK [1] and SoPC Builder by Altera [2].

It is possible to identify 5 major steps during the design of an MPSoC: (*i*) application development; (*ii*) platform configuration; (*iii*) code generation; (*iv*) application mapping onto the platform; (*v*) debugging. Most MPSoCs works focus on one such steps, for example platform configuration or application mapping. An integrated environment covering most of these steps is a gap in the literature.

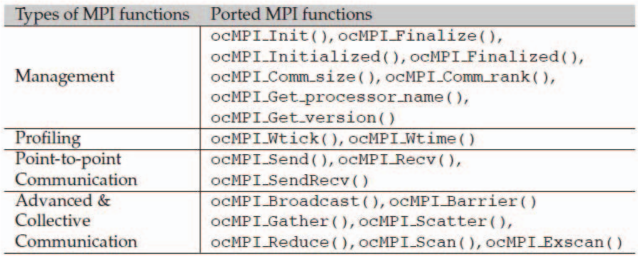
Our work presents an open-source framework for heterogeneous MPSoC generation. Applications are described as task graphs, and it is possible to assign to each task a specific processor architecture. The platform is based in the HeMPS MPSoC [3], supporting two different architectures: Plasma (MIPS architecture) and MBLite (Microblaze architecture). Each processor executes the same tiny operating system, named *microkernel*, customized according to the processor architecture. Static and dynamic mapping is supported [4], enabling to load applications at design and execution time. A dedicated GUI enables to debug the system, displaying the execution results of each task.

The *contributions* of this paper are twofold. The first one is to provide details of the platform, including the heterogeneous processing element, the parameterizable microkernel and application modeling. The second contribution is to present the framework responsible to cover the macro design steps presented above.

This paper is organized as follows. Section II reviews related works for MPSoC generation tools. Section III details the proposed MPSoC architecture. Section IV presents the open-source framework responsible to generate instances of the proposed MPSC. Section V present results, and section VI concludes this paper.

II. RELATED WORKS

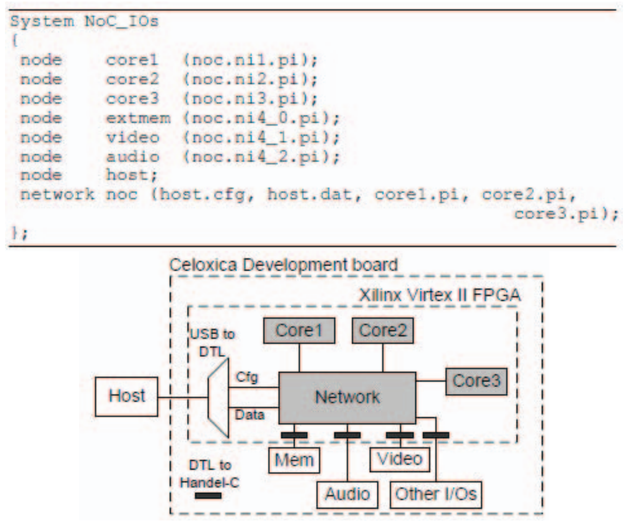
xENoC [5] is an environment for the automated generation of NoC-based MPSoCs. It is based on *NoCWizard* tool, which allows the generation of NoCs (described in Verilog RTL) from the specification of various parameters such as topology, flow control, switching mode and routing algorithm. The environment has an IP library with different processors and accelerators, which enables the generation of heterogeneous MPSoCs. The whole system is described in an XML file (NoC characteristics, and IP mapping), which is used as input for the generation tools. Besides the hardware infrastructure, xENoC also includes a software library for messaging and synchronization between tasks named ocMPI (on-chip MPI) [6]. This library is an embedded version of the MPI standard, and it is independent of the operating system. MPI primitives supported are listed in Figure 1. Depending on the application requirements, only a subset of the primitives needed to be included. The amount of memory needed to store the library can vary from 4942 bytes (basic primitives) to 13258 bytes (complete set). Since the implemented NoC only supports unicast transmission, collective communication services as broadcast (*ocMPI_Broadcast* ()) are implemented from the basic primitives *ocMPI_Send*() and *ocMPI_Recv*(). Experiments on a 2x2 platform instance report speed-ups next to the number of processors, for applications with high degree of data interdependence.



Types of MPI functions	Ported MPI functions
Management	ocMPI_Init(), ocMPI_Finalize(), ocMPI_Initialized(), ocMPI_Finalized(), ocMPI_Comm_size(), ocMPI_Comm_rank(), ocMPI_Get_processor_name(), ocMPI_Get_version()
Profiling	ocMPI_Wtick(), ocMPI_Wtime()
Point-to-point Communication	ocMPI_Send(), ocMPI_Recv(), ocMPI_SendRecv()
Advanced & Collective Communication	ocMPI_Broadcast(), ocMPI_Barrier() ocMPI_Gather(), ocMPI_Scatter(), ocMPI_Reduce(), ocMPI_Scan(), ocMPI_Exscan()

Figure 1 - MPI primitives supported in the ocMPI library.

Kumar et al. [7] propose an integrated design flow for MPSoC generation, targeting FPGA devices. The architecture is based on the Silicon Hive processor [8] and the Aethereal NoC [9]. The input to the generation tool is a specification file that describes the MPSoC. The tool generates a RTL VHDL description of the system and simulation models for each of its components. Figure 2 shows part of one specification file, with the corresponding architecture. The host (Figure 2) is the system master, and can be a computer or an embedded processor. In addition to debugging the system, its main functions include the load of the object code of the tasks in the local memories of processors and configure the NoC through the establishment of connections between communicating IPs. The communication service based on connections implemented by Aethereal allows the specification of connection parameters (e.g. bandwidth and maximum latency) at design time. Since the host is responsible for establishing connections, these are transparent to the IPs. Communication is performed through memory mapped input/output registers. To validate the flow, two system configurations were generated. The first consisting of three processors (one being the host) connected to a router running a single producer/consumer application. In the second the host is an external computer and the architecture is similar to Figure 2, using some of the prototyping board components, as memory and audio/video. For the second system, various network topologies were used, from a single router to a 2x2 mesh.



```
System NoC_IOs
(
  node    core1  (noc.ni1.pi);
  node    core2  (noc.ni2.pi);
  node    core3  (noc.ni3.pi);
  node    extmem (noc.ni4_0.pi);
  node    video  (noc.ni4_1.pi);
  node    audio  (noc.ni4_2.pi);
  node    host;
  network noc (host.cfg, host.dat, core1.pi, core2.pi,
                                               core3.pi);
);
```

Figure 2 - Example of system proposed in [7], including a partial system specification file and the corresponding architecture.

Singh et al. [10] present a design flow for NoCs-based MPSoCs, also targeting FPGAs devices. Initially, the NoC is generated, using the tool *NoC Generator* [11]. This tool allows the generation of NoCs with guaranteed throughput, using spatial division multiplexing. Next, MicroBlaze processor-based PEs are connected to the network interfaces through FSL (Fast Simplex Link) ports, completing the generation of the hardware infrastructure. Communication between PEs is based on connections, which are established by a processor that controls the communication of the MPSoC. The connections are generated at design time by the *NoC Generator*, and depend on the communication requirements of applications to be executed. The processor responsible for the MPSoC control stores these connections, and establishes connections before starting the execution of applications. Experiments carried out in a 3x3 platform instance evaluate only the connections establishment time, which the authors claim to be very low. The authors report a time of 2885 clock cycles, while the proposed platform (Section III) takes only 500 clock cycles.

Alonso et al. [12] present a methodology that helps designers to build NoC-based MPSoCs, develop applications and evaluate performance. The processing elements (PEs) are created using the tool *Altera SOPC Builder* [2], which allows selection of various peripherals around the NIOS II processor. The NoC is created using the tool *NoCMaker* [13], which offers a simulation environment where it is possible to evaluate performance, power consumption and area. As a case study, the Authors created an MPSoC with15 slaves and 1 master PEs, interconnected by a 2-D mesh NoC. The master PE has exclusive access to a 16MB off-chip SDRAM, and is responsible to control the slave PEs. The used NoC adopts wormhole packet switching, handshake flow control, and offers only basic communication services. For message exchange between tasks it is used the ocMPI library.

This set of recent environments to generate NoC-based MPSoCs support heterogeneous architectures, but there is a lack of automation, including not only the platform generation, but also the software development/compilation, simulation/prototyping and a debug interface enabling designers to evaluate the applications performance. This is the *goal* of our work: an extension of the previous homogeneous MPSoC framework to present the MPSoC infrastructure and the framework supporting generation and debugging of the heterogeneous MPSoC.

III. HETEROGENEOUS ARCHITECTURE

This section describes the MPSoC heterogeneous architecture. The features of the reference MPSoC, HeMPS [3], includes: homogeneous architecture, supporting only Plasma Processors (MIPS architecture); processing elements interconnected through a NoC; message passing communication scheme; parameterizable size; support to multitask execution.

One goal of the present work is to connect a new processor to the NoC, making the MPSoC heterogeneous. Despite the apparent simplicity of the proposal, it implies the definition of the processing element characteristics, which enables its correct connection to the NoC, as well as the identification of the operating system functions requiring modifications according to the employed processor.

Processing elements (PE) may be configured as *slaves* or *master*. The MPSoC is composed by one *master* PE (which should be always a Plasma processor), responsible for managing system resources, accessing the external task repository, and mapping tasks to *slave* PEs. *Slave* PEs execute application tasks. The present work focuses on *slave* PEs, enabling them to be either Plasma or MB-Lite [14] processors. Figure 3 presents an example of an MPSoC instance, with 2 MB-Lite and 7 Plasma processors (one Plasma is configured as *master*).

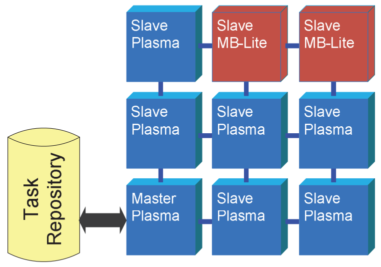


Figure 3 – Heterogeneous HeMPS: The processing element core can be MB-Lite or Plasma processor.

A. Processing Element Architecture

The MPSoC is composed by a parameterizable number of PEs interconnected by the 2-D mesh HERMES NoC [15]. This NoC has the following features: *(i) w*ormhole packet-switched mesh topology; *(ii)* credit-based flow control; *(iii)* XY routing algorithm; *(iv)* 16-bit flit size; *(v)* 16-flit buffers. The flit size is half of the processor word, to reduce the silicon area. Each PE, Figure 4, contains 5 main modules:

(i) **Processor:** Plasma or MB-Lite;
(ii) **Network Interface – NI:** responsible to treat packets sent/received to/from the NoC;
(iii) **DMA:** responsible to transfer data among PEs, decoupling communication from computation;
(iv) **dual port RAM memory:** stores the object code of the tasks and the microkernel;
(v) **Router:** main component of the NoC, responsible to interconnect neighbor PEs.

The interconnection of a given processor in the PE is a function of its memory architecture. Figure 4 shows the interconnection for Von Neumann (a) and Harvard (b) processor architectures. In (a) the processor has only one bus accessing the local RAM, enabling to assign one memory port to the processor and the second memory port to the DMA module, allowing parallel accesses to the memory. In (b) one of the memory ports is shared between the DMA and the processor, disabling the simultaneous accesses to the memory.

It is important to point out that just the top module of the PE is modified. The NI, DMA, RAM and Router modules are the same, regardless the processor architecture.

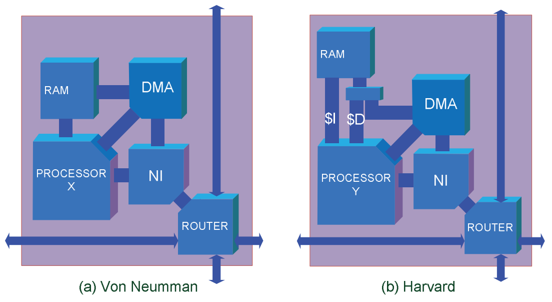


Figure 4 – PE architecture according to the processor memory architecture: (a) Von Neumann. (b) Harvard.

The minimum requirements to include a new processor in the present MPSoC includes:

(i) **Memory mapped register**. The address/data interface should enable access to memory mapped registers, located in the PE description. These registers are used by the microkernel to control the communication among the PE modules.
(ii) **Interrupt Interface**. From the viewpoint of the hardware it is necessary to have at least one interrupt pin. From the viewpoint of software it is necessary for the execution of an interrupt handler function.
(iii) **Toolchain for binary code generation**. A toolchain to generate the binary code for the microkernel and tasks, as the *gcc* suite, is required. The suite should include a compiler, an assembler and some disassembly tools to analyze the generated code, for debugging purposes.
(iv) **Word width**. The processor must have a 32-bit word to be connected to the PE modules.

B. Microkernel Architecture

Each slave processor executes a tiny operating system named microkernel. The master processor executes a different version of the microkernel, executing only resource management, e.g. task mapping and control of debug messages. The microkernel is written mostly in C code and some special functions in assembly language. The source code of the microkernel is the same for both processors. *Pragmas* (e.g. `#IFDEF <processor>`) differentiate the sections in the source code that are architecture dependent.

The microkernel architecture is divided in three layers, as shown in Figure 5. The first layer, *boot*, is responsible for the initialization of variables, as stack and global pointers. The second layer, *drivers*, contains the drivers which access the hardware modules as NI and DMA. The third layer implements the interrupt handler, the system calls and the task scheduler.

Figure 5 highlights the functions that are processor dependent. The *boot* layer should be written for each processor, according to its registers and rules to initialize the data memory. It corresponds roughly to 5% of the microkernel code.

For each service of the third layer there is a set of general functions, associated to processor dependent functions. The interrupt handler function is responsible to call some interrupt service according to values stored in the interruption vector memory mapped register (interruption from NI, DMA or *time_slice* scheduler). When a new task is scheduled (function *scheduler)*, the current task context is stored in the TCB (Task Control Block) and the scheduled task restores its previous state. The context saving/restoring is a function of the number of registers in the processor architecture. The communication functions access low-level drivers responsible to program the DMA module, for example.

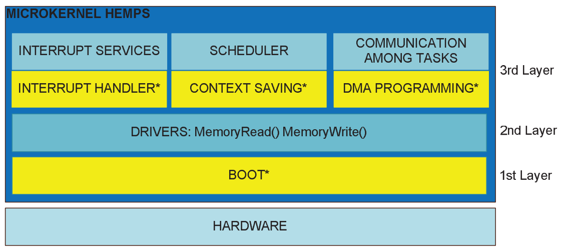


Figure 5 – Microkernel layers: Functions marked with * are processor dependent.

The local memory is segmented in equally sized pages. The first page is used by the microkernel, and the subsequent pages are employed by user tasks. A memory mapped register, named *page*, indicates the task being executed. If the *page* value is zero, the microkernel is being executed. Otherwise, some other task is in execution. The physical address value is obtained concatenating the *page* value with the address generated by the processor (instruction and data). This simple mechanism ensures protection among tasks, constraining the microkernel or the tasks to their corresponding pages.

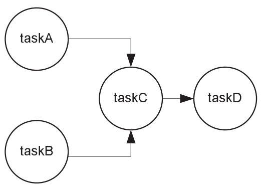
Message passing is supported through communication primitives *Send()* and *Receive()*. The *Send()* primitive is executed by user tasks, firing a system call. The microkernel stores the message in a vector, named *pipe*, and the task continues its execution (non blocking writing). The *Receive()* primitive seeks the message in the *pipe*. If the data to be read is located in the same processor, data is transferred from the *pipe* to the task. Otherwise, a request is sent through the NoC.

The microkernel has a *task table*, with the location of local and remote tasks. The location of each task is received from the *master* PE, when it maps a new task into the system. With this mechanism, tasks do not need to know where other tasks are located, only their identifier is required. When a *Send()* or *Receive()* is executed, the microkernel obtains the address of the target task from the task table.

The scheduler enables multitasking execution. The adopted algorithm is a simple round robin. A counter, named *time_slice*, interrupts the processor when it reaches a predefined value. This interruption causes the execution of the context saving, the scheduler function, and context restoring for the new scheduled task.

C. *Application Modeling*

User applications are modeled as task graphs, where vertices represent tasks and edges the communication between tasks. Figure 6 illustrates an application composed by 4 tasks, where tasks A and B send messages to task C, and this to the task D. All user tasks are described in C language.



```
int main()
{
    Echo("Communication task C started.");
    Receive(&msg1, TASKA);
    Send(&msg1, TASKD);
    ...
    Receive(&msg1, TASKB);
    Send(&msg1, TASKD);
    Echo("Communication task C finished.");

    return 0;
}
```

Figure 6– Application Modeling: in the left the task graph and in the right the taskC code, with Send() and Receive() primitives.

IV. HeMPS Framework

The HeMPS framework encapsulates a set of tools to help designers in the following MPSoC design steps: (*i*) platform configuration; (*ii*) code generation; (*iii*) application mapping onto the platform; (*iv*) debugging. The HeMPS framework GUI is presented in Figure 7. The left panel of the framework (number 1 in the Figure) contains the list of applications to be executed in the MPSoC. Applications are inserted/removed from this panel using buttons *add application/ delete application*. In this example, two applications were loaded: mpeg (composed by 5 tasks: start, ivlc, idtc, iquant, print) and communication (composed by 4 tasks: taskA, taskB, taskC, taskD). Note that a set of tasks were assigned to some processors, e.g. iquant to processor 02. This corresponds to the mapping of the initial tasks (those without dependences), which is defined by the user by moving a task from the application panel to a given processor.

The repository panel (number 3 in the Figure) receives the remaining tasks, which are dynamically mapped during system execution. Note that the user may select in which processor each task will execute. For example, if a given task has some feature that improves its performance in one of the processors, this processor is selected to execute the task.

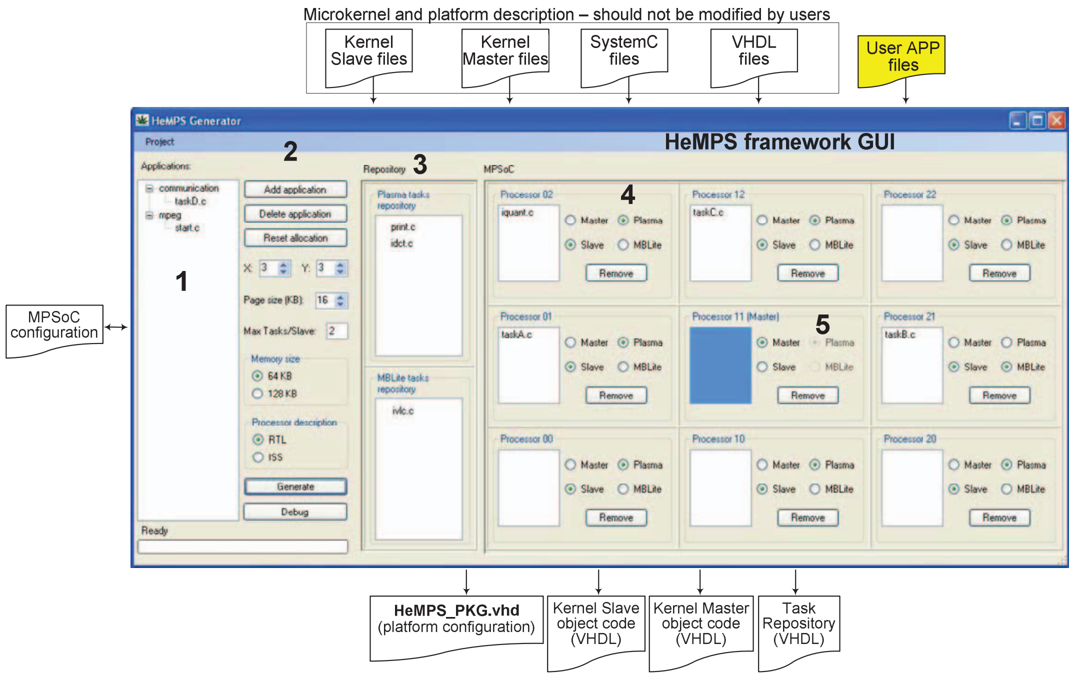
The region of the framework identified by the number 2 configures the platform:

- X/Y: size of the MPSoC;
- Page size, max task/slave, memory size: configure the processor memory;
- Processor description (RTL/ISS): RTL corresponds to the VHDL description, enabling synthesis and detailed debugging. ISS corresponds to a cycle accurate Instruction Set Simulator, written in SystemC, enabling faster simulations.
- *Generate* button: generate all output files (detailed in Figure 17). It is important to mention that a dedicated makefile is generated for each MPSoC configuration.
- *Debug button*: opens a debug GUI (Figure 11), displaying results obtained during simulation (presented in the Results Section).

Each processor has also a set of possible configurations (see numbers 4 and 5 in Figure). Processors may be slave or master (only one master is allowed), and also it is possible to select the processor type: Plasma or MB-Lite. The master processor is always configured as Plasma, because only slave microkernel was written for both processors. Note that the set of tasks assigned to a given processor is compiled using the toolchain of the select processor, when the button generate is pressed. The remove button clears all selected tasks assigned to the processor.

Figure 7 also presents the set of input and output files treated by the framework. The configuration of the MPSoC under design can be stored, enabling the exploration of different application scenarios. The input files include:

- Microkernel source codes. Set of C and assembly files.
- Platform description. Source code of the platform, in



communication: taskD.c
mpeg: start.c

X: 3 Y: 3
Page size (KB): 16
Max Tasks/Slave: 2
Memory size: [✓ 64 KB] 128 KB
Processor description: [✓ RTL] ISS

Plasma tasks repository: print.c, idct.c
MBLite tasks repository: ivlc.c

Processor 02: iquant.c — Master [✓ Plasma] [✓ Slave] MBLite
Processor 12: taskC.c — Master [✓ Plasma] [✓ Slave] MBLite
Processor 22: Master [✓ Plasma] [✓ Slave] MBLite
Processor 01: taskA.c — Master [✓ Plasma] [✓ Slave] MBLite
Processor 11 (Master): [✓ Master] Plasma Slave MBLite
Processor 21: taskB.c — Master Plasma [✓ Slave] [✓ MBLite]
Processor 00: Master [✓ Plasma] [✓ Slave] MBLite
Processor 10: Master [✓ Plasma] [✓ Slave] MBLite
Processor 20: Master [✓ Plasma] [✓ Slave] MBLite

Figure 7 – HeMPS framework GUI.

synthesizable VHDL or SystemC.

- User application files.

The output files generated by the framework include:

- Platform configuration: a VHDL package containing the MPSoC size, the type of each processor, the master PE position, among other parameters detailed later.
- Object code for the master and slave microkernels, described as a memory initialization file in VHDL. These files are static loaded to each local memory at design time.
- Task repository, corresponding to a unique file with all user application object codes, described as a VHDL memory initialization file.

V. RESULTS

To evaluate the generated MPSoC instances, the application illustrated in Figure 8(a) is used as benchmark. Each task of the application executes simple loops (to emulate the execution time of real applications). Initial tasks *A* and *B* transmit data to task C, which retransmit the received data to tasks *E* and *D*. Finally, task *F* receive data from *E* and *D*, sending the results to the master PE. This process is repeated 100 times.

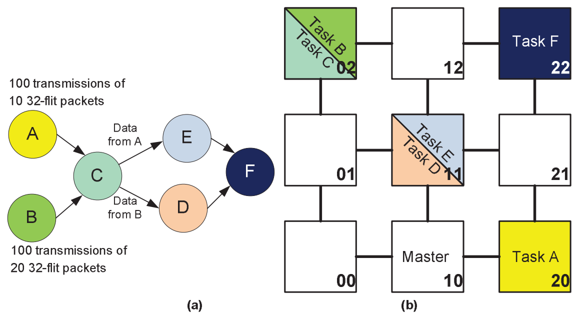


Figure 8– (a) Synthetic benchmark to validate the MPSoC and (b) its corresponding task mapping.

All simulated scenarios adopt a 3x3 MPSoC, with a fixed task mapping, as illustrated in Figure 8(b). The goal of this scenario is to: (*i*) evaluate local task communication (e.g in PE 02 task B sends data to task C), and remote task communication (e.g in PE 02 task C sends messages for tasks E and D, located at PE 11); (*ii*) evaluate multitasking execution, PEs 02 and 11 execute two tasks each.

Five scenarios were simulated, as presented in Table 1. Scenarios SC1 and SC2 are homogenous. The other 3 scenarios are heterogeneous. As can be observed, the most complex task, in terms of communication, is task C, mapped to PE 02. Task C sends data to tasks E/D, located in PE 11. Scenarios SC3 and SC4 vary the type of the processor chosen to PEs 02 and 11. Scenario SC5 employs two instances of MB-lite and two instances of the Plasma processor.

Table 1 – Simulation scenarios: 1 and 2 are homogeneous MPSoCs; 3, 4 and 5 are heterogeneous MPSoCs. Tasks C and F uses multiplication.

	PE02 tasks B/C	PE 11 tasks E/D	PE 20 task A	PE 22 task F	Execution time
SC1	Plasma	Plasma	Plasma	Plasma	168 ms
SC2	MB-Lite	MB-Lite	MB-Lite	MB-Lite	188 ms
SC3	MB-Lite	Plasma	Plasma	Plasma	177 ms
SC4	Plasma	MB-Lite	Plasma	Plasma	165 ms
SC5	MB-Lite	MB-Lite	Plasma	Plasma	177 ms

A. Execution time

The last column of Table 1 shows the total execution time to execute the benchmark, with the MPSoC executing at 50 MHz, (each loop iteration consumed, in average, 85,000 clock cycles). The difference between the best and worst case is 14%. The execution in SC2 takes longer, since the MB-Lite shares the memory access with the NI, stalling its execution when transmitting/receiving data.

The simulation of SC1 with processors/memories described in VHDL took in average 240 minutes, while with the Plasma processor and the RAM described in SystemC (ISS), the simulation for the same scenario took 20 minutes (Intel Xeon 64 bits, quad-core, 12 GB RAM). In both cases, the obtained results were the same, validating the cycle-accurate Plasma ISS.

Figure 9 presents the average execution time to execute task C. Scenarios SC2, SC3 and SC5 present higher execution time, since in these scenarios task C was executed in an MB-Lite processor.

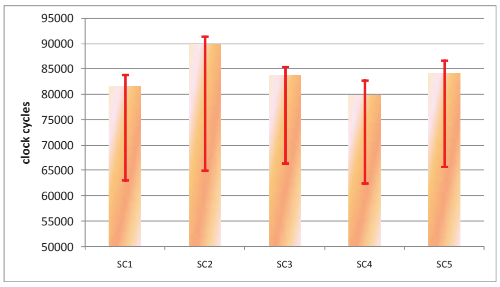


Figure 9 – Average execution time, in clock cycles, for each iteration of taskC. Red bars indicate min/max execution time for each simulated scenario.

The most relevant result is the validation of the system with different processors, automatically generated from the framework.

B. Area and Operating Frequency

Figure 10 presents area and frequency estimation for both PEs (processor, DMA, NI and Router) on a Xilinx FPGA 5vlx330tff1738-2. Despite the higher area consumption (LUTs) of the MB-Lite (1,882) compared to the Plasma (1,198), these processors are smaller than other embedded processors, as Leon3 [16] and OpenRisc [17], both with roughly 3,000 LUTs.

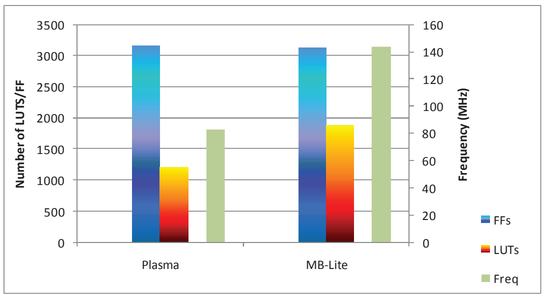
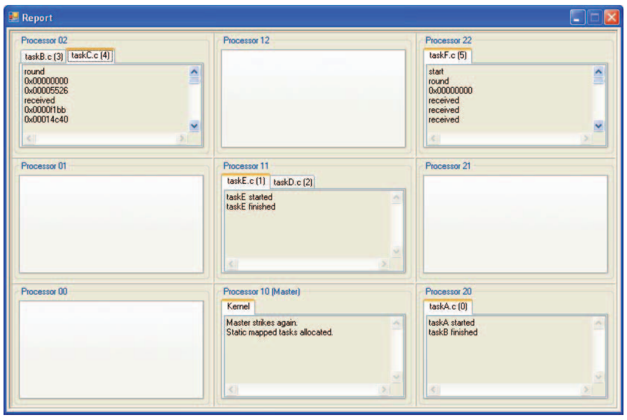


Figure 10 – Area and frequency operation for Plasma and MB-Lite processor.

One remarkable result is the frequency estimation. The MB-Lite achieves a frequency 73% higher than Plasma, due to the higher number of pipeline stages and the Harvard memory organization. This feature enables to improve the system performance, by increasing the processor frequency, while keeping the router in the same frequency. This is achievable including an asynchronous interface between the router and the network interface.

C. *Debug Interface*

The debug GUI (Figure 11), invoked after the simulation, contains one panel for each processor, and tabs for each task executing in a given processor. In the panel corresponding to the master processor (processor 10), two microkernel messages are displayed. The other panels contain the messages sent by tasks. Observe for example processor 02, it contains two tabs, one for *task B* and the other one for *task C*, accordingly to task mapping in Figure 8(b). The numbers displayed corresponds to the generate data, and to the number of clock cycles spent since the beginning of the system execution, obtained through the *gettick()* system call. The use of this system call enables to compute the task execution time, latency, and throughput.



```
round
0x00000000
0x00005526
received
0x0000f1bb
0x00014c40
```

```
start
round
0x00000000
received
received
received
```

```
taskE started
taskE finished
```

```
Master strikes again.
Static mapped tasks allocated.
```

```
taskA started
taskB finished
```

Figure 11 – HeMPS debug GUI.

VI. CONCLUSION AND FUTURE WORKS

The main contribution of the present work is the validation of an open-source framework for heterogeneous NoC-based MPSoC generation, available at www.inf.pucrs.br/~gaph. Even if the two integrated processors have similar architectures (32-bit RISC processors), the method to include new processors is well defined, and the microkernel supports parameterization. Our results shown several MPSoC instances, and the simulation reported the correct operation of the executed applications.

Future work includes the addition of: (*i*) hardware monitors to collect data related to power, latency and throughput; (*ii*) new processors, with specialized functions; *(iii)* a decentralized control mechanism, to avoid a communication bottleneck on the master processor, *(iv)* an task migration mechanism and (v) port the mapping algorithm to the heterogeneous platform (today, in heterogeneous configurations only static mapping is allowed).

ACKNOWLEDGMENTS

The Author Fernando Moraes acknowledge the support of CNPq and FAPERGS, projects 301599/2009-2 and 10/0814-9, respectively.

REFERENCES

[1] http://www.xilinx.com/tools/platform.htm
[2] http://www.altera.com/support/software/system/sopc/sof-sopc_builder.html
[3] Carara, E. A.; Oliveira, R. P.; Calazans, N. L. V.; Moraes, F. G. *HeMPS - a framework for NoC-based MPSoC Generation*. In: ISCAS, 2009, pp. 1345-1348.
[4] Mandelli, M.; Ost, L.; Carara, E. A.; Guindani, G. M.; Rosa, T.; Medeiros, G.; Moraes, F. G. *Energy-Aware Dynamic Task Mapping for NoC-based MPSoCs*. In: ISCAS, 2011.
[5] Joven, J.; Font-Bach, O.; Castells-Rufas, D.; Martinez, R.; Teres, L.; Carrabina, J. *xENOC – An eXperimental Network-on-Chip Enviroment for Parallel Distributed Computing on NoC-based MPSoC Archtectures*. In: Euromicro pp. 141-148.
[6] Joven, J. *A Lightweight MPI-based Programming Model and its HW Support for NoC-based MPSoCs*. In: PhD Forum DATE'09, 2009.
[7] Kumar, A.; Hansson, A.; Huisken, J.; Corporaal, H. *An FPGA Design Flow for Reconfigurable Network-Based Multi-Processor Systems on Chip*. In: DATE, 2007.
[8] Silicon Hive. Available from: http://www.silicon-hive.com.
[9] Goossens, K.; Dielissen, J.; Radulescu, A. *Æthereal network-on-chip: concepts, architectures, and implementations*. IEEE Design & Test of Computers, v.22(5), 2005, pp. 414-421.
[10] Singh, A. K.; Kumar, A.; Srikanthan, T.; Ha, Y. *Mapping Real-life Applications on Run-time Reconfigurable NoC-based MPSoC on FPGA*. In: FPT, 2010, pp. 365 – 368.
[11] Yang, Z. J.; Kumar, A.; Yajun H.. *An Area-efficient Dynamically Reconfigurable Spatial Division Multiplexing Network-on-Chip with Static Throughput Guarentee*. In: FPT, 2010, pp. 389 – 392.
[12] Fernandez-Alonso, E.; Castells-Rufas, D.; Risueno, S.; Carrabina, J.; Joven, J. *A NoC-based multi-{soft}core with 16 cores*. In: ICECS, 2010, pp. 259-252.
[13] Castells-Rufas, D.; Joven, J.; Risuefto, S.; Fernandez, E.; Carrabina, J. *NocMaker: A Cross-Platform Open-Source Design Space Exploration Tool for Networks on Chip*. In: INA-OCMC Workshop, 2009.
[14] Kranenburg, T.; van Leuken, R. *MB-LITE: A Robust, Light-Weight Soft-Core Implementation of the MicroBlaze Architecture*. In: DATE, 2010, pp. 997-1000.
[15] Moraes, F.; Calazans, N.; Mello, A.; Möller, L.; Ost, L. *Hermes: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip*. Integration, the VLSI Journal, Vol. 38(1), 2004, pp. 69-93.
[16] Leon3 Processor, http://www.gaisler.com
[17] OpenRisc 1200, http://opencores.org/project,or1200_hp