HeMPS - A Framework for NoC-Based MPSoC Generation

Everton A. Carara, Roberto P. de Oliveira, Ney L. V. Calazans, Fernando G. Moraes
PUCRS – FACIN – Av. Ipiranga 6681 – Porto Alegre – 90619-900 - Brazil
carara@inf.pucrs.br, robertoportdeoliveira@yahoo.com.br, ney.calazans@pucrs.br, fernando.moraes@pucrs.br

Abstract—**Multi-Processor Systems-on-Chip (MPSoCs) are increasingly popular in embedded systems. Due to their complexity and huge design space to explore for such systems, CAD tools and frameworks to customize MPSoCs are mandatory. Some academic and industrial frameworks are available to support bus-based MPSoCs, but few works target NoCs as underlying communication architecture. A framework targeting MPSoC customization must provide abstract models to enable fast design space exploration, flexible application mapping strategies, all coupled to features to evaluate the performance of running applications. This paper proposes a framework to customize NoC-based MPSoCs with support to static and dynamic task mapping and C/SystemC simulation models for processors and memories. A simple, specifically designed microkernel executes in each processor, enabling multitasking at the processor level. Graphical tools enable debug and system verification, individualizing data for each task. Practical results highlight the benefit of using dynamic mapping strategies (total execution time reduction) and abstract models (total simulation time reduction without losing accuracy).**

I. INTRODUCTION AND RELATED WORKS

MPSoCs are complex architectures, composed by processors, IPs, memories and specialized IPs. CAD tools and frameworks to customize such architectures for implementing specific applications are mandatory, given the huge design space to explore.

The goal of this paper is to present the design of a framework for NoC-based MPSoC generation. This framework generates a synthesizable RTL VHDL system description together with C/SystemC simulation models for processors (ISS) and memories, which reduce the simulation time up to 91%, compared to the pure RTL simulation. Besides the hardware infrastructure, the framework provides a software infrastructure which includes a *multitask microkernel*, inter-task communication primitives and support to *dynamic workloads*.

The rest of this Section presents a short review of similar proposals available in the literature. Section 2 gives an overview of the proposed parameterizable MPSoC platform, called HeMPS. Next, Section 3 describes the design flow to build applications on top of HeMPS. Section 4 discusses some practical results of building an instance of the HeMPS platform and using it to prototype applications. Finally, Section 5 proposes a set of conclusions and directions for future works.

In [1] Lyonnard et al. presented an automatic design flow to generate application-specific heterogeneous MPSoC architectures. The architecture generation relies on generic multiprocessor architectures templates, which are composed by four element types: (*i*) processors (e.g. ARM7, 68000); (*ii*) communication coprocessors (interface to the interconnection architecture); (*iii*) IP components (e.g. memory modules, bus bridges); (*iv*) interconnection architecture (point-to-point or bus). The architecture templates are parameterizable for the four element types.

STARSoC [2] is a framework for hardware/software codesign and design space exploration. The input description consists in a set of software and hardware processes described in C. After specifying the number of processors (instances of the freely available core processor called OpenRisc), a HW-SW partitioning is executed. The hardware part is synthesized in an RTL description and the software part is distributed among the set of processors. As result, STARSoC generates a bus-based MPSoC platform from a high-level application specification.

xENOC [3] is an environment for hardware/software automated design of NoC-based MPSoC architectures. The core of this environment is an EDA tool, called NoCWizard, which can generate RTL Verilog NoCs. The whole system is described in an XML file (NoC features, IPs and mapping), which is used as input for the automatic generation tools. In addition to the hardware infrastructure, xNoC also includes an Embedded Message Passing Interface (eMPI) supporting parallel task communication.

Some commercial design environments support creation of bus-based MPSoCs. Examples are Altera SOPC [7] and Xilinx EDK [8]. These environments provide graphical tools for system integration along with an extensive IP cores library. The HDL description of the final system and hardware-software integration are strongly automated. SOPC allows MPSoCs design based on NIOS, ARM and ColdFire processors connected through the proprietary Avalon bus. EDK allows the design based on MicroBlaze and PowerPC processors integrated through the IBM CoreConnect bus architecture.

II. ARCHITECTURE OVERVIEW

The architecture proposed here, named HeMPS, is a homogeneous NoC-based MPSoC platform. Figure 1 presents a HeMPS instance using a 2x3 mesh NoC. The main hardware components are the HERMES NoC [4] and the mostly-MIPS processor Plasma [5]. A processing element, called Plasma-IP, wraps each Plasma and attaches it to the NoC. This IP also contains a private memory, a network interface, and a DMA module.

Typical applications running in MPSoCs, such as multimedia and networking, often present a dynamic workload. This implies a varying number of tasks running simultaneously, and their number or load often exceeds the available resources. To tackle this issue,

978-1-4244-3828-0/09/$25.00 ©2009 IEEE

HeMPS assumes that: (*i*) applications are modeled using task graphs; (*ii*) only a subset of tasks is initially loaded into the system. Remaining tasks are stored in an external memory, named *task repository*. This memory keeps all task codes necessary in any instance of the applications execution.

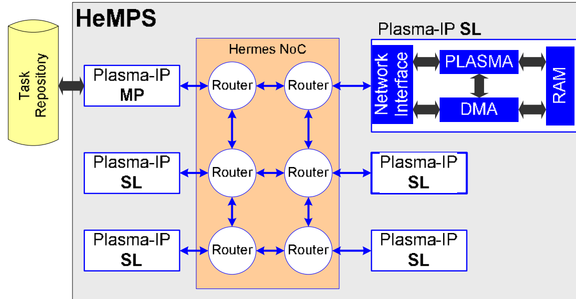


Figure 1 - HeMPS instance using a 2x3 mesh NoC.

The system contains a *master processor* (Plasma-IP MP), responsible for managing system resources. This is the only processor having access to the task repository. When HeMPS starts execution, the master processor allocates initial tasks to the *slave processors* (Plasma-IP SL). During execution, tasks are dynamically loaded from the task repository to slave processors on demand. Also resources may become available when a given task finishes execution. Such dynamic behavior enables smaller systems, since only those tasks effectively required are loaded into the system at any given moment.

The Hermes NoC [4] employs a 2D mesh topology. Routers have only input buffers, a control logic shared by all router ports, an internal crossbar and up to five bi-directional ports. A single round-robin arbitration schedules grants access to incoming packets, and a deterministic distributed XY routing algorithm determines the path between source and target IPs.

To achieve high performance in the processing elements, the Plasma-IP architecture targets the separation between communication and computation. The network interface and DMA modules are responsible for sending and receiving packets, while the Plasma processor performs task computation and wrapper management. The local RAM is a true dual port memory allowing simultaneous processor and DMA accesses, which avoids extra hardware for elements like *mutex* or cycle stealing techniques.

A. Microkernel

Each slave processor runs a microkernel, which supports *multitasking* and *task communication*. The microkernel segments memory in pages, which it allocates for itself (first page) and tasks (subsequent pages). Each Plasma-IP has a *task table*, with the location of local and remote tasks. A simple preemptive scheduling, implemented as a round robin, provides support to multitasking.

The microkernel protects the memory pages and all communication among tasks occurs through message passing. Message passing is supported through a global message pipe located in the microkernel and communication primitives (*WritePipe()* and *ReadPipe()*), which compose the current HeMPS API.

The underlying model of computation for ensuring synchronization between tasks is base on Kahn Process Networks (KPN) [5]. KPN is a distributed model of computation where unbounded FIFOs communication channels (*pipes*) connect processes to each other, forming a process network. KPNs rely on the fundamental principle that communication must be blocking for channel read operations and non-blocking for channel write operations.

When a given task executes a *WritePipe()* primitive, the message is stored in the processor global *pipe*, and computation continues. This characterizes a non-blocking writing. The global pipe is software implemented as a parameterizable array with random access. In this way, problems such as head-of-line (FIFO) blocking and deadlocks are avoided.

For the *ReadPipe()*, a system function is executed. If the target task is located in the same processor, the task executes a read in the local global pipe. If the task is located in another processor, the microkernel sends a *request* message through the NoC and the task enters in *wait* state. When the message arrives from network, the microkernel stops the executing task and reschedules the waiting task. Figure 2 illustrates this process. In Figure 2(a) assumes task t2 has written a message in the global pipe, addressed to task t5 (*WritePipe(&msg,t5)*), and task t5 is requesting the message from task t2 (*ReadPipe(&msg,t2)*). In Figure 2(b) processor 1 sends the requested message to processor 2 through the NoC. The system ensures in-order message delivering, because the *WritePipe()* adds to each message the order in which they were written.

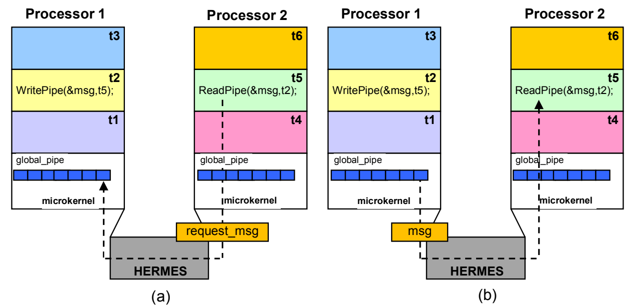


Figure 2 – HeMPS reading of an available message.

B. Dynamic workload

The master processor is responsible for system management, including: (*i*) task allocation; (*ii*) broadcast of control messages (unicast based), such as placement of allocated tasks and release of finished tasks; *(iii)* reception of control messages, as end of task and debug packets. It does not execute application tasks.

Dynamic workload takes place through on demand task allocation at application runtime. The system designer defines an initially needed set of tasks. The trigger to fire a new task allocation is the *WritePipe()* primitive. Each time a task executes *WritePipe()*, the microkernel executing this function verifies in the task table if the target task is already allocated in the system. If the task is already in the system, the microkernel writes the message into its global pipe. Otherwise, it sends a *task request message* to the master processor. Figure 3 illustrates this process.

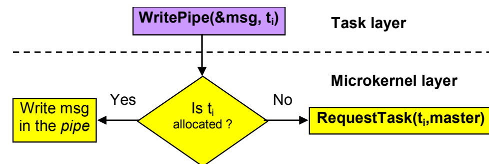


Figure 3 – Application-transparent dynamic task allocation in HeMPS.

When the master processor receives a task request message, it configures the DMA module, which accesses the task repository and transmits the task code to the target Plasma-IP SL memory. After task transmission, the master processor notifies all slaves with the task ID and its new position. The present HeMPS implements a very simple mapping heuristic, based on the number of available pages in each processor. The task is scheduled for the processor with more free

pages, and if there are no free pages in the system, the task is scheduled to be transmitted when a given page becomes available. Note that the dynamic allocation is completely transparent at the task layer.

III. HeMPS Design Flow

The HeMPS design flow follows the platform-based design methodology depicted in Figure 4.

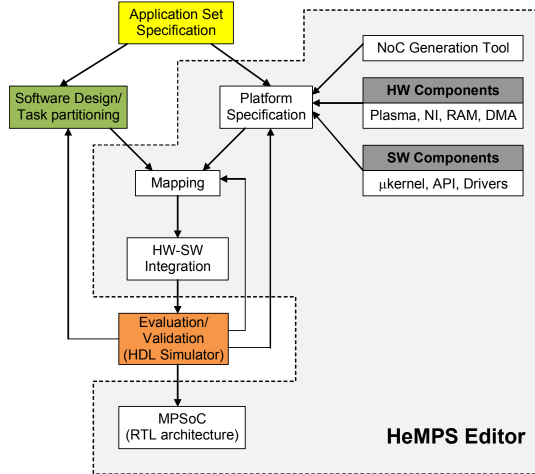


Figure 4 – HeMPS Design Flow.

The design flow starts with the specification of the set of applications to be executed by the MPSoC. Next, *software design* and *platform specification* steps can be carried out in parallel, due to the HeMPS API, which isolates the application software from the platform implementation. The *software design* includes task partitioning, with the definition of the task graph for each application. The platform specification includes: (*i*) definition of the NoC size; (*ii*) customization of hardware components, as memory size, and number of pages per processor memory; (*iii*) customization of the microkernel according to the NoC parameters and hardware features.

In the sequel, tasks are mapped in the platform, followed by hardware-software integration. The MPSoC is then evaluated, and if design constraints are not met, it is possible to redefine task graphs, mapping, or platform parameters. The last step in the flow is the generation of the platform hardware HDL code.

IV. HeMPS Framework Editor

The HeMPS Editor covers several of the design steps presented in HeMPS design flow, helping to automate the platform generation step. Figure 5 presents the HeMPS Editor graphic interface main window.

This framework allows quick platform customization, with the user setting the number of processor connected in a mesh NoC through the parameters *X* and *Y*. The *maximum number of tasks per slave* is parameterizable, and is a function of two parameters, *page size* and *memory size*. For performance evaluation purposes, processors and local memories are modeled using cycle accurate instruction set simulators (ISSs) and C/SystemC models, respectively. This enables faster design space exploration. The left panel in Figure 5 presents two applications (*mpeg, and communication*) along with their task composition. Drag and drop actions allows to easily perform the initial static task mapping to slave processors. The master processor receives the remaining tasks, which correspond to the contents of the task repository.

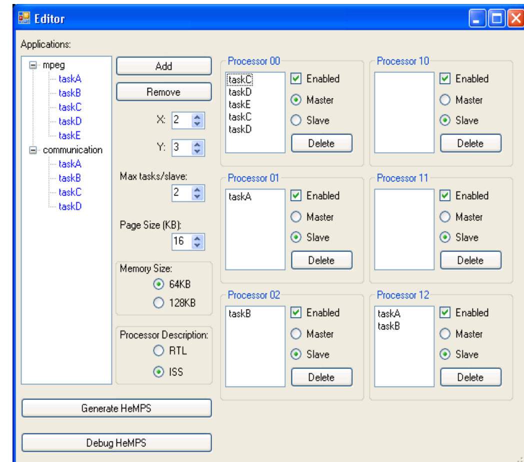


Figure 5 – HeMPS Editor main window.

The *Generate HeMPS* button executes the hardware-software integration. This action fills the memories (microkernel, API and drivers) and the task repository with all task codes.

System evaluation takes place using a commercial RTL simulator, such as ModelSim. The *Debug HeMPS* button calls a graphic debug tool (Figure 6). During simulation, when a given task executes a *print()* system call, a debug packet is sent to the master processor tagged with the source task and processor IDs. All debug data received by the master processor is stored in a dedicated memory area, used by the debug tool. This tool contains one panel for each processor. Each panel has tabs, one for each task executing in the corresponding processor (in Figure 6 processors 10 and 01 execute 2 tasks each one). In this way, messages are separated, allowing to the user to visualize the execution results for each task. Another system call useful for debug message is *getick()*, which returns the current execution time in clock cycles. Using this system call, it is possible to compute the task execution time, latency and throughput.

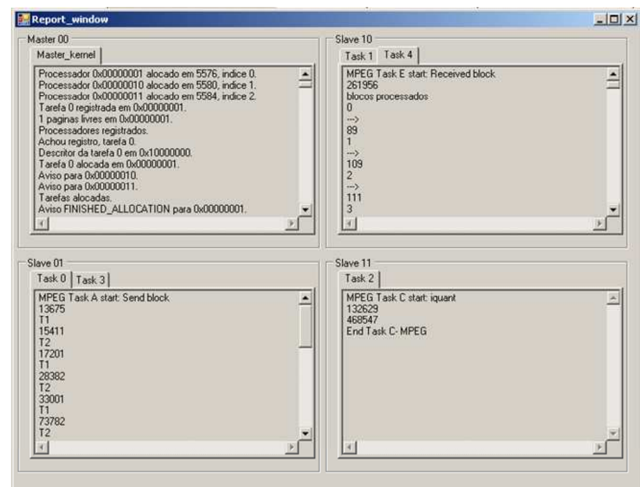


Figure 6 – HeMPS debug interface.

Once design constraints are met, the MPSoC can be synthesized replacing ISSs and RAMs C/SystemC simulation models by VHDL code, which is generated by the HeMPS Editor.

V. RESULTS

A Motion-JPEG (MJPEG) decoder application has been chosen for the experiments. This application is partitioned in four tasks: (*i*) data producer, T0; (*ii*) Inverse Variable Length Coding, T1; (*iii*) Inverse Quantization, T2; (*iv*) Inverse Discrete Cosine Transform, T3; (*v*) output data printing, T4. No color space conversion has been implemented here. Therefore, only grey-level images are processed.

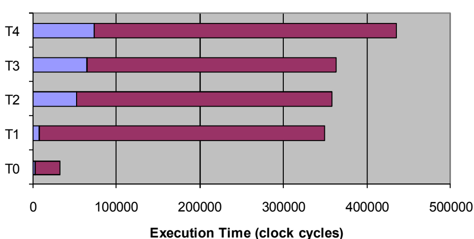
Four simulations scenarios are evaluated, in a 2*x*3 HeMPS instance: (*i*) ISS/RAM models and static mapping; (*ii*) ISS/RAM models and dynamic mapping; (*iii*) RTL models and static mapping; (*iv*) RTL models and dynamic mapping. The application has been evaluated according to three criteria: simulation time, execution time, and abstract model accuracy. As mentioned before, the use of ISS/RAM models enables fast design space exploration. Table I displays the total simulation time (first criterion) for four scenarios, using a 1.66GHz Intel Core 2 Duo PC with 1GB of RAM. The use of ISS/RAM models reduced the total simulation time up to 91%.

TABLE I. SIMULATION TIME USING DIFERENT SIMULATION MODELS.

Simulated Scenario	Total Simulation Time (seconds)
ISS/RAM models – static mappping	383
ISS/RAM models – dynamic mappping	386
RTL models – static mappping	4677
RTL models – dynamic mappping	4550

The second criterion relates execution time to the choice of mapping strategy: static or dynamic. Light gray bars in Figure 7(a) denote the allocation time, whereas dark gray ones denote task execution time. It should be noted that tasks are sequentially loaded into the system. In Figure 7(b), dynamic mapping, only task T0 is initially loaded into the system. The remaining tasks are loaded on demand, resulting in shorter execution time with regard to static mapping. In the static approach the loaded tasks wait for available data (looping) before start the computation, whereas in the dynamic approach the data is already available when the tasks are loaded. The overall performance in both strategies is quite similar, however, the dynamic approach favors the multitasking due to the shorter execution time, maximizing the CPU processing.

(a) MJPEG tasks execution time: static mapping (ISS/RAM models)



(b) MJPEG tasks execution time: dynamic mapping (ISS/RAM models)

Figure 7 – Application execution time, in clock cycles, for different mapping strategies. Black bars denote the execution time.

The third criterion evaluates the relative accuracy of ISS/RAM models against RTL models. Even if the ISS is cycle accurate, some small differences to RTL implementation exist, as in multiply and divide operations. In the RTL implementation such operations may be executed in parallel with subsequent operations, while in the Plasma ISS these always execute in 32 clock cycles, during which no other instruction can be executed. The error observed comparing the total execution time in clock cycles is 2.14% and 1.07% for static and dynamic mapping, respectively. Thus, the use of ISS does not imply significant loss of accuracy and adds efficiency to system simulation.

VI. CONCLUSIONS AND FUTURE WORKS

HeMPS supports several steps of platform-based MPSoC design including platform customization, application mapping and hardware-software integration. It provides cycle accurate simulation models (C/SystemC) for microprocessors (ISSs) and memories for fast design space exploration. Results showed that the abstract modeling has a significant impact on the simulation time. A graphical tool supports application level debug, providing an efficient high level approach as alternative to traditional RTL waveform debug.

Significant HeMPs features related to the state-of-the-art in MPSoCs are: (*i*) *NoC* used as interconnection architecture; (*ii*) *microkernel supported multitasking*, increasing the amount of simultaneous executing tasks in each processor; (*iii*) *dynamic workload*, enabling on demand system task loading; and (*iv*) *inter-task communication primitives* (HeMPS API), which abstracts the platform hardware, supporting software development independent of mapping strategies and platform dimensions.

The conducted experiments showed the effectiveness of some framework features like simulation models and dynamic mapping strategies. Simulation models reduced the total simulation time by 91%, with a cycle accurate error lower than 3%. Dynamic mapping performed better than traditional static mapping, due to the reduced amount of simultaneity of messages request.

Some works in the path to make HeMPS evolve include: (*i*) support to other processors, DSPs and specialized IPs to enable the generation of heterogeneous MPSoCs; (*ii*) load balancing dynamic mapping heuristics; (*iii*) enable ordinary task execution on the master processor; and (*iv*) improvements on NoC architecture such as QoS and broadcast/multicast services along with support to these features at the application layer. In the current implementation, messages are broadcasted sending several unicast copies. The final goal is to have an API which includes support to all services provided by the NoC. In this way many kinds of different application requirements can be efficiently fulfilled.

REFERENCES

[1] Lyonnard, D. et al "*Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip*". In: DAC, 2001, pp. 518-523.

[2] Samahi, A.; Bourennane, E. "*Automated Integration and Communication Synthesis of Reconfigurable MPSoC Platform*". In: AHS, 2007, pp. 379-385.

[3] Joven, J.; Carrabina, J.; et al *"xENOC – An eXperimental Network-on-Chip Enviroment for Parallel Distributed Computing on NoC-based MPSoC Archtectures*". In: Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2008, pp. 141-148.

[4] Moraes, F.; et al. "*HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip*". Integration the VLSI Journal, 38(1), Oct. 2004, pp. 69-93.

[5] OpenCores, www.opencores.org.

[6] G. Kahn. "*The semantics of a simple language for parallel programming*". In: Information Processing, 1974, pp 471-475.

[7] Altera, www.altera.com

[8] Xilinx, www.xilinx.com