Hindawi Publishing Corporation
EURASIP Journal on Embedded Systems
Volume 2006, Article ID 18526, Pages 1–11
DOI 10.1155/ES/2006/18526

# Generation of Embedded Hardware/Software from SystemC

**Salim Ouadjaout and Dominique Houzet**

*Institut d'Electronique et de Télécommunications de Rennes (IETR), UMR CNRS 6164, Institut National des Sciences Appliquées (INSA), 20 avenue des Buttes de Coësmes, 35043 Rennes Cedex, France*

Received 30 November 2005; Revised 26 June 2006; Accepted 27 June 2006

Designers increasingly rely on reusing intellectual property (IP) and on raising the level of abstraction to respect system-on-chip (SoC) market characteristics. However, most hardware and embedded software codes are recoded manually from system level. This recoding step often results in new coding errors that must be identified and debugged. Thus, shorter time-to-market requires automation of the system synthesis from high-level specifications. In this paper, we propose a design flow intended to reduce the SoC design cost. This design flow unifies hardware and software using a single high-level language. It integrates hardware/software (HW/SW) generation tools and an automatic interface synthesis through a custom library of adapters. We have validated our interface synthesis approach on a hardware producer/consumer case study and on the design of a given software radiocommunication application.

## 1. INTRODUCTION

Technological evolution—particularly shrinking silicon fabrication geometries—enables the integration of complex platforms in a single system on chip (SoC). In addition to specific hardware subsystems, a modern SoC can also include sophisticated interconnects and one or several CPU subsystems to execute software. New design flows for SoC design have become essential in order to manage the system complexity in a short time-to-market. These flows include hardware/software (HW/SW) generation tools, the reuse of predesigned intellectual property (IP), and interface synthesis methodologies which are still open problems requiring further research activities [1].

EDA tools propose their own solutions to HW/SW generation. Some use SystemC as a starting point for the hardware design, like Cynthesizer from Forte Design [2] or Agility Compiler from Celoxica [3]. Several tools use the C language as a starting point for both hardware and software with a custom application programming interface (API) for HW/SW interfaces. It is the case of DK Design Suite from Celoxica [3] with its DSM API and CatapultC from Mentor [4]. In SiliconC [5], structural VHDL is generated for the C functions. Prototypes of the functions become the entities. There are other variants which start from Matlab to produce both hardware and software like SPW from CoWare [6]. Many design methodologies exist for the design of embedded software [7–9]. Some are based on code generated from an abstract model (UML [10]), graphical finite state machine design environments (e.g., StateCharts [11]), DSP graphical programming environments (e.g., Ptolemy [8]), or from synchronous programming languages (e.g., Esterel [12]). A software generation from a high-level model of operating system is proposed by several authors [13–16]. In [15], a software generation from SystemC is based on the redefinition and overloading of SystemC class library elements. In [13], a software-software communication synthesis approach by substituting each SystemC module with an equivalent C structure is proposed. It requires special SystemC modeling styles (i.e., with macrodefinitions and preprocessing switches in addition to the original specification code). In [16], software is generated from SpecC with no restrictions on the description of the system model.

Several approaches have been developed to deal with IP integration. Fast prototyping enables the productive reuse of IPs [17]. It describes how to use an innovative system design flow that combines different technologies, such as C modelling, emulation, hard virtual component reuse, and CoWare tools [6]. Prosilog's IP creator, as part of Magillem, aims to improve the integration and reuse of non-VCI compliant IPs by wrapping them into a compatible structure. This tool allows the generation of wrappers from a RTL VHDL description of the IP interface [18]. The Cosy approach is based on the infrastructure and concepts developed in the VCC framework [19]; it defines interfaces at multiple levels of abstraction. Most of those approaches deal with low-level protocol adaptation in order to integrate RTL, level IPs. A few approaches provide a ready network on Chip (NoC) to allow easy integration of communication. But these approaches require that the IPs have to be compliant with the NoC interface. Consequently, the designers have to modify the IPs codes. All these approaches deal with system-level synthesis which is widely considered as the solution for closing the productivity gap in system design. System-level models are developed for early design exploration. The system specification of an embedded system is made of a hierarchical set of modules (or processes) interconnected by channels. They are described in a system-level language as a set of behaviour, channel, and interface declarations. Those behaviours mapped onto general or application-specific microprocessors are then implemented as embedded software and hardware. The predominant system-level languages are C/C++ extensions [13, 20]. We consider here the SystemC language but another language can be used. SystemC is mainly used to model and to simulate designs at system level. However, dedicated powerful hardware description languages like VHDL and Verilog are used for RTL. Embedded software languages like C with static scheduling or POSIX RTOS are used for embedded processors. This leads to a decoupling of behavioral descriptions and implementable descriptions. This decoupling usually requires the recoding of the design from its specification simulation in order to meet the very different requirements of the final generated code. The recoding step often results in new coding errors that must be identified and debugged. The derivation of embedded software and hardware from system specifications described in a system-level language requires to implement all language elements (e.g., modules, processes, channels, and port mappings). It is known that SystemC allows the refinement for hardware synthesis, but up to now SystemC has not been used as an embedded software language. Considering the limited memory space and execution power of embedded processors, the SystemC overhead makes the direct compilation to produce the binary code for target embedded microprocessors highly inefficient. Obviously, it is due to the large SystemC kernel included in the compiled code. This kernel introduces an overhead to support the system-level features (e.g., hierarchy, concurrency, communication), but these features are not necessary to the target embedded software code. In addition to direct SystemC compilation inefficiency, some cross-compilers for embedded processors may only support the C language. Thus, SystemC has to be translated to C code.

To address system-level synthesis, we propose in this paper a top-down methodology. Our challenge is to automate the codesign flow generating the final code for both embedded processors and hardware from a unifying high-level language (SystemC). In our methodology, we have developed methods to make the codesign flow smooth, efficient, and automated. These methods allow two improvements: a rapid integration of communication and a fast software generation for embedded processors with an efficient interface synthesis. The proposed methodology includes several parsing steps and intermediate models. The first main step is the communication integration based on a custom library of interface adapters that uses the virtual component interface (VCI) standard from VSIA consortium [21]. This library aims to perform the interface synthesis. It allows heterogeneous IPs to communicate in a plug-and-play fashion in the same system. The second main step is the generation of embedded C code from the system specification written in SystemC. Our approach proposes the use of static scheduling and POSIX-based RTOS models. It enables also an automatic refinement, while [14] requires its own proprietary simulation engine and needs manual refinement to get the software code. Our method also differs from [13–16] in that our high-level SystemC code is translated to a C code with optimized interface synthesis. Optimization is performed according to the processors busses and the NoC as well as according to the SystemC parallel programming model. Other recent propositions have been published in that direction [22].

The paper is organized as follow. In Section 2 we describe the main features of our proposed design flow. The main innovative parts of the design flow are detailed in the next two sections. The first one presents our hardware interface library and our integration methodology of functional IPs, with implementation results from a simple design example. The second one describes the translation process of SystemC elements to C code. This C code targets either an RTOS for dynamic scheduling or a stand alone solution with a generated static scheduling. This translation process is validated in Section 5 with implementation results of a producer/consumer and a code-division multiple-access (CDMA) radio-communication applications. This work is the result of a project started in 2001 [23–25].

## 2. DESIGN FLOW

SoC design requires the elaboration and the use of radically new design methodologies. The main parts of a typical system-level design flow are the specification model, the partition into HW/SW elements, and the implementation of the models for each element. In Figure 1 we describe the proposed top-down methodology of automatic generation of binary files from SystemC to both embedded software and hardware. The design flow starts with a high-level model described in a high-level programming language (SystemC). The system is described either through direct programmation or through IP reuse. We use Celoxica tools to develop, simulate, analyze, and validate the SystemC code (step (1)). The first SystemC description is at the functional level. The system is a set of functional IPs including functional models of architectural IPs for fast simulation. The communication between IPs uses SystemC channel mechanisms like sc_signal or sc_fifo with read() and write() primitive functions. From the Celoxica graphical tool, we select the IPs which are associated with the hardware side (the architectural IPs substituted by their already VCI-compliant version), and the IPs which are associated with the software side (the monitoring IPs, stimulating IPs, host IPs, etc.). The remaining IPs of the

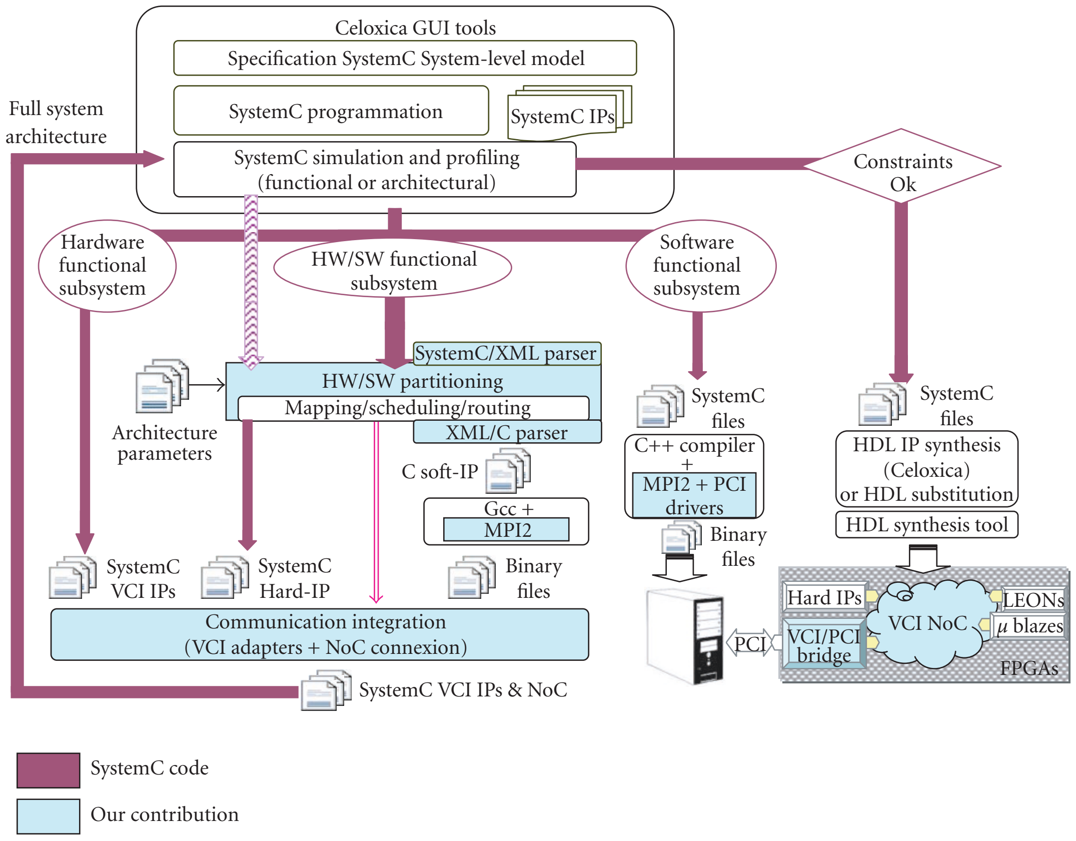

Figure 1: Top-down design flow.

system are targeted to the codesign side, as we need to optimize and well balance hardware and embedded software to meet several stringent design constraints simultaneously: hard real-time performance, low power consumption, and low resources.

Considering the software side (step (2)), the SystemC IPs are directly compiled to become binary files targeted to the host processor. This set of software tasks communicates with the remaining IPs contained in the FPGA platform through the PCI bus. Because software components run on processors, the SystemC abstract communication needed to describe the interconnection between the software and hardware components is totally different from the existing abstraction of wires between hardware components as well as the function calls abstraction that describes the software communication.

In this part, the communication is abstracted as an API which calls PCI bus drivers through an operating system layer. The API hides hardware details such as interrupt controllers or memory and input/output subsystems. We have implemented the message passing interface (MPI-2) library on the host processor and on the embedded processors of our platform [26]. MPI-2 is our HW/SW interface API.

Step (3) is the performing of our SCXML parser tool which allows to convert a given SystemC source code into an XML intermediate representation. The XML format is a subset of the standardized SPIRIT 2.0 format [27]. The system is interpreted as a set of XML files. Each XML file contains the most important characteristics of a SystemC IP, such as

(i) name, type, and size of each in/out ports, name and type of processes declared in the constructor, and also the sensitivity list of each process;
(ii) name and type of IPs building a hierarchical IP, the names of connections between the sub-IPs, and the binding with the IP ports.

Both XML files and profiling reports from Celoxica tool are treated by our HW/SW partitioning tool (step (4)) in order to partition IPs as hardware or software according to the architecture parameters and constraints. After this step we use SynDEx tool (step (5)) to perform an automatic mapping, routing, and static scheduling of IPs on the software and hardware architecture based on a predefined NoC topology [28]. The different SynDEx inputs are the following.

(i) A hierarchical conditioned data-flow graph of computing operations and input/output operations. The operations are just specified by the type and size of input/output data and execution time of the IPs. The XML files and profiling reports are parsed to produce these inputs. We need also to provide manually information on the nonexclusive execution of IPs in order to help SyndEx optimize parallelism.

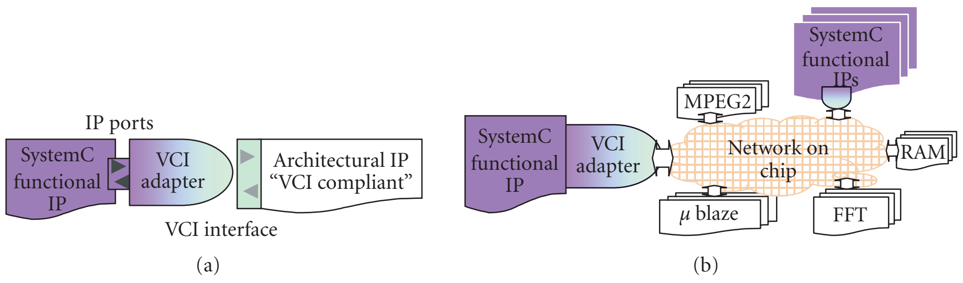

Figure 2: VCI connections of non-VCI IPs through VCI adapters. (a) "Wire" point-to-point connection. (b) NoC connection.

(ii) Specification of the heterogeneous architecture as a graph composed of software processors and hardware processors, interconnected through communication medias. Processors characteristics are supported tasks, their execution duration, worst-case transfer duration for each type of data on the interconnect. The profiling reports and architecture parameters are parsed to produce these inputs.

SynDEx implements the IPs onto the multicomponent architecture through a heuristic mapping, routing, and scheduling. After the implementation, a timing diagram gives the mapping of the different IPs on the components and the real-time predicted behavior of the system. The communication links are represented in order to show all the exchanges between processors; they are taken into account in the execution time of IPs. The mapping/routing code generated by SynDEx tool is then parsed (step (6)) in order to manage the NoC configuration and to switch software IPs to the XML/C parser. This parser translates the XML markups to C code with either RTOS calls or a static scheduling provided by SynDEx tool. With our SCXML and XML/C parsers, we obtain an embedded C generation tool (SCEmbed) from SystemC. This SCEmbed tool has about 5000 C++ and JAVA code lines. This tool and its XML format can be easily adapted to a different RTOS.

The embedded C code is then treated in step (7) with the Gcc compiler in order to obtain binary executables for the embedded processors. As the C software IPs are mapped on several heterogeneous processors, they need to use a communication library (MPI-2).

In the communication integration (step (8)), the identified SystemC hardware IPs are completed with our SystemC VCI adapter library. This point is detailed in the section below. Then point-to-point communications are established between the new VCI-compliant IPs and the VCI hardware IPs through the VCI NoC. We use SynDEx configuration information to initialize the VCI adapters, plug the IPs on the NoC, and load the binary code of the software IPs on their corresponding processor memory. Once all the SystemC architecture is produced, we can either simulate it back in the Celoxica tool for evaluation. After validation, we continue with the implementation step.

The last hardware synthesis step plays a very important role in the methodology described above. There have been various research efforts to come up with a good hardware compiler which can generate a synthesizable HDL from high-level C/SystemC specifications. The Agility Compiler from Celoxica can help the generation of synthesizable VHDL from SystemC. The final product of the design flow is a set of binary files representing programs for the host processor, LEON and Microblaze (Xilinx) processors and FPGAs. These files can be loaded onto the respective components of the prototyping platform (FPGA boards) to build a prototype with a real-time communication system.

## 3. HW/SW INTERFACE CODESIGN

### *3.1. Introduction*

An SoC can include specific hardware subsystems and one or several CPU subsystems to execute the software tasks. The SoC architecture includes hardware adapters (bridges or communication coprocessors) to connect the CPU subsystems to other subsystems. The HW/SW interface abstraction must hide the CPU. On the software side, the abstraction hides the CPU under a low-level software layer ranging from basic drivers and I/O functionality to sophisticated operating system. On the hardware side, the interface abstraction hides CPU bus details through a hardware adaptation layer generally called the CPU interface. This can range from simple registers to sophisticated I/O peripherals including direct memory access queues and complex data conversion and buffering systems.

### *3.2. Hardware-to-hardware interface synthesis: VCI adaptation methodology*

We show in Figure 2 the way to establish a communication between IPs with different abstraction levels. We consider here functional IPs and architectural IPs.

The connection can be through wires or through an NoC. The VCI adapters library aims to simplify the (re)use of functional IPs (non-VCI compliant) in any SoC based on the VCI protocol. This adapter library is designed in order to change neither the IP cores nor their interface description.

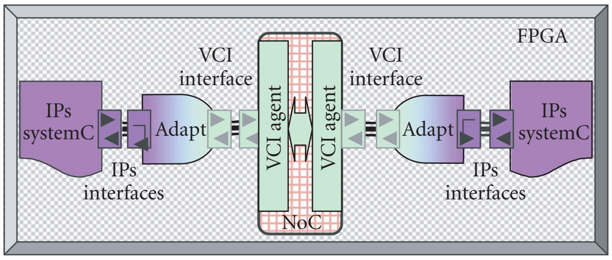

Figure 3: Layers between heterogeneous interfaces of two sets of IPs.

The generic architecture shown in Figure 3 helps to clarify the relationship between two hardware IPs connected through a sophisticated VCI NoC. The communication between heterogeneous component interfaces imposes the existence of a wrapper on each side of the communication media (bus or NoC). This wrapper behaves like a bridge which translates the RTL interface between the media and the component. These wrappers (agents) have to be compatible with VCI interface to build a standard media. Thus, an initiator wrapper is connected to VCI initiator ports of a master IP and a target wrapper is connected to VCI target ports of a slave IP.

Considering that these two VCI wrappers are available, the interface synthesis of SystemC functional IPs is a set of steps to replace a primitive channel with a refined channel in order to connect it to the wrappers. A refined channel will often have a more complex interface (e.g., VCI) than the primitive channel previously used. The main step in refining the interfaces is to create adapters that connect the original modules to the refined channel. Adapters can help to convert the interfaces of the IPs instances into VCI interfaces. The interface refinement can be made more manageable if new interfaces are developed without making changes to their associated module. The adapter translates the transaction-oriented interface consisting of methods such as *write(data)* into VCI RTL-level interface for hardware IPs. Figure 3 depicts the use of adapters to connect functional IPs to the NoC VCI agents. Hook arrow boxes indicate the interface provided by the adapters while the rightleftarrows square boxes represent ports. Our contribution consists in the design of VCI master adapters and VCI slave adapters which manage the VCI initiator and VCI target interfaces, respectively. We have chosen a convention that each SystemC output port is an initiating port of transaction and each input port is a target port. Thus, the release of a transaction results in a nonblocking write of data on the output port for an sc_signal and in a blocking write for an sc_fifo. This corresponds to the semantics of the SystemC sc_signal and sc_fifo primitive channels. Thus, initiating ports of functional IPs are connected to a master adapter and target ports are connected to a slave adapter. In this case, several IPs may be connected to the same adapter.

The adaptation methodology approach is implemented using a micronetwork stack paradigm, which is an adaptation of the OSI protocol stack.

### 3.2.1. *Application layer*

This layer describes the functional behaviour of a complex system. A system is a set of functional IPs with behavioural models, not architectural IPs such as processors or memories. The communication mechanism is performed with classical read(*data*) and write(*data*) SystemC primitives without additional parameters and no protocol implementation.

### 3.2.2. *VCI adapter layer*

The VCI adapter layer is responsible for converting an IP interface towards a lower-level interface. A VCI adapter core can manage different ports of different non-VCI-compliant IPs. Functional hardware IP ports are implemented as a memory segment accessed through its VCI adapter. The VCI adapter layer is composed of the following sublayers.

#### (a) *Presentation layer*

This layer is responsible for translating an abstract data-type port towards a SystemC synthesizable data-type port.

#### (b) *Session layer*

The session layer generates a single VCI address between two ports connected to each other in the system-level description. This address is divided into two fields: the most significant bits (MSB) identify the destination wrapper and the least significant bits (LSB) identify the local offset at destination. Each agent of the NoC needs to be configured in order to know the separation position between MSB and LSB, and thus be able to perform address translation to correctly route the data to be sent.

The LSB field is itself divided first according to the target IP port addressed among the different IP ports connected to the same VCI adapter, and second according to the local address segment managed by the transport layer. VCI adapter address is finally divided into three fields.

(i) Field 1: agent number is the address field decoded/generated by the NoC agents and routed in the NoC.
(ii) Field 2: port number is the address field decoded/generated by the VCI adapter to switch data to the corresponding IP port.
(iii) Field 3: word number is the address field decoded/generated by the transport layer. It represents the address in the memory segment of the selected port.

The address translation of each VCI adapter is configured during its connection to the NoC with its NoC agent number and its port number. Already VCI-compliant IPs have to provide configurability of addresses in order to communicate to any IP on the NoC. This configuration of IP VCI adapters is performed during VCI adapter integration step based on SynDEx mapping/routing information. For already VCI-compliant IPs, addresses are provided manually as it is IP dependant. This is the second of the very few non-fully automated parts of the flow.

(c) *Transport layer*

The basic function of the transport layer is multiple: it accepts data from the IP ports, splits them into smaller units (segments) according to the VCI master adapter data bus size, passes them to the network layer, and ensures that the pieces all arrive correctly at the other end. In addition, the transport layer is responsible for the generation of the segment number which constitutes the third field of VCI address.

(d) *Network layer*

This layer is responsible for the identification of the initiating port. In the case of a multiport master adapter, the network layer launches an arbiter to solve the conflicts and ensures that only one port can have an access to the resource (media). The second treatment is the operation of transfers multiplexing and demultiplexing.

(e) *Datalink layer*

The datalink layer defines the format of data on the interface and the communication protocol. It is responsible for VCI transactions.

### 3.2.3. Physical layer

The physical layer is the physical way of communication. Wires are used for point-to-point connection between VCs. An NoC is used for sophisticated communications.

We have synthesized an example of a simple producer/consumer on the Xilinx FPGA technology. We have used the PVCI master/slave adapters with an 8-bit data bus and a 5-bit address bus on both IPs. Each adapter unit allows two IP data bus connections of 64-bit and 32-bit size, respectively, with a static IP port priority management. This implementation was performed with Xilinx Virtex II xc2v3000-6 technology. We present here the post placed/routed results. We have obtained a master adapter cost of 489 units of 4-entries logic and 136 flipflop units, with a 100 MHz clock frequency. So, it occupies 1.7% of the FPGA. The slave adapter requires 144 4-entries logic units and 204 flipflop units with the same clock frequency. It needs 0.46% of the FPGA resources. A master adapter is four times larger than a slave adapter.

## 3.3. Software-to-software interface synthesis

For embedded software, the SystemC read(*data*) and write (*data*) are implemented with POSIX elements in the case of dynamic scheduling with an RTOS and message passing interface (MPI) elements in the case of static scheduling. We have used the POSIX compliant real-time embedded multiprocessor scheduler (RTEMS) as RTOS.

For RTEMS, the read and write primitive functions are replaced with the *rtems_message_queue_receive()* function and the *rtems_message_queue_send()* function, respectively. The sc_fifo blocking read() function is implemented with the RTEMS_WAIT option set in rtems_message_queue_receive(). The nonblocking sc_signal functions are implemented for RTEMS through message queues which are flushed before each data write. The nonblocking read is implemented with the option RTEMS_NO_WAIT.

For an RTOS-less solution, the SystemC read(*data*) and write(*data*) are implemented as one-sided remote memory access (RMA) with the MPI MPI_put(*data*) primitive only. The blocking mechanism for sc_fifo is implemented with the MPI_wait() primitive which waits for an acknowledgment.

## 3.4. Software-to-hardware interface synthesis

For software IP on embedded CPUs, communication with the NoC VCI agent is managed with dual-ported memory buffers and DMA from its VCI adapter (dedicated to the CPUs) directly connected to this dual-ported memory. The DMA is controlled by software driver subroutines overloading MPI or RTEMS message queues.

In the case of host processor, the read(*data*) and write(*data*) SystemC primitives are overloaded in order to call the PCI driver services through MPI calls. This software driver configures the hardware DMA which manages the data transactions between host memory and the NoC on the prototyping board through the VCI/PCI bridge.

Using one-sided RMA is an efficient implementation solution of MPI [26, 29] and the SystemC programming model is also very well suited to RMA implementation as sc_signal reads and writes are not correlated. In practice, efficiency of HW/SW interfaces is obtained with a direct integration of SystemC high-level communication library in hardware, that is, by a joint optimization of the implementation of the SystemC programming model with the MPI_put() and MPI_wait() primitives (RMA model) as well as with the underlying NoC design. The RMA mechanism is limited to write-only transfers between IPs allowing the design of a specific NoC optimized for those transfers with DMA. This approach is similar to the joint optimization of compilers and microarchitectures of microprocessors.

We have designed optimized network interfaces for two custom NoC [30] with write-only communications, connected to Microblazes, LEONs, and PowerPCs processors through their dedicated ports. The MPI_put() primitive needs two I/O access to configure the DMA of the network interface and to launch the DMA transfer in the NoC. Thus the MPI_put() takes only 8 processor clock cycles: 6 clock cycles to prepare the DMA configuration and 2 clock cycles for I/O access. In that case, the result for the SystemC sc_signal write() primitive is 25 clock cycles of overhead comprising two MPI_put() executions (one for the control and one for the data), that is, 16 clock cycles, and 9 clock cycles to prepare the data to be transferred. Also there is no overhead for the SystemC sc_signal read() which is only a local variable access due to the RMA mechanism.

For comparison, the main difference between MPI RMA subset and DSM API from Celoxica presented in Table 1 is that the MPI_put is a nonblocking mechanism which in conjunction with MPI_Wait can implement a blocking

Table 1: DSM and RMA MPI subset comparison.

| DSM | MPI |
|---|---|
| DsmInit() | MPI_Init() |
| DsmExit() | MPI_Finalize() |
| DsmWrite() & DsmRead() | MPI_Put() & MPI_Wait() |
| DsmPortS2HOpen() | — |
| — | MPI_Barrier() |

mechanism, compared to the DsmWrite and DsmRead which are only blocking mechanisms. Also the DSM API is a two-sided communication compared to the one-sided RMA subset.

## 4. GENERATION OF EMBEDDED C CODE

### 4.1. Generation process

In modern complex SoCs, the software as an integral part of the SoC is gaining more and more importance. At the system level, the system is composed of a set of hierarchical behaviors connected together through channels. However, for the implementation, many designers use a task-based approach, where the tasks are scheduled by a real-time kernel. A whole system design is composed of a set of globally asynchronous/locally synchronous reactive processes that concurrently perform the system functionalities.

Inside the SystemC process code, only wait() primitives are allowed and processes lack a sensitivity list except for one signal which is considered as a clock. Therefore, a process will only block when it reaches a wait(). These restrictions that we have required are only for the code involved in the embedded HW/SW partitioning process. They help our SCEmbed tool to generate the embedded C code [30]. These restrictions on SystemC coding are also required by Celoxica tools for the SystemC synthesis.

The XML format used by the XML/C parser is easily adaptable for a new target RTOS. The main idea behind is to redefine the SystemC class library elements for the new target RTOS. The original code of these IPs calls the SystemC kernel functions to support process concurrency and communication. The new code calls the embedded RTOS functions that implement the equivalent functionality. Thus, SystemC kernel functions are replaced either by typical RTOS functions or through direct generation of a statically scheduled code. The functional behavior is not modified during the hardware, software, and interfaces generation.

We illustrate the C generation process for the RTOS target with a producer/consumer example. The SystemC main code named sc_main() is converted to the RTEMS RTOS main code "init." The channels are implemented with message queues for blocking sc_fifo channels and shared variables for nonblocking sc_signal channels. The clock in the SystemC code is converted into a task sending an event value broadcasted on a message queue. All the tasks read this clock message queue for there synchronization.

SystemC concurrent processes need to be converted into RTOS-based tasks. We instantiate the child tasks in a parent one corresponding to the SC_MODULE in the system specification. This step is illustrated by our example in Figure 4. The producer and the consumer instances are converted into Tprod and Tcons parent tasks. In RTEMS, each parent task (SC_MODULE in systemC) launches the child tasks (processes in SystemC) and an additional task which is responsible for interprocess communication. This task is created to manage sc_out ports writing delay corresponding to the behavioral delay of the SystemC write function (the data are validated after the wait event). The RTEMS equivalent code of the SystemC Producer is shown in Figure 5.

At the system level, synchronization is implemented using channels or SystemC events. During the generation process, the RTOS model provides routines to replace the SystemC synchronization primitives.

In the case of POSIX generation, synchronization between tasks is managed by semaphores for sc_signal implementation with global shared variables. A special clock management task is generated which schedules the two-step signal assignment process in order to respect the semantic of sc_signal. All the signal assignments are performed simultaneously after all the processes are stopped on a wait() instruction. The wait() instruction is implemented by a semaphore synchronization. The clock task is waiting for all the tasks which are sensitive to the same clock to stop on a wait instruction. Then the second step is performed by this clock management task, which corresponds to the assignment of all the shared global variables with the temporary variables assigned by the different blocked tasks. These blocked tasks are then freed and can read the shared global variables which are now updated. This mechanism is generated for each independent clock in the whole system. When different tasks are mapped on different processors, we assume that they communicate through asynchronous sc_fifo channels. Otherwise, the clock management tasks of the different processors have to be synchronized before the assignment of the shared global variables.

The second approach uses an RTOS-less static scheduling. In this solution, the SystemC scheduler is replaced by our custom simulation engine optimized for embedded applications. This scheduler is called from each wait() instruction or from sc_fifo blocking read() or write() functions. This scheduler also manages the synchronization of clock sensitive tasks with barrier primitives.

A channel implementation library is provided for all the solutions. Up to now, only primitive channels are available (sc_signal, sc_fifo). There are three versions of implementation for each channel: SW/SW, HW/HW, and SW/HW. SW/SW channels are direct shared variables or message queues implementation. HW/HW channels are RTL-level NoC wrappers. SW/HW are C drivers for embedded processors connected to the NoC.

### 4.2. Application example

In order to evaluate the proposed technique, two designs have been experimented for the SW part in this section.

```
                                    // RTEMS declaration part
int sc_main (){                   {rtems_task init(rtems_task_argument* unused){

                                  ⎧medium = rtems_build_name("m," "e," "d," "i");
sc_signal <char> medium;          ⎨rtems_message_queue_create(medium, . . . ,
                                  ⎩   & mediumID);

                                  ⎧Tclk = rtems_build_name("H," "L," "G," "A");
                                  ⎪rtems_task_create(Tclk, . . . , & TclkID);
sc_clock clock("clock");          ⎨clock = rtems_build_name("c," "l," "o," "c");
                                  ⎪rtems_message_queue_create(clock, . . . , & clockID);
                                  ⎩Port_clock[0] = clockID;

  producer prod_inst("prod");
                                  ⎧Tprod = rtems_build_name("p," "r," "o," "d");
    prod_inst.out(medium);        ⎪rtems_task_create(Tprod, . . . , & TprodID);
    prod_inst.clk(clock);         ⎨Port_prod_inst[1] = mediumID;
                                  ⎩Port_prod_inst[0] = clockID;

consumer
                                  ⎧Tcons = rtems_build_name("c," "o," "n," "s");
    cons_inst("Consumer");        ⎪rtems_task_create(Tcons, . . . , & TconsID);
    cons_inst.in(medium);         ⎨Port_cons_inst[0] = mediumID;
    cons_inst.clk(clock);         ⎩Port_cons_inst[1] = clockID;

                                  ⎧rtems_task_start(TclkID, clock_task, & Port_clock);
sc_start(−1);                     ⎨rtems_task_start(TprodID, producer, & Port_prod_inst)
                                  ⎩rtems_task_start(TconsID, consumer, & Port_cons_inst);
return 0;
}                                     rtems_task_delete(RTEMS_SELF);
                                    }
           (a) before                              (b) after
```

Figure 4: From SystemC main code to RTEMS code.

The first one is a simple consumer/producer case with two SystemC components linked together. The second one, a more realistic case, is a CDMA radiocommunication example. The consumer/producer system description has about 86 SystemC code lines and the CDMA system description has about 976 SystemC code lines. The CDMA includes 7 modules with 8 concurrent processes. Both examples have been implemented in a SPARC-based platform that includes 1 MB SDRAM and a LEON2 processor synthesized on one 4Mgate Xilinx FPGA with 128 KB of RAM. The open source POSIX-compliant RTEMS operating system has been selected as the target embedded RTOS.

The CDMA system has 7 modules: the top (CDMA), one module that generates samples, three modules that compute the QPSK modulation, the THR and the interleaving, one that models the real environment channel behavior by introducing noise, and the last ones that do the reverse treatment that is deinterleaving, ITHR and demodulation. All the modules work in a pipelined dataflow way. Several channel models have been implemented with our design flow. The CDMA application example uses one of them: a nonblocking channel (the sc_signal channel). The proposed channel models have different implementations depending on the HW/SW partition. Several experiments have been performed with semaphores, mutex condition variables, and signals in order to synchronize threads with RTEMS.

Table 2 shows the code size of the different codes on the different operating systems. Table 3 presents their binary size and Table 4 their average execution time per treatment iteration.

```
/*===Myproducer.h File==*/
class prod : public sc_module
 {
   public:
     sc_out<char> out;
     sc_in< bool > clk;

int i;
void main();
SC_HAS_PROCESS(prod);
prod(· · · ): sc_module(name){

    SC_THREAD(main);
    sensitive_pos ≪ clk;
     }
};
```

```
rtems_task prod(rtems_task_argument *port) {
Tsend = rtems_build_name("F," "C," "o," "m");
Tmain = rtems_build_name("m," "a," "i," "n");

rtems_task_create(Tsend[0], . . . , & TsendID);
rtems_task_create(Tmain[1], . . . , & TmainID);
rtems_task_start(TsendID, ComTask, & port);
rtems_task_start(TmainID, main, & port);

rtems_task_delete(RTEMS_SELF);
}

rtems_task main(rtems_task_argument *port){
// main code
rtems_task_delete(RTEMS_SELF);
}
// Communication task
rtems_task ComTask (rtems_task_argument *port)
{       // task code
}
```

Figure 5: Producer RTEMS code.

Table 2: Line number of prod/cons and CDMA source code.

| | SystemC Linux | POSIX Linux | RTEMS LEON | POSIX LEON | Static C Linux | Static C LEON |
|---|---|---|---|---|---|---|
| Prod/Cons | 86 | 130 | 203 | 161 | — | — |
| CDMA | 976 | 1350 | 1479 | 1387 | 950 | 950 |

Table 3: Binary code size of prod/cons and CDMA.

| | SystemC Linux | POSIX Linux | RTEMS LEON | POSIX LEON | Static C Linux | Static C LEON |
|---|---|---|---|---|---|---|
| Prod/Cons | 592 K | 14 K | 106 K | 83 K | — | — |
| CDMA | 1.8 M | 32 K | 119 K | 97 K | 188 K | 12 K |

Table 4: Execution time of prod/cons and CDMA.

| | SystemC Linux | POSIX Linux | RTEMS LEON | POSIX LEON | Static C Linux | Static C LEON |
|---|---|---|---|---|---|---|
| Prod/Cons | 43 $\mu$s | 81 $\mu$s | 2.43 ms | 1.85 ms | — | — |
| CDMA | 170 $\mu$s | 310 $\mu$s | 12.5 ms | 9.2 ms | 17 $\mu$s | 153 $\mu$s |

In Table 2, the number of lines of the generated embedded C code is nearly the double for the first simple case which includes 27% of SystemC primitives. For the CDMA, the generated code size is nearly half more important with only 13% of SystemC primitives. The size of the embedded C generated code is directly linked to the number of SystemC elements included in the original code. As each SystemC primitive is translated with a set of embedded C instructions, a large proportion of read(), write(), wait(), and others primitives can result in an important size. However, the increase of the generated code size remains low. Moreover, this generated C code is entirely "readable" and can be completed or optimized manually.

In Table 3, the size of the statically scheduled code for embedded processor is nearly ten times lower than the RTOS one. Thus, when it is possible, it is more interesting to use an RTOS-less solution for embedded processors. For the Linux implementation, the kernel is not included in the code, thus its size is lower than for the standalone one which includes its own kernel.

In Table 4, the code execution time on the embedded processor with a static scheduling is nearly 60 times faster than the RTOS one. We have to consider here that the CDMA application highly communicates and thus highly requests RTOS services with context switching for each communication. In addition, the validation of the embedded software can be operated on the host processor, through direct POSIX execution, as it obtains comparable execution times compared to SystemC execution. We obtain also better execution times for a dedicated static scheduling that is nearly ten times faster than pure SystemC execution times.

It is thus possible to evaluate more rapidly the whole SystemC model by parsing it in C and execute it on the user computer instead of using pure SystemC simulations. The results collected in Tables 2, 3 and 4 show the feasibility of our SystemC parsing process to POSIX or static scheduling C code.

We have also experimented different CDMA multiprocessor implementations with static scheduling in order to

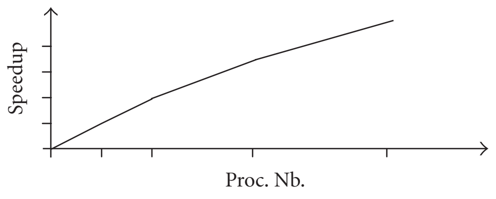

Figure 6: Speedup of CDMA application.

Table 5: Average design time of prod/cons and CDMA.

| Solutions | Automated | Manual | |
|---|---|---|---|
| | | *POSIX C* | *Static C* |
| Prod/Cons | 1.5 H | 6 H | 1.5 days |
| CDMA | 3 H | 2 days | 4 days |

evaluate the impact of HW/SW communication in term of time overhead. This overhead includes software delays from device drivers and hardware delays due to the NoC crossing. We have experimented several configurations with 1, 2, 4, and 7 processors connected with a one-dimension linear NoC with two processors per node. The speedup obtained is presented on Figure 6. The overhead of HW/SW communication (25 clock cycles for a write) added to the NoC crossing time (almost one clock per NoC node crossed) makes the impact of communication low compared to the execution time of the CDMA functions on the different processors. We obtain a speedup of 1.8 with 2 processors and 5 with 7 processors.

These results show the low implications of such a higher-level interface approach.

These previous design experiments conducted to measures of design cycle times used for time-to-market evaluation. Several groups of students have implemented these examples with the SystemC code as a starting point. As we evaluate here only the mapping of software IPs on embedded processors, we have provided also the VHDL code of the hardware platform. This platform is composed of LEON-2 processors and an NoC. We have compared the implementation time between our automated flow and two manual solutions. These experiment results presented in Table 5 show consequent implementation time differences. Even for a simple case, the manual design of a multithreaded (POSIX/MPI) code needs a debugging step which is time consuming. This is of course even more important for a fully static code which needs to design the scheduling of IPs and their communication (MPI). This debugging step needs to be conducted first as pure C/MPI code on the user computer and then on the VHDL platform. Moreover, even if no VHDL is designed here, they need at least to configure the platform in terms of addresses, and thus validate the entire system. A more complex system with VHDL integration would conduct to even more time-consuming implementation.

## 5. CONCLUSION

This paper deals with the idea of unifying the use of SystemC to implement both hardware and embedded software. We propose an automated HW/SW codesign methodology which reduces the design time of embedded systems.

The proposed methodology uses the redefinition of SystemC elements to generate the embedded software. A first solution is to replace each SystemC element by typical RTOS functions and MPI primitives. This solution provides a significantly smaller code size than the equivalent SystemC code size. A second complementary solution is to generate a statically scheduled stand alone C code which exhibits better results in code size and execution time.

The main advantage of this methodology relies on the optimization of the different steps of the flow. These steps are jointly designed and optimized by integrating the features of the SystemC programming model. Actually, the NoC architecture is jointly designed with the MPI software of the communication layer. This approach is similar to the joint optimization of compilers and microarchitectures of microprocessors. According to the previous experiments, our optimized flow exhibits efficient results in terms of execution times and hardware resources. Finally, this automated flow can help to obtain fast design cycle times.

Future works concern the full support of the SPIRIT standard as well as the improvement of SynDEx mapping and routing solution.

## REFERENCES

[1] P. Sánchez, "Embedded SW and RTOS," in *Design of HW/SW Embedded Systems*, E. Villar, Ed., University of Cantabria, Santander, Spain, 2001.
[2] Forte Design Systems, "Cynthesizer 3.0," http://www.forteds.com/.
[3] Celoxica, "Agility compiler user guide," Celoxica, 2005.
[4] Mentor Graphics, "CatapultC," http://www.mentor.com/.
[5] C. S. Ananian, "SiliconC: a hardware backend for SUIF".
[6] CoWare, "SPW and Platform Architect," http://www.coware.com/.
[7] R. K. Gupta, *Co-Synthesis of Hardware and Software for Digital Embedded Systems*, Kluwer Academic, Norwell, Mass, USA, 1995.
[8] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck, "Software synthesis for DSP using Ptolemy," *Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology*, vol. 9, no. 1-2, pp. 7–21, 1995.
[9] F. Baladin, M. Chiodo, P. Giusto, et al., *Hardware-Software Codesign of Embedded Systems: The POLIS Approach*, Kluwer Academic, Norwell, Mass, USA, 1997.
[10] Rational, http://www.rational.com/uml/index.html.
[11] D. Harel, H. Lachover, A. Naamad, et al., "STATEMATE: a working environment for the development of complex reactive systems," *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 403–414, 1990.
[12] F. Boussinot and R. de Simone, "The ESTEREL language," *Proceedings of the IEEE*, vol. 79, no. 9, pp. 1293–1304, 1991.
[13] T. Grötker, S. Liao, G. Martin, and S. Swan, *System Design with SystemC*, Kluwer Academic, Norwell, Mass, USA, 2002.

[14] D. Desmet, D. Verkest, and H. De Man, "Operating system based software generation for systems-on-chip," in *Proceedings of 37th Design Automation Conference*, pp. 396–401, Los Angeles, Calif, USA, June 2000.

[15] F. Herrera, H. Posadas, P. Sanchez, and E. Villar, "Systemic embedded software generation from systemC," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE '03)*, pp. 10142–10149, Munich, Germany, March 2003.

[16] H. Yu, R. Dömer, and D. Gajski, "Embedded software generation from system level design languages," in *Proceedings of the Asia and South Pacific Design Automation Conference (ASP-DAC '04)*, pp. 463–468, Yokohama, Japan, January 2004.

[17] F. Pogodalla, R. Hersemeule, and P. Coulomb, "Fast Prototyping: a system design flow for fast design, prototyping and efficient IP reuse," in *Proceedings of the 7th International Conference on Hardware/Software Codesign (CODES '99)*, pp. 69–73, Rome, Italy, May 1999.

[18] OCP Adoption Adds Value to Prosilog. www.prosilog.com/news/press/documents/.

[19] J.-Y. Brunel, W. M. Kruijtzer, H. J. H. N. Kenter, et al., "COSY communication IP's," in *Proceedings of 37th Design Automation Conference*, pp. 406–409, Los Angeles, Calif, USA, June 2000.

[20] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, and S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic, Norwell, Mass, USA, 2000.

[21] "Virtual Component Interface Standard (OCB 2 1.0)," VSIA *on-Chip Bus Development Working Group*, March 14, 2000.

[22] W. Klingauf, "Systematic transaction level modeling of embedded systems with systemC," in *Proceedings of Design, Automation and Test in Europe (DATE '05)*, vol. 1, pp. 566–567, Munich, Germany, March 2005.

[23] W. Gropp, E. Lusk, and R. Thakur, *Using MPI-2 Advanced Features of the Message Passing Interface*, MIT Press, Cambridge, Mass, USA, 1999.

[24] S. Ouadjaout and D. Houzet, "Easy SoC design with VCI systemC adapters," in *Proceedings of the EUROMICRO Systems on Digital System Design (DSD '04)*, pp. 316–323, Rennes, France, August-September 2004.

[25] S. Ouadjaout, M.-F. Albenge, and D. Houzet, "VSIA interface cosynthesis," in *Proceedings of the 1st IEEE International Workshop on Electronic Design, Test and Applications (DELTA '02)*, pp. 43–46, Christchurch, New Zealand, January 2002.

[26] SPIRIT Consortium, "SPIRIT V2.0 Alpha release," 2006.

[27] T. Grandpierre and Y. Sorel, "From algorithm and architecture specifications to automatic generation of distributed real-time executives: a seamless flow of graphs transformations," in *Proceedings of 1st ACM and IEEE International Conference on Formal Methods and Models for Co-Design (MEMOCODE '03)*, pp. 123–132, Mont Saint-Michel, France, June 2003.

[28] S. G. Ziavras, A. V. Gerbessiotis, and R. Bafna, "Coprocessor design to support MPI primitives in configurable multiprocessors," to appear in *Integration, the VLSI Journal*.

[29] S. Evain, J.-P. Diguet, and D. Houzet, "$\mu$Spider: a CAD tool for efficient NoC design," in *Proceedings of 22nd Norchip Conference*, pp. 218–221, Oslo, Norway, November 2004.

[30] S. Ouadjaout and D. Houzet, "Embedded hardware/software generation from high level design languages," in *Proceedings of IEEE International Computer Systems & Information Technology Conference (ICSIT '05)*, Algiers, Algeria, July 2005.

**Salim Ouadjaout** received the M.S. degree in computer science from the National Institute of Computers (INI), Algeria, in 2000, and the M.S. degree from INP, ENSEEIHT, Toulouse, France, in 2001. He is a Ph.D. candidate in the Electrical and Computer Engineering Department at the Institute of Electronics and Telecommunication, Rennes, France. He is also working as a Research Engineer at M3Systems, Inc. He has been an ACM Student Member. His research interests include design methodologies, interface synthesis, micronetworks for SoC, and embedded multiprocessors SoC.

**Dominique Houzet** received the M.S. degree in computer sciences in 1989 from Paul Sabatier University, Toulouse, France, and the Ph.D. degree and HDR degree in computer architecture in 1992 and 1999, both from INPT, ENSEEIHT, Toulouse, France. He worked at IRIT Laboratory and ENSEEIHT Engineering School from 1992 to 2002 as an Assistant Professor and at IETR Laboratory INSA Engineering School in Rennes from 2002 to 2006 and also as a Digital Design Consultant with SME and large companies. He is now a Professor at LIS-INPG, Grenoble. He has published a number of research papers in the area of parallel computer architecture and SoC design and a book on VHDL principles. His research interests include codesign and SoC design methodologies applied to image processing and radiocommunications. He is a Member of the IEEE Computer Society.