Multiprocessor System-on-chip Platforms: a Component-Based Design Approach[1]

W. Cesário, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A.A. Jerraya
TIMA Laboratory, SLS Group
46, av. Félix Viallet
38031 Grenoble Cedex – France
{Wander.Cesario,Ahmed.Jerraya}@imag.fr
phone: (+33)476574870
fax:(+33)476473814

L. Gauthier
Institute of Systems & Information Technologies
Fukuoka SRP Center Building
7F, 2-1-22, Momochihama Sawara-ku
814-001 Fukuoka City - Japan

M. Diaz-Nava
STMicroelectronics
AST Grenoble Lab
12, rue Jules Horowitz, BP217
38019 Grenoble Cedex - France

ABSTRACT

Component-based design provides primitives to build complex architectures from basic components. This bottom-up approach allows design-architects to reuse efficient custom solutions with best performances. This paper presents a high-level component-based methodology and design environment for multiprocessor system-on-chip architectures. The design environment provides automatic wrapper-generation tools able to synthesize hardware interfaces, device drivers, and operating systems that implement a high-level interconnect API. This approach, experimented over the design of a VDSL system, shows a drastic design time reduction without any significant efficiency loss in the final circuit.

1. INTRODUCTION

Modern system-on-chip (SoC) design shows a clear trend towards integration of multiple processor cores, the SOC System Driver section of the "International Technology Roadmap for Semiconductors" (http://public.itrs.net/) predicts that the number of processor cores will increase four-fold per technology node in order to match the processing demands of the corresponding applications. Typical multiprocessor SoC (MPSoC) applications like network processors, multimedia hubs and base-band telecom circuits have particularly tight time-to-market and performance constraints which require a very efficient design cycle.

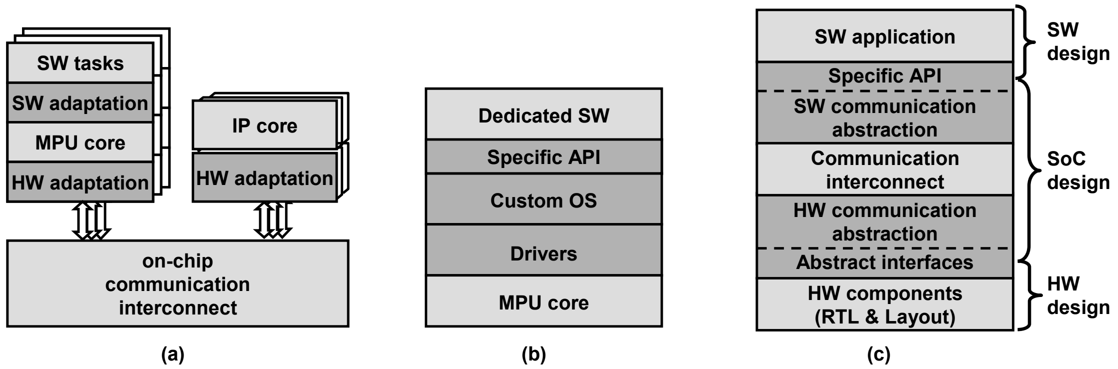

Figure 1. MPSoC platform (a), software stack (b) and concurrent development environment (c)

Our conceptual model of the MPSoC platform is composed of four kinds of components: software tasks, processor and intellectual property (IP) cores and a global on-chip interconnect IP (see Figure 1a). Moreover, to complete the MPSoC platform we must also include hardware and software elements that adapt each platform component to each other. This

[1] This article is derived in part from [1].

MPSoC platform is quite different from single-master processor SoCs. The implementation of system communication becomes much more complicated since heterogeneous processors are involved in communication and complex communication protocols and topologies are used. The hardware adaptation layer must deal with some specific issues:

1. In single-master processor platforms most peripherals (excepting DMA controllers) operate as slaves in respect to the shared communication interconnect. MPSoC platforms may use many different types of processor cores; in this case, sophisticated synchronization is needed to control shared communication between several heterogeneous masters.
2. While single-master processor platforms use simple master/slave shared-bus interconnections, MPSoC platforms often uses several complex system buses or micro-networks as global interconnect. In MPSoC platforms we can separate computation and communication design by using communication coprocessors and profiting from the multi-master architecture. Communication coprocessors/controllers (masters) implement high-level communication protocols in hardware and execute them in parallel with the computation executed on processor cores.

Application software is generally organized as a stack of layers that runs on each processor core (see Figure 1b). The lowest layer contains drivers and low-level routines to control/configure the platform. For the middle layer we can use any commercial embedded OS and configure it according to the application. The upper layer is an application programming interface (API) which provides some pre-defined routines to access the platform. All these layers correspond to the software adaptation layer in Figure 1a, coding application software can then be isolated from the design of the SoC platform (software coding is not the topic this paper and will be omitted). One of the main contributions of this work is to consider the same layered approach for dedicated software (often-called firmware). Firmware is the software that controls the platform, and, in some cases, executes some non-performance critical application functions. In this case, it is not realistic to use a generic OS as the middle layer due to code size and performance reasons. A lightweight custom OS supporting an application-specific and platform-specific API is required. Custom OS design automation is a new research area brought by MPSoC platform design [8].

Software and hardware adaptation layers isolate platform components enabling concurrent development as shown in Figure 1c. With this scheme, the software design team uses APIs for both application and dedicated software development. The hardware design team uses abstract interfaces provided by communication coprocessors/controllers. SoC design team can concentrate on implementing hardware and software abstraction layers for the selected communication interconnect IP. Designing these hardware and software abstraction layers is a major design effort, and design tools are lacking. Established EDA tools are not well adapted to this new MPSoC design scenario, and consequently many challenges are emerging in design automation; some major issues are:

1. Higher abstraction level is needed: the register-transfer level (RTL) is not well adapted to model and verify the interconnection between multiple processor cores because this would be too much time consuming.
2. Higher-level programming is needed: MPSoCs will include hundred thousands lines of dedicated software (firmware). This software cannot be programmed at the assembler level as today.
3. Efficient hardware/software (HW/SW) interfaces are required: microprocessor interfaces, bank of registers, shared memories, software drivers, and operating systems must be optimized to each application.

This paper presents a component-based design automation approach for MPSoC platforms. Section 2 introduces the basic concepts for MPSoC design and discusses some related platform and component-based design automation approaches. Section 3 details our component-based specification model and design flow for MPSoC platform design. Section 4 presents the application of this flow for the design of a VDSL circuit and the analysis of the results.

2. BASICS: SYSTEM-ON-CHIP DESIGN

2.1 SYSTEM-LEVEL DESIGN FLOW

This section gives and overview of current SoC design methodologies using the template design flow that is shown in Figure 2. The basic theory behind this design flow is the separation between communication and computation refinement for platform and component-based design [2][3]; it has five main design steps:

1. System specification: system designers and the end-customer must first agree on an informal model that must contain all application's functionality and requirements. Based on this informal model, system designers build a more formal specification that can be validated by the end-customer.
2. Architecture exploration: system designers build an executable specification model and iterate through a performance analysis loop to decide the HW/SW partitioning for the SoC architecture. This executable specification uses abstract models

for HW and SW components. For instance, an abstract software model can concentrate on I/O execution profiles, or most frequent use cases, or worst-case scheduling. Abstract hardware can be described using transaction-level models or behavioral models. This step produces the "golden" architecture model which is the customized SoC platform or a new architecture created by system designers after selecting processors, the global communication interconnect and other IP components.

3. Software design: once HW/SW partitioning is decided, software and hardware development can be done concurrently. Since the final hardware platform will not be available during software development, some kind of hardware abstraction layer (HAL) or API must be provided to the software design team.

4. Hardware design: hardware IP designers implement the functionality described by the abstract hardware models at the RT-level. Hardware IPs can use specific interfaces for a platform or standard component interfaces as defined by Virtual Socket Interface Alliance (VSIA, http://www.vsi.org).

5. HW/SW integration: SoC designers create hardware and software interfaces to the global communication interconnect. The golden architecture model must specify performance constrains to assure a good HW/SW integration. Software and hardware communication interfaces are designed to conform to these constrains.

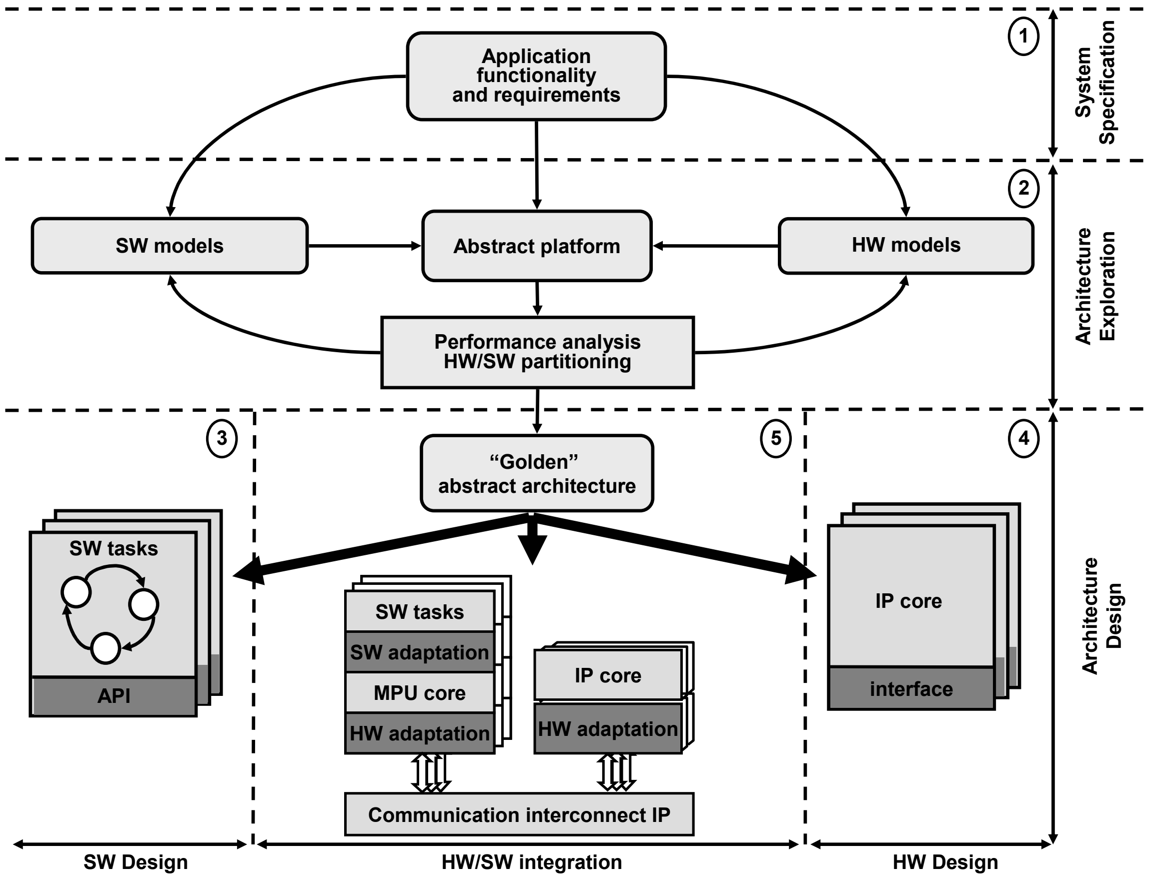

Figure 2. System-level design flow for SoCs

2.2 SoC DESIGN AUTOMATION – RELATED STRATEGIES

Many academic and industrial works provide tools for SoC design automation that covers many but not all design steps presented before. Most SoC design automation approaches can be classified into three groups: system-level synthesis, platform-based design and component-based design.

System-level synthesis methodologies are fully top-down approaches, the SoC architecture and software models are produced by synthesis algorithms from a system-level specification. COSY [4] proposes a HW/SW communication refinement process that starts with an extended Kahn Process Network model on design step (1), uses VCC [6] on step (2), callback signals over a standard RTOS for the API in (3), and VSIA interfaces for steps (4) and (5). SpecC [5] starts with an un-timed functional specification model written in extended C on design step (1), uses performance estimation for a structural architecture model on step (2), HW/SW interface synthesis based on a timed bus-functional communication model for step (5), synthesized C code for step (3), and behavioral synthesis for step (4).

Platform-based design is a meet-in-the-middle approach that starts with a functional system specification and a pre-designed SoC platform. Performance estimation models are used to try different mappings between the set of application's functional modules and the set of platform components. During these iterations, designers can try different platform customizations and functional optimizations. VCC [6] can produce a performance model using a functional description of the application and a structural description of the SoC platform for design steps (1) and (2). CoWare N2C (http://www.coware.com/) is a good complement for VCC for design steps (4) and (5). Still the API for software components and lots of architecture details must be implemented manually.

Most component-based design approaches build SoC architectures from the bottom-up using pre-designed components with standard interfaces and/or a standard bus. For instance, IBM defined a standard bus called CoreConnect (http://www.chips.ibm.com/bluelogic/), Sonics proposes a standard on-chip network called Silicon Backplane μNetwork [7], and VSIA defined a standard component protocol called VCI. When needed, wrappers adapt incompatible buses and/or component interfaces. Frequently, internally developed components are tied to in-house (non-public) standards; in this case, adopting public standards implies a big effort to redesign interfaces for old components.

We have developed a higher-level component-based design environment for MPSoC platforms that starts with a virtual architecture model composed of HW/SW components and that allows to automate design step (5), by providing automatic generation of hardware interfaces (4), device drivers, operating systems, and application programming interfaces (3). Even if this approach does not provide much help on automating design steps (1) and (2), it provides a considerable reduction of design time for design steps (3),(4) and (5) and facilitates component reuse. The key improvements over other state-of-art platform and component-design approaches are:

1. Strong support for software design and integration: the generated API completely abstracts the hardware platform and OS services. Software development can be concurrent to and independent of platform customization.
2. Higher-level abstractions: the use of a virtual architecture model allows designers to deal with HW/SW interfaces at a high abstraction level. Behavior and communication are separated in the system specification, thus, they can be refined independently.
3. Flexible HW/SW communication: automatic HW/SW interfaces generation is based on the composition of library elements. It can be used with a variety of IP interconnect components by adding the necessary supporting library.

3. COMPONENT-BASED DESIGN FOR MPSOC

3.1 DESIGN FLOW

Our design flow (Figure 3) starts with a virtual architecture model that corresponds to the "golden" architecture in Figure 2, and allows automatic generation of communication coprocessors/controllers (wrappers), device drivers, operating systems, and application programming interfaces.

The virtual architecture model is a set of virtual modules interconnected using point-to-point virtual channels and/or a communication interconnect IP. The goal is to produce a synthesizable RTL model of the MPSoC platform that is composed of processor cores, IP cores, the communication interconnect IP, and HW/SW wrappers. Interfaces of virtual components are used to automatically generate application specific hardware and software wrappers (indicated by the arrows in Figure 3). Software written for the virtual architecture specification can run without modification on the implementation because the same APIs are provided by the generated custom operating systems.

3.2 VIRTUAL ARCHITECTURE

The virtual architecture represents a system as an abstract netlist of virtual components (see Figure 3a). A virtual component consists of an internal component (or module) and its wrapper. The internal component contains a set of software tasks or represents a hardware function. The wrapper adapts accesses from the internal component to the external channels connected to the virtual component. The internal component and external channel(s) can be different in terms of: (1) communication protocol, (2) abstraction level, and (3) specification language. The wrapper is a set of virtual ports that contain internal and external ports. There could be an *n* to *m* (*n* and *m* are natural numbers) correspondence between internal and external ports.

Virtual channels hide many details of communication protocols, for instance, FIFO communication is realized using high-level communication primitives (e.g. *put* and *get*). In our design flow, the virtual architecture is specified using an extended SystemC library containing classes for virtual components with configuration parameters:

1. virtual module: consists of a module and its wrapper;
2. virtual port: groups some internal and external ports that have a conversion relationship. The wrapper is the set of virtual ports for a given virtual module;
3. virtual channel: groups several channels having a logical relationship (e.g. multiple channels belonging to the same communication protocol).

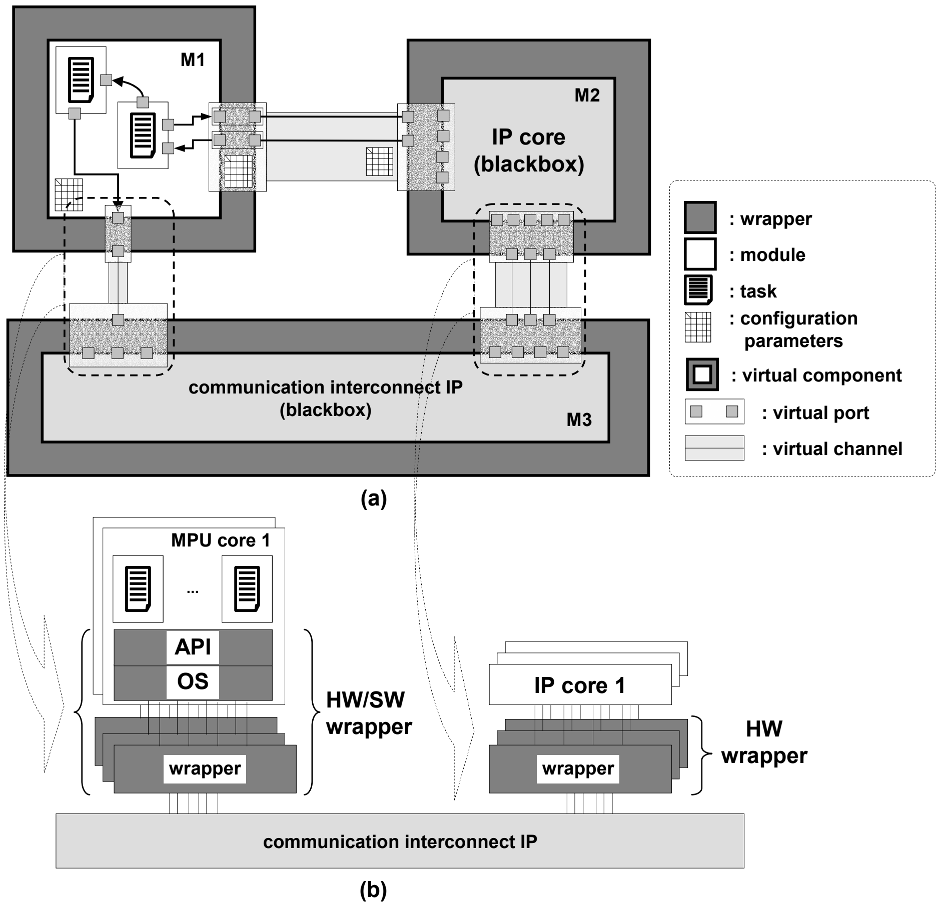

Figure 3. MPSoC design flow: virtual architecture (a), target MPSoC platform (b)

For system refinement, this model needs to be annotated with architecture configuration parameters (e.g. protocol and physical addresses of ports). Configuration parameters specify a unique way to map the virtual architecture into the final architecture, with hardware interfaces, operating systems and drivers customized to the application. They are set directly in the module, task, port, and channel as attributes:

1. For a module, there is an attribute for the type of the processor and a blackbox flag indicating an IP block.
2. For a task, the user can set the operating system services needed, its priority and the files that store the description of its behavior.
3. For a port, there is a set of attributes to configure: the type of the data transmitted, the set of addresses needed, interrupt allocation, and other parameters that depend on the communication protocol (e.g. the size of the data packet that will be transmitted each time).
4. For a channel, most configuration parameters are the same as for a port.

There are special ports for services that are not related to data transmission: *Service Access Ports* (SAP). They are implemented as hardware or software wrapper components that provide the requested services. There are two kinds of SAP ports: the *synchro* and the *timer*. The first one is used by a task to signal that it is sensitive to the clock of the system, and it implicitly means that functionality does not change when data exchanges are postponed to the next clock edge. The *timer* SAP can be used to request an interrupt from a hardware timer after a given delay.

This model is not directly synthesizable/executable because the behaviors of wrappers are not described. The main goal of our design methodology is to enable automatic generation of these wrappers, in order to produce a detailed architecture that can be both synthesized and simulated.

3.3 TARGET MPSoC ARCHITECTURE MODEL

We use a generic MPSoC architecture where processors are connected to a communication interconnect IP via wrappers (see Figure 3b). In fact, processors are separated from the physical communication IP by wrappers that act as communication coprocessors or bridges. Such a separation is necessary to free the processors from communication management and it enables parallel execution of computation tasks and communication protocols. Software tasks need also to be isolated from hardware through an OS that plays the role of SW wrapper. When defining this model our goal was to have a generic model where both computation and communication may be customized to fit the specific needs of the application. For computation, we may change the number and kind of components and, for communication, we can select a specific communication IP and protocols. This architecture model is suitable to a wide domain of applications; more details can be found in [12].

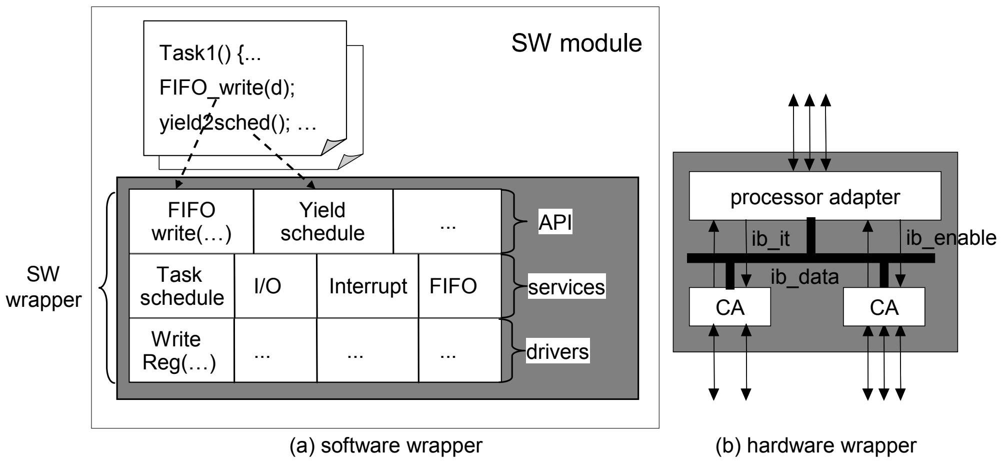

Figure 4. HW/SW wrapper architecture

As shown in Figure 4, the wrapper is made of a software part and a hardware part. On the hardware side, the internal architecture of the wrapper consists of a processor adapter, a channel adapter, and an internal bus. The number of channel adapters depends on the number of channels that are connected to the corresponding virtual module. On the software side, wrappers provide the implementation of high-level communication primitives (API) used in the software module and the drivers to control the hardware. If required, the software wrapper will also provide sophisticated OS services such as task scheduling and interrupt management.

3.4 DESIGN TOOLS

Figure 5 shows an overall view of our design environment, the input model may be imported automatically from a specification analysis tools (e.g. [6]) or manually coded using our extended SystemC library. All design tools use a unified design model (Colif [9]) which contains only a structural view of the platform and configuration parameters. Hardware wrapper generation [12] transforms the input model into a synthesizable architecture. The software wrapper generator [8] produces a custom OS for each processor on the target platform. For validation, we use the co-simulation wrapper generator [11] to produce simulation models for the input and output models. Details about these tools can be found in the references, only their principle will be discussed here.

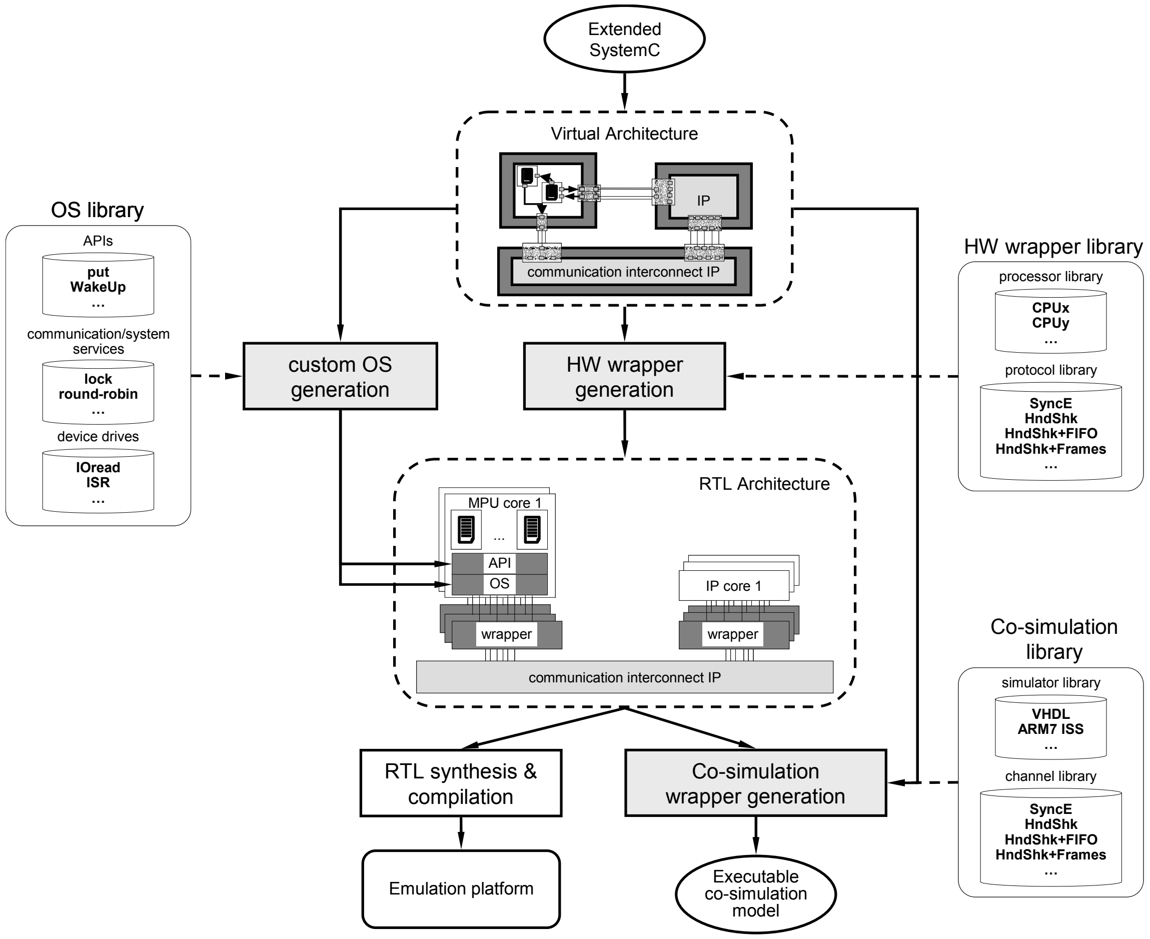

Figure 5. Design automation tools for MPSoC

Hardware wrapper generation assembles library components using the architecture template presented before (Figure 3b) to produce the output architecture. Architecture configuration parameters are used to instantiate library components; they are the result of decisions made during system architecture exploration. This library contains generalized descriptions of hardware components in a macro language (**m4** like), it has two parts: the processor library and the protocol library. The former contains local template architectures for processors with four types of elements: processor cores, local buses, local IP components (e.g. local memory, address decoder, coprocessors, etc.) and processor adapters. The latter consists of a list of channel adapters. Each channel adapter has simulation, estimation, and synthesis models that are parameterized (by the channel parameters, e.g. direction, storage size and data type) as the elements in the processor library. The estimation models enable the performance/cost estimation of communication protocol implementation in terms of HW area, power consumption, run-time, utilization, etc.

The software wrapper generator produces operating systems streamlined and pre-configured for the software module(s) that run(s) on each target processor. It uses an operating system library that is organized in three parts: APIs, communication/system services, and device drivers. Each part contains elements that will be used in a given software layer in the generated OS. The generated OS provides services specific to embedded systems: communication services (e.g. FIFO communication), I/O services (e.g. PCI bus drivers), memory services (e.g. cache or virtual memory usage), etc. Services can have dependency between them. For instance, communication services are dependent on I/O services. The OS library also has dependency information. This mechanism is used to keep the size of the generated OS at a minimum, by avoiding inclusion of unnecessary elements from the library.

There are two types of service code: re-usable (or existing) code and expandable code. As an example of existing code, AMBA bus-master service code can exist in the OS library in the form of C language. As an example of expandable code, OS kernel functions can exist in the OS library in the form of macro code (**m4** like). The main functionality of OS kernel is tasks scheduling, there are several preemptive schedulers available in the OS library such as round-robin scheduler, priority-based scheduler, etc. In the case of round-robin scheduler, time-slicing (i.e. assigning different CPU load to tasks) is supported. To make the OS kernel very small and flexible, (1) the task scheduler can be selected from the requirements of the application code and (2) a minimal amount (less than 10% of kernel code size) of processor specific assembly code is used (for context switching and interrupt service routines).

The co-simulation wrapper generator [11] produces an executable model that is used to validate the internal model. This executable model is composed of a SystemC simulator that acts as a master for other simulators. A variety of simulators can participate in this co-simulation: SystemC, VHDL, Verilog, and Instruction-set simulators. Co-simulation wrappers have the same structure that hardware wrappers (see Figure 4b), with simulation adapters in the place of processor adapters and simulation models of channel adapters. In the co-simulation wrapper library, there are simulation adapters for the different simulators supported. There are also channel adapters that implement all supported communication protocols in different languages.

In terms of functionality, the co-simulation wrapper transforms channel access(es) via internal port(s) to channel access(es) via external port(s) using the following functional chain: channel interface, channel resolution, data conversion, module communication behavior. Internal ports use channel functions (e.g. FIFO available, FIFO write) to exchange data. Channel interface provides the implementation of these channel functions. Channel resolution maps 1-to-N or N-to-1 correspondence between internal and external ports. Data conversion is required since different abstraction levels can use different data types to represent the same data. Module communication behavior is required to exchange data via external port(s), i.e. to call port functions of external ports.

4. COMPONENT-BASED DESIGN OF A VDSL APPLICATION

4.1 THE VDSL MODEM SPECIFICATION

The design we present in this section was taken from the implementation of a VDSL modem using discrete components [10]; the block diagram for this prototype implementation is shown in Figure 6a. The subset we will use in the rest of this paper is shaded in Figure 6. It is composed of two ARM7s and part of the datapath: the TX_Framer, described at the RT-level and used as a blackbox in this experiment. We adopted a partition of processors/tasks suggested by the design team of the VDSL-modem prototype (Figure 6b). CPU1 and CPU2 exchange data using three asynchronous FIFO buffers. The TX_Framer block can be configured through 17 configuration registers, and CPU2 can input a data stream to this block via a synchronous FIFO buffer. Tasks use a variety of control and data-transmission protocols to communicate. For instance, task **T1** can use control communication to change the execution status of tasks **T2** and **T3**, it can block/unblock the execution of these tasks by sending them an OS *signal*. For data transmission, tasks use: FIFO memory buffer, shared memory (with or without semaphores), and direct register access (pooling).

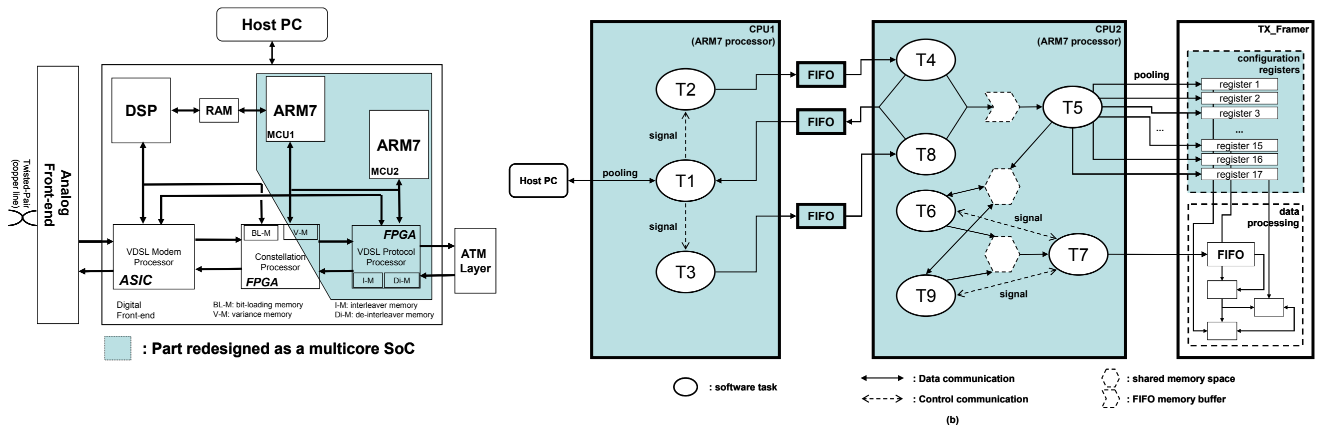

Figure 6. VDSL modem prototype (a) and specification (b)

Despite all simplifications, the design of the selected subset remains quite challenging. In fact, this application uses two processors executing parallel tasks. The control over the three modules of the specification is fully distributed. All three modules act as masters when interacting with their environment. Additionally, the application includes some multipoint communication channels requiring sophisticated OS services.

4.2 VDSL Virtual Architecture

Figure 7 shows the virtual architecture model that captures the VDSL specification using only point-to-point communications. This model can be used as a reference model for all designers because it has all the information necessary to produce an RTL implementation. Modules **VM1** and **VM2** correspond to the ARM7s on Figure 6a and module **VM3** represents the TX_Framer block (only the interface is known so it is represented as a blackbox). The virtual architecture can be mapped onto different architectures depending on the configuration parameters annotated in modules, ports, and channels. For instance, the three point-to-point connections (**VC1**, **VC2**, and **VC3**) between **VM1** and **VM2** can be mapped onto a bus or onto a shared memory if the designer changes the configuration parameters placed on these virtual channels and virtual ports. The callout shows some of the parameters for a multi-point guarded shared memory channel: the address and size of the memory buffer (32 positions in this case), OS virtual interrupt number, and data type (long int).

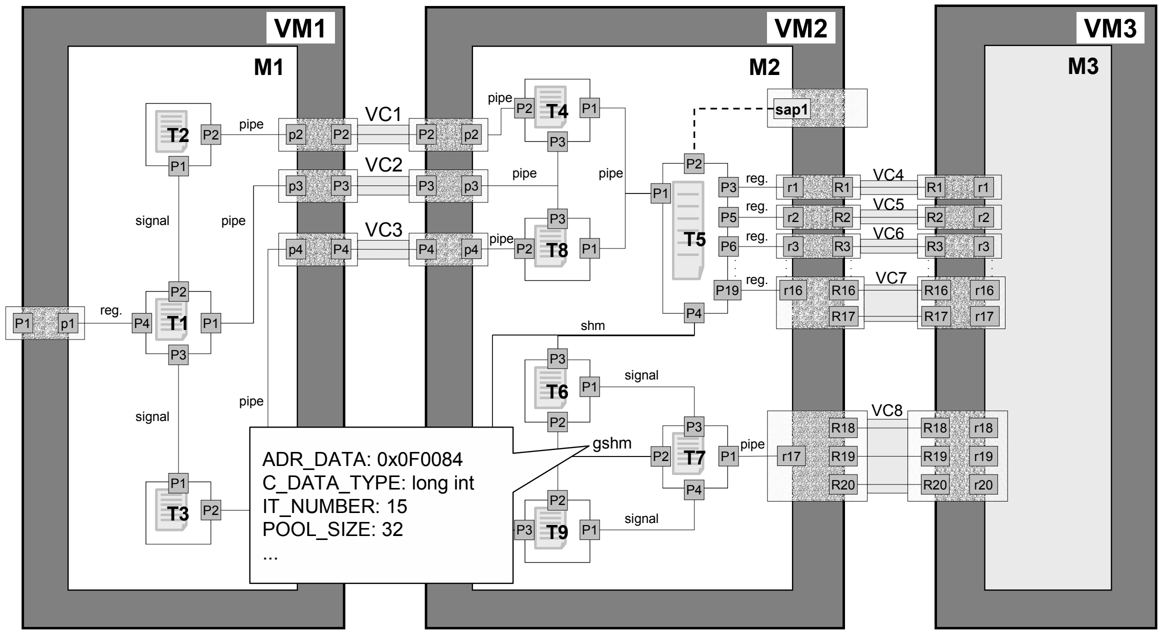

Figure 7. VDSL virtual architecture specification

4.3 Results

The manual design of a full VDSL modem requires several person-years; the presented subset was estimated as a more than five persons-year effort. When using this high-level component-based approach, the overall experiment took only one person during four months (not counting the effort to develop library elements and debug design tools). This corresponds to a 15-fold reduction in design effort. Running all wrapper generation tools takes only a few minutes on a Linux PC, most of the time was spent in writing the virtual architecture model with all the necessary configuration parameters. The behavior of each task was described using the SystemC C++ library (http://www.systemc.org/).

Figure 8 shows the RTL architecture model obtained after HW/SW wrapper generation: two ARM7 cores with their local architectures and the TX_Framer block. Each hardware wrapper acts as a communication co-processor for the ARM7, it contains an ARM7 processor adapter that bridges the ARM7 local bus to the communication adapters (CAs). There is a CA for each virtual channel in the virtual architecture specification. For instance, module **VM1** reads test vectors from the environment through a simple register using the *Pooling* CA, and communicates with **VM2** through asynchronous FIFOs using 3 *HNDSHK* CAs (corresponding to virtual channels **VC1**, **VC2**, and **VC3)**.

Each software wrapper (custom OS) is customized to the set of tasks executed by the processor core. For example, software tasks running on **VM1** access the custom OS using an API composed of two functions: *Pipe* for communication with **VM2**, and *Signal* to modify the task scheduling on runtime. The custom OS contains a round-robin scheduler (*Sched*) and resource management services (*Sync*, *IT*). The driver layer contains low-level code to access the CAs (e.g. *Pipe LReg* for the *HNDSHK* CA), and some low-level kernel routines.

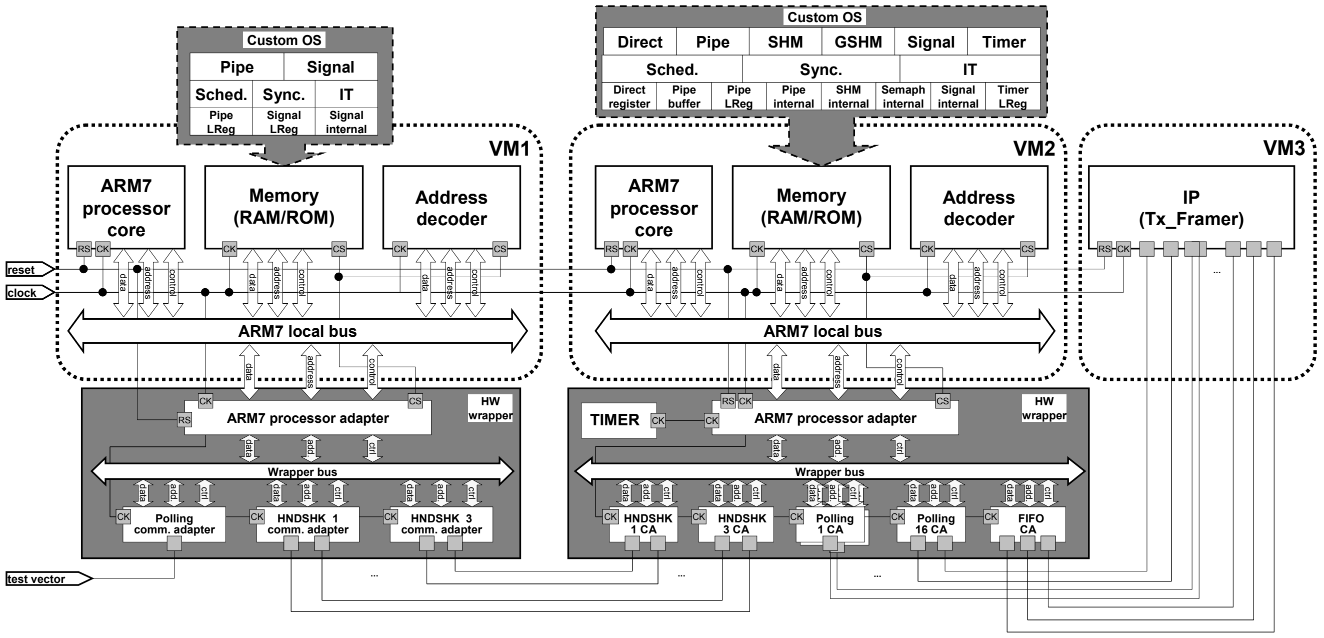

Figure 8. Generated MPSoC architecture

The hardware wrapper for processor **VM2** includes a timer module because task **T5** must wait 10ms before starting its execution. A hardware interrupt is generated by the TIMER block, the task can configure this block using the *Timer* API provided by the service access port **sap1** (see Figure 7). The custom OS for **VM2** provides a more complex API: the *Direct* API is used to write/read to/from the configuration/status registers inside the TX_Framer block; *SHM* and *GSHM* are used to manage shared-memory communication between tasks.

Table 1. Results for OS generation

OS results	# of lines C	# of lines Assembly	Code size (bytes)	Data size (bytes)
VM1	968	281	3829	500
VM2	1872	281	6684	1020
Context switch (cycles)				36
Latency for interrupt treatment (cycles)				59(OS) + 28(ARM7)
System call latency (cycles)				50
Resume of task execution (cycles)				26

Table 2. Results for the hardware wrapper generation

HW interfaces	# of Gates	Critical path delay (ns)	Max. freq. (MHz)
VM1	3284	5.95	168
VM2	3795	6.16	162
Latency for read operation (clock cycles)			6
Latency for write operation (clock cycles)			2
Number of code lines (RTL VHDL)			2168

Application code and generated OS are compiled and linked together to be executed on each ARM7 processor. The HW wrapper can be synthesized using RTL synthesis. Table 1 presents the results regarding the generated OSs. Part of the OS is written in Assembly, it includes some low-level routines (e.g., context switch and processor boot) that are specific to each processor. If we compare the numbers presented in Table 1 wih configurable commercial embedded OSs, the results are still very good. Generally, the minimum size for commercial OSs is around 4kbytes; but with this size, few of them could provide the required functionality. Performance was also very good: context-switch takes 36 cycles, latency for hardware interrupts is 59 cycles (plus 4 to 28 cycles needed by the ARM7 to react), latency for system calls is 50 cycles, and task reactivation takes 26 cycles.

Table 2 shows the numbers obtained after RTL synthesis of the HW wrappers using a CMOS 0.35µm technology. These results are good because wrappers account for less than 5% of the ARM7s core surface and have a critical path that corresponds to less than 15% of the clock cycle for the 25MHz ARM7 processors used in this case study.

4.4 EVALUATION

The results extracted from the RTL model show that our approach can generate hardware/software interfaces and operating systems that are as efficient as the manually coded/configured ones. Wrappers are build-up from library components so the HW/SW frontier in wrapper implementation can be easily displaced. This choice is transparent to the final user since everything that implements the interconnect API (hardware interfaces and OS) is generated automatically. Designers do not need to rewrite the application code because the API does not change (only its implementation does). Furthermore, correctness and coherence can be verified inside tools and libraries against the API semantics without having to impose fixed boundaries to the hardware/software frontier (in contrast to standardized interfaces and buses).

The generation of layered wrappers by assembling modular library components provides lots of flexibility for this design methodology. One of the most important consequences of having this flexibility is that the design environment can be easily adapted to accommodate: different languages to describe system behavior, different task scheduling and resource management policies, different global communication interconnect topologies and protocols, a diversity of processor cores and third-party blocks, and different memory architectures. Modular library components are at the roots of the methodology; the principle followed by these components is to contain a unique functionality, and to respect well-defined interfaces that enable easy composition. In most cases, inserting a new design element in this environment only requires to add the appropriate library components. Furthermore, the modular wrapper structure prevents library size explosion since components which behavior depend on the processor are separated from components that implement communication protocols.

5. REFERENCES

[1] W. Cesário, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, M. Diaz-Nava, A.A. Jerraya, "Component-Based Design Approach for Multicore SoCs," Proc. of 39th Design Automation Conference, New Orleans, June 2002.

[2] K. Keutzer, A.R. Newton, J.M. Rabaey, A. Sangiovanni-Vincentelli, "System-level design: orthogonalization of concerns and platform-based design," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol.: 19 Issue: 12, pp. 1523 –1543, Dec. 2000.

[3] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, A. Sangiovanni-Vincentelli,, "Addressing the System-on-Chip Interconnect woes through Communication-Based Design," Proc. of 38th Design Automation Conference, Las Vegas, June 2001.

[4] J-Y. Brunel, W.M. Kruijtzer, H.J.H.N. Kenter, F. Pétrot, L. Pasquier, E.A. de Kock, W.J.M. Smits, "COSY Communication IP's," Proc. of 37th Design Automation Conference, Los Angeles, CA, June 2000.

[5] D. Gajski, J. Zhu, R. Domer, A. Gerslauer, and S. Zhao, "SpecC Specification Language and Methodology," Kluwer Academic Publishers, 2000.

[6] Cadence Design Systems, Inc., Virtual Component Co-design: http://www.cadence.com/products/vcc.html

[7] D. Wingard, "MicroNetwork-Based Integration for SOCs," Proc. of 38th Design Automation Conference, Las Vegas, June 2001.

[8] L. Gauthier, S. Yoo, and A.A. Jerraya, "Automatic Generation and Targeting of Application Specific Operating Systems and Embedded Systems Software", IEEE TCAD, Vol. 20 Nr. 11, November 2001.

[9] W.O. Cesário, G. Nicolescu, L. Gauthier, D. Lyonnard, and A.A. Jerraya, "Colif: A design representation for application-specific multiprocessor SOCs," IEEE Design & Test of Computers, Vol.: 18, Issue: 5, pp. 8-20, Sept.-Oct. 2001.

[10] M. Diaz-Nava, G.S. Okvist, "The Zipper prototype: A Complete and Flexible VDSL Multi-carrier Solution", ST Journal special issue xDSL, September 2001.

[11] S. Yoo, G. Nicolescu, D. Lyonnard, A. Baghdadi, A. A. Jerraya, "A Generic Wrapper Architecture for Multi-Processor SoC Cosimulation and Design," Int. Symposium on HW/SW Codesign (CODES) 2001.

[12] D. Lyonnard, S. Yoo, A. Baghdadi, A. A. Jerraya, "Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip," Proc. of 38th Design Automation Conference, Las Vegas, June 2001.