Multiprocessor SoC Platforms: A Component-Based Design Approach

Wander O. Cesário, Damien Lyonnard, Gabriela Nicolescu, Yanick Paviot, Sungjoo Yoo, and Ahmed A. Jerraya
TIMA Laboratory

Lovic Gauthier
Institute of Systems & Information Technologies

Mario Diaz-Nava
STMicroelectronics

A high-level, component-based methodology and design environment for multiprocessor SoC architectures reduces design time without significant efficiency loss in the final circuit. This design environment provides tools for automatic wrapper generation that synthesize hardware interfaces, device drivers, and operating systems implementing high-level interconnect APIs.

Modern system-on-a-chip (SoC) design shows a clear trend toward 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 in a typical SoC will increase fourfold per technology node to match the corresponding applications' processing demands. Typical multiprocessor SoC (MPSoC) applications, such as network processors, multimedia hubs, and baseband telecommunications circuits, have particularly tight time-to-market and performance constraints that demand a very efficient design cycle.

This article, derived from a paper presented at the 39th Design Automation Conference,[1] describes a component-based design automation approach for MPSoC platforms.

MPSoC platform model

Our conceptual model of the MPSoC platform, shown in Figure 1a, includes four types of components: software tasks; processor and IP cores; and an IP core for a global, on-chip interconnect. Moreover, to complete the MPSoC platform, we include hardware and software elements that adapt platform components to one another. MPSoC platforms are quite different from single-master processor SoCs (SMSoCs). For example, their implementation of system communication is more complicated due to heterogeneous processors, and complex protocols and topologies. The hardware adaptation layer must deal with several issues:

- In SMSoC platforms, most peripherals (except direct-memory-access controllers) operate as slaves. MPSoC platforms can use many different types of processor cores, making sophisticated synchronization necessary to control shared communication among several heterogeneous masters.
- Whereas most SMSoC platforms use simple master-slave shared-bus interconnects, MPSoC platforms often use several complex system buses or micronetworks as global interconnects. The multimaster architecture

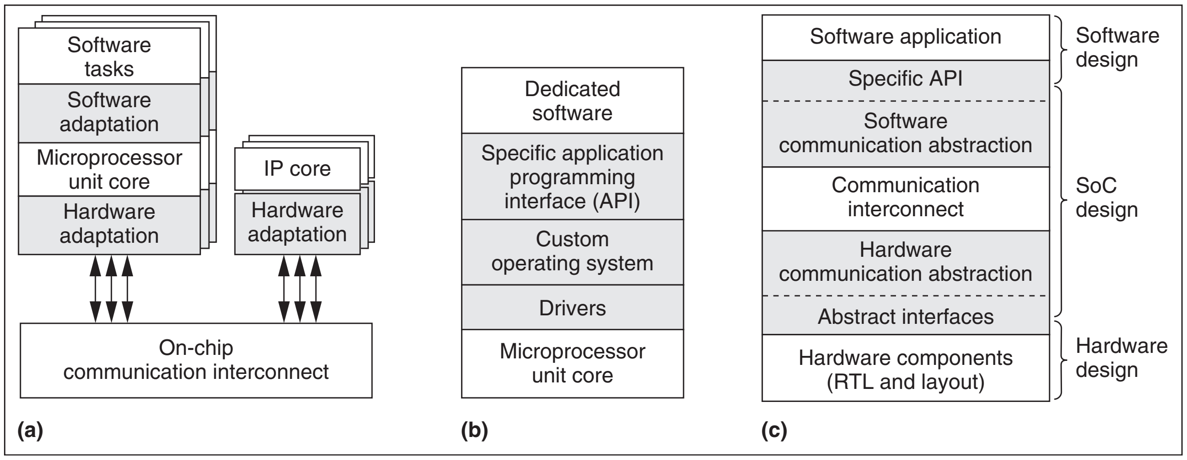

Figure 1. Multiprocessor system-on-a-chip (MPSoC) platform (a), software stack (b), and concurrent development environment (c).

used in MPSoC platforms allows a separation between computation and communication design. Communication coprocessors and controllers (masters) implement high-level communication protocols in hardware and execute them in parallel with the computation executed on processor cores.

Software developers typically organize application software as a stack of layers running on each processor core, as Figure 1b shows. The lowest layer contains drivers and low-level routines to control and configure the platform. The middle layer can use any commercial embedded operating system, configured according to the application. The upper layer is an application programming interface (API) that provides predefined routines for accessing the platform. All these layers correspond to the software adaptation layer in Figure 1a. Designers can then isolate coding of application software from the design of the SoC platform. One of the main contributions of our 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 application functions that are not performance critical. Because of code size and performance concerns, it is not realistic to use a generic operating system as the middle layer. A lightweight custom operating system, supporting an application- and platform-specific API, is necessary. Custom-operating-system design automation is a new research area motivated by MPSoC platform design.[2]

Software and hardware adaptation layers isolate platform components, thus enabling the concurrent development of the components in Figure 1c . With this scheme, the software developers use APIs for both application and dedicated software development. The hardware design team uses abstract interfaces from communication coprocessors and controllers. The SoC design team can concentrate on implementing hardware and software abstraction layers for the selected IP communication interconnect. Designing such hardware-software abstraction layers constitutes a major effort, and design tools are lacking. Established EDA tools are not well adapted to this new MPSoC design scenario; consequently, many challenging requirements are emerging:

- *Higher abstraction level.* It is too time-consuming to model at RTL and verify the interconnection between multiple processor cores.
- *Higher-level programming*. MPSoCs will include hundreds of thousands of dedicated software (firmware) lines. Developers cannot program this software at the assembly level, as they do today.
- *Efficient hardware-software interfaces.* Designers must optimize microprocessor interfaces, register banks, shared memories, software drivers, and operating systems for each application.

System-level design flow

Here we discuss current SoC design methodologies, using the template design flow in Figure 2. The basic principle behind this flow is the separation between communication and computation refinement for platform and component-based design.[3,4] The flow has five main design steps:

1. *System specification.* System designers and customers agree on an informal model for the application's functionality and requirements. On the basis of this model, system designers build a more formal specification that the customers can validate.
2. *Architecture exploration.* System designers build an executable model of the specification and iterate through a performance analysis loop to determine the hardware-software partitioning for the SoC architecture. This executable specification uses an abstract platform comprising abstract models for hardware and software components. For example, an abstract software model can concentrate on I/O execution profiles, most-frequent-use cases, or worst-case scheduling. Transaction-level or behavioral models can describe abstract hardware. This step produces the *golden* architecture model—either a customization of an existing SoC platform or a new architecture created by system designers after selecting processors, the global communication interconnect, and other IP components. After system designers define the hardware-software partitioning, software and hardware development can occur concurrently.
3. *Software design.* Because the final hardware platform will not be available during software development, the software design team must use a hardware abstraction layer or API.
4. *Hardware design.* Hardware IP designers work at RTL to implement the functionality described by the abstract hardware models. Hardware IP components can use specific interfaces for a given platform or standard interfaces—for instance, the virtual component interface (VCI) defined by the Virtual Socket Interface Alliance (VSIA, http://www.vsi.org).
5. *Hardware-software integration.* The golden architecture model specifies performance constraints to ensure good hardware-software integration. SoC designers create hardware and software interfaces to the global communication interconnect that conform to these constraints.

SoC design automation strategies

Many academic and industrial papers propose tools for SoC design automation that incorporate many of the preceding design steps. Most approaches fall into one of three groups: system-level synthesis, platform-based design, or component-based design.

System-level-synthesis methodologies are top-down approaches; synthesis algorithms produce the SoC architecture and software models from a system-level specification. Brunel et al. propose a process for refining hardware-software communication; it uses an extended Kahn process network model for design step 1,[5] virtual component codesign (VCC) for step 2,[6] callback signals over a standard real-time operating system for the API in step 3, and VSIA interfaces for steps 4 and 5.

SpecC, a specification language and design methodology, proposes a full set of synthesis tools:[7] It uses an untimed functional specification model written in extended C for design step 1, and performance estimation for a structural architecture model for step 2. It uses C code synthesis for step 3, and behavioral synthesis for step 4. SpecC performs step 5 using hardware-software interface synthesis based on a timed bus-functional communication model.

Platform-based design is a meet-in-the-middle approach; it starts with a functional system specification and a predesigned SoC platform. Performance estimation models can help analyze different mappings between the functional modules of the application and the platform components. During these iterations, designers can try different platform customizations and functional optimizations. VCC 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 to VCC for steps 4 and 5.

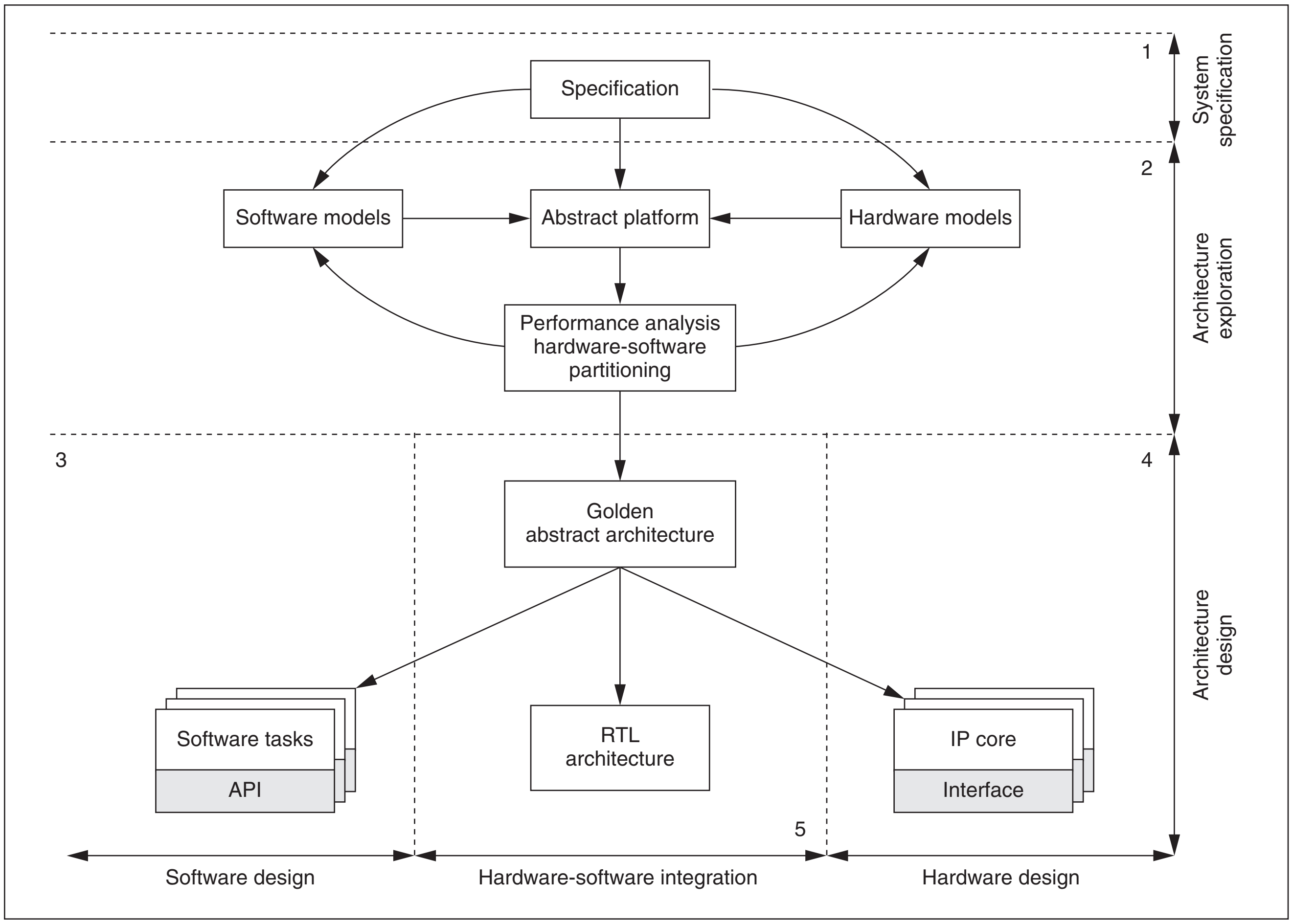

Figure 2. System-level design flow for SoCs.

Despite these automation methods and tools, designers must manually implement the API for software components (in step 3) as well as many architecture details (in step 5).

Most component-based design approaches build SoC architectures from the bottom up, using predesigned components with standard interfaces and/or a standard bus. For example, IBM defined a standard bus called CoreConnect (http://www.chips.ibm.com/bluelogic/), Sonics proposed a standard on-chip network called Silicon Backplane μNetwork,[8] and VSIA defined VCI, a standard component protocol. When needed, wrappers adapt incompatible buses or component interfaces. Typically, internally developed components are tied to in-house (proprietary) standards, so adopting public standards implies significant effort to redesign interfaces or wrappers for legacy components.

We have developed a higher-level component-based design environment for MPSoC platforms that starts with a virtual architecture model composed of hardware-software components. This environment automates design step 5 by automatically generating hardware interfaces (step 4); and device drivers, operating systems, and APIs (step 3). Although this approach does not provide much help toward automating design steps 1 and 2, it reduces design time considerably for steps 3, 4, and 5, and facilitates component reuse. The key improvements over other state-of-the-art platform and component-design approaches include the following:

- *Strong support for software design and integration.* The generated API completely abstracts the hardware platform and operating-system services. Software development can be concurrent with and independent of platform customization.

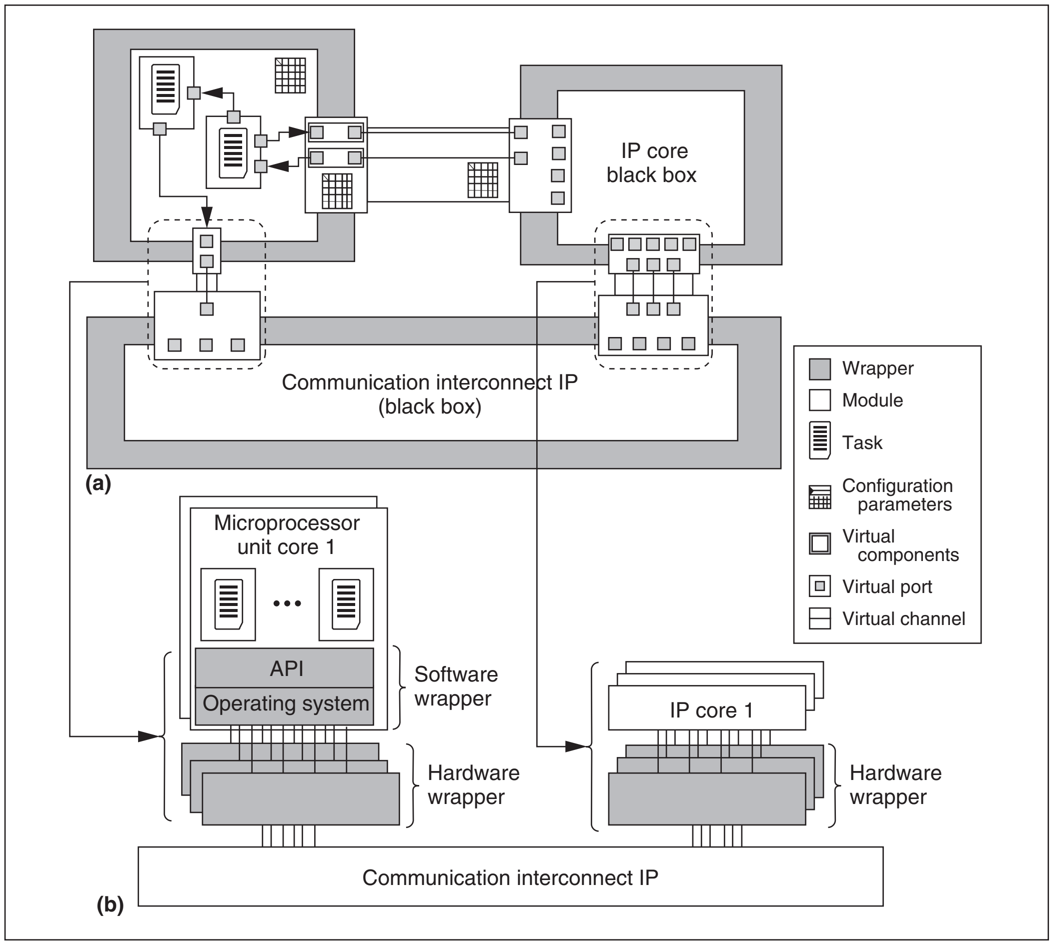

Figure 3. Design flow automatically generates the interfaces from the virtual architecture (a) for the target MPSoC platform (b).

- *Higher-level abstractions.* The virtual architecture model lets designers handle hardware-software interfaces at a high abstraction level. The system specification separates behavior and communication, permitting their independent refinement.
- *Flexible hardware-software communication.* Generating automatic hardware-software interfaces depends on the composition of library elements. User-extendable libraries permit the use of various IP interconnect components.

Component-based design for MPSoC

Our design flow starts with a virtual architecture model that corresponds to the golden architecture in Figure 2. This flow allows automatic generation of communication coprocessors and controllers (wrappers), device drivers, operating systems, and APIs. The goal is a synthesizable RTL model of the MPSoC platform that comprises processor cores, IP cores, the communication interconnect IP, and hardware-software wrappers. Hardware and software wrapper generation depends on the abstract interfaces of virtual components, as the arrows in Figure 3 indicate. Software written for the virtual-architecture specification runs without modification on the implementation because the custom operating system provides the same APIs.

Virtual architecture

The virtual architecture represents a system as an abstract netlist of virtual components (see Figure 3a). Virtual components use wrappers to adapt accesses from the internal component (a set of software tasks or a hardware function) to

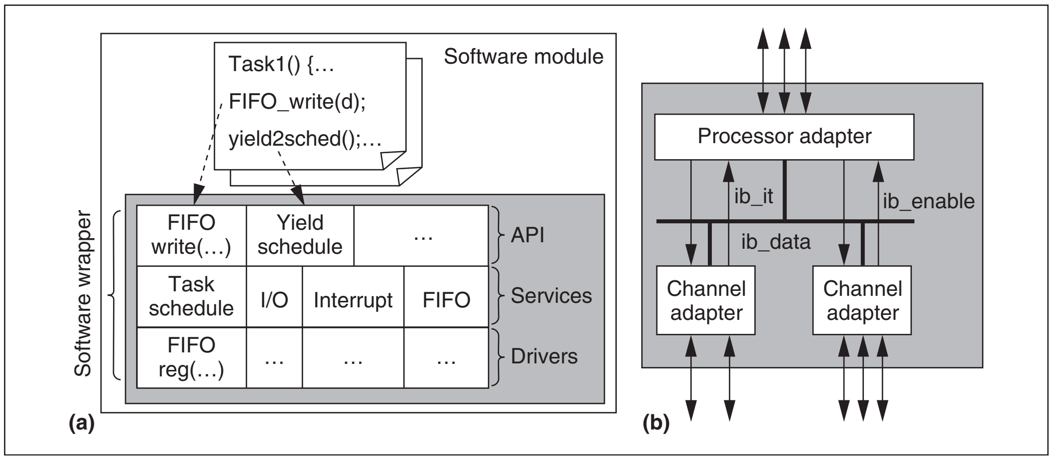

Figure 4. Software (a) and hardware (b) wrapper structure.

the external channels. The virtual architecture models the wrapper as a set of virtual ports containing internal and external ports. These ports can differ in terms of the communication protocol, the abstraction level, or the specification language. This model is not directly synthesizable or executable, because the virtual architecture does not describe a wrapper's behavior. The main goal of our methodology is to enable automatic generation of these wrappers and produce a detailed architecture that can be both synthesized and simulated.

The virtual architecture has service access ports (SAPs) for services implemented as hardware or software wrapper components. For example, you can use the timer SAP to request an interrupt from a hardware timer after a delay. Our design flow specifies the virtual architecture using the SystemC C++ library (http://www.systemc.org) extended with the following classes:

- A *virtual module* consists of a module and its wrapper (the set of virtual ports for a given virtual module).
- A *virtual port* groups together internal and external ports that have a conversion relationship.
- A *virtual channel* groups together several channels having a logical relationship (for example, multiple channels belonging to the same communication protocol).
- *Parameters* customize hardware interfaces (for example, to define buffer sizes and physical addresses of ports), operating systems, and drivers.

Target MPSoC architecture model

Our methodology harnesses a generic MPSoC architecture, using wrappers to connect processors and other IP cores to an IP component for the global communication interconnect, as Figure 3b shows. In fact, wrappers, acting as communication coprocessors or bridges, separate processors from the physical communication IP. This separation frees the processors from communication management and enables parallel execution of computation tasks and communication protocols. An operating system, acting as a software wrapper, isolates software tasks from hardware. Our goal was to define a generic model that would let designers customize both computation and communication to fit an application's specific needs. For computations, designers can change the number and type of components; for communication, they can select specific communication IP and protocols. This architecture model is thus suitable for a wide domain of applications.[9]

A wrapper's implementation contains both hardware and software parts. On the software side, wrappers provide the implementation of high-level communication primitives (APIs) in the software module, as well as drivers to control the hardware. If required, the software wrapper can also provide sophisticated operating-system services such as task scheduling and interrupt management (see Figure 4a). The hardware part includes a processor/IP adapter, channel adapters, and an internal bus, as Figure 4b shows. The number of channel adapters depends on the

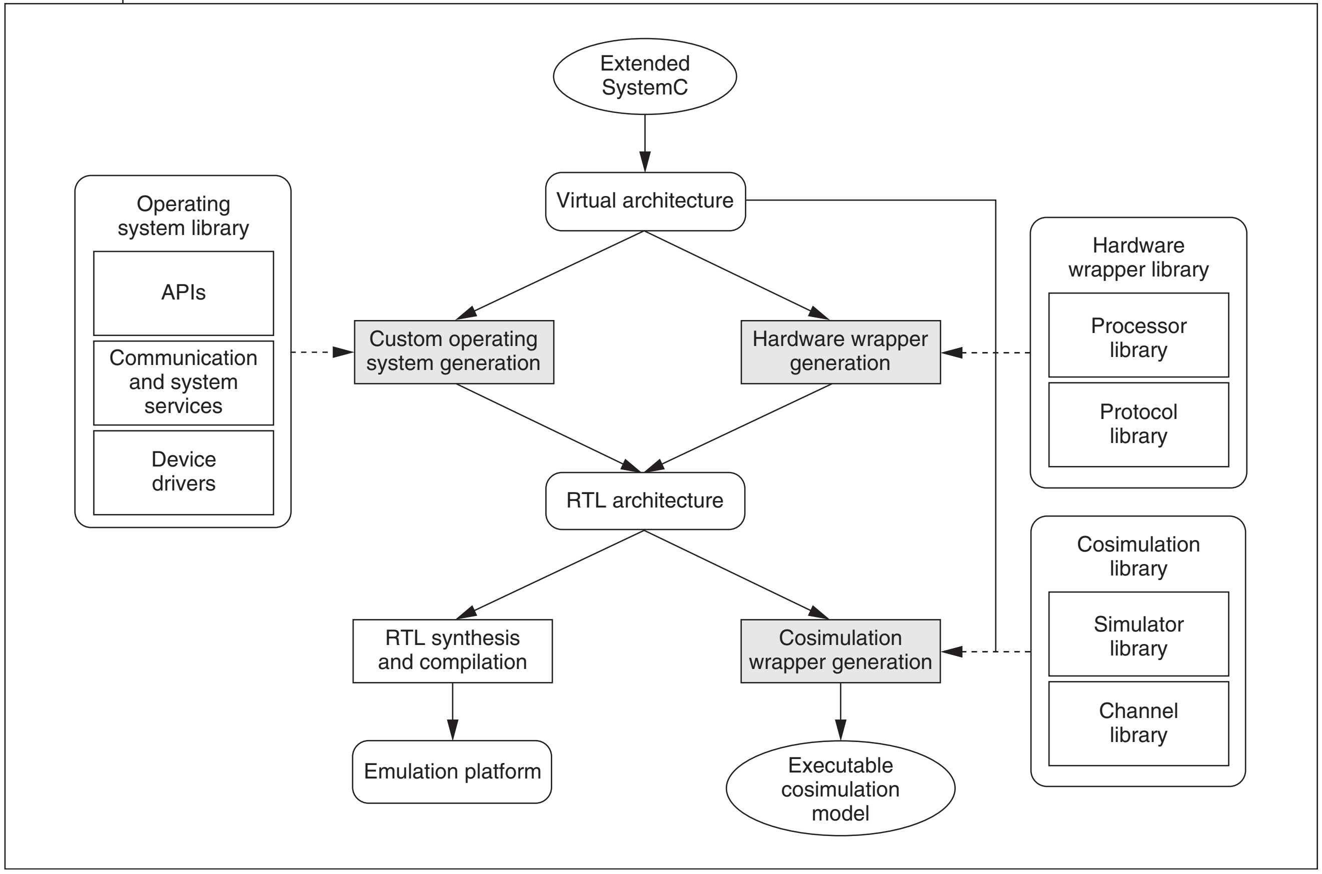

Figure 5. Design automation tools for MPSoCs.

number of channels connected to the corresponding virtual module.

Design tools

Figure 5 gives an overall view of our design environment. Designers can either import the input model from specification analysis tools or manually code it using our extended SystemC library. All design tools use a unified design model containing an abstract hardware-software netlist annotated with parameters.[10] Hardware wrapper generation transforms the input model into a synthesizable architecture.[9] The software wrapper generator produces a custom operating system for each processor on the target platform.[2] For validation, the cosimulation wrapper generator produces simulation models.[11]

Hardware wrapper generator. This design tool assembles library components using the architecture template in Figure 3b to produce the RTL architecture. This library contains generalized descriptions of hardware components in a macrolanguage (like GNU's m4). The library has two parts: The *processor library* contains local template architectures for processors with four types of elements: processor cores, local buses, local IP components (such as local memory, address decoders, and coprocessors), and processor adapters. The *protocol library* includes a list of channel adapters, each having simulation, estimation, and synthesis models. The models in the processor and protocol library are specialized during instantiation according to the parameters annotated in the virtual architecture model (for example, with channel parameters such as direction, storage size, and data type).

Software wrapper generator. This tool produces optimized and preconfigured operating systems for the software tasks that run on each target processor. It uses a library organized into APIs, communication/system services, and device drivers. Each of these three parts has elements that the generated operating system will use in a given software layer. The generated operating system provides services, including those for

- communication—for instance, FIFO communication;
- I/O—such as Advanced Microcontroller Bus Architecture (AMBA) bus drivers; and
- memory—for example, shared memory spaces.

The software wrapper generator uses a dependency graph where services and library elements are represented as nodes. Arcs on this graph connect services that are interdependent; for example, communication services depend on I/O services. Services also depend on elements of the operating-system library used to implement the service. Resolving this graph for the minimal set of necessary library elements minimizes the size of the generated operating system. The generated operating system does not include elements that provide unnecessary services.

There are two types of service codes: reusable (or existing) and expandable. As an example, the operating-system library can contain reusable C code for an AMBA bus-master service. In the case of expandable code, the library might contain operating-system kernel functions in the form of (m4-like) macrocode. Several preemptive schedulers—for example, round robin or priority based—are available in the operating-system library. Round-robin schedulers support time slicing, the assignment of different CPU loads to tasks. To make the operating-system kernel small and flexible, you can select the task scheduler from the application code's requirements.

Cosimulation wrapper generator. This tool produces an executable model containing a SystemC simulator that acts as a master for other simulators. Various simulators—SystemC, VHDL, Verilog, and Instruction Set Simulator (ISS)—can participate in cosimulation. Cosimulation wrappers have the same structure as hardware wrappers (see Figure 4b) but substitute simulation adapters for processor adapters, and simulation models for channel adapters.

The cosimulation wrapper library has simulation adapters to support various simulators, and channel adapters with simulation models for all supported protocols.

In terms of functionality, the cosimulation wrapper transforms channel accesses via internal ports to channel accesses via external ports, using the following functional chain:

1. channel interface,
2. channel resolution,
3. data conversion, and
4. module communication behavior.

Internal ports use channel functions (for instance, FIFO_available or FIFO_write) to exchange data. The channel interface implements these channel functions. Channel resolution establishes the correspondence between *N* internal ports and *M* external ports. Data conversion is necessary because different abstraction levels can use different data types to represent the same data. Module communication behavior allows data exchange through external ports—that is, via port functions.

Component-based design of a VDSL application

As a case study, we redesigned part of a very-high-bit-rate digital subscriber line (VDSL) modem.[12] We replaced a microcontroller with two ARM7s and used part of the VDSL protocol processor, TX_Framer, as an IP core. The virtual architecture specification has all the information necessary to produce an RTL implementation. The model for this case study uses only point-to-point communication, as Figure 6 (next page) shows. Virtual modules 1 and 2 represent the ARM7 processors, and virtual module 3 represents the TX_Framer IP (only the interface is known, so Figure 6 represents virtual module 3 as an empty box). The callout shows some of the parameters for a multipoint, guarded, shared-memory channel:

- the memory buffer's address and size (32 positions, in this case),

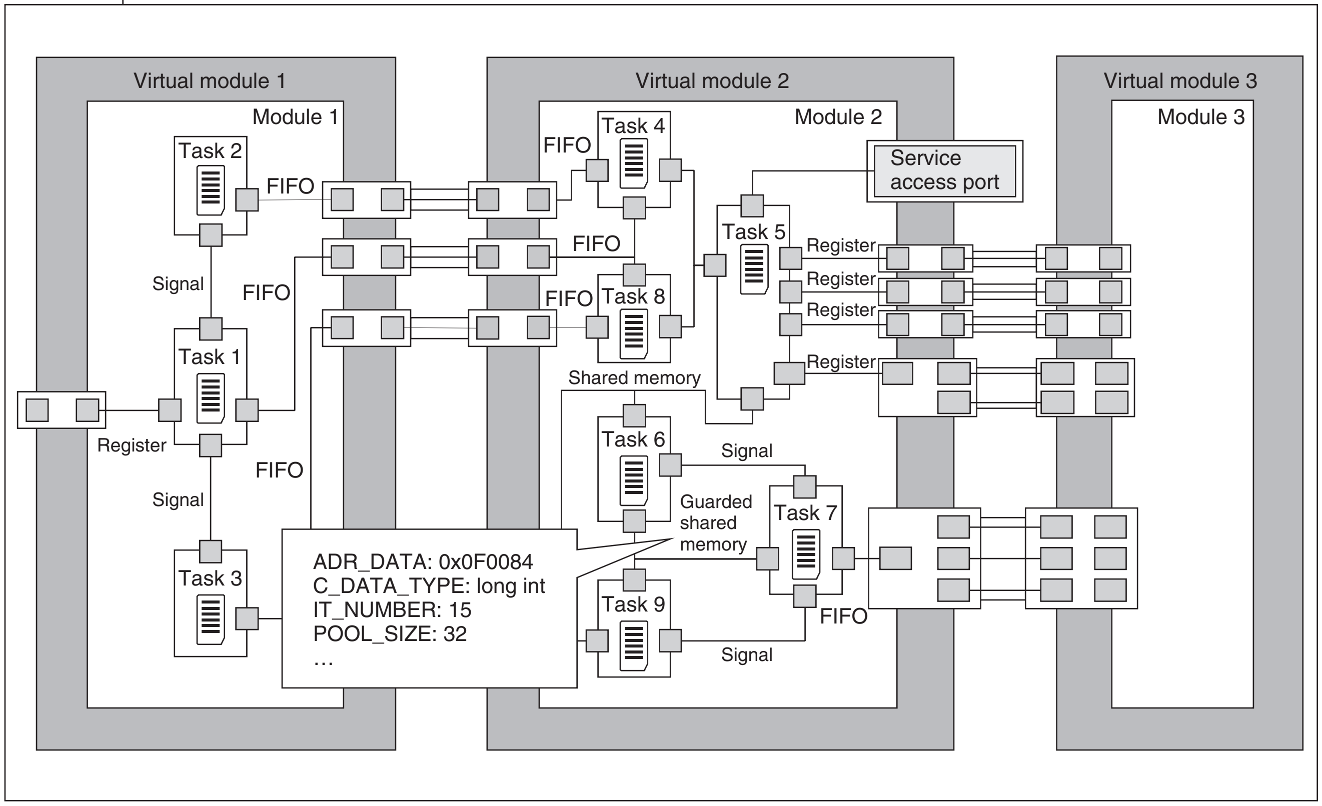

Figure 6. VDSL virtual-architecture specification.

- the operating system's virtual interrupt number, and
- the data type (long integer).

Following a suggestion by the VDSL modem's design team, we partitioned processors and tasks. Processors exchange data using three asynchronous FIFO buffers. Tasks use various control and data transmission protocols to communicate. For example, a task can block or unblock the execution of other tasks by sending them an operating-system signal.

For data transmission, tasks use a FIFO memory buffer, two shared memories (with or without semaphores), and direct register access. Despite simplifications, this design remains quite complex: It uses two processors executing parallel tasks and fully distributed control. The three modules can act as masters when interacting with their environment. Additionally, some multipoint communication channels require sophisticated operating-system services.

Results

It took about four person-months to produce a synthesizable RTL architecture using our design environment (not counting the effort to develop library elements and debug design tools). The effort to manually code, integrate, and debug custom operating systems, APIs, and communication coprocessors for this design was estimated to be about five person-years. For this case study, we estimate a 15-fold reduction in design effort. Running all wrapper generation tools takes only a few minutes on a Linux PC; writing and debugging the virtual architecture model consumes most of the time.[1]

Software developers can compile and link application code and the generated operating system for execution on an ARM7 processor or ISS. Hardware designers can synthesize the hardware wrapper using RTL synthesis. A small part of the generated operating systems is in assembly; this portion includes some low-level routines, such as for context switch and processor boot. The results in Table 1 compare well with those of commercial embedded operating systems. The minimum size for such operating systems is around 4 Kbytes, but this size precludes many of these systems from providing the required functionality. Table 2 shows results after RTL synthesis of the hardware wrappers for implementation in a 0.35-micron CMOS technology. These results are encouraging; they show that wrappers account for less than 5% of the ARM7s' core surface, and they have a critical path that corresponds to less than 15% of the clock cycle for the 25-MHz ARM7 processors used in this case study.

Table 1. Results for operating-system generation.

Implementation characteristic	Result
Virtual module 1	
No. of lines in C	968
No. of lines in assembly	281
Code size (bytes)	3,829
Data size (bytes)	500
Virtual module 2	
No. of lines in C	1,872
No. of lines in assembly	281
Code size (bytes)	6,684
Data size (bytes)	1,020
No. of cycles for context switch	36
Interrupt latency (cycles)	59 (operating system) + 28 (ARM7)
System call latency (cycles)	50
Resumption of task execution (cycles)	26

Table 2. Results for hardware wrapper generation.

Implementation characteristic	Result
Virtual module 1	
Critical path delay (ns)	5.95
Maximum frequency (MHz)	168
No. of gates	3,284
Virtual module 2	
Critical path delay (ns)	6.16
Maximum frequency (MHz)	162
No. of gates	3,795
Read operation latency (cycles)	6
Write operation latency (cycles)	2
RTL VHDL code	
No. of lines	2,168

Evaluation

Results show that our approach can generate hardware-software interfaces and operating systems as efficient as manually coded/configured ones. We can easily displace the hardware-software frontier in wrapper implementation by changing some library components. This choice is transparent to the user because our environment automatically generates everything that implements the interconnect API: The API doesn't change; only its implementation does. Furthermore, we can verify correctness and coherence inside tools and libraries against the API semantics. We can do so without imposing fixed boundaries to the hardware-software frontier (in contrast to using standardized component interfaces or buses).

Use of layered library components provides considerable flexibility. The design environment can easily adapt to accommodate different

- languages to describe system behavior,
- task-scheduling and resource management policies,
- global-communication interconnect topologies and protocols,
- processor cores and IP cores, and
- memory architectures.

In most cases, inserting a new design element in this environment requires only adding the appropriate library components. Layered library components are at the root of our methodology: The principle is to contain a unique functionality and respect well-defined interfaces that enable easy composition. This layered structure prevents library size explosion, using the composition of library components to implement complex functionality and increase component reuse.

The high-level, component-based design methodology lets MPSoC designers handle hardware-software interfaces at a high abstraction level. Our design environment, called Roses, integrates tools for hardware, software, and cosimulation wrapper generation. High-level modeling and automatic generation of efficient hardware-software interfaces are our main contribution to increase the efficacy of MPSoC design cycles. Nevertheless, the MPSoC design scenario discussed in the introduction raises many challenges and motivates many new research areas. We are exploring debugging strategies with Roses for hardware-software interfaces—for example, automatic generation of transactional test benches. We are also investigating use of operating-system generation to support execution of transactional test benches in the implementation. Finally, we are exploring MPSoC emulation strategies using an ARM multiprocessor platform. ■

■ References

1. W. Cesário et al., "Component-Based Design Approach for Multicore SoCs," *Proc. 39th Design Automation Conf.* (DAC 02), ACM Press, New York, 2002, pp. 789-794.
2. L. Gauthier, S. Yoo, and A.A. Jerraya, "Automatic Generation and Targeting of Application-Specific Operating Systems and Embedded Systems Software," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 11, Nov. 2001, pp. 1293-1301.
3. K. Keutzer et al., "System-Level Design: Orthogonalization of Concerns and Platform-Based Design," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 19, no. 12, Dec. 2000, pp. 1523-1543.
4. M. Sgroi et al., "Addressing the System-on-Chip Interconnect Woes through Communication-Based Design," *Proc. 38th Design Automation Conf.* (DAC 01), ACM Press, New York, 2001, pp. 667-672.
5. J-Y. Brunel et al., "COSY Communication IP's," *Proc. 37th Design Automation Conf.* (DAC 00), ACM Press, New York, 2000, pp. 406-409.
6. "Virtual Component Co-design (VCC)," Cadence Design Systems, San Jose, Calif.; http://www.cadence.com/products/vcc.html.
7. D. Gajski et al., *SpecC Specification Language and Methodology*, Kluwer Academic, Norwell, Mass., 2000.
8. D. Wingard, "Micronetwork-Based Integration for SOCs," *Proc. 38th Design Automation Conf.* (DAC 01), ACM Press, New York, 2001, pp. 673-677.
9. D. Lyonnard et al., "Automatic Generation of Application-Specific Architectures for Heterogeneous Multiprocessor System-on-Chip," *Proc. 38th Design Automation Conf.* (DAC 01), ACM Press, New York, pp. 518-523.
10. W.O. Cesário et al., "Colif: A Design Representation for Application-Specific Multiprocessor SOCs," *IEEE Design & Test of Computers*, vol. 18, no. 5, Sept.-Oct. 2001, pp. 8-20.
11. S. Yoo et al., "A Generic Wrapper Architecture for Multi-Processor SoC Cosimulation and Design," *Proc. 9th Int'l Symp. Hardware/Software Codesign* (CODES 01), ACM Press, New York, 2001, pp. 195-200.
12. M. Diaz-Nava and G.S. Okvist, "The Zipper Prototype: A Complete and Flexible VDSL Multi-carrier Solution," *ST J. System Research*, vol. 2, no. 1, Sept. 2001, pp. 1/3-21/3.

Wander O. Cesário is a research engineer at TIMA (Techniques of Informatics and Microelectronics for Computer Architecture) Laboratory. His research interests include system-level design modeling, analysis, simulation, and synthesis for SoC embedded systems and related CAD algorithms. Cesário has a PhD in microelectronics from Institut National Polytechnique de Grenoble, France.

Damien Lyonnard is a PhD student in the System-Level Synthesis Group at TIMA Laboratory. His research interests include automation in application-specific architecture refinements and the design of heterogeneous multiprocessor SoCs. Lyonnard has an MSc in microelectronics from Joseph Fourrier University, Grenoble, France.

Gabriela Nicolescu is a PhD student in the System-Level Synthesis Group at TIMA Laboratory. Her research interests include communication synthesis and validation of multiprocessor SoCs. Nicolescu has a master's degree in systems reliability from Polytechnic University, Bucharest, Romania.

Yanick Paviot is a PhD student at TIMA Laboratory. His research interests include hardware-software partitioning of interprocessor communications and its influence on the design of heterogeneous multiprocessor SoCs. Paviot has an MSc in microelectronics from Joseph Fourrier University.

Sungjoo Yoo is a researcher at TIMA Laboratory. His research interests include application-specific operating-system modeling and generation, networks on chips, mixed-level SoC simulation, low-power design, and reconfigurable SoC design. Yoo has a PhD in electronic engineering from Seoul National University.

Ahmed A. Jerraya leads the System-Level Synthesis Group at TIMA Laboratory and is research director of the National Center for Scientific Research. His research interests include SoCs and system-level specification languages. Jerraya has a Dr Ing in computer sciences from the University of Grenoble, France.

Lovic Gauthier is a researcher at the Institute of Systems & Information Technologies in Fukuoka, Japan. His research interests include operating systems for embedded systems. Gauthier has a PhD in microelectronics from Institut National Polytechnique.

Mario Diaz-Nava is a SoC emulation program manager in the Advanced System Technology Division at STMicroelectronics. His research interests include digital communications circuits; multiprocessor systems; and system-level design methodologies, including architecture exploration. Diaz-Nava has a PhD in ICs for digital communication networks from Institut National Polytechnique.

■ Direct questions and comments about this article to Wander O. Cesário, TIMA Laboratory, SLS Group, 46 av. Félix Viallet, 38031 Grenoble Cedex, France; wander.cesario@imag.fr.

For further information on this or any other computing topic, visit our Digital Library at http://computer.org/publications/dlib.