Des Autom Embed Syst (2006) 10:253–283
DOI 10.1007/s10617-006-0654-9

# Platform designer: An approach for modeling multiprocessor platforms based on SystemC

**Cristiano Araujo · Millena Gomes · Edna Barros · Sandro Rigo · Rodolfo Azevedo · Guido Araujo**



**Abstract** This paper[3.5pc] presents the Platform Designer (PD) framework, a set of SystemC based tools that provide support for modeling, simulation and analysis of multiprocessor SoC platforms (MPSoC), at different abstraction levels. PD provides mechanisms for interconnection specification, process synchronization and communication, thus allowing the modeling of a complete platform, in a unified environment. To do that it uses an extension of the ArchC ADL and acsys, a tool that enables the automatic generation of a SystemC simulator of the platform. The main advantages of this approach are twofold. First, designers have more flexibility since they can integrate and configure different processors to the platform, using a single environment. Second, it enables a faster design space exploration, given that it automatically generates SystemC simulators of whole platforms at distinct abstraction levels. A number of platform variations can be tried out with minor design changes, thus reducing design time. Experimental results show the suitability of the platform simulator for design space exploration. Real applications (with medium complexity) run in the platform in few minutes. Combined with the facility to generate platforms with minor changes, this feature allows an improvement of the design space exploration.

**Keywords** Platforms · SystemC platform descriptions · Architecture description languages · Platform simulation · Communication and synchronization architecture

## 1 Introduction

Typical embedded systems are traditionally designed to run a single embedded application under stringent resource constraints. Shared memory multiprocessor SoCs (MPSoCs)

C. Araujo (✉) · M. Gomes · E. Barros
Informatics Center (CIn), Federal University of Pernambuco,
Cidade Universitaria 50740-540, Recife PE, Brazil

S. Rigo · R. Azevedo · G. Araujo
Institute of Computing, University of Campinas,
Cidade Universitaria Zeferino Vaz, PO. Box 6176, Campinas-SP, Brazil

have been widely used in today high-performance embedded systems, such as network and multimedia processors. They combine the high-performance of multiprocessor parallelism with the level of integration provided by systems-on-chip technology [31].

MPSoC performance is not only determined by the capacity of the node processors (e.g. CPU speed, cache size, etc.), but the network that interconnects processors and memories also limit it. The design and optimization of such networks are critical for MPSoC performance. On the other hand, tools for modeling and analyzing such systems, at an early design phase, are mandatory in order to reduce the design time and time-to-market.

To support the design space exploration, at an early design phase, the application should be mapped onto a multiprocessor platform model, which should support an analysis of the functional and non-functional requirements of the application.

Simulation is the most used technique for the analysis of functional and non-functional requirements of multiprocessor platforms (MPSoCs) [12, 24, 25, 30]. It is used to verify the correct behavior of the embedded application, which is distributed on the processors, and the impact of the interconnection structure on the communication cost. In the later, simulation should provide mechanisms to detect the occurrence of performance penalties and communication *hazards*, like task starvation.

The simulation of multiprocessor platforms is not a trivial task. Instruction Set Simulators (ISSs) for the processors must be integrated in order to simulate the programs running on each processor. It is also necessary to synchronize these simulators in accordance with the communication protocols implemented by the underlying interconnection structure of the system. Validation of these components is critical as they define most of the functional and non-functional requirements of the system. Traditional approaches use encapsulation of third party ISSs as components of the system [8, 10, 24]. One problem that arises from component-based approaches is that the design space is normally constrained. In most cases, a small set of processors is available, while in others, just a family of processors. This is critical when the processors at disposal do not meet the design requirements.

This paper presents a processor centric approach for modeling and simulation of MPSoCs. It leverages on the *ArchC* architecture description language, which has been extended in order to provide a suitable mechanism for modeling multiprocessor platforms at a high abstraction level. Platforms are modeled as an extension of processor constructs, creating a smooth hierarchical view of the whole design. Together with the language extension, a tool has been developed that takes the platform description and generates SystemC executable simulation models of the platform, at different abstraction levels: functional and cycle accurate. Besides platform modeling, the proposed methodology also includes a mechanism for implementing process synchronization and communication in the target platform.

The benefits from this approach are the increase in productivity and a more efficient design space exploration. The first benefit comes from the automatic generation of simulation models, while the second one is due to the availability of a common environment for specifying processors, devices, and interconnection mechanisms using the same design language.

The rest of the paper is structured as follows. Section 2 describes related works. Multiprocessor platform modeling and simulation are discussed in Section 3. The ArchC language is shortly introduced in Section 4. The proposed approach for platform modeling is described in Section 5, whereas the technique for implementing process synchronization and communication, in the target platform, is detailed in Section 6. An Eclypse based framework that integrates all tools is shortly described in Section 7. A case study is presented in Section 8. Finally, some conclusions and future developments are discussed in Section 9.

## 2 Related work

The CoWare [10] approach uses the *Platform Architect* and *Processor Designer* tools to model multiprocessor platforms. The Processor Designer tool allows the specification of processors using LISA-2.0 [16] while the Platform Architect tool uses SystemC to model and simulate the platform.

STepNP [24, 25] is a platform exploration tool for the design of Network Processor Units (NPUs). In STepNP designers are restricted to a pre-defined set of processors. They can choose from ARM, PowerPC and DLX simulators. STepNP developers are also planning to include Xtensa [19] and LISATek [16] processor models.

In the *Component Based Design* methodology [8, 9, 12, 23] processor simulators are wrapped in components. An *ad hoc* wrapper generation scheme is built, and the configuration parameters for the processors instantiation are defined. In a second phase, the designers set the configuration parameters values in order to construct the platform simulation model.

EXPRESSION [15] is an architecture description language that is suited for the description of different types of architectures: VLIW, ASIP, DSP and conventional processor architectures, like RISC. Functional, cycle accurate and compiled simulators can be generated from an EXPRESSION description. Despite of being a powerful ADL, EXPRESSION based simulators are basically used for the processor toolkit generation and not for describing multiprocessor platforms.

The SpecC methodology focuses on the system specification without making any assumption about the target architecture. Its objective is to allow the gradual refinement of a system level description down to a synthesizable description. A detailed tutorial is given in [13].

The SpecC environment supports architecture allocation and communication synthesis. The automatic interface synthesis is described in [2, 26], and focuses on the automatic refinement of system level communication, which is based on channels through the bus functional model of the target architecture.

Sesame is a modeling and simulation environment for system-level design, based on the Y-chart design approach [29], where application and system architecture are described independently. Its goal is to enable a fast performance evaluation of the system in the early steps of the design cycle. Two features distinguish the simulation in the Sesame environment. First, application and architecture are simulated in separate simulators. During the execution of the application it generates operation and communication traces that are used by the architecture. Actually, the application is not executed in the target architecture. Traces annotated with the actions performed by the application processes work as stimuli to the architecture model.

Sesame appears to support partial simulation because it does not simulate the application running on the target architecture, it uses trace driven simulation or co-simulation between the application and the architecture.

SPADE was the base of Sesame, and basically has the same features. The main difference comes from the fact that it uses a C++ library called YAPI [11] to describe the Kahn process network that implements the application functionality. SPADE uses *Kahn process Network* (KPN) [20] for application modeling with unbounded fifo channels for communication. The work performed in SPADE [22] also uses communication traces generated by the application as an input to the architecture.

COSY [7] uses a transaction level approach to ease the task of building and exchanging IP cores. The main idea behind COSY is to have an application level description of the IP core, and possibly several models for hardware and software. Application level descriptions are based on the Kahn Process Networks that communicate through unbounded fifo channels.

**Table 1** system level design tools comparison

| | CoWare | StepNP | CBD | SpecC | SPADE | COSY | Sesame | Metropolis |
|---|---|---|---|---|---|---|---|---|
| simulation | x | x | x | x | p | p | p | x |
| high level analysis | – | x | – | x | x | x | x | x |
| generic app. domain | x | – | x | x | – | – | – | x |
| proc. integration | x | p | p | p | – | – | – | – |
| proc. modification | x | – | – | – | – | – | – | – |
| comm. mapping | p | x | p | x | x | x | x | x |
| system level spec. | SystemC | Click | heterog. | SpecC | – | – | YML | heterog. |
| architecture spec. | SystemC | SystemC | SystemC | SpecC | YAPI | ? | YML | SystemC |

A generic implementation model of the communication scheme has been developed, and concrete implementations of hardware and software combinations have been proposed [7].

Metropolis [6] is a framework that provides support for the modeling, simulation and analysis of digital systems at different abstraction levels. It defines a *metamodel* that describes the system behavior formally, allowing refinement of the system in a secure manner. The Metropolis meta-model can also be used to model the architecture of the system at a high abstraction level, this includes the processors and interconnection structures of the architecture.

In order to compare the above-mentioned approaches some features have to be carefully analyzed: (1) the support for automatic processor integration; (2) the availability of mechanisms for mapping the application into the target platform; (3) support to processor architecture description and modification; (4) availability of analysis mechanisms at different abstraction levels; (5) system-level specification language; (6) architecture specification language; (7) support to generic application domains; (8) simulation support. The results from the analysis of these features are summarized in Table 1. The legend in the table is interpreted as follows. A "x" means full feature support. An entry marked with "p" means partial feature support. An "?" mark means that the researched literature is not clear about that feature. Finally a "-" indicates that the feature is definitely not supported.

The tools in the table above can be divided in two groups. The first group, including CoWare, StepNP, component based design (CBD) and SpecC, uses simulation or co-simulation for system evaluation. This shows that simulation continues to be one of the most used evaluation mechanisms for digital systems. Besides the use of simulation StepNP provides high-level analysis mechanisms to check system level communication and behavior in the target architecture. For example, the designer can follow packet processing while simulating the routing application. The limitation of StepNP is that this analysis mechanism is not generic, but specific to the packet routing application domain.

The tools in the second group (SPADE, COSY, Sesame) provide partial support for simulation, since the execution of the application in the target architecture is trace driven. The Metropolis is an exception in this group as it provides support for simulation running on the target architecture.In these tools, the designer can analyze communication in the target architecture at the system-level. The same lack of flexibility of StepNP is also observed in

these tools. They are used for specific application domains: multimedia and digital signal processing applications.

Another relevant point in Table 1 is the lack of support to the automatic integration and description/modification of the processor specification. The tools in the first group provide just partial integration support as they provide *ad hoc* schemes for integrating processor in the target architectures. The tools in the second group do not provide this support. This is a very desirable feature, since processors are very critical components on almost all embedded system designs. If designers cannot modify processor features, such as cache configuration, they are severely limited in their ability to optimize the whole embedded system architecture.

A closer look to the first group shows that they use two languages for architecture description: SystemC and SpecC. These are system level languages that allow the description of high-level behavior as well as low-level hardware behavior. CoWare uses also SystemC as the specification language. The same happens to component based design that accepts heterogeneous languages but which has SystemC as a base language. SpecC tools use SpecC for both system specification and architecture. Thus, the use of a single language in all the parts of the system seems to be tendency that is gradually consolidating. Heterogeneous languages are used mainly for compatibility and reuse. Even in the second group, Sesame uses the same language for system specification and architecture model.

## 3 Modeling multiprocessor platforms using ArchC and SystemC

Multiprocessor platform simulation presents its own nuances and particularities that are depicted in the platform example of Fig. 1. First, it is necessary to integrate an instruction set simulator (ISS) for each processor instance in the platform. Different ISSs can be used depending on the design space exploration done by the designer. In the example of Fig. 1, which shows a typical platform for computing intensive control applications, four ISSs are being used: 1 RISC processor that can be a MIPS or SPARCV8, and three instances of the 8051 micro-controller (a CISC architecture).

The embedded application running on a multiprocessor platform is composed of concurrent tasks, each running on one processor and communicating using the interconnection

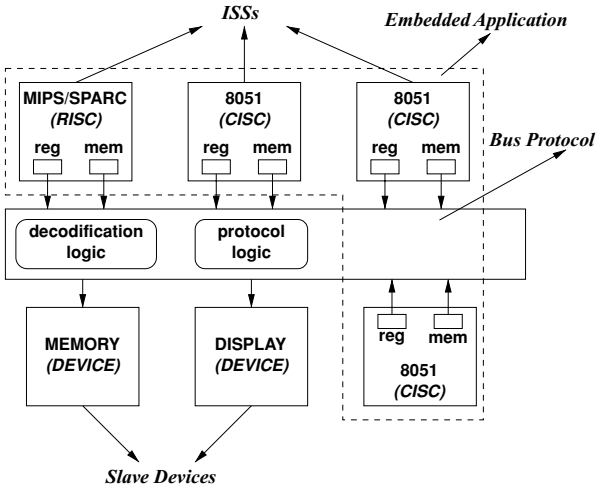



**Fig. 1** A multiprocessor platform model

structure protocol of the platform. The ISS simulators must be synchronized to run in parallel and also to implement the platform interconnection protocol. In other words, read and write instructions (memory-mapped IO) and IO instructions, that use special registers must make use of the platform protocol while executing in the simulator.

It is also necessary to define how the addresses issued to the interconnection structure protocol will be decoded. The decoding logic determines the address ranges for each slave component connected to the bus. We are assuming a bus as an interconnection structure, though our approach can be easily extended to any other mechanism, like a Network-on-a-Chip (NoC).

The first difficulty in implementing a simulation model for a multiprocessor platform is to find the proper ISS simulators for the processors. The ISS simulators can be obtained basically from three sources: stand-alone simulators, third party components or from the output of ADL based tools. Stand-alone simulators are programs designed to run executable code compiled to a target architecture. The problem in using this kind of simulator is that two applications need to run in parallel, the platform simulator and the ISS simulator. This results in complicated *ad hoc* schemes that are difficult to handle. A third party component ISS is a processor simulator normally described in some hardware description language and distributed as it is by its creator. The most common are described in VHDL/Verilog and recently in SystemC. In this case, it is necessary to create a wrapper around the component and to add it to the platform model. When the platform simulator is the same as the processor component, the integration task is easier. On the contrary, it is also necessary to implement the communication between the platform and the HDL simulator. A third alternative makes use of an ADL based tool to generate a component from a processor architecture description. The component is then used as a third party simulator. In this case, the designer has more flexibility to modify the processor, or even to build new ones. Most of the platform simulation environments use the first and second approaches. The consequence is a reduced set of available processors. Others use some ADL languages that allow the modifications in the processors, but that are limited to some families. Finally, in cases like the Coware Platform ArchitectProcessor Designer [10], a platform simulation and a processor development tools are used together.

The approach in this paper is based on the use of SystemC to model system level behavior, and also the simulation platform. The SystemC simulation models of platform processors can be obtained from ArchC descriptions, a SystemC based ADL; additionally, SystemC is also used to specify other platform components, such as bus and devices.

An overview of this approach can be seen in Fig. 2. The system behavior is modeled as concurrent processes that communicate through SystemC *sc_fifo* channels. The concurrent threads functionality and communication are extracted from the system behavior description and mapped to the components of the target architecture.

Each concurrent process in the system behavior is refined, manually, to C code and is assigned to run in one of the processors in the target platform. The C code is compiled using a cross-compiler that generates executable code for the specific processor. This code is loaded in the platform simulator and executed.

When modeling the target platform, ArchC is used for specifying the processors and PArchC for describing the interconnections structures and devices that compose the structure of the platform. This structure is extracted using the acxtor tool prior to the mapping. The platform description resulting from this phase does not include any mechanism for implementing synchronization and communication among processors. Such kind of mechanism is introduced during the mapping phase. The main goals of mapping are twofold. First the application must be compiled for some processor in the platform. The second goal

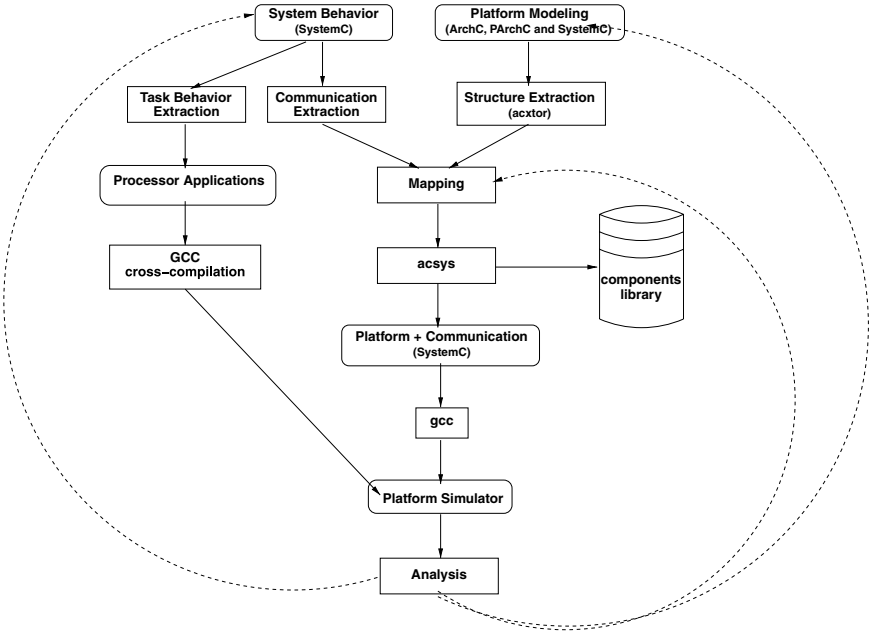



**Fig. 2** Methodology overview

is to generate a platform description including a mechanism for processor communication through the platform interconnection, if the application includes communicating processors. The proposed mapping approach provides some support for implementing this communication mechanism in a (semi) automated way. During this phase, the designer should define the processes allocation by taking into account the processors available in the platform. The mapping phase produces the compiled application and a platform description, which are capable of implementing process synchronization and communication when the application runs.

After the mapping phase it is used the *acsys* tool that takes the description of the platform, the mapping information and the components (interconnection structure and devices) from the components library and generates SystemC code of the platform.

The platform description with embedded communication can be compiled resulting in an executable file, which is the platform simulator. When running this file in conjunction with the application code some metrics can be extracted, allowing the designer to change either the platform or the application mapping, in order to meet the design constraints. The next section gives a short introduction to ArchC, followed by a description of the platform modeling mechanism, and the proposed strategy to implement communication among processes.

## 4 Describing processors using ArchC

ArchC is an open-source Architecture Description Language (ADL) [5] designed to enable the description of processor cores. ArchC supports different levels of abstraction, allowing the designer to start at a functional level and refine the processor until it reaches a cycle-accurate model. A set of tools can be generated using ArchC: interpreted SystemC simulators and compiled C++ simulators, GNU assembly (`gas`) and linker (`ld`) back-end, GNU GDB support interface and co-verification checker between two ArchC models.

**Fig. 3** Sample code of an architectural description of the SPARCV8 processor using ArchC (sparcv8.ac)

```
1 AC_ARCH(sparcv8){
2     ac_wordsize 32;
3     ac_mem MEM:5M;
4     ac_regbank RB:32;
5     ARCH_CTOR(sparcv8) {
6         ac_isa("sparcv8_isa.ac");
7         set_endian("big");
8     };
9 };
```

Every processor description in ArchC should follow a strict design roadmap indicating (by its version number) the milestones reached in the model implementation. For example, a 0.4 version processor model is able to emulate the Linux operating system calls and a 0.7 version model is capable of running MediaBench [21] and MiBench [14] programs.

### 4.1 ArchC syntax and semantics

An ArchC description is divided into two parts: the `AC_ARCH`, which contains the architectural resources used by the model, and `AC_ISA` that describes the instruction set architecture.

Figure 3 shows the architectural description (`AC_ARCH`) of a SPARCV8 [27] functional processor model. The keywords used to specify the resources in the example and their meanings are:

- `ac_wordsize`: specifies the architectural word size in bits (line 2)
- `ac_mem`: declares a memory vector that the simulator will use (line 3)
- `ac_regbank`: declares a register file and assigns a name to it[1] (line 4)
- `ac_isa`: specifies the file that contains the processor ISA description (line 6)
- `set_endian`: specifies the processor endianness (big or little) (line 7)

There are also keywords to describe pipelines, pipeline registers, caches, etc. For more information, refer to [28].

Figure 4 shows a subset of the `AC_ISA` description for the SPARCV8 processor presented in Fig. 3. Three instruction formats are declared using the keyword `ac_format`. After that, some SPARCV8 instructions are declared using the keyword `ac_instr` and one format as template. Notice that the instruction identifier, declared here, does not need to be the instruction mnemonic. Every instruction has one or more `set_asm` method calls to declare its assembly syntax and a `set_decoder` method to describe the decoding sequence. As an example, the `add` instruction, identified as `add_reg`, uses the `Type_F3A` format and is encoded with fields `op=0x02`, `op3=0x00` and `is=0x00` (line 14).

One important feature of ArchC is that a SystemC instruction decoder is automatically generated based on the `set_decoder` methods for each instruction. The processor designer has only to specify the behavior of the instruction in SystemC but not the behavior of the instruction decoder.

The next step, after modeling `AC_ARCH` and `AC_ISA`, is to describe the instruction behavior. Figure 5 shows the simplest implementation for instructions `add_reg` (add register) and

[1] In this example, the SPARCV8 processor has only one register window. A full SPARCV8 model is available at the ArchC web site [17].

```
AC_ISA(sparcv8){
  ac_format Type_F1 = "%op:2 %disp30:30";
  ac_format Type_F2A= "%op:2 %rd:5 %op2:3 %imm22:22";
  ac_format Type_F2B= "%op:2 %an:1 %cond:4 %op2:3 %disp22:22:s";
  ac_format Type_F3A= "%op:2 %rd:5 %op3:6 %rs1:5 %is:1 %asi:8 %rs2:5";
  ac_format Type_F3B= "%op:2 %rd:5 %op3:6 %rs1:5 %is:1 %simm13:13:s";

  ac_instr<Type_F3A> ld_reg, add_reg, sub_reg, and_reg,
                     or_reg, xor_reg, umul_reg, smul_reg;
  ac_instr<Type_F3B> ld_imm, and_imm, or_imm, xor_imm, xnor_imm;

ISA_CTOR(sparcv8){
    add_reg.set_asm("add %reg, %reg, %reg", rs1, rs2, rd);
    add_reg.set_decoder(op=0x02, op3=0x00, is=0x00);
    ld_imm.set_asm("ld [%reg + \%lo(%expL10)], %reg", rs1, simm13, rd);
    ld_imm.set_asm("ld [%reg + %imm], %reg", rs1, simm13, rd);
    ld_imm.set_asm("ld [%imm + %reg], %reg", simm13, rs1, rd);
    ld_imm.set_asm("ld [%imm], %reg", simm13, rd, rs1="%g0");
    ld_imm.set_decoder(op=0x03, op3=0x00, is = 0x01);
  };
};
```

**Fig. 4** Sample code of an instruction set description for the SPARCV8 processor using ArchC

```
1  ac_behavior(instruction) {
2    ac_pc += 4;
3    }
4  ac_behavior(Type_F3A) {}
5  ac_behavior(add_reg) {
6    RB.write(rd, RB.read(rs1) + RB.read(rs2));
7    }
8  ac_behavior(ld_imm) {
9    RB.write(rd, MEM.read(RB.read(rs1) + simm13));
10   }
```

**Fig. 5** Sample behavior for instructions add_reg and ld_imm in ArchC (sparcv8-isa.cpp)

`ld_imm` (load immediate). As an example, the instruction `add_reg` behavior, declared as `ac_behavior(add_reg) (lines 5–7)`, is just one line of SystemC code, containing two register reads from the previously declared register file (`RB`) and one register write. ArchC also allows a hierarchical behavior description. Since every SPARCV8 instruction is 4 bytes long, the program counter (`ac_pc`) increment is done only once inside the global behavior[2] (lines 1–3), `ac_behavior(instruction)`, which is automatically called before every instruction execution. ArchC also has one behavior specific for each instruction format. In this case, they are empty as they are not necessary for this particular model. In the example, the actual execution of the add instruction follows this calling sequence: `ac_behavior(instruction)` followed by `ac_behavior(Type_F3A)` and finally `ac_behavior(add_reg)`. This sequence is automatically handled by the simulator.

[2] This simplified version does not take into account the branch annul bit. You can see how it is implemented in the model available from the ArchC web site [17].

### 4.2 ArchC design support

ArchC provides a set of tools to help in design space exploration. The interpreted simulator generator (`acsim`) is the most used. It receives, as input, an ArchC model, composed of `AC_ARCH` and `AC_ISA` and generates a SystemC simulator that is linked to the instruction behaviors described above. In its 2.0 version the interpreted simulator can reach up to 10MIPS on a Intel Mobile Celeron 1.7 GHz, 512 Mb RAM, running Kubuntu Linux 5.10, using g++ 4.0.2 (200500808) with the compilation parameters $-$O3 -march $=$ pentium4m. The simulation speed can be improved using the C++ compiled simulator (`accsim`), which can reach up to 200 M instructions per second on the same machine. This huge speedup comes from improvements on the simulator structure and on pre-processing [5] the binary program to be executed. Both kinds of simulators are able to emulate a set of the Linux system calls, so that big programs, with inputs and outputs, can run without any source code modification.

A GNU GDB debug interface can be enabled into the interpreted simulator to allow debugging and inspection of the execution environment. A GNU assembler (`gas`) and linker (`ld`) back-end can also be generated automatically. For these three features, a few more information must be given in the processor description, to expand the assembly syntax and to identify the processor resources.

Processors already implemented in ArchC, range from small micro-controllers like PIC and Intel 8051 to big processors like MIPS, Sparc and PowerPC. They are all available for download, along with the ArchC toolset, at the ArchC web site [17].

In order to provide support for platform modeling the ArchC language has been extended. From now on this extension will be called PArchC (Platform ArchC) and will be discussed in the following sections, while the original unmodified version of the language will be called ArchC.

## 5 Describing multiprocessor platforms

When using a design paradigm based on platforms the designer must have platform models at distinct abstraction levels: functional models are necessary for validating the application, transaction level models are interesting to evaluate bus contention, whereas cycle-accurate models are necessary to performance evaluation. The approach described below allows to specify platforms at these different abstraction levels.

In the current version we are supporting the platform model depicted in Fig. 6. Each processor has local memory and a shared memory is used only for process communication. In order to allow the specification at distinct abstraction levels a wrapper is located between the processor and the interconnection mechanism. This wrapper provides an interface of the processor with different buses, or the specifications of a bus at distinct abstraction levels. Currently we are supporting OCP and AMBA interfaces.

Besides the role of interface adapters, wrappers are responsible for allowing the synchronization between processors and the interconnection structure. For that purpose they implement a master/slave protocol enabling a SystemC processor to send and receive information from a SystemC bus model, through read and write operations. The adaptation of processors and buses with different endiannesses, data and address widths, is also done at the wrapper.

The standard ArchC language supports the description and implementation of a standalone processor simulator. In order to support the specification of a multiprocessor platform, as well as the implementation of a mechanism for synchronization and communication between

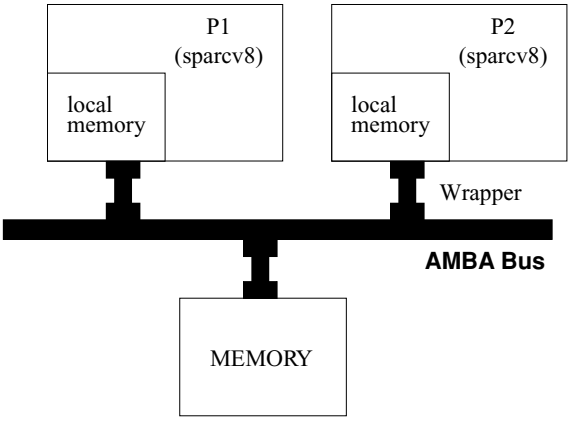



**Fig. 6** A multiprocessor platform

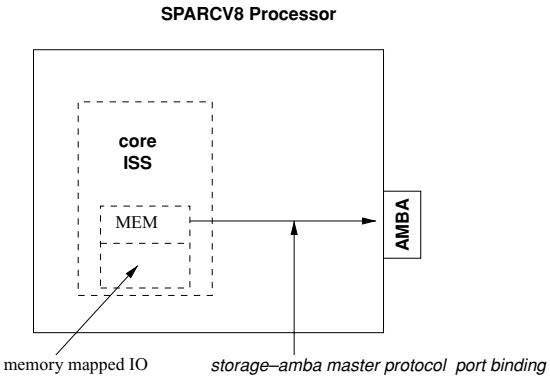



**Fig. 7** The supported processor model

processors, the language has been extended as depicted in the SPARCV8 specification of Fig. 8 [4].

The extension includes new constructs for specifying mechanisms to connect the processor with platform devices, such interconnection, memories, etc. The `ac_protocol` declaration has been added in order to specify ports that implement a specific protocol. It supports the declaration of master or slave ports and is defined by the 6-tuple $\langle t, d, n, aw, dw, s\rangle$. The parameter $t$ determines the type of supported protocol. The $d$ parameter determines the direction of the port, i.e. master or slave. Parameter $n$ represents the name of the protocol port, and parameter $aw$ is the address bus width. Parameter $dw$ determines the data bus width. The last parameter $s$ gives the number of protocol ports of that type in the processor.

The protocol port declaration can be seen in line 6 of Fig. 8. It declares one master port which implements the AMBA protocol [1]. The port has an address and data bus width of 32 bits.

The protocol port declaration simply defines the processor ports and the protocol each port implements. The `bindsTo` declaration is a method that specifies which storage element of the processor is connected to the processor port (registers or memory). Line 12 of Fig. 8 shows one example of such binding. In this case the memory `MEM` is connected to the `AMBA_BUS` protocol port implementing a memory mapped IO access. The use of `bindsTo`

```
1  AC_ARCH(sparcv8){
2      ac_wordsize 32;
3      ac_mem MEM:5M;
4      ac_regbank RB:32;
5      //!Protocol port declaration (EXTENSION)
6      ac_protocol<AMBA, MASTER> AMBA_BUS(32, 32);
7
8      ARCH_CTOR(sparcv8) {
9          ac_isa("sparcV8_isa.ac");
10         set_endian("big");
11         //!connection declaration (EXTENSION)
12         MEM.bindsTo(AMBA_BUS);
13         //!Memory map declarations (EXTENSION)
14         MEM.set_range(0x0, 0x500000);
15         AMBA_BUS.set_range(0x500001, 0xa00000);
16     };
17 };
```

**Fig. 8** AC_ARCH description and architecture extensions (modified sparcv8.ac)

implements the connection of the memory to the protocol port. It is also necessary to define the addresses range used for the I/O operations.

The `set_range` method is used to define the memory map of the processor. This method is used when an storage element has been bound to a protocol port. It determines the address ranges that are accepted by the storage element and by the protocol port. Lines 14 and 15 of Fig. 8 declare that addresses in the range (0x00000000,0x05000000) are recognized by the memory `MEM` while addresses in the range (0x0500001, 0x0a00000) are accepted by the protocol port `AMBA_BUS`. The memory map for a component can be composed of several address ranges, i.e., the designer can use the `set_range` method multiple times for a single storage element or protocol port.

In summary, the processor description in ArchC must include the ports declarations, how these ports are connected with memory or registers, and the memory map in the case of memory mapped IO. New constructors have been defined, with a syntax very similar to the ArchC syntax, in order to ease the task of plugging a core to the platform.

In order to be able to communicate through a bus, the processor must include some interrupt mechanism. In this sense, PArchC extended ArchC to support the modeling and simulation of interrupts. The extensions include the definition of the INTERRUPT protocol port, the interrupt declaration, and mechanisms to describe the interrupt controller and interrupt behavior.

The interrupt protocol port also describes a master/slave relation where the processor contains a slave interrupt protocol port. External devices can interrupt processors with this port.

The processor of Fig. 9 uses memory mapped I/O and has been modeled by binding memory MEM to the AMBA master protocol port, named *AMBA*. In the example we also show one of the interrupts provided by the SPARCV8 processor, *interrupt*`_level_1` and its interrupt controller. This interrupt is bound to the interrupt slave protocol port INT1. Any interrupt request issued at the INT1 port is handed to the `interrupt interrupt_level_1`.

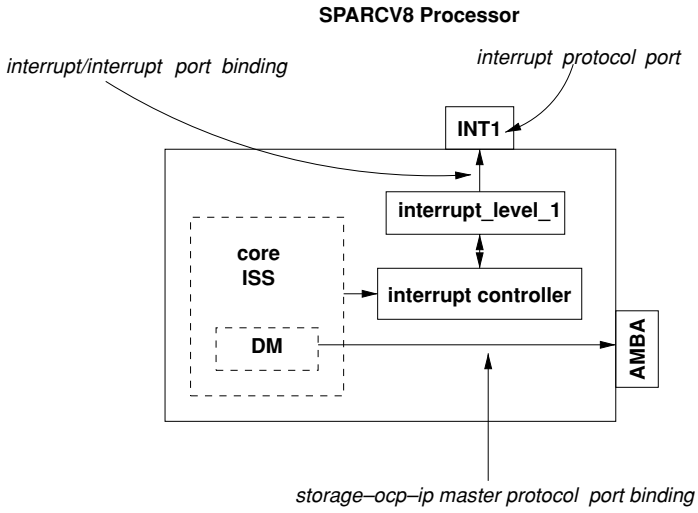



**Fig. 9** SPARCV8 processor model example

The `ac_interrupt` declaration is a simple way for the processor designer to specify processor interrupts (see figure below). Using this construct the designer declares an interrupt with a specified name. For each interrupt the simulator engine checks the occurrence of a valid interrupt request prior to the instruction execution. This procedure is transparent to the processor designer.

The `ac_interrupt` declaration syntax is shown below. It is composed of the `ac_interrupt` keyword followed by the interrupt name provided by the user and ending with a semicolon.

*ac_interrupt* interrupt_name;

Associated with the interrupt declaration it is also necessary to describe the interrupt behavior. PArchC extended ArchC similarly in the ISA declaration scheme, i.e., a template of the behavior of the interrupt is generated automatically. The designer must complete this template using SystemC constructs. The simulator generated for the processor checks automatically the occurrence of a request for the interrupt and when this happens it executes the interrupt behavior specified by the designer.

The interrupt controller functionality of the processor is implemented by two methods, one implements the processor behavior when an interrupt request is issued and the other implements the arbitration when multiple requests are pending. The implementation of the `request_trap` and `get_pending_request` methods for the sparcv8 processor are shown in Fig. 11. The request trap method is called when an internal or external interrupt is issued to the processor while the get pending request method is used to implement the interrupt priorities when more than one interrupt has been requested. These methods are available to the processor designer and preserves the ArchC style to describe the ISA. In the same way an "architecture_name"-isa.cpp template file is automatically generated allowing the designer to describe the instruction behavior, it is also generated a "architecture_name"-

```
...
ac_interrupt interrupt_level_1;
...
```

```
void ac_behavior (interrupt_level_1) {
  PSR.write(PSR.read() & 0xfffffffd);
  CWP = CWP  - 0x10;
  ...
}
```

**Fig. 10** Interrupt declaration and behavior

```
void request_trap (int trap_id, int dev_id) {
  switch (trap_id) {
    case (interrupt_level_1_TRAP) :
      //TODO: Insert your code here
      trt[trap_id] = true;
      break;
    default :
      break;
  }
}

int get_pending_request() {
  //TO_DO: Insert your code here
  //Checks if interrupts are enabled
  if (!(PSR.read() && 0x00000020)) {
    //interrupts are disabled
    return -1;
  }
  typedef map<int, bool>::const_iterator CI;
  for (CI p = trt.begin(); p != trt.end(); ++p) {
    if (p->second) {
      trt[p->first] = false;
      return p->first;
    }
  }
  return -1;
}
```

**Fig. 11** SPARCV8 interrupt controller request and arbitration methods

trap.cpp template file with the `request_trap` and `get_pending_request` methods that implement the interrupt request and arbitration respectively.

The processor designers have full access to any processor resources declared in the processor architecture description. For instance in the `get_pending_request` example of

```
1  // Processors, memory and bus descriptions
2  #include "sparcv8.ac"
3  #include "AMBA.ac"
4  #include "AMBASlaveMem.ac"
5
6  AC_SYSTEM(platform) {
7      //Master port
8      ac_protocol<AMBA> AMBA_BUS(32, 32);
9      //Processor declaration
10     ac_processor<sparcv8> P1, P2;
11     //Bus declaration
12     ac_system_bus<AMBA> BUS(32,32);
13     //Device instance declaration
14     ac_device<AMBASlaveMem> MEMORY;
15     SYSTEM_CTOR(platform) {
16         //set component parameter
17         MEMORY.set_parameter(size, 5242880);
18         //components connection
19         P1.AMBA_BUS.bindsTo(BUS);
20         P2.bindsTo(BUS);
21         BUS.bindsTo(MEMORY);
22         //Memory map
23         MEMORY.set_range(0x600000, 0xA00000);
24         AMBA_BUS.set_range(0xA00001, 0xB00000);
25         //Sets abi support
26         sparcv8.set_abi();
27         //Loading applications
28         P1.load_obj("producer");
29         P2.load("consumer");
30     };
31 };
```

**Fig. 12** AC_SYSTEM and include declarations (platform.ac)

Fig. 11 it can access register PSR. It can make use of typical C++ constructs like the map declaration using the `typedef` C/C++ keyword, a SystemC feature.

The `request_trap` method takes the interrupt identification and device identification as arguments. The device identification feature allows the designer to share a single interrupt by several external devices.

Figure 12 shows the specification of a platform including two SPARCV8 processors connected to an AMBA bus, as depicted in Fig. 6. Each processor has its local memory hierarchy but a shared memory is included in the platform for processor communication. The memory address range dedicated to IO is (0xA00001, 0xB00000).

The constructor `#include` allows the reuse of processor descriptions in ArchC, as well as SystemC bus and device specifications. The use of this constructor can be seen in lines 2–3 of Fig. 12. The first line includes the AC_ARCH files for the SPARCV8 processor, whereas the next two lines include an AMBA bus SystemC TL model and a memory model.

The platform itself is modeled using the AC_SYSTEM constructor. This declaration has a similar syntax to the AC_ARCH declaration for processors. AC_SYSTEM is a composed declaration divided in two parts. The first part is used to declare the platform components such as processors, buses and devices.

The processor and device declarations have a syntax similar to the protocol port declaration, in the processor description. Processor declaration is specified using the ac_processor constructor, which includes the processor name (same name of the ArchC description) and names of processor instances in the platform, separated by commas. An example of an ac_processor declaration can be seen in line 10 of Fig. 12, where two SPARCV8 processors named P1 and P2 are declared. Device declaration is done similarly, but using the ac_device constructor. Line 14 declares a memory of type AMBASlaveMem.

By using the ac_system_bus template, the designer can declare interconnection structures. The declaration specifies the bus name, bus instances names and the address and data bus widths. An example of system bus declaration can be seen in line 12 of Fig. 12, where an AMBA bus named BUS is declared. Its address and data bus widths are 32 bits.

The second part of the AC_SYSTEM declaration, delimited by the SYSTEM_CTOR keyword, is used to build the platform. It has a similar syntax as the AC_CTOR part of the processor description. It is used to set the memory size parameter value and specifies the connection of the memory hierarchy to the processor. The memory size is set to 5,242,880 (5 Mb) by the set_parameter declaration. The connection of the components is performed with the same bindsTo statement used in the AC_ARCH declaration. Despite having a similar syntax, the semantics of the this statement is *master connects to slave*. The master connects to the slave in one of two ways: explicitly declaring the ports to be connected or implicitly (only one port). In the explicit form, the port names of master and slave devices being connected must be provided. This is necessary, in the case master and/or slave having more than one protocol port. One example of an explicit connection is given at line 17 of Fig. 12. Port AMBA_BUS of the instance P1 of the SPARCV8 processor is explicitly connected to the BUS. Examples of implicit connections are given at lines 18 and 19 of Fig. 12.

Once the platform structure has been specified, it is necessary to specify how the platform will behave, by giving information about the memory map and the application running on each processor. For this purpose, three news methods have been defined: set_range, load and load_obj.

The set_range method is used to define how the decoding logic of the interconnection structure will decode the addresses issued by the master elements. Using set_range the designer informs, in a simple way, the address ranges for each slave in the system. He/she does not have to take care on how the interconnection structure will implement the address decoder. Method set_range has a similar syntax and semantics as the memory map declaration used for the memory hierarchy of the platform. Line 21 of Fig. 12 shows an example of a memory map using set_range. Any address in the range (0x600000, 0xA00000) will be decoded to the MEMORY device .

The embedded application is composed of executable code for each one of the processors in the system. The mapping of the embedded application is performed using the other two behavioral declarations. Method load_obj allows the designer to directly map binary code to the chosen processor. Using load it is possible to map executable code in the ArchC hexadecimal format [28]. Lines 26 and 27 of figure 12 show the mapping of the application, composed of one executable code in binary format and one in hexadecimal format.

The AC_BUS declaration is used to define the interface of the interconnection structures of the platform. In the example of Fig. 13 an AMBA bus is declared. The bus has the following

```
1  AC_BUS(AMBA) {
2      ac_protocol<AMBA, MASTER> m_port(32,32) : 1;
3      ac_protocol<AMBA, SLAVE>  s_port(32,32) : 2;
4  };
```

**Fig. 13** AC_BUS declaration (amba.ac)

```
1  AC_DEVICE(AMBASlaveMem) {
2      ac_parameter<INT> size;
3      ac_protocol<AMBA, SLAVE> s_port(32,32);
4  };
```

**Fig. 14** AC_DEVICE declaration (ambaslavemem.ac)

AMBA compliant ports, one instance of a master port and two instances of a slave port. As the parameters in the protocol port declaration are literals, the bus address and data ports are parameterized. The implementation of the each declared protocol should be available in a library.

Similarly, the `AC_DEVICE` declaration states that there is a SystemC module that implements the declared interface. In the example of Fig. 14 the AMBASlaveMem device is declared. In this case, however, the parameters of the protocol port are integer values, meaning that the device can only be connected to an AMBA bus with 32 bits of address and data widths.

Using the `ac_parameter` declaration the component designer can define parameters types that can be set by the system designer during the platform creation. In the case of the AMBASlavMem component of Fig. 14 there is the size parameter of type INT. In this case the size of the memory can be set by the system designer.

The above description is processed by the *acsys* tool. It takes the platform description and generates a SystemC simulation model of the platform. By using a simple command line, the designer can generate code for the platform.

The functional implementation is based on the SystemC AMBA channel model [1]. This channel allows one master and one slave components to exchange information that is parameterizable in the data and address types. The main advantage of this type of channel is that they are timeless and provide a much better performance during simulation. On the other hand, the lack of time information makes it difficult to detect synchronization problems. Using this type of channel the designer can validate the functionality of the application running on the platform, but cannot identify synchronization problems nor evaluate communication performance.

The usage of the *acsys* tool is quite simple. It is a command line tool that takes three arguments. The first is the name of the PArchC file containing the platform description. The second argument determines the abstraction level of the communication protocols ports of the platform. The last argument determines whether or not the system should generate trace files for the protocols. The usage of *acsys* is shown below:

> acsys *input file* [-p*[abstraction level flag]*] [-tp]

## 6 Mapping an application onto a simulation platform model

In order to tune the platform for an application, or application domain, the designer should be able to simulate the application by taking into account different platforms, or platform configurations. For each platform or platform configuration to be evaluated, the designer must first map the application into it. If the application includes communicating processes, it is necessary to introduce additional components for implementing synchronization and communication between the processors using the platform interconnection structure. Embedded software must also be developed to implement communication at the processor side.

In the case of applications written in SystemC, our methodology includes a mechanism for implementing synchronization and communication processes running on different processors, in a semi automated way.

Synchronization by event waiting and notification, as well as communication through channels, are supported by mechanism as explained below.

### 6.1 Process synchronization and communication in SystemC

SystemC 2.0 supports the concept of dynamic sensitivity. This mechanism allows processes to be sensitive to the notification of different events during its execution. The dynamic sensitive mechanism can be better understood by using the simple producer consumer example of Fig. 15. The producer process halts the execution when it executes the `wait()` method, and resumes the execution after the consumer process notifies the *start_event* using the `notify()` method.

By using dynamic sensitivity for synchronizing processes, the designer can define several different events and make the processes sensitive to any of them during its execution.

Task communication is implemented in SystemC by the use of channels. SystemC specifies two types of channels: primitive and hierarchical. Hierarchical channels are used to model

**Fig. 15** Producer consumer example

```
1  void producer_process()
2   { sc_int<32> j = 0;
3     while (1)
4       {wait(start_event);
5        for (j = 0; j < 100; j++)
6          { CH0->write(j); }
7       }
8     }
9
10 void consumer_process()
11  { sc_int<32> j;
12    while (1)
13     { notify(start_event);
14       j = CH0->read();
15       printf("j == %d\n", j);
16     }
17  }
```

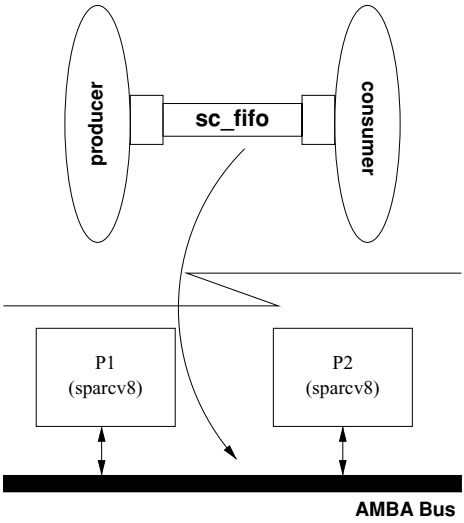



**Fig. 16** The mapping problem

complex communication like buses and are not in the scope of this paper. Primitive channels, on the other hand, are the basis to model system level communication mechanisms like FIFOs, mutual exclusion (MUTEX), etc.

Channels are characterized by the interface they implement. An interface defines the methods the channel must implement. For instance the `sc_fifo` channel implements the *write()* and *read()* methods of the sc_fifo_read_if and sc_fifo_write_if interfaces. Fifo channels have been used in the producer consumer example of Fig. 15.

Another feature of channels is that they include the above mentioned synchronization mechanisms. In that example, the consumer processes will block when the CH0 fifo is empty, during a read operation. It will remain blocked until the notification of the *data_written_event*, by the write operation, is executed in the producer process.

When executing a SystemC application including communicating processes, the synchronization and communication is implemented by the simulation kernel. If this application will run in a multiprocessor platform, the synchronization and communication must be implemented by using the platform components.

The problem is how to map these resources to a multiprocessor target architecture, as depicted in Fig. 16. Designers have to do a lot of manual work to achieve that. It is necessary to define the memory addresses used by the channel. It is also necessary to define how processor resources like interrupts are used.

In the following we describe how synchronization and communication are implemented in our methodology.

## 6.2 Implementing process synchronization and communication

Once the functionality of the system specification has been validated by simulation, it is necessary to map its behavior to the target platform. One example of such target platform is given in Fig. 16. It shows a platform composed of two processors and an AMBA bus. Some scheme must be included in order to support the implementation of communication and synchronization among processors using the interconnection structure and inserting a shared memory for communication purposes.

A question that arises from the mapping task is how to implement communication between application tasks in the SystemC model of the target platform. In our approach a communication architecture including event managers, critical region controller and embedded software is automatically generated and inserted in the platform description [3].

The mapping step is taken after the communication and architecture extraction as hab been seen in Fig. 2. Once the designer has specified the platform, the *acxtor* tool takes the platform description and extracts structural information. The structural information includes a number of features of the processors, the address range of the shared memories and the interconnection structure of the platform.

In parallel, information from the application must also be extracted from the description, or provided by the designer. Among others, application information includes tasks, event, and channels names. The tool also extracts, from the application, which channels are being used at each task, as well as through which interface are the events notified and expected. A technique for extracting the information application has been developed by using introspection tools. Once the application and platform information has been extracted, the designer defines process and communication allocation, i.e., which processor, processes and bus will be used to implement communication.

The designer gives the allocation information, either using a text file or by fulfilling electronic forms available from a graphical interface. The *Application Mapper* tool takes the mapping, the application information, and the above mentioned platform information and generates appropriate C code. This code runs communicating processes inside the processors, implementing the wait and notification events at the processor side.

The *Platform Mapper* tool also takes the mentioned information to generate a platform model which is able to implement process synchronization and communication. A complete SystemC platform model can be obtained using the *acsys* tool. In the next sections, the strategy for implementing communication and synchronization will be explained.

### *6.2.1 Implementing process synchronization*

As SystemC synchronization is represented by assynchronous events that processes can notify or wait for nofification, the proposed approach for the implementation of synchronization between tasks running on different processors is based on hardware and software support. Processors that are running tasks that wait for a event notification are interrupted and the interrupt routine handles the notification. Hardware support is provided by Event Manager Modules (EVM). These devices are responsible for generating the appropriate interrupts when an event notification occurs.

Another feature of SystemC is that the implementation of its primary channels, like sc_fifo, makes use of assynchronous event. In the case of the sc_fifo channel used in this work, there are two events *data_read_event* and *data_written_event*. The modified platform model, which capable of implementing process synchronization and communication is depicted in Fig. 17. For each processor an event manager module (EVM) is inserted, which is specified in SystemC. The main goal of this device is to store the events expected by the task running in the processor, as well as the events notified by the task. Additionally, it must interrupt the processor in the case an expected event has been notified by some processor. Each EVM device is connected to the bus and also to the processor, through a protocol port.

In order to allow each task to notify and wait on events, embedded software is automatically inserted into the application code. This code is organized in layers as explained later.

The EVM device is responsible for handling the wait and notification of events between the tasks running in the processors of the platform. It is used to implement *wait(event)* calls in Sys-

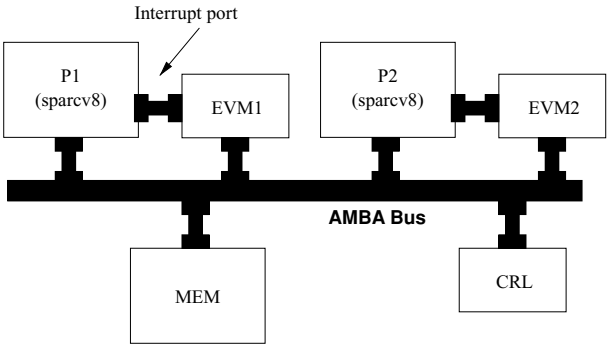



**Fig. 17** A platform model for implementing process communication and synchronization

temC in the system behavior description. One event manager device is assigned to each processor in the platform and it interrupts the processor every time one event that is being expected by the task running on the processor is notified by another task running in a second processor.

It is composed of two main components: the notification fifo and wait event table. The wait event table stores the events that are expected by the task. It plays a fundamental role in the system as event notifications that are not being expected by the processor associated with this EVM are ignored by the EVM. This avoids the generation of unnecessary interrupts in the processor. The number of table entries in the EVM is equal to the maximum number of events that can be expected by the tasks running in the processor.

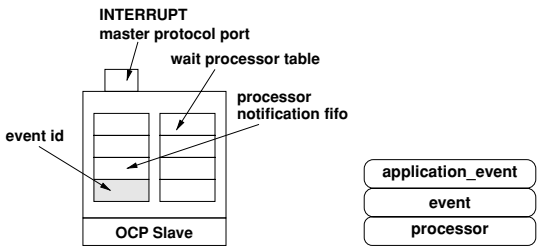
The role of the notification fifo is to discipline the order of event notifications that must be treated by the processor whose task has been notified. The size of the table is also equal to the maximum number of events that can be notified.

The event manager also contains two ports. One protocol port that implements the interface with the interconnection structure of the platform. The second is a master interrupt protocol port that is connected to the slave interrupt protocol port of the processor.

The structure of an EVM device is shown in Fig. 18. It is composed of a fifo including all events to be notified and a wait table, which includes the expected events. The communication with the processor is done through a protocol port, which supports a master protocol to interrupt the processor.

In order to better understand the synchronization scheme consider the producer consumer example running in the platform of Figure 17. Task Producer is running on processor P1, whereas task Consumer is executing in processor P2. Consider the scenario depicted in

**Fig. 18** EVM device and software layers




(a) event manager device (b) event sw layers

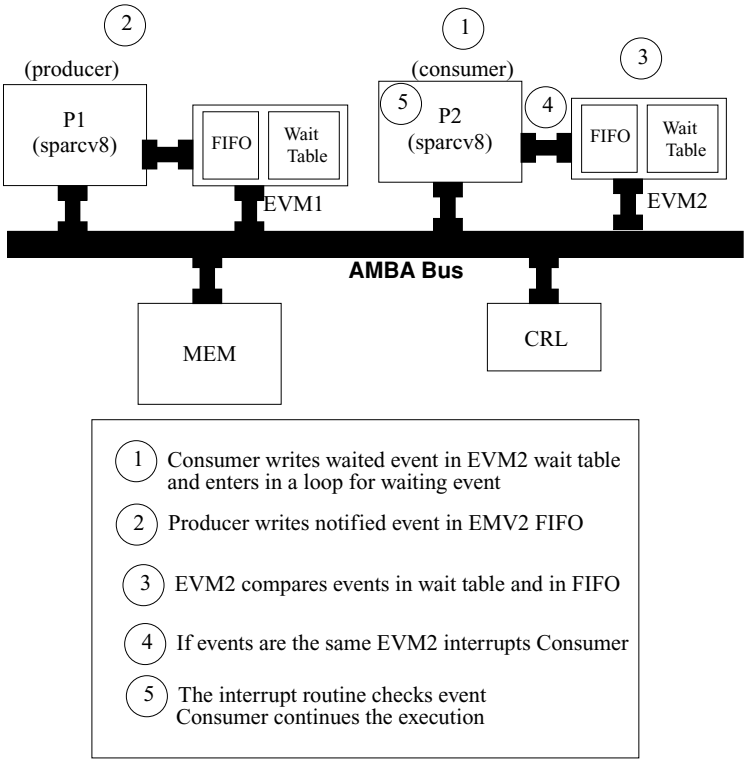



**Fig. 19** A platform model for implementing process communication and synchronization

Fig. 19, where the Consumer is waiting on event *start_event* (1). The Consumer writes the event id of the *start_event* in the wait table of EVM 2 (2), and checks the corresponding notification flag (initially this flag is false) until it becomes true. When task Producer notifies the event *start_event* it writes the event id in the FIFO of EVM 2. The device EVM2 is always looking for some write into the FIFO, and when something is written, it compares the id of the FIFO with the id in the wait table (3). If the ids are the same, EVM 2 interrupts task Consumer (4). The interrupt handling routine receives the notified event id and changes the corresponding notification flag (5).

In order to allow the platform processors to access the corresponding EVM, embedded software is generated. This is a set of C files organized in layers as depicted in Fig. 18(b)). The application is modified by including the application_event layer and functions for waiting and synchronizing events as depicted in Fig. 20. The application_event layer is a C file including instances of the event `struct` for all events that are expected or notified by the application. An event is specified in C as the `struct` Fig. 21.

The `event` layer includes functions for notifying and waiting events as well as the event data type definition. The event `struct` includes the event id field, as well as a notification flag indicating if the event has been notified or not, and two pointers: one for notification fifo and the other for the wait table.

The `processor` layer includes processor dependent code and is customized for each processor in the platform (data type and processor endianness). This layer includes also interruption handling functions and initialization code.

**Fig. 20** SystemC producer code and the generated C code

```
1  void producer_process()
2  {sc_int<32> j = 0;
3   while (1)
4   { wait(start_event);
5     for (j = 0; j < 100; j++)
6       {CH0->write(j);
7       }
8   }
9  }
10
11 #include "processor.h"
12 #include "application_channels.h"
13 #include "application_events.h"
14 #include "stdio.h"
15
16 int main(int argc, char *argv[])
17 { INT32 j = 0;
18   disable_interrupts();
19   init_interrupts();
20   init_events();
21   init_channels();
22
23   wait_event (start_event);
24   while (1)
25    { for (j = 0; j < 100; j++)
26        ch0_write(j);
27    }
28
29   return 0;
30 }
```

**Fig. 21** Event structure

```
1
2 struct event {
3   char notification_flag;
4   int id;
5   int *wait_address;
6   int *notification_address;
7 };
```

### *6.2.2 Implementing process communication*

The implementation of communication is very similar to the synchronization implementation since each communication includes several read and write events. Our model implements process communication by using a shared memory. In the case of channel communication, the designer assigns the communication channel to an address range of the shared memory.

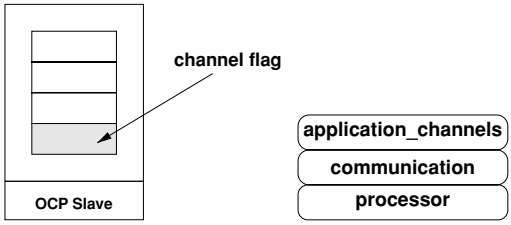



**Fig. 22** Event manager device and software layers

(a) CRL device (b) communication sw layers

```
1  struct fifo_INT32_channel_data {
2    INT32 num_free;
3    INT32 size;
4    INT32 p_in, p_out;
5    INT32 _data[16];
6  };
7
8  struct fifo_INT32_channel {
9    struct fifo_INT32_channel_data *_data;
10   struct event *data_read_event;
11   struct event *data_written_event;
12   char *lock;
13 };
```

**Fig. 23** C communication structs

A SystemC module called *critical region locker* (CRL) is inserted into the platform in order to control the concurrent access to the shared memory. Embedded code to access CRL is also automatically generated and used by the application code running on the processor side.

When implementing blocking communication through FIFOs it is necessary to guarantee synchronization in the read and write operations. The related events are automatically mapped to the bus the shared memory is connected. Beyond that, the data part of the channel is assigned to the shared memory. In this case, it is assigned to the smallest free address range in the memory that holds the channel's data.

The CRL device is very simple and has a one byte flag for each channel. When the CRL is written it works like a register and holds the value written. After being read it sets the corresponding value to $-1$, indicating that the channel is locked. The generated embedded software is also organized in layers. The CRL device and software layers are depicted in Fig. 22. C code for communication layer can be seen in Fig. 23.

## 7 A Framework for platform modeling and communication analysis

All mentioned tools have been integrated in the *Platform Designer* Framework as plugins in the *Eclipse* [18] framework. The advantages of this approach is that users have an unified

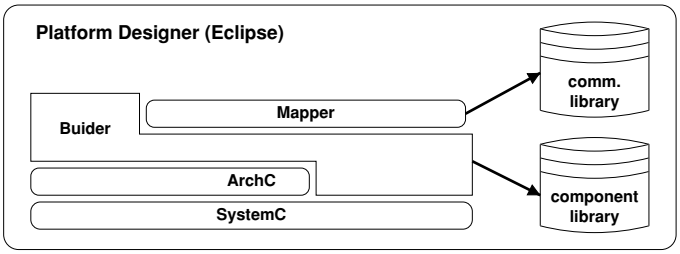



**Fig. 24** Platform designer framework architecture

environment with the Eclipse facilities such as a friendly interface and project and files handling resources.

Figure 24 depicts the *Platform Designer* framework architecture. Two Eclipse plugins have been developed the Builder and the Mapper plugins. The Eclipse environment includes the SystemC simulator, the ArchC and mapping tools, as well as the library of components.

On the bottom of the framework architecture is the SystemC simulator. On top of this level are the ArchC and PArchC tools for generating the SystemC platform model.

The Builder plugin provides graphical support for describing multiprocessor platforms. By connecting ArchC processor models, devices, and buses stored in the component library, the designer can build the multiprocessor platform. The Builder plugin also allows the designer to set component parameters and to define memory address mapping for devices in the bus. Using the Builder plugin the designer can also generate the platform model in ArchC compatible code, compile it using gcc and also execute some application by running the simulator.

The Mapper plugin provides support for application modeling and its mapping into the platform by allocating processes to processors and channels to bus. Using the Mapper the designer can also load executable code to run in the platform processors.

Figure 25 depicts a screen shot of the Platform Designer framework. The reader can see the Builder and Mapper menus on the menu bar of the Eclipse. The outline of the platform can be seen on the bottom of the figure. It shows the components the jpeg/adpcm example with the four processors and the bus.

A SystemC platform model can be seen in the editor view of the framework. The editor can also be used to visualize the generated SystemC code. On the left hand side is the project management area for the management of ArchC, PArchC, SystemC and C files.

## 8 Experimental results

In order to show the effectiveness of our approach for synchronization and communication synthesis, we have specified a platform with four processors (SPARCV8), and have run four applications from the *Mibench* benchmark [14].

The system behavior can be seen in Fig. 26. It consists of the jpeg coder, which reads an image file in ppm format, compress it and transfers the contents of the file to the jpeg decoder running on other processor, using a fifo channel. Besides image processing, the application also includes encoding/decoding of voice data on the ADPCM modules. It also uses a fifo channel for transmitting the encoded voice to the process running on other processor.

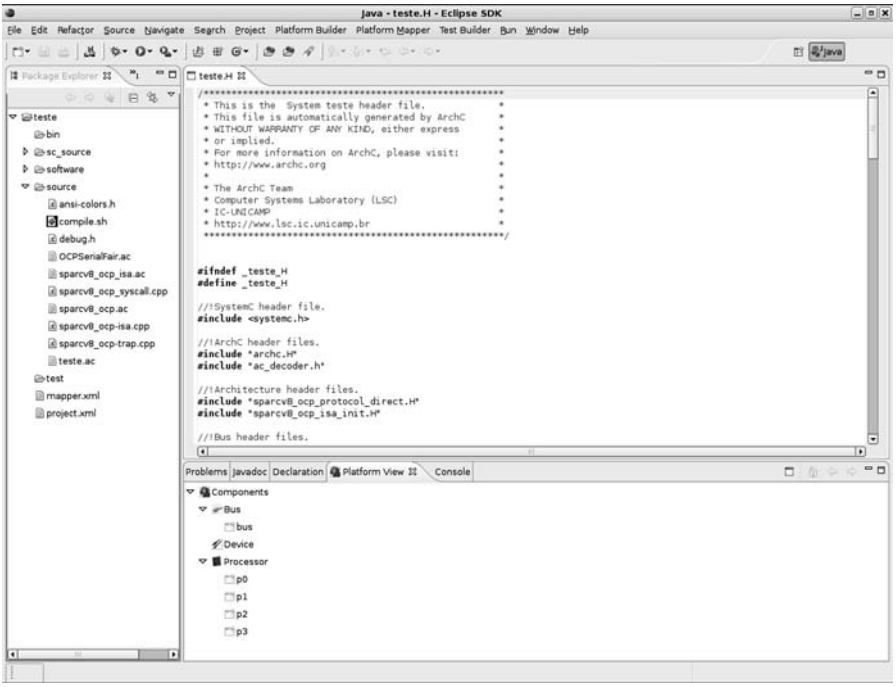



**Fig. 25** Platform designer framework

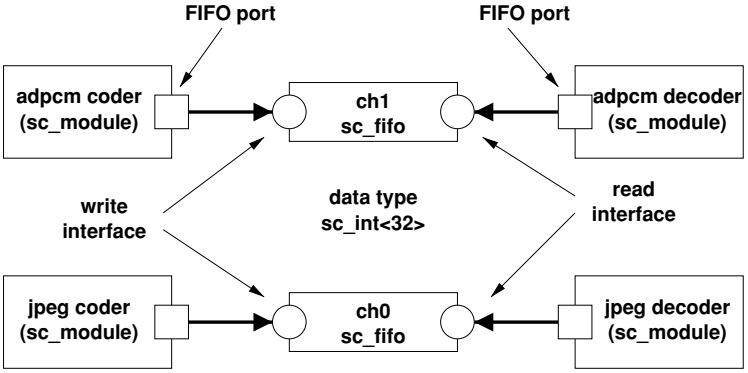



**Fig. 26** jpeg and adpcm coder/decoder application

The mentioned application will be executed on the platform shown in Fig. 27. It has been modeled and compiled using the extensions to the ArchC language and the *acsys* tool.

Table 2 shows the effort in number of lines for describing the mentioned multiprocessor platform taking also into account the effort to describe the SPARCV8 processor. As it can be seen, most of the effort concentrates in the processor description (1656 lines of ArchC and SystemC code). The reader must also notice that the processor can be reused in other platforms. The design effort for a processor like the SPARCV8 took 1 month, including the

**Table 2** Coding effort and generated code

| | Input code | Output code | obj code (bytes) |
|---|---|---|---|
| Processor | 1,656 | 11,128 | |
| Platform | 59 | 158 | |
| Syncronization management | – | 108 | 7760 |
| Interrupt handler | – | 17 | – |
| Processor initialization | – | 63 | 3136 |
| Communication management | – | 277 | 13988 |

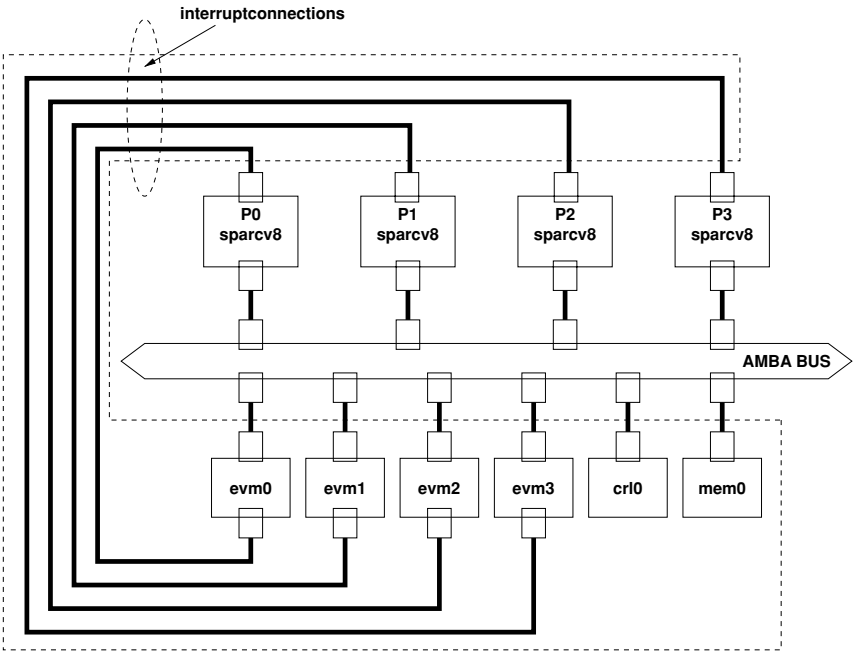



**Fig. 27** jpeg and adpcm coder/decoder platform

time to study its specification and to test the instructions. It should be clear that the design effort must no be confused with the coding effort (1656 lines of code).

The tool has automatically generated 11128 lines of SystemC code for the processor, plus 158 lines for the platform, releasing the designer from a hard and error prone task. Beyond that, the tool has also generated 359 lines of C code to implement the synchronization management, interrupt routine, processor initialization and communication between the application tasks running on the platform. Most of the code, 277 lines, is due to the implementation of the SystemC sc_fifo behavior. The compilation of these files with the sparc-elf-gcc, a gcc based cross-compiler for the sparc architecture, resulted in 21,748 bytes of object.

The compilation time for a platform description takes less than a second in a Pentium IV machine.

The application mapping the mentioned platform can be seen in Table 3. The jpeg coder runs on processor P0, and jpeg decoder on processor P1. The adpcm coder and decoder run

**Table 3** jpeg and adpcm mapping

| app component | Platform component |
|---|---|
| jpeg coder | P0 |
| jpeg decoder | P1 |
| adpcm coder | P2 |
| adpcm decoder | P3 |
| ch0 | AMBA BUS |
| ch1 | AMBA BUS |

on processors P2 and P3, respectively. The fifo channels, ch0 and ch1, have been mapped to the AMBA-AHB bus.

When the mapping is done, components are added in the platform for synchronization and communication purposes. This components are inside the dashed line. Four event managers (EVM0, EVM1,EVM2 and EVM3), one critical region locker (CRL0) and one shared memory (MEM0) devices have been created. Four interrupt connections between the event manager devices and the processors have also been added to the platform.

The multimedia application has been run in two models of the same platform: the first one includes a functional model of the AMBA bus, and the second one includes a cycle accurate model of the same bus.

The results of both platform models are presented in Tables 4 and 5, respectively. These results have been obtained by using a Pentium IV machine with 256 Mb of RAM under the linux Ubuntu 5.10 distribution.

Table 4 shows the results for the application running on a platform with a functional version of the AMBA-AHB bus. The simulation takes 959,88 sec., with the combined processors performance of 355 Kinstr/s and a simulation frequency of 184 Kcycles/sec. The instruction execution rate is compatible with related works, and the simulation performance seems to be good enough to allow the designer analyse distinct platforms.

**Table 4** Functional simulation results

| | | |
|---|---|---|
| P1 | 3,236,885 | ch1 |
| P2 | 13,835,574 | ch1 |
| P3 | 147,146,851 | ch0 |
| P4 | 176,811,467 | ch0 |
| Total | 341,030,777 | |
| Simulation time | 959.88 s | |
| Processors performance | 355 Kinstr/s | |
| Simulation frequency | 184 Kcycles/s | |

**Table 5** Simulation results

| | | |
|---|---|---|
| P1 | 28,815,995 | ch1 |
| P2 | 39,450,656 | ch1 |
| P3 | 160,489,928 | ch0 |
| P4 | 191,517,699 | ch0 |
| Total | 420,214,278 | |
| Simulation time | 1,912 s | |
| Processors performance | 219.83 Kinstr/s | |
| Simulation frequency | 105 Kcycles/s | |

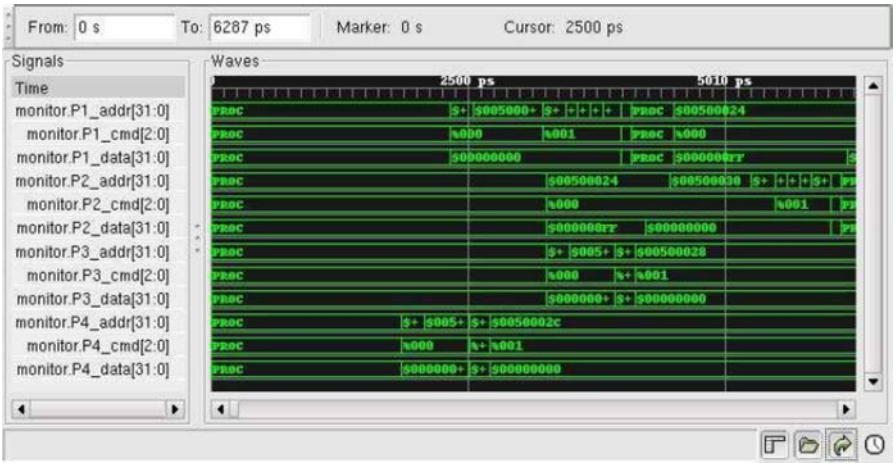



**Fig. 28** Communication analysis

Simulation results using a cycle accurate bus model are given in Table 5. Processor P0 executes 28,815,995 instructions to compress a 196,666 bytes ppm image into a 9,810 bytes compressed jpeg image. It transfers the image file to the decoder module running in P1 through channel ch0. P1 takes 39,450,656 instructions to receive the compressed image and decompress it to the original ppm format.

The audio application running on the platform takes considerably longer since the input audio file (pcm format) is bigger (1,368,864 bytes). Processor P2 executes 160,489,928 instructions to compress it to the adpcm format and to send it through channel ch1. The adpcm decoder modules takes 191,517,699 to receive the compressed data and convert it to the original pcm format.

By taking a bus accurate AMBA bus model the simulation time is approximately 1,912 s, two times slower than the previous situation. In this case, we have an instruction execution rate of 219 Kinstr/s or 105 Kcycles/s. Also here, the performance results show the suitability of the platform simulator for design space exploration.

In order to analyze the communication between the processors in the platform it has been inserted monitors in the bus. These devices monitor the read and write addresses and associate these addresses to the sc_fifo channels used in the communication. Addresses that are not related to communication are marked as *PROC*. Each three lines of Fig. 28 show the address, command (read or write) and the data being transferred. In this case the data can be the sc_fifo channel data or control data like the pointers to the buffers the implement the sc_fifo in memory.

The first three lines refer to channel ch0 used by processor 1. It shows that the transfer of 1 32 bit word using the channel takes approximately 1800 PS that gives 18 clock cycles with a simulation clock period of 100 ps. As can be seen the in the figure the transfer times vary a lot from transfer to transfer this is due to the assynchronous nature of the sc_fifo channel. When it is full an event must be notified and acknowledge to complete the transfer.

Our approach results in a simulation performance, which is compatible with existing approaches. Even in the case of using a cycle accurate bus model, the simulation can be done in a reasonable amount of time and the designer has detailed information on the system to tune the platform for the application.

This simulation performance allows the designers to have a good feedback of the application in a reasonable amount of time, supporting a more effective design space exploration.

## 9 Conclusions

This paper presents a processor centric approach for the modeling and simulation of multiprocessor platforms. The ArchC ADL has been extended to support the modeling of multiprocessor systems at a very high abstraction level. The results shown that the modeling effort is minimum, since processors and platforms are modelled in an unified environment.

Besides modeling support our approach allows the automatic generation of platform simulators in SystemC at distinct abstraction levels. The great advantage of this feature is that the designer does not need to make any change on the platform description to obtain simulators. It hides all the simulation scheme details from the designer by just issuing command line options or clicking the corresponding buttons in a graphical framework. Simulation results have also shown the impact of RTL simulation performance, and on finding errors in the platform.

Our framework includes also a mechanism for generating platform simulators, which are able to run concurrent processes communicating through channels. For this purpose, a mechanism has been developed which supports the semi-automatic synthesis of system-level synchronization and communication into the multiprocessor target platform. One advantage of our approach is its ability to explore distinct platforms with a minor effort. Embedded C code and SystemC modules are generated to implement communication and synchronization among processes running on distinct processors.

In order to improve the functionality of the framework we have work going on to provide component models like buses with analysis support like performance and power consumption. In the case of performance the idea is to obtain detailed information regarding contention and transfer times. We are also including performance and power cache analysis support that will give the system designer more information during the design space exploration of the platforms.

## References

1. AMBA specification Rev(2.0).
2. Abdi, S., D. Shin, and D. Gajski. Automatic Communication Refinement for System Level Design. In *Proceedings of the 40th conference on Design automation*, ACM Press, 2003, pp. 300–305.
3. Araujo, C. and E. Barros. Communication Mapping in Multiprocessor Platforms. In *Proceedings of the IFIP International Conference on Very Large Scale Integration of System-on-Chip*, 2005.
4. Araujo, C., E. Barros, R. Azevedo, and G. Araujo. Processor Centric Specification and Modeling of MPSoCs Using ArchC. In *Proceedings of the Forum on Specification & Design Languages FDL'05*, 2005.
5. Azevedo, R., S. Rigo, M. Bartholomeu, G. Araujo, C. Araujo, and E. Barros. The ArchC Architecture Description Language. *International Journal of Parallel Programming*, 33(5):453–484, 2005.
6. Balarin, F., Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An Integrated Electronic System Design Environment. *Computer*, 36(4):45–52, 2003.
7. Brunel, J.-Y., W.M. Kruijtzer, H.J.H.N. Kenter, F. PÈtrot, L. Pasquier, E.A. de Kock, and W.J.M. Smits. COSY Communication IP's. In *Proceedings of the 37th Conference on Design Automation*, ACM Press, 2000, pp 406–409.
8. Cesario, W., A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A.A. Jerraya, and M. Diaz-Nava. Component-based Design Approach for Multicore SoCs. In *Proceedings of the 39th Conference on Design Automation*, ACM Press, 2002, pp. 789–794.
9. Cesário, WO., L. Gauthier, D. Lyonnard, G. Nicolescu, and A.A. Jerraya. Object-based Hardware/Software Component Interconnection Model for Interface design in System-on-a-Chip Circuits. *Journal of Systems and Software*, 70(3):229–244, 2004.

10. Coware company. available at http://www.coware.com, February 2006.
11. de Kock, E.A., W.J.M. Smits, P.van der. Wolf, J.-Y. Brunel, W.M. Kruijtzer, P. Lieverse, KA. Vissers, and G. Essink. YAPI: application modeling for Signal Processing Systems. In *Proceedings of the 37th Conference on Design Automation*, ACM Press 2000, pp. 402–405.
12. Dziri, M.-A., WO. Cesário, F.R. Wagner, and A.A. Jerraya. Unified Component Integration Flow for Multi-Processor SoC Design and Validation. In *Proceedings of DATE*, 2004, pp 1132–1137.
13. Gajski, D. et al. System-on-Chip Environment (SCE) Tutorial. Technical report, Center for Embedded Computer Systems Information and Computer Science, University of California, Irvine, September 2002.
14. Guthaus, M.R., J.S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge, and R.B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, December 2001.
15. Halambi, A., P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: a Language for Architecture Exploration Through Compiler/Simulator Retargetability. In *Proceedings of the Conference on Design, Automation and Test in Europe*, ACM Press, 1999, page 100.
16. Hoffmann, A., T. Kogel, A. Noah, G. Braun, O. Schliebush, O. Wahlen, A. Wieferink, and H. Meyer. A Novel Methodology for the Design of Application Specific Instruction Set Processors (ASIP) Using a Machine Description Language. In *IEEE Transactions on Computer-Aided-Design*, November 2001, pp. 1338–1354.
17. http://www.archc.org. The ArchC Resource Center.
18. http://www.eclipse.org. Eclipse Open Source Community.
19. http://www.tensilica.com. Tensilica Company.
20. Kahn, G. The Semantics of a Simple Language for Parallel Programming. In *Proceedings of the IFIP Congress 74*, North Holland Publishing Co., 1974, pp. 471–475.
21. Lee, C., M. Potkonjak, and WH. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proceedings of the 30th Annual International Symposium on Microarchitecture(Micro-30)*, December 1997.
22. Lieverse, P., T. Stefanov, PV. Wolf, and Ed. Deprettere. System Level Design with Spade: an M-JPEG Case Study. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, IEEE Press, 2001, pp. 31–38.
23. Eugenia Gabriela Nuta Nicolescu. *SpÈcification et validation des systËmes hÈtÈrogËnes embarquÈs*. PhD thesis, TIMA Laboratory, November 2002.
24. Paulin, P.G., C. Pilkington, and E. Bensoudane. StepNP: A System-Level Exploration Platform for Network Processors. *IEEE Design & Test of Computers*, 2002, pp. 17–26.
25. Paulin, P.G., C. Pilkington, E. Bensoudane, M. Langevin, and D. Lyonnard. Application of a Multi-Processor SoC Platform to High-Speed Packet Forwarding. In *Proceedings of DATE*, pp. 58–63, 2004.
26. Shin, D., S. Abdi, and D. Gajski. Automatic Generation of Bus Functional Models from Transaction Level Models. In *Proceedings of ASP-DAC*, 2004, pp. 756–758.
27. SPARC International, Inc. *The SPARC Architecture Manual - Revision SAV080SI9308*.
28. The ArchC Team. *The ArchC Architecture Description Language Reference Manual*. Computer Systems Laboratory (LSC) - Institute of Computing, University of Campinas, http://www.archc.org, 2006.
29. Vahid, F., and T. Givargis. Platform Tuning for Embedded Systems Design. *IEEE Computer*, 34(3):112 – 114, 2001.
30. Wieferink, A., T. Kogel, R. Leupers, G. Ascheid, H. Meyr, G. Braun, and A. Nohl. A System Level Processor/Communication Co-Exploration Methodology for Multi-Processor System-on-Chip Platform. In *Proceedings of DATE*, 2004, pp. 1256–1263.
31. Wolf, W. The Future of Multiprocessor Systems-on-Chips. In *Proceedings of the 41st Annual Conference on Design Automation*, ACM Press, New York, NY, USA, 2004, pp. 681–685.