ISSN 1392 – 124X INFORMACINĖS TECHNOLOGIJOS IR VALDYMAS, 2004, Nr. 1(30)

COMMUNICATION CO-PROCESSOR DESIGN BY COMPOSITION OF PARAMETERIZED CELLS

Giedrius Majauskas, Vytautas Štuikys

Kaunas University of Technology
Studentų st. 50-324, LT – 3031, Kaunas, Lithuania

Damien Lyonnard, Wander O. Cesàrio, Yannick Paviot, Ahmed A. Jerraya

TIMA, France

Lovic Gauthier

Kyushu, Japan

Abstract. We deal with hardware block interconnection in Systems-on-Chip. The cost of writing the glue code grows together with the complexity of such systems. To write such code manually is time consuming. We present a method of communication co-processor generation for multi-processor SoCs. The method is based on composition of the parameterized library cells. The cells are parameterized using external macro language. The parameterization and decomposition of the CC allows decreasing the size of the library, increases code reuse and testability of the components without loss of performance and flexibility. We present a VDSL application as a case study for our approach.

1. Indroduction

1.1. Context

The ITRS road-map predicts that by 2004, 70% of ASICs will be Systems-on-Chip (SoCs) and include at least one embedded instruction-set processor [8]. Many applications already in the market include several processors with different instruction-sets: mobile terminals, set-top boxes, game processors and network processors [7, 11, 15]. These mass-market products will be integrated on a single chip for production cost reasons. It is expected that these applications will act as the main drivers for the semiconductor industry. Most system and semiconductor houses develop IP platforms, which allow the integration of several cores (CPU, DSP, MCU, coprocessors and other IP's) and sophisticated communication networks (hierarchical bus, TDMA-based bus, point-to-point connections and packet-routing switches) on a single chip. The trend is to build large designs using an on-chip network by interconnection standard components.

SoCs will include many different instructions-set processors executing dedicated functions in order to increase the flexibility of the whole system. Complex, on-chip, HW/SW communications interfaces are required to implement these SoCs: multi-core architectures may require an application-specific communication network interconnect. When systems are integrated on a single chip, the hardware (microprocessor interfaces, bank of registers, memories) and software (drivers, operating systems) parts require communication protocols, which need to be adapted according to the type of core [8].

This integration requires a large number of different HW communication protocols. In order to minimize the number of communication components, we focus on a solution based on the usage of communication co-processors (CC's). Each CC is application-specific and depends on the component, the communication protocol used, and other design requirements. It has to be adapted to each new design and can not be used as-is. Due to the variety of requirements for CCs (performance, verifiability requirements, specific needs of a designer), the implementation may require a huge library. Using our approach, we can generate a particular CC for a specific application automatically, thus reducing the complexity of the library.

1.2. Related Work

In this subsection, we will review and analyze approaches for the automatic creation of CCs. CCs

and communication networks are necessary to connect CPU's, DSP's and other IP blocks because of large number of different communication protocols. These are also called wrappers, communication coprocessors and adapters. Their main role is to adapt the component to the rest of the system. The manual development of such components is discussed in [6], however coding is time consuming. Therefore, we need to generate these CCs automatically.

The existing approaches to generate CCs can be divided into two groups. The first one relies on generating custom CCs, the second one relies on composing them from basic cells. Both methods have their strengths and weaknesses.

In generation-based approaches, the CCs are created from a formal descriptions using chosen model. Usually the specification is converted into 2 or 3 FSMs described in some kind of HDL. The examples of such generators are Synopsys Protocol Compiler[19], PIG [17], POLARIS [18], ProGram[14]. UIC [12] tool generates the customized bus for the data transfer. Coral [1] connects Core-Connect compliant IP's through predefined busses from virtual specification. For third-part components, a CC has to be written manually. For all the above mentioned approaches, the interface is generally limited to a single communication protocol. This can generate area overhead when connecting an IP block with its own predefined communication protocol. The generator is application-domain dependent. Specification writing can be time consuming too, because the language is generally specific to communication protocols and may use esoteric concepts.

In library-based approaches, the CCs is created by finding and (optionally) composing predefined library cells[9, 10, 2, 20]. The user is able to extend the library. However, it is hard to cover all possible protocols without exploding the size of the library. In [20] components are connected together using rendez-vous (handshake) protocol with CCs from the library. COSY [2] system uses layered communication model for the HW/SW communication refinement. The disadvantage of these approaches is a limited set of the protocols supported. Although library-based approaches provide better flexibility for designers, they are harder to maintain. The main problem is that there are many possible protocols and it is hard to find the proper library component. Components often have to be adapted to new protocols, even if they share a significant part of the code. Whole component has to be validated after each modification again.

Our approach makes use of a powerful parametrization approach to solve these problems. The coprocessor is assembled from parameterized cells that are kept in the library. However, different cells have a significant amount of the similar code that is not reused between these components.

1.3. Contribution

Our contribution is a macro-generation based method for generating communication co-processors. This method allows us to reduce the number of elements required in HW component library without loss of flexibility and performance. It can be used together with any HW description language.

The structure of the paper is as follows. Section 2 presents the design flow. In section 3 we explain the co-processor generation flow. In section 4 we present an VDSL application as a case study and evaluate the results. Then we conclude.

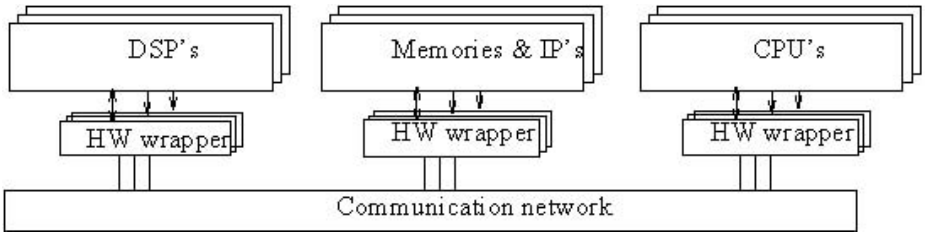

Figure 1. The Architecture of SoC

2. Design Flow

2.1. Target Architecture for SoC

In this subsection we present a generic architecture model for multi-processor SoC's. We explain the purpose of the modules, focusing on communication co-processors. This model is used in several approaches [3, 20, 10].

The main difference between classical multiprocessor architectures [5] and multiprocessor SoC architectures is that the multiprocessor SoC architectures have specific application domains while the classical architectures have general purposes. In multiprocessor SoC architectures application-specific optimization of the architecture is necessary, since the specific applications have tight design constraints (e.g. low area and power consumption and high performance). Thus, we have to use various kinds of processors (to use a processor specific to the application, e.g. usage of a DSP for voice processing). The communication networks have to be application-specific to locally meet the requirements (e.g., circuit switch network in multimedia applications), too. The

use of the existing components for communication networks and processors requires hardware and software adaptation. That also needs to be application-specific.

Figure 1 shows a generic multi-core SoC architecture where processors are connected to communication networks via CCs. In fact, IP blocks and processors are separated from the physical communication network by co-processors. Such a separation is necessary to free the processors from communication management and it enables parallel execution of computation tasks and several communication protocols.

2.2. Automated Design Flow

In this subsection we will discuss a typical automatic design flow of SoCs. We will show the similarities between the tools that use automatic architecture refinement.

Figure 2 presents the generic design flow. The design flow starts with a high level specification that captures the global organization and functionality of the application. The functionality of application is validated by simulation or formal methods. The validated specification is used for generating the code for communication co-processors. For this, the communication parameters are extracted from the specification. Appropriate library cells are instantiated using these parameters. The computational blocks are refined to match implementation requirements. These modifications are beyond the scope of this paper. The specification is modified by the tools to reflect the changes. The co-processors are connected by channels and shared busses. The modified specification is validated using RTL simulation tools.

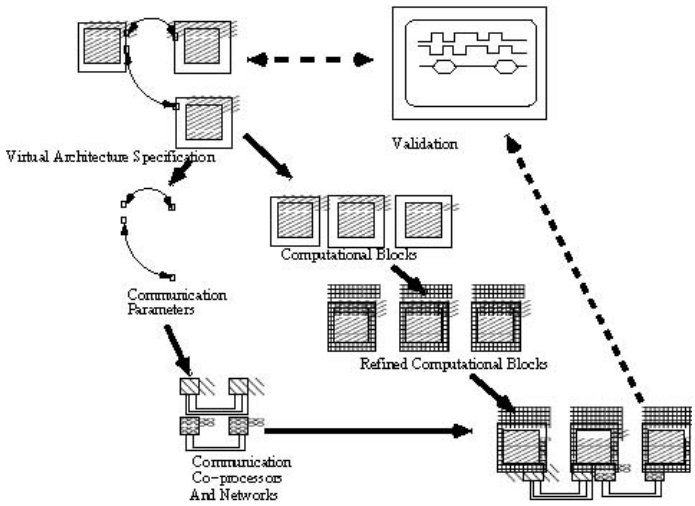

Figure 2. Generic design flow

A high level specification should allow the designer to re-map the design easily into a new configuration. This may be done using architecture exploration tools such as VCC[3], or some specification languages like SystemC[16]. Typically, specification has very little implementation details. It consists of hardware modules that have abstract ports. The ports are connected through abstract channels. Their implementations are not defined at this level. For convenience, we call this specification a virtual architecture specification.

The validated design is refined using mapping parameters. Suitable library cells are picked and instantiated using these parameters. The co-processors and processor architectures are built using these instantiated cells. The specification is automatically modified to reflect these changes. Then the design is validated using RTL simulation tools.

2.3. The Structure and Generation of Communication Coprocessors

In this subsection, we present a general architecture of communication co-processor and describe the responsibilities for each component of the co-processor.

Figure 3 shows a generic processor-centric architecture structure. The CC is organized as a bridge between the microprocessor and a communication network that provides several access points. This scheme is used in several works in literature (eg. [3, 10]) for increasing scalability and flexibility of the systems as it allows the separation of the communication network from the computational modules. We will use a similar scheme.

The typical processor-centric architecture consists of several smaller modules. Firstly, it translates processors data to some intermediate protocol. Secondly, it provides some additional hardware to adapt the processor, like address decoder, interrupt handler, etc. Thirdly, it provides several instances of

communication-specific hardware, that transforms the data from the intermediate protocol to the channel-specific format. These modules are called channel adapters(CAs).

Most of the processor-centric architecture has to be adapted for each kind of processors. As the processors can differ greatly, these parts are implemented as separate library cells and are insignificantly parameterized. The most parameterizable parts of the co-processor are CAs. This is because of great similarity between communication protocols used in the SoCs today.

The CA implements (1) the communication protocol of the virtual-architecture level channel and (2) the protocol of the connected communication network at the micro-architecture level. For each channel in the virtual architecture level specification, several CAs are instantiated from the protocol library with the architecture parameters (e.g. input/output, master/slave, data type, buffer size, interrupt usage). In this paper we focus on CAs parameterization as they have to be reusable

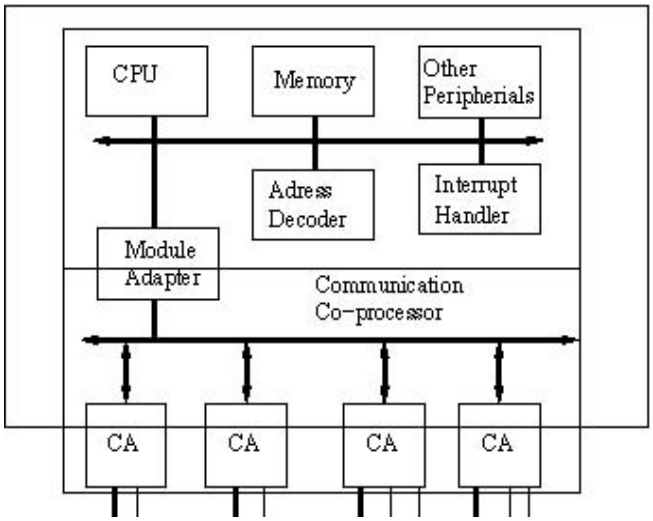

Figure 3. The processor -centric architecture in different environments

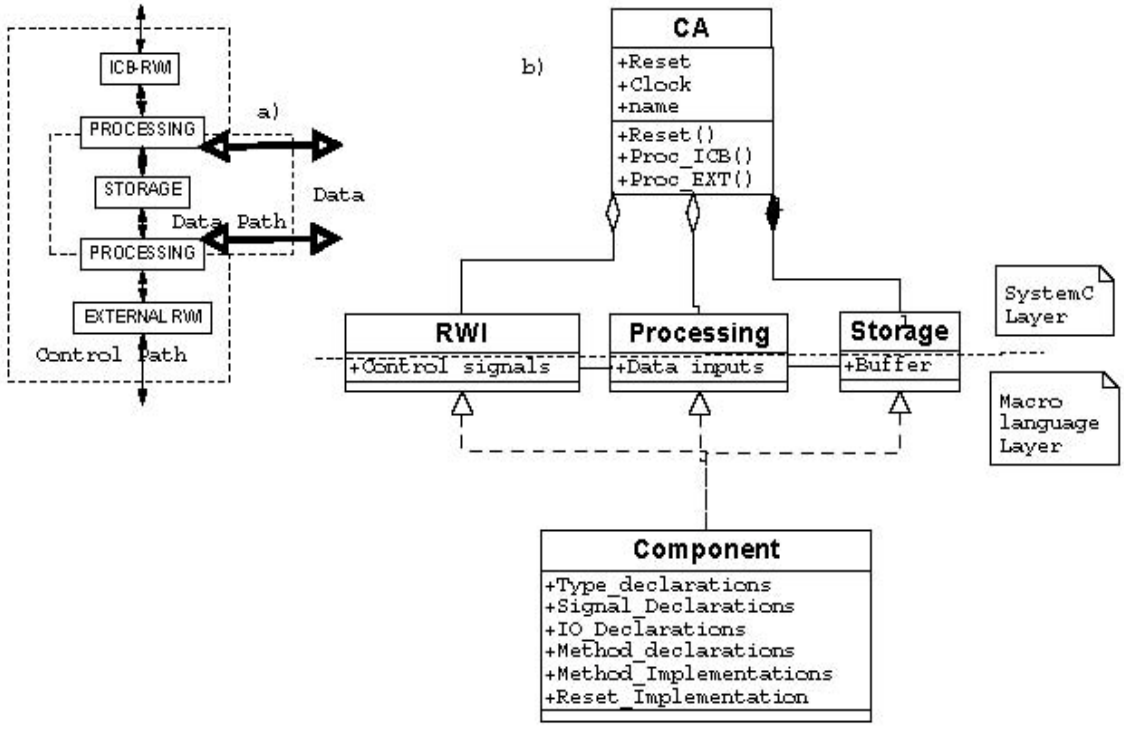

Figure 4. Structural model of the channel adaptor a) Control/Data flow b) Simplified class diagram

3. Communication Co-Processor Generation

3.1. The Generic Channel Adapter Structure

In this subsection, we show a way to decompose a CA into smaller functional blocks. This decomposition is required to reduce the complexity of hardware library.

Although CAs can vary greatly, there are some similarities between them. Firstly, CAs have 2 interfaces. They read and write to a common storage. These interfaces are usually not exactly the same. Firstly, the directions are different. Secondly, synchronization method is not the same.

We can decompose each CA into two interfaces, connected by a storage block (Fig. 4 a). One interface connects CA to a local processors (internal) communication bus, another interface connect with CAs of other modules. Each interface consists of read-write synchronization and processing parts. Protocols, available for internal communication bus side of CA are a subset of the ones used for external communication. Thus we have to describe only three types of blocks in the library (Figure 4b). We have to use an external macro language for synthesizable implementation of the generic CA.

3.2. Basic Cells

In this subsection, we present the tasks for each type of the cells.We clearly separate the responsibilities of each type of the cells from others. This is required for declaring an interface between these blocks, which will be discussed later in the next subsection.

The read-write interface is responsible for the synchronization, activation and processing task calling. The implementation can vary from simple polling access to different handshake or shared bus specific protocols. It is a control block and does not process data itself.

The processing block is responsible for incoming data preparation for storage or for transforming stored data to output format. It can split or join long bit vectors, or for example, parity check. The block does not produce any new data itself. The execution of this block can take from zero to several clock cycles, so the block should inform other read-write interfaces about its state after each operation.

The storage block is responsible for storing and retrieving the data. It manages the storage buffer itself. The block provides some information to other blocks about the status of the buffer. It acts as a joining part between internal and external interfaces.

3.3. Parameterizable Structure

In this subsection, we will present a way cells are composed together to obtain a customized CA. We will show how macro language can be used for code generation.

A simple example of macro language is shown in Figure 5. This example illustrates how the macro language can be used for code generation. It consists of parts of top-level netlist and storage cell. Top level cell passes the parameters to other cells.

Each block is parameterized using a macro language. There are 4 types of parameters : global, storage specific and specific to each read write interface and processing block on each side of the CA. The separation of these parameters helps to reduce the complexity of the components and to separate the tasks from each other. The global parameters are used to guarantee the coherency of layers.

Part of top level netlist

```
DEFINE {OPTS} IMPLEMENT_COMPONENT=
….
IMPLEMENT_RESET{OPTS}
STORAGE_MPLEMENT_METHODS
{OPTS[GLOBAL].OPTS[STOR].storage,0}
….
ENDDEFINE
```

Part of storage implementation

```
DEFINE {OPT, SUF, NO}
STORAGE_IMPLEMENTATION =
DEFINE TP=OP{OPT,"STYPE"} ENDDEFINE
IF (TP==T_FIFO) DO
"void_"OP{OPT,["entity_name"}"::m_storageWrite(t_storage data) {
if (fifo_full!=OP{OPT,"SNEG"}") return;
else {
fifo_empty="OP{OPT,"SNEG"}";t_storage_counters ttail=tail;
data_storage[ ttail]=data; …."
ELSE IF (TP=T_REG) DO …
```

Figure 5. Part of processor architecture described in macro language

We have to define two kinds of interface between the blocks for the macro-language and for the target language. The first interface consists of names of the macro definitions and is used to compose the text blocks into complete files. The second one enables the communication between these composed blocks.

This allows higher flexibility of the library, as the cells are independent from each other.

The cells are composed together by top-level macro files. The description instantiates 2 read-write and processing cells and one storage cell. The result is two files for each CA: SystemC module declaration and SystemC method implementation.

The main differences from other library-based approaches are the following. We use smaller cells to build the coprocessor. Thus we need less library cells. Theoretically we can generate $C=RWI^2*Proc^2*Stor$ different CA's with different instances of library cells. RWI, Proc, Stor are amounts of instances available from read write interface, processing and storage cells, respectively. Thus, the library is able to produce much more CA's from the same amount of library cells that

in other library-based approaches. It is easy to add new component as we have defined an interface cells have to provide. We can achieve higher reusability of the code, because we have to write only new parts of the CA and not to rewrite parts existing in other components.

4. Application

4.1. Design of VDSL Application

In this subsection we present a design of a VDSL application[13]. We explain application architecture and specification. We present the parameters used for the refinement of the application.

Figure 6 gives a graphical representation of the virtual architecture model that captures a subset of the VDSL modem specification [13]. The virtual architecture represents a system as a hierarchical network of modules. Each module consists of an internal behavior and ports. The modules communicate with each other through channels connected to their ports. Modules VM1 and VM2 correspond two CPU's and module VM3 represents the TX-Framer block handled as an IP(For the TX-Framer block, only the interface is known so it is represented as a black-box).

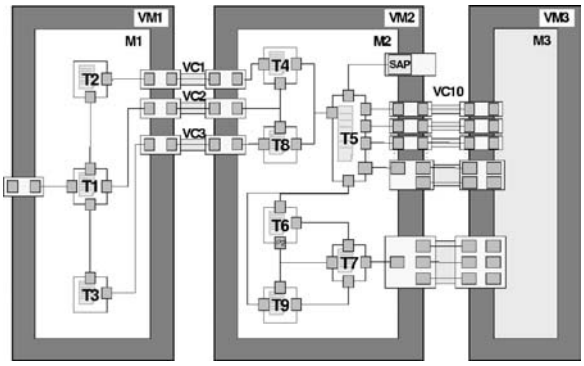

Figure 6. Virtual architecture description of the VDSL modem

The specification shown in Figure 6 describes only virtual architecture of the application. This specification could be mapped onto different architectures depending upon the configuration parameters annotated in modules, ports, and nets.

For instance, the three point-to-point connections (VC1, VC2, and VC3) used in the communication between VM1 and VM2 can be mapped onto a bus or onto a shared memory if the designer changes the configuration parameters placed on these virtual channels and/or virtual ports.

Each module, task, port and net has specific parameters annotated in SystemC specification. For example, we have chosen to implement the modules M1 and M2 as 2 ARM7 CPUs and add this parameter to the modules in the top-level specification. The user can set task priority and the files that store the description of its behaviour. For a port, there are a set of attributes to configure the operating system services that a task needs, the type of data transmitted, the set of addresses needed and other parameters. The net parameters are similar to ones of ports.

Each port has HW and SW specific parameters. The SW parameters define how the channel can be accessed through the port. This group of parameters includes the address of the memory the CA can be accessed through, SW driver used, etc. The HW parameters define how the channel is implemented in HW. For the HW important parameters are the protocol CA should communicate, data width of both internal and external information, mask parameters for transmission of status information and storage type.

4.2. Macro Generation of Communication Co-processors

VDSL application is a good example for showing the diversity of CAs. Each module has to communicate with other modules through several kinds of protocols. In this subsection, we will show how macro generation technique was used for CA generation for the VDSL modem application. Lastly, we will present parameter sets for two particular channels in the design.

The parameters for CAs can vary greatly. There are two main reasons for protocol diversity and library size explosion. The first one is that each module can require different communication protocols. For example, the IP block in VDSL application has some predefined communication protocols. And, depending on OS implementation, there can be some restrictions for other communication protocols. The second one is that we have to customize the implementations of CAs to meet the constraints of the design. The implementations of CAs can vary in area, latency, delay and quality of service. For example, the designer may change the buffer size for each of the protocols to explore the possible implementations.

For example, virtual port VP2 connecting module VM1 with module VM2 requires a handshake protocol output CA, that operates with 32 bit data. It stores intermediate data in FIFO buffer which can keep 16 data blocks. Thus, the component needs : a read-write interface supporting handshake protocol, an 32 bits data input and output, FIFO storage buffer with 32 bit access and supporting 16 elements. The internal interface of CA should provide status information for module adapter (to avoid buffer overflow), that is kept in the first and second bits of the transmitted data. Another example is port VP10 of Module VM2, that connects it to the IP block. It is a simple polling output register, operating with 32 bit data, used to configure the TX Framer. It does not require synchronization, so it does not have an acknowledge signal. As it is only a register, the storage buffer does not need to provide any status information.

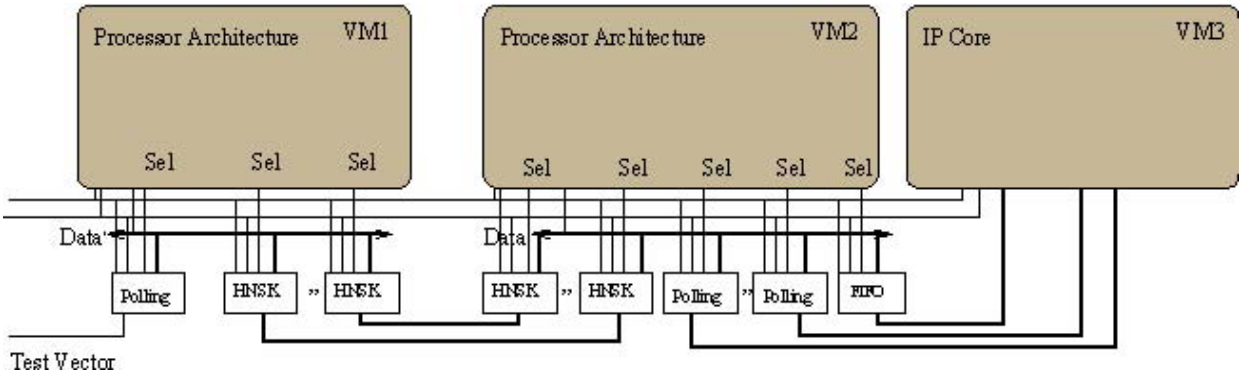

Figure 7. Refined architecture of VDSL modem

4.3. Automatic Refinement of VDSL Application

In this subsection, we present the results of the automatic refinement of VDSL application design.We will describe the resulting architecture and show two generated CAs.

The resulting architecture of the VDSL application after the refinement is shown in Figure 7. We decided to implement modules VM1 and VM2 as 2 ARM7 CPUs. The matching co-processors are built for both processors. An application-specific address decoder, an ARM7 processor adapter, a local bus and several CAs are generated for each of the communication co-processors. The netlist is modified to reflect the refined architecture. More details about the architecture refinement and Co-processor can be found in [4].

In Figure 8 we show the parts of the two generated CAs whose parameters are discussed above. The first two methods are taken from the port VP10 of module VM2 and the last three methods are taken from the port VP2 of module VM1. We present only storage output functions in this example. Although textually the method interface is the same, it can cover a large amount of different types and functionality.

```
/* Storage access methods for simple register */
T_storage channel_output_POOLING::m_storageRead(void) {
        return data_storage;};
/* Storage access methods for FIFO buffer */
t_storage channel_output_HNSK::m_storageRead(void) {
if (fifo_empty !=0) return t_storage('Z');
else {
        t_fifo_full=0;
        t_storage_counters thead = head;
        t_storage td = data_storage [thead];
        thead=thead+1;
        if ( thead== tail ) fo_empty=1;
        head=thead ; return td ;
}};
```

Figure 8. An example of macro-generated storage access methods for two CA

Table 1 shows the numbers obtained after synthesis of the coprocessors of CPU1 and CPU2 in AMS 3.20 (0.8 S m) technology. As the first processor has less communication channels, its co-processor is smaller. These results are good since they account for less than 5% of the total system surface and have a critical path that corresponds to less than 15% of the clock cycle of the 25 MHz ARM7 processor used in this case study.

Table 1. Synthesis results

Co-processor results	Num. of gates	Area	Critical path delay(ns)
CPU1	3284	8168	5.95
CPU2	3795	5100	6.16

4.4. Results

The main advantages of the automatic generation are the following. The refined VDSL architecture is application-specific. The system generation is fast. It takes only several minutes to parse the architecture of the application and generate the refined structure on Linux PC 500 MHz. The coprocessors are generated in several seconds. The generation approach allows finding and fixing errors early. A designer is able to explore different choices of implementation quickly. For example, the designer can pick different processor types to implement VM1 and VM2, or change communication protocols and their parameters. The most time consuming task is to create specification.

Comparing with the generative approach, (the channels are generated automatically from formal specification), this approach provides greater flexibility. In generative approach, the language provided capabilities limit the designer. The designer can hardly extend the generator. This is not the case in our approach. If a user needs different functionality, he can easily add new components or extend the old ones without recompiling the environment.

The library can provide the same functionality with fewer components than in other library-based approaches. The de-signer's choices result in implementation of the features the components require. In comparison with [10], the library is easier to maintain, as only one generic component is visible to the other tools. Other data are passed as the parameters of the channel adapter. The integration of new communication protocols or storage methods is easier. A designer

needs to create specific parts of the components only and not re-implement the whole component from scratch. Additionally, it is much easier to locate the errors in the component design while adding new cells in the library.

5. Conclusions

We have presented a method for creation of parameterized channel adapter by decomposing them into smaller parts of code, parametrization and composition of those using external language. This method allows effective and scalable implementation of component library for generating communication coprocessors for multiprocessor SoC's. Our approach allows easy extension of component library with new communication methods, simpler validation of the components.

We are planning to apply this method not only for coprocessors, but for memory bridges, too. A formal, language related layer validation method would be useful to ease the development of new protocols.

Another possible extension of this work is the extension of macro-language itself to ease domain-specific code generation, implementing interfaces for connecting common shared bus solutions (e.g., AMBA bus), network-on-chip implementations.

References

[1] **R.A. Bergamaschi, W.R. Lee.** Designing Systems-on-Chip Using Cores. In *Proc. of* 37*th DAC*, June 2000.

[2] **J.-Y. Brunel, W.M. Kruijzer, H.J.H.N. Kenter, F. Peron, L. Pasquier, E.A. de Kock, W.J.M. Smits.** COSY Communication IP's. *In Proc. of* 37*th DAC*, *June* 2000.

[3] Cadence Design Systems. *Inc. VCC: Virtual Component Codesign, http://www.cadence.com/producs/vcc/html.*

[4] **W.O. Cesario, A. Baghdadi, L. Gauthier, D. Lyonnard, G. Nicolescu, Y. Paviot, S. Yoo, A.A. Jerraya, M. Diaz-Nava.** Component-Based Design Approach for Multicore SoCs. *In Proc. to DAC'*02, *June* 2002.

[5] **D. Culler, J.P. Singh, A. Gupta.** Parallel Computer Architecture. *A Hardware/Software Approach*. *Morgan Kaufmann Publishers, Aug.* 1998.

[6] **A. Gerstlauer, R. Domer, J. Peng, D.D. Gajski.** System Design: A Practical Guide With SpecC. *Kluwer Academic Publishers*, 2001.

[7] IBM, inc. 28.4G Packet Routing Switch. *Networking Technology Datasheets, available at http://www.chips.ibm.com/product/coreconnect/docs/crcon wp.pdf.*

[8] International Technology Roadmap for Semiconductors. *http://public.itrs.net/Files/2001ITRS/Home.htm,* 2001.

[9] **P.V. Knudsen, J. Madsen.** Integrating communicating protocol selection with partitioning in hardware/software codesign. *In ISSS*, 1998.

[10] **D. Lyonnard, S. Yoo, A. Baghdadi, A.A. Jerraya.** Automatic Generation of Application-Specific Architectures for Heterogeneneous Multiprocessors Systems-on-Chip. *In Proc. to* 38*th design automation post-conference, Las Vegas,* 2001.

[11] **A. Nagari, A. Mecchia, E. Viani, S. Pernici, P. Confalonieri, Nicollini G., and et al.** A 2.7v 11.8 mW Baseband ADC with 72 db Dynamic Range for GSM Applications. *In 21st annual Custom Integrated Circuits Conference*, 1999.

[12] **S. Narrayan, D.D. Gajski.** Synthesis of System-Level Bus Interfaces. *In Proc. of EDAC*, 1994.

[13] **M.D. Nava, G.S. Okvist.** The Zipper prototype: A Complete and Flexible VDSL Multi-carrier Solution. *ST Journal special issue xDSL*, *Sept*. 2001.

[14] **J. Oberg.** ProGram: A Grammar-Based Method for Specification and Hardware Synthesis of Communication Protocols. *PhD thesis, Departmen of Electronics, Electronics System Design, Royall Institute of Technology,* 7 *May* 1999.

[15] **Oka and Suzuoki.** Designing and programming the emotion engine. *IEEE Micro*, 19(6):20–28, *Nov*. 1999.

[16] Open SystemC Initiative. *http://www.systemc.org.*

[17] **R. Passerone, J.A. Rowson.** Automatic synthesis of interfaces between incompatible protocols. *In Proc. of* 35*th DAC*, *June* 1998.

[18] **J. Smith, G. De Micheli.** Automated Composition of Hardware Components. *In Proc. of 35th DAC*, 1998.

[19] Synopsys web page. *http://www.synopsys.com.*

[20] **S. Vercauteren, B. Lin, H.D. Man.** Constructing Application-Specific Heterogeneous Embedded Architectures from Custom HW/SW Applications. *In Proc. of DAC*'96*, June* 1996.

DOI: 10.5755/j01.itc.30.1.11797