Constructing Application-Specific Heterogeneous Embedded Architectures from Custom HW/SW Applications

Steven Vercauteren Bill Lin Hugo De Man

IMEC, Kapeldreef 75, B-3001 Leuven, Belgium
E-mail: {vercaut,billlin,deman}@imec.be
Tel: +32/16/28.15.25 ; Fax: +32/16/28.15.15

Abstract

Deep sub-micron processing technologies have enabled the implementation of new application-specific embedded architectures that integrate multiple software programmable processors (e.g. DSPs, microcontrollers) and dedicated hardware components together onto a single cost-efficient IC. These application-specific architectures are emerging as a key design solution to today's microelectronics design problems, which are being driven by emerging applications in the areas of wireless communication, broadband networking, and multimedia computing. However, the construction of these customized heterogeneous multiprocessor architectures, while ensuring that the hardware and software parts communicate correctly, is a tremendously difficult and highly error proned task with little or no tool support. In this paper, we present a solution to this embedded architecture co-synthesis problem based on an orchestrated combination of architectural strategies, parameterized libraries, and software tool support.

1 Introduction

Significant advances in broadband digital communication, wireless technologies, and multimedia computing have led to many new emerging business and consumer applications. The design of VLSI chips in these applications are often subject to stringent requirements in terms of programmability, processing performance, and power dissipation. Due to strong economic pressures, they must be highly cost-efficient and delivered in increasingly shorter time-to-market windows.

To enable flexible low-cost designs in a short design cycle, emerging designs are based on heterogeneous embedded system architectures that integrate multiple software programmable components, e.g. DSP and microcontroller cores, together with dedicated hardware components on to a single cost-efficient VLSI chip. Programmability is introduced in these system-on-silicon architectures (thus offering the desired flexibility in the design process), while maintaining most of the advantages of customized VLSI architectures (such as the potential to optimize the processing performance and power dissipation). Depending on the application, different application-specific multiprocessor architectures utilizing different combinations of software and hardware components may be required.

Despite these new tradeoff opportunities, designers of embedded systems are currently confronted with the enormously difficult task of constructing these complex architectures. Particularly error proned and timing consuming are the tasks of interfacing the hardware and software components together and providing communication between them. Designers currently spend a tremendous amount of time with low-level design details and extensive debugging to ensure correctness, thus leaving little room for optimization or exploration of alternative solutions. Surprisingly, there is remarkably little CAD support today to assist the designers in these key design tasks. Most research efforts have focussed on higher level issues such as system behavioral modeling, system level transformations, and system partitioning [6, 7, 9, 3, 15, 1, 16, 10, 12, 18]. Although these efforts have led to promising results, they are addressing largely complementary problems. The lower-level problems of embedded architecture co-synthesis and system integration still remain largely open. Recent efforts, e.g. [4, 24, 14], have led to promising progress in this area.

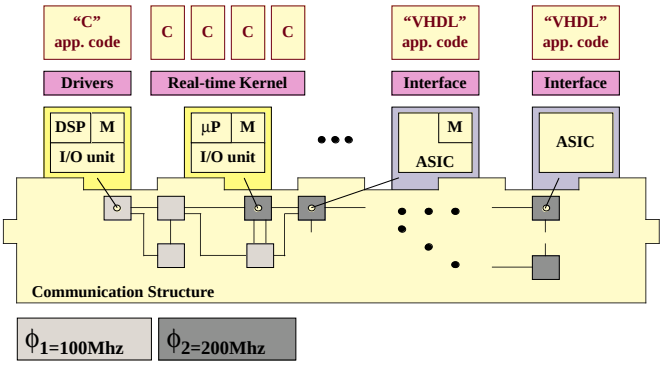

Fig. 1. Symphony: an orchestrated solution

In this paper, we present *Symphony* as a solution to the system architecture co-synthesis and integration problems, based on an orchestrated combination of architectural strategies, parameterized libraries, and CAD tools for automating low-level design tasks that are error proned and time consuming. This system is part of a larger heterogeneous system co-design environment called CoWare under construction at IMEC [5].

A key mechanism employed in Symphony to enforce the above design principles is the use of *layering*, as shown in Figure 1. In our approach we first build up *the physical architecture layer*, viewing the system to be composed of unified hardware modules and software modules with well defined interfaces at the circuit level. This is described in Section 2 where we propose a scalable target architecture model to construct the physical architecture layer. Secondly, we construct a *HW/SW communication and run-time abstraction layer*, that provides a language level communication interface by supporting VHDL packages and C call-able functions. This "HW/SW kernel" hides the details of the underlying physical layer, thus simplifying the programming of each component significantly. This is described in Section 3 and Section 4, where we present our strategy for modeling and programming software programmable processors and hardware components respectively. Section 5 describes some implementation issues. Section 6 highlights our preliminary application experience

33rd Design Automation Conference ®

DAC 96 - 06/96 Las Vegas, NV, USA

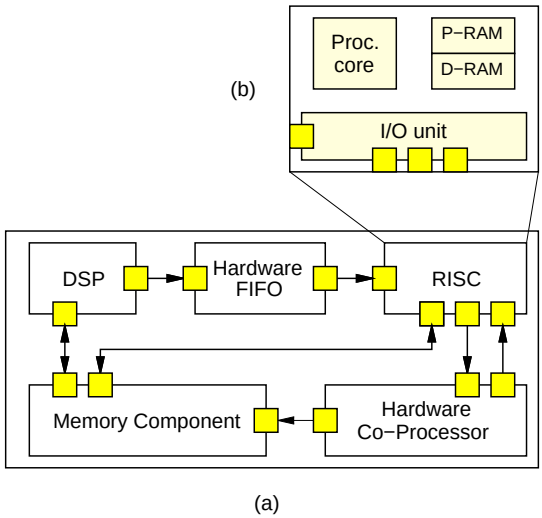

Fig. 2. A scalable component architecture model.

with using the proposed architectural strategies and design tools. Finally, conclusions are drawn in Section 7.

2 A Scalable Target Architecture

This section describes a scalable component architecture model that we use as the basis for constructing the physical architecture of application-specific embedded systems. Section 2.1 presents the architecture model. Section 2.2 describes our implementation strategy for realizing the architecture.

2.1 Architecture Model

In our model, the physical architecture is abstracted as an interconnection of "Processor Component Units" (PCUs) and unidirectional or bidirectional point-to-point communication channels, as depicted in Figure 2(a). Communication between the different component units is based on sending and receiving data to each other via the communication channels. We use Hoare's model of rendezvous, as defined for CSP [12], to define our channel communication semantics. In this model, the sender must block until the receiver is ready to receive, and vice versa. This rendezvous semantics ensures that both parties are synchronized with each other before the data transfer takes place. We have chosen the CSP communication model because it has a rigorously defined semantics along with a well defined algebra to reason about the communication behavior, supported by existing formal verification methods [22].

In this channel model, processor components can communicate directly via explicit "send" and "receive" operations on a specific channel or indirectly via an intermediate shared memory component. In Figure 2 this is illustrated by allocating a memory component that can be accessed by the DSP processor, the RISC processor and the hardware co-processor. This configuration will allow the three processor components to communicate with each other via shared memory. We believe this architecture model is sufficiently general as other communication models, such as buffered communication can be mimicked in a similar way by using intermediate component units that implement that communication behavior. In Figure 2 this is depicted by the hardware FIFO that buffers the communication between the RISC and the DSP processor.

A processor component unit can either be a hardware component or a software programmable component (e.g. DSP, ASIP, or microcontroller core). In the case of hardware, the processor component

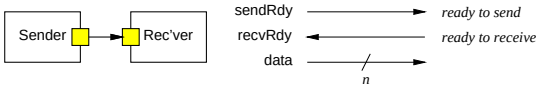

Fig. 3. Channel implemented using a synchronous wait protocol.

unit can either be a "pre-designed" library component, including parameterized communication components like buffers or memories, or an "application-specific" hardware processor that has still to be implemented. The hardware component may consist of internal storage. In the case of a software programmable component, the processor component unit consists of the processor core itself, an internal memory structure for storing the program instructions and run-time data, and a hardware I/O unit that implements the communication interface to its external environment, as depicted in Figure 2(b). The I/O unit acts as a "hardware wrapper" that effectively encapsulates a software programmable component into a hardware component. The I/O unit is driven by the processor core via the software program that executes on it. The implementation of this software side architecture is described in Section 3.

2.2 Implementation Strategy

To ensure different components can be integrated together at the "implementation" level, they must communicate with each other in a well-known and consistent manner at the "circuit" level. Our strategy to this problem is to define a common circuit level channel protocol that will be used by all components to implement the control mechanism for synchronizing the channel transfers.

We use in this work a synchronous transfer protocol called a *synchronous wait protocol* to implement our channels. In the synchronous wait protocol, the sender and receiver partners synchronize the communication by a pair of `sendRdy` and `recvRdy` signals, as shown in Figure 3. The sender partner implements a `send` operation by setting its `sendRdy` signal high and placing valid data on the `data` lines. This is shown in Figure 4. If the receiver is not yet ready, as indicated by the input `recvRdy` signal being low, then the sender enters into a "wait state" until the receiver is ready. This ensures synchronization. When the receiver is ready, as indicated by the input `recvRdy` signal being high, then the transfer is assumed to be completed in that clock cycle; thus the completion of of the data transfer is left *implicit.*

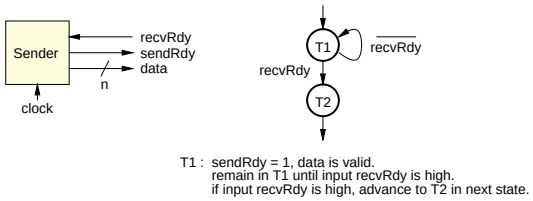

Fig. 4. Sender's abstraction.

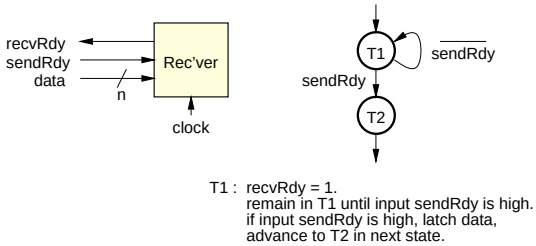

Fig. 5. Receiver's abstraction.

Similarly, the receiver partner implements a `receive` operation

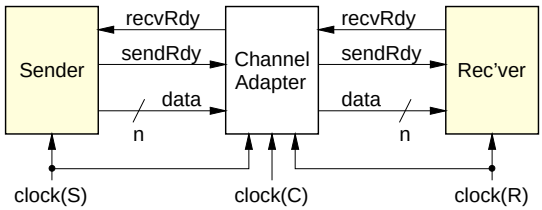

Fig. 6. Channel adapter for frequency conversion.

by setting its `recvRdy` signal high. This is shown in Figure 5. If the sender is not yet ready, as indicated by the input `sendRdy` signal being low, then the receiver enters into a wait state until the sender is ready. When the sender is ready, as indicated by the input `sendRdy` signal being high, then the receiver latches the data and moves to the next state.

For bidirectional channels the same reasoning applies, as in our model bidirectional channels are regarded as the merging of two opposite directed unidirectional channels sharing the same pair of control signals.

Despite its simplicity, the synchronous wait protocol offers several important advantages. One advantage is that the completion of communication is *implicit.* This means that when both partners are "ready", the communication behaves like a register transfer operation between two components. "Burst" transfer modes, where the sender transfers consecutively a sequence of data to the receiver, can be implemented very efficiently.

When the communicating components are clocked by different clock frequencies, we automatically produce a small *channel adapter* to handle the clock conversion. This is shown in Figure 6. When the communicating components operate under clocks that are derived from the same global system clock, this channel adapter is in effect a frequency converter, and it can be implemented using simple state machine logic. When the communicating components operate under unrelated clocks where the skew between clocks is non-deterministic, then the channel adapter is implemented using an internal handshake protocol and synchronizers.

Using this channel adapter approach, we can integrate "predesigned" reusable library components into a new custom embedded system architecture without the need to modify the description or the behavior of the reusable component itself, if the component has been designed in compliance with our component architecture model using the synchronous wait protocol. This issue is addressed in more details in Section 4.

3 Software Component Architecture

3.1 Processor Template

A processor component unit in our component architecture model may in fact be based on a *software* programmable processor core (e.g. DSP or micro-controller cores). Our approach to incorporating software processor component units into a custom target embedded architecture is based on building a parameterized "architecture template" around the processor core. There are three main components to this architecture template: the processor core itself, an internal memory structure for storing the program instructions and run-time data, and a hardware I/O unit that implements the hardware communication interface to the external environment. These components are interconnected via the "processor bus", which consists of the data bus, the address bus, and the control bus.

Using an architecture template to model a software processor, a software component is seen to other components in our component architecture model simply as another hardware processor component that communicates via dedicated channels, and the channels are implemented using a common circuit-level channel protocol, i.e. the synchronous wait protocol, as described in Section 2.2. From the user's perspective, the architecture template can be customized to provide a specified number of "physical" channels that supports user-defined directions and data widths. In fact, it is the hardware I/O unit in the architecture template that actually implements these physical channels for interconnection with other components. Based on this processor architecture model, a software processor component unit can be integrated into a custom target embedded architecture in the same way as another hardware processor component, using the approach described in Section 2.

3.2 Details of the Processor Template

In our architecture template, the I/O unit implements the communication protocol between the processor core and the external environment. Communication with the external environment is accomplished through one or more input or output ports attached to the I/O unit. These input or output ports are connected to communication channels that are connected to other processor components. These ports implement channel control using the implementation protocol described in Section 2.2. A more detailed architecture template using the ARM processor is shown in Figure 7. The input and output of data to and from the processor core is accomplished through one of two different methods: *memory-mapped I/O* or *instruction-programmed I/O*.

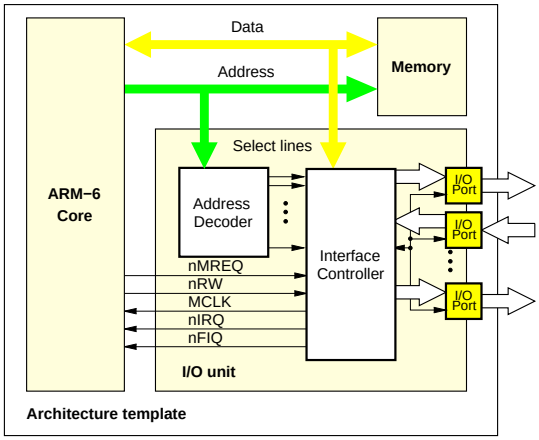

Fig. 7. Architecture template with hardware I/O unit expanded.

Memory-Mapped I/O. Memory-mapped I/O provides a data-transfer mechanism that is convenient because it does not require the use of special processor instructions, and can implement practically as many input or output ports as desired. In memory-mapped I/O, portions of the address space are assigned to input and output ports. Reads and writes to those addresses are interpreted as commands to the I/O ports. "Sending" to a memory-mapped location involves effectively executing a "Store" instruction on a pseudo-memory location connected to an output port, and "Receiving" from a a memory-mapped location involves effectively executing a "Load" instruction on a pseudo-memory location connected to an input port. When these memory operations are executed on the portions of address space assigned to memory-mapped I/O, the memory system ignores the operation. The I/O unit, however, sees the operation and performs the corresponding operation to the connected I/O ports.

For custom embedded architecture synthesis, the number of memory locations assigned for memory-mapped I/O will depend on the number of "channels" that a software processor component has to "physically" implement. Here, the assignment of address locations to channels can be user-defined. However, it is usually preferred that the address locations assigned for memory-mapped I/O be a "contiguous" memory region. This greatly simplifies the address decoding logic for the I/O unit. In this case, address location assignment is automatically performed by off-setting from a user defined base address location.

Instruction-Programmed I/O. Some processors also provide special instructions for accessing special I/O ports provided with the processor itself. Using this scheme, these special communication ports of the processor are connected the external channels via the I/O unit. We use a simple greedy algorithm (like in [4]) that uses these programmed I/O ports first if they are less expensive than using memory-mapped I/O. If communication via special programmed I/O instructions is more expensive, or not available, then only memory-mapped I/O will be used.

Interrupt Control. In addition to providing hardware support for memory-mapped and instruction-programmed I/O, the I/O unit also provides support for hardware interrupt control. Interrupts are used for different purposes, including the coordination of interrupt-driven I/O transfers, as described in Section 3.4. Different processors provide different degree of hardware interrupt support. Some processors provide direct access to a number of dedicated interrupt signals. Our I/O unit architecture makes use of these signals when available. If more interrupt "channels" are required, as for example required to support a number of interrupt-driven communication channels, we use the strategy of *interrupt vectors*. Interrupt vectors are pointers or addresses that tell the processor core where to jump to for the interrupt service routine. In effect, this is a kind of memory-mapped interrupt handling.

Direct Memory Access Support. Optional to our I/O unit architecture template is the addition of a direct memory access (DMA) controller. This DMA controller can access directly the data memory of the software processor component unit via the processor bus. It can be used to support high-speed data exchange directly between the (external) communication channels and the processor data memory without intervention of the processor. This leaves the processor to proceed with other computation tasks rather than being consumed by managing data transfers. The processor needs to initiate the DMA transfer by indicating to the DMA controller the start address of memory, the number of items to be transferred, the (external) communication channel where the transfer will take place, and the direction of transfer (i.e. reads or writes). Once initiated, the direct memory access transfers are accomplished between the specified communication channel and the memory. Upon completion of the transfer, the DMA controller will interrupt the processor to indicate completion. If the DMA controller option is desired by the user, we will also produce automatically the necessary bus arbitration logic to manage the bus contention on the processor bus and the control logic to enable the processor to control the DMA controller.

3.3 Generation of I/O Unit

The I/O unit must be connected to the processor bus of the processor core. Different processors use different processor bus protocols, usually described as timing diagrams in data books, for implementing their memory "read" or "write" operations, or their special programmed I/O operations. Our interface synthesis tools can also automate the design of the necessary protocol matching hardware for communication with a specific processor bus protocol [20].

3.4 Software Communication Synthesis

On the software side, the processor can be programmed using the C or C++ language. To facilitate external communication, we automatically generate a custom library of C or C++ "call-able" functions with `send` and `receive` operations. This library is automatically generated depending on the number of channels that the software component must support, their directions, and their data width. The library can be thought of as a "customized" communication kernel. From the programmer's perspective, the application program communicates with the external world via function calls to the appropriate `send` and `receive` operations. This abstraction isolates the programmer from the low-level details of how the processor actually interacts with the environment.

The send and receive routines are implemented using the memory "read" and "write" instructions if memory-mapped I/O is used for the specific channel, or the corresponding special programmed I/O instructions if instruction-programmed I/O is used.

Because we use a rendezvous semantics to implement the channels in our component architecture model, a communication operation to another processor component unit may take an arbitrary amount of time since both the "sender" and the "receiver" must be "ready". To avoid unnecessary "idling", we use an *interrupt-driven I/O scheme*. In this scheme, the "send" and "receive" routines in software are each split into two operations: the *initiation* and *continuation* operations. The initiator routine is responsible for getting an I/O operation started. In the case of a "send" operation, the processor transfers the data to the I/O unit. Once the data has been transferred to the I/O unit, the processor can proceed with other tasks if it is using a multi-tasking kernel, or a compiler can insert instructions at compiled time after the initiator operation that don't depend on the completion of the send operation. When the actual send operation to the external environment is completed, the I/O unit interrupts the processor. The continuator routine is then responsible for notifying the calling routine that the send operation has completed. Similar, in the case of a "receive" operation, the processor initiates the operation by notifying the I/O unit. The I/O unit then coordinates the receive operation via the I/O ports along the specified channel. When the actual receive operation from the external environment is completed, the I/O unit interrupts the processor. The continuator routine is then responsible for transferring the data to the processor. To support the above I/O operations, the necessary interrupt controller functionalities are also synthesized into the I/O unit hardware.

If DMA support is also included into the I/O unit, as specified by the user, then the necessary software routines for coordinating "burst" direct memory access transfers are also automatically generated, similar to the single "send" and "receive" operations above, but parameterized for block transfers.

Shared Memory Communication. For certain applications in the domain of modern telecommunication systems, the desired behavior is often characterized by complex algorithms that operate on large dynamically allocated data structures. Designing an embedded architecture for such applications will involve allocating a large memory that is shared by different tasks within the application. From a programmer's point of view shared memory communication will then be a much more "natural" paradigm to use, compared to message passing. We have chosen to support the designer with this facility. If desired, the programmer can then work with traditional memory pointers for managing the shared data structures, instead of calling explicit

send and receive functions. As it was clear from the discussion in Section 2, shared memory communication can be supported in our component architecture model by communicating with a shared memory component unit. From a user's perspective, the designer only has to allocate a memory space for the shared data structures. Each memory access that falls within this memory space will then activate appropriate channel communications with the component memory component.

4 Hardware Component Architecture

4.1 Hardware Communication Synthesis

In our target architecture model, a hardware processor component in an application-specific embedded system architecture may still need to be implemented. In this case, we automatically generate for the user a "container" that essentially implements a "communication wrapper" around the hardware that the user will later provide. From a user's perspective, the designer only needs to declare the number of communication channels required for the hardware processor component, their directions, and the data width that they must support. We then automatically generate a customized VHDL package that implements the communication channels, according to the synchronous wait protocol described in Section 2.2.

This VHDL package provides a set of `send` and `receive` operations that can be "called" by an application program in VHDL, which provides an abstraction of the external communication. The designer can then "program" this "container" by writing a VHDL program that uses the `send` and `receive` operations provided by the VHDL package for external communication. If the designer chooses to use another programming language to program the hardware, e.g. Silage [11], then VHDL and the VHDL packages generated can be used as an intermediate interface to a lower level hardware implementation trajectory.

4.2 Parameterized Libraries

Using our component architecture model and our common circuit-level protocol, reusable library components can be modeled and integrated into a custom embedded architecture. The library component can be in the form of synthesizable VHDL source code or already at the circuit level. The main requirement is that all external communication with the outside world must be implemented using the CSP rendezvous channel concept (cf. Section 2.1), and the implementation protocol is the synchronous wait protocol described in Section 2.2. The components may operate under different clocks. Such components can be automatically integrated into a custom architecture, without the need to modify the description or the behavior of the reusable component itself, by synthesizing channel adapters (cf. Section 2.2). To represent parameterizable components, we use the parameterization features of VHDL.

Communication Components. An important class of parameterized library hardware components is the class of communication components. For example, we keep in the library a parameterized synthesizable VHDL model of a FIFO communication buffer. This model is parameterized by depth and data width. It is designed to communicate using the CSP channel model and the synchronous wait protocol. This hardware "component" can be instantiated and integrated like any other hardware component. As already mentioned in Section 2, we only model rendezvous channels in our component architecture model. We instead mimic other communication models by means of intermediate communication components that implement that communication behavior. This strategy has the advantage that different communication models can be supported and the semantics of the communication paradigm is made explicit. Given a library of parameterized communication components, a custom embedded architecture can be defined at a high level using them as intermediate communication units, as for example illustrated in Figure 1.

4.3 Shared Memories

Particularly interesting is the integration of memory components as they will allow for shared memory communication. Memory components can be seen as special cases of hardware components as they consist of large internal storage with minor combinational logic. In a application-specific embedded system architecture, a memory component can be used from a previous design or may still need to implemented. In the former case, the memory component is instantiated from the library of predefined communication components. In the latter case we build a parameterized architecture template around the specific memory. In this architecture template, communication with the other components is accomplished through a number of I/O ports attached to a memory controller. These I/O ports are connected to communication channels and implement channel control using the implementation protocol described in Section 2.2. The memory controller arbitrates the different accesses of the I/O ports to the memory and reflects the number of ports, the storage size, the word width and other specific characteristics of the selected memory. Depending on the required channel layout, the architecture template can then be hardwired automatically.

5 Implementation

We have implemented the proposed techniques that have been described in this paper. As mentioned above, Symphony is part of a larger heterogeneous system co-design environment called CoWare [5]. In CoWare the communication programming model is built on top of the concept of the Remote Procedure Call (RPC), i.e. one component can trigger the execution of a procedure in another component. From the designer's perspective, the different processors can then be programmed using the C, C++ or VHDL language, communicating with the external world via explicit RPC calls. To facilitate this RPC mechanism, the automatically generated libraries also provide *communication stubs* that actually refine the abstract RPC calls by marshaling and passing argument parameters, if necessary, somewhat akin to remote procedure calls in computer communication and networking. To transfer data to another component, the communication stubs call the generated `send` and `receive` operations for external communication on the appropriate communication channels.

6 Application Experience

The demonstrator application discussed in this paper is a Segment Protocol Processor (SPP)[27]. This chip is a crucial component of a connectionless server in an Asynchronous Transfer Mode (ATM) network[28]. ATM is a fast packet switching transfer mode that supports high-speed integrated services by splitting all communications into equal 53-byte cells. These cells can be used to carry every kind of information, be it computer data, video, or voice. In addition to the original packet data, the cells also carry various fields for control and management purposes. Packets from different applications are also interleaved.

In this demonstrator we present an experiment that considers a partitioning, as depicted in Figure 8, with two ARM RISC processor cores, one hardware fifo and a shared memory. The code mapped onto the upper ARM processor receives incoming ATM cells, does

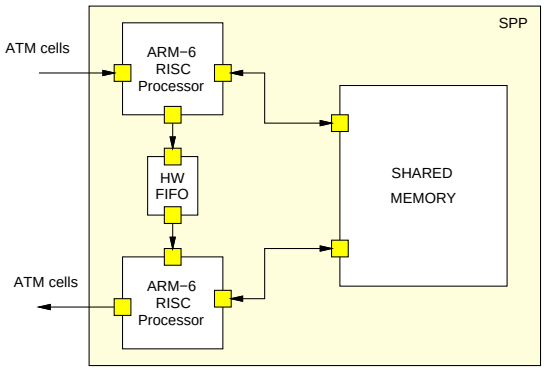

Fig. 8. The system architecture

some processing, checks the cells for their integrity, creates and updates some dynamically allocated data structures and stores cells into the hardware fifo. The code on the lower ARM processor is responsible for getting cells out of the hardware fifo, deallocating the appropriate data structures in the shared memory, outputting updated ATM cells, etc.

The "plug-and-play" concept offered by our scalable architecture model reduced the problem of constructing this customized architecture to simply specifying the processor and communication channel layout. For the shared memory we used an architecture template built around a VTI memory VHDL model. The memory controller and the I/O ports were then automatically generated. The hardware fifo was programmed in VHDL and using the `send` and `receive` procedures from the VHDL communication packages synthesized. The software running on the ARM processors was implemented in C using the generated library of C call-able `send` and `receive` functions for communication with the hardware fifo. Manipulations and modifications of the data structures allocated in the shared memory, were specified with "ordinary" pointer references.

The whole design flow was supported by tools, for generating the hardware containers, software libraries and VHDL packages, partly based on parameterized libraries.

7 Conclusions

In this paper, we presented an approach to the embedded architecture co-synthesis problem. Our approach is based on an orchestrated combination of architectural strategies, parameterized libraries, and software tool support. We showed how software programmable and dedicated hardware components can be integrated together to form an application-specific multiprocessor architecture. In the future, we plan to expand our support for a larger portfolio of processor cores and to further apply our approach to other practical applications. It would also be interesting to see how system partitioning techniques (e.g. [9, 6, 7, 15]) can be applied on our customized architectures.

Acknowledgments

We would like to thank K. Van Rompaey, D. Verkest, I. Bolsens, G. Goossens, F. Catthoor, B. Gyselinckx, J. Silva, C. Ykman, G. de Jong, T. Kolks, and E. Verhulst for numerous insightful discussions on the system integration problem.

References

[1] G. Berry, P. Couronne, and G. Gonthier. The synchronous approach to reactive and real-time sysems. *IEEE Proceedings*, 79, September 1991.

[2] J. T. Buck et al. Ptolemy: A framework for simulating and prototyping heterogeneous systems. *International Journal on Computer Simulation*, January 1994.

[3] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavagno, and A. Sangiovanni-Vincentelli. A formal methodology for hardware/software codesign of embedded systems. *IEEE Micro*, August 1994.

[4] P. H. Chou, R. B. Ortega, G. Borriello. The Chinook Hardware/Software Co-Synthesis System, *International Symposium on System Synthesis*, September 1995.

[5] H. De Man, I. Bolsens, B. Lin, K. Van Rompaey, S. Vercauteren, and D. Verkest. Co-design of DSP systems. *NATO ASI Hardware/Software Co-Design*, Tremezzo, June 1995.

[6] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Design and Test of Computers*, 10(4):64–75, December 1993.

[7] D. Gajski, F. Vahid, S. Narayan, and J. Cong. *Specification and Design of Embedded Systems*. Prentice-Hall, 1994.

[8] G. Goossens, I. Bolsens, B. Lin, and F. Catthoor. Design of heterogeneous ICs for mobile and personal communication systems. *IEEE International Conference on Computer-Aided Design*, November 1994.

[9] R. Gupta and G. De Micheli. Hardware-software cosynthesis for digital systems. *Computers and Electrical Engineering*, 10(3)29–41, September 1993.

[10] D. Harel. StateCharts: A visual formalism for complex systems. *Science of Computer Programming*, (8):231 – 274, 1987.

[11] P. .N. Hilfinger et al. DSP specification using the Silage language. *Proceedings International Conference on Acoustics, Speech and Signal Processing*, page 1057 – 1060. April 1990.

[12] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[13] IEEE, *IEEE/ANSI Standard 1014, Versatile Backplane Bus: VMEbus*, IEEE Service Center, Piscataway, NJ, 1987.

[14] C. A. Valderrama, A. Changuel, P. V. Raghavan, M. Abid, T. Ben Ismail, and A. A. Jerraya. A unified model for co-simulation and co-synthesis of mixed hardware/software systems. *Proc. ED&TC-95*, Paris, France, March, 1995.

[15] T. B. Ismail, K. O'Brien, and A. A. Jerraya. Interactive system-level partitioning with PARTIF. *Proc. EDAC-94*, Paris, France, February, 1994.

[16] A. Kalavade, E. A. Lee. A Hardware/Software Codesign Methodology for DSP Applications. *IEEE Design and Test of Computers*, vol.10, no.3, pp.16-28, September, 1993.

[17] D. Lanneer, J. Van Praet, K. Schoofs, W. Geurts, A. Kifli, F. Thoen, and G. Goossens. Chess: retargetable code generation for embedded processors. In P. Marwedel and G. Goossens, editors, *Code Generation for Embedded Processors*. Kluwer Academic Publishers, Boston, 1995.

[18] R. Lauwereins et al. Grape-II: A system level prototyping environment for DSP applications. *IEEE Computer*, pages 35 – 43, February 1995.

[19] E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *IEEE Proceedings*, September 1987.

[20] B. Lin and S. Vercauteren. Synthesis of concurrent system interface modules with automatic protocol conversion generation. In *Proceedings of the IEEE International Conference on Computer-Aided Design, ICCAD 94*, pages 101 – 108. San José, CA, November 1994.

[21] P. G. Paulin, C. Liem, T. C. May, and S. Sutarwala. FlexWare : a flexible firmware development environment for embedded systems. In *Code generation for embedded processors*, pp. 67–84, Kluwer Acad. Publ., Boston, 1995.

[22] A. W. Roscoe, C. A. R. Hoare. Laws of Occam Programming. *Theoretical Computer Science*, 60, 177–229, 1988.

[23] C. L. Seitz. System Timing. In C. .A. Mead and L. A. Conway, editors, *Introduction to VLSI Systems*, Chapter 7, Addison-Wesley, 1980.

[24] M. Srivastava, B. Richards, and R. W. Brodersen. System level hardware module generation. *IEEE Transactions on VLSI Systems*, 3(1), March 1995.

[25] K. van Berkel. *Handshake circuits: an intermediary between communicating processes and VLSI*. Ph.D thesis, Philips Research Laboratories, Eindhoven, The Netherlands, 1992.

[26] A. van Someren and C. Atack. *The ARM RISC Chip, A programmer's Guide*. Addison-Wesley Publishing Company, 1994. ISBN 0201406950.

[27] Y. Therasse, G. Petit, M. Delvaux, "VLSI Architecture of a SMDS/ATM Router", in Annales des Télécommunications, Vol. 48, No. 3-4, 1993.

[28] M. De Prycker, "Asynchronous Transfer Mode, Solution for Broadband ISDN", Ellis Horwood, 1991.