Automatic Code-Transformation and Architecture Refinement for Application-Specific Multiprocessor SoCs with Shared Memory

Samy Meftali, Ferid Gharsalli, Frédéric Rousseau and Ahmed. A. Jerraya
TIMA laboratory, 46 av. Felix Viallet 38031 Grenoble cedex (France)

Abstract: Memory represents a major bottleneck in embedded systems. For multimedia applications bulky of data in these embedded systems require shared memory. But the integration of this kind of memory implies some architectural modifications and code transformations. And no automatic tool exists allowing designers to integrate shared memory in the SoC design flow. In this work, we present a systematic approach for the design of shared memory architectures for application-specific multiprocessor systems-on-chip. This work focuses on the code-transformations related to the integration of a shared memory.

Key words: Shared memory, code transformation, architecture refinement.

1. INTRODUCTION

The design of modern digital systems is influenced by several technological and market trends, including the ability to manufacture ever more complex chips but with increasingly shorter time-to-market.

The choice of a shared memory architecture for a given SoC implies the integration of some new modules in the application description (memory and controllers) and many code transformations at several abstraction levels in the design process.

The goal of transformations and code generation in the case of multiprocessor SoCs with a shared memory is to adapt the code of the application to a such memory architecture. In fact, we imperatively need to replace the simple shared data accesses at a high abstraction level by explicit requests to the shared memory block.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35597-9_40

M. Robert et al. (eds.), *SOC Design Methodologies*
© IFIP International Federation for Information Processing 2002

Unfortunately, nowadays, there is not a complete and automatic method allowing designers to integrate all these memory types (particularly the shared memory) in the SoC from a high abstraction level. Our objective is to provide designers with a global and fast method and tools to design such a systems in order to satisfy the time-to-market constraints. We focus in this work on the code-transformations due to the integration of the shared memory into the SoC and on the automatic code generation.

Our approach is easily automatisable and allows a completely automatic generation of an architecture level specification of the application. Now, multiprocessor SoC integrate more and more elements, and the description of such a system at the architecture level can reach 200k lines of code (SystemC, C, VHDL), which makes this work very beneficial to the designers from the time-to-market point of view.

This work is organized as follows: in Section 2, we give an overview of related work on code transformations on SoC with shared memory. In Section 3, we present our multiprocessor SoC design methodology, then our three abstraction levels and the memory representations in section 4. Section 5 describes the code transformations. These transformations are illustrated by an application in Section 6. We conclude this work in Section 7.

2. RELATED WORK

In this work we are only concerned with SoCs. These system architectures are different from classic general purpose architectures [3] because they target a specific application. This makes the memory architecture and the communication network specific to the application and then simpler. For instance, in most of these applications data regularity is quite trivial or non existing and thus no sophisticated data cache is required.

In the literature three kinds of code transformations exist depending on the level where they are performed.

The high level transformations concern the application code at the system level. Their goal is to improve the code quality. In fact they consist mainly in modifying loop structures in order to reduce the number of memory accesses [2]. This kind of code-transformations does not take into account the specificity of the memory architecture chosen for the application, and these transformations are not generally needed when the code is written by an experimented designer. Many research groups [13] [8] [4] work on these high level transformations.

The low level transformations are generally very related to the low level (RTL) characteristics of the architecture. Some of them are due to memory characteristics. For example in [10] [11] they use fast access modes (read

and write page mode) on a typical DRAM to improve memory cache performances. Using these access modes, different data can quickly be read or write in the same page. Moreover, a good scheduling of elementary instructions (generated by compiler) which access to the memory allows to obtain better performances. Unfortunately the low level transformations appear at the end of the design flow. So, the integration of a shared memory in the system is done manually, which is very time consuming.

Some code transformations in the literature concern more than one abstraction level in the design flow. IMEC [2] works on the generation of the optimized memory part for embedded systems (single process). The main idea is to study the application and generate memory architecture, for single process applications with a parallelizing compiler.

The contribution of our work is the proposal of a full systematic approach allowing a multi level automatic code transformations and generation for application-specific multiprocessor SoCs with a shared memory.

3. MP SOCS DESIGN WITH SHARED MEMORY

The methodology and tools we developed to design multiprocessor SoC are described in [1]. It starts a system level specification (Figure 1). Processors and communication components are allocated and system behavior and communication (ports and channels) are mapped/scheduled on processors and communication components of the architecture template (by the designer). After the allocation/mapping step, an architecture level specification is obtained.

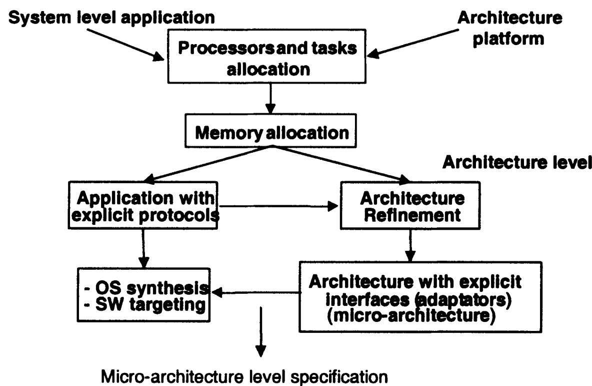

Figure 1. Our design flow

For each processor, the software code (Operating System and application code of tasks) is assembled (from libraries). Communication interfaces between processors and the communication network are also generated. We obtain the micro-architecture of the system.

The choice of the processor was based on availability (only ARM7 and MC68K processors). The communication network is a point-to-point network. Designers can modify some parameters such as the number of CPUs, the memory size and I/O ports for each processor, interconnection between processors, the communication protocols and the external connections (peripherals).

One of the most important steps in our design flow is the memory allocation. During this step, we try to choose an optimal memory architecture for the SoC. Therefore, we use an integer linear programming model generated from the system specification of the application. It allows us to choose the best memory architecture for the design [9].

4. ABSTRACTION LEVELS IN OUR DESIGN FLOW

In this section, we define the three abstraction levels used in the design flow [12]. As our objective is to integrate the shared memory into the SoC from a high abstraction level, one will focus in the remainder of this work primarily on the first two levels.

4.1 System level

At this level (*Figure 2*), modules communicate through abstract channels. No assumption about communication implementation is made. Hence, the abstract channels ensure independent protocol communication of concrete generic data types by providing abstract level communication primitives (c.g. send/receive). Such primitives encapsulate all the communication details. Basic module behavior are described by tasks communicating by sending and receiving messages. SDL is a typical system level language.

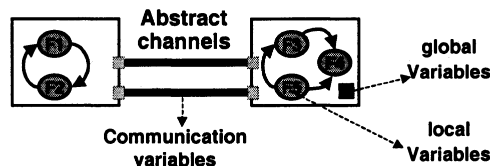

Figure 2. System level

4.2 Architecture level

At this level (Figure 3), modules correspond to the architecture blocks. Communication is modeled by logical interconnections encapsulating architecture level protocols (e.g. handshake or finite FIFO). The communication primitives on the module ports are read/write fixed data types in conformity with a certain protocol (e.g. read-handshake or write-handshake). SystemC1.0 is an example of languages that describe systems at this abstraction level.

In our design flow, the architecture level description is automatically generated. It is obtained after the memory blocks allocation step. In fact, if we decide for example to integrate a shared memory into the SoC, we have to insert a block into the application description. This block will mainly contain two modules:

- The memory matrix: it is a generic code describing the memory block. It is independent of the application which permits an easy automation. At this level, this module is connected only to the controller by the address channel, data read and data write channels. So it does not depend at all on the number of processors, accessing the shared data.
- The memory controller: it is built of two parts connected to the memory matrix: one input and one output controller. The input controller is also connected to all the processors writing data in the shared memory and the output controller is connected to those which read data from the shared memory. These controllers will be the memory adaptator at the micro-architecture level.

The fact of separating the shared memory into 3 modules (matrix, input and output controllers) at this level gives a better modularity to the SoC. For example, if we decide to reuse the SoC for an other application with an additional processor accessing to the shared data for writing, we do have to modify neither the description of the matrix nor the output controller.

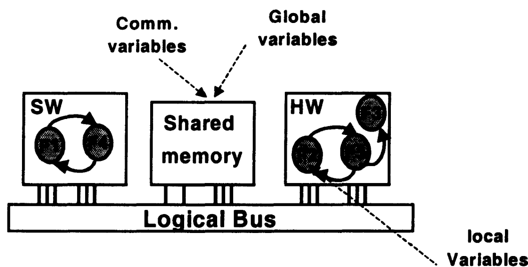

Figure 3. Macro-architecture level

4.3 Micro-architecture level

At the micro-architecture level, the modules are physical blocks (DSP, CPU, IP ...) Communication is modeled by physical signals and communication primitives are consequently set/reset of signals [1]. Communication time is based on the clock cycle. VHDL and Verilog are languages permitting to describe systems at the micro-architecture level.

At this level, memories are physical (SDRAMs). In order to connect the shared memory to the communication network, we insert between them one memory adaptator which adapts the access protocol of the memory to that of the network. The memory interface is independent of the processors, which increases the flexibility of the target architecture. It depends only on the communication network and the memory. The memory interface is assembled using basic components in our libraries [5].

5. APPLICATION CODE-TRANSFORMATIONS

In this section, specific code-transformations are presented. Some of them are related to memory accesses, when a shared memory is added, and some others concern memory controllers refinement.

5.1 Architecture level transformations

At the system level we have only processors communicating by messages passing, and we do not find any shared memory or any memory controller blocks in the application description. The data exchange between the blocks at this level is made by simple Send/Receive primitives.

After deciding which data would be in the shared memory block at the memory allocation step, we have to deal with two kinds of shared data. In fact, we distinguish global variables and communication data. Each one of these types needs an appropriate code-transformation in order to generate a new application specification taking into account the shared memory.

5.1.1 Synchronization signals (binary type)

We assume in our applications that the data consistency problems are entirely resolved by the system level designer, by synchronization. So, we choose in our tools to never modify such signals. In fact, synchronization signals are boolean so, we do not put any binary variable/signal in a shared memory. This does not decrease the performances of our SoC due to the small size of such a type of data.

5.1.2 Non-binary communication variables

When we decide in the allocation step to insert a shared memory in the SoC, we have to modify all accesses to the variables that we decide to put in such a memory by explicit accesses to the shared memory. We insert the shared memory into the application, and we generate an abstract allocation table that contains for each shared variable, in which memory it will be placed. So, at this level each data in the shared memory must be characterized by a logical address (index in the memory matrix) and a name in the abstract allocation table.

Suppose that at the system level "X" is a shared data between the processors P1 and P2. The system level code in the two processors corresponding to the exchange of "X" will be as in *Figure 4*

Processor_sender_P2	**Processor_receiver-P1**

{
..
Send(X,P1);
Send(synch_signal,P1);
Wait();
..
}
	{
	..
	Wait for synch_signal_P2
	Receive(X);
	..
	}

Figure 4. System level communication

In order to send "X" to P1, the processor P2 has just to send the variable value then a synchronization signal informing P1. This later waits on the synchronization signal from P1, then receives "X" through the channel connecting it to P2.

At the architecture level, if we decide to insert a shared memory into the system, the shared data "X" will be into this memory and not in the P1 and/or P2 local memories.

5.1.2.1 Sending data

The primitive Send(X) in P2 behavior code will be transformed in a writing request to the input shared memory controller as shown in the following code (*Figure 5*).

Processor_sender_P2	Input_Controller
{	{
..	..
write_SM_input_ctr("X",X);	ind = allocation_table ("X");
write(synch_signal);	write_value(ind,X);
wait();	wait();
..	..
}	}

Figure 5. Sending data at macro-architecture level

After receiving a such request, the input memory controller takes the data index in the memory matrix from the abstract allocation table. Then writes the data value in the corresponding memory cell.

5.1.2.2 Receiving data

As in the case of sending a data, the primitive Receive(X) in P1 behavior code will be transformed as in *Figure 6*

Processor_receiver_P1	Output_Controller
{	{
..	...
Wait for synch_signal;	Ind = allocation_table ("X");
Ask_for_data_in_shared_memory ("X");	X = Read_value(ind);
Wait();	Write_data_2_P1(X);
Read_data(X);	Wait();
..	....
}	}

Figure 6. Receiving data at macro-architecture level

Notes:

- For the architecture level transformation of the Send primitive, the processor sender must give the variable value (X) and its label ("X").
- All the instructions corresponding to a write or read operation in the memory controller code are executed in one clock cycle.

5.1.3 Global variables

In the system level application code, we find some accesses to the global data in some expressions/assignments in the behavioral part of processes. In fact, if "X" is a global variable, we can find in a process in the system level description an expression as: Y = X + 2, or X = Y/2;

The first instruction correspond to a read access and the second one to a write access to "X".

If "X" is in the shared memory, we must generate explicit accesses to this variable in the new application code. So, in the two cases (read and write) we replace the occurrence of "X" in the code as in *Figure 7.*

The instruction (1) consists in sending a signal through the channel connecting process to the shared memory output controller, to ask for the variable "X". After receiving a such signal (after the synchronization instruction (2)), the controller finds the index corresponding to "X" in the abstract allocation table then reads the variable in the memory matrix, and sends it to the processor. In the code, this value is read and copied in a local variable "var" (3). Then the expression is performed in (4).

For the second instruction (write) "X = Y/2", the instruction (5) consists in sending a writing message from "A" to the input shared memory controller. This message contain two parameters: the name of the shared variable "X" and its new value (Y/2).

"Y = X + 2"	"X = Y / 2"
{	{
...	...
Sig_to_read_shared_mem(X) ; --- (1)	write_shared_mem("X", Y/2) ; --- (5)
wait(); --- (2)	wait(); --- (6)
var = read_shared_mem(X); --- (3)	....
Y = var + 2; --- (4)	}
....	
}	

Figure 7. Code transformation for global variables

5.2 Architecture level application-code generation

One of the main contributions of our work is the systematic and completely automatic generation of the architecture level description of the application. This step consists in inserting the high level memory matrix block in the application specification, then generating the application specific code of the input and output memory controllers in order to connect the memory with other modules of the SoC. After that we perform the necessary code-transformations described in this work in order to adapt the accesses to the shared memory architecture.

5.3 Micro-architecture level transformations

At this level, read and write operations in the shared memory become very explicit and dependent of the shared memory characteristics.

The input and output controllers of the architecture level are refined to be a communication adapter between the memory and the communication network. In our case we consider that the memory adaptator is a slave processors adaptators. The adaptator receives an access request containing the address and the data in the case of a writing request. If there is any

competitor access, the controller performs the address decoding while using an allocation table more detailed than that of the previous level, and updates the memory signals (in read or write modes). After a certain memory latency, the adaptator sends an acknowledge to the processors adaptator allowing it to ask for new accesses.

In the case of several simultaneous accesses, the memory adaptator does the same thing while respecting mutual accesses exclusion to the memory.

6. APPLICATION

In order to illustrate the efficiency of the proposed code-transformations and code generation methodology, we detail in this section the flow steps on a packet routing switch. It constitutes a powerful solution for large-frame or cell-switching systems [7]. The version we present here consists of two input controllers and two output controllers. Each of the controllers handles one communication channel. The communication links between input and output controllers are configured by an external signal to be direct or switched. *Figure 8* shows the block diagram of the packet routing switch.

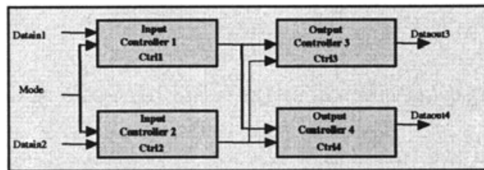

Figure 8. Block diagram of the packet routing switch

6.1 Architecture level code-transformations

In this application, after the memory allocation step, we decide to put two variables in the shared memory block. This stage modifies the application code by taking into account the shared memory architecture.

At this step a shared memory module, an input and an output memory controllers are generated and integrated to the application as shown in the *Figure 10*. The memory input controller is connected to controller1 and controler2 because these later access memory to write data. The memory output controller is connected to controller3 and controller4 as they access the memory to read the shared data. We generate also an abstract allocation table which contain label, type, index in memory of each shared data.

```
//Implementation of Ctrl1//

void ctrl1 ::entry()
 {
   ...
if( ordre.read()==true)
{
dataCtrl1_mem_in.write(x);
dataCtrl1_mem_in.write("x");
signalCtrl1_C3.write(true);
Wait();

...
```

```
/Implementation of Ctrl 3 //

void ctrl3 ::entry()
 {
   ...
if (sig_Ctrl1_C3.read()==true)
{
Signal_mem_out.wrtite("X");
Wait();
y = data_mem3_out.read();
output3.write(y)

...
```

Figure 9. Communication between Ctrl1 and Ctrl2 using a global shared memory

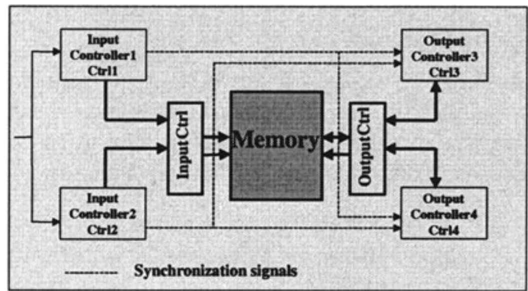

Figure 10. Packet routing switch description at macro-architecture level

6.2 Analysis

This application was described at the functional level mainly in 4 interface files and 4 implementation files. Automatic refinement adds 4 files to the specification (2 interfaces and 2 implementations) corresponding to the memory body and to the memory controller (200 lines at the functional level). The interfaces of the 4 processors were modified automatically in order to connect them to the global shared memory, and all the accesses to the data resident in this memory were modified (Figure 9). We obtained the application code at the architecture level with a shared memory architecture in a complete automatic way.

The automatic code-transformation and generation is surely very profitable to the designer in time-to-market point of view because of the huge size of the SoC descriptions especially at micro-architecture level.

7. CONCLUSION

In this work, we presented an automatic architecture refinement and code-transformations flow for application-specific SoC with a shared memory, starting from a parallel system-level description of a given application. We focus on the shared memory representation at different abstraction levels and the code-transformations. The proposed methodology permits a systematic code-transformation and the generation of a generic memory architecture for multiprocessor embedded SoC, from a high abstraction level distributed specification of the application. We have seen the effectiveness of our approach on an example.

8. REFERENCES

[1] A. Baghdadi, D. Lyonnard, N-E. Zergainoh, A.A. Jerraya, "An Efficient Architecture Model for Systematic Design of Application-Specific Multiprocessor SoC", DATE'2001.

[2] F. Cathoor & al, Custom Memory Management Methodology, Kluwer Academic Publishers, 1998.

[3] D. Culler, J.P. Singh, A. Gupta, "Parallel computer architecture: A Hardware/Software approach", Maurgan Kauffman publishers, August 1998.

[4] A. Fraboulet, G. Huard, A. Mignotte, "Loop Alignment for Memory Accesses Optimization", Proc. of ISSS 1999.

[5] F. Gharsalli, S. Meftali, , F. Rousseau, A.A. Jerraya, " Automatic Generation of Embedded Memory Wrapper", Proc. of DAC 2002.

[6] J. Hennessy , M. Heinrich, A. Gupta, "Cache-Coherent Distributed Shared Memory: Perspectives on Its Development and Future Challenges", Special issue on distributed Shared-Memory Systems, Match 1999.

[7] IBM, Inc. "28.4G Packet Rooting Switch", Networking Technology Data sheets, http://www.chii)s.ibm.com/techlib/products/commun/datasheets.html

[8] D. Kulkami, M. Sturnm, "Linear loop transformations in optimizing compilers for parallel machines" in The australian computer journal, pp.41-50, May 1995.

[9] S. Meftali, F. Gharsalli, F. Rousseau, A.A. Jerraya, "An Optimal Memory Allocation for Application-Specific Multiprocessor System-on-Chip", Proc. of ISSS 2001.

[10] P. R. Panda, N. Dutt, A. Nicolau, Memory Issues in Embedded Systems-on-chip: Optimization and exploration, Kluwer Academic Publishers, 1999.

[11] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, J. D. Owens, "Memory Access Scheduling", Proc. of ISCA 2000.

[12] K. Svarstad, G. Nicolescu, A. A. Jerraya, "A Model for Describing Communication between Aggregate Objects in the Specification and Design of Embedded Systems", DATE'2001.

[13] M. Wolf, "improving locality and parallelism in nested loops", Ph.D dissertation, Stanford University, USA, August 92.