Communication Synthesis in a multiprocessor environment

Claudiu Zissulescu, Bart Kienhuis, Ed Deprettere

Leiden Embedded Research Center,
Leiden Institute of Advanced Computer Science (LIACS),
Leiden University, The Netherlands
{claus,kienhuis,edd} @liacs.nl
Presented at
Field-Programmable Logic and Applications conference (FPL'05)
Tampere, Finland August 24 - 26, 2005

Abstract. At Leiden University, we are developing a design methodology that allows for fast mapping of nested-loop applications (e.g. DSP, Imaging, or Multi-Media) written in a subset of Matlab onto reconfigurable devices. This design methodology is implemented into a tool chain that we call COMPAAN/LAURA [8]. This methodology generates a process network in which the inter-process communication takes place in a point-to-point fashion. Four types of point-to-point inter-processor communication exist in the PN. Two of them use a FIFO like communication and the other two use a cache like memory to exchange data. In this paper, we investigate the realizations for the four communication types and show that point-to-point communication at the level of scalars can be realized automatically and very efficiently in today's FPGAs.

1 Introduction

To better exploit the reconfigurable hardware devices that are coming to market, a number of technologies are developed to handle billions of transistors available in these new chips. A key idea in these technologies is decoupling the communication from computation. This decoupling allows the IP cores (the computation part) and the interconnect (the communication part) to be design separately [5]. Respecting this design concept, we are developing a design methodology that allows fast mapping of nested-loop applications (e.g. DSP, Imaging, or Multi-Media) written in a subset of Matlab onto reconfigurable devices. This design methodology is implemented into a tool chain that we call COMPAAN/LAURA [8].

The COMPAAN tool analyzes the Matlab application and derives automatically a parallel representation, expressed as a Process Network (PN). A PN consists of concurrent processes that are interconnected via asynchronous FIFOs. The control of the input Matlab program is distributed over the processes and the memory is distributed over the FIFOs. The LAURA tool synthesizes a network of hardware processors from the given PN. A key operation in LAURA is to generate the proper hardware communication mechanism for the point to point inter-processors communication, which may be a different communication structure than the PN communication FIFO model.

Our tool flow has been developed for data-flow algorithms, having communication at the level of scalars (e.g. bytes or words). The communication topology of the PN is static, derived at compile time. To realize the inter-processor communication, we use point-to-point communication mechanism. Employing busses and/or complex Networks-on-Chips (NoCs) [7, 4] for the communication is not feasible due to the delays in the routing process and the usage of large packets instead of scalars in the communication protocol.

As we found out in [10], four types of point-to-point inter-processor communication exist in the PN we generate. Two of them use a FIFO like communication and the other two use a cache like memory to exchange data. In this paper, we investigate the realizations for the four communication types and discuss the effectiveness of the realizations. The rest of the paper is organized as follows: first, we present the COMPAAN/LAURA flow in order to understand how the hardware mapping is done by LAURA. Next, we address the communication channel generation and propose an approach to solve each communication type. We finish this paper with a discussion over the merits and improvements of these approaches in the context of our tool chain.

1.1 COMPAAN/LAURA design flow

The process networks we consider in this paper are derived using the COMPAAN tool chain. COMPAAN takes as input parameterized static nested loop programs written in Matlab and converts this code to process networks. An intermediate step is transformation of the initial Matlab code into single assignment code (SAC) using exact data flow analysis [3]. The last tool of the flow, called LAURA, is used to generate a VHDL description of an architecture from a PN description. During this step, each process of the PN is mapped to an abstract architectural model called *Virtual Processor*. Each virtual processor consists of three distinct components:

- An *Execute Unit*, which is the computational part of the virtual processor. This unit wraps in an IP core that implements the functionality of the process. Its interface consists of a number of *Input arguments* and *Output arguments*.
- A *Read Unit*, which is responsible for assigning valid tokens to the input arguments of the Execute Unit. Since there are more input ports than arguments, the Read Unit has to select at run-time from which Channel to read tokens using a control program that is derived by COMPAAN.
- And a *Write Unit*, which is responsible for distributing the results of the Execute Unit to different Channels. A write operation can execute only when all the output arguments of the Execute Unit are available for the Write Unit. Similarly to the Read Unit, the Write Unit has to select a channel at run-time to write tokens into, using a control program that is derived by COMPAAN.

The applications targeted by COMPAAN are usually data-flow intensive, requiring large computational power. Therefore, an important issue in LAURA is the derivation of efficient communication structures in hardware and is the focus of this paper. Initially, COMPAAN finds in the input Matlab file, all the possible producer-consumer pairs. At that level the communication between two processes is done using a multidimensional

array, represented as a polytope. To select the type of communication a linearization procedure is employed by the COMPAAN tool which selects the right type of communication channel.

2 Communication Generation

In a hardware network generated by LAURA, each processor executes an internal control program at both the Read and Write Units. This program describes a local schedule in terms of Execute Unit executions. At each execution, also refer to as an *iteration*, a Read Unit reads data from a Channel and a Write Unit writes data to a Channel. In the original Matlab code, the Channel represents the communication on a n-Dimensional array (e.g., a[i,j]). This array is replaced by 1-D array by our tool chain in the linearization step.

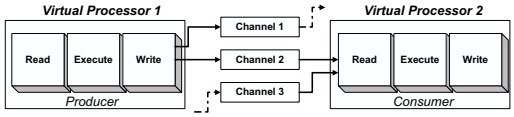

Fig. 1. A Producer-Consumer pair

Figure 1 depicts a classical producer consumer pair. A *Virtual Processor 1* sends data to the second processor called *Virtual Processor 2* via the *Channel 2*. The channel represents the data dependency between a Read Unit and a Write Unit. This relation is given by a *Mapping* function. The linearization step replaces the addressing of an array with relative addressing scheme based on *put* and *get* primitives. Usually, the derived communication channel is a FIFO, however, there are cases in which a FIFO is not sufficient to linearize a n-dimensional array [10]. We have found that four types of communication can be distinguished as given in Figure 2. They result from the *ordering* of the iterations at the Producer and the Consumer processes and the existence of *multiplicity* for a given token, which means that a token that is sent by Producer is read more than once at the Consumer side. Hence, depending on the order and existence of multiplicity, an arbitrary communication channel belongs to one of four disjoint classes: *in-order without multiplicity (IOM-), in-order with multiplicity (IOM+), out-of-order without multiplicity (OOM-)*, and *out-of-order with multiplicity (OOM+)*. For each class an adequate communication mechanism needs to be efficiently synthesized in hardware in terms of cycles per operation, area and speed.

From experience [10], we know that on average the following distribution can be expected over the various communication types: type IOM- (80%), IOM+ (10%), OOM- (9%), OOM+ (1%). Type IOM- together with type IOM+, result in that 90% of the communication channels, require a FIFO buffer to realize the communication. In the remaining 10% of the cases, a more complex Reordering Channel is needed.

2.1 In Order communication (IOM-)

In the *In Order* communication (IOM-) case, the Producer writes data in the Channel in the same order as the Consumer reads from the Channel. Therefore, this Channel is

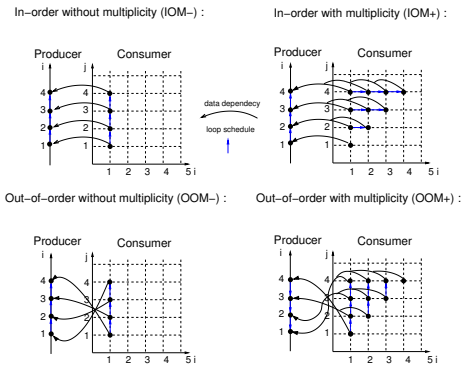

Fig. 2. The four cases of communication between Producer and Consumer

implemented in hardware using a FIFO buffer. It is accessed using the two primitives *put* (implemented in the Write unit) and *get* (implemented in the Read unit). Because highly optimized implementations of FIFO buffers exist for today's FPGAs, it takes each primitive only a single cycle to write data or to read data from a Channel. A hardware FIFO has finite memory, thus both primitives are *blocking*, e.g. they halt a processor when no data is available in a FIFO or when a FIFO is full. Finding a lower bound on a Channel is a hard problem in PNs and it is outside the scope of this paper, although a small discussion is given in Section 3.

2.2 In Order with Multiplicity communication (IOM+)

In the *In Order with Multiplicity* Communication (IOM+) case, the order data is produced is the same as the order in which data is consumed. However, some data is consumed more than once, breaking the communication model of a FIFO where a get operation is destructive. In this model, the life-time of a token needs to be taken into account. Only at the end of the life-time of a token, the token can be released from the FIFO. While the *put* primitive remains the same as in the case of IOM-, we added a new communication primitive which we called the *peek* primitive. The *peek* primitive fetches data from a FIFO buffer without destroying it. To destroy the current FIFO data a release control is synthesized in the Consumer Read Unit. Also, the output of the FIFO is registered by the *Multiplicity Register* which is controlled by the release control. A *peek* is only reading the contents of the register, while a *get* operation is reading a new value from the FIFO buffer and place it in the Multiplicity Register. The control that determines the life-time of a token is expressed in the same way the control programs in the Read and Write Unit are expressed.

2.3 Out of Order communication (OOM-)

In the *Out-of-Order* communication (OOM-) case, a Consumer reads data in a different order it has been written by the Producer. Hence, the communication channel allows a Consumer to fetch data in the order it expect it. We refer to this kind of Channel,

which allows for run-time reordering of tokens, as a *Reorder Channel*. The main elements of this Reorder Channel is the reorder memory and the tagging of tokens that are written/read to/from the reordering memory. Each token needs to be tagged to allow the Consumer Process to request particular tokens in the order given by its local schedule. The tag computation takes place in both parts involved in the transaction, i.e., the Producer side and the Consumer side.

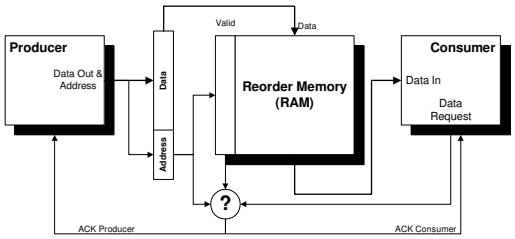

Fig. 3. The organization of the Reorder channel

In Figure 3, the organization of the Reorder Channel is given. The main element is the random access memory, called the *Reorder Memory*. The figure also shows a Producer that wants to write tokens to the Channel and a Consumer that wants to read tokens. A token (Data) that is written by the Producer, is temporarily stored in a register together with an address. This address is calculated by the tag generator of the Producer. Each token that is stored in memory has a *valid bit*, which indicates that a particular location (address) contains valid data. If the valid bit is set, the Producer is not allowed to write data and is completely stalled until the address becomes available again (ACK Producer). Otherwise, the Producer writes the temporarily stored data into the memory and sets the valid bit. At the other side, the Consumer places a request command to the Reordering Channel for a particular location given by an address. If the requested location contains valid data, the Consumer receives an acknowledge signal(ACK Consumer), and, at the same time, the desired data. If the location does not contain valid data, the Consumer stalls until valid data becomes available. Given the organization of the Reordering Channel in Figure 3, two issues determine the design. One is related with the complexity of the tag generation and the other one is related with the performance of the reorder channel in terms of clock cycles.

2.4 Tag generation

In the generation of tags, we take advantages of the fact that we operate on polytopes within COMPAAN. This leads to two different approach we can use to generate the tag. One approach is based on the Ehrhart enumeration theory [2]. Using this theory, we obtain a pseudo-polynomial expression that gives an unique integer value for each point enclosed by the polytope. This approach has been successfully explored in software in [9]. However, this approach is not suitable for hardware implementation due to the complexity of the obtained pseudo-polynomial expression. In our example in Figure 4,

the pseudo polynomial for the polytope enclosing the producer points is given by the follow expression: $(-1/4) * i^2 + (N - 5/2) * i + j + [-1, 5/2]_i$. In this expression, the pseudo polynomial term $[-1, 5/2]_i$ indicates that when the evaluation of $mod(i, 2)$ is equal to zero, the value -1 is selected; otherwise the value 5/2 is selected in the polynomial.

In the second approach, we relax the shape that encloses the producer polytope to a hyper-rectangular shape. We call this hyper-rectangular shape the *Bounding Box*. For a Bounding Box, we can make use of classical linearization to convert a n-dimension rectangle to an one-dimensional array [1, 6]. Find the Bounding Box that best encloses the polytope is a minimization problem that we solve using integer linear programming. The improvement over the Ehrhart approach is that each Bounding Box can be addressed using a simple polynomial that can be implemented efficiently in hardware. The tag for a token is obtained as a function of the iterators of a processor and has the form $tag = \sum_{k=1}^{N} c_k * x_k + x_0 + c_0$ where c_k represents a constant, x_k is an iteration space index. Each tag becomes an address for a RAM memory.

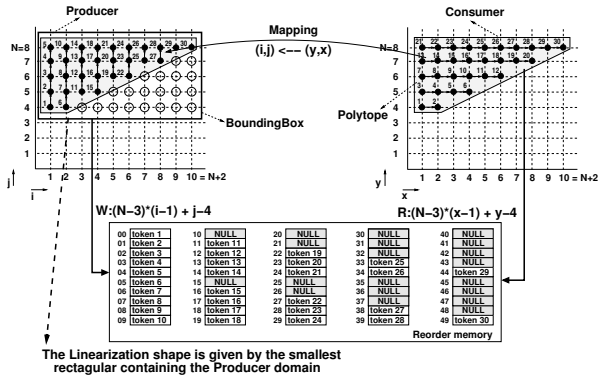

Fig. 4. The Linear realization of the reorder memory

If we extract the Bounding Box at the Producer side in Figure 4, we obtain the *write address*, which is $(N - 3) * (i - 1) + j - 4$. From the address generated by the write address, the Consumer has to read tokens in the order it desires. It therefore requires a *read address* to access the tokens. Since there is a well defined mathematical relationship between the Producer and Consumer as given by the Mapping, we can obtain the read address by mathematically composing the write address with the mapping function. Thus, the parameterized read address is equal to $(N - 3) * (x - i) + y - 4$. In this case, reordering is induced by the fact that the points in the Consumer polytope are accessed in a different order then the Producer has written than to the Reorder Channel.

The down side of the Bounding Box approach is that the memory allocated for the Reorder Memory is larger than needed when compared with the Ehrhart approach. In Figure 4, the content of the Reorder Memory is given. It shows at which address a particular token is written. For example, the token from location $(6, 7)$ is written at address 28. This is the 23rd token written by the Producer. This token is read by the Consumer

at location $(6, 7)$ as the 18th token read. From the content of the reordering memory, we observe that memory locations exist that are never written or read during the execution of a network, as given by the "Nulls". In the case of an Ehrhart generated address, these "Nulls" would not exist at the expense of more complex address polynomials that are difficult to be implemented efficiently in hardware.

2.5 Communication protocol

The Reordering Channel implements a particular communication protocol that involves writing and reading a token. A write operation uses only the *put* communication primitive, like we would have done in case of a FIFO Channel. This primitive provides to the Reorder Channel a token that consists of a tag and data. If place is available at the location given by the tag, the token is written; otherwise the Producer is stalled realizing a blocking write. A read operation consists of three communication primitives; a check, a peek and a get. Together they realize the reading of data from the Reordering Channel in any desired order. Each primitive performs a particular task:

- *check* inquires if specific data is present in the Reorder Memory. The desired tag given to the Reordering Channel as a Request. If that data is present at the address given by the tag, the Reordering Channel sends an acknowledge signal to the Consumer; otherwise the Consumer blocks realizing a blocking read.
- *get* reads a token from the requested address and set the valid bit to false indicated that the location if available for writing again. A get operation is therefore destructive.
- *peek* reads a token from the requested address but keeps the valid bit high to indicated that the location is still needed for reading. Using the peek operation, the life-time of a token is controlled.

In the OOM- communication type, we only use the check and get primitives to read data since no multiplicity is involved. A memory location can be immediately released when reading a token.

2.6 Out of Order with multiplicity communication (OOM+)

In the *Out-of-Order with Multiplicity* communication (OOM+) case, a channel has the same characteristics as the OOM- case. Additional release logic is however added at the Consumer side to keep track of the life-time of tokens. Consequently, the OOM+ communication type also uses the peek primitive if a token does not yet need to be released from memory. The get primitive is used when the release logic indicates that the life-time of a token has come to its end.

3 Memory Allocation Schemes

When implementing a Channel, whether it is a FIFO or a Reordering Channel, we need to determine a lower bound on the amount of tokens that can be stored in a Channel, without causing a deadlock to occur. Finding this lower bound in process networks

is in general a difficult problem. A process network is specified in terms of partial orderings between a producer and consumer, i.e., the Producer/Consumer pair. To find a lower bound on a Channel requires, however, a total order on the execution of the processes in the network. Many different total orders exist; leading to different trade-offs in evaluation speed and memory requirement. A total order for a PN can be obtained by scheduling or by doing a run-time evaluation of a process network. Nevertheless, we would like to avoid both methods. The scheduling interferes with the notion of distributed control and the run-time evaluation is not a compile time analysis that can be performed as part of the COMPAAN tool. Also, both the run-time and schedule approach cannot handle the parameterized nature of the PNs we derive. Instead, we use compile time allocation schemes. Without going to deep into detail, we currently distinguish two compile time allocation schemes:

- *Memory Allocation without releasing the memory (SAC)*
  This memory allocation scheme is derived from the single assignment data allocation found by one of the intermediate step of our tool chain. In this scheme, a memory location is assigned data only once. Hence a Consumer does not need to release a memory location after it has consumed data from memory. The memory required on a Channel is the dimension needed to accommodate all produced data, which is given by the Bounding Box.
- *Memory Allocation With releasing the memory (SEQ)*
  This memory allocation scheme is derived from the original data allocation found in the sequential input Matlab program. In this scheme, memory locations are re-used and hence each read or write operation on the Channel has to check if a memory location is either free or contains valid data.

The SAC allocation scheme results in the fastest execution of a PN as the network can run with maximum parallelism. Any other allocation scheme, like the SEQ allocation scheme, may restrict the network parallelism to a more sequential execution scheme, as explicit checks need to be performed on the validity of data. The guarantee that the two described allocation schemes do indeed not introduce deadlock due to under dimensioned communication buffers, is given by the fact that the allocation schemes are able to run in bounded amount of memory in the sequential execution. We know from simulation that tighter lower bounds are possible. Further research is needed to develop techniques to derive these tighter bounds at compile time. We expect that results from research in memory optimization for sequential code can also be applied to improve the memory allocation in PNs.

4 Hardware realization

All four communication types can be automatically generated by LAURA and have been implemented on a VirtexII-6000 platform from Xilinx. A FIFO channel is realized using different types of memory at compile time depending on the calculated size. If less than 1024 bits are required, we use RAM16x1D memories, otherwise we use RAMB16 memory blocks. If only a single location is required, we simply instantiate a FIFO

channel that uses only a register. In case of the Reorder Channels, we always use Block RAMs to realize the Reordering Memory as a full dual port memory is required.

The hardware realization of a FIFO buffer is the fastest and most efficient one in terms of cycles per operation, as it requires one cycle per read or write operation. In the case of a Reordering Channel, the SAC implementation requires still one cycle per write operation as the availability of a location does not need to be checked. However, the read operation requires three cycles to read one scalar from the channel. The SEQ implementation of the Reorder Channel requires up to 2 cycles for a write and 3 cycles for a read operation. In both the read and write operation, the availability of a location needs to be checked.

5 Example

To highlight the different characteristics of the communication channels, we looked at the QR algorithm [11]. This algorithm requires 1 IOM+ and 10 IOM- channels. However, we can implement these channels also using 1 OOM+ and 10 OOM- channels. This highlights the differences in the hardware implementation and performance of the different channels and the benefits of selecting the the right channel type at compile time. In both cases, the lower bounds on the channels are determined at compile time using the SAC allocation scheme.

Compile time estimations	Memory location	154	Reorder Channel	Cycles	258	FIFO Channel	Cycles	128
	Memory Size	4928 bits		RAM16	11		RAM16x1D	320
	Bit width	32		Memory Size	180224 bits		Memory Size	5120 bits
				Slices	1771		Slices	890
				Frequency	100Mhz		Frequency	102Mhz

Table 1. Experimental results for various hardware channels.

From Table 1, we observe that the Reorder Channels are more inefficient than FIFO channels in terms of usage of FPGA memories. The Reorder Memory is realized using only Select Block RAMs (RAMB16). For the 11 channels, 11 RAMB16 memory blocks allocated, which is 180224 bits in memory on the FPGA. For each Reordering channel, we need to allocation at least a RAMB16 block which can accommodate 512 tokens of 32 bit, even if we only need space for 10 tokens. Hence, it is difficult to use the memory block efficiently; there can be quite some spill. On the other hand, for a FIFO channel, we can select between Select Block RAMs or RAM16x1D memories, depending on the required channel size. In this particular case, we use 320 RAM16x1D memory blocks and the memory used for the FIFO implementation is only slightly larger than the calculated size using the SAC allocation scheme, i.e., 5120 bits versus the 4928 bits calculated. Since the SAC allocation scheme is used, the Reorder Channels implement a fast write operation. Nevertheless, it takes twice the number of cycles to run the QR algorithm (258 cycles) compared to when using FIFO channels (128 cycles). That a read operation requires 3 cycles, has a big influence on the execution speed of the algorithm. The number of slices used to implement the QR algorithm with FIFOs is 890 and with Reordering Channels is 1771. The Reordering Channel requires twice as many slices

due to the hardware tag generators and the use of RAMB16 memories. Both the execution speed and number of slices used show that it is indeed very important to select the proper communication type in order to evaluate an algorithm as fast as possible with the least amount of resources.

6 Conclusions

The data-flow intensive applications, we target with COMPAAN, require large computational power and it is important to derive efficient communication structures in hardware. Four point-to-point communication types are distinguished in COMPAAN and we have shown that for each of them, we can derive efficient hardware in LAURA. Two types use hardware FIFO implementations and two types use a Reorder Channel implementation, as we presented in this paper. From a case study, we have shown that a FIFO implementation is the most optimal implementation of a Channel from any point of view (throughput, hardware resources, memory usage). Because a FIFO can read and write in a single cycle it can keep up with the maximum data flow through our virtual processors. From experience, we know that on average 90% of the channels in an application can be realized with FIFO buffers. In the remaining 10%, we employ the Reordering Channel implementation. As future work, we are interested to further optimize the Reorder Channel and porting it to other FPGA architectures, e.g. Altera. Also, we want to investigate new compile time allocation schemes to obtain tighter lower bounds on the memory needed in the communication channels.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley Publishing Co., 1986.
2. E. Ehrhart. *Polynômes arithmétiques et Méthode des Polyédres en Combinatoire*. Birkhäuser Verlag, Basel, international series of numerical mathematics vol. 35 edition, 1977.
3. P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.
4. P. Guerrier and A. Greiner. A generic architecture for on-chip packet-switched interconnections. In *DATE '00: Proceedings of the conference on Design, automation and test in Europe*, pages 250–256. ACM Press, 2000.
5. K. Keutzer, S. Malik, R. Newton, J. Rabaey, and A. L. Sangiovanni-Vincentelli. System level design: Orthogonalization of concerns and platform–based design. *IEEE Trans. on CAD*, 2000.
6. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, Inc., 1997.
7. E. Rijpkema, K. Goossens, A. adulescu, J. van Meerbergen, P. Wielage, and E. Waterlander. Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip, 2003.
8. T. Stefanov, C. Zissulescu, A. Turjan, B. Kienhuis, and E. Deprettere. System design using kahn process networks: The compaan/laura approach. In *Proceedings of DATE2004*, Paris, France, Feb 16 – 20 2004.

9. A. Turjan, B. Kienhuis, and E. Deprettere. A compile time based approach for solving out-of-order communication in Kahn Process Networks. In *Proceedings of IEEE 13th International Conference on Application-specific Systems, Architectures and Processors*, July 17-19 2002.
10. A. Turjan, B. Kienhuis, and E. Deprettere. An Integer Linear Programming Approach to Classify Communication in Process Networks. In *8th International Workshop on Software and Compilers for Embedded Systems (SCOPES)*, Amsterdam, Sept. 2–3 2004.
11. R. Walke and R. Smith. 20 gflops qr processor on a xilinx virtex-e fpga. In *Advanced Signal Processing Algorithms, Architectures, and Implementations X*, volume 4116, 2000.