Chapter 10

Automatic Synthesis of Efficient Interfaces for Compiled Regular Architectures

Steven Derrien
Anne-Claire Guillou
Patrice Quinton
Charles Wagner
Irisa, Campus de Beaulieu
35042 Rennes Cedex, France
email: {sderrien,aguillou,quinton,wagner}@irisa.fr

Tanguy Risset
LIP, ENS Lyon, 46 allée d'Italie
69364 Lyon Cedex 07, France
email: {trisset}@ens-lyon.fr

10.1 Introduction

The technological road map for embedded system design foresees important changes in the design methodologies. IP *based* designs and *platform based* designs [30] are becoming mandatory because the complexity of the designs increases very fast, and time to market requirements are shorter and shorter. However, it is well-known that technology improvements do not affect uniformly all elements of a chip: memories, buses and power supply integration do not follow Moore's law. Hence, the main focus in hardware design has moved from performance and parallelism to other concerns such as the optimization of memory size, hierarchy, and traffic, the elimination of the bus bandwidth bottleneck, and the minimization of power consumption. Power-aware design can be considered as today's most important problem, but with the emergence of *network on chip* [7], interfacing IPs efficiently becomes a major issue. The present chapter addresses this latter problem.

Many designs rely on bus protocols to interface IP, for example, AMBA or CORE-connected bus. This tendency is amplified by the spreading of the *platform based design* methodology which imposes the use of particular buses on the chip. But the gap between the design clock cycles and bus clock cycles increases, and therefore, the speed at which data can be fed into a design becomes the major bottleneck for performance. Hence, a good design must be delivered with an interface protocol that uses the bus bandwidth efficiently. Such an interface is most often designed by hand, but clearly, automatic generation of interfaces must be considered. This research is mainly concerned with efficient bus protocol interface definition for highly parallel designs. Note here that some attempts to plug FPGA chips directly to memories have been made [16]. In such a framework, the interface synthesis may be very different from the one presented here. However, one part of the present work – how to extract interface information from the high-level specifications, – remains valid, even if it has to be retargeted.

To be complete, the analysis of technological evolution should include trends in the electronic design automation tools (EDA). Currently the software technology provides satisfying tools for logic synthesis (i.e., synthesis from RTL specification.) But trends lead towards providing higher level design methodologies, using new languages (SystemC for instance) and new methodologies (UML specifications for instance). The research presented here belongs to a specific domain which aims at compiling loops to hardware [26, 13, 10, 2]. In this field, interfacing is also a major issue because applications operate on large data sets (usually streams in signal processing or images in multimedia) which must be efficiently brought onto the chip. Again, design automation is a major issue and the approach proposed in this chapter focus on automatic interface design for architectures compiled from loop nest specifications.

Based on the work done around the MMAlpha tool [10, 8], we propose a solution to this problem for linear regular arrays. In this chapter, we introduce the concept of *application interface* which can be seen as the application dependent part of the interface for linear systolic arrays. As all highly pipelined designs share many properties, it is possible to define a *generic application interface*, i.e. an interface skeleton that is valid for all linear arrays and that can be easily parameterized for each application implemented. The experiments we report in this chapter are oriented towards a FPGA platform, but the concepts presented can be used for interfacing IPs on a SoC, provided the IP has some features that we will describe hereafter.

The underlying interface architecture that we consider in this chapter is composed of a bus (with fixed bandwidth and throughput, possibly including a faster DMA mode) and of a Fifo that interconnects the bus and the application. The Fifo allows data to be buffered when an interruption occurs on the bus. The bus and the Fifo form what we call the *hardware interface*. On top of the hardware interface is built an *application interface* whose rôle is to rearrange the data between the hardware interface and the application. This part of the interface is application dependent and can be automatically produced by the same kind of tools that generates the hardware of the application.

This chapter is organized as follows. In section 10.2, we present briefly the MMAlpha environment. After a short description in section 10.3 of the DLMS application which serves as an illustration throughout the chapter, we explain in section 10.4 the model of our application architecture. Section 10.5 details the various elements of the interface we target. In section 10.6, we describe how data transfers are structured in phases and patterns in order to obtain efficient communications. The generation of the interface, both software and hardware, is presented in section 10.7. We then describe in section 10.8 the use of our interface generator to implement automatically a DLMS filter on a FPGA board. Finally, we present in section 10.9 related work and we conclude in section 10.10

10.2 The MMAlpha system

MMAlpha is a programming environment for transforming Alpha programs into parallel architectures based on research on automatic systolic synthesis [25]. Alpha [19, 15] is a functional language that was designed to express both recurrence equation specifications and hardware descriptions of systolic arrays. MMAlpha has been used to design applications pertaining to signal processing, multimedia, and bio-informatics [21, 20].

MMAlpha targets both Asics and FPGAs. The final output of the compilation process is a synthesizable VHDL program that can be compiled with vendor tools. The interest of targeting FPGAs stems from the observation that regular designs are well-suited to the internal architecture of most FPGAs which are made of locally connected configurable logic blocs (CLB).

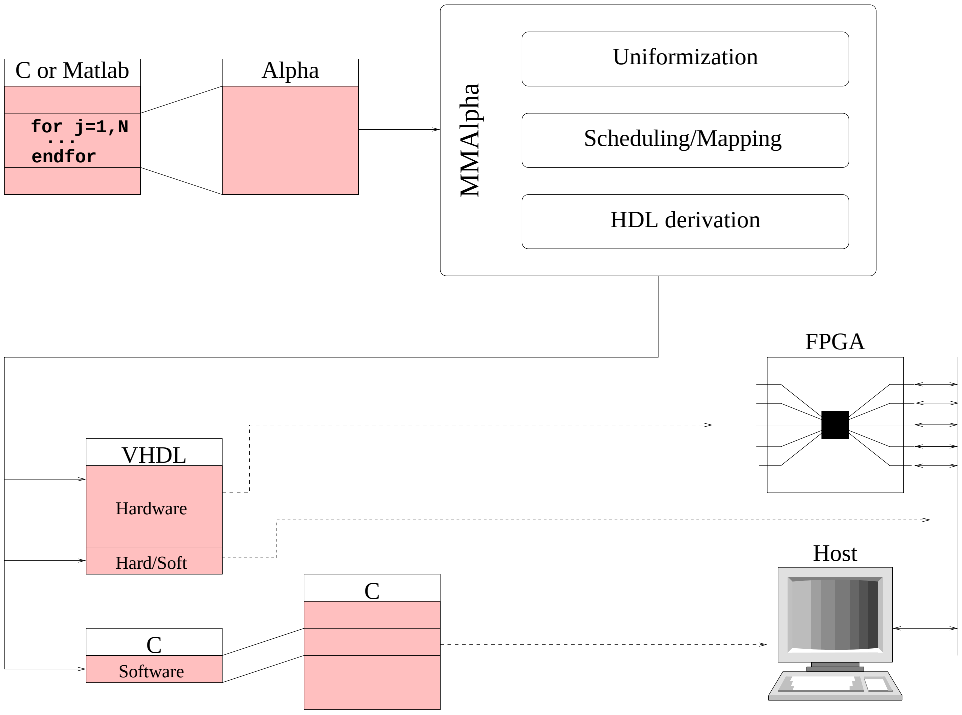

Figure 10.1 Design flow with MMAlpha. Square shaded boxes represent programs of various languages, and round boxes represent transformations.

The design flow with MMAlpha is illustrated in Fig. 10.1. A computation intensive part of a program (usually a loop) is translated into Alpha and is then transformed using MMAlpha into a parallel hardware description at register transfer level (RTL). This RTL description comprises three parts:

- The hardware part, written in VHDL, represents the implementation of the loop on the FPGA.
- The software part is what replaces the loop in the original program.
- The hardware/software part handles the communication of data between the host and the FPGA.

Fig. 10.2 shows an example of Alpha program for a delayed least mean square adaptive filter that we shall describe in details in section 10.3. Such a program, called a *system*, represents a set of recurrence equations that allows iterative algorithms to be easily represented. Here, this system, named `firr`, is parameterized by the size `N` of the filter, the number `D` of delays, and the number of input samples `M`. It has two inputs, `x` and `d`, and produces one output `y`. Local variables `W`, `Y` and `E` define intermediate equations necessary to compute `y`.

Each equation of this program is an indexed expression whose index space is an integral polyhedron, that is to say, a set of integer coordinate points delimited by linear inequalities. Expressions are indexed by affine combination of indices and of parameters. Operators, e.g. `+`, `*`, etc., combine expressions as collections of data. Case expressions (see for example the definition of `W`) allow variables to be defined by different expressions on different polyhedral regions of the index space.

The synthesis process consists of applying a set of transformations, such as the ones mentioned in Fig. 10.1: uniformization allows affine index expressions to be replaced by translations thus removing potential broadcasts of the target architecture; scheduling and mapping assign to each expression an execution time and a processor; finally, a RTL description is generated by translating the scheduled and placed Alpha code in a HDL language such as VHDL.

10.3 The DLMS example

In this section, we detail the delayed least mean square algorithm (DLMS) for channel error correction in signal processing applications. This algorithm is used throughout this chapter to illustrate our methods.

Least mean squares adaptive filters are commonly used in signal processing applications such as echo cancellation, system identification, speech coding and channel equalization [11]. Unlike fixed coefficient Finite Impulse Response (FIR) or Infinite Impulse Response (IIR) digital filters, the coefficients of adaptive digital filters such as the LMS filter are adapted at each iteration to obtain better convergence properties. It is well-known that recursive or adaptive digital filters are difficult to pipeline due to the presence of a feedback loop. However, it is possible to obtain a pipelined implementation by inserting delays in the recursive loop of the coefficient update part. The corresponding algorithm is thus called a *delayed least means square* (DLMS) algorithm.

By assuming that the adaptive digital filter is a FIR filter whose impulse response is denoted by $w_i(n)$, the output signal $y(n)$ is given by:

```
system firr: {N,M,D | 3<=N<=(D-1,M-D-1)}
  (x : {n | 1<=n<=M} of real;
   d : {n | N<=n<=M} of real)
returns
  (y : {n | N<=n<=M} of real);
var
  W : {n,i | N<=n<=M; 0<=i<=N-1} of real;
  Y : {n,i | N<=n<=M; -1<=i<=N-1} of real;
  E : {n | N<=n<=M} of real;
  mu : integer;
let
 mu[] = 0.5[];
 W[n,i] =
   case
    { | n<=N+D-1} : 0[];
    { | N+D<=n} : W[n-1,i] + E[n-D] * x[n-i-D];
   esac;
 Y[n,i] =
   case
    { | i=-1} : 0[];
    { | 0<=i} : Y[n,i-1] + mu[] * W[n,i] * x[n-i];
   esac;
  E[n] = d[n] - y[n];
  y[n] = Y[n,N-1];
tel;
```

Figure 10.2 Alpha program for a DLMS filter

$$y(n) = \boldsymbol{x}^T(n)\boldsymbol{w}(n) = \sum_{i=0}^{N-1} x(n-i)w_i(n) \quad , \tag{1}$$

with $x(0) = 0$ and $\boldsymbol{w}(0) = 0$ (bold variables denotes N-vectors: $\boldsymbol{x}(n) = (x(n-N-1), \ldots, x(n))$ and $\boldsymbol{w}(n) = (w_0(n), \ldots, w_{N-1}(n))$). The weight update equations of the DLMS [12] are given by:

$$\boldsymbol{w}(n+1) = \boldsymbol{w}(n) + \mu\ e(n-D)\boldsymbol{x}(n-D) \quad , \tag{2}$$

$$e(n) = d(n) - y(n) \quad , \tag{3}$$

where $d(n)$ is the desired signal.

A possible VLSI implementation of the DLMS [12] is represented in Fig. 10.3. The RTL description of this architecture can be derived automatically from program of Fig. 10.2 with the MMAlpha software as shown in [10].

At the end of the synthesis process, the architecture presented in Fig. 10.3 is expressed as an AlpHard program, which is a subset of Alpha, and the mapping

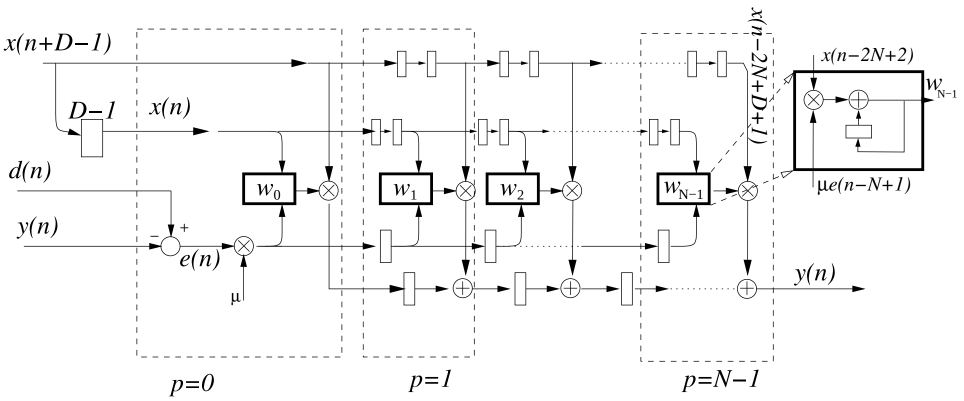

Figure 10.3 Snapshot (at $t = n - N + D$) of the architecture obtained for the DLMS after MMAlpha design process. N is the number of taps of the filter, D is the number of delays in the feedback loop.

between the functional specification and the hardware implementation is obtained by means of the program of Fig. 10.4. More precisely, the input flow `x` and the coefficient vector `d` of the initial `firr` algorithm are mapped to new variables `x_mirr1`, `x_mirr2`, and `d_mirr1`, and the architecture itself is represented by an instantiation of another Alpha program called `firrModule` (by a `use` statement), which returns an output stream `Y1`. This stream is assigned to the output variable `y` of program `firr`. All inputs and outputs of `firrModule` are indexed by `t` and `p` which represent respectively the time and the processor number to which these streams are assigned.

10.4 Application architecture model

This section models the interface we want to synthesize. We first state the assumptions that we make regarding the type of application hardware that we want to interface. Then we detail the information which is needed for interfacing correctly the application architecture (e.g. `firrModule` in the DLMS application) with its host architecture. Then we abstract this architecture by a number of features that will constitute the input to interface generation.

10.4.1 Assumptions

Our assumptions regarding the application architecture concern four aspects: a virtual clock, the model of linear array, inputs and outputs, and the bit width of streams.

Virtual clock.

The application architecture is a *globally synchronous digital circuit* in which all registers are controlled by a *common virtual clock*. This virtual clock regulates the operation of the architecture, and can be frozen if, for instance, the host is not ready to send input data or to read output data. This assumption significantly reduces the control complexity inside the interface. Indeed, the designer can assume that as soon as the clock of the architecture is running, input data arrive

```
system firr :  {N,M,D | 3<=N<=(M-D-1,D-1)}
               (x : {n | 1<=n<=M} of integer[S,16];
                d : {n | N<=n<=M} of integer[S,16])
       returns (y : {n | N<=n<=M} of integer[S,16]);
var
  d_mirr1 : {t,p | D<=t<=-N+M; p=0} of integer[S,16];
  y_mirr1 : {t,p | D<=t<=-N+M; p=0} of integer[S,16];
  x_mirr1 : {t,p | -N+D+1<=t<=-N+M; p=0} of integer[S,16];
  x_mirr2 : {t,p | -N+2<=t<=-N+M+1; p=0} of integer[S,16];
  Y1 : {t,p | N<=t<=M; p=N-1} of integer[S,16];
let
  y_mirr1[t,p] = y[t+N-D];
  d_mirr1[t,p] = d[t+N-D];
  x_mirr2[t,p] = x[t+N-1];
  x_mirr1[t,p] = x[t+N-D];
  y[n] = Y1[n,N-1];
  use  firrModule[N,M,D] (d_mirr1, y_mirr1, x_mirr1, x_mirr2)
                returns  (Y1) ;
tel;
```

Figure 10.4 The part of the AlpHard program (`firr` system, also called AlpHard *interface*) which maps the functional specification (input `x` and `d`, result `y`) to the architecture (`firrModule` system also called AlpHard *module*). This program has three parameters: `N` is the number of taps of the filter, `D` is the number of delays along the feedback loop, and `M` is the number of input samples of the filter (for simulation purposes.) This system contains the information about the date and place where data should be entered (here for example, input in the first processor: $p = 0$, and output in the last processor: $p = N - 1$).

as needed, and output data are captured by the bus when they are produced. In our experimental designs which targets FPGA chips, the virtual clock is naturally implemented using the *clock enable* signal of the FPGA. We also assume that the operation of the architecture begins on a `start` signal.

From now on, we call *virtual date* of a computation the number of virtual clock cycles elapsed between the computation and the starting time of the algorithm.

Linear arrays.

The architecture must be a linear (1-dimensional) array. Inputs and outputs are continuous streams of data, which means that all input or output streams have a virtual starting date and a virtual ending date, and no interruption occur in-between these dates. It should be noticed that this assumption prevents one from interfacing *partitioned arrays* [4], and *2-dimensional arrays.* In partitioned arrays, data arrives in *burst* mode, i.e. uninterrupted streams of data separated by long empty periods, and do not meet our assumption of continuous streams. However, the interface proposed in this chapter could be easily extended to cover this case because these bursts of data are known statically [5]. In MMAlpha, architectures can be generated for any 2-dimensional regular array, but in practice, interfacing a 2-dimensional array requires a more complex architecture, since more than one data has to be provided during one virtual clock cycle. This problem is beyond the scope of this chapter.

Inputs and outputs.

Any given input stream (resp. output stream) must arrive in (resp. leave) a given fixed processor, called *connection processor* of the stream. Notice that this processor may be different for each input or output stream. This assumption is most often met by the type of architecture that we are dealing with, and is easy to enforce if not.

Width of streams.

We assume that the bit width of the streams is a divisor of the data bus width. As the bus is usually 32 or 64 bit wide, the bit width of variables has to be a power of two (if it is not, the protocol will choose the smallest power of two greater than the actual bit width, hence reducing the efficiency of the interface.)

10.4.2 Information needed for the interface

If the above assumptions are met, then the interface of the architecture can be determined from the following information.

- The number of input and output streams. In our example, the `firrModule` system has four inputs (`d_mirr1`, `y_mirr1`, `x_mirr1`, `x_mirr2`) and one output (`Y1`).
- The name, bit width, connection processor, virtual starting and ending time of each stream. For instance input stream `x_mirr1` is 16 bit wide and is input in processor $p = 0$. The starting time is $t = -N + D + 1$ and the ending time is $t = -N + M + 1$.

All these informations can be extracted from the AlpHard interface shown in Fig. 10.4 (see for example the declaration of the `x_mirr1` variable in Fig. 10.4).

10.5 Interface model

The previous section describes *what* we want to interface as well as the information needed to define this interface. We now eplain in more details our interface model. Fig. 10.5 presents a typical interface architecture, in the case of the FPGA Spyder board [28]. The application hardware, here a DLMS filter circuit, is mapped on the FPGA. The host and the DLMS are interconnected by means of a PCI bus. A PCI bridge consisting of Fifos allows for a smooth synchronization between the PCI bus and the application hardware.

In this section, we review in more details the elements of this interface: the low-level interface, the application interface, and the software and hardware parts of the application interface.

10.5.1 Low-level interface, and application interface

From now on, the interface is logically divided into two parts: the *low-level interface*, and the *application interface*.

The low-level interface behaves logically as a Fifo. A parameter of the low-level interface is the Fifo *bit width* which we assume to be a power of 2. Notice that

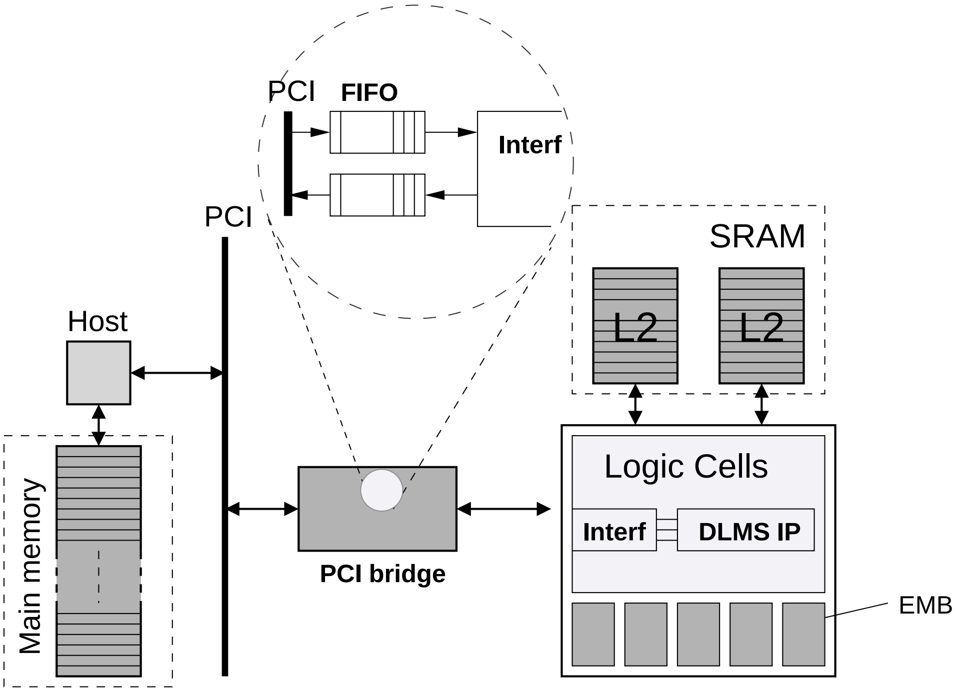

Figure 10.5 Standard architecture of the interconnection between a board and the host (taken from a Spyder board [28]), together with a logical view of the Fifo mechanism used between the bus and the application architecture.

the implementation of the low-level interface can take different forms, depending on the target platform. In the case shown in Fig. 10.5, the Fifos of the low-level interface are implemented using the memory of the Spyder board. Most commercially available FPGA boards provide a similar low-level interface [1, 3].

The *application interface* is the part of the interface that sends input data to (resp. gets output data from) the application IP from (resp. to) the host in order to implement a correct execution of the algorithm. The application interface is naturally divided into the *input interface* which sends data to the array, and the *output interface* which receives data from the array. These two parts are very similar, and from now on we only deal with the input interface.

10.5.2 Software and hardware parts

The application interface is naturally divided into a *software part* and a *hardware part*. The software part is composed of a program that sends data to the Fifo. The hardware part is more complex: it is the de-multiplexing system which gets the words out of the Fifo and sends them into the array. Of course, these two parts must be compatible, i.e. the data should be taken by the hardware part in the same order as they are produced by the software part. This is why we propose to generate the software and the hardware simultaneously from the interface AlpHard program shown in Fig. 10.4.

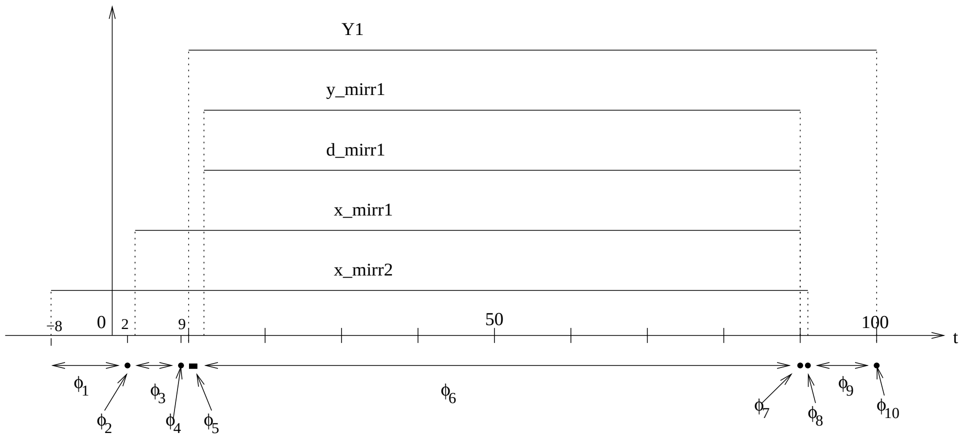

Figure 10.6 The phases of the program of Fig. 10.4 for parameters values $N = 10$, $D = 12$ and $M = 100$

10.6 Structuring streams

To generate the application interface, we structure the input and output streams using two notions: *phases* and *patterns*.

10.6.1 Phases

A phase is a sequence of successive virtual clock ticks during which all inputs and outputs of the architecture are the same. For instance, one can see in the program of Fig. 10.4 that between clock cycles $t = -N+2 = -8$ and $t = -N+D =$

2, only the `x_mirr2` stream enters the array. Hence the period of time

$$\phi_0 = \{\, t \,|\, -N + 2 \le t \le -N + D\} \tag{4}$$

is a phase.

Finding out the phases of the program of Fig. 10.4 can be done with the help of elementary computations on polyhedra. The algorithm is the following:

1. Compute the *time domain* of each input and output variable, i.e. the period of time during which this variable is alive. This is obtained by projecting the space-time domains (see Fig. 10.4) of these variables on the time index `t`. Since we assume that inputs and outputs consist of uninterrupted streams of data, for each input or output variable v, the result is a set of intervals of the form $\{t \mid l_v \le t \le u_v\}$. Note that projections of polyhedra can be computed using the Polylib library [29].

2. Sort the lower and upper bounds of these time domains. The result is an ordered list of dates $(b_i)_{i=1,\ldots}$ such that $b_i \le b_{i+1}$.

3. The phases are the intervals $\{t \mid b_i \le t < b_{i+1}\}$ for all i.

For instance, after projecting the domains of variables `x_mirr2` and `x_mirr1` in the program of Fig. 10.4, we obtain the time domains

$$\{\, t \mid -N + 2 \le t \le -N + M + 1\} \quad ,$$

and

$$\{\, t \mid -N + D + 1 \le t \le -N + M\} \quad .$$

As the calculation of all other variables starts later, the first phase is

$$\phi_0 = \{\, t \mid -N + 2 \le t \le -N + D\} \quad .$$

In practice, we constraint phases to meet an additional requirement: the number of bits that are sent during a phase must be a multiple of the bus width. Indeed, during a phase, data are sent to the Fifo word by word, each word having the size of the Fifo width. For instance the `x_mirr2` variable being 16 bit wide, and assuming the Fifo to be 32 bit wide, two `x_mirr2` data can be placed in one Fifo word (remember that we assume the bit width of all streams to be a divisor of the Fifo width.) Hence, we require the length of a phase to be a multiple of the $\frac{\text{Fifo width}}{\text{data width}}$ ratio. If this condition is not fulfilled, one can always split the phase in two subphases, one of them meeting this condition, and the other one having less bits than one Fifo word.

This requirement forces us to fix the value of the parameters at this stage. In our example (32 bit wide Fifo and 16 bit wide variable), the ratio $\frac{\text{Fifo width}}{\text{data width}}$ is 2, and phase ϕ_0 in (4) contains $D - 1$ virtual clock cycles. If D is even, this interval has to be divided into two phases: phase $\phi_1 = \{t \mid -N + 2 \le t \le -N + D - 1\}$ and phase $\phi_2 = \{t \mid t = -N + D\}$. This situation is illustrated in Fig. 10.6 which shows the phases of the program of Fig. 10.4, for $N = 10$, $D = 12$ and $M = 100$. We can see that $\phi_1 = \{t \mid -8 \le t \le 1\}$ and $\phi_2 = \{t \mid t = 2\}$.

10.6.2 Patterns

Inside each phase, a *pattern* describes the order in which data are sent to the Fifo. A pattern is repeated cyclically during one phase. Recall that each Fifo word contains only values related to one variable. For instance, during phase ϕ_3 of Fig. 10.6, one can choose to fill a Fifo word with two `x_mirr1` data and then the next Fifo word with two `x_mirr2` data and repeat this pattern 3 times so that 6 data of each stream are sent. A pattern is therefore simply an ordered list of variables that will be sent through the bus to the application interface. In the case of phase ϕ_3, the pattern is $(\texttt{x\_mirr1}, \texttt{x\_mirr2})$. In this particular case, the pattern is simple since `x_mirr1` and `x_mirr2` have the same bit width.

In general, the choice of a pattern must be done in such a way that deadlock situations may not occur due to a lack of memorization resources in the interface.

Say for instance that the IP to be interfaced has two input signals `A` and `B` during a particular phase. Denote respectively $\mathrm{BW}_{\mathtt{A}}$ and $\mathrm{BW}_{\mathtt{B}}$ the bit width of `A` and `B`, and assume that $\mathrm{BW}_{\mathtt{A}} = 16$ and $\mathrm{BW}_{\mathtt{B}} = 2$. Let $\mathrm{BW} = 32$ be the bus width. Denote respectively $\mathrm{ND}_{\mathtt{A}}$ and $\mathrm{ND}_{\mathtt{B}}$ the *number of data* contained in one bus word for each variable. Then $\mathrm{ND}_{\mathtt{A}} = \frac{\mathrm{BW}}{\mathrm{BW}_{\mathtt{A}}} = 2$ and $\mathrm{ND}_{\mathtt{B}} = \frac{\mathrm{BW}}{\mathrm{BW}_{\mathtt{B}}} = 16$. Assume that the interface has only one BW bits word of memory available to store each variable (this assumption is reasonable in order to keep the interface simple.)

To start the computations of the phase, the bus must send to the Fifo one bus word containing `A` values (i.e. 2 values) followed by one bus word containing `B` values (i.e. 16 values). These values will feed the IP architecture during 2 virtual clock cycles. If the next word sent by the bus was composed of `B` values, then the IP would be blocked, as the interface would not be able to store this second word before getting the next `A` value. Hence the next data on the bus *must* be an `A` as well as the next six following bus words. In the end, we have the *pattern* `BAAAAAAAA` which has to be cyclically repeated during the whole phase. Note that the pattern `ABAAAAAAA` is also valid, and the solution is therefore not unique.

Two methods can be used to compute a deadlock-free pattern. First, one can *enumerate* the patterns, which amounts to simulating the process described in the previous paragraph. This method may be effective, as the pattern length is less than 64, and as we assumed the bit width of variables to be less than 32. Second, a formulae describing valid patterns can be obtained by the following algorithm, that we call the *parallel pattern* method.

Assume that each variable has its own bus, instead of sharing a common bus between the interface and the IP. Then, a *parallel pattern* is a list of dates when each variable can be sent to its bus. (In such a model, in the previous example, a `A` word and a `B` word could be sent in parallel.) A method to generate deadlock-free patterns consists in finding out a parallel pattern, and then in serializing it without introducing deadlock.

To find parallel patterns, the method is as follows:

1. Compute the *parallel pattern length* PPL. This length is the smallest number of data that can be sent cyclically to the bus without deadlock. It can be proved that this value is

$$\mathrm{PPL} = \frac{\max_{\mathtt{V}} \mathrm{ND}_{\mathtt{V}}}{\min_{\mathtt{V}} \mathrm{ND}_{\mathtt{V}}}$$

where $\mathrm{ND}_{\mathtt{V}}$ is the number of data contained in one bus word for variable V. For instance in the above example, $\mathrm{PPL} = 16/2 = 8$. Indeed, A is repeated 8 times in the valid patterns mentioned for the example, and this value is always an integer because we assume that the bit width are powers of 2. The *pattern cycle length* PCL, that is to say, the number of virtual clock cycles executed by the IP during one pattern, is therefore

$$\mathrm{PCL} = \mathrm{PPL} \times \min_{\mathtt{V}} \mathrm{ND}_{\mathtt{V}} = \max_{\mathtt{V}} \mathrm{ND}_{\mathtt{V}} \quad .$$

2. For each variable V, $\frac{\mathrm{PCL}}{\mathrm{ND}_{\mathtt{V}}}$ bus words must be sent during the pattern of length PPL. It is easy to see that these words must be regularly spaced. The time delay between two consecutive bus word of variable U is thus:

$$\frac{\mathrm{PPL}}{\mathrm{PCL}/\mathrm{ND}_{\mathtt{U}}} = \frac{\mathrm{ND}_{\mathtt{U}}}{\min_{\mathtt{V}} \mathrm{ND}_{\mathtt{V}}} \quad ,$$

and these data must be sent at time steps:

$$0,\ \frac{\mathrm{ND}_{\mathtt{U}}}{\min_{\mathtt{V}}(\mathrm{ND}_{\mathtt{V}})},\ 2 \times \frac{\mathrm{ND}_{\mathtt{U}}}{\min_{\mathtt{V}}(\mathrm{ND}_{\mathtt{V}})},\ \ldots,\ (\frac{\mathrm{PCL}}{\mathrm{ND}_{\mathtt{U}}} - 1) \times \frac{\mathrm{ND}_{\mathtt{U}}}{\min_{\mathtt{V}}(\mathrm{ND}_{\mathtt{V}})} \quad .$$

Using this method, we obtain a parallel order for sending the data to the bus. In our example, we have $\mathrm{PPL} = 8$, $\frac{\mathrm{ND}_{\mathtt{A}}}{\min(\mathrm{ND}_{\mathtt{A}},\mathrm{ND}_{\mathtt{B}})} = 1$ and $\frac{\mathrm{ND}_{\mathtt{B}}}{\min(\mathrm{ND}_{\mathtt{A}},\mathrm{ND}_{\mathtt{B}})} = 8$, so that the parallel order is:

Time steps:	0	1	2	3	4	5	6	7
Dates for A:	A	A	A	A	A	A	A	A
Dates for B:	B	|	|	|	|	|	|	|

where | means no data is sent.

From a parallel pattern, one obtain easily a sequential pattern by interleaving the branches of the parallel pattern in such a way that the partial order imposed by the parallel pattern is satisfied. In our example, the initial A and B can be sent in any order, which leads to one of the two solutions `ABAAAAAAA` or `BAAAAAAAA` proposed previously. This also shows that these are the only valid solutions.

10.7 Generating the interface

As seen in the previous section, the *interface model* is abstracted by the description of its phases and, for each phase, of its patterns. We now explain how the software and the hardware part of the interface are generated from these informations.

10.7.1 Generating the software part

The software part of the application interface is a C program that the host runs in order to send data to the input Fifo or read data from the output Fifo through the PCI bus. The exact behavior of the C code is completely specified by the phase and pattern information. However, during the generation of the C code, many details must be handled carefully, like the conversion between the Alpha variable indexing and the implementation in memory (array starting a index 0) for instance. All these technical difficulties have been solved in the Alpha to C compiler described elsewhere [24]. The software part is therefore produced by a C program which was obtained by retargeting the Alpha to C compiler.

```
 /* phase 6 : from 12 to 89 with variables:
       {d_mirr1, y_mirr1, Y1, x_mirr1, x_mirr2}*/
 for (t = 12; t <= 89; t = t + 2) {
    WriteFifo( (int *)(_d_mirr1 + (t -12)));
    WriteFifo( (int *)(_y_mirr1 + (t -12)));
    ReadFifo( (int *)(_Y1 + (t -10)));
    WriteFifo( (int *)(_x_mirr1 + (t -3)));
    WriteFifo( (int *)(_x_mirr2 + (t+8)));
 }
 /* phase 7 : from  90 to 90 with variables:
      {d_mirr1, y_mirr1, Y1, x_mirr1, x_mirr2} (1 data sent)*/
 t = 90; {
    WriteFifo( (int *)(_d_mirr1 + (t -12)));
    WriteFifo( (int *)(_y_mirr1 + (t -12)));
    ReadFifo( (int *)(_Y1 + (t -10)));
    WriteFifo( (int *)(_x_mirr1 + (t -3)));
    WriteFifo( (int *)(_x_mirr2 + (t+8)));
 }
/* phase 8 : from  91 to 91 with variables:
     {Y1, x_mirr2} (1 data sent)*/
 t = 91; {
    ReadFifo( (int *)(_Y1 + (t -10)));
    WriteFifo( (int *)(_x_mirr2 + (t+8)));
 }
```

Figure 10.7 C code generated for phases ϕ_6, ϕ_7, ϕ_8 of Fig. 10.6 (for a 32 bit wide Fifo). Array `_d_mirr1` stores the values of the `d_mirr1` variable of the Alpha program.

Fig. 10.7 shows a part of the C program which was generated to performs inputs and outputs of phases ϕ_6, ϕ_7 and ϕ_8 of Fig. 10.6. Each call to `WriteFifo` or `ReadFifo` activates a function of a low-level communication library. Note that the `t` index corresponds precisely to the `t` index of the Alpha program of Fig. 10.4 as well as to the `t` coordinate of Fig. 10.6, hence the designer can easily understand what is going on during the execution of the software interface.

Notice that, in this C program, inputs and outputs are interleaved, hence the Fifo is not really needed (a Fifo of size one would be sufficient). Inserting a Fifo between the architecture and the host is however useful in order to interface the IP to a more general system. Instead of a single C program, another possibility would be to have inputs and outputs handled by two different processes running in parallel on the host.

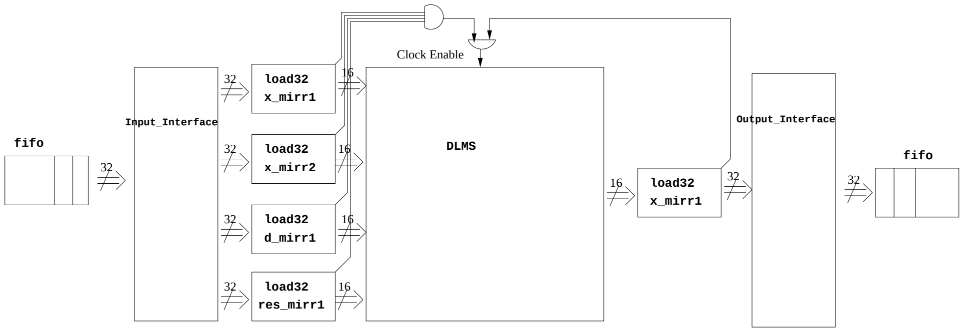

Figure 10.8 Hardware part of the application interface for the DLMS

10.7.2 Generating the hardware part: principles

The hardware part is generated in VHDL and is synthesized for the FPGA chip. The architecture of the hardware part is illustrated in Fig. 10.8.

The hardware part is divided into the input interface and the output interface which are almost symmetrical. The main difference is that the interface is started with a `start` signal (set up by the user, here indicating the virtual date $t = -8$), while the output interface is started by a `start_out` signal set up automatically (here at virtual date $t = 10$).

We detail the organization of the input interface represented on Fig. 10.8. Consider an input, say input `I1`, ot the architecture, and let $W_{\mathtt{I1}}$ be the bit width of `I1`. This input is connected to a `load32` component which is parameterized by the bit width $W_{\mathtt{I1}}$ of `I1`. The `load32` component contains a shift register that allows 32 bits to be read in parallel from the Fifo and $W_{\mathtt{I1}}$ bits to be output during $32/W_{\mathtt{I1}}$ clock cycles.

Each `load32` component is connected to the `Input_Interface` component. The `Input_Interface` component receives data from the Fifo and stores them in the appropriate `load32` shift register.

The control of this architecture is provided by a hierarchical two-level finite state machine. The states of first level are the phases of the interface, and the states of the second level are the variable names which define the patterns inside a phase. Switching from one phase to the other is done by counting the number of elapsed virtual clock cycles (for instance, we see on Fig. 10.6 that phase ϕ_1 must last 10 virtual clock cycles). An efficient control of the `load32` shift register allows the loading of a new Fifo word to be overlapped with the output of the last data word to the application architecture. Hence, provided that the FPGA clock frequency is high enough, the array is fed at the throughput allowed by the bus. Usually, the FPGA clock is fast enough to cope with the bus bandwidth, since the application design is highly pipelined. However, one can imagine extreme situations where this is not the case. For example, if the input data was only 2 bit wide, the FPGA clock frequency would have to be 16 times the bus clock frequency which is not very realistic.

Note that the operation of the interface and of the application architecture is frozen when the Fifo is empty.

10.7.3 Efficient synthesis of the hardware part

The originality of our approach lies in the fact that we designed a *family* of interfaces that can be parameterized for different IPs provided they respect the model of section 10.4. Most of the produced VHDL code is generic, i.e. identical for all IP, and, for each IP, a *configuration file* which drives the control of the IP is generated. In this section, we show how to efficiently implement this genericity in VHDL. Indeed, the performance of the final IP is greatly influenced by the quality of the interface description.

The VHDL program describes a finite state machine which is identical for all application architectures meeting the assumptions of section 10.4 (see appendix 10.11.) The information on one phase (i.e. pattern, duration, active variables, etc) is stored in a VHDL `record`. The finite state machine uses an `array of record` whose size is the number of phases (here 10 phases.) With this implementation, one can gather all the application dependent information in one configuration file. This file is built by extracting information from the interface program of Fig. 10.4 using the *same* function for finding the phases and pattern as the one used during the C code generation for the software part. Hence, and this is a very important feature of our methodology, compatibility between the software part and the hardware part is very easy to ensure as they both derive from the same source code (program of Fig. 10.4 with the same function used for common information.) This way of representing the finite state machine simplifies the automatic generation of the VHDL interface program for all applications. Moreover it guarantees a very efficient synthesis with vendor's tools.

10.8 Experiments

The automatic interface generation was implemented and experimented for a Spyder-X2 PCI board [28] based on a Xilinx Virtex 800 device as shown in Fig. 10.5. In this architecture the host processor communicates with the FPGA through a PCI interface using memory-mapped or DMA transfers. The observed bandwidth between the host CPU and the FPGA is 8 MB/s at most.

The VHDL for the DLMS was synthesized automatically from an Alpha specification down to AlpHard (see [10].) An important issue during the design process was functional simulation. Thanks to the flexibility of the MMAlpha environment, we were able to use the same data from high-level simulation down to the detailed VHDL simulation, hence speeding up the design time and allowing fast back annotation from hardware simulation to the high-level synthesis process.

The application interface was generated automatically from the AlpHard interface for the DLMS algorithm (Fig. 10.4) for the following values of the parameters: $N = 10$, $D = 12$ and $M = 100$. The width of the interface Fifos was 32 bits. The synthesis was realized with the `Synplify` software [27]. Table 10.1 gives the number of look-up tables (Lut) necessary for the synthesis of the DLMS alone, of the DLMS and the interface without the Fifos, and of the total design. The clock cycle, as estimated by the synthesis tool is also given. Finally, the throughput of the interface is evaluated from the cycle time and the number of data that are produced by the architecture during each cycle. More precisely, the DLMS produces one 16 bit `y_mirr` value during each cycle. The table shows also the maximum throughput of the PCI bus of the board, as observed for several designs.

Design	Number of LUT	Clock cycle (ns)	Throughput (MB/s)
DLMS	5938 (31%)	30	66.67 MB/s
DLMS + interface	6501 (34%)	30	66.67 MB/s
DLMS + interface + Fifos	6928 (36%)	31	64.5 MB/s
PCI bus (Max)	–	–	8 MB/s

Table 10.1 Result of the interface generation for the DLMS algorithm. Parameters: $N = 10$, $D = 12$ and $M = 100$. This table gives the number of look-up tables occupied by the design, in the Virtex XCV800 chip (the percentage of total LUTs used in a Virtex XCV800 is given between parentheses), the clock cycle estimated by the synthesis tools, and the (one-way) throughput of the interface. The maximum observed throughput of the PCI bus of the Spyder board is given for comparison.

One can draw some conclusions from this table.

- First, the hardware interface is not a limiting factor of speed for this design. Indeed, the clock cycle is increased only by 1 ns by the Fifos.
- Secondly, the PCI bus is clearly the limiting factor of the interface: there is a factor 8 in the best case between the bandwidth of the bus and the bandwidth that could be achieved by the design. As was expected, the design of such a high performance device is therefore limited by the communication with the host.

Notice that this interface could have been optimized since the `x_mirr1` and `x_mirr2` streams are a time shift of one another, hence only one of the two streams needed to be sent through the bus. Moreover, the `y_mirr1` stream should in practice be taken at the output of the architecture and not sent from the host (the values sent where obtained during simulations.) We kept this non optimized implementation for our experiments in order to validate the interface protocol on a non trivial example.

10.9 Related work and discussion

Most of the research on FPGA design focuses on the efficiency of the design itself rather than the efficiency of the interface of the design. Tests are usually made with data already in the on-board memory or with data arriving directly on board via an Analog/Digital converter. Among others, references [14, 9] describe manually written interfaces.

Approaches dealing with automatic generation of interfaces can be classified in two categories: co-design methods, and high-level synthesis methods.

Co-design methods deal with very general types of applications, and therefore communications are usually implemented using a high-level synchronization mechanism which leads to complex protocols (for instance, remote procedure call in CoWare [6].) Many FPGA compilation projects rely on control dominated models like Petri nets [31, 18], or Communicating Sequential Processes [22]. The advent of real-time operating system (RTOS) can also be a solution but efficiency will probably be degraded.

In high-level synthesis methods, attempts have been made to automate interface generation. These attempts, as in the work presented here, restrict the types of interfaced architectures. In Pico [26], the interface problem is solved at run time by a bus arbiter, but since the target architecture has only a small number of processors, the efficiency of the interface is not the major issue. Artemis [23, 13] relies on the Spade methodology [17] to implement efficiently communicating Kahn process networks on buses.

The methodology that we propose for the interface generation presents several novelties. Firstly, the dynamic control is as low as possible. Indeed, it is restricted to ensuring a correct behavior when an external event such as a bus interruption occurs. Therefore, the efficiency of the interface can be statically predicted. Secondly, the design is safe because the interface is compiled *together with the architecture*, and moreover, the hardware and software parts are derived from *the same* AlpHard program and using *the same* tool. Thirdly, our method provides the user with simulation facilities that are available as part of the MMAlpha tool. Finally, our model is generic, and it is not only an implementation dedicated to the MMAlpha design flow but can be applied to any compiled architecture which meets the characteristics of our model.

10.10 Conclusion

In this chapter, we have presented a tool for synthesizing interfaces for linear regular architectures. Our model of interface is bus based and can cope with interruptions in the flow of data arriving from the host. Our interface is application dependent, and is generated automatically from a high-level description of the application, as obtained using the MMAlpha tool. Both the software part and the hardware part of the interface are generated from the same description using the same set of tools, therefore ensuring these parts to be coherent. The synthesis of the interface structures the data communications into phases and patterns, and generates a C program to be run on the host, and a VHDL program to be synthesized on the hardware platform. We have experimented this interface generator on a DLMS algorithm automatically compiled on a Spyder FPGA board and we have shown that the interface allows high performances to be reached for this application.

10.11 Appendix: VHDL code of the application-dependent interface

This part of the program stores into an array (`Input_PhaseTable`) the information about input phases (i.e. phase where input variables are concerned). `Input_ToPhase` is a function that fills the record with the input values.

```
constant Input_Nb_IO_Port: integer :=4;

- Total number of input phases
constant Input_NbPhase: integer :=9;

- Maximum number of variables in pattern
```

```
constant Input_MaxStep: integer := 18;

- Maximum size of a pattern
constant Input_MaxLength: integer := 5;

- Max Number of virtual clock cycle in a phase
constant Input_MaxCount: integer := 79;

subtype Input_TIO_select is
      std_logic_vector(Input_Nb_IO_Port-1 downto 0);
subtype Input_TStep is integer range 0 to Input_MaxStep;
subtype Input_TLength is integer range 0 to Input_MaxLength;
subtype Input_TCount is integer range 0 to Input_MaxCount;

- Record used to store all Phase Information
Type Input_TPhaseInfo is record
      - offset of the pattern in the pattern table
      Offset          : Input_TStep;
      - Length of the pattern of the phase
      PatternLength   : Input_TLength;
      - Which port are active in the phase
      ActivePorts     : Input_TIO_select;
      -  how many virtual clock cycles in the phase
      NbArrayCycles   : Input_TCount;
end record;

type Input_TPhaseTable is array (0 to Input_NbPhase)
     of Input_TPhaseInfo;

constant Input_PhaseTable : Input_TPhaseTable :=
(
 -        Offset   PatternLength PhaseLength NBCycle   Activity
     0 => Input_ToPhase( 0,  1,  4, (others => '0')),
     1 => Input_ToPhase ( 1,  1,  10,  Input_Portx_mirr2),
     2 => Input_ToPhase( 2,  1,  1,  Input_Portx_mirr2),
     3 => Input_ToPhase ( 3,  2,  6,
                         Input_Portx_mirr1 or Input_Portx_mirr2),
     4 => Input_ToPhase( 5,  2,  1,
                         Input_Portx_mirr1 or Input_Portx_mirr2),
     5 => Input_ToPhase ( 7,  2,  2,
                         Input_Portx_mirr1 or Input_Portx_mirr2),
     6 => Input_ToPhase ( 9,  4,  78,
                         Input_Portd_mirr1 or Input_Porty_mirr1 or
                         Input_Portx_mirr1 or Input_Portx_mirr2),
     7 => Input_ToPhase( 13,  4,  1,
                         Input_Portd_mirr1 or Input_Porty_mirr1 or
                         Input_Portx_mirr1 or Input_Portx_mirr2),
     8 => Input_ToPhase( 17,  1,  1,  Input_Portx_mirr2),
```

```
    9 => Input_ToPhase( 0,  1,  4, (others => '0'))
);
```

REFERENCES

[1] Annapolis. WildStar Datasheet, 2001.

[2] F. Catthoor, K. Danckaert, C. Kulkarni, and T. Omnes. Data Transfer and Storage Architecture Issues and Exploration in Multimedia Processors. In *Programmable Digital Signal Processors: Architecture, Programming, and Applications*. Marcel Dekker, Inc, New York, 2000.

[3] Celoxica. RC1000 Datasheet, 2001.

[4] A. Darte. Regular Partitioning for Synthesizing Fixed-Size Systolic Arrays. *Integration, The VLSI Jounal*, 12:293–304, December 1991.

[5] A. Darte and B. Rau et F. Vivien R. Schreiber. A Constructive Solution to Juggling Problem in Systolic Array Synthesis. Technical Report 1999-15, Laboratoire de l'informatique du parallélisme, 1999.

[6] H. De Man, I. Bolsens, B. Lin, K. Van Rompaey, S. Vercauteren, and D. Verkest. Hardware-Software Codesign of Digital Telecommunication Systems. *Proceedings of the IEEE*, 85(3):391–418, 1997.

[7] G. De Micheli. Network on Chip: a new Paradigm for System on Chip Design. In *Design Automation and Test in Europe (DATE) 2002*, pages 418–420, Paris, 2002. IEEE Computer Society Press.

[8] S. Derrien and T. Risset. Interfacing Compiled FPGA Programs: the MMAlpha Approach. In A. Arabnia, editor, *PDPTA2000: Second International Workshop on Engineering of Reconfigurable Hardware/Software Objects*. CSREA Press, June 2000.

[9] Jan Frigo, Maya Gokhale, and Dominique Lavenier. Evaluation of the Streams-C C-to-FPGA Compiler: an Applications Perspective. In *Ninth international symposium on Field programmable gate arrays*, pages 134–140. ACM Press, 2001.

[10] A.C. Guillou, P. Quinton, T. Risset, and D. Massicotte. High-Level Design of Digital Filters in Mobile Communications. DATE Design Contest 2001, March 2001. Second place, available at http://www.irisa.fr/bibli/publi/pi/2001/1405/1405.html.

[11] Simon S. Haykin. *Adaptive Filter Theory*. Prentice-Hall information and system sciences series. Prentice-Hall, Upper Saddle River, NJ 07458, USA, third edition, 1996.

[12] M. Katsushige, N. Kiyoshi, and K. Hitoshi. Pipelined LMS Adaptative Filter Using a New Look-Ahead Transformation. *IEEE Transactions on Circuits and Systems*, 46:51–55, January 1999.

[13] B. Kienhuis, E. Rijpkema, and E.F. Deprettere. Compaan: Deriving Process Networks from Matlab for Embedded Signal Processing Architectures. In *8th International Workshop on Hardware/Software Codesign (CODES'2000)*, 2000.

[14] D. Lavenier. SAMBA: Systolic Accelerator for Molecular Biological Application. Technical Report 988, Irisa, March 1996.

[15] P. Le Moenner, L. Perraudeau, S. Rajopadhye, T. Risset, and P. Quinton. Generating Regular Arithmetic Circuits with AlpHard. In *Massively Parallel Computing Systems (MPCS'96)*, May 1996.

[16] P.H.W. Leong, M.P. Leong, O.Y.H. Cheung, T. Tung, C.M. Kwok, M.Y. Wong, and K.H. Le. Pilchard - A Reconfigurable Computing Platform with Memory Slot Interface. In *Symposium on Field-Programmable Custom Computing Machines (FCCM)*, California, 2001. IEEE Computer Society Press.

[17] P. Lieverse, P. Van der Wolf, E. F. Deprettere, and K. Vissers. A Methodology for Architecture Exploration of Heterogeneous Signal Processing Systems. *Journal of VLSI Signal Processing for Signal, Image and Video Technology*, 29(3):197–207, November 2001. Special issue on SiPS'99.

[18] B. Lin and S. Vercauteren. Synthesis of Concurrent System Interface Modules with Automatic Protocol Conversion Generation. In *international Conference on Computer-Aided Design (ICCAD)*, pages 101–109, 1994.

[19] C. Mauras. *Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones*. Thèse de doctorat, Ifsic, Université de Rennes 1, December 1989.

[20] E. Mémin and T. Risset. Full Alternate Jacobi Minimization and VLSI Derivation of Hardware for Motion Estimation. In *Int. Workshop on Parallel Image Processing and Analysis, IWPIPA'99*, Madras, India, Jan. 1999.

[21] A. Mozipo, D. Massicotte, P. Quinton, and T. Risset. A Parallel Architecture for Adaptative Channel Equalization Based On Kalman Filter Using MMAlpha. *1999 IEEE Canadian Conference on Electrical & Computer Engineering*, pages 554–559, May 1999.

[22] I. Page. Constructing Hardware-Software Systems from a Single Description. *Journal of VLSI Signal Processing*, 12:87–107, 1996.

[23] A. D. Pimentel, L. O. Hertzberger, P. Lieverse, P. Van Der Wolf, and E. F. Deprettere. Exploring Embedded-Systems Architectures with Artemis. *IEEE Computer*, 34(11):57–63, 2001.

[24] Fabien Quilleré and Sanjay Rajopadhye. Optimizing Memory Usage in the Polyhedral Model. *ACM Transactions on Programming Languages and Systems*, 22(5):773–815, 2000.

[25] P. Quinton and Y. Robert. *Systolic Algorithms and Architectures*. Prentice Hall and Masson, 1989.

[26] R. Schreiber, S. G. Aditya, B.R. Rau, S. Mahlke, V. Kathail, D. Cronquist, and M. Sivaraman. PICO-NPA: High-Level Synthesis of Nonprogrammable Hardware Accelerators. *Journal of VLSI Signal Processing*, 2001.

[27] Synplify Pro 7.0 Reference Manual, October 2001.

[28] K. Weiss, C. Oetker, I. Katchan, T. Steckstor, and W. Rosenstiel. Power Estimation Approach for SRAM-based FPGAs. In *IEEE Symposium of Field Programmable Gate Array*, 2000.

[29] D. Wilde. A Library for doing Polyhedral Operations. Technical Report 785, Irisa, Rennes, France, 1993.

[30] W. Wolf and N. Martinez. Session: Platform Based Design and Virtual Component Reuse. In *Design Automation and Test in Europe (DATE) 2002*, pages 296–316, Paris, 2002. IEEE Computer Society Press.

[31] X. Zhu and B. Lin. Hardware Compilation for FPGA-Based Configurable Computing Machines. In *Design Automation Conference(DAC)*, pages 697–703, New Orleans, 1999. ACM.