Hardware Synthesis for Multi-Dimensional Time

Anne-Claire Guillou
Irisa
Campus de Beaulieu
35042 Rennes cedex
France
{aguillou}@irisa.fr

Patrice Quinton
Irisa
Campus de Beaulieu
35042 Rennes cedex
France
{Patrice.Quinton}@irisa.fr

Tanguy Risset
Inria, Lip, ENS-Lyon
46 Allée d'Italie
69363 Lyon cedex 07
France
{Tanguy.Risset}@ens-lyon.fr

Abstract

This paper introduces basic principles for extending the classical systolic synthesis methodology to multi-dimensional time. Multi-dimensional scheduling enables complex algorithms that do not admit linear schedules to be parallelized, but it also implies the use of memories in the architecture. The paper explains how to obtain compatible allocation and memory functions for VLSI *(or* SIMD*-like code) generation. It also presents an original mechanism for controlling a* VLSI *architecture which has a multi-dimensional schedule. A structural* VHDL *code has been derived and synthesized (for implementation on* FPGA *platform) using these systematic design principles. These results are preliminary steps to the possibility of a systematic hardware synthesis for multi-dimensional time.*

Keywords: High level synthesis, systolic architecture, multi-dimensional scheduling, FPGA compilation

1: Introduction

In the wide context of System on Chip (SoC) design, the different components that are assembled use many different design technologies. All these design technologies (or at least the most time consuming ones) must be accelerated in order to reduce the *time to market* which is essential for commercial purpose. Among these different tasks, designing special-purpose chips is a key issue because the design of a new custom hardware is still very long. High-level design automation tools are still in the research domain because the so called *behavioural synthesizers* tools have not proved to be very efficient.

Our research is oriented toward the automatic design of regular applications (signal processing mainly) from recurrence equation specifications [11, 8]. The Alpha language is used in the MmAlpha environnement [13] to provide high-level designs of systolic-like architectures. A typical design flow includes uniformization, linear scheduling, mapping and hardware generation. This design methodology has been prototyped for simple algorithms but still needs to be extended to more complex applications in order to be accepted by designer as a useful tool for the design of special-purpose blocks. Among the interesting extensions, partitioning has been targeted (as in [6]) but multi-dimensional scheduling [9] has not been successfully applied to hardware design. The automatic design of hardware for non linear time complexity algorithms will be very useful because, unlike the simple applications mentioned above, the resulting architectures are very difficult to see intuitively.

The main novelty of this paper is to extend the classical systolic design flow to multi-dimensional scheduling, in particular, to provide a clearly defined methodology for memory synthesis in Alpha. In addition, the results presented here can be useful for promoting ideas of high-level design of complex applications, involving memory use and not only register files; it also can be seen as a way to perform a special type of partitioning within the polyhedral model (while the classical partitioning scheme impose non linear transformations). We start the paper by presenting a motivating example in section 2, then we review in section 3 the existing tools that we will gather to provide our methodology. In section 4, we present the main results of the paper: the combination of memory function and allocation and a mechanism for controlling multi-dimensional timed architectures. Implementation results are presented in section 5 and provide an idea of the additional complexity resulting from the use of a multi-dimensional schedule.

2: Motivating example

The matrix-multiplication algorithm is often used to prototype the synthesis of systolic arrays because it combines a three-dimensional index space with a simple, easy to understand dependence structure. The initial specification we start from is shown in Fig. 1(a). It is a set of *uniform recurrence equations* written using the Alpha language: each equation is a single-assignment which defines one variable.

Most often, matrix-multiplication is mapped onto a 2-dimensional systolic array [18] using a linear scheduling. The schedule function could be for example:

$$T_A[i,j,k] = T_B[i,j,k] = i+j+k, \quad T_C[i,j,k] = i+j+k+1, \tag{1}$$

and the algorithm would then be executed in $N+M+P$ steps on a 2-dimensional architecture.

In practice [13], this architecture is controlled by a *clock enable* signal that allows the execution of all processors to be frozen if, for instance, input data are not ready. This signal acts as a *virtual clock* and establishes a correspondence between the virtual time $i+j+k$ given by the schedule and the actual clock of the circuit. The use of virtual clock ensures that the behavior of the architecture is really the one that is expected after the scheduling, and in addition, it allows the architecture to be easily integrated as an IP [8] in a complex design.

For many different reasons (area, power consumption or throughput of the resulting architecture), a designer might prefer a multi-dimensional schedule for this program, for instance:

$$T_A[i,j,k] = T_B[i,j,k] = \begin{pmatrix} i+j \\ k \end{pmatrix}, \quad T_C[i,j,k] = \begin{pmatrix} i+j \\ k+1 \end{pmatrix} \quad . \tag{2}$$

```
system MatMat :{M,N,P | 3<=M; 3<=N; 3<=P}
   (a : {i,k | 1<=i<=M; 1<=k<=N} of real;
    b : {k,j | 1<=k<=N; 1<=j<=P} of real)
returns
   (c : {i,j | 1<=i<=M; 1<=j<=P} of real);
var
  B : {i,j,k | 1<=i<=M; 1<=j<=P; 2<=k<=N}
            of real;
  A : {i,j,k | 1<=i<=M; 1<=j<=P; 2<=k<=N}
            of real;
  C : {i,j,k | 1<=i<=M; 1<=j<=P; 1<=k<=N}
            of real;
let
  B[i,j,k] =
      case
        { | i=1} : b[i+k-1,j];
        { | 2<=i} : B[i-1,j,k];
      esac;
  A[i,j,k] =
      case
        { | j=1} : a[i,j+k-1];
        { | 2<=j} : A[i,j-1,k];
      esac;
  C[i,j,k] =
      case
        { | k=1} : 0[];
        { | 2<=k} : A * B + C[i,j,k-1];
      esac;
  c[i,j] = C[i,j,N];
tel;
```

(a) Original Alpha system

```
system MatMat :{M,N,P | 3<=M; 3<=N; 3<=P}
  (a : {i,k | 1<=i<=M; 1<=k<=N} of real;
   b : {k,j | 1<=k<=N; 1<=j<=P} of real)
returns
  (c : {i,j | 1<=i<=M; 1<=j<=P} of real);
var
  Acom : {t1,t2,p | p+2<=t1<=p+M+1;
             2<=t2<=N; 1<=p<=P-1} of real;
  B : {t1,t2,p | p+1<=t1<=p+M;
               2<=t2<=N; 1<=p<=P} of real;
  A : {t1,t2,p | p+1<=t1<=p+M;
               2<=t2<=N; 1<=p<=P} of real;
  C : {t1,t2,p | p+1<=t1<=p+M;
               2<=t2<=N+1; 1<=p<=P} of real;
let
  Acom[t1,t2,p] = A[t1-1,t2,p];
  B[t1,t2,p] =
      case
        { | t1=p+1} : b[t1+t2-p-1,p];
        { | p+2<=t1} : B[t1-1,t2,p];
      esac;
  A[t1,t2,p] =
      case
        { | p=1} : a[t1-1,t2];
        { | 2<=p} : Acom[t1,t2,p-1];
      esac;
  C[t1,t2,p] =
      case
        { | t2=2} : 0[];
        { | 3<=t2} : A[t1,t2-1,p] *
               B[t1,t2-1,p] + C[t1,t2-1,p];
      esac;
  c[i,j] = C[i+j,N+1,j];
tel;
```

(b) After scheduling of (2)

Figure 1. Initial specification of the matrix-multiplication algorithm using the Alpha (a) and the same specification after applying the schedule of (2) (b).

The lexicographic order imposed by this schedule is denoted as $\preceq$ in what follows. It guarantees that the dependencies between computations are satisfied. Here for example, $C[i,j,k]$ is computed strictly after $A[i-1,j-1,k]$ because $\begin{pmatrix} i+j-2 \\ k \end{pmatrix} \prec \begin{pmatrix} i+j \\ k+1 \end{pmatrix}$.

Fig. 1(b) shows the matrix-multiplication program of Fig. 1(a) once all variables have been reindexed by a space-time transformation, whose first components are the schedule.

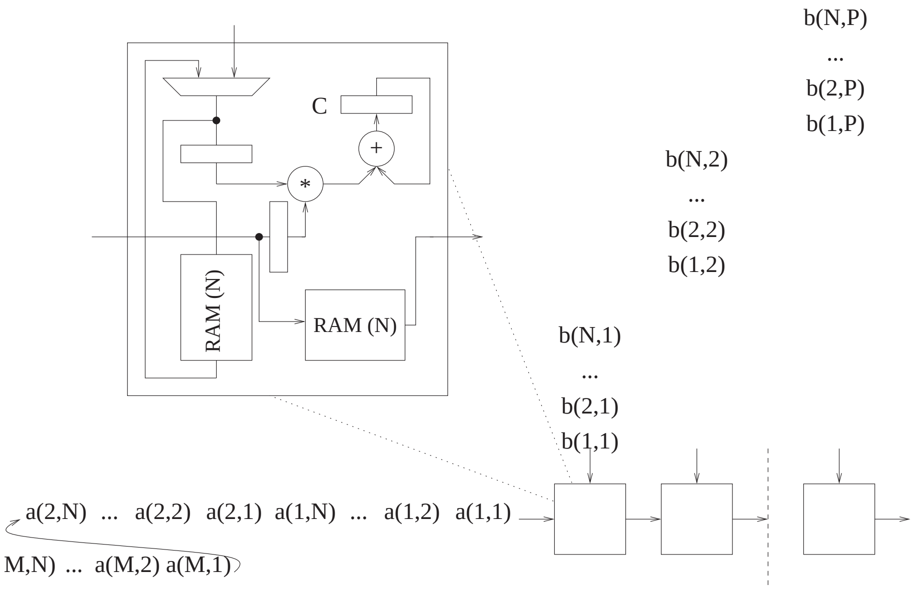

Figure 2. Architecture for the matrix-multiplication with the multi-dimensional schedule of equation (2).

In spite of its power, multi-dimensional scheduling is seldom used in practice, since translating such a schedule into a real architecture is difficult. Indeed, establishing a correspondence between the *logical time* given by the scheduling and the *physical time* in a chip is not simple: a *virtual clock* is not as easy to identify as for a linear schedule.

The multi-dimensional scheduling of equation (2) could lead to the architecture sketched in Fig. 2 where all computations done at logical time (t_1, t_2) take place at virtual clock cycle number $(N+1)t_1 + t_2$. One notice on this figure that memories are needed to store the data between some virtual time instants. Note also that the control is not shown on this figure: the signals controlling the loading of the registers and memories are quite tricky to set up. This is explained in the remaining of the paper.

3: Existing design tools

In this section, we briefly review the different tools that will be assembled to enable automatic design of hardware for algorithms with multi-dimensional schedule. For a more complete description of these tools, please refer to [12, 14, 9, 17].

Systems of uniform recurrence equations Our initial specification is a *system of uniform recurrence equations*, that is to say a set of equations of the form:

$$z \in D_U : U[z] = \mathcal{F}(\ldots, V[d(z)], \ldots) \tag{3}$$

where D_U, the *iteration space*, is the set of integral points of a *domain* (usually a convex polyhedron or a union of convex polyhedra) of $\mathbb{Z}^n$, and n is the dimension of U. Note that as the system is uniform, all variables have the same dimension. Function d is an affine function of z which happens to be, as the system is uniform, a translation: $d(z) = z + z_1$ where z_1 is a constant vector. These equations recursively define the values of the variables (U, V, etc.). We represent systems of recurrences by means of the Alpha programming language (e.g. the Alpha programs of Fig. 1(a) and 1(b) are uniform systems of recurrence equations). Alpha and its associated synthesis environment MmAlpha provide a framework for the synthesis of regular architectures [13].

Multi-dimensional schedules The *multi-dimensional schedule* [14, 9]. of a system of recurrence equations is a family of functions (one function for each variable U) $T_U(z) = \left[t_U^1(z), .., t_U^k(z)\right]^t$ such that $t_U^i(z)$ is an affine function for all i. Here, k is the *dimension* of the schedule. A schedule specifies an execution order for the operations of the system: $U[z]$ is computed after $V[z']$ if and only if $T_V(z') \prec T_U(z)$.

In this paper we will assume that the schedule have k dimension and have a common linear part for all variable (as we did in the classical systolic design method). We also assume that this linear has rank k and can be completed in a unimodular matrix. We can see that the schedule given by equation (2) meets these conditions. The common linear part of all functions is $i+j$ at level one and k at level two. A possible right inverse is $T^{-1}(t_1, t_2) = [t_1, 0, t_2]^t$. Here, the schedule can be completed into the function $M = (i + j, k, j)$ whose matrix is unimodular.

Allocation function and memory function Given an n-dimensional system of uniform recurrence equations with k-dimensional schedule, an *allocation* is a $n - k$ dimensional function $A(z) = [a_1(z), \ldots, a_{n-k}(z)]^t$ which specifies the coordinates of a processor where each computation is to be executed. In the example of Fig. 1(b), the allocation function $A(i, j, k) = j$ was choosen.

Given a system of recurrence equations with schedule T, the *memory function* of a variable V is a m_V-dimensional linear function M_V which specifies for each value of V, an address in a memory attached to V where the value can be stored for its whole life time. Memory functions can be built in a systematic way as shown in [17, 16] for example.

In the example of Fig. 1(a) and for the schedule of equation (2), the following memory functions are valid:

$$M_{\mathtt{A}}(t_1, t_2, p) = (p, t_2 - 1) \quad M_{\mathtt{B}}(t_1, t_2, p) = (p, t_2 - 1) \quad M_{\mathtt{C}}(t_1, t_2, p) = (p) \quad . \tag{4}$$

In other words, $\mathtt{A}[t_1, t_2, p]$ is stored in a common memory at address $(p, t_2 - 1)$, or equivalently, on a memory located on processor p at address $t_2 - 1$. The same memory function is user for variable `B`. As far as `C` is concerned, all its instances for a given processor are stored on a single element memory, which could be implemented using a single register.

4: Methodology improvements for multi-dimensional scheduling

This section presents the original contributions of the paper: how to combine allocation and memory function and how to control a multi-dimensional scheduled architecture.

4.1: Merging allocation and memory functions

The method roposed by Quillere and Rajopadhye (QR method in what follows)[17] assumes an underlying shared memory architecture model. In the context of VLSI, we want each processor of our architecture to store the values that it computes. It is not exactly the *owner computes rule* because variables are moving in the architecture it is rather a *computer owns rule* (each data is stored on the processor that have computed it). This adds to the constraints identified by the QR method a new constraint on the memory function: its *first* $n-k$ *rows should be the equal to the allocation function.* Note that, as a consequence, these first $n-k$ rows are common to all variables of the program. In this section, we show how allocation functions and memory functions can be combined in order to meet the computer owns rule.

In the QR method, the memory function for V (denoted by M_V) is a *projection* which can be specified by a projection direction, that is to say by the m_v vectors of its kernel ($m_v \leq k$). We now show that it is possible to find $(n-m_v) \times n$ memory functions matrices whose first $n-k$ rows have the same linear part.

Given an Alpha program and a schedule T for this program, let V_0 be the variable of the program whose lifetime d_{V_0} is lexicographically minimum[1]. First, let us prove lemma 4.1 which ensures that we can choose kernel vectors of M_V in such a way that they are also kernel vectors of M_{V_0}. Intuitively, the memory of a variable with a short lifetime will always fit into the memory of a variable with a longer lifetime.

Lemma 4.1 *Given a system of recurrence equations with a multi-dimensional schedule, for all variables V of the system, the algorithm proposed by QR [17] chooses kernel vectors of the memory function M_V in such a way that they are also kernel vectors of M_{V_0} (where V_0 is the variable with the shortest lifetime).*

Proof: see [12]

Hence we can safely assume that $Ker(M_V) \subset Ker(M_{V_0})$. On the other hand, as $Ker(T) \cap Ker(M_{V_0}) = 0$ (see [12]), the function M_{V_0} is a $(n - m_{V0}) \times n$ matrix (with $m_{V0} \leq k$) such that the matrix $\begin{pmatrix} T \\ M_{V0} \end{pmatrix}$ is full-column rank. Therefore, it is possible to reorganize the rows of M_{V_0} into a new matrix M'_{V_0} in such a way that the upper $n \times n$ square sub-matrix of $\begin{pmatrix} T \\ M'_{V_0} \end{pmatrix}$ is non-singular. Note that M'_{V_0} is still a valid memory function for V_0 as $Ker(M_{V_0}) = Ker(M'_{V_0})$. Let A be the sub-matrix of M'_{V_0} composed of its first $n-k$ rows. Then the matrix $\begin{pmatrix} T \\ A \end{pmatrix}$ is non singular and A is a valid allocation ($Ker(T) \cap Ker(A) = 0$).

[1]Given a variable V, the *lifetime vector* d_V of V is the lexicographical maximum, upon all the operations $V(z)$, of the difference between the time at which $V(z)$ is produced and the time of its last consumption.

Our proposition is the following one: for all other variables V of the program, A can be completed into a valid memory function. Indeed, the m_V vectors $(\rho_1, \ldots, \rho_{m_V})$ which generate the kernel of the memory function of V are already in $Ker(A)$ (because of Lemma 4.1). Hence the problem amounts to find a $(k - m_V) \times n$ matrix N_V such that $M_V = \begin{pmatrix} A \\ N_V \end{pmatrix}$ is a memory function for V, or in other words, $Ker(M_V) = Vect(\rho_1, \ldots, \rho_{m_V})$ and M_V is full-row rank. The following theorem solves this problem.

Theorem 4.2 *Given m_V linearly independent integral vectors $(\rho_1, \ldots, \rho_{m_V})$ and a $n-k \times n$ full row rank integral matrix A such that $Vect(\rho_1, \ldots, \rho_{m_V}) \subset Ker(A)$ ($k \geq m_V$), it is possible to find a $(n - m_V) \times n$ full-row rank matrix M_V which is built by completion of A and which satisfies $Ker(M_V) = Vect(\rho_1, \ldots, \rho_{m_V})$.*

Proof: see [12], the proof is constructive, it uses the Hermite decomposition of the matrix whose columns are the ρ_i to build the matrix M_V. The construction is illustrated on the example below.

Hence, we have the following situation: A is a valid allocation function for the computations of the program and moreover, for all variables V, the memory function of V has the form $M_V = \begin{pmatrix} A \\ N_V \end{pmatrix}$. As a consequence, in the resulting architecture operation $V[i_1, \ldots, i_n]$ is computed on processor $A(i_1, \ldots, i_n)^t$ and is stored in memory location $M(i_1, \ldots, i_n)^t = \begin{pmatrix} A(i_1, \ldots, i_n)^t \\ N_V(i_1, \ldots, i_n)^t \end{pmatrix}$. One can interpret this location as being memory $N_V(i_1, \ldots, i_n)^t$ of processor $A(i_1, \ldots, i_n)^t$, which meets the computer owns rule.

We illustrate this method on our program of Fig. 1(a) with schedule of equation (2). The dependencies are $d_{\texttt{C}} = (0, 1)^t$, $d_{\texttt{A}} = d_{\texttt{B}} = (1, 0)^t$. QR method gives the following kernel basis vectors: $Ker(M_{\texttt{C}}) = \{(1, 0, 0)^t, (0, 0, 1)^t\}$, $Ker(M_{\texttt{A}}) = Ker(M_{\texttt{B}}) = \{(1, 0, 0)^t\}$. On the other hand, *texttt*C is the variable with the minimal lifetime, and $m_{\texttt{C}} = k = 2$. A memory function matrix for *texttt*C is $M_{\texttt{C}}(i, j, k) = (j)$.

We can complete the $M_{\texttt{C}}$ matrix with one row, using the technique explained in [12], to obtain $M_{\texttt{A}}$ and $M_{\texttt{B}}$, which gives for instance: $M_{\texttt{A}}(i, j, k) = M_{\texttt{B}}(i, j, k) = (j, k - 1)$. Thus we end up with a change of basis $COB(i, j, k) = (i + j, k, j)$ which associates an execution time and a processor number to all computations resulting in a linear architecture similar to that of Fig. 2. In addition, we also have the information that in each processor p, `C` needs only a register to be stored (its memory function has a 0 local dimension) while `A` and `B` both need a memory of size $N - 1$: $\texttt{A}[i, j, k]$ is stored in processor j at memory location $k - 1$, for example.

4.2: Controllers for multi-dimensional schedules

We now turn to the problem of generating a controller for a multi-dimensional scheduling. An architecture with a linear schedule can be controlled by a single counter enumerating the time steps. To extend this idea to multi-dimensional time, we propose here to control the architecture by means of a *multi-dimensional counter*. A simple example of multi-dimensional counter is a watch enumerating hours, minutes and seconds. Here, however, each hour may have a *different number of minutes* and each minute may have a *different number of seconds*, as our time may span a polyhedron of any dimension.

Let us call *time domain* of a variable the values that its time indexes may take. The time domain of a variable, when restricted to a given processor, corresponds to the space-time domain of the variable (i.e. the domain of the variable after the space-time change of basis has been performed, e.g. in Fig. 1(b)), where processor indexes are considered as parameters. the time indices vector ((t_1, t_2) on Fig 1(b)) is exactly the multi-dimensionnal counter we are looking for.

Eventually, this multi-dimensional clock counter has to be converted in a one dimensional clock which will ensure a correspondence with the counter of the physical clock of the architecture. The idea is to scan the convex union of all time domains and to provide activation signal (`write_enable`) for each variable. Among the various techniques which have been proposed to scan the integer points of a given polyhedron, Boulet and Feautrier [1] express the scanning program as a finite automaton. This method fits very well the context of hardware synthesis, as automata are efficiently mapped to hardware by synthesis tools.

```
     t1=p+1
     t2=2
     enableA=1
     enableB=1
     enableAcom=0
     enableC=1
2 if (t2<=N-1) then
          t2=t2+1
          GOTO 2
          endif
     t1=t1+1
     t2=2
// Acom start at p+2
     enableAcom=1
3 if (t2<=N-1) then
          t2=t2+1
          GOTO 3
          endif
  if (t1<=p+M) then
          t2=2
          t1=t1+1
          GOTO 3
          endif
     t1=t1+1
     t2=2
// Acom last up to p+M+1
     enableA=0
     enableB=0
     enableC=0
4 if (t2<=N-1) then
          t2=t2+1
          GOTO 4
          endif
1     // end of time
```

Figure 3. Automata used to enumerate time domains of the program of Fig. 1(b)

Hence, one can define the *largest time domain*, scan this domain with an automaton and generate for each variable an activation signal indicating whether the current scanned point belongs to the time domain of this variable. This new activation signal will control the loading of the memory elements storing this variable. This amounts to attach to each variable an individual *clock enable* signal, which is possible in most FPGA architectures.

A possible instance of a control automaton for the program of figure 1(b), is shown in Fig. 3. The `enableX` signals indicate the virtual cycles at which an instance of the array `X` has to be computed.

5: Implementation results

This section presents preliminary implementation results. The architecture of Fig. 2 was written in VHDL. To this end, this architecture was first expressed in Alpha and this Alpha program was then translated into VHDL by following systematic rules so that this translation can be easily implemented in the MmAlpha system (see [12]).

Implementing memories Memories are implemented by instantiating predefined components. Our abstraction for such component is the following: a memory has two (read and write) ports, and these ports are controlled with clock enables. We choose to use the predefined block Ram existing on the Xilinx platform which can be accessed in one clock cycle. Would the size available of these memory be not sufficient, one would have to use external memories which are bigger and slower; the design would then be much more complex because data would have to be *pre-fetched* in order not to slow down the execution.

Synthesis results The synthesis has been done for the matrix-matrix product program of Fig. 1(a) with the schedules of equations (1) (classical two dimensional array) and (2) (linear architecture with memory represented in Fig. 2). The target FPGA platform is a Xilinx Virtex XCV800. Area complexity is expressed in term of Slices (on CLB is two slices which contains itself 2 look up tables). Value chosen for the parameters where: $P = 6$, $N = 8$, $M = 10$. Coefficients of the matrices where chosen to be 8 bits integers.

	One cell of the array			Clock	Complete	Complete
	Control	Memory	Datapath	Cycle	execution	array
Multi-dimensional time (Fig. 2)	65 Slices	2 Ram blocks	26 Slices	16.5 ns	363 ns	581 Slices
Linear time	-	-	26 Slices	16.5 ns	2227 ns	1560 Slices

Table 1. Result of the synthesis of the Matrix-Matrix product of Fig. 1(b) for values $P = 4$, $N = 8$, $M = 10$.

These results show that the cost of the additional control is not negligeable (size of a cell approximately multiplied by 3), but the complete area is still decreasing. A good point is that this complex control mechanism does not affect the frequency which is mainly constrained by the datapath. Both control and datapath could be optimised further. The RAMs used in each cell are not included in the area complexity because these RAM blocks are already present on the chip: if not used, they are lost anyway.

6: Conclusion

We have shown that we could extend the classical systolic space-time mapping to multi-dimensional scheduled uniform recurrence equations. This raises the issue of mapping computations to memories. To our knowledge, our work is the first attempt to automate the hardware synthesis of multi-dimensional scheduled parallel programs. Wilde et al [2] consider the issue of generating control signals when control conditions are polynomials in the time counter. Our implementation of control is inspired from [1] and covers a larger variety of control signals, as parameters do not need to be fixed during controller generation, and can be used for other purposes such as address generation for instance. Some related papers can be found in the work on HPF [3, 15, 19] as well as the work on partitionning [4, 5, 7, 6, 10, 20].

Starting from an Alpha representation of uniform recurrences together with a k-dimensional schedule, we have shown that we can map such a program on a $n - k$ dimensional parallel systolic architecture, where each processor has local memories. We have proven that linear allocation functions and linear memory functions as obtained by previous research can be combined to obtain a memory function which maps the data to local memories. We have presented a method to generate a controller for this architecture, using an existing polyhedron scanning method. These theoretical results can be used to generate in a systematic way a VHDL description of these architectures. Our method was illustrated on the matrix-multiplication architecture, and VHDL code resulting of this method has been written, validated and the additional complexity (compare to a classical systolic design) has been evaluated.

The work presented here is already completed enough to be implemented in synthesis tools such as MmAlpha. Nevetherless, several question remain open, especially concerning the complexity of the solutions to problems such as the control of the architecture, the memory implementation, broadcast versus pipelined control information, etc., but also concerning many minor technical problems not solved here (unimodular completion of the schedule function, automatic detection of FIFOs, handling large memories outside the FPGA, etc.). All these problems request an implementation of the design methodology presented here because manual execution of the different steps (scheduling, memory function and allocation, control automaton generation, VHDL generation and simulation) is impossible. We therefore plan to implement this methodology in the MmAlpha environment.

References

[1] P. Boulet and P. Feautrier. Scanning polyhedra without do-loops. In *IEEE PACT*, pages 4–11, 1998.

[2] S. Bowden, D. Wilde, and S. Rajopadhye. Quadratic control in linear systolic arrays. In *IEEE International Conference on Application-specific Systems, Architectures and Processors*, July 2000.

[3] F. Coelho. Compiling dynamic mappings with array copies. In *Proceedings of the sixth ACM SIGPLAN symposium on Principles & practice of parallel programming*, pages 168–179. ACM Press, 1997.

[4] A. Darte. Regular partitioning for synthesizing fixed-size systolic arrays. *Integration, the VLSI Journal*, 12(3):293–304, 1991.

[5] A. Darte and B. R. e. F. V. R. Schreiber. A constructive solution to juggling problem in systolic array synthesis. Technical Report 1999-15, Laboratoire de L'informatique du parallélisme, 1999.

[6] A. Darte, R. Schreiber, B. R. Rau, and F. Vivien. Constructing and exploiting linear schedules with prescribed parallelism. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 7(1):159–172, 2002.

[7] S. Derrien and S. Rajopadhye. Loop tiling for reconfigurable accelerators. In *Eleventh Intl. Symp. on Field Programmable Logic (FPL'2001)*, 2001.

[8] F. Dupont de Dinechin, M. Manjunathaiah, T. Risset, and M. Spivey. Design of highly parallel architectures with alpha and handel. In *Forum on Specification & Design Langages*, Marseille, Sept. 2002.

[9] P. Feautrier. Some efficient solution to the affine scheduling problem, part II, multidimensional time. *Int. J. of Parallel Programming*, 21(6), Dec. 1992.

[10] D. Fimmel. Generation of scheduling functions supporting lsgp-partitioning. In *IEEE International Conference on Application-specific Systems, Architectures and Processors (ASAP 2000)*, Boston, July 2000.

[11] A.-C. Guillou, F. Quilleré, P. Quinton, S. Rajopadhye, and T. Risset. Hardware Design Methodology with the Alpha Language. In *FDL'01*, Lyon, France, Sept. 2001.

[12] A.-C. Guillou, P. Quinton, , and T. Risset. Hardware synthesis for multi-dimensional time (extended version). Technical Report ??, Irisa, Rennes, France, 2003.

[13] A.-C. Guillou, P. Quinton, T. Risset, and D. Massicotte. High Level Design of Digital Filters in Mobile Communications. DATE Design Contest 2001, Mar. 2001. Second place.

[14] R. Karp, R. Miller, and S. Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[15] K. Kennedy, N. Nedeljkovic, and A. Sethi. Efficient address generation for block-cyclic distributions. In *International Conference on Supercomputing*, pages 180–184, 1995.

[16] V. Lefebvre and P. Feautrier. Storage management in parallel programs. In *5th Euromicro Workshop on Parallel and Distributed Processing*, pages 181–188, London, January 1997. IEEE Computer Society Press.

[17] F. Quilleré and S. Rajopadhye. Optimizing memory usage in the polyhedral model. *ACM Trans. Prog. Lang. Syst.*, 22(5):773–815, sep 2000.

[18] P. Quinton and Y. Robert. *Algorithmes et architectures systoliques.* Masson, 1989. English translation by Prentice Hall, *Systolic Algorithms and Architectures*, 1991.

[19] W. Shang and J. Fortes. On time mapping of uniform dependence algorithms into lower dimensional processor arrays. *IEEE Transaction on parallel and distributed systems*, 3(3):350–363, May 1992.

[20] J. Teich and L. Thiele. Partitioning of processor arrays: A piecewise regular approach. *Integration: The VLSI Journal*, 14(3):297–332, Feb 1993.