Hardware Synthesis for Systems of Recurrence Equations with Multi Dimensional Schedule

Anne-Claire Guillou
Irisa
Campus de Beaulieu
35042 Rennes cedex
France

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 methods 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 requires the use of memories in the architecture. The synthesis of such an architecture requires the definition of an allocation function that maps the calculations on the processors, and memory functions that define where the data are stored during execution. As our approach targets custom VLSI architectures, we constrain the synthesis method to produce parallel architectures that that satisfy the *computer owns rule,* i.e., each processor computes the data which are stored in its local memory. We explain how to combine the allocation and memory functions in order to meet the computer owns rule, and we present an original mechanism for controlling the operation of the architecture. We detail the different steps needed to generate a HDL description of the architecture, and we illustrate our method on the matrix multiplication algorithm. We describe a structural VHDL program that has been derived and synthesized for a FPGA platform using these design principles. Our results show that the complexity added in each processor by the memories and the control is moderate and justifies in practice the use of such architectures.**

Keywords: High-level synthesis, systolic architecture, multi-dimensional scheduling, hardware synthesis for FPGA platforms, System on a Chip, parallel architectures for signal processing.

I. Introduction

In the context of System on Chip (SoC) design, the components that are assembled use many different design technologies ranging from general purpose processors to custom VLSI blocks. In order to reduce the time to market of such chips, the global design time needs to be shortened as much as possible, and of course, the most time-consuming design technologies have to be considered first: accelerating the design of custom blocks is therefore a key issue.

Today, custom VLSI blocks are mostly synthesized from a register transfer level description, using languages such as VHDL or VERILOG. Attempts to synthesize such elements from higher level descriptions, called *high-level synthesis* of *behavioural synthesis*, still belong to research, because tools based on this approach have not proved to be efficient enough.

The research presented in this paper belongs to this research domain. It is directed towards the automatic design of architectures for regular applications pertaining mainly to the signal processing domain. We start from recurrence equation specifications [14], [10] that we express using the Alpha language, and we use the MMAlpha environment [15] to synthesize systolic-like architectures. In this context, a typical design flow comprizes the following steps: uniformization of the recurrences, scheduling, mapping, and hardware generation. This design methodology has been prototyped and shown to be effective for kernel signal processing algorithms such as filters or basic linear algebra operations, but it needs to be extended to complex applications in order to become a useful tool for the design of special-purpose blocks.

Multi-dimensional scheduling [11] is one of the extensions worth to be considered for at least two reasons. First, some algorithms do not admit simple linear schedules, and they are therefore excluded de facto from the usual synthesis approach. Second, multi-dimensional scheduling reduces the number of processors – typically, by decreasing the dimension of the allocation function – and this can be seen as a special type of *partitioning* technique; this allows the method to span a larger range of target architectures, thus increasing its flexibility. Although multi-dimensional scheduling of recurrence equations is a well known technique, it has not been yet successfully applied to hardware design, because it raises several new issues such as memory management and control generation.

We start the paper by presenting a motivating example in Section II. Then we review in Section III the existing techniques and tools that constitute the background of our methodology. In Section IV, we present the main results of the paper: the combination of memory function and allocation functions, and a mechanism for controlling multi-dimensional scheduled architectures. Implementation results are presented in Section V: they give an idea of the additional complexity that results from the use of a multi-dimensional schedule. In Section VI, we compare our method to related work, and we conclude the paper by Section VII.

II. Motivating Example

The matrix-multiplication algorithm is often used to illustrate and 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. It is a set of *uniform recurrence equations* written in the Alpha language: each equation is a single assignment statement which defines one variable.

Most often, matrix-multiplication is mapped onto a 2-dimensional systolic array [22] using a linear – or affine, – schedule. Let $T_V[i,j,k]$ denote the schedule of variable V, where $V \in \{A, B, C\}$. The schedule function could be for example:

$$\begin{array}{ll} T_A[i,j,k] = T_B[i,j,k] = i+j+k & , \\ T_C[i,j,k] = i+j+k+1 & , \end{array} \qquad (1)$$

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

In practice [15], the architecture of Fig. 2 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 clocks ensures that the behavior of the architecture is really the one that is expected after scheduling, and in addition, it allows the architecture to be easily integrated as an IP [13] in a complex design.

```
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;
```

Fig. 1. Initial specification of the matrix-multiplication algorithm using the Alpha language

For reasons related to the size, the power consumption, or the throughput of the resulting architecture, a designer might prefer a multi-dimensional schedule for this program, for instance:

$$\begin{array}{ll} T_A[i,j,k] = T_B[i,j,k] = \begin{pmatrix} i+j \\ k \end{pmatrix} & , \\ T_C[i,j,k] = \begin{pmatrix} i+j \\ k+1 \end{pmatrix} & . \end{array} \qquad (2)$$

The lexicographic order imposed by this schedule is denoted as $\preceq$ in what follows, and $\prec$ denotes the corresponding strict order. It guarantees that the dependencies between computations are satisfied. Here for example, $C[i,j,k]$ is guaranteed to be computed strictly after $A[i-1,j-1,k]$ because

$$\begin{pmatrix} i+j-2 \\ k \end{pmatrix} \prec \begin{pmatrix} i+j \\ k+1 \end{pmatrix} \quad .$$

Fig. 3 shows the matrix-multiplication program of Fig. 1 once all variables have been re-indexed by a space-time transformation whose first two components are the schedule.

In spite of its power, multi-dimensional scheduling is seldom used in practice, since translating such a

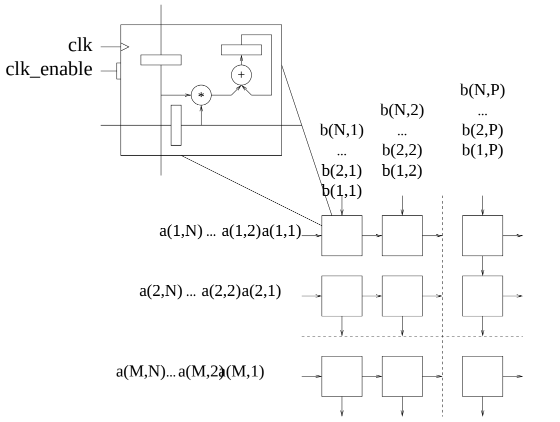


Fig. 2. Two-dimensional architecture for the matrix-multiplication with the linear schedule of equation (1)

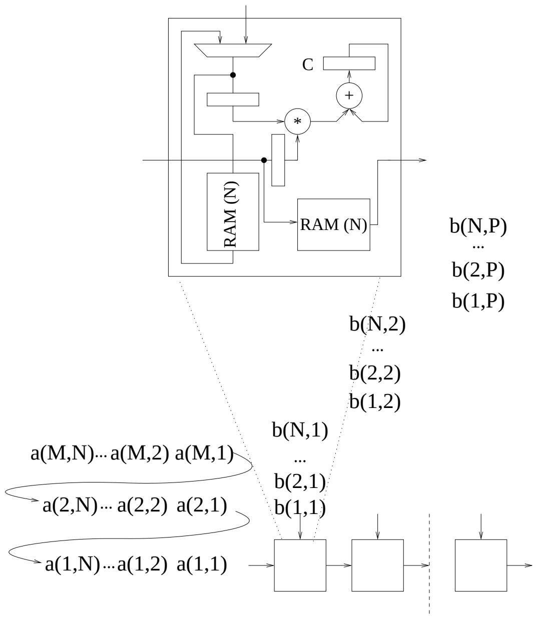


Fig. 4. Architecture for the matrix-multiplication with the multi-dimensional schedule of equation (2). Two random access memories of size $N-1$ are used to store rows of a and columns of b during the calculation.

```
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;
```

Fig. 3. Alpha system of Fig. 1 after applying the schedule of (2)

schedule into a real architecture is difficult. Indeed, establishing a correspondence between the *logical time* given by the schedule and the *physical time* in the chip is not simple: a *virtual clock* is not as easy to identify as for a linear schedule.

The multi-dimensional schedule of Equation (2) leads to the architecture sketched in Fig. 4 where computations carried at logical time (t_1, t_2) take place at virtual clock cycle number $(N+1)t_1 + t_2$. Fig. 4 also shows that memories are needed to store the data in each cell. Note also that the control is not displayed: the signals controlling the loading of the registers and memories are quite tricky to set up, and this issue will be solved in the remaining of the paper.

III. Basic concepts and techniques

In this section, we briefly review the different concepts and techniques that will be assembled to enable the automatic design of hardware for algorithms with multi-dimensional schedule. For a more complete description of these techniques, the readed is refered to [16], [11], [21].

A. *Systems of uniform recurrence equations*

Our initial specification is a *system of uniform recurrence equations*, i.e., a set of equations of the form:

$$z \in D_U : U[z] = \mathcal{F}(\ldots, V[z + d_{UV}], \ldots) \qquad (3)$$

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 all variables have the same dimension. These equations recursively define the values of the variables (U, V, etc.). For a given point z, we call *operation* the instance $V(z)$ of a variable.

We represent systems of recurrences by means of the Alpha programming language. For instance, the Alpha programs of Fig. 1 and Fig. 3 are uniform systems of recurrence equations. Alpha and its associated synthesis environment MMAlpha provide a framework for the synthesis of regular architectures. In particular, how to go from an *affine* system of recurrence equation to a *uniform* system of recurrence equation has been studied and implemented in the MMAlpha environment [19]. We have also prototyped the design method proposed here in this environment (see Section V).

B. *Multi-dimensional schedules*

A *multi-dimensional schedule* of a system of recurrence equations is a family of functions (one function for each variable U) $T_U(z) = \left[t_U^1(z), .., t_U^k(z)\right]$ such that $t_U^i(z)$ is an affine function of z 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 $T_V(z') \prec T_U(z)$. Moreover, we suppose that the schedule meets the following set of constraints.

1) For all variables V, the dimension of T_V is k.
2) All functions have the same linear part. In other words, for a given level i, $1 \le i \le k$, and for all variables V, the linear part of t_V^i is the same and we thus let $t_V^i(z) = t^i(z) + \alpha_V^i$. The $k \times n$ matrix T whose rows are the common linear part of the t^i fonctions is called the *schedule matrix*.
3) The rank of all schedule functions is exactly k. This is equivalent to say that the T function has a right inverse.
4) The schedule matrix T can be completed to form a $n \times n$ unimodular matrix.

For example, 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, thus $T = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$. A possible right inverse of T is $T^{-1}(t_1, t_2) = [t_1, 0, t_2]$. whose matrix is $T^{-1} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{pmatrix}$. (In the following, we shall sometimes identify a linear application with its matrix, whenever no confusion can occur.) Finally, the schedule can be completed into the function $M(i, j, k) = (i + j, k, j)$ whose matrix is unimodular.

C. *Allocation function*

Given an n-dimensional system of uniform recurrence equations with a k-dimensional schedule, an *allocation function* is a $(n - k)$ dimensional function $A(z) = [a_1(z), \ldots, a_{n-k}(z)]$ that specifies the coordinates of a processor where each computation is to be executed. In the example of Fig. 3, the allocation function $A(i, j, k) = j$ was chosen. Two computations allocated on the same processor must not be executed simultaneously. This condition is expressed by the following constraints between T an A:

$$\mathrm{Null}(T) \cap \mathrm{Null}(A) = \{0\} \quad . \qquad (4)$$

D. *Space-time transformation*

Once a schedule and an allocation function have been found, one can rewrite a system of recurrence equations by applying to all variables a *space-time transformation*: the initial indexes of each variable are replaced by their image by the schedule and the allocation function. For example, the system of Fig. 3 is the result of a space-time mapping of the program of Fig. 1.

Given a variable V, we call *space-time domain* of V, and denote $\mathrm{STD}(V)$, the domain of this variable after space-time transformation. For example, the space-time domain of variable A is

$$\mathrm{STD}(A) = \{t_1, t_2, p \,|\, p + 1 \le t_1 \le p + M\,; \\ 2 \le t_2 \le N\,; 1 \le p \le P\} \quad .$$

E. *Memory function*

Given a system of recurrence equations with schedule T, the *memory function* of a variable V is a $(n - m_V)$-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 during its whole lifetime. Memory functions can be built in a systematic way as shown in [21], [18] for example. A memory function is *valid* if it ensures that the value $V(z)$ is held in memory until it is not needed anymore. The difficulty is to find memory functions leading to small memories, possibly by re-using the same memory for different data.

When the schedule is mono-dimensional (i.e., $k = 1$), one can prove that $m_V = n - 1$. Therefore, memories correspond exactly to registers as data are stored only during a constant number of clock cycles.

If the schedule is multi-dimensional, we have in general $m_V \geq n - k$. In the example of Fig. 1 and for the schedule of Equation (2), the following memory functions are valid:

$$\begin{array}{l} M_{\mathsf{A}}(i,j,k) = (j, k-1) \quad , \\ M_{\mathsf{B}}(i,j,k) = (j, k-1) \quad , \\ M_{\mathsf{C}}(i,j,k) = (j) \quad . \end{array} \tag{5}$$

It is often convenient to express these functions in the space-time basis, i.e., in term of the indexes of Fig. 3. We have then:

$$\begin{array}{l} M_{\mathsf{A}}(t_1,t_2,p) = (p, t_2-1) \quad , \\ M_{\mathsf{B}}(t_1,t_2,p) = (p, t_2-1) \quad , \\ M_{\mathsf{C}}(t_1,t_2,p) = (p) \quad . \end{array} \tag{6}$$

In other words, in the program of Fig. 3, $\mathsf{A}[t_1, t_2, p]$ 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 used for variable B. All instances of variables C for a given processor are stored in a single memory element, which could be implemented using a simple register.

The schedule, allocation and memory functions define an *operational representation* of the program. Once the designer has choosen these functions, his work is to refine (or rewrite) the description with respect to a particular target architectural model (hardware or software) and hence to obtain an implementation *derived* from the initial functional representation.

IV. Methodology improvements for multi-dimensional scheduling

This section presents the original theoretical contribution of this paper: how to combine allocation and memory functions and how to control a multi-dimensional scheduled architecture.

A. Merging allocation and memory functions

The method proposed by Quilleré and Rajopadhye [21] assumes an underlying shared memory architecture model (we refer to this method as the SM method in what follows). In the context of custom VLSI, we want each processor of our architecture to store the values that it computes in its local memory, whenever these values need to be memorized; this policy is named the *computer owns rule* (in opposition to the well-known *owner computes rule* of parallel computing, where data stay in the memory of one processor, and processor compute the data which are located in their own memory.) In other words, data may be moving between different processors, but the processor which computes a data stores the data it has computed in its own memory as long as this data may be needed, and sends it when necessary to the processors that need it.

As a consequence, the $n - k$ first coordinates of the memory function of any variable must identify the place where the variable is computed. This adds to the constraints defining a memory function, as identified by the SM method, a new condition: its *first $n - k$ rows should be common to all variables and be equal to the allocation function*. In this section, we show how allocation functions and memory functions can be combined in order to meet the computer owns rule. We begin by recalling definitions and results given by [21], then we present an prove our main result, and finally, we illustrate the method on our matrix multiplication running example.

1) Background: In the following, we denote by $\text{Null}(M)$ the null-space of a linear mapping M, and by $\text{Vect}(v_1, \ldots, v_k)$ the linear subspace generated by vectors $v_1, \ldots, v_k$.

Let us denote by M_V the memory function of variable V. In the SM method, M_V is a *projection* which can be specified by a projection direction, that is to say by the m_V vectors of the null space of M_V. We use the following property shown in [21]:

Proposition 4.1: Memory functions satisfy

$$\text{Null}(T) \cap \text{Null}(M_V) = \{0\} \quad ,$$

hence, $m_V \leq k$, where k is the dimension of the schedule, i.e., the number of rows of T.

Proof: See [21] □

Given a variable V, let us define the *lifetime vector* d_V of V as 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. Notice that the first non zero component of a lifetime vector is allways positive, otherwise the schedule would not be causal.

Let $(\rho_1, \ldots, \rho_{m_V})$ be an integral basis of the null space of M_V. Quillere and Rajopadhye show the following proposition:

Proposition 4.2: Let T be the linear part of the schedule. Then $\{\rho_1, \dots, \rho_{m_V}\}$ are integral null space vectors of the memory function M_V if and only if for any integral linear combination $\eta = \sum_{i=1}^{m_V} \mu_i \rho_i$:

$$0 \prec T.\eta \Rightarrow d_V \preceq T.\eta \ . \qquad (7)$$

Proof: See [21] $\square$

2) Memory functions satisfying the computer owns rule: We now show that it is possible to find memory matrices of size $(n - m_v) \times n$ whose first $n - k$ rows have the same linear part. First, we show in Lemma 4.3 that we can choose null space vectors of M_V in such a way that they are also null space vectors of a common matrix 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.)

Second, we show in Lemma 4.4 that the memory function of this common V_0 variable can be re-arranged in such a way that its upper $n \times n$ sub-matrix is non singular.

Third, Lemma 4.5 shows a technical result used to extend the memory function of V_0 to any other variable.

Finally, Proposition 4.6 proves the main result.

Lemma 4.3: Given a system of recurrence equations with a multi-dimensional schedule, there exists a variable V_0 such that, for all variables V of the system, one can choose the vectors of the memory function M_V in the null space of M_{V_0}.

Proof: The proof builds upon a procedure for choosing the vectors of $\mathrm{Null}(M_V)$ which is explained in section 4.1 of [21].

The dimension m_V of $\mathrm{Null}(M_V)$ is the number of leading 0 of the lifetime d_V of V plus one (in our example, $d_A = (1, 0)$ hence, the dimension of $\mathrm{Null}(M_A)$ is 1). Let T^{-1} be a right-inverse of T. The method of [21] proposes the following null space vectors for M_V:

- $\rho_i = T^{-1}(e_i)$ for $1 \le i \le m_V - 1$, where e_i is the i^{th} canonical basis vector,
- $\rho_{m_V} = T^{-1}(d_V) \quad .$

Consider the following partition of the set of variables

$$\mathcal{V}_m = \{V | m_V = m\}, m = 1, 2, \dots, k \quad .$$

(In our example, for instance, $\mathcal{V}_1 = \{\mathtt{A}, \mathtt{B}\}$, $\mathcal{V}_2 = \{\mathtt{C}\}$.) Define $\max_{\mathrm{lex}}(E)$ as the lexicographic maximum of a set. Let now $\mathcal{D}_m$ be defined as follows:

$$\mathcal{D}_m = \begin{cases} T^{-1}(e_m) \text{ if } \mathcal{V}_m = \emptyset & , \\ T^{-1}(\max_{\mathrm{lex}}(\cup_{V \in \mathcal{V}_m} \{d_V\})) \text{ otherwise.} \end{cases}$$

In other words, $\mathcal{D}_m$ is the pre-image by the schedule function of the lexicographic maximum lifetime vector of the variables of $\mathcal{V}_m$. We now prove, using Proposition 4.2, that for any m the set $\{\mathcal{D}_i\}_{1 \le i \le m}$ is a valid set of null space vectors for the memory function of any variable $V \in \mathcal{V}_m$.

Consider a variable $V \in \mathcal{V}_m$, and let $\eta = \sum_{i=1}^{m} \eta_i \mathcal{D}_i$ be a linear combination of the $\mathcal{D}_i$ such that $0 \prec T.\eta$. We must prove that $d_V \preceq T.\eta$. Notice that $d_i = T.\mathcal{D}_i$ is the lifetime vector of some variable $V_i \in \mathcal{V}_i$, and therefore, its $i - 1$ first components are 0. On the other hand, we have

$$\begin{aligned} T.\eta &= \sum_{i=1}^{m} \eta_i T.\mathcal{D}_i \\ &= \sum_{i=1}^{m} \eta_i d_i \quad . \end{aligned}$$

Let η_n be the first non null coefficient of this sum. We consider two cases.

Case 1: $n < m$. Then the first non null component of $T.\eta$ is the first non null component of $\eta_n d_i$. But since by hypothesis, $0 \prec T\eta$, necessarily $0 \prec \eta_n d_n$. On the other hand, since $n < m$, and since the first $m - 1$ coordinates of d_V are 0, $d_V \preceq T.\eta$.

Case 2: $n = m$. By hypothesis and for the same reason

$$0 \prec \eta_m T.D_m = \eta_m d_m \quad .$$

Thus the first non null component of $\eta_m T.D_m$ is the first non null component of $\eta_m d_m$ and

$$d_V \preceq d_m \quad ,$$

by definition of d_m. As the first non null component of a lifetime vector is positive, and as $0 \prec \eta_m d_m$, necessarily, $\eta_m > 0$. Therefore, $d_V \preceq \eta_m d_m = T.\eta$, which proves that $\{\mathcal{D}_i\}_{1 \le i \le m}$ is a valid set of null space vectors for the memory function of any variable $V \in \mathcal{V}_m$.

Let now m_0 be the largest m such that $\mathcal{V}_m \neq \emptyset$, and let V_0 be any variable belonging to $\mathcal{V}_{m_0}$. Then, for any other variable V, the null-space vectors of M_V are also null-space vectors of M_{V_0}. $\square$ For our example, $d_A = d_B = (1, 0) = e_1$ and $d_C = (0, 1) = e_2$. The memory functions shown in equation (5) are such that $\mathrm{Null}(M_A) = \mathrm{Null}(M_B) =$

$\text{Vect}((1,0,0)) = \text{Vect}(T^{-1}e_1)$ and $\text{Null}(M_C) = \text{Vect}((1,0,0),(0,0,1))) = \text{Vect}(T^{-1}e_1, T^{-1}e_2)$.

Lemma 4.4: Let V_0 the variable selected by Lemma 4.3. It is possible to find a permutation of the rows of M_{V_0} such that the permuted matrix M'_{V_0} is a valid memory function for V_0 and the upper $n \times n$ submatrix of M'_{V_0} is non singular.

Proof: By Proposition 4.2, $\text{Null}(T) \cap \text{Null}(M_{V_0}) = \{0\}$. As a consequence, M_{V_0} is a $(n - m_{V_0}) \times n$ matrix (with $m_{V_0} \leq k$) such that the matrix $\begin{pmatrix} T \\ M_{V0} \end{pmatrix}$ is full-column rank. Therefore, it is possible to find a permutation of the rows of M_{V_0} which gives a new matrix M'_{V_0} such that the upper $n \times n$ square sub-matrix of $\begin{pmatrix} T \\ M'_{V_0} \end{pmatrix}$ is non-singular. As a permutation does not change the null space, $\text{Null}(M_{V_0}) = \text{Null}(M'_{V_0})$. Therefore, M'_{V_0} is still a valid memory function for V_0. Now let A be the sub-matrix of M'_{V_0} composed of its first $n-k$ rows. Then matrix $\begin{pmatrix} T \\ A \end{pmatrix}$ is non singular. Moreover, as $\text{Null}(T) \cap \text{Null}(A) = \{0\}$, A is a valid allocation function.

The following technical Lemma will be used to complete the A matrix into a memory function for all other variables.

Lemma 4.5: 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 $\text{Vect}(\rho_1, \ldots, \rho_{m_V}) \subset \text{Null}(A)$ ($k \geq m_V$), one can find a $(n - m_V) \times n$ full-row rank matrix M_V, built by completion of A, such that $\text{Null}(M_V) = \text{Vect}(\rho_1, \ldots, \rho_{m_V})$.

Proof: Consider the $n \times m_V$ matrix R whose columns are the ρ_i vectors. This matrix is clearly full column rank. Consider the Hermite normal decomposition of R:

$$R = (\rho_1 \ldots \rho_{m_V}) = UR'$$

where U is a $n \times n$ unimodular matrix and R' is $n \times m_V$ upper triangular. Let $A' = AU$. We claim that the columns of R' belong to $\text{Null}(A')$. Indeed, if we denote by ρ'_i the i^{th} column of R', then

$$A'\rho'_i = AU\rho'_i = A\rho_i = 0 \quad .$$

Hence as the columns of R' span exactly the first m_V dimensions of the space, A' is of the form $\begin{pmatrix} 0 & A'' \end{pmatrix}$, where A'' is a $(n-k) \times (n - m_V)$ full row rank matrix. Therefore, A'' can be completed with $k - m_V$ rows to obtain a square $n - m_V$ non singular integral matrix $A''' = \begin{pmatrix} A'' \\ X \end{pmatrix}$.

Consider now the $(n - m_V) \times n$ matrix

$$M'_V = \begin{pmatrix} 0 & A'' \\ 0 & X \end{pmatrix} \quad .$$

It is such that $Vect(\rho'_1, \ldots, \rho'_{m_V}) \subset \text{Null}(M'_V)$ because the last $n - m_V$ components of each ρ'_i are null. Moreover, we know that $dim(\text{Null}(M'_V)) = m_V$ because A''' is non singular. Thus, M'_V is full-row rank and $\text{Null}(M'_V) = Vect(\rho'_1, \ldots, \rho'_{m_V})$. If we note $M_V = M'_V U^{-1}$, then M_V is the matrix we are looking for. Indeed,

$$M_V \rho_i = M'_V U^{-1} \rho_i = M'_V \rho'_i = 0 \quad ,$$

and, as

$$\begin{pmatrix} 0 & A'' \end{pmatrix} U^{-1} = A'U^{-1} = AUU^{-1} = A \quad ,$$

M_V is a full row rank $(n - m_V) \times n$ matrix whose first $n-k$ rows are exactly A. □

Proposition 4.6: Let V_0 the variable chosen by application of Lemma 4.3, and A the allocation function resulting from application of Lemma 4.4. Then for any variable $V \neq V_0$ of the program, A can be completed into a valid memory function M_V of V.

Proof: By application of Lemma 4.3, the m_V vectors $(\rho_1, \ldots, \rho_{m_V})$ which generate the null space of the memory function of V are already in $\text{Null}(A)$.

Hence the problem amounts to finding for variable V 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.

Applying Lemma 4.5 to $(\rho_1, \ldots, \rho_{m_V})$ and A, we can extend A into a matrix M_V such that $\text{Null}(M_V) = \text{Vect}(\rho_1, \ldots, \rho_{m_V})$. It follows that M_V is a valid memory function for V. □

In summary, we have been able to find an allocation function A for the computations of the program and moreover, for all variables V, we found memory functions which have 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 $A(i_1, \ldots, i_n)$ and is stored in memory location $M(i_1, \ldots, i_n) = \begin{pmatrix} A(i_1, \ldots, i_n) \\ N_V(i_1, \ldots, i_n) \end{pmatrix}$. One can interpret this location as being a memory $N_V(i_1, \ldots, i_n)$ of processor $A(i_1, \ldots, i_n)$, which meets the computer owns rule.

3) Illustration of the method: We illustrate this method on our program of Fig. 1 with schedule of Equation (2). The lifetime vectors are $d_{\mathsf{C}} = (0,1)$, $d_{\mathsf{A}} = d_{\mathsf{B}} = (1,0)$. our method gives the following null

```
     t1=1
3    t2=1
2    if (t2 <= t1-1) then
          t2=t2+1
          GOTO 2
     endif
     if (t1 <= N-1) then
          t1=t1+1
          GOTO 3
      endif
1     // end of time
```

(a) Automaton for time domain of equation (9)

```
     t1=3
3    t2=2
2    //enable settings
     set all enable to 0
     if (p+2<=t1) && (t2<=N)  enableAcom <- 1
     if (t1<=p+M) && (t2<=N)  enableA <- 1
                              enableB <- 1
     if (t1<=p+M)             enableC <- 1
      //control settings
     if (t2<=N-1) then
          t2=t2+1
          GOTO 2
          endif
     if (t1<=p+M) then
          t1=t1+1
          GOTO 3
          endif
1     // end of time
```

(b) Automaton for controlling the architecture deduced of the program of Fig. 3

Fig. 5. Automata used to enumerate time domains.

space basis vectors: $\text{Null}(M_{\mathsf{C}}) = \{(1,0,0),(0,0,1)\}$, $\text{Null}(M_{\mathsf{A}}) = \text{Null}(M_{\mathsf{B}}) = \{(1,0,0)\}$. C is the variable with the minimal lifetime, a valid memory function matrix for C is $M_{\mathsf{C}}(i,j,k) = (j)$, this gives us the allocation function.

As explained above, we can complete the M_{C} matrix with one row to obtain M_{A} and M_{B}, which gives for instance: $M_{\mathsf{A}}(i,j,k) = M_{\mathsf{B}}(i,j,k) = (j,k-1)$. The resulting linear architecture is represented in Fig. 4. 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 $N-1$: $\mathsf{A}[i,j,k]$ is stored in processor j at memory location k.

B. Controllers for multi-dimensional schedules

We now turn to the problem of generating a controller for a multi-dimensional scheduling.

In a linear schedule, the resulting architecture can be controlled by a single counter which enumerates the time steps. To extend this idea to multi-dimensional time, we propose to control the architecture by means of a *multi-dimensional counter*. An illustration of this idea 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 spans a polyhedron of any shape. In the following, we propose to provide each processor with a simple hardware mechanism that implements this multi-dimensional counter. However, we must guarantee that in our solution, a given hour has the same number of minutes in every processor, in order to avoid the need of synchronizing the processors.

1) Time domains: Let us introduce the notion of *time domain* of a variable. For a given variable V of an Alpha program with a schedule T and an allocation A, we call *time domain* of V with respect to processor p, and we denote $\text{TD}(V,p)$ the values that its time indexes may take, for processor p. For instance, if a variable has the space time domain:

$$\text{TD}(V,p) = \{t_1, t_2, p \mid 1 \le t_2 \le t_1 \le p; 1 \le p \le N\} \tag{8}$$

its time domain in processor $p = 1$ is

$$\text{TD}(V,1) = \{t_1, t_2 \mid t_1 = 1;\ t_2 = 1\} \quad ,$$

and its time domain on processor $p = 2$ is

$$\text{TD}(V,2) = \{t_1, t_2 \mid 1 \le t_2 \le t_1 \le 2\} \quad .$$

We call *global time domain* of V, and we denote $\text{TD}(V)$, the union, over all processors p, of the time domains of V with respect to p. The global time domain of a variable can equivalently be obtained by projecting the space-time domain $\mathrm{STD}(V)$ of this variable on the time dimensions. For our example, the time domain of variable V is:

$$\mathrm{TD}(V) = \{t_1, t_2 \mid 1 \leq t_1, t_2 \leq N;\ t_2 \leq t_1\} \quad . \tag{9}$$

Similarly, we define the *global time domain* TD of a system as being the union of all the time domains of the variables of the program.

Our proposed multi-dimensional counter scans the global time domain of a system in lexicographic order, one new point at each clock cycle. In any given processor, it generates an enable signal for each variable. For example, for the above variable V with space-time domain of equation (8), each hour contains N minutes as indicated on the global time domain of equation (9), but on processor $p = 1$, V is only computed in the first minute of the first our, i.e., for $t_1 = 1$, $t_2 = 1$, and the computation of V must be cancelled on all other clock cycles. The `enableV` signal in processor $p = 1$ is true only when $t_1 = 1$ and $t_2 = 1$, and the loading of the computed value of V is only validated when the `enableV` signal is true. This signal acts exactly as a clock enable signal: the computation is performed in the combinatorial logic of the hardware, but the result is stored in the register of memory only when the `enableV` signal is valid.

2) Scanning polyhedra by means of automata: In this section, we briefly survey a method to generate the controller.

Among the various techniques which have been proposed to scan the integer points of a given polyhedron, Boulet and Feautrier [2] 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.

For the time domain presented in equation (9), the automaton is shown on Fig. 5(a) with the convention that the labels represent states and `goto` statements, transitions. A possible control automaton for the program of Fig. 3 is shown in Fig. 5(b). The exact setting of the automaton must be discussed further with respect to complexity of the resulting hardware (many optimization are already presented in [2]).

Assuming that such an automaton is available in each processor p, we dispose of several useful signals: two clock signals indicating new hours and new minutes, and two counters giving the current values of t_1 and t_2. The latter counter can be used for address generation for instance. In addition, we can also easily build a monodimensional clock clk which generates the actual clock cycles of the architecture. As explained in section II, one can implement clk using a *clock enable* signal controlling all memory elements, so that the entire architecture may be frozen during some time steps.

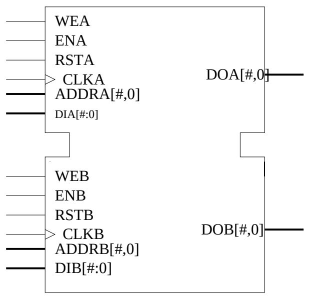
V. Putting it all together in Alpha

In this section, we describe how the results of Section IV is implemented in the Alpha language. Our goal is to provide an Alpha program which is as close as possible to the VHDL code we would write for implementing the architecture. In such a way, most of the transformations are done within the Alpha formalism and we make as few semantic changes during the translation to VHDL, which helps in performing correct designs. Here we describe how we *systematically* derive a new Alpha program from the one of Fig. 3 with explicit use of memories. We first present the program containing the memory of `A` and `B`, then we explain how the domains of the new program can be built using polyhedral computations and we detail a few optimizations. This Alpha program is then translated into VHDL by following systematic rules so that this translation can be easily implemented in the MMAlpha system. We presents results of the synthesis of the architecture and compare it with a classical systolic design for the same application (Fig. 2).

A. Implementing memories

Memories are usually implemented by instantiating predefined components which greatly depend of the target technology. As we primarily target FPGA technologies, we consider memories for reconfigurable platforms. There are usually several level of memories on FPGA chips. Memories can be implemented as FIFO using the Configurable Logic Blocks (CLBs) of the FPGA; recent chips provide additional predefined area for memory blocks of various kinds. These memory blocks can be accessed in one clock cycle, and have a limited storage capacity.

The left part of Fig. 6 shows for instance the interface of the *True Dual Port Synchronous Ram* available on a Virtex XCV800 chip. Based on this architecture, we abstract our vision of a memory as shown on the right part of Fig. 6: a memory has two (read and write) ports, and these ports are controlled by means of clock enables signals which come directly from the controller mentioned in section IV-B; these signals indicate at which virtual clock steps the data stored in the memory are either read or written.



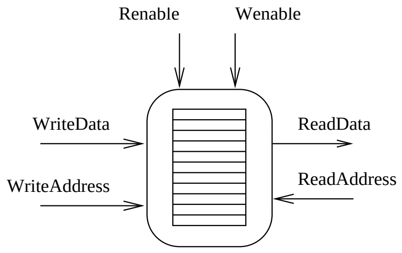


Fig. 6. Architecture of a dual port RAM as proposed on Xilinx Virtex chips (left part). The size of such a memory can be parameterized. The abstraction of such a memory in our design flow is shown on the right part

This choice brings several restrictions which could be easily overcome in future designs (but again, this depends largely on the target architecture implementation): one memory block is used for a single Alpha variable (sharing of resources is not considered); as the size needed to store the Alpha variable is not necessary a power of two, the size used on chip is the smallest power of two larger that the needed size. Would the size available on a one cycle access 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.

B. *Alpha code for the memories*

a) The A memory: Consider first the case of the `A` variable. Fig. 7 recalls the equations of the initial program (equations defining `A` and `C`) and Fig. 8 shows the corresponding elements in the final program, after memory interpretation. First, note that the transformation of the Alpha program has been chosen in such a way that each processor computes the variables it stores: each processor read its `A` memory, assigns it to `Acom` and ”sends” it to the next processors.

In the original program (Fig. 7), the `A` variable is read twice (`A[t1-1,t2,p]` in the definition of `Acom` and `A[t1,t2-1,p]` in the definition of `C`) and written once.

The memory function of `A` is $(p, t2-1)$, which means that, on processor p, `A[t1,t2,p]` is stored at address `t2-1`. Thus, a memory of size `N-1` is needed in each processor. In general, the adresses are expressed by a linear function of the time and space indices.

Address generation should be done by the controller of the program (seen in section V-E), and one can manage to have a single address generator shared by

```
... A : {t1,t2,p | p+1<=t1<=p+M;
         2<=t2<=N; 1<=p<=P} of real;
 let
...
  A[t1,t2,p] =
      case
        { | p=1} : a[t1-p,t2+p-1];
        { | 2<=p} : Acom[t1,t2,p-1];
      esac;
  Acom[t1,t2,p] = A[t1-1,t2,p];
  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;
... tel;
```

Fig. 7. Initial definition of variable `A`

every variables, because the initial program has uniform dependencies. We do not address the problem of efficient address generation here, the adaptation of classical compilation techniques such as strength reduction would provide efficient mechanism. Note, for instance that the second read occurrence `A[t1,t2-1,p]` is exactly the data that has been written in the `A` memory at previous `t2` clock cycle: hence this read does not require a memory access, and one can just store the written value of `A` in a register clocked on `t2`. This simplification is important because it allows one to implement the `A` memory with a dual port memory (one port for writing, one port for reading, see section V-D below).

The first read occurrence (`A[t1-1,t2,p]`) requires one address port named `ReadAddrA` and one data port named `ReadDataA`. Similarly, the write occurrence requires one address port named `WriteAddrA` and one data port named `WriteDataA`. The `A` memory is represented in Alpha as three variables named `MemA`,

```
  ...
 1  MemA_Prec :   {ad,t1,t2,p |1<=ad<=N-1; p+1<=t1<=p+M+1; 2<=t2<=N;  1<=p<=P} of real;
 2  MemA :        {ad,t1,t2,p |1<=ad<=N-1;  p+1<=t1<=p+M+1; 2<=t2<=N; 1<=p<=P} of real;
 3  MemA_Succ :   {ad,t1,t2,p |1<=ad<=N-1; p+1<=t1<=p+M; 2<=t2<=N;  1<=p<=P} of real;
 4  ReadAddrA :   {t1,t2,p | p+2<=t1<=p+M+1; 2<=t2<=N; 1<=p<=P} of integer;
 5  ReadDataAcom :{t1,t2,p | p+2<=t1<=p+M+1; 2<=t2<=N; 1<=p<=P} of real;
 6  WriteAddrA :  {t1,t2,p | p+1<=t1<=p+M; 2<=t2<=N; 1<=p<=P} of integer;
 7  WriteDataA :  {t1,t2,p | p+1<=t1<=p+M;  2<=t2<=N; 1<=p<=P} of real;
  ....
 8  ReadAddrA[t1,t2,p] = t2-1;
 9  WriteAddrA[t1,t2,p] =  t2-1;
10  Acom[t1,t2,p] = ReadDataAcom[t1,t2,p];
11  use {t1,t2,p |p+2<=t1<=p+M+1; 2<=t2<=N; 1<=p<=P}
12    Read[N-1](MemA_Prec,ReadAddrA) returns (ReadDataAcom);
13  WriteDataA[t1,t2,p] =
14       case
15        { | p=1} :a[t1-1,t2];
16        { | 2<=p} :Acom[t1,t2,p-1];
      esac;
18  use {t1,t2,p | p+1 <=t1<=p+M;  2<=t2<=N; 1<=p<=P}
      Write[N-1] (MemA_Prec, WriteAdrA, WriteDataA) returns (MemA_Succ);
20  MemA[ad,t1,t2,p]=
      case
         { | p+1<=t1<=p+M; 2<=t2<=N; 1<=p<=P}: MemA_Succ[ad,t1,t2,p];
         { | t1=p+M+1 } : MemA_Prec[ad,t1,t2,p];
       esac;
25  MemA_Prec[ad,t1,t2,p]=
      case
27      { | t2=2; t1=p+1} : 255[];
        { | t2=2; t1>p+1} : MemA[ad,t1-1,N,p];
        { | t2>2} : MemA[ad,t1,t2-1,p];
30       esac;
  ...
  tel;
```

Fig. 8. AlpHard code showing the implentation of the A memory

MemA_succ and MemA_prec. These variables are defined in Fig. 8 on lines 1-3: they have four indices, the first one being the address of each memory cell in each processor. These variables have the following meaning:

- MemA_Prec represents the state of the memory at the beginning of a clock cycle (or equivalently at the end of previous clock cycle);
- MemA represents the memory at the end of a cycle;
- MemA_Succ is a subset of memA and represents the memory cells that are written during the cycle.

Note that we made the choice of a cylindric extension for the memories variables (basically, as soon as one cell of the memory is alive, all the memory is alive).

Consider now the equations defining the use of the MemA memory. Two equations (lines 8 and 9) define the read and write addresses, which are given by the memory function. Line 11-12 is a call to a Read subsystem which emulates a read operation on the memory: its input variables are the value of the memory at the beginning of the cycle (MemA_Prec), the read address (ReadAddrA), and it returns the value of the read data (ReadDataAcom).

Lines 13 to 16 represent the definition of the data to be written in the memory, depending on the value of the processor number p. Line 18-19 represents a write operation on the memory. It takes the value of the memory at the beginning of the current cycle (MemA_Prec), the write address (WriteAddrA) and the data to be written (WriteDataA) and returns the value of the memory at the end of the cycle (MemA_Succ).

Lines 20-24 define the value of MemA: it is updated using MemA_Succ (for the part which is written during the current cycle) and with MemA_Prec (i.e. state of the memory at the end of the previous clock cycles) everywhere else (see section V-C below).

Lines 25 to 30 define the value of MemA_Prec:

MemA_Prec gets the value of MemA at the previous cycle. Line 27 is used for initialisation (before being written, memory words are initalized to an arbitrary value).

The final program (Fig. 8) represents both the behavioural and the structural implementation of the architecture:

- it is a behavioural description, in the sense that this program can be executed, and behaves functionally as the initial one;
- it is also a structural description, in the sense that its expressions can be translated directly into a hardware description langage to describe the use of a memory.

b) The B memory: The setting of the B memory is very similar to the setting of the A memory. The B variable has two read occurrences (B[t1-1,t2,p] and B[t1,t2-1,p]) and one write occurrence. As for the A case, the second read can be implemented as a register applied on the write occurrence. In addition, an obvious optimization can be performed here. It is very easy to see on the program of Fig. 3 that the B memory is initialized (with b[t1+t2-p-1,p]) and then *refreshed* with the same value at each write in the memory (the value read is just written at the place it has been read without any modification). Hence, in the particular case, the memory of B can just be written once at the beginning (for t1=p+1) and then only read. These *hidden refreshing* situations often occur because of the uniformization process used in the design methodology. Detecting, during uniformization, which dependencies should not be uniformized is difficult because they depend on the schedule which is usually found after uniformization. In the VHDL implementation that we present in Section V-E, we did the optimization.

C. Domain calculation

One tricky part of the translation process is how to compute the domains of the variables. This is done using the POLYLIB library.

In the original program of Fig. 3, domains of variables are *spatio-temporal domains*: indices are either time indices (t1,t2), or spatial indices (p). This is also true in the program after transformation in Fig. 8, except for the variable representing the memory for which domains have been extended with a new index (ad) representing the address in the memory of a given processor. For this program to be correct, we need to set up the spatio-temporal domains of all the variables precisely. We illustrate this with the A variable.

Given a read occurrence, we define its *read domain* as the domain on which it occurs (expression domain of the read expression). Similarly, the *write domain* of a variable is its declaration domain (expression domain of the write expression). The memory variables have to be defined on both read and write domains. Thus, their domain is the union of the read domains and of the write domain, extended with a new address dimension, which has the size of the memory. (see the definition of MemA for example.) For technical reasons related to the semantics of the Alpha language, we need two additional variables: MemA_Prec which has the same domain as the MemA variable (in a given clock cycle, we can read from and write to the memory, MemA_Prec represents the state of the memory before the write of the current cycle occurred) and MemA_Succ which corresponds to the write domain (i.e. MemA restricted to the domain where the memory is written at each step). The use Read occurs on the read domain and the use Write occurs on the write domain. At each clock cycle, the memA variable is updated with memA_succ (when a write occurs in the memory) and memA_prec (when no write occur in the memory).

In our example, the domain of the read occurrence A[t1-1,t2,p] is computed using the MMAlpha command: expDomain(A[t1-1,t2,p]). The result gives:

$$\begin{array}{rl}\{t_1,t_2,p\,| & p+2\le t_1\le p+M+1;\\ & 2\le t_2\le N;\ 1\le p\le P\}\end{array}\quad .$$

The definition domain of A is:

$$\begin{array}{rl}\{t_1,t_2,p\,| & p+1\le t_1\le p+M;\\ & 2\le t_2\le N;\ 1\le p\le P\}\end{array}\quad .$$

We do the union of these domains, and extedn it to the address set by adding new dimensions which correspond the memory function. The memory function of A is: $M_A[t1,t2,p]=t2-1$, its range is $\{ad \mid 1\le ad\le N-1\}$. All these computations can be done with elementary commands of the MMAlpha system. The result gives the domain of MemA (and of MemA_Prec):

$$\begin{array}{rl}\{\mathrm{ad},t_1,t_2,p\,| & 1\le \mathrm{ad}\le N-1;\ p+1\le t_1;\\ & t_1\le p+M+1; 2\le t-2\le N;\\ & 1\le p\le P\}\quad .\end{array}$$

Note that the write domain, i.e., the domain of MemA_succ, is different because the A memory is read but not written at each time (t_1,t_2) where $t_1 = p+M+1$.

D. Optimizations

As mentioned previously, many optimizations have to be associated with this systematic translation. We mention a few such optimization.

1) ROM detection: ROM detection occurs when, as for the `B` variable, the memory is written once and then only read until the end of the program. In such cases, the Alpha program describes a memory where the same data is written at every clock cycle. Usually the detection of such situation is easy, but it depends on the memory function. If, for instance the memory function for `B` had been $M_B(t1, t2, p) = t1 + t2$ (which is valid) then, the `B` data would have been moving at each hour and the systematic complete rewrite of the memory would have been necessary, which is obviously very bad for power consumption (the global behaviour of the program is unchanged). Most often, this problem appears because of the uniformization process. A solution to avoid it would be to compute the memory function using the program before uniformization, but using the schedule of the program obtained after uniformization.

2) Use of FIFOs: The memory functions proposed in [21] assume that a data stays in the memory location where it has been written, which forbids the use FIFOs in our designs. In order to model FIFO memories, other types of memory functions could be considered. This would lead to a solution simular to the one chosen in Pico [1].

3) Sharing address generation: In our example, another obvious optimization is the sharing of adress generation: all our read and write adresses for `A` and `B` occur at the same address, `t2-1`. As our programs are uniform, different accesses to the same variable at a given cock cycle differ by a constant translation. If this translation concerns the last time index – as for instance between the two accesses `A[t1,t2,p]` and `A[t1,t2-1,p]`, – one of them can be replaced by storing in a register the data used in the other one. This register is loaded at each clock cycle of the read domain.

If the translation concerns another time index – as for instance the two accesses `A[t1,t2,p]` and `A[t1-1,t2,p]`, – then those accesses *really* represent access of different data and several ports are needed. If we end up with more than two ports, it might be interesting to slow down the schedule so that these accesses occur on different clock cycles. This is a classical resource constrained problem.

E. The VHDL program

The VHDL code was manually written from the AlpHard program described above and the controller presented in Fig IV-A.3. The architecture is shown on Fig V-D.3 it is composed of a linear array of identical cells (except the first one), each cell containing a controller, a datapath and an address generator. Writing a cell of the array was quite easy because the memory access modeling in Alpha was very close to the corresponding VHDL expression. Fig 10 present a portion of the VHDL code for the first cell of the array which instantiates the `A` memory. This code is close to the code of Fig 8 and the correspondence can be easily made because the name of the different signals are preserved.

The behaviour of the controller can be systematically derived from Fig. IV-A.3. Our implementation is composed of two VHDL processes: one increasing the counter over the two clocks (`t1` and `t2`) and selecting the states, the other selecting the `write_enable` depending on the state. The address generator was trivial is our case (all addresses where equal to `t2-1`. In the general case, it might be more efficient to include the address generator in the controller to take advantage of an incremental update of the adresses at each steps.

F. Synthesis results

The synthesis has been done for the matrix-matrix product program of Fig. 1 with the schedules of equations (1) (classical array represented in Fig. 2) and (2) (linear architecture with memory represented in Fig. 4). The target FPGA platform is a Xilinx Virtex XCV800. The area complexity is expressed in term of Slices (one CLB contains two slices, each one containing two look-up tables). The value chosen for the parameters are $P = 6$, $N = 8$, $M = 10$; the coefficients of the matrices are 8 bits integers.

These results show that the cost of the additional control is not negligible, as the size of a cell is approximately multiplied by 3, but the total area is still decreasing. A good point is that this complex control mechanism does not affect the frequency which is mainly constrained by the data path. Both control and data path could be optimized 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 would be lost anyway.

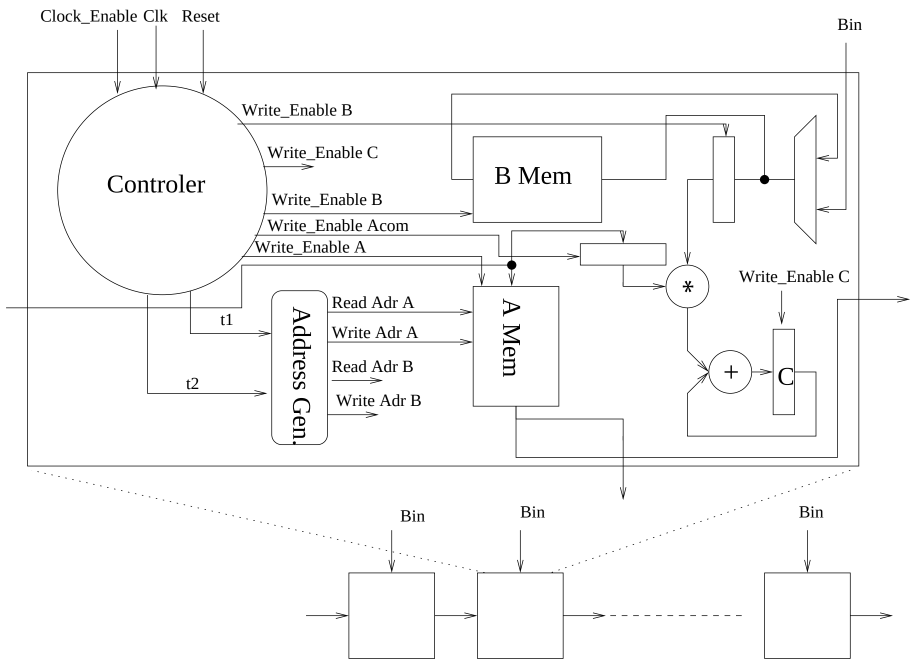


Fig. 9. Architecture of the hardware described in VHDL

```
...
WriteDataA <= A_In;
A_Out <= Acom;
Acom <= ReadDataAcom;

A_memory : RAMB4_S8_S8
  port map (ADDR1 => ReadAddrA,   ADDR2 => WriteAddrA,
            DI1   => gnd_bus,     DI2   => WriteDataA,
            WE1   => gnd,         WE2   => write_allowA
            CLK1  => CLK,         CLK2  => CLK,
            RST1  => gnd,         RST2  => gnd,
            EN1   => read_allowA, EN2   => pwr,
            DO1   => ReadDataAcom);
....
```

Fig. 10. Part of the VHDL code implementing the access to the A memory. Port 1 (first column of the port maps) is used for reading, port 2 is used for writing. Signals `read_allowA, write_allowA, ReadAddrA, WriteAddrA` are generated in the controller.

	One cell of the array			Clock	Execution	Full
	Control	Memory	Data path	Cycle	time	array
Multi-dimensional time (Fig. 4)	65 Slices	2 Ram blocks	26 Slices	16.5 ns	2227 ns	581 Slices
Linear time	-	-	26 Slices	16.5 ns	363 ns	1560 Slices

TABLE I

RESULT OF THE SYNTHESIS OF THE MATRIX-MATRIX PRODUCT OF FIG. 1 FOR VALUES $P = 4$, $N = 8$, $M = 10$

VI. RELATED WORK

This paper deals with several topics related to automatic hardware design: scheduling, computation allocation, memory allocation and HDL code generation.

It is also linked to research concerning efficient code generation for data parallel programs on SIMD or

SPMD architectures, as studied for High-Performance Fortran (HPF) compilers [4], [17] for example. However, results obtained for HPF cannot be applied directly, because they rely upon the strong assumption that communications between processors are much more (up to one thousand times more) costly than computations. Here to the contrary, we target custom single chip VLSI architectures, where processors are hard-wired, and communications between processors can be done in one clock cycle.

To our knowledge, our work is the first attempt to automate the hardware synthesis of multi-dimensional scheduled parallel programs, except for the development made in Pico [7] which targeted a particular class of multidimensional scheduling: systolic array with partitionning. They also made a different choice for memories: they used FIFOs where data is constantly moving, while we use classical memories. Our method also allow the parameterized design while parameter have to be fixed at design time in [7]. Wilde et al [3] consider the issue of generating control signals when control conditions are polynomials in the time counter. Our implementation of control is inspired from [2] 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.

Multi-dimensional scheduling has been introduced by Karp et al [16] and studied by Feautrier [11], and Darte [8] among others. The particular problem of scheduling uniform equations is however simpler than the general problem of scheduling affine systems of recurrence equations. Expressing the dependency constraints that must be satisfied by the schedule functions and optimizing the *dimension* of the schedule are well understood problems. Scheduling of both uniform or affine recurrences is implemented in MMAlpha and has been applied to real-life applications [20]. The problem of finding ”optimal” multi-dimensional schedules is however still open, and in MMAlpha we have adopted a *guided* scheduling method in which the user can either find the scheduling completely automatically or guide the scheduler by adding constraints that he/she wants the schedule to satisfy.

As far as computation allocation is concerned, most authors use linear schedules with $n-1$ dimensional allocation, combined with partitionning techniques [5], [6], [9], [7], [12], [24]. This approach allows the resulting architecture to meet constraints on available hardware or bandwidth resources. Some approaches propose to choose the allocation direction before scheduling [23], but this tends to be more complicated. Allocating computations of a system of recurrence equations with a multi-dimensional schedule simply adds the additional condition that $Ker(T) \cap Ker(A) = \{0\}$ [21], and does not differ much from the monodimensional schedule case. Combining our results with partitioning would be an interesting extension to investigate.

Two independent papers [18], [21] have shown how to find, given a multi-dimensional schedule, an efficient memory function for a shared memory. The memory functions targeted by these authors are more general than the one we have considered here, as they may contain modulo operations. Extending the results presented here to modulo functions is easy provided that the memory function is not combined with the allocation function as is done in this paper; indeed, it only amounts to adding a modulo operation to the address generator. A general extension to modulo memory functions remains to be done, however.

VII. Conclusion

We have shown that we can extend the classical systolic space-time mapping to multi-dimensionally scheduled uniform recurrence equations. This raises the issue of mapping computations to memories. 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. Nevertheless, several questions 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 require 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 painful. We therefore plan to implement this methodology in the MMAlpha environment.

References

[1] S. Aditya and M. Schlansker. ShiftQ: A Buffered Interconnect for Custom Loop Accelerators. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, Atlanta, Georgia, USA, 2001.

[2] P. Boulet and P. Feautrier. Scanning Polyhedra without Do-loops. In *IEEE PACT*, pages 4–11, 1998.

[3] S. Bowden, D. Wilde, and R. Rajopadhye. Quadratic Control in Linear Systolic Arrays. In *IEEE International Conference on Application-specific Systems, Architectures and Processors*, July 2000.

[4] 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.

[5] A. Darte. Regular Partitioning for Synthesizing Fixed-Size Systolic Arrays. *Integration, the VLSI Journal*, 12(3):293–304, 1991.

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

[7] 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.

[8] A. Darte and F. Vivien. Revisiting the Decomposition of Karp, Miller and Winograd. *Parallel Processing Letters*, 5(4):551–562, 1995.

[9] S. Derrien and S. Rajopadhye. Loop Tiling for Reconfigurable Accelerators. In *Eleventh Intl. Symp. on Field Programmable Logic (FPL'2001)*, 2001.

[10] 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.

[11] P. Feautrier. Some Efficient Solution to the Affine Scheduling Problem, Part II, Multidimensional Time. *Int. J. of Parallel Programming*, 21(6), Dec. 1992.

[12] 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.

[13] A. Fraboulet, T. Risset, and A. Scherrer. Cycle accurate simulation model generation for soc prototyping. In *Proc. of Samos IV*, Samos, Greece, July 2004.

[14] 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.

[15] 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.

[16] R. M. Karp, R. E. Miller, and S. Winograd. The Organization of Computations for Uniform Recurrence Equations. *Journal of the ACM*, 14(3):563–590, July 1967.

[17] K. Kennedy, N. Nedeljkovic, and A. Sethi. Efficient Address Generation for Block-Cyclic Distributions. In *International Conference on Supercomputing*, pages 180–184, 1995.

[18] 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.

[19] M. Manjunathaiah, G. Megson, T. Risset, and S. Rajopadhye. Uniformization of Affine Dependence Programs for Parallel Embedded System Design. In L. Ni and M. Valero, editors, *Internationnal Conference on Parallel Processing*, pages 205–213, 2001.

[20] A. Mozipo, D. Massicote, P. Quinton, and T. Risset. A Parallel Architecture for Adaptative Channel Equalization Based on Kalman Filter Using MMAlpha. In *1999 IEEE Canadian Conference on Electrical & Computer Engineering*, Calgary, Canada, May 1999.

[21] F. Quilleré and S. Rajopadhye. Optimizing Memory Usage in the Polyhedral Model. *ACM Toplas*, 22(5):773–815, Sept. 2000.

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

[23] W. Shang and J. Fortes. On Time Mapping of Uniform Dependence Algorithms into Lower Dimensional Processor Arrays. *IEEE Tr on Parallel and Distributed Systems*, 3(3):350–363, May 1992.

[24] J. Teich and L. Thiele. Partitioning of Processor Arrays: A Piecewise Regular Approach. *Integration: The VLSI Journal*, 14(3):297–332, Feb 1993.