Structured Scheduling of Recurrence Equations: Theory and Practice

Patrice Quinton[1] and Tanguy Risset[2]

[1] Irisa, Campus de Beaulieu, 35042 Rennes Cedex, France
{quinton}@irisa.fr
[2] LIP, ENS Lyon, 46 allée d'Italie
69364 Lyon Cedex 07
{trisset}@ens-lyon.fr

Abstract. We present new methods for scheduling structured systems of recurrence equations. We introduce the notion of structured dependence graph and structured scheduling. We show that the scheduling of recurrence equations leads to integer linear programs whose practical complexity is $O(n^3)$, where n is the number of constraints. We give new algorithms for computing linear and multi-dimensional structured scheduling, using existing techniques for scheduling non-structured systems of affine recurrence equations. We show that structured scheduling is more than one order of magnitude more efficient than the scheduling of corresponding inlined systems.

Keyword: parallelisation of loop nests, structured recurrence equations, scheduling, automatic synthesis of parallel architectures, parallel VLSI architectures.

1 Introduction

The synthesis of parallel programs or of architectures for the execution of loops has two main applications: the generation of parallel programs in order to accelerate applications, or the design of special purpose architectures for embedded systems. In both cases, the method is basically the same: isolate the loops which are good candidate to parallel implementation, extract the parallelism available in the loop, transform the loop into a parallel program, and finally, generate either a program or the description of a parallel architecture.

Loop analysis and transformation can be carried out either by keeping the imperative form of the code, or by transforming it into a single assignment program before applying the transformations. In the latter case, imperative code is rewritten as a set of recurrence equations, a formalism that was introduced by Karp, Miller and Winograd [1]. This approach has been taken in [2–5] in the context of the automatic parallelization of programs, and in [6–9] for the synthesis of regular arrays. Recurrence equations is also the basis of several applicative languages: Lucid [10], Lustre [11], Signal [12], Crystal [13], Pei [14] and Alpha [15], for example. All these languages aim at better supporting the formal derivation of a system from high level specification. Throughout this paper, we will consider Systems of Affine Recurrence Equations (SARE) and express them using the Alpha language, a precise description of which can be found in [16, 17].

Since recurrence equations are not imperative, their evaluation order relies upon a scheduler. This problem has been extensively studied [18, 19, 8, 3, 20] and results have been used in many parallelizing compilers such as Ape [21], Pips [22], Paf [23], Loopo [24], or in high level synthesis tools for the design of special-purpose architectures such as Arrest [25], Cathedral IV [26], Compaan [27], and MMAlpha [28]. Methods proposed all rely upon expressing the problem as an Integer Linear Program (ILP).

Structured specifications based on recurrence equation are obviously needed in order to deal with large, real-world applications. Primarily addressing the need for program structuring, the extension of recurrences to structured systems of recurrences described in [29] allows one to write and manipulate large specifications in a structured form. Therefore, to fully benefit from structuring, one needs to adapt existing analysis methods to structured systems, and this raises some new issues.

In order to find out a schedule for a structured system of recurrence equations, one can inline this system to obtain a single set of recurrence equations, and then schedule it using classical methods. But this approach has two drawbacks: it looses the information on the structure of the program, and it leads to ILP which become difficult to solve, as we shall see in this paper.

The other approach, which is taken in this paper, is to look for a *structured scheduling*. Basically, the idea is to use existing scheduling techniques to find out in several steps a schedule for the complete system. In this sense, we are looking for methods which extend the scheduling of SAREs to structured SAREs.

There are several ways to do so. For example, one can first schedule the called subsystems, then reuse the obtained schedules in the calling system. This approach corresponds to a divide and conquer strategy for scheduling. It has many advantages:

- The structure of the specifications, often meaningful from the point of view of the design flow, is kept unchanged.
- If some components of the system have already been designed, on can try to use their schedule and integrate them as such in the global design.
- The scheduling process is faster, as the problems to be solved are smaller.
- Finally, the schedules obtained in such a way are often easier to use in a design process. For instance, we can easily obtain a pipe-line between successive uses of the same system as shown in [30].

However, this raises new questions. For instance, when a SARE is used in several different contexts, it is not always possible to define a single schedule of this SARE suitable for all these contexts. For this reason, the scheduling of structured SAREs cannot be directly deduced from the scheduling techniques dealing with unstructured SAREs.

There are very few contributions reporting research on structured scheduling. A preliminary research to this paper [16] gives some theoretical results on structured scheduling. Later, Crop and Wilde [31] adressed the problem of scheduling SSAREs. Although it was not formally defined, the notion of structured scheduling considered is their paper correspond to the definition of *linear* structured scheduling that we provide here, but they do not handle mutlidimensionnal scheduling.

The particularity of the methods proposed here is that they use classical scheduling techniques (resolution of a linear programming problem) and hence can be directly implemented on top of a tool for the scheduling of SAREs.

The paper is organised as follows. Section 2 reviews the main results on scheduling of recurrence equations, and introduces the notion of structured dependence graph as an extension of the classical dependence graph. In section 3, we recall how a linear schedule can be modelized as an ILP, and we show on a set of benchmarks programs that the practical complexity of this problem is $O(n^3)$, where n is the number of constraints or variables. Section 4 addresses linear scheduling, and provides a systematic method for computing a structured linear scheduling for a structured SARE. Then, in section 5, this method is extended to find multi-dimensional structured scheduling. Finally, section 6 shows the practical efficiency of the structured scheduling by comparing it to the schedule of the corresponding inlined systems.

2 Background and definitions

In this section, we introduce the main notions and definitions needed to formalize the problem: affine recurrence equations, Alpha programs, scheduling, reduced dependence graph, and the extensions of these notions to structured systems of recurrence equations.

2.1 Affine recurrence equations

A System of Affine Recurrence Equation (SARE), is a finite number of equations of the form:

$$z \in \mathcal{D} \quad : \quad V_0(z) = f(V_0(I_0(z)), \ldots, V_p(I_p(z)))$$

where:

1. $\mathcal{D}$ is the *domain* of the equation: it is the set of points with integral coordinates included in a convex domain of Z^n.
2. n is the *dimension* of variable V_0.
3. For all i, I_i, the *dependency function*, is an integral affine function from Z^n to Z^p, where p is the dimension of variable V_i.

For example

$$\{k \mid k = 0\} : \mathtt{Acc}(k) = 0 \tag{1}$$

$$\{k \mid 1 < k \leq N\} : \mathtt{Acc}(k) = \mathtt{Acc}(k-1) + \mathtt{V1}(k) * \mathtt{V2}(k) \tag{2}$$

is a SARE which computes the dot product of two N-vectors `V1` and `V2`.

For a given variable V, we denote as $\mathrm{Dom}(V)$ the definition domain of this variable.

2.2 Alpha systems

The Alpha language [15] allows SAREs to be expressed. Fig. 1 show an Alpha program for the dot product of equations (1-2), with the addition of a variable `res` which isolates the result. In the rest of this paper, we will describe SAREs as Alpha systems.

```
system dot : {N | N >= 2}
  (V1,V2 : {k | 1<=k<=N} of integer)
returns (res:  integer);
var
  Acc : {k | 0<= k <= N } of integer;
let
  Acc[k] =
     case
        {| k = 0 } : 0[];
        {| k > 0 } : Acc[k-1]+V1[k]*V2[k];
     esac;
  res[] = Acc[N];
 tel;
```

Fig. 1. Alpha program for the dot product of two vectors

2.3 Scheduling SAREs

Scheduling a SARE aims at assigning an execution date to the computations specified by the system. As there are usually a very large number of such computations, we look for a closed form in order to express the schedule of a possibly infinite number of computations in a finite manner.

The formalism of recurrence equations naturally leads to the notion of *linear scheduling*: for each variable V of the system, the computation date is expressed as a linear function $T_{\mathtt{V}}(z) = \tau_{\mathtt{V}} \cdot z + \alpha_{\mathtt{V}}$ of its indexes. A linear schedule is valid if it is positive and respects dependencies between the computations. For instance, in equation (2), $\mathtt{Acc}(k)$ depends on $\mathtt{Acc}(k-1)$ for all k in $\{k \mid 1 < k \leq N\}$, hence a valid schedule $T_{\mathtt{Acc}}$ must satisfy the constraint $T_{\mathtt{Acc}}(k) - T_{\mathtt{Acc}}(k-1) > 0$. Therefore, $T_{\mathtt{Acc}}(k) = k$ is a valid linear schedule for this SARE, and a valid schedule for whole system dot is:

$$T^{\mathtt{dot}}_{\mathtt{V1}}(k) = 0 \quad T^{\mathtt{dot}}_{\mathtt{V2}}(k) = 0 \quad T^{\mathtt{dot}}_{\mathtt{Acc}}(k) = k \quad T^{\mathtt{dot}}_{\mathtt{res}} = 1 + N \quad . \qquad (3)$$

2.4 Reduced dependence graph

Scheduling techniques base their analysis on the data dependencies. For a SARE, the data dependency information is usually represented in the form of a (reduced) dependence graph.

Definition 1. *((reduced) dependence graph) The dependence graph (DG) of an Alpha system is a graph $G = (V, E)$ whose vertices V are Alpha variables and edges E represent dependencies between variables. Vertices are labeled by the variables names and domains, and edges are labeled by the dependency function and the definition domain of this function.*

As an example, Fig. 2 shows the dependence graph of the Alpha system of Fig. 1.

2.5 Structured systems of recurrence equations

The Alpha language provides a means to describe Structured Systems of Affine Recurrences Equations (SSARE) [29]. For instance, Fig. 3 shows the definition of a matrix multiplication MM by means of N^2 dot products. The third equation, called a use statement, instanciates N^2 times the dot product for all points (i, j) such that $1 \leq i, j \leq N$. The general form of a use statement in Alpha is the following:

$$\mathtt{use}\ \{J | AJ \leq b\}\ \mathtt{sub}[f(J,N)]\ (U_1, \ldots, U_n)\ \mathtt{return}\ (V_1, \ldots, V_m) \quad . \qquad (4)$$

This statement represents a multiple instanciation of the sub Alpha system, and it is made out of the following elements:

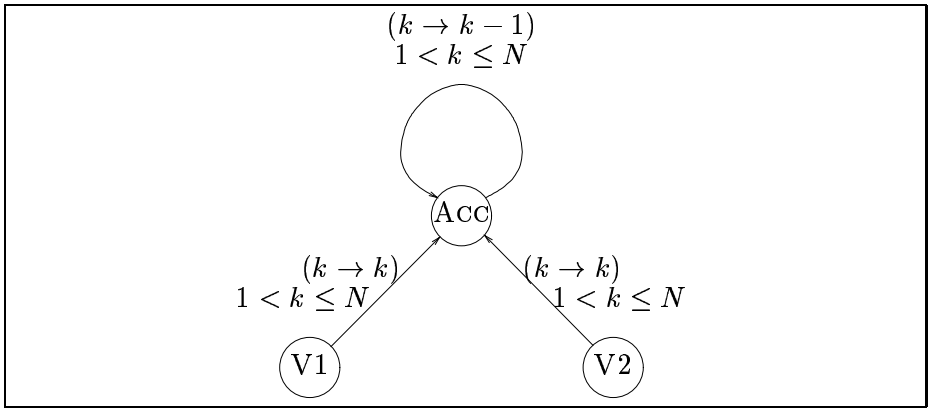

Fig. 2. Dependence graph of the Alpha system for the dot product

```
system MM : {N | N >= 2}
  (M1,M2 : {i,j | 1<= i,j <= N} of integer)
returns (Mres : {i,j | 1<= i,j <= N} of integer);
var
  Mdup1,Mdup2: {k,i,j | 1<= i,j,k <= N} of integer;
let
  Mdup1[k,i,j] = M1[i,k];
  Mdup2[k,i,j] = M2[k,j];
  use {i,j| 1<= i,j <= N} dot[N] (Mdup1,Mdup2) returns (Mres);
tel;
```

Fig. 3. Structured Alpha specification of the matrix product using N^2 dot products.

- The domain $\mathcal{E} = \{J|AJ \leq b\}$ is called the *extension domain* of the statement. It represents the instances of `sub` used. Note on Fig. 3 that, as the formal input of `dot` is one-dimensional and $\mathcal{E} = \{i, j \mid 1 \leq i, j \leq N\}$ is two-dimensional, the actual inputs of the use of `dot` are three-dimensional. In system `MM`, i and j are *extension indices*.
- The function $f(J, N)$, the *parameter assignment* function, gives the value of the size parameters of the called subsystem `sub` in term of the parameters of the caller system. In system `MM` of Figure 3, the parameter assignment function $f(i, j, N) = N$ simply transmits the size parameter N from `MM` to `dot`.

2.6 Structured scheduling

As the research on scheduling recurrence equations has always focused on unstructured recurrence equations, we have to give a formal definition of a *structured scheduling*. The notion that we introduce here allows one to reuse components of libraries of (already scheduled) systems.

Definition 2. *(*structured scheduling*) Consider an Alpha system* sys*, and a schedule* T^{sys} *of* sys*. Let*

$$\texttt{use } \{J|AJ \leq b\} \texttt{ sub[}f(J,N)\texttt{] } (U_1, \ldots, U_n) \texttt{ return } (V_1, \ldots, V_m) \qquad (5)$$

be a use statement in sys*, calling a subsystem* sub*. Denote by* u_i *(resp.* v_i*) the formal input (resp. output) variable of* sub *corresponding to* U_i *(resp.* v_i*). We say that* T^{sys} *is* structured *with respect to this use statement if there exists a schedule* T^{sub} *of* sub*, and a linear function* $c(J)$ *such that:*

$$\forall U_i, \forall (I,J) \in \mathrm{Dom}(U_i),\ T^{sys}_{U_i}(I,J) = c(J) + T^{sub}_{u_i}(I)$$
$$\forall V_i, \forall (I,J) \in \mathrm{Dom}(V_i),\ T^{sys}_{V_i}(I,J) = c(J) + T^{sub}_{v_i}(I)$$

We say that T^{sys} *is a structured scheduled if it is structured with respect to all use statements in* sys*.*

The key property of a structured schedule is that the c function is the same for all input and output parameters of a use statement.

To illustrate this definition, consider the example of the MM Alpha program shown in Fig. 3.

We have seen that a schedule for the dot system is

$$T^{\texttt{dot}}_{\texttt{V1}}(k) = 0 \quad T^{\texttt{dot}}_{\texttt{V2}}(k) = 0 \quad T^{\texttt{dot}}_{\texttt{Acc}}(k) = k \quad T^{\texttt{dot}}_{\texttt{res}} = 1 + N \quad . \qquad (6)$$

In program MM of Fig. 3, there are N^2 such calls to dot, each one being attached to a unique pair (i,j) in the extension domain $\{i,j \mid 1 \leq i,j \leq N\}$. For a given dot instance, say instance (i_1, j_1), scheduling this instance at time $c(i_1, j_1)$ means exactly that the inputs Mdup1[k,i_1,j_1] and Mdup2[k,i_1,j_1], $1 \leq k \leq N$ are available at time $c(i_1, j_1)$ and that, according to (3), the output Mres[i_1,j_1] is available at time $c(i_1, j_1) + N + 1$. Therefore, the schedule

$$\begin{aligned} T^{\texttt{MM}}_{\texttt{Mdup1}}(k,i,j) &= i + j \\ T^{\texttt{MM}}_{\texttt{Mdup2}}(k,i,j) &= i + j \\ T^{\texttt{MM}}_{\texttt{Mres}}(i,j) &= i + j + N + 1 \end{aligned} \qquad (7)$$

is a valid structured scheduling for system MM.

2.7 Structured dependence graph

We propose now an extension of the notion of dependence graph to structured SAREs.

Definition 3. *(structured dependence graph,* SDG*) The structured dependence graph of a structured Alpha system is a graph* $G = (V, E)$ *whose vertices* V *are labeled by the variables of the system and their domains, and edges* E *represent either:*

1. *dependencies between variables, these edges being labeled by the dependency function and the domain on which it applies;*
2. *dependencies between actual inputs and actual outputs of a use statement, these edges being labeled with the name of the instanciated system, the extension domain and the parameter assignment function.*

The structured dependence graph of the system of Fig. 3 is shown in Fig. 4. Note that, unlike the dependance graph, the structured dependance graph does

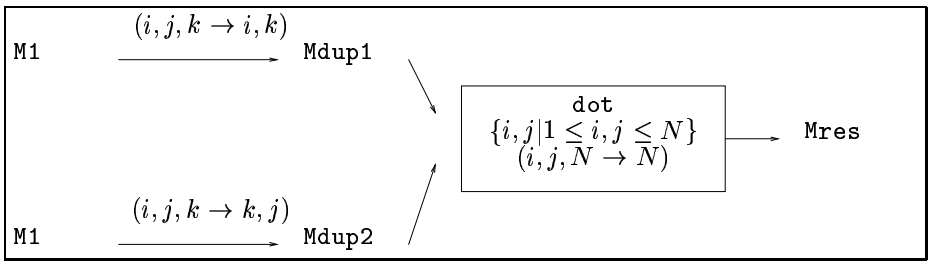

Fig. 4. Structured dependence graph of the Matrix product `MM` of Fig. 3. The rectangular box corresponds actually to 2 edges from `Mdup1` to `Mres` and `Mdup2` to `Mres`.

not represent all the information contained in the program. For example, in the SDG of Fig. 4, the dependency between inputs `Mdup1`, `Mdup2` and output `Mres` is not explicitely expressed. This is fair because this information could only be extracted by analyzing the body of the `dot` system.

3 The complexity of scheduling SAREs

In this section, we briefly explain how to find a schedule for a SARE, and we describe the complexity, both theoretical and practical, of this problem.

3.1 Method

Scheduling SARE was addressed in two contexts: the synthesis of systolic arrays [7–9] and the automatic parallelization of programs [3, 20, 4, 5, 32]. Scheduling uniform recurrences was first considered by Karp, Miller and Winograd [1]. Later on, extensions to linear recurrences [3, 19, 9] were examined. All these methods are based on results of linear programming. For instance, the method depicted in [3] uses the affine form of Farkas Lemma, and the one of [19] uses the duality theorem of linear programming.

In all cases, the problem is to solve an ILP, obtained in the following way.

Consider a recurrence equation

$$z \in \mathcal{D} \quad : \quad V_0(z) = f(V_0(I_0(z)), \ldots, V_p(I_p(z))) \quad . \tag{8}$$

To find out a linear schedule T_{V_0}, two types of constraints should be met :

- First, T_{V_0} must be positive on the domain $\mathcal{D}$. As $\mathcal{D}$ is a convex polyhedron, this property amounts to satisfy a finite set of linear inequalities, whose variables are the coefficients of the T_{V_0} function.
- Second, for all pair of dependent variables (V_0, V_j) in equation (8), the schedule of T_{V_0} and T_{V_j} must be such that $T_{V_0}(z) \geq T_{V_j}(I_j(z))$, for all point $z \in \mathcal{D}$. Again, this property is satisfied if a set of linear inequalities whose variables are the coefficients of T_{V_0} and T_{V_j} are met.

Finding out a schedule consists therefore of building the system of linear inequalities, and solving it using ILP. In practice, one chooses between all the valid schedules the one that minimizes the total time of the computation for a system.

3.2 Complexity of scheduling

To solve the ILP, one uses methods based on the simplex algorithm. The method of choice to solve this problem is the Parametric Integer Programming (PIP) method of Feautrier [33]: in all the experiments reported in this paper, this is the software that was used. It is well known that the number of pivoting steps of the integral simplex method is exponential in the number of variables (or linear contraints) of the problem [34], but when the simplex is run to find out a rational solution, the number of pivoting steps is in practice linear. Although we solve the scheduling ILP using an integral simplex, we shall see here that the behaviour of the algorithm is the same as the rational one. The reason is that most polyhedra we are dealing with have integral vertices.

To have a more precise idea of this complexity, we have run this method on a set of Alpha systems.

Table 1 gives the list of systems considered, and the number of vertices, constraints, and total scheduling time. These programs were run on a Sun Workstation. Fig. 5 gives a plot of the number of contraints in term of the number of variables, and shows that there is a very strong linear relationship between these variables. Fig. 6 gives a plot of the scheduling time in term of the number of variables. This curve was fitted to a cubic expression with a very high significance.

The conclusions of this experiment are three-fold:

1. First, the practical complexity of the scheduling fits near perfectly to that of the rational simplex algorithm. This shows that the problem has a very good behaviour.
2. Second, the $0(n^3)$ complexity is a very strong incentive to find out structured methods for solving the scheduling of systems, as we may save in practice a lot of time.
3. Finally, we can see that the scheduling time is reasonable for small systems, but becomes prohibitive in the context of automatic design of circuits for large systems.

X

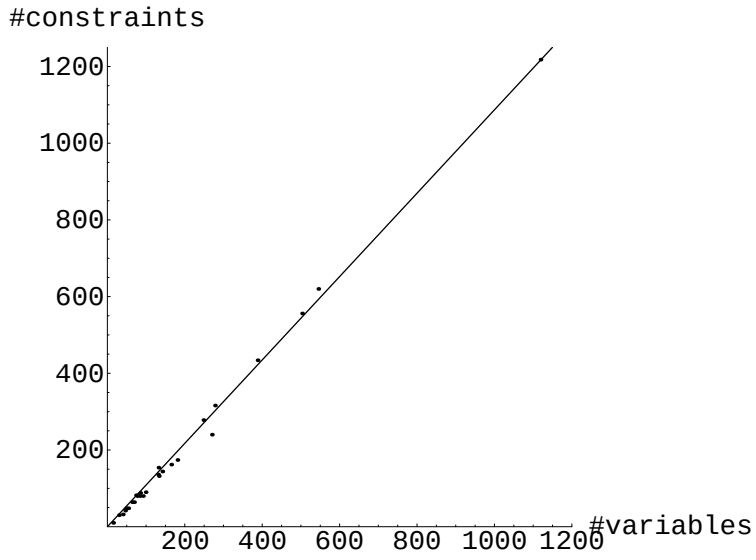

Fig. 5. Plot of the number of constraints against the number of variables for the scheduling ILP of the Alpha system of Table 1, together with the line $y = 1.087x$ which represents the best linear fit.

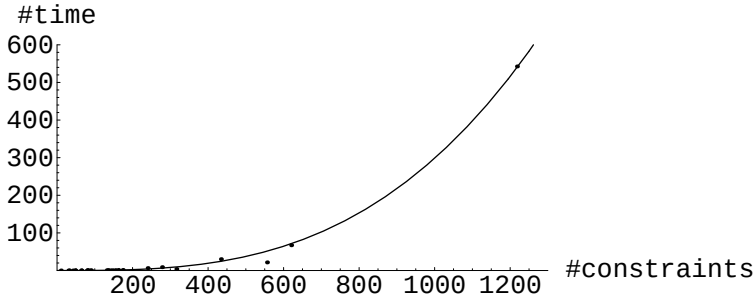

Fig. 6. Plot of the execution time of the scheduler against the number of constraints for the Alpha systems of Table 1, together with the curve $y = 2.9810^{-7}x^3$ which is the best cubic fit.

Program	#vertices	#constraints	Time (s)
Dot product Vector	74	82	0.14
Fir filter	84	80	0.25
Forward substitution	85	88	0.28
Full adder	40	32	0.03
Binary addition	181	174	1.48
Sum	99	90	0.26
ROM	131	136	0.94
Gaussian elimination	165	162	1.40
Givens rotations	388	434	30.13
Lower matrix vector	69	64	0.14
Matrix multiplication	92	80	0.19
Two matrix multiplication	142	144	0.72
Three matrix multiplication	270	240	6.39
Multiplexer	132	154	0.98
Fuzzy set22	1119	1218	542.56
Neural Network	545	620	67.21

Table 1. Result of a flat scheduler on a set of Alpha systems.

4 Linear scheduling of structured SAREs

In this section we study the following problem: given an Alpha structured program, how can we find a structured linear scheduling for this program using the same technique as that used for finding out linear scheduling for recurrence equations?

A necessary condition for this scheduling to exist is obviously that the inlined system has a linear schedule, but this condition is not sufficient. Indeed, our definition of structured scheduling imposes that any output of a use statement depends on any input of it: some of these constraints may not be necessary when one knows the exact structure of the subsystem called.

In this section, we propose an algorithm for finding out structured linear scheduling using a linear scheduler such as the one presented in section 3 [3]. We provide necessary and sufficient conditions to find a structured linear scheduling in the case of a bottom-up approach, i.e. when the called subsystems are scheduled before the calling system.

We consider successively two cases: when the SDG does not have cycles with use statement, and then the general case.

4.1 The SDG has no cycle with a use statement

In the program of Fig. 3 whose dependence graph is represented on Fig. 4, the structured dependence graph does not have dependence cycles with use statements: it corresponds to N^2 independent use of the system `dot`. A natural method to obtain a structured scheduling is therefore to schedule the `dot` system first, and then to deduce constraints for the scheduling of the `MM` system.

Based on these ideas, we propose a method for finding structured schedulings for a system `sys` whose dependence cycles do not contain use statements:

- First decompose the SDG into strongly connected components, each one of which contains at most one system call to `sub`.
- Then solve the scheduling problem for each component, in the topological order of the components of the graph.

Let us examine first the constraints that a linear structured scheduling of `sys` must satisfy. Given an edge of the SDG, two cases may occur:

- The edge is not labeled with a system name. In this case, the scheduling constraints are defined in the usual way, as defined for example by Feautrier in [3].
- The edge is labeled by a system name, i.e., the edge is part of a use statement of the form

$$\texttt{use } \{J|AJ \leq b\} \texttt{ sub[f(J,N)] } (U_1, \ldots, U_n) \texttt{ return } (V_1, \ldots, V_m) \quad . \tag{9}$$

  Let u_i and v_j be the formal arguments of system `sub` corresponding to U_i and V_j. Because of the restricted information contained in the SDG, the corresponding dependencies are: each output $V_j(I, J)$ of `sub` depends on each input $U_i(I', J)$, for all J in the extension domain $\mathcal{E}$ and for all $I \in \mathrm{Dom}(v_j)$ and $I' \in \mathrm{Dom}(u_i)$.

Actually, the following theorem shows that these constraints are exactly those needed to define a structured scheduling.

Theorem 1. *Let `sys` be an Alpha program with a call to subsystem `sub`. Let $T^{\mathtt{sub}}$ be a schedule of `sub`. Then the following constraints are necessary and sufficient to obtain a valid structured schedule $T^{\mathtt{sys}}$:*

$$\begin{array}{ll} \forall I \in \mathrm{Dom}(u_i), & T^{\mathtt{sys}}_{U_i}(I,0) = T^{\mathtt{sub}}_{u_i}(I) \\ \forall I \in \mathrm{Dom}(v_i), & T^{\mathtt{sys}}_{V_j}(I,0) = T^{\mathtt{sub}}_{v_i}(I) \\ \forall J \in \mathcal{E}, & T^{\mathtt{sys}}_{U_i}(0,J) - T^{\mathtt{sys}}_{V_j}(0,J) = T^{\mathtt{sub}}_{u_i}(0) - T^{\mathtt{sub}}_{v_i}(0) \end{array} \tag{10}$$

The proof of this theorem is given in appendix A.

To illustrate this result, consider again the example of Fig. 3. The constraints deduced from schedule (3) are:

$$\begin{array}{ll} \forall k, 1 \leq k \leq N, & T^{\mathtt{MM}}_{\mathtt{Mdup1}}(k,0,0) = T^{\mathtt{MM}}_{\mathtt{Mdup2}}(k,0,0) = 0 \\ & T^{\mathtt{MM}}_{\mathtt{Mres}}(0,0) = N+1, \\ \forall i,j, 1 \leq i,j \leq N, & T^{\mathtt{MM}}_{\mathtt{Mdup1}}(0,i,j) = T^{\mathtt{MM}}_{\mathtt{Mdup2}}(0,i,j) = T^{\mathtt{MM}}_{\mathtt{Mres}}(i,j) - N - 1 \end{array}$$

The scheduling of (7) meets these constraints. Another valid structured scheduling is:

$$\begin{array}{l} T^{\mathtt{MM}}_{\mathtt{Mdup1}}(k,i,j) = 0 \\ T^{\mathtt{MM}}_{\mathtt{Mdup2}}(k,i,j) = 0 \\ T^{\mathtt{MM}}_{\mathtt{Mres}}(i,j) = N+1 \end{array} \tag{11}$$

If the SDG do not contains cycles with use statements, the method presented here can be applied even if the schedule of the called system `sub` has not already been computed. In that case, when the computation dates of the actual inputs U_i to `sub` are known, one just computes the schedule of `sub` with the additional constraints on formal inputs u_i inherited from the computation dates of the actual inputs U_i. This top-down approach always provides a linear structured scheduling whenever then exists one and moreover, it chooses the fastest linear schedule if we try to minimize the execution time. Indeed, if there is no use statement in a cycle, each strongly connected component can be scheduled in minimum time and the total time is the sum of the execution times of the components. The interested reader may find more details in [16].

4.2 The SDG has cycles with use statements

In most reasonably complex structured programs, `use` statements will appear in cycles, i.e. there will be a repetitive use of a subsystem. An example of such program is shown in Fig. 7: this program computes the product of `P` square matrices using `P-1` times the `MM` matrix-matrix product of Fig. 3.

If we want to schedule such a program and if we restrict ourselves to linear scheduling, the method is very similar to the one explained above:

- First compute the schedules of all subsystems called.
- Then schedule the calling system with the additional constraints provided by equation (10).

This bottom-up approach has one drawback: depending on the schedule choosen for the subsystem, the calling system may or may not have a linear scheduling. Hence, the scheduling algorithm would have to be a trial and error process where the scheduling of the called subsystem is changed until a satisfying solution is obtained for the calling system. However, one can cut this trial and error process with the result presented in Theorem 2 hereafter.

But first, we need to introduce the notion of separate dependencies. Consider again a calling system `sys` and a subsystem `sub` used in `sys` (`use` equation in 5). Consider a dependency from an output of `sub` to an input of `sub`:

$$U_i[I, J] \leftarrow V_j[f(I, J), g(I, J)] \tag{12}$$

where J denotes the vector of extension indices (we have splitted the dependency into two parts, f and g, where g corresponds to the extension indices introduced in section 2.6). If $f(I, J)$ really depends upon J, then the time-shift between different inputs to `sub` may vary with the instance executed, as it may depend on J. Similarly, if $g(I, J)$ really depend upon I, then a single variable U_i input to one subsystem may depend on several other instances of the same subsystem. In both cases, it may be impossible to use the same schedule for all instances of `sub`.

There is a situation when this problem disappears. Consider dependencies of the form:

$$U_i[I, J] \leftarrow V_j[f(I), g(J)] \quad . \tag{13}$$

```
system MMM: {M,P |P,M>=2}
  (matList : {i,j,p | 1<=i,j<=M; 1<=p<=P} of real)
returns
  (res : {i,j | 1<=i,j<=M } of real);
var
  matAcc:       {i,j,p | 1<=i,j<=M; 2<=p<=P} of real;
  matAccNext:   {i,j,p | 1<=i,j<=M; 2<=p<=P} of real;
let
  matAcc[i,j,p] = case
        {|p=2} : matList[i,j,p-1];
        {|p>2} : matAccNext[i,j,p-1];
      esac;
  use {p| 2<=p<=P} matmult[M] (matList,matAcc) returns (matAccNext);
  res[i,j] = matAccNext[i,j,P];
tel;
```

Fig. 7. Structured Alpha specification of the product of a serie of `P` square matrices

where $f(I)$ depends only on I and $g(J)$ on J. If all dependencies from outputs to inputs of called subsystems have this form, we say that the system has the *separated dependency property*. In practice, all the Alpha structured programs have this property, as this is the most intuitive way of expressing structuring. The following theorem indicates exactly when a linear structured scheduling exists.

Theorem 2. *If a system* `sys` *has the separated dependency property for all its calls, then it admits a linear structured scheduling only if, for each subsystem call (to* `sub`*) contained in a cycle of the* SDG*, on each such dependence cycle the time delay of the path from an input to an output of subsystem* `sub` *is constant.*

The proof of this theorem is given in appendix B. This condition is very restrictive, but this is not surprising because most of the programs containing a `use` statement in a cycle have a linear time schedule. Indeed, consider the program of Fig. 7, and say that the schedule of system `MM` is the one presented in equation (10), i.e., an earliest time schedule. Computing `P` successive such matrix products will last `P*(N+1)` unit of times, which is impossible with a linear schedule.

We now propose an algorithm for computing a linear structured scheduling of an Alpha system `sys`.

Algorithm A

(Given: a system* `sys`*, and a call to subsystem* `sub` **)*
Perform a topological sort of the structured dependence graph of `sys`
`for` *each component c (in topological order)* `do`
    determine the constraints of the input to the component
    `for each` *call C to a subsystem* `sub` *in c (written as (5))* `do`

build the set $\mathcal{IO}_C$ *of pairs* (U_i, V_j) *such*
that there is a dependence from V_j *to* U_i *in* `sys` *as in (12)*
schedule `sub` *by Algorithm A, with constraints from input to the*
component and minimization of the latency between
the consumption of U_i *and the production of* V_j *for* $(U_i, V_j) \in \mathcal{IO}_C$
If no schedule is found `then return` *fail*
`endfor`
schedule c using the schedule found for `sub` *in constraints (10)*
If no schedule is found `then return` *fail*
`endfor`

This algorithm is valid, because we have proven that constraints (10) are sufficient to provide a structured linear schedule.

Algorithm A can easily be modified in order to use the same schedule for different calls to the same system `sub`. To do so, we replace the second schedule of `sub` in algorithm A by the first schedule found for `sub`, thus giving priority to the first call in the SDG.

As already said, algorithm A may fail, for instance, if the schedule is non linear. In the next section, we look for non-linear structured scheduling.

5 Structured multi-dimensional scheduling

In the context of loop parallelization, non linear scheduling has been studied through *multi-dimensional* scheduling. In this section, we present methods to obtain structured multi-dimensional schedules, after introducing the background material necessary to formalize this notion.

5.1 Multi-dimensional scheduling

The principles of multi-dimensional scheduling were settled down by Karp, Miller and Winograd [1] and explored further in [20, 35]. The idea is as follows. Consider an Alpha system which does not have a linear schedule. This means that no linear expression on the indices provides a partial order that respects the dependencies of the system. Instead of looking for a unique linear expression T_V associated with each variable, we look for a vector of linear expressions upon the indices in order to represent the computation date. Let $\ll$ denote the lexicographic order between vectors, i.e.:

$$X \ll Y \text{ if there exists } i \text{ such that } \begin{cases} x_k = y_k, \forall k < i \\ x_k < y_k, \text{if } k = i \end{cases} .$$

Then, if U depends on V in the SARE, we impose that $\forall z \in \mathrm{Dom}(U)$, $T_U(z) \ll T_V(z)$.

For instance, the following multi-dimensional schedule is valid for the program of Fig. 3:

$$T_{\texttt{Mdup1}}(i,j,k) = T_{\texttt{Mdup2}}(i,j,k) = \begin{pmatrix} i \\ k \end{pmatrix} \quad ,$$

$$T_{\texttt{Mres}}(i,j) = \begin{pmatrix} i \\ N+1 \end{pmatrix} \quad .$$

This schedule has two dimensions. It does not really give an absolute date of computation for each variable but merely a relative order: first compute the coefficients of the first row of `Mres` ($i = 1$), then when this is finished, compute the coefficients of the second row ($i = 2$, because $\forall j, k,\ T_{\texttt{Mres}}[1,j] \ll T_{\texttt{Mdup1}}[2,j,k]$) and so on. As an intuitive explanation of what a multi-dimensional schedule represents, one can interpret the components of the time vector as hours, minutes, seconds, etc. (Notice however that this interpretation is realistic only if all the variables have the same rectangular domains, which is not always the case.)

5.2 Computing multi-dimensional schedules

Computing multi-dimensional schedules is based on the following idea. If a dependency, say for instance

$$\forall i,j,k, T_{\texttt{Mres}}(i,j) - T_{\texttt{Mdup1}}(i,j,k) \geq 1 \tag{14}$$

cannot be met using a linear schedule, we introduce an additional dimension in the timing vector, and T becomes thus $\begin{pmatrix} T^1 \\ T^2 \end{pmatrix}$. Then we try to satisfy the relaxed constraint

$$\forall i,j,k, T^1_{\texttt{Mres}}(i,j) - T^1_{\texttt{Mdup1}}(i,j,k) \geq 0$$

by means of the first component of T and leave it up to T^2 to *strictly satisfy* the strict dependency (14). We say that this dependency is *satisfied at level 2.*

The schedule shown in (14) respects these constraints on T^1 and T^2. Hence, computing multi-dimensional schedules can be done with a linear scheduler provided that we implement the process described here. Feautrier [20] does this by minimizing the number of dimensions of the resulting schedule.

Multi-dimensional scheduling provides a lot of flexibility in the loop parallelization process. However, the space of possible valid multi-dimensional schedules is huge and few strategies have yet been proposed, which limits the practical use of such methods.

In the next paragraph, we propose a strategy for finding structured multidimensional scheduling.

5.3 Multi-dimensional schedule and structuring

In section 4, we were looking for linear schedules by enforcing constraints on the computation dates of inputs and outputs of system calls. In the multi-dimensional case, we do not know in advance which dependency is satisfied

at which level, hence, it is impossible to enforce values for timing vectors, and we must propose another method.

On way of doing so is to assume that each call to a subsystem takes no time, with respect to the schedule of the calling system: in other words, the inputs and outputs are assumed to be available at the same instant of time. Then, in order to satisfy the dependency constraints inside the subsystem, it suffices to add extra dimensions to the time vector. This method is always applicable to Alpha programs: indeed, as two systems cannot call each other, the graph of the system calls of an Alpha program is always acyclic.

Let us formalize this idea. Assume that we have a procedure (Algorithm A for example) that computes a linear structured scheduling. Given an Alpha system, the following algorithm tries to find out a structured multidimensional scheduling.

Algorithm B

1. *Decompose the* SDG *of* `sys` *in strongly connected components*
2. `for each` *component c of the* SDG `do`
   determine the constraints on the inputs of the component
   compute a schedule T_c for this component by Algorithm A
   `if` *no schedule is found* `then do`
   `for each` *system call to s_i of the component* `do`
   schedule s_i with Algorithm B and get a k_i-dimensional schedule T_c^i
   `end for`
   get a k-dimensional schedule T_c' of the component c with the constraints that inputs and outputs of a system call have exactly the same schedule
   Build a $k + \max_i(k_i)$-dimensional schedule $T_c = \begin{pmatrix} T_c' \\ T_c^i \end{pmatrix}$ (padd T_c^i with 0's if needed)
   `end if`
   `enddo`
3. *If the T_c schedules have different dimensions for different c, padd the smaller ones with 0's.*

Notice that in this algorithm, each call to a subsystem may have a different schedule.

5.4 Validity of Algorithm B

In this section, we prove that the schedules obtained by algorithm B are valid structured scheduling, i.e. that all dependencies are met by the schedule found. We have already proven that algorithm A provides valid linear schedules. Thus, it remains to prove that the schedule found by algorithm B when algorithm A

fails satisfies the dependencies inside a strongly connected component and that dependencies between strongly connected components are satisfied.

Inside a strongly connected component c, dependencies in a subsystem `sub` (from inputs to outputs) are satisfied because all variables have the same coefficients in T'_c, and the recursive use of algorithm B provides coefficients of T^i_c that satisfy dependencies (a simple induction argument would be needed to prove this, but we assume that this property is clear enough not prove it so formally). Inside a strongly connected component `c`, a dependency occurring in the `sys` system is ensured by T'_c. As all our dependence constraints are *strict* (i.e. we forbid computation which last 0, except for the calls), appending T^i_c (and possibly 0's) at the end, will not change the lexicographic order.

Dependencies between strongly connected components are satisfied because, before scheduling one component, we compute the constraints (on the first level of schedule) derived from the schedules of all previous (in the topological order sense) components. Hence, the addition of 0's performed in step *3* of Algorithm B concern variables for which dependency constraints have been (strictly) satisfied at a higher level.

Algorithm B provides one possible strategy for structured multi-dimensional scheduling. Of course this approach has drawbacks. First, as for the linear case of section 4, this will not work on all Alpha program which admits multi-dimensional schedules. Again, this is due to the way our SDG models dependencies inside a system call: some outputs may not depend on some inputs, hence a cycle in the SDG may not be a real cycle. For instance, if the dependency occurring from an output to an input is the identity, algoritm B will fail. But, let us point out that writing a call where inputs depend on outputs is not coherent with the usual notion of structuring: this will never happen if, for instance, the SAREs come from imperative code which has been automatically translated.

Another limitation of our approach is that it forbids the possible pipeline between successive instances of call of the same (or different) system: the output of a system cannot be used before the system is completely executed. This is a limitation in VLSI synthesis since pipeline is very often used. In our approach, pipeline can only be achieved with linear schedules obtained by algorithm A.

6 Practical efficiency of structured scheduling

To establish the practical performance of structured scheduling, we have run the structured scheduler on 6 benchmark programs, and we have run the non structured scheduler on the sames programs after fully inlining the called subsystems.

In all cases, the schedules obtained by both techniques were the sames, up to a factor which is due to the difference of structuration of the programs. Indeed, the default option of our scheduler is that the computation of each equation costs one unit of time, and therefore, splitting an equation may introduce extra artificial delays: a careful setting of the delays allows such differences to disappear. Table 2 gives the time needed to schedule these programs. It clearly shows that there is a gain of more than one order of magnitude when running a structured scheduling.

Program	Structured scheduling	Inlined scheduling
Binary adder	0.26	1.48
Gaussian elimination	0.42	1.40
Three matrix multiplications	0.42	6.39
Neural network	5.36	67.21
Fuzzy set	9.48	542.56
One step of SVD	4.21	21.59

Table 2. Comparison of the structured scheduling and inlined scheduling

7 Conclusion

We have presented the basic concepts of structured scheduling for recurrence equations. We have given a precise definition of the structured dependence graph for a structured SARE, and of the notion of structured scheduling. With respect to these definitions, we have presented an algorithm for finding a valid (linear or multi-dimensional) structured scheduling. The main interest of this approach is that it permits a scheduling tool for unstructured SAREs to be reused, and thus can be implemented very quickly.

We have also proven that, if we are looking for linear structured scheduling with a "bottom-up" approach, our algorithm was *optimal* in the sense that if ever there exists a linear structured scheduling, we will find it.

We have shown that a structured scheduler gives in practice the same results than scheduling the inlined program, with a much lower complexity.

Structured scheduling allows the structuring information to be kept and reduces the scheduling computation complexity. It also provides more readable and natural schedulings.

This work has been done in order to provide a strategy for scheduling large Alpha systems in the MMAlpha environment. Many open questions remain. First, we do not know if it is possible to statically determine all the cases where a structured SARE admits a structured linear scheduling (neither for structured multi-dimensional scheduling of course). Algorithm B proposed for multi-dimensional structured scheduling is one possible method among many variants, and it remains to compare it to other approaches. In particular, in this work, we wanted to use an existing scheduler as the basis of the structured scheduler. If we get rid of this implementation constraint, it may be possible to define completely different scheduling algorithms.

References

1. Karp, R., Miller, R., Winograd, S.: The organization of computations for uniform recurrence equations. Journal of the ACM **14** (1967) 563–590
2. Lamport, L.: The Parallel Execution of DO Loops. Communications of The ACM **17** (1974) 83–93

3. Feautrier, P.: Some efficient solutions to the affine scheduling problem, part I, one dimensional time. Int. J. of Parallel Programming **21** (1992) 313–348
4. Wolf, M., Lam, M.: Loop transformation theory and an algorithm to maximize parallelism. IEEE Transactions on Parallel and Distributed Systems **2** (1991) 452–471
5. Darte, A., Khachiyan, L., Robert, Y.: Linear scheduling is nearly optimal. Parallel Processing Letters **1** (1991) 73–81
6. Kung, H.: Why systolic architectures? Computer **15** (1982) 37–46
7. Lee, P., Kedem, Z.: Mapping nested loop algorithms into multidimensional systolic arrays. IEEE Transaction On Parallel and Distributed System **1** (90) 64–76
8. Moldovan, D.: On the analysis and synthesis of VLSI systolic arrays. IEEE Transactions on Computers **31** (1982) 1121–1126
9. Mauras, C., Quinton, P., Rajopadhye, S., Saouter, Y.: Scheduling affine parameterized recurrences by means of variable dependent timing functions. In Kung, S., E.E. Swartzlander, J., Fortes, J., Przytula, K., eds.: Application Specific Array Processors, IEEE Computer Society Press (1990) 100–110
10. Ashcroft, E., Wadge, W.: Lucid, a formal system for writing and proving programs. SIAM j. Comp. **3** (1976) 336–354
11. Caspi, P., Halbwachs, N., Pilaud, D., Plaice, J.: Lustre: a declarative language for programming synchronous systems. In: 14th Symposium on Principles of Programming Languages, ACM, Munich (1987)
12. Le Guernic, P., Benveniste, A., Bournai, P., Gautier, T.: SIGNAL: A data flow oriented language for signal processing. In: IEEE Workshop on VLSI 1984. (1984)
13. Chen, M., Choo, Y., Li, J. In: Crystal: Theory and Pragmatics of Generating Efficient Parallel Code. ACM Press (1991) Chapter 7
14. Perrin, G., Genaud, S., Violard, E.: PEI: a theoretical framework for data-parallel programming. Technical report, ICPS, Strasbourg (1994)
15. Mauras, C.: Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones. Thèse de doctorat, Ifsic, Université de Rennes 1 (1989)
16. Dupont De Dinechin, F., Robert, S., Risset, T.: Structured scheduling of recurrence equations. Technical Report 1140, Irisa, Rennes, France (1997)
17. Wilde, D.: The Alpha language. Technical Report 827, Irisa, Rennes, France (1994)
18. Saouter, Y.: A propos de systèmes d'équations récurrentes. Thèse de doctorat, Ifsic, Université de Rennes 1 (1992)
19. Darte, A.: Techniques de parallélisation automatique de nids de boucles. Thèse de doctorat, LIP ENS-Lyon (1993)
20. Feautrier, P.: Some efficient solution to the affine scheduling problem, part II, multidimensional time. Int. J. of Parallel Programming **21** (1992)
21. Chaudhary, V., Xu, C.Z., Roy, S., Ju, J., Sinha, V., , Luo, L.: Design and evaluation of an environment ape for automatic parallelization of programs. In: Int. Symp. on Parallel Architectures, Algorithms, and Networks. (1996) 77–83
22. Irigoin, F., Jouvelot, P., R.Triolet: Overview of the PIPS project. In: Procs of the Int. Workshop on Compiler for Parallel Computers, Paris. (1990) 199–212
23. Raji-Werth, M., Feautrier, P.: On parallel program generation for massively parallel architectures. In Durand, M., Dabaghi, F.E., eds.: High Performance Computing II, North-Holland (1991)
24. Griebl, M., Lengauer, C.: The loop parallelizer LooPo. In Gerndt, M., ed.: Proc. Sixth Workshop on Compilers for Parallel Computers. Volume 21 of Konferenzen des Forschungszentrums Jülich. Forschungszentrum Jülich (1996) 311–320

25. Burleson, W.: Using regular array methods for DSP module synthesis. In: 27th Hawaii Int. Conf. System Science Vol 1: Architecture. (1994) 58–67
26. Catthoor, F., Danckaert, K., Kulkarni, C., Omnes, T.: Data transfer and storage architecture issues and exploration in multimedia processors. In: Programmable Digital Signal Processors: Architecture, Programming, and Applications. Marcel Dekker, Inc, New York (2000)
27. Kienhuis, B., Rijpkema, E., Deprettere, E.: Compaan: Deriving process networks from matlab for embedded signal processing architectures. In: 8th International Workshop on Hardware/Software Codesign (CODES'2000). (2000)
28. Dupont de Dinechin, F., Quinton, P., Rajopadhye, S., Risset, T.: First Steps in Alpha. Technical Report 1244, Irisa (1999)
29. De Dinechin, F.D., Quinton, P., Risset, T.: Structuration of the Alpha language. In Giloi, W., Jahnichen, S., Shriver, B., eds.: Massively Parallel Programming Models, IEEE Computer Society Press (1995) 18–24
30. Dupont De Dinechin, F.: Libraries of schedule-free operators in Alpha. In: Application Specific Array Processor. (1997)
31. Crop, J., Wilde, D.: Scheduling Structured Systems. In: Fifth International Europar Conference. LNCS, Toulouse, France, Springer Verlag (1999) 409–412
32. Darte, A., Robert, Y.: Scheduling uniform loop nests. Technical Report 92-10, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, France (1992)
33. Feautrier, P.: Parametric integer programming. RAIRO Recherche Opérationnelle **22** (1988) 243–268
34. Schrijver, A.: Theory of Linear and Integer Programming. John Wiley and Sons, New York (1986)
35. Darte, A., Vivien, F.: Revisiting the decomposition of Karp, Miller and Winograd. In: Application Specific Array Processor. (1997) 13–25

A Proof of Theorem 1

In the following, we will call *degenerated* a domain which contains equalities in its definition (e.g. $\mathcal{D} = \{i, j \mid i = j; i \geq 0\}$ is degenerated). We to prove two lemmas. The first one deals with the case when there are no degenerated domains concerned with the call, the other one is a generalization of the first one to any situations.

Lemma 1. *Consider the call (15) to* `sub` *in* `sys` *where domains of input/output variables and the extension domain are not degenerated:*

$$\texttt{use } \{J|AJ \leq b\} \texttt{ sub[F(J,N)] } (U_1, \ldots, U_n) \texttt{ return } (V_1, \ldots, V_m) \quad . \tag{15}$$

Let u_i and v_j be the formal arguments of system `sub` *corresponding to U_i and V_j. If the schedule T^{sub} of* `sub` *is given, the constraints (16) added to the usual dependency constraints between variables of* `sys` *are sufficient to obtain a valid structured scheduling T^{sys} of* `sys`*. Moreover these constraints are necessary, in the sense that they are satisfied by every structured scheduling.*

$$\begin{array}{ll} T^{sys}_{U_i}(I,0) = T^{sub}_{u_i}(I) & \forall I \in Dom(u_i) \\ T^{sys}_{V_j}(I,0) = T^{sub}_{v_i}(I) & \forall I \in Dom(v_i) \\ T^{sys}_{U_i}(0,J) - T^{sys}_{V_j}(0,J) = T^{sub}_{u_i}(0) - T^{sub}_{v_i}(0) & \forall J \in \{J|AJ \leq b\} \end{array} \tag{16}$$

Proof: we look for structured scheduling (definition 2), hence $T^{\texttt{sys}}$ must meet the following conditions:

for all actual input $U_i[I, J]$:
$\forall (I, J) \in Dom_{U_i}$ $\quad T^{\texttt{sys}}_{U_i}(I, J) = c(J) + T^{\texttt{sub}}_{u_i}(I)$
for all actual output $V_i[I, J]$:
$\forall (I, J) \in Dom_{V_i}$ $\quad T^{\texttt{sys}}_{V_i}(I, J) = c(J) + T^{\texttt{sub}}_{v_i}(I)$

These equalities hold for all $(I, J) \in \mathrm{Dom}(U_i)$ (resp. $(I, J) \in \mathrm{Dom}(V_i)$). Unless these domains are degenerated, these equalities can be extended to the whole space (any value of (I, J)). Hence, for non-degenerated domains, we can deduce (setting I then J to 0):

- $T^{\texttt{sys}}_{U_i}(0, J) - T^{\texttt{sys}}_{V_j}(0, J) = T^{\texttt{sub}}_{u_i}(0) - T^{\texttt{sub}}_{v_j}(0)$ and,
- $T^{\texttt{sys}}_{U_i}(I, 0) = T^{\texttt{sub}}_{u_i}(I), \quad T^{\texttt{sys}}_{V_j}(I, 0) = T^{\texttt{sub}}_{v_i}(I)$.

As a consequence, conditions (16) are necessary when the domains are not degenerated.

We must now show that, for each dependence d_1 between input and output of `sub`, constraints (16), impose that d_1 is satisfied by $T^{\texttt{sys}}$. Here, we will note $\mathcal{D}(N)$ to express the fact that domain $\mathcal{D}$ is parameterized by N.

Consider the following dependence constraints occuring from a dependence path d_1 in subsystem `sub`:

$$\forall I \in \mathcal{D}_{d1}(M), T^{\texttt{sub}}_{v_i}(I) - T^{\texttt{sub}}_{u_j}(d_1(I, M)) \geq d(I, M) \quad , \tag{17}$$

where M represents the parameters of sub, d_1 is the value of the dependence and d is the *duration* (affine function of I and M). As $T^{\texttt{sub}}$ is a valid schedule of sub, this constraint is met by $T^{\texttt{sub}}$. From the definition of the structuring mechanism (see [29]) in Alpha, the corresponding constraint that must be respected by the schedule $T^{\texttt{sys}}$ of sys, which contains the call (15), is:

$$\begin{array}{l}\forall J \in \{J|AJ \leq b\}, \forall I \in \mathcal{D}_{d1}(F(J,N)), \\ T^{\texttt{sys}}_{V_i}(I,J) - T^{\texttt{sys}}_{U_j}(d_1(I,F(J,N),J) \geq d(I,F(J,N)) \quad ,\end{array}$$

where N represents the parameters of sys and $F(J,N)$ is the parameter assignment function in (15). As we have $D_{d_1}(F(J,N)) \subset D_{v_i}(F(J,N))$, from the constraints (16) we can deduce:

$$\begin{array}{l}\forall J \in \{J|AJ \leq b\}, \forall I \in \mathcal{D}_{d1}(F(J,N)) \\ T^{\texttt{sys}}_{V_i}(I,J) - T^{\texttt{sys}}_{U_j}(d_1(I,F(J,N),J) = T^{\texttt{sub}}_{v_i}(I) - T^{\texttt{sub}}_{u_j}(d_1(I,F(J,N)))\end{array}$$

By the construction of the use, if J is in the extension domain: $\{J|AJ \leq b\}$ and if N is in the parameter domain of sys, $F(J,N)$ is in the parameter domain of sub. Hence we can use the inequality (17) to bound the right hand side of the above equality, and we get the result:

$$\forall J \in \{J|AJ \leq b\}, \forall I \in \mathcal{D}_{d1}(F(J,N)), T^{\texttt{sys}}_{V_i}(I,J) - T^{\texttt{sys}}_{U_j}(d_1(I,F(J,N),J) \geq d(I,F(J,N)) \quad .$$

Hence the constraints generated by dependence d_1 is respected by a scheduling of sys provided it respect respects constraints (16). □

Lemma 2. *Consider a call to* sub *in* sys *(see (15)) in which degenerated domains occur. Then the constraints (16) are still sufficient, but may not be satisfied by a valid structured scheduling* T^{sys} *of* sys*. However, if this is the case, it is always possible to find another expression of the same schedule in which constraints (16) are met.*

Proof: the second part of the previous proof can be repeated here, and we obtain that the dependency constraints are respected if constraints (16) are added to the schedule. The problem with degenerated domains come from the fact that we may not be able to extend the relation of definition (2) to $J = 0$.

Say we have this relation happening on the domain of U_i:

$$T^{\texttt{sys}}_{U_i}(I,J) = c(J) + T^{\texttt{sub}}_{u_i}(I) \quad \forall (I,J) \in Dom_{U_i} \quad .$$

This relation can be extended to the affine space containing Dom_{U_i} that we note $\mathrm{Aff}(Dom_{U_i})$. By hypothesis this space contains equalities but one important property (due to the structuring mechanism) is that none of these equalities involve together extension indices (J) and local indices (I). Hence, this set of equalities can be splitted in two sets: n_J equalities occurring on extension indices and n_I equalities occurring on local indices. The real dimension of $\mathrm{Aff}(Dom_{U_i})$ is therefore $n - n_J - n_I$ (where n is the dimension of the global space).

Let us note m_J the number of extension indices (length of J) and $\{eq_1, \ldots, eq_{n_J}\}$ the set of equalities occurring on J. Without loss of generality, we can suppose that this set of equalities has been simplified and the J components permuted such that equality eq_i defines J_i in term of $\{J_{n_J+1}, \ldots, J_{m_J}\}$. Suppose, for instance, that we cannot extend the relation along the first component of J. it means that we have: $T^{\mathtt{sys}}_{U_i}(I, j_1, \ldots, j_m) = c(j_1, \ldots, j_m) + T^{\mathtt{sub}}_{u_i}(I)$ and $T^{\mathtt{sys}}_{U_i}(I, 0, j_2, \ldots, j_{m_J}) \neq c(0, j_2, \ldots, j_{m_J}) + T^{\mathtt{sub}}_{u_i}(I)$. This means that the first equality holds *only* in the context of the relation between j_1 and $\{J_{n_J+1}, \ldots, J_{m_J}\}$. In that case, it sufficies to replace j_1 in c by its definition in term of $\{J_{n_J+1}, \ldots, J_{m_J}\}$ as defined by eq_1, we will obtain a new expression of the timing function of U_i which take the same value as $T^{\mathtt{sys}}_{U_i}$ on $\mathrm{Aff}(Dom_{U_i})$ and which respects the equality wanted along one the j_1 dimension (obviously because j_1 is no more involved in c) . Just by counting the number of dimension added, one can check that we will use at most n_j equality on J to obtain a timing function $\mathcal{T}^{\mathtt{sys}}_{U_i}$ which is equal to $T^{\mathtt{sys}}_{U_i}$ on $\mathrm{Aff}(Dom_{U_i})$ and respects: $\mathcal{T}^{\mathtt{sys}}_{U_i}(I, 0) = T^{\mathtt{sub}}_{u_i}(I)\ \forall I \in Dom(u_i)$

The same transformation can be done with the n_I equalities on I to obtain the other equality: $\mathcal{T}^{\mathtt{sys}}_{U_i}(0, J) = c(J) + T^{\mathtt{sub}}_{u_i}(0)\ \forall J \in \{J | AJ \leq b\}$ And we have the result. □

B Proof of Theorem 2

$$U_i[I, J] \leftarrow V_j[f(I), g(J)] \quad . \tag{18}$$

We have two dependence path, $p_1 : U_i[I, J] \leftarrow V_j[f(I), g(J)]$, $\forall (I, J) \in Dom_{p_1}$ and also $p_2 : V_j[I, J] \leftarrow U_i[h(I), J]$, $\forall I \in Dom_{p_2}, \forall J \in \mathcal{E}$. Suppose that the condition is not respected, i.e. $T^{\mathtt{sub}}_{V_j}(I) - T^{\mathtt{sub}}_{U_i}(I') >= k(I, J, N)$ (N is the parameter of `sys`, J are the extension indices, k is *not* a constant function). we have the constraints:

$$\begin{array}{l} T^{sys}_{U_i}[I, J] \geq T^{sys}_{V_j}[f(I), g(J)] \geq k(f(I), g(J), N) + T^{sys}_{U_i}[h(f(I)), g(J)] \geq \\ k(f(I), g(J), N) + k(f(h(f(I))), g^2(J), N) + T^{sys}_{U_i}[h(f(h(f(I)))), g^2(J)] \geq \\ \underbrace{\ldots}_{\text{P cycles unrolled}} \qquad \geq \sum_{l=1}^{P} k((f \circ h)^l \circ f(I), g^l(J), N) + T^{sys}_{V_j}[(f \circ h)^{(P)}(I), g^{P+1}(J)] \end{array}$$

None of the term of the sum above is constant, hence, unless the cycle is not a real cycle (i.e. it can only be unrolled a constant number of time), the above constraint is not linear (but quadratic). Thus, there is no hope to find a linear schedule (for sake of clarity we have assumed that the parameter assignement function $\mathcal{F}$ was simply $(J, N \rightarrow N)$, the computation above can be arried out with any parameter assignement function). □