Some efficient solutions to the affine scheduling problem
Part I
One-dimensional Time

Paul Feautrier

Laboratoire MASI, Institut Blaise Pascal
Université de Versailles St-Quentin
45 Avenue des Etats-Unis
78035 VERSAILLES CEDEX FRANCE

e-mail : `feautrier@masi.ibp.fr`

April 23, 1993

Abstract

Programs and systems of recurrence equations may be represented as sets of actions which are to be executed subject to precedence constraints. In many cases, actions may be labelled by integral vectors in some iteration domain, and precedence constraints may be described by affine relations. A schedule for such a program is a function which assigns an execution date to each action. Knowledge of such a schedule allows one to estimate the intrinsic degree of parallelism of the program and to compile a parallel version for multiprocessor architectures or systolic arrays.

This paper deals with the problem of finding closed form schedules as affine or piecewise affine functions of the iteration vector. An efficient algorithm is presented which reduces the scheduling problem to a parametric linear program of small size, which can be readily solved by an efficient algorithm.

Résumé

De nombreux programmes ou systèmes d'équations de récurrence peuvent se représenter comme un ensemble d'actions qui doivent être exécutées en respectant des contraintes de précédence. En général, une action peut être repérée par un vecteur d'itération à coordonnées entières, et les contraintes de précédence se représentent par des relations affines. Un ordonnancement pour un tel programme est une fonction qui donne la date d'exécution de chaque action. La connaissance d'un ordonnancement permet d'estimer le parallélisme intrinsèque du programme et d'en écrire très facilement une version parallèle bien adaptée aux multiprocesseurs ou aux réseaux systoliques. Le présent travail traite de la recherche d'un ordonnancement explicitement représenté par des fonctions affines ou affines par morceaux. On présente une méthode très simple qui ramène la question à la résolution d'un programme linéaire paramétrique de petite taille, problème qui peut être efficacement résolu.

Keywords Automatic parallelization, automatic systolic array design, scheduling.

1 Introduction

The basic problem of parallel programming may be stated in the following terms:

- We are given a set Ω of *operations*, i.e. a set of transformations which are to be applied to a given initial store.
- The order in which the operations of Ω are to be applied is not totally arbitrary. We are given a binary relation Γ on Ω, the *dependence graph*,

with the understanding that if:

$$u, v \in \Omega,\ u\Gamma v,$$

then application of v cannot begin before the end of the application of u. Conversely, if the pair $\langle u, v\rangle$ does not belong to the transitive closure of Γ, then u and v may be applied in any order and may even be applied simultaneously if there are enough resources to support their parallel execution.

- Constructing a parallel program is the process of selecting a partial order which is as close as possible to the transitive closure of Γ while being susceptible of a simple description.

Note that, in general, we are not interested in a single dependence graph $\langle\Omega, \Gamma\rangle$ but in a possibly infinite family of similar graphs, each member of the family being specified by one or more integer parameters, which will be collectively denoted by n, and called the *structure parameters* of the graph. Consider for instance the family of programs which compute the product of two $n \times n$ matrices. We will suppose implicitly that the complexity of the programs in the family increases with n, and we will be mostly interested with what happens when n gets large – after all, there is no need of a super-computer and sophisticated programming techniques for multiplying small matrices.

A graph $\langle\Omega, \Gamma\rangle$ does not necessarily describe a computing process. If there is a vertex of Γ with an infinite incoming path, the corresponding operation will start only after an infinite number of operations have terminated, i.e., never. This occurs mainly when there is a cycle in Γ. A dependence graph is said to be *consistent* if it has no infinite incoming path. As is well known, consistency is a necessary and sufficient condition for the dependence graph to define a parallel program. All dependence graphs which are derived from sequential programs in various ways are consistent.

Many researchers in the field of parallel programming have realized that a simple way of describing a parallel program uses a *schedule*, i.e. a function θ from Ω to R which gives the epoch at which each operation must be executed.

The idea of using schedules comes from Operation Research and PERT charts. In OR, one tries to tabulate the function θ, it being understood that the number of operations is small. The idea has been applied to the construction of parallel programs under the name *macro-tasking*. However, the method has several disadvantages:

- Constructing the operations (or *tasks*) by chunking elementary operations is no trivial matter. In fact, to the author's knowledge, the only algorithm for the automatic construction of tasks has been given by Ayguadé et. al.[1].
- One cannot proceed unless one knows the execution time of each task and the exact number of processors. This is difficult: for instance, the execution time of a task may depend on the iteration count of a loop, which may vary from execution to execution.
- Lastly, when all elements of the problem are given, the general scheduling problem is NP-complete.

To solve these difficulties, several authors have proposed to work with the smallest possible tasks – for instance, the execution of only one statement. Since there will be a very large number of operations one can use asymptotic evaluation of the resulting program. In that case, exact knowledge of the operation execution times is no longer critical, provided they all are of the same order. One may even suppose that all execution times are equal to one time unit [2].

In contrast, it will no longer be feasible to tabulate the schedules; one must look for *closed forms*. This is possible only if one is given a synthetic description of the dependence graph. For instance, when the dependence graph derives from a sequential program, the operations are iterations of statements. It is possible to name them by using the name of the statement and the iteration vector coordinates. One may then seek schedules with simple expressions in term of the iteration vector.

Affine scheduling is the special case in which both the set of operations Ω and the dependence graph Γ are described by sets of linear inequalities. We will show that the equivalent parallel program can be described by an affine or piecewise affine schedule.

The problem of finding affine schedules has been mainly studied with the aim of designing systolic arrays [3, 4] from systems of recurrence equations. The use of affine schedules for the construction of parallel programs may be related to the wavefront method [5], and has be advocated in several recent publications [2, 6, 7, 8, 9, 10]. The starting point is the causality condition:

$$u, v \in \Omega,\ u\Gamma v \Rightarrow \theta(v) \geq \theta(u) + 1. \tag{1}$$

To find a closed form schedule, one assumes θ to be of the selected form with unknown coefficients. One then writes (1) for all edges of Γ and solves.

> For instance, in many cases u and v are vectors. One looks for a schedule of the form:
>
> $$\theta(u) = \tau.u + \alpha,$$
>
> where τ is an unknown vector and α an unknown number. All instances of (1) are linear in τ and α, and one has to solve a linear programming problem.

In general, this method is not practical, because one gets a prohibitively large system of linear inequalities. In fact, the size of the set Ω is equal to the total operation count of the program (e.g., $O(n^3)$ for a matrix inversion), and the size of Γ may be even larger. Another point is that, if one is interested in program *families*, then the system to be solved may well become infinite. The main theme of the present paper is how to take into account the special form of usual dependence graphs in order to compress (1) into a set of inequalities whose size does no longer depend on the program complexity.

The paper is organized as follow. Section 2 is a review of previous work on the subject and general results. Section 3 will be devoted to the computation of affine schedules. I will show how to summarize an indefinite number of instances of (1) by a few linear constraints with the help of a classical result of the theory of linear inequalities, the affine form of Farkas Lemma, and how to solve these constraints for an efficient schedule, efficiency being defined in several ways.

We will show in the conclusion that there are programs for which no affine schedule exists. The solution in that case is the construction of multidimensional schedules [11].

A few words on notations are in order here. The main objects in this work will be vectors and set of vectors with positive coordinates. Such vectors will be denoted by italic letters. All classical scalar operators will be tacitly extended component-wise to such vectors.

$$a \leq b$$

e.g., will be interpreted as:

$$\forall i,\, a_i \leq b_i,$$

and, as such, will be a partial order on the set of vectors. Similar conventions will apply to other binary connectives. We will sometime use $a[i]$ instead of a_i, and $a[i..j]$ for the vector built from the components i to j of a. This notation will be undefined if $j < i$. *Lexicographic ordering*:

$$\exists i\,(a[1..i] = b[1..i] \wedge a[i+1] < b[i+1])$$

will be written $a \ll b$. Lexicographic ordering is a strict total order.

The basic reference for the theory of linear inequalities is Schrijver's textbook [12].

2 A review of definitions and general results

2.1 Generalized Dependence Graph

Dependence graphs are used in every form of parallel programming as a mean of specifying timing relationships between operations. There are nearly as many dependence graphs as there are workers in the field. However, most definitions may be subsumed under the following generalization.

A Generalized Dependence Graph (GDG) is a directed multigraph. A vertex represents a set of related operations (e.g. all iterations of the same instruction in the case of imperative programs, or all evaluations of instances of the same variable in the case of systems of recurrence equations). Each operation *belongs* to one (and only one) vertex of the GDG. An edge represents a timing constraint between two operations which belong respectively to its source and destination. The GDG may have loops (i.e. an edge whose source and destination are the same vertex) and cycles.

Each vertex is labeled with a description of its operations. Similarly, an edge is labeled with a description of the associated dependence relation. This paper will deals only with *linearly described* GDG's, in which all sets and relations are set of integral vectors belonging to convex polyhedra. Namely:

- To each vertex S is associated a polyhedron $\mathcal{D}_S$ in $\mathbb{Q}^{p_S}$, its *domain*. The integer p_S is the dimension of the iteration space of S. For instance, if the GDG has been extracted from a program, p_S is the number of loops enclosing S. Each operation of S may be denoted by a pair (S, x) where x is integral and belongs to $\mathcal{D}_S$.

- To each edge e from R to S may be associated a polyhedron $\mathcal{R}_e$ in $\mathsf{Q}^{p_R+p_S}$ such that a causality condition like (1) must be imposed between (R, x) and (S, y) iff $\langle x, y\rangle \in \mathcal{R}_e$.

In short, a GDG is a tuple $\langle V, E, \mathcal{D}, \mathcal{R}\rangle$ where V is the set of vertices, E is the set of edges, $\mathcal{D}$ is a function from V to the associated domains and $\mathcal{R}$ a function from E to the associated dependence relations. The source and destination of edge e will be written $\sigma(e)$ and $\delta(e)$ respectively.

> Observe that all $\mathcal{D}_S$ and $\mathcal{R}_e$ are included in the first octant of iteration space. This restriction is more a matter of technical convenience than a strict necessity. In case of need, it may be lifted by appropriate change of variables.

The Detailed Dependence Graph (DDG) associated to $\langle V, E, \mathcal{D}, \mathcal{R}\rangle$ is the graph whose vertices are:

$$\Omega = \bigcup_{S\in V} \{(S, x) \mid x \in \mathcal{D}_S\}$$

and whose edges are:

$$\Gamma = \bigcup_{e\in E} \{\langle(\sigma(e), x), (\delta(e), y)\rangle | x \in \mathcal{D}_{\sigma(e)}, y \in \mathcal{D}_{\delta(e)}, \langle x, y\rangle \in \mathcal{R}_e\}.$$

When working with GDG families, V and E will stay fixed, the only variables being the structure parameters n, which will occur linearly in the description of the polyhedra $\mathcal{D}_S$ and $\mathcal{R}_e$. One cannot construct a DDG from a given GDG unless the actual values of the structure parameters are known.

I will now show that this definition cover most cases of interest.

Dependence Graph of Sequential Programs In the case of ordinary DG, vertices are data processing statements and the associated domains are derived from an analysis of the surrounding loops[13]. As to the dependence relation, its form depends mainly on the thoroughness of the analysis which has been performed on the program. For instance, Allen and Kennedy[14] have shown that a fairly good parallel program can be deduced simply from a knowledge of the *depth* of each dependence, i.e. of an integer p_e such that $\mathcal{R}_e$ is[1]:

$$\langle x, y\rangle \in \mathcal{R}_e \equiv x[1..p_e] = y[1..p_e] \wedge x[p_e+1] < y[p_e+1].$$

Dependence direction vectors (DDV [15]) are a refinement on the idea of depth. A DDV is a word on the alphabet of comparison operators, $\{<, \leq, =, \geq, >\}$. Each letter of the word is associated with the corresponding components of x and y; it states that those components stand in the indicated relation to each other. If no particular relation is extant, one uses the neutral connective, $*$.

A depth p dependence corresponds to a DDV of the form:

$$\langle \overbrace{=, \cdots, =}^{p}, <, *, \cdots\rangle.$$

In some cases it may be possible to find a constant vector d such that the dependence exists only if:

$$y = x + d.$$

Such a d is called a dependence vector and the dependence is said to be *uniform*. The idea may be extended to cases in which the difference $y - x$ may be expressed as a positive linear combination of a finite set of dependence vectors. Such vectors span the *dependence cone* [16]. Both cases are easily included in the framework above.

As an extreme case, one may take as $\mathcal{R}_e$ the polyhedron whose description is:

$$\langle x, y\rangle \in \mathcal{R}_e \equiv f(x) = g(y) \wedge (x[1..p_e] = y[1..p_e]) \wedge (x[p_e+1] < y[p_e+1]),$$

where f and g are the indexing functions of the colliding references. One should then test $\mathcal{R}_e \cap \mathcal{D}_{\sigma(e)} \cap \mathcal{D}_{\delta(e)}$ for nonemptiness. This kind of GDG will contain much redundant information. The aim of Data Flow Analysis [13] is to reduce $\mathcal{R}_e$ to minimal form . One obtains a polyhedron $\mathcal{P}_e$ and an affine transformation h_e such that $\mathcal{R}_e$ is given by:

$$\langle x, y\rangle \in \mathcal{R}_e \equiv (x = h_e(y) \wedge y \in \mathcal{P}_e) \qquad (2)$$

[1]The present depth is one less than Allen and Kennedy's definition.

```
        do i = 0,n
1         s(i) = 0.
          do j = 0,n
2           s(i) = s(i) + a(i,j)*x(j)
          end do
        end do
```

Figure 1: A sample program

In such a case, one usually modifies the definition of $\mathcal{P}_e$ in such a way that:

$$y \in \mathcal{P}_e \Rightarrow y \in \mathcal{D}_{\delta(e)} \wedge h_e(y) \in \mathcal{D}_{\sigma(e)}.$$

A dependence is uniform in the special case where the associated h_e is a translation.

All such cases share a common property: the dependence relation is coarser than the sequential execution order of the source program. As a consequence, the GDG is automatically consistent.

Let us consider a simple kernel (Figure 1). There are two statements whose domains are:

$$\begin{aligned} \mathcal{D}_1 &= \{i \mid 0 \leq i \leq n\}, \\ \mathcal{D}_2 &= \{i, j \mid 0 \leq i, j \leq n\}. \end{aligned}$$

Standard dependence analysis indicates two dependences: one from statement 1 to 2, the other a loop on statement 2. Both dependences are at depth 1. Consider the first edge. The corresponding dependence polyhedron describes a relation between the (one-dimensional) iteration vector of statement 1, say i', and the iteration vector of of statement 2, say $\langle i, j \rangle$. The corresponding relation is:

$$\langle i', i, j \rangle \in \mathcal{R}_{1,2} \equiv i' = i.$$

With similar conventions, the dependence polyhedron for the loop is:

$$\langle i', j', i, j \rangle \in \mathcal{R}_{2,2} \equiv i' = i \wedge j' < j.$$

Dataflow Analysis gives more precise results. The first dependence exists only for the first iteration of the `j` loop ($j = 1$), and the second one exists only from iteration $j-1$ to iteration j provided $j \geq 2$:

$$\langle i', i, j\rangle \in \mathcal{R}_{1,2} \equiv i' = i \wedge j = 1.$$

$$\langle i', j', i, j\rangle \in \mathcal{R}_{2,2} \equiv i' = i \wedge j' = j-1 \wedge j \geq 2.$$

The loop on statement 2 describes a uniform dependence.

Systems of Recurrence Equations The recent interest in systems of recurrence equations (SRE) as a mean of specifying parallel processes results from the convergence of several trends of research. A pioneer paper by Karp et.al.[17] was motivated by the study of explicit difference schemes for the solution of systems of differential equations. Other authors [18, 19] were interested in the mathematical simplicity of SRE and their use for proving program correctness. The last motivation [20] was the fact that in SRE parallelism detection (if not parallelism exploitation) is obvious. Hence, it is felt that SRE are well adapted to the specification of massively parallel computing processes and in particular as a starting point for systolic array design. All these trends have converged in the definition of SRE languages [21, 22] and their associated programming environments.

The basic objects of an SRE are *variables*, which really are functions from a convex polyhedron to an arbitrary value space (e.g. the real numbers or the booleans). The domain of U will be $\mathcal{D}_U$. Each variable is defined by an equation of the form:

$$\begin{array}{lll} U(x) = & \texttt{case} & x \in \mathcal{D}_{U1} : \mathcal{E}_{U1}; \\ & & \ldots \\ & & x \in \mathcal{D}_{Un} : \mathcal{E}_{Un}; \\ & \texttt{esac} & \end{array}$$

The $\mathcal{D}_{Ui}$ are disjoint subdomains of $\mathcal{D}_U$; the $\mathcal{E}_{Ui}$ are expressions whose general form is:

$$f_U^i(\ldots, V(h_{UV}^i(x)), \ldots).$$

The f_U^i functions are supposed to be strict. The h_{UV}^i transformations are affine. The SRE is uniform (and is called a SURE), if all these transformations are translations. Transforming an arbitrary SRE to a SURE is called *uniformization* or *pipelining* and is an important step in the construction of a systolic array.

A SRE may be trivially translated into a Generalized Dependence Graph, whose vertices are the variables. The domain of each vertex is the domain of the corresponding variable. For each occurrence of a variable U in expression $\mathcal{E}_{Ui}$ in the right hand side of the equation for U, there is an edge e from V to U whose associated polyhedron is:

$$\langle x, y\rangle \in \mathcal{R}_e \equiv (x = h^i_{VU}(y) \wedge y \in \mathcal{D}_{Vi}).$$

One may note the similarity of this description with the Dataflow Graph (2). This should not be a surprise, since the one of the motivations of dataflow analysis was the automatic conversion of imperative programs to single assignment form [23].

The following SRE is in some sense equivalent to the program in Fig. 1:

$$\begin{array}{l} 1 \leq i \leq n : A_1(i) = 0.; \\ 1 \leq i \leq n : \texttt{case}\ j = 1 : A_2(i,j) = A_1(i) + a(i,j)x(j); \\ \qquad\qquad\qquad\quad j \geq 2 : A_2(i,j) = A_2(i,j-1) + a(i,j)x(j); \\ \qquad\qquad\quad \texttt{esac} \end{array}$$

They have the same DFG. One of the important properties of SRE's is that one may "replace variables by their values". For instance, one may substitute 0 to $A_1(i)$ in the first case of the equation for A_2.

2.2 Schedules

Definition 1 *A schedule for a given GDG $\langle V, E, \mathcal{D}, \mathcal{R}\rangle$ is a positive function $\theta : \Omega \mapsto \mathsf{N}$ such that for every edge e of E with source R and destination S,*

$$x \in \mathcal{D}_R, y \in \mathcal{D}_S, \langle x, y\rangle \in \mathcal{R}_e, \Rightarrow \theta(S, y) \geq \theta(R, x) + 1.$$

The number 1 in the above definition stands for the duration of statement R, which is taken as the unit of time. In a more realistic setting, one should apparently introduce a duration $\partial(R)$ for each statement, and rewrite definition 1 accordingly. Most often, this refinement is not really necessary.

For any operation u, $\theta(u)$ is the date at which u may be initiated on a computer with an unlimited number of processors. Such a schedule is a guideline for the construction of a program for a limited number of processors. Let $\mathcal{F}(t)$ be the set of operation which are initiated at time t (the *front* at t):

$$\mathcal{F}(t) = \{u \in \Omega \mid \theta(u) = t\}. \tag{3}$$

Build the following program:

```
do t = 0, L
      doall F(t)
      barrier
end do
```

In this program sketch, L is the latency of the schedule:

$$L = \max_{u \in \Omega} \theta(u), \tag{4}$$

and `barrier` is the classical synchronization operation, in which each processor stops until all processors have reached the barrier[2]. All operations of the front are distributed among the available processors. This program is not optimal in general. However, if one is interested only in large calculations, the interesting figure of merit is *asymptotic efficiency*, for which I have proved the following lower bound[2]:

$$\epsilon_P \geq \frac{1}{1 + \nu P \frac{L}{T}},$$

where:

- P is the number of processors,
- ν is the duration of the synchronization operation,
- T is the total workload,

$$T = \text{Card}\ \Omega.$$

[2]`barrier` is a null operation on a synchronous computer.

The ratio T/L is the mean degree of parallelism of the schedule. Its maximum value – which is obtained with a minimum latency schedule – is a characteristics of the object program.

> We will show later that the program in Fig. 1 has an affine schedule whose latency is n. Since the sequential operation count is $O(n^2)$, the mean degree of parallelism is $O(n)$, and the efficiency tends to 1 when n grows large.

This lower bound on the efficiency does not depend on the precise duration of operations, provided that they are all of the same order, and that the run-time system has reasonable load-balancing facilities. In the interest of simplicity, definition 1 will be used without modification. Most of the results and algorithms in this paper will still stand on more general assumptions. The form of (1) which corresponds to this hypothesis is:

$$u, v \in \Omega,\ u\Gamma v \Rightarrow \theta(v) \geq \theta(u) + 1. \tag{5}$$

2.3 Existence and decidability theorems

2.3.1 Basic Results

The first point is deciding whether a schedule exists, i.e. whether the GDG is consistent.

Proposition 2 *A necessary and sufficient condition for the existence of a schedule is that in $\langle \Omega, \Gamma \rangle$ there is no vertex u with an infinite incoming path.*

An order $\prec$ on the set of operations Ω is said to be *causal* iff

$$u\Gamma v \Rightarrow u \prec v.$$

If one knows a causal order, the best (or free) schedule is given by:

$$\theta(u) = \max\{\theta(v) \mid v\Gamma u\} + 1. \tag{6}$$

To be effective, this formula must be evaluated according to any linear extension of the given causal order. One then has the certainty that at any given time, all needed values of θ have been computed previously. This formula is the basis of *inductive methods* for computing schedules – compute enough values from (6), guess a closed form representation, and check that (1) is satisfied everywhere. Another use is in *run-time parallelization* [24], in which a schedule is computed according to (6) by a so-called inspector loop, then exploited for the actual parallel execution of the program.

2.3.2 Decidability and Undecidability results

The following theorem has been proved by Karp, Miller and Winograd [17]:

Theorem 3 *The problem of deciding whether a system of uniform recurrence equations is consistent is decidable.*

The situation is quite different for nonuniform dependence graphs.

Theorem 4 *The consistency problem for a nonuniform GDG with at least one infinite domain is undecidable.*

The proof[25] proceeds by coding the halting problem of a Turing machine into a consistency problem for a GDG.

Theorem 5 *The consistency problem for an infinite family of nonuniform GDG with finite domains is undecidable.*

The basic idea of the proof[26] is to code any instance of Hilbert tenth problem as the consistency problem of some GDG family.

2.3.3 Approximations to the Best Schedule

In the light of the above results, the problem of finding the best schedule looks quite intractable. In the general case, theorems 4 and 5 preclude the existence of a well defined algorithm. The situation is the opposite for uniform dependences, but this fact shows that there are GDG which cannot be converted to uniform GDG. It may be possible that uniform GDG have sufficient expressive power for the needs of numerical computing and signal processing, but this is at present an unproved conjecture. Existing uniformization algorithms [3] may not be applied to a GDG unless one knows a schedule; they offer no help for the solution of the present problem.

When working with GDG's derived from sequential programs, the question of consistency does not apply. One could then hope to find an exact algorithm whose convergence would be guaranteed in that case and that case only, thus sidestepping theorem 5. How to do that is unclear at present. Furthermore, one should expect to get results at least as complex as in the uniform case, and this may be too complex to be useful.

The solution out of this dilemma is to look for sub-optimal schedules, i.e. schedules which, while giving acceptable performance, are not guaranteed to be the best possible. One likely approach is to search not in the class of all functions from Ω to N, but in a smaller class. One may in that case hope to devise a total algorithm. Getting a negative answer will only mean that there is no schedule in the selected class, another way of sidestepping theorem 5. In the case of linearly described GDG, the class of choice is the class of affine functions – affine in terms of the iteration vectors and structure parameters. One may even dispense with integrality conditions, in view of the following theorem [27]:

Theorem 6 *If θ is a real valued solution to (1), then $\lfloor\theta\rfloor$ is an integral solution to the same scheduling problem. Obviously, $\lfloor\theta\rfloor$ is at least as good as the original θ.*

Proof Suppose that:

$$u \Gamma v.$$

It follows that:

$$\theta(v) \geq \theta(u) + 1.$$

Then:

$$\lfloor\theta(v)\rfloor \geq \lfloor\theta(u) + 1\rfloor \geq \lfloor\theta(u)\rfloor + 1.$$

$\square$

3 Computing Affine Schedules

Basically, the problem is the following. Let us be given a GDG $\langle V, E, \mathcal{D}, \mathcal{R}\rangle$. Postulate that each schedule is of the form:

$$\theta(S, x) = \tau_S.x + \sigma_S.n + \alpha_S. \tag{7}$$

τ_S and σ_S are fixed but unknown vectors with rational coordinates, α_S is an unknown number. Recall that n is the vector of structure parameters of the

GDG. The form above will be called a *prototype schedule*. The problem is to characterize the values of the unknowns such that the causality condition is satisfied, and then to select one particular solution in such a way as to optimize some quality criterion like maximal throughput or minimum latency.

> The above choice for the prototype schedule is not arbitrary. Darte and Robert [28] have shown that if one solves the scheduling problem for larger and larger values of n, the solution has a limit which is linear in n.

When the GDG has a linear description, the causality condition may be expressed in a more precise form. For each edge e of E, one must verify the condition:

$$x \in \mathcal{D}_{\sigma(e)}, y \in \mathcal{D}_{\delta(e)}, \langle x, y\rangle \in \mathcal{R}_e \Rightarrow \theta(\delta(e), y) \geq \theta(\sigma(e), x) + 1. \qquad (8)$$

This condition has to be verified for all values of n. This universal quantification will be kept implicit in what follows. For special forms of the GDG, this condition may be somewhat simplified. In the case of a DFG, if one takes into account the form (2) of the relation $\mathcal{R}_e$, one gets:

$$y \in \mathcal{P}_e \Rightarrow \theta(\delta(e), y) \geq \theta(\sigma(f), h_e(y)) + 1. \qquad (9)$$

One may use simpler prototype schedules. Most authors do not include a dependence on n in (7). In the case of uniform GDG's, the problem is further simplified by postulating that all schedules have the same spatial behaviour, i.e. that there is only one vector τ. The schedules one gets in that case are wavefronts[5].

When one substitutes appropriate values for x, y and n in (8) or (9), one gets systems of linear inequalities in the unknown coefficients. The situation is similar with the positivity constraints, $\theta(S, x) \geq 0$. These systems may be solved by any standard technique, e.g. the simplex algorithm. The problem is that there will be a very large number (perhaps an infinity of) values of x, y and n, and that one can never be sure that all important constraints have been included. The problem is how to take advantage of the special form of the GDG and of the prototype schedule to compress all these inequalities into a finite set. There are two basic methods for that; one of them may be called the vertex method [27] and will be briefly reviewed presently. The other one uses a fundamental result of the theory of linear inequalities, the affine form of Farkas lemma[12], and is the basic theme of this paper.

Before starting on the actual methods themselves, let us investigate a subtle point. Any affine form ϕ satisfies the following identity:

$$\lambda, \mu \geq 0, \lambda + \mu = 1 \Rightarrow \phi(\lambda x + \mu y) = \lambda\phi(x) + \mu\phi(y). \tag{10}$$

Hence, if an inequality $\phi(x) \geq 0$ is satisfied at all points of a set $\mathcal{D}$ it is also satisfied at all points of the convex hull of $\mathcal{D}$. (8) or (9) must be verified at all integral points inside a polyhedron $\mathcal{R}_e$. As a consequence, they are satisfied at all points of the convex hull of the set of integral points of $\mathcal{R}_e$ – the so-called *integer hull* of $\mathcal{R}_e$. But the integer hull of a polyhedron is not necessarily equal to the polyhedron itself – see the cover picture of Schrijver's book for a nice counterexample. However:

- The integer hull of a polyhedron is included in the polyhedron. Hence, enforcing (8) or (9) everywhere in the polyhedron is a pessimistic approach, which may eliminate some valid solutions but never give false results.

- While no systematic study has been made of the problem, it seems that the cases in which the discrepancy is significant are very rare.

- Lastly, the integer hull of a polyhedron is also a polyhedron, which may be constructed, in case of need, by adding *cutting planes* to the original description [12, chapter 23].

For all these reasons, I will ignore the difficulty and proceed by relaxing the integrality constraint on vectors like x and y in (8).

3.1 The vertex method

There are two ways of describing a polyhedron. One is as the set of points which satisfy a finite set of linear inequalities, and the other one is as the convex hull of a finite set of points. The two points of view are dual in some sense, and there are well defined algorithms for going from one representation to the other. Now it is obvious from (10) that a linear inequality is satisfied at all points of a convex hull if and only if it is satisfied at all extremal points.

Hence the method: find generating systems for the polyhedra $\mathcal{D}_s$ and $\mathcal{R}_e$, write (8) or (9) at each of these points, and solve for the unknown coefficients.

It has been shown that for each polyhedron there exist a finite set of points, called its *vertices*, which together generate the polyhedron, none of them being a convex combination of the other. Such a set of points is a minimal generating system. If the polyhedron is unbounded, some of the vertices will be at infinity – those are called *rays* – but since both cases can be nicely subsumed by using homogeneous coordinates, I will not bother with the distinction in the sequel.

Here is a summary of the vertex method:

- Compute a generating system for all polyhedra $\mathcal{D}_S, S \in V$ and $\mathcal{R}_e, e \in E$.
- Write instances of (8) at all vertices of $\mathcal{R}_e$. Similarly, write instances of $\theta(S, x) \geq 0$ at all vertices of $\mathcal{D}_S$.
- Solve the resulting finite system of linear inequalities by any standard algorithm.

Finding all the vertices of a polyhedron is a well known problem – Schrijver gives about twenty references. Chernikova's algorithm seems particularly well adapted[29]. Researchers in the field have a tendency to use the same algorithm for solving the resulting inequality system, but this appears to be a case of overkill.

3.2 The Farkas algorithm

While the transformation from a set of inequality to a set of vertices is an involution, it is by no means a polynomial process. For instance, a hypercube in p-space may be described by $2p$ inequalities, or by 2^p vertices! Hence, the size of the problem may be largely increased as one goes from the initial representation of the domains, which is in term of inequalities, to the vertex representation. The problem will still be compounded if one again uses the transformation algorithm to select the final solution, hence the interest in a method which uses directly the original inequations. This is based on the following well known result [12, Corollary 7.1h]:

Theorem 7 (Affine Form of Farkas Lemma) *Let $\mathcal{D}$ be a nonempty polyhedron defined by p affine inequalities*

$$a_k.x + b_k \geq 0,\ k = 1, p.$$

Then an affine form ψ is non negative everywhere in $\mathcal{D}$ iff it is a positive affine combination:

$$\psi(x) \equiv \lambda_0 + \sum_k \lambda_k (a_k.x + b_k),\ \lambda_k \geq 0.$$

Proof While the *if* part is obvious, the proof of the *only if* part relies on deep results of the theory of Linear Inequalities, like the Duality Theorem or the completeness of the Fourier-Motzkin elimination method. For a proof see Schrijver's book.

The positive constants λ_k whose existence is asserted by Farkas Lemma will be called *Farkas multipliers.* $\square$

3.2.1 Presentation of the method

Let us first introduce some notations. Let:

$$a_{Sk}.\begin{pmatrix} x \\ n \end{pmatrix} + b_{Sk} \geq 0,\ k = 1, m_S, \tag{11}$$

be the inequalities which define each domain $\mathcal{D}_S$. Similarly,

$$c_{ek}.\begin{pmatrix} x \\ y \\ n \end{pmatrix} + d_{ek} \geq 0,\ k = 1, m_e, \tag{12}$$

will define the dependence polyhedron $\mathcal{R}_e$. In case the dependence graph comes from a SRE or from Dataflow Analysis, let:

$$c_{ek}.\begin{pmatrix} y \\ n \end{pmatrix} + d_{ek} \geq 0,\ k = 1, m_e \tag{13}$$

be the description of $\mathcal{P}_e$.

Let us consider first the sign restriction on schedules. Affine forms like (7) will be nonnegative in the associated domain iff there exists Farkas multipliers μ_{Sk} such that:

$$\theta(S,x) \equiv \mu_{S0} + \sum_{k=1}^{m_S} \mu_{Sk}(a_{Sk}.\begin{pmatrix} x \\ n \end{pmatrix} + b_{Sk}). \tag{14}$$

These formulas are a new representation of the prototype schedules. They may be used in place of (7) for all subsequent calculations. When numerical values have been found for the unknowns μ_{Sk}, it will be a simple matter to reorder terms and to recover (7).

Now in case the schedules $\theta(S,x)$ are affine, the delay associated to edge e in the GDG:

$$\Delta_e = \theta(\delta(e),y) - \theta(\sigma(e),x) - 1$$

is a nonnegative affine form in $\mathcal{R}_e$. Hence, there exists Farkas multipliers λ_{ek} such that:

$$\theta(\delta(f),y) - \theta(\sigma(f),x) - 1 \equiv \lambda_{e0} + \sum_{k=1}^{m_e} \lambda_{ek}(c_{ek}.\begin{pmatrix} x \\ y \\ n \end{pmatrix} + d_{ek}). \tag{15}$$

Observe that the aim of classical dependence analysis is to prove that $\mathcal{R}_e$ is not empty, thus supplying the required minor premise for the application of Farkas Lemma. In the case of (14) the relevant hypothesis is the fact that instruction S is executed at least once for at least one value of the structure parameters. If this hypothesis is false, the offending instruction may as well be removed from the program.

In the case of a Dataflow Graph, one can see that the delay:

$$\Delta_e = \theta(\delta(e),y) - \theta(\sigma(e),h_e(y)) - 1$$

is affine, since h_e is an affine transformation. Hence the same reasoning applies. One can introduce Farkas multipliers λ_{ek} and get:

$$\theta(\delta(e),y) - \theta(\sigma(e),h_e(y)) - 1 \equiv \lambda_{e0} + \sum_{k=1}^{m_e} \lambda_{ek}(c_{ek}.\begin{pmatrix} y \\ n \end{pmatrix} + d_{ek}). \tag{16}$$

All these formulas are identities: one can gather coefficients of each independent variables (x, y and n) and equate the resulting sums to zero. From an algorithmic point of view, this may be seen either as a standard application of elementary computer algebra techniques or as a matrix transposition.

The result of the identification process is a system of linear equations whose unknowns are Farkas multipliers, which are constrained to be positive. As a consequence, the solution must use an algorithm like the simplex for solving linear inequalities. To get a feeling of the work to be done, let us evaluate the size of the problems to be solved. This will be done only for the DFG case, which is simpler.

The first step is counting unknowns. Domain $\mathcal{D}_S$ is defined by m_S inequalities. As a consequence, schedule $\theta(S, x)$ will use $m_S + 1$ Farkas multipliers. Similarly, to each edge e will correspond a dependence domain $\mathcal{P}_e$ with m_e inequalities, i.e., $m_e + 1$ Farkas multipliers. The total number of unknowns therefore is:

$$X = \sum_{S \in V} (m_S + 1) + \sum_{e \in E} (m_e + 1).$$

On the other hand, for each edge e, identity (16) will give one equation for each independent variable, i.e. $p_{\delta(e)} + |n| + 1$ equations, where $|n|$ is the number of structure parameters.Hence the total number of equations will be:

$$Q = \sum_{e \in E} p_{\delta(e)} + |n| + 1.$$

In nearly all practical cases, each iteration domain is defined by two inequalities per loop:

$$m_S \approx 2p_S.$$

Similarly, it is an empirical fact that the dependence domain $\mathcal{P}_e$ is a subset of $\mathcal{D}_{\delta(e)}$ which is defined by adding one supplementary constraint to those defining $\mathcal{D}_{\delta(e)}$:

$$m_e \approx 2p_{\delta(e)} + 1.$$

From these observations one deduces the approximate relation:

$$X = 2\sum_S p_S + 2Q + \text{Card } V - (|n| - 1)\text{Card } E.$$

Since $|n|$ is always a small integer and since Card V and Card E are of the same order of magnitude – another empirical fact[13] – one concludes that there are about twice as much unknowns than equations. The first consequence is that, if there are solutions at all, they will form a very large family among which one will have to choose the "best" schedule, in a sense yet to be defined.

The second point is that the size of the problem may be reduced by using a kind of Gauss-Jordan elimination:

- Select an equation and an unknown which occurs in it with a nonzero coefficient and solve it, to obtain an equality of the form $x = e$.
- Set up the constraint $e \geq 0$.
- Eliminate x in favor of e both in previously obtained inequalities and in all unprocessed equations.
- Iterate until there are no equations left.

All equational constraints will be replaced by the same number of inequalities, while the number of unknowns will be reduced by Q. In order to facilitate the subsequent reconstruction of the schedules, one should preferably eliminate the λ_{ek} Farkas multipliers.

3.2.2 An example

Let us go back to example 1. As we have said earlier, Dataflow Analysis shows that there are two edges in the DFG. The first one represents the initialization of `s` by instruction 1, for the use of the first j-iteration of 2. The second edge is associated to the transmission of the successive values of `s` from one j-iteration of the second instruction to the next one. The first edge has the following characteristics:

$$\begin{aligned} \mathcal{P}_1 &= \mathcal{D}_2 \cap \{i, j | j \leq 0\}, \\ h_1(i, j) &= i. \end{aligned}$$

The elements of the second edge are:

$$\begin{aligned} \mathcal{P}_2 &= \mathcal{D}_2 \cap \{i, j | j \geq 1\} \\ h_2(i, j) &= \langle i, j - 1 \rangle \end{aligned}$$

The prototype schedules may be written as:

$$\begin{aligned} \theta(1, i) &= \mu_{1,0} + \mu_{1,1} i + \mu_{1,2}(n - i), \\ \theta(2, i, j) &= \mu_{2,0} + \mu_{2,1} i + \mu_{2,2}(n - i) + \mu_{2,3} j + \mu_{2,4}(n - j). \end{aligned}$$

For the first edge, the equivalent of (16) is:

$$\begin{aligned} & \mu_{2,0} + \mu_{2,1} i + \mu_{2,2}(n-i) + \mu_{2,3} j + \mu_{2,4}(n-j) \\ - & (\mu_{1,0} + \mu_{1,1} i + \mu_{1,2}(n-i)) - 1 \\ \equiv & \lambda_{1,0} + \lambda_{1,1} i + \lambda_{1,2}(n-i) + \lambda_{1,3} j + \lambda_{1,4}(n-j) - \lambda_{1,5} j, \end{aligned}$$

which gives, by a process of identification, the following equations:

$$\begin{aligned} \mu_{2,0} - \mu_{1,0} - 1 &= \lambda_{1,0}, \\ \mu_{2,1} - \mu_{2,2} - \mu_{1,1} + \mu_{1,2} &= \lambda_{1,1} - \lambda_{1,2}, \\ \mu_{2,3} - \mu_{2,4} &= \lambda_{1,3} - \lambda_{1,4} - \lambda_{1,5}, \\ \mu_{2,2} + \mu_{2,4} - \mu_{1,2} &= \lambda_{1,2} + \lambda_{1,4}. \end{aligned}$$

The second edge is a uniform loop, with the result that the delay does not depend on the iteration vector. There is no need to use Farkas Lemma; one obtains directly:

$$\mu_{2,3} - \mu_{2,4} - 1 \geq 0. \tag{17}$$

The next step is to eliminate as much unknowns as possible. One possible result is:

$$\begin{aligned} \lambda_{1,0} &= \mu_{2,0} - \mu_{1,0} - 1 \geq 0, && (18) \\ \lambda_{1,1} &= \mu_{2,1} + \mu_{2,4} - \mu_{1,1} - \lambda_{1,4} \geq 0, && (19) \\ \lambda_{1,3} &= \mu_{2,3} - \mu_{2,4} + \lambda_{1,4} + \lambda_{1,5} \geq 0, && (20) \\ \lambda_{1,2} &= \mu_{2,2} + \mu_{2,4} - \mu_{1,2} - \lambda_{1,4} \geq 0, && (21) \\ & \mu_{2,3} - \mu_{2,4} - 1 \geq 0. && (22) \end{aligned}$$

These inequalities should be systematically solved by the simplex algorithm. It is interesting, however, to get a feeling of the form of the result by attempting to solve them by straightforward reasoning. Now $\lambda_{1,5}$ has only one positive occurrence in (20). Hence this constraint may always be satisfied by giving $\lambda_{1,5}$ a sufficiently large value, and is therefore void. On the other hand, (19) and (21) will never be satisfied whatever the value of $\lambda_{1,4}$ unless:

$$\begin{aligned} \mu_{2,1} + \mu_{2,4} - \mu_{1,1} &\geq 0, \\ \mu_{2,2} + \mu_{2,4} - \mu_{1,2} &\geq 0. \end{aligned}$$

These two constraints, together with (18) and (22), completely describe the solution space. A more explicit description is as follows: arbitrary select positive values for $\mu_{1,0}$, $\mu_{2,1}$, $\mu_{2,2}$ and $\mu_{2,4}$. Other parameters should satisfy:

$$\begin{array}{lcl} 0 \leq & \mu_{1,1} & \leq \mu_{2,1} + \mu_{2,4}, \\ 0 \leq & \mu_{1,2} & \leq \mu_{2,2} + \mu_{2,4}, \\ \mu_{2,0} & \geq & 1 + \mu_{1,0}, \\ \mu_{2,3} & \geq & 1 + \mu_{2,4}. \end{array}$$

The simplest solution is obtained for $\mu_{1,0} = \mu_{2,1} = \mu_{2,2} = \mu_{2,4} = 0$, from which follows $\mu_{1,1} = \mu_{1,2} = 0$ and $\mu_{2,0} = \mu_{2,3} = 1$. The corresponding schedules are:

$$\begin{aligned} \theta(1,i) &= 0, \\ \theta(2,i,j) &= j+1 \end{aligned} \tag{23}$$

Another possible solution is $\mu_{2,1} = \mu_{1,1} = 1$, which corresponds to:

$$\begin{aligned} \theta'(1,i) &= i, \\ \theta'(2,i,j) &= i+j+1 \end{aligned} \tag{24}$$

There are many other possibilities.

One may construct the fronts for θ according to (3) and build the corresponding parallel program. The resulting program is shown in figure 2. Statement 1 is the front at time 0. The `j` loop is the time loop. Statement 2 is the front at time $j+1$. In term of transformations, the new program is the result of a loop splitting, a loop inversion and the detection of two parallel loops.

3.3 Selecting a Good Schedule

It is obviously not possible to enumerate all causal schedules: one should devise some criterion and choose a best one by as simple a process as possible. One of the possibilities is to consider that the set of constraints defines a polyhedron in μ-space, and to construct its generating system. Whatever the selection principle, it seems plausible that vertices will dominate all other

```
        doall (i = 0:n)
1         s(i) = 0.
        end do
        do j = 0,n
          doall (i = 0:n)
2           s = s + a(i,j)*x(j)
          end do
        end do
```

Figure 2: The parallel form of program 1

points in the solution space. The best schedule may then be found by a process of inspection. This procedure seems to be quite wasteful; I will investigate another approach, in which one select a performance factor, and try to optimize it over the solution space.

Quite paradoxically, a possible solution is not to do anything. Suppose that the set of constraints is submitted to some version of the simplex algorithm. A classical version must be supplied with a linear objective function; the PIP software[30] uses lexical ordering as a ranking principle. In non parametric mode, this means that, given a polyhedron $\mathcal{P}$, PIP will find its lexicographic minimum $\underline{x}$. Equivalently, one may say that PIP will first find the minimum $\underline{x}_1$ of the first component of $x \in \mathcal{P}$. Next, PIP will find the minimum of the second component of x in $\mathcal{P} \cap \{x \mid x_1 = \underline{x}_1\}$, and so on[3].

Whatever the method chosen for solving the problem, there will be a tendency to obtain solutions with small values of the μ's. Since the schedules are linear combinations of the μ's with positive coefficients – see (14) – this will tend to produce good schedules. In the case of the example, one will directly obtain the solution (23).

In more complicated cases, the resulting schedule will be sensitive to such irrelevant details as the ordering of the domain definitions or of the Farkas multipliers, an unsatisfactory state of affairs. More definite results are needed here.

[3] The above is not a precise description of the way PIP works. The minimum is found directly; there is no successive minimization steps.

3.3.1 Minimum latency schedules

In the special case where all iteration domains are bounded – possibly with bounds depending on the structure parameters – one can define a total latency: the maximum value of all schedules, and try to find a minimum latency schedule. The method relies on the following

Lemma 8 *If all domains are bounded, and if there exists at least one affine schedule θ, then there exist at least one affine form in the structure parameters:*

$$L = h.n + k,$$

such that

$$\forall S, x \in \mathcal{D}_S : L - \theta(S, x) \geq 0. \tag{25}$$

> **Proof** Let us consider the set $\{v_{S1}, \ldots, v_{Sn}\}$ of vertices of $\mathcal{D}_S$. Since $\mathcal{D}_S$ is bounded, there are no rays in this set. Since a vertex is the intersection of a certain subset of the bounding hyperplanes of $\mathcal{D}_S$, and since the structure parameters occurs linearly in their equations, the coordinates of the vertices are affine forms in the structure parameters. As a consequence, for each vertex v_{Si}, $\theta(S, v_{Si})$ is an affine form in the structure parameters:
>
> $$\theta(S, v_{Si}) = h_{Si}.n + k_{Si}.$$
>
> Selecting $h \geq h_{Si}$ and $k \geq k_{Si}$ for all S and i, which is always possible, will satisfy (25) at all vertices, and as a consequence, everywhere in $\mathcal{D}_S$. $\square$

Structure parameters are supposed to be a good characterization of the size of the problem. This means that the latency increases with the structure parameters, i.e. that the coordinates of h are positive. By Farkas Lemma, (25) implies the existence of positive numbers ν_{Sk} such that:

$$L - \theta(S, x) \equiv \nu_{S0} + \sum_k \nu_{Sk}(a_{Sk} \begin{pmatrix} x \\ n \end{pmatrix} + b_{Sk}).$$

By the preceding lemma, this does not reduce the search space. One then proceeds as before. Identifying coefficients of independent variables gives a set of equations. Some ν's may be eliminated to give inequalities, which are added to those obtained in the previous paragraph. The problem is then solved, the unknowns in (25) being given the leading role. Since PIP finds the lexicographic minimal solution, this process will tend to give minimal values for the components of h, i.e. to find a schedule with minimal asymptotic latency. Applying this method to the above example again gives the solution (23), with a latency $L = n + 1$.

There are two problems with this method. The first one is that, in the case where there are several structure parameters, the solution may depend on their ordering. The other problem is well known in standard Operation Research. Minimizing the total latency is equivalent to finding a critical path in the DDG. The start time of operations on the critical path is fixed. For other operations, there is some leeway, and the algorithm will choose a solution more or less haphazardly, again depending on the ordering of the unknowns.

3.3.2 Bounded delay schedules

A bounded delay schedule[31] is a schedule such that, for all edges of the GDG:

$$\langle x, y\rangle \in \mathcal{R}_e \Rightarrow 1 \leq \theta(\delta(e), y) - \theta(\sigma(e), x) \leq \delta, \tag{26}$$

where δ is a constant integer.

The interest of bounded-delay schedules is twofold. Firstly, having a small delay is obviously an indication that the schedule is good, and this valuation may be used even in the presence of unbounded domains (i.e. for the case of on-line computational processes). Secondly, in the case of SRE or Dataflow Graphs, δ is a bound on the *lifespan* of each value in the computation, thus greatly facilitating memory allocation for the object program.

The technique for obtaining bounded-delay schedules should by now be familiar to the reader: introduce a set of Farkas multipliers for each inequality in (26), identify, and solve, δ being treated as the leading unknown, so as to obtain the minimum delay. In the case of the kernel in Fig. 1, one again obtains (23).

Bounded delay schedules are more constrained than minimum latency schedules. However, the minimum delay schedule is not unique. The reader may care to verify that (24) also is a schedule with delay 1 for the program in Fig. 1. In such a case the result may depend on ordering details. Beside

```
do i = 1,n
  s(i) = s(i) + a(i)
end do
do i = 1,n-1
  t(n-i) = t(n-i+1) + s(n-i)
end do
```

Figure 3: A program without a bounded-delay schedule

that, there is another difficulty, namely that there are GDG which have no bounded delay schedule. An exemple is given by the program of Fig. 3.

3.3.3 The dual method

All these difficulties may be solved at one step by attempting to find the best affine schedule. The order on schedule is *pointwise ordering*:

$$\theta_1 \prec \theta_2 \equiv \forall u \in \Omega : \theta_1(u) \leq \theta_2(u),$$

a partial order. As a consequence, the problem may appear to be very difficult. This is not so, on account of the following

Theorem 9 *If θ_1 and θ_2 satisfy the causality condition for a DDG $\langle \Omega, \Gamma \rangle$, then so does*

$$\theta_3(u) = \min(\theta_1(u), \theta_2(u)).$$

Proof The proof relies on two properties of the minimum function:

1. the minimum function is nondecreasing in both its arguments,
2. the identity $\min(x+1, y+1) = \min(x, y) + 1$.

Let u and v be two points of Ω such that $u \Gamma v$. As a consequence:

$$\begin{aligned} \theta_1(u) + 1 &\leq \theta_1(v), \\ \theta_2(u) + 1 &\leq \theta_2(v). \end{aligned}$$

By property 1:

$$\begin{aligned}\min(\theta_1(u)+1,\theta_2(u)+1) &\leq \min(\theta_1(v),\theta_2(u)+1)\\ &\leq \min(\theta_1(v),\theta_2(v)),\end{aligned}$$

and the conclusion follows by property 2. $\square$

This result suggests that one should look for schedules in the closure of the set of affine fonctions by the min operator. The functions in this set are piecewise affine functions. If $\mathcal{G}$ is the set of affine causal schedules for a given GDG, then a better piecewise affine schedule is given by:

$$\theta(u) = \min_{t\in\mathcal{G}} t(u). \tag{27}$$

Since affine functions are trivially concave and the min operator conserve concavity, θ is concave; it will be called in the following the *best concave schedule*. Consider the system of inequalities which is obtained from identities like (16) by a process of identification and elimination. Its unknowns are the Farkas multipliers in (14) – let them be the components of the vector μ – and some of the Farkas multipliers in (16) – let them be called λ. This system of constraints will be written:

$$G\begin{pmatrix}\mu\\ \lambda\end{pmatrix} \geq h, \tag{28}$$

where h is a constant vector. These inequalities fully characterize the set $\mathcal{G}$. The transcription of (27) is the following linear programming problem:

$$\begin{aligned}\theta(S,x) &= \min\ \mu_{S0} + \sum_{k=1}^{m_S}\mu_{Sk}(a_{Sk}.\begin{pmatrix}x\\ n\end{pmatrix}+b_{Sk}) \qquad (29)\\ &\quad\ \mu\geq 0,\\ &\quad\ \lambda\geq 0,\\ &\quad\ G\begin{pmatrix}\mu\\ \lambda\end{pmatrix}\geq h.\end{aligned}$$

For each value of x, this is a standard linear program which may be solved by some classical algorithm. This process is not satisfactory, because one will only get isolated numerical values of $\theta(S,x)$, rather than a closed form solution. What is needed is a parametric solution, the parameters being the coordinates of x. Now since the parameters in program (29) occur in the objective function, this program is not in a form suitable for solution by the PIP software[30] . This situation may be remedied by using one variant of the Linear Programming Duality theorem [12]:

Theorem 10 *If both linear programs:*

$$\begin{aligned} Z &= \min k.\mu, \\ &\mu \geq 0, \\ &G\mu \geq h, \end{aligned} \tag{30}$$

and

$$\begin{aligned} Y &= \max \nu.h, \\ &\nu \geq 0, \\ &\nu G \leq k, \end{aligned} \tag{31}$$

have solutions then these solutions are equal.

Application of this theorem directly gives the value of $\theta(S, x)$:

$$\begin{aligned} \theta(S,x) &= \max \nu.h, \\ \nu &\geq 0, \\ \nu G &\leq (\overbrace{0,\ldots,0}^{N}, 1, a_{S1}\begin{pmatrix} x \\ n \end{pmatrix} + b_{S1}, \ldots, a_{Sm_S}\begin{pmatrix} x \\ n \end{pmatrix} + b_{Sm_S}, \overbrace{0,\ldots,0}^{N'}). \end{aligned} \tag{32}$$

where:

$$\begin{aligned} N &= \sum_{R<S} (m_R + 1), \\ N' &= \sum_{S<T} (m_T + 1) + |\lambda|. \end{aligned}$$

There will be one such problem for each sattement S. Each of them may be solved in closed form by the PIP algorithm. Using the constraints:

$$a_{Sk}.\begin{pmatrix} x \\ n \end{pmatrix} + b_{Sk} \geq 0,\ k = 1, m_S,$$

```
   do i = 1,n
1    s(i) = 0.
   end do
   do i = 1,n
2    t(i) = t(i-1) + s(i)
   end do
```

Figure 4: A program with two different schedules

as context will simplify the resulting solution, which will be piecewise affine, since, by theorem 6, there is no need to impose integrality conditions. Application of this method to the program of Fig. 1 will again give (23).

It is interesting to compare the schedules obtained in this way to minimum latency schedules and bounded delay schedules.

In the first case, observe that the minimum latency schedule, θ_L, belongs to the set $\mathcal{G}$. Hence, according to (27), one has:

$$\forall u \in \Omega, \theta(u) \leq \theta_L(u).$$

Conversely, consider the maximum value of θ. Since this function is piecewise affine, the extremal point lies on an affine piece of θ which belongs to $\mathcal{G}$. This piece cannot have a lower latency than θ_L. As a consequence, θ has the same latency as θ_L.

The situation is quite different for bounded delay schedules. Consider the program of fig. 4 The best schedule is:

$$\begin{aligned} \theta(1, i) &= 0, \\ \theta(2, i) &= i, \end{aligned}$$

whose maximum delay is n. However, the program has the following unit delay schedule:

$$\begin{aligned} \tau_1(i) &= i - 1, \\ \tau_2(i) &= i. \end{aligned}$$

In this case, θ and τ have the same latency. This is not true in general.

3.3.4 Uniform recurrences

Let us consider the case of one recurrence equation with uniform dependences, or equivalently, of several equations with the added constraint that all schedules have the same linear part. The delay in this case is a constant. Furthermore, the constant term μ_0 may be taken as 0. Hence, one does not need Farkas Lemma for the expression of the causality condition. For each edge e with dependence vector d_e, the condition is:

$$\begin{aligned} \theta(x) - \theta(x - d_e) &= \sum_k \mu_k (a_k . \begin{pmatrix} x \\ n \end{pmatrix} + b_k) - \sum_k \mu_k (a_k . \begin{pmatrix} x - d_e \\ n \end{pmatrix} + b_k) \\ &= \sum_k \mu_k a_k . \begin{pmatrix} d_e \\ 0 \end{pmatrix} \geq 1. \end{aligned}$$

This is the uniform translation of (28). When going to the dual, the unknown vector ν has as many components as there are constraints, i.e. one component per dependence vector. The dual problem may be written in the following form:

$$\begin{aligned} \theta(x) &= \max \sum_{e \in E} \nu_e, \\ & \nu \geq 0, \\ & a_k . \begin{pmatrix} x - \sum_{e \in E} \nu_e d_e \\ n \end{pmatrix} + b_k \geq 0. \end{aligned}$$

This is exactly the generalization of linear program I in Karp et. al. paper [17] to the case where the iteration domain is arbitrary – Karp et. al. consider only the case of the first octant of iteration space.

As an exemple, consider the following problem[32]: find a causal schedule for

$$v(j_1, j_2) = g(v(j_1 - 2, j_2 - 2), v(j_1, j_2 - 3)) \tag{33}$$

in the domain

$$\mathcal{D} = \{j_1, j_2 \mid j_1 \geq 0, j_2 \geq 0, j_1 + j_2 \geq s, j_1 + j_2 \leq 2s\}. \tag{34}$$

The DFG of (33) is uniform and is described by two translations:

$$h_1(j_1, j_2) = \begin{pmatrix} j_1 - 2 \\ j_2 - 2 \end{pmatrix},$$

and

$$h_2(j_1, j_2) = \begin{pmatrix} j_1 \\ j_2 - 3 \end{pmatrix}.$$

The prototype schedule is:

$$\theta(j_1, j_2) = \mu_1 j_1 + \mu_2 j_2 + \mu_3(j_1 + j_2 - s) + \mu_4(2s - j_1 - j_2).$$

Computing the delays gives two constraints:

$$\begin{aligned} 2\mu_1 + 2\mu_2 + 4\mu_3 - 4\mu_4 &\geq 1, & (35)\\ 3\mu_2 + 3\mu_3 - 3\mu_4 &\geq 1, & (36) \end{aligned}$$

and we have to find the minimum value of θ under those constraints. The dual problem is:

$$\begin{aligned} t &= \max u + v, \\ u, v &\geq 0, \\ 0 &\leq 1, \\ 2u &\leq j_1, \\ 2u + 3v &\leq j_2, \\ 4u + 3v &\leq j_1 + j_2 - s, \\ -4u - 3v &\leq 2s - j1 - j2. \end{aligned}$$

Submitting this problem to PIP gives the following solution:

$\theta(j_1, j_2) =$ **if** $j_2 - j_1 + s \geq 0$
    then **if** $j_1 - s \geq 0$
        then $\frac{j_1 + 2j_2 - s}{6}$
        else $\frac{j_1 + j_2 - s}{3}$
    else $j_2/2$

This result may be interpreted as containing three schedules:

$$\begin{aligned} \theta_1(j_1, j_2) &= \left\lfloor \frac{j_1 + 2j_2 - s}{6} \right\rfloor, & (37)\\ \theta_2(j_1, j_2) &= \left\lfloor \frac{j_1 + j_2 - s}{3} \right\rfloor, & (38)\\ \theta_3(j_1, j_2) &= \lfloor j_2/2 \rfloor, & (39) \end{aligned}$$

each one being optimal in some subset of $\mathcal{D}$. For instance, θ_2 should be used in:

$$\mathcal{D}_2 = \{j_1, j_2 \mid j_1 \geq 0, j_2 \geq 0, j_1 + j_2 \geq s, j_1 - j_2 \leq 2s, j_1 < s\}.$$

All three solutions are found by Shang and Fortes' procedure, and only one is kept, namely θ_2, since it gives the minimum latency, $s/3$. The present method gives all three solutions and partition the iteration space, each subdomain corresponding to one particular schedule. This situation is depicted in figure 5.

The difference between the two approaches is brought out by another example of Shang and Fortes: solve recurrence (33) in the new domain:

$$\mathcal{D} = \{j_1, j_2 \mid j_1 \geq 0, j_2 \geq 0, j_1 \leq s_1, j_2 \leq s_2\}.$$

In that case, the prototype schedule is:

$$\theta(j_1, j_2) = \mu_0 + \mu_1 j_1 + \mu_2 j_2 + \mu_3(s_1 - j_1) + \mu_4(s_2 - j_2).$$

The delay constraints are:

$$2\mu_1 + 2\mu_2 \quad -2\mu_3 - 2\mu_4 \geq \quad 1, \tag{40}$$
$$3\mu_2 \qquad -3\mu_4 \geq \qquad 1. \tag{41}$$

The schedule is given by the solution of the dual problem:

$$\begin{aligned} \theta(j_1, j_2) &= \max u + v, & (42)\\ u, v &\geq 0, \\ 2u &\leq j_1, \\ 2u + 3v &\leq j_2, \\ -2u &\leq s_1 - j_1, & (43)\\ -2u - 3v &\leq s_2 - j_2.. & (44) \end{aligned}$$

It is quite clear that (43) and (44) are always true and may be removed. The remaining system is so simple that it can be solved graphically (see Figure 6). The solution is:

$$\theta(j_1, j_2) = \textbf{if } j_2 \geq j_1 \textbf{ then } \frac{j_1 + 2j_2}{6} \textbf{ else } j_2/2.$$

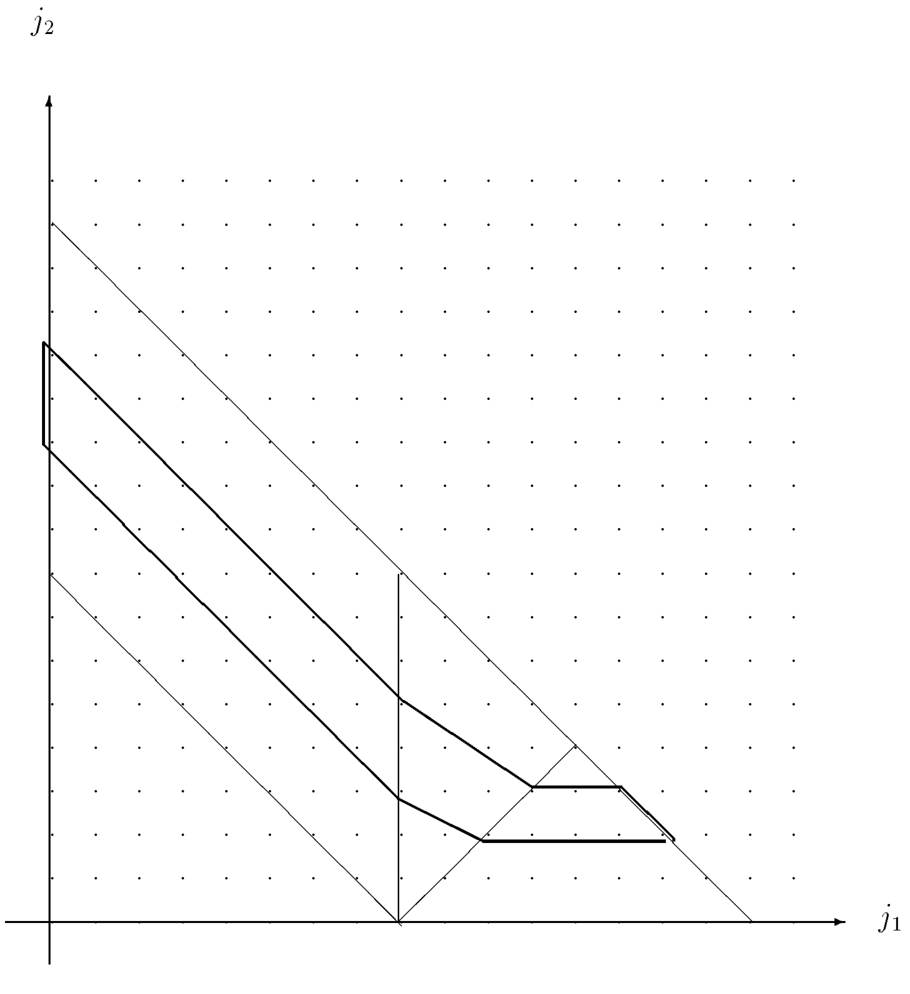

Figure 5: Shang and Fortes' exemple 3.1. The heavy line encloses the set $\{j_1, j_2 \mid \theta(j_1, j_2) = 1\}$.

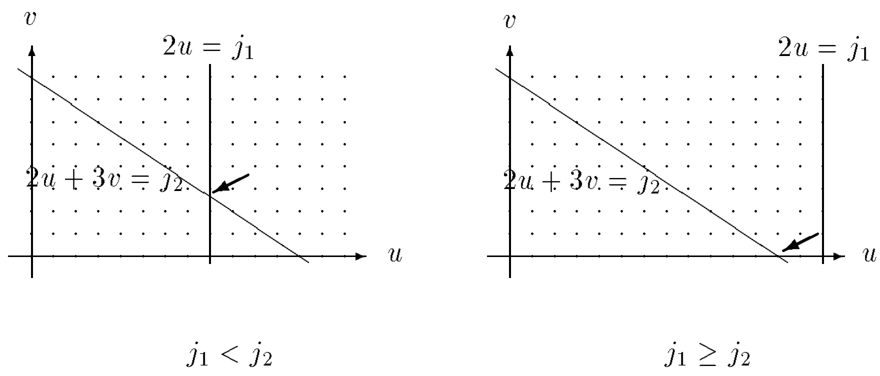

Figure 6: Solving program (42). The arrow indicates the position of the maximum.

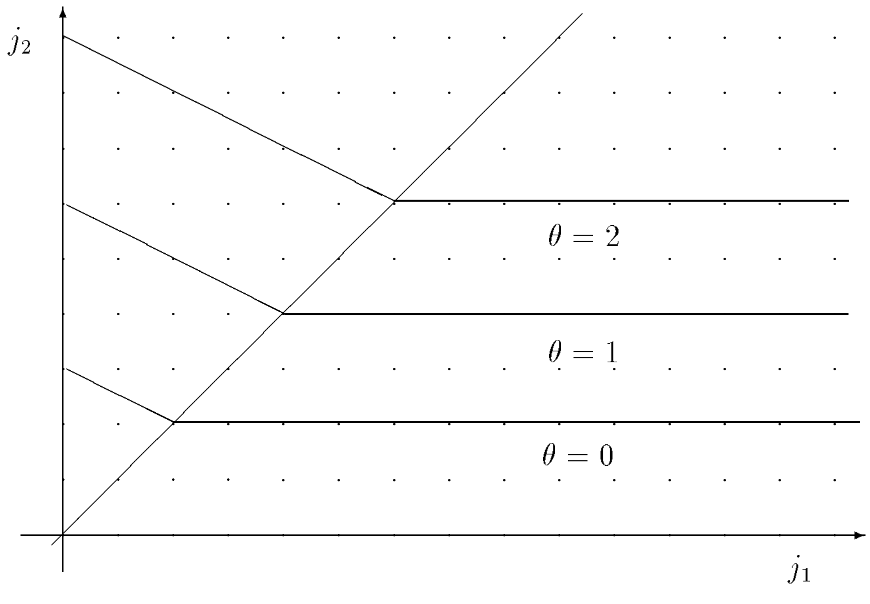

Figure 7: The solution of Shang and Fortes' example 3.4

The resulting schedule is depicted in Figure 7. Here again, the initial domain has been dissected in two subdomains and an affine schedule has been found in each of them. These partial schedules are the candidate schedules of Shang and Fortes, but instead of selecting one of them according to the value of s_1 and s_2, the iteration domain has been partitioned in two subsets where each component schedule is optimal. As a result, one does not have to build two different programs and select one of them at run time. Since the subdomains which are built by the PIP algorithm always are convex polyhedra, building the corresponding programs – or the corresponding systolic arrays – is no more difficult than processing the original domain.

4 Conclusion

This conclude the description of a new method for constructing affine schedules. The method is fast and simple; no enumeration is needed. Its main advantage is that it is not limited to uniform dependences. However, if, as is often the case, a DFG has many uniform dependences and some nonuniform ones, a very simple test allows one to substantially reduce the complexity of the solution process. Other laborsaving devices have been used in section 3.2.2: eliminating some Farkas multipliers, taking advantage of unknowns with only positive or only negative occurences. Ultimately, the solution is found by solving a parametric integer program of relatively small size, which is readily done by the PIP software. The method is even able to solve problems with unbounded domains: the reader may care to verify that in Shang and Fortes example 3.4 as treated above, the upper limits of the domain may be removed without changing the result. One may either try for a bounded delay schedule or for a best concave schedule. In the latter case, the result has minimum latency; if the DFG is uniform, the work of Darte et. al. [33] indicates that under quite natural hypothesis on the shape of the domain, the latency of the best concave schedule is asymptotically optimal.

The method has two drawbacks The first one is that it cannot find all piecewise affine schedules. The reason is that there are programs whose free schedule is not concave.

The program of figure 8 has the following free schedule:

$$\theta(i) = \mathbf{if}\ i > n\ \mathbf{then}\ 1\ \mathbf{else}\ 0,$$

```
for i = 0,2n
   x(i) = x(2n-i)
end
```

Figure 8: A program with a nonconcave schedule

while the dual Farkas algorithm gives simply:

$$\theta'(i) = i. \tag{45}$$

Furthermore, the free schedule has latency 1 while the best concave schedule has latency $O(n)$, thus showing that Darte et. al. result cannot be extended to the non uniform case.

One should note, however, that is is quite simple to check whether the result of any scheduling algorithm is optimal or not. One simply verify that each operation which has no predecessor in the extended DFG is scheduled at time 0, and that for others operations, one of the delays is 1. The reader may be interested in checking in this way that schedule (23) is the free schedule of program 1, while (45) is not the free schedule of program 8.

Most importantly, there are GDG which have no affine schedule. An example with its DFG is given in Fig. 9. The reader may care to verify, as a straightforward application of the above methods, that linear program (30) for that example is unfeasible. This fact may be understood if we remember lemma 8 which says in effect that an affine schedule has a latency which is asymptotically linear in the structure parameters. Now it is clear that the program of Fig. 9 has no parallelism, and that its minimum running time is about $\frac{n^2}{2}$, which is quadratic.

On the other hand, Dowling[6] proved that any program *instance* (in which the structure parameters have been given numerical values), has a linear schedule, and concluded, wrongly, that any program with sufficient loop nesting has lot of parallelism. In the case of Fig. 9, the Dowling schedule is:

$$\theta(i, j) = ni + j,$$

which is linear when n is given. This schedule has latency n^2; its mean degree of parallelism is about 1/2!

```
          do i = 0,n
             do j = 0,i
1               s = s + a(i,j)
             end do
          end do
```

Figure 9: A simple program with no linear schedule

Edge	Source	Destination	Condition
1	$(1, i, j-1)$	$(1, i, j)$	$j \geq 1$
2	$(1, i-1, i-1)$	$(1, i, j)$	$j < 1 \wedge i \geq 1$

Figure 10: The DFG of program 9

While finding a schedule for example 9 is not very rewarding, it may still happen, in more complicated cases, that a program has parallelism but no affine schedule. For instance, its latency may be $O(n^2)$ with a sequential execution time $O(n^3)$, giving a mean degree of parallelism of order n. The design of a method for scheduling such programs is the subject of Part II of this paper [11].

5 Acknowledgments

This work has been supported by PR C^3 and by the French Defense Ministry under contract DRET/87/280.

It is a pleasure to acknowledge many fruitful discussions with Patrice Quinton and his team in Rennes.

References

[1] E. Ayguadé, J. Labarta, J. Torres, J.M. Llaberia, and M. Valero. Nested-loop partitioning for shared-memory multiprocessor systems. In Paul Feautrier and François Irigoin, editors, *Procs of the Int. Workshop on Compiler for Parallel Computers, Paris*, December 1990.

[2] Paul Feautrier. Asymptotically efficent algorithms for parallel architectures. In M. Cosnard and C. Girault, editors, *Decentralized System*, pages 273–284. IFIP WG 10.3, North-Holland, December 1989.

[3] Patrice Quinton and V. van Dongen. The mapping of linear recurrence equations on regular arrays. *submitted to The Journal of VLSI Signal Processing*, September 1988.

[4] Sanjay V. Rajopadhye and Richard M. Fujimoto. Synthesizing systolic arrays from recurrence equations. *Parallel Computing*, 14:163–189, 1990.

[5] Leslie Lamport. The parallel execution of do loops. *CACM*, 17:83–93, February 1974.

[6] Michael L. Dowling. Optimal code parallelization using unimodular transformations. *Parallel Computing*, 16:157–171, 1990.

[7] Mourad Raji-Werth and Paul Feautrier. Systematic construction of programs for distributed memory systems. In Paul Feautrier and François Irigoin, editors, *Procs of the Int. Workshop on Compiler for Parallel Computers, Paris*, December 1990.

[8] William Pugh. Uniform techniques for loop optimization. *ACM Conf. on Supercomputing*, pages 341–352, January 1991.

[9] Lee-Chung Lu. A unified framework for systematic loop transformations. *SIGPLAN Notices*, 26:28–38, July 1991. 3rd ACM SIGPLAN Symp. on Priciples and Practice of Parallel Programming.

[10] Alain Darte and Yves Robert. Mapping uniform loop nests onto distributed memory architectures. Technical Report 93-03, LIP-IMAG, January 1993.

[11] Paul Feautrier. Somme efficient solution to the affine scheduling problem, part ii, multidimensional time. *Int. J. of Parallel Programming, to appear*, 1993. also available as Research Report IBP/MASI No. 92.78.

[12] A. Schrijver. *Theory of linear and integer programming.* Wiley, NewYork, 1986.

[13] Paul Feautrier. Dataflow analysis of scalar and array references. *Int. Journal of Parallel Programming*, 20(1):23–53, February 1991.

[14] J.R. Allen and Ken Kennedy. Automatic loop interchange. *SIGPLAN Notices*, 19(6):233–246, June 1984.

[15] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers*. Pitman and The MIT Press, 1989.

[16] François Irigoin and Rémi Triolet. Supernode partitioning. In *Proc. 15th POPL*, pages 319–328, San Diego, Cal., January 1988.

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

[18] Jacques Arsac. *La construction de programmes structurés*. Dunod, Paris, 1977.

[19] E. A. Ashcroft and W. W. Wadge. *Lucid, the Data-flow Programming Language*. Academic Press, 1985.

[20] L. G. Tesler and H. J. Enea. A language design for concurrent processes. In *SJCC*, pages 403–408, 1968.

[21] Marina C. Chen. A parallel language and its compilation to multiprocessor machines for vlsi. In *Proc. 1986 ACM POPL*, 1986.

[22] Hervé Leverge, Christophe Mauras, and Patrice Quinton. The ALPHA language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing*, 3:173–182, 1991.

[23] Paul Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing, St Malo*, 1988.

[24] Joel H. Salz, Ravi Mirchandaney, and Kay Crowley. Run-time parallelization and scheduling of loops. *IEEE Trans. on Computers*, 40:603–612, May 1881.

[25] Brigitte Jouannault. *Conception d'algorithmes et d'architectures systoliques*. PhD thesis, Université de Rennes I, September 1987.

[26] Yannick Saouter and Patrice Quinton. Computability of recurrence equations. Technical Report 521, IRISA, February 1990.

[27] Patrice Quinton. The systematic design of systolic arrays. In F. Fogelman, Y. Robert, and M. Tschuente, editors, *Automata networks in Computer Science*, pages 229–260. Manchester University Press, December 1987.

[28] Alain Darte and Yves Robert. Affine-by-statement scheduling of uniform loop nest over parametric domains. Technical Report 92-16, LIP-IMAG, April 1992.

[29] Hervé Leverge. A note on chernikova's algorithm. Technical Report 1992, INRIA, May 1992. Référence à vérifier.

[30] Paul Feautrier and Nadia Tawbi. Résolution de systèmes d'inéquations linéaires ; mode d'emploi du logiciel pip. Technical Report 90.2, IBP-MASI, January 1990.

[31] Christophe Mauras, Patrice Quinton, Sanjay Rajopadhye, and Yannick Saouter. Scheduling affine parameterized recurrences by means of variable dependent timing functions. Technical Report 1204, INRIA, April 1990.

[32] Weijia Shang and Jose A.B. Fortes. Time optimal linear schedules for algorithms with uniform dependencies. *IEEE Transactions on Computers*, 40(6), June 1991.

[33] Alain Darte, Leonid Kachian, and Yves Robert. Linear scheduling is nearly optimal. Technical Report 91-35, LIP-IMAG, November 1991. to appear in Parallel Processing Letters.

```
@ARTICLE{Feau:92aa,
AUTHOR = "Paul Feautrier",
TITLE = "Some Efficient Solutions to the Affine Scheduling Problem,
Part I, One Dimensional Time", VOLUME = 21, NUMBER = 5, MONTH = Oct,
JOURNAL = "Int. J. of Parallel Programming", YEAR = 1992
NOTE = "also available as Research Report IBP/MASI 92.28"
}
```