A method for static scheduling of dynamic control programs (preliminary version).

Jean-Francois Collard, Paul Feautrier

Laboratoire de l'Informatique du Parallélisme

Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

A Method for Static Scheduling of Dynamic Control Programs
Preliminary Version

Jean-François Collard
Paul Feautrier

December 1994

Research Report N° 94-34

Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00   Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

A Method for Static Scheduling of Dynamic Control Programs
Preliminary Version

Jean-François Collard
Paul Feautrier

December 1994

Abstract

Static scheduling consists in compile-time mapping of operations onto logical execution dates. However, scheduling so far only applies to static control programs, i.e. roughly to nests of `do` (or `for`) loops. To extend scheduling to dynamic control programs, one needs a method that 1) is consistent with unpredictable control flows (and thus unpredictable iteration domains) 2) is consistent with unpredictable data flows, and 3) permits speculative execution. This report describes a means to achieve these goals.

Keywords: Automatic parallelization, dynamic control program, while loop, scheduling, speculative execution

Résumé

L'ordonnancement statique consiste à attribuer lors de la compilation des dates logiques d'exécution aux opérations du programme. Cependant, les techniques d'ordonnancement ne s'appliquent jusqu'à présent qu'aux programmes à contrôle statique, c'est-à-dire typiquement aux imbrications de boucles `do` (ou `for`). Pour étendre ces techniques aux programmes à contrôle dynamique, il est nécessaire de trouver une méthode qui 1) soit compatible avec des flots de contrôle imprévisibles (et donc avec des domaines d'itérations imprévisibles) 2) soit compatible avec des flots de données imprévisibles, et 3) autorise éventuellement l'exécution spéculative. Ce rapport propose une telle méthode.

Mots-clés: Parallélisation automatique, programme à contrôle dynamique, boucle while, ordonnancement, exécution spéculative

Contents

1 Introduction

Static Control Programs (SCPs) have always been a central paradigm in compilers. Such programs have a structure that can be known at compile time. More precisely, one may statically enumerate all the operations spawned when executing an SCP. This enumeration may be parametrized w.r.t. symbolic "size" or "structure" parameters. To decide whether a program is an SCP, one must find syntactical criteria. SCPs in imperative languages are made of `for` (or `do`) loops and sequencing. `while` loops and `goto`s are forbidden. Moreover, `for` loop bounds must be tractable, and are usually restricted to affine forms. (SCPs in applicative languages are first order expressions [19].) SCPs generally have the additional constraint that array subscripts are affine functions of surrounding loop counters and size parameters.

In the case of SCPs, each execution spawns the same operations in the same order. Notice that this order may be partial [19, 8]. This is the very aim of automatic parallelization: find a partial order on operations respecting either all data dependences or just dataflow dependences [8]. The more partial the order, the higher the parallelism. Obviously, this partial order cannot be expressed as the list of relation pairs. One needs an expression of the partial order that does not grow with problem size. Such an expression may be a closed form, thus restricting the class of orders we can handle. Additional constraints on the choice of a partial order expression are: have a high expressive power; be easily found and manipulated; allow optimized code generation.

Well-known closed form expressions are *schedules*, i.e. mappings from operations onto *logical execution dates* [18]. These mappings are often functions from loop counters to integers. Two operations are not comparable iff they are scheduled to the same logical execution date, i.e., they may simultaneously execute on two distinct (virtual) processors.

So it seems that we have a sound and comprehensive framework for automatic parallelization. However, little work has been done so far on *Dynamic Control Programs* (DCPs). Such programs are just any programs, and include SCPs. Section 2, however, will give a more constrained definition of DCPs. The aim of this paper is to schedule DCPs, and the two contributions of this paper are: 1) to provide a single method to handle control dependences *or not*, depending on whether *speculative execution* is desired; and 2) to derive schedules that respect *parameterized sets* of data dependences, since no more precise information can be obtained in general.

Section 2 also gives necessary definitions and a brief review of dependence and array dataflow analyses for DCPs. Section 3 then describes how parallelism can be expressed thanks to (possibly multi-dimensional) schedules. Section 4 introduces speculative execution, an optional optimization. Section 5 details the algorithm which mechanically constructs the (possibly speculative) schedules. Section 6 concludes and discusses related works.

2 Definitions

The k-th entry of vector $\vec{x}$ is denoted by $\vec{x}[k]$. The dimension of a given vector $\vec{x}$ is denoted by $|\vec{x}|$. The subvector built from components k to l is written as: $\vec{x}[k..l]$. If $k > l$, then this vector is by convention the vector of dimension 0. Furthermore, $\underline{\ll}$ and $\ll$ denotes the non-strict and strict lexicographical order on such vectors, respectively. $\ll$ is defined by:

$$\begin{array}{rlll} \vec{x} \ll \vec{y} & \Leftrightarrow & \exists k, & 1 \leq k \leq \min(|\vec{x}|, |\vec{y}|),\ s.t. \\ & & & (\forall k',\ 1 \leq k' < k,\ \vec{x}[k'] = \vec{y}[k']) \\ & & \wedge & ((\vec{x}[k] < \vec{y}[k])\ \vee\ (\vec{x}[k] = \vec{y}[k] \wedge |\vec{x}| = k < |\vec{y}|))\,. \end{array} \quad (1)$$

In this paper, "max" always denotes the maximum operator according to the $\ll$ order. The integer division operator and the modulo operator are denoted by $\div$ and %, respectively. The true and false boolean values are denoted by *tt* and *ff*, respectively.

We first have to stress the difference between a *statement*, which is a syntactical object, and an *operation*, which is a *dynamic instance* of a statement. If a statement is included in a loop, then the execution yields as many instances of the statement as loop iterations. When only `do` loops appear in a program, giving names to statement instances is easy: one just has to label the operation by the statements' name and the corresponding loop counters' values.

Take for instance the following program:

```
     program A
     do i = 1 , n
          do j = 1, n
S :          a(i,j) = a(i,j-1)
          end do
     end do
```

The iteration vector for this nest is (i, j). The iteration domain of S is $\mathbf{D}(S) = \{(i,j) | 1 \leq i \leq n, 1 \leq j \leq n\}$. So, S spawns n^2 operations. An operation is denoted by $\langle S, i, j\rangle$.

However, we can easily add an artificial counter to any `while`-loop, whose initial value is also arbitrary, whose step is 1, and for which no upper bound is known. Note that detecting inductive variables may exhibit natural counters to `while`-loops. Hereafter, we will mimic the PL/1 syntax, i.e. use the construct below:

`while`-loop	Equivalent loop with explicit counter
`do while (` P `)` S	`do` $w = 1$ `by` 1 `while(` P `)` S

The program model we will restrict ourselves to is as follows:

- The only data structures are integers, reals, and arrays thereof.
- Expressions do not include any pointer or pointer-based mechanism such as aliasing, `EQUIVALENCE`, etc.
- Basic statements are assignments to scalars or array elements.
- The only control structures are the sequence, the `do` loop, the `while`- or `repeat`-loop, and the conditional construct `if..then..else`, without restriction on stopping conditions of `while` loops, nor on predicates in `if`'s. `goto`s are thus prohibited, together with procedure calls.
- Array subscripts must be affine functions of the counters of surrounding `do`, `while` or `repeat` loops and of structure parameters. The input program is supposed to be correct, thus subscripts must stay within array bounds.

  The fact that array subscripts stay within array bounds cannot be checked at compile-time when subscripts are expressions involving `while`-loop counters. On the other hand, one may do the opposite deduction: since the program is assumed to be correct, and subscripts stay within bounds, an affine subscript expression gives very informative affine constraints on `while`-loop counters. For instance, if the program below is correct,

  ```
  program lwn
  integer a(0:n)
  do w = 1 by 1 while ( ... )
    a(w) = ...
  end do
  ```

  then we can deduce that $0 \le l \le w \le n$. If the program is not correct, then so is this deduction, and the parallelized program is as incorrect as the input one.

For instance, Program `WW` follows our program model.

```
     program WW
G1 : do w  = 0 by 1 while ( P1(w) )
G2 :      do x  = 0 by 1 while ( P2(w,x) )
S  :         a( w + x ) = a( w + x - 1 )
          end do
     end do
```

For now, we will suppose that predicates P_1 and P_2 in Program `WW` do not depend on array `a`, but only on w and w and x, respectively.

We can now extend the definition of iteration vectors to `while` loops: the iteration vector of a statement appearing in a nest of `do` and/or `while` loops is the vector built from the counters of the surrounding loops. The dimension of iteration vector $\vec{x}$ is equal to the number of surrounding loops. For example, the iteration vector of statement S in Program `WW` is (w, x). An instance of S for a given value $\vec{x}$ of the iteration vector is denoted by $\langle S, \vec{x} \rangle$.

The true and false boolean values are denoted by *tt* and *ff*, respectively.

2.1 Iteration domains

The *iteration domain* $\mathbf{D}(S)$ of statement S is the set of values that the iteration vector takes in the course of execution. Unfortunately, iteration domains for dynamic control programs cannot be predicted at compile-time. In the particular case where there is only one outermost `while`-loop, we know at compile-time that the iteration domain is built from the integral points inside a convex polyhedron; this polyhedron is bounded if the loop terminates, but this bound cannot be known statically [2]. In more general cases, the iteration domain has no particular shape and looks like a (possibly multi-dimensional) "comb" [13].

An additional difficulty of DCPs when compared to SCPs lies in the handling of `while`-loop predicates. For instance, there is not a one-to-one correspondence between the evaluations of predicate $P_2(w, x)$ in program `WW` and the instances of S. Two frameworks have been proposed to describe such a phenomenon.

- Griebl and Lengauer [13] map to the same point of the iteration domain the evaluation of one or more `while`-loop predicates *plus possibly* the execution of a statement S appearing in the loop nest. Then, a single `while`-loop that does not iterate at all yields a one-element iteration domain.

- An alternative method is to consider the predicates of `if`s and `while`s as full-fledged statements having their own iteration domains, and to regard their instances as regular operations. We adopt this method since it allows to disambiguate the meaning of iteration domain elements, and to clarify the study of scheduling and speculative execution (to be discussed later).

  Let us go back to Program `WW`. Throughout this report, we consider an arbitrary execution such that the loop on `w` iterates 5 times (Predicate $P_1(w)$ evaluates to *ff* when $w = 5$), and the loop on `x` iterates 4, 0, 3, 5 and 2 times. G_2 executes one time more than S, i.e. 5, 1, 4, 6 and 3 times respectively.

  The method chosen by Griebl and Lengauer is illustrated for Program `WW` in Figure 1. Iteration domains of S, G_1 and G_2, according to our method, are displayed in Figure 2.

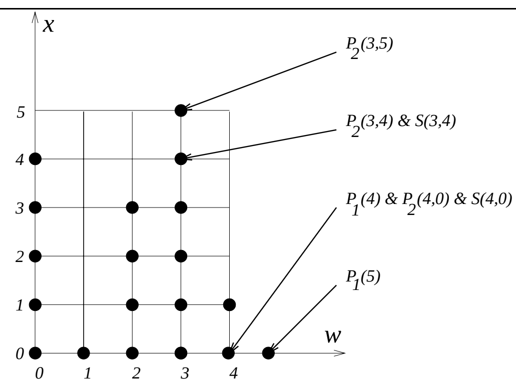

Figure 1: Instances of G_1, G_2 and S, for the arbitrary execution we consider in this report, according to the convention of Griebl and Lengauer.

2.2 Approximate iteration domains

Definition 1 *The approximate iteration domain* $\widehat{\mathbf{D}}(S)$ *of a statement* S *is the set of all instances of* S *when the predicates of all* `while` *loops and* `if`*s surrounding* S *evaluate to true.*

This unique approximate domain of S is a conservative superset of the (actual) iteration domain.

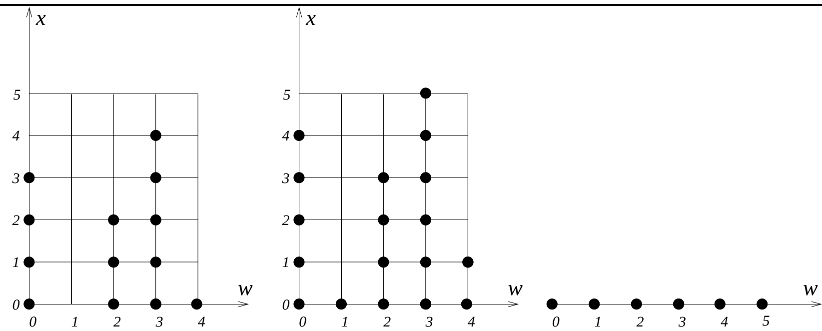

Figure 2: Iteration domains of S, G_2 and G_1, from left to right, for Program `WW`. Each dot represents an instance of one of these statements.

For example, the approximate iteration domain of S in Program `WW` is:

$$\widehat{\mathbf{D}}(S) = \{(w,x)|\, w \geq 0, x \geq 0\} \tag{2}$$

The approximate domain of G_1 is:

$$\widehat{\mathbf{D}}(G_1) = \{w|\, w \geq 0\} \tag{3}$$

The approximate iteration domain $\widehat{\mathbf{D}}(G_2)$ is equal to $\widehat{\mathbf{D}}(S)$. However, recall that for any given w, G_2 executes one more time than S. In Figure 3, black dots represent the corresponding instances of the three statements.

A very important remark is that, in a static control program, the approximate domain of any statement S is equal to the actual iteration domain, i.e. $\widehat{\mathbf{D}}(S) = \mathbf{D}(S)$ for any S, and there is no need for handling control dependences since they are already taken into account in the expression of $\mathbf{D}(S)$.

2.3 Scanning iteration domains

Griebl and Lengauer have shown that the image of the iteration domain of a nest of `do` and `while`-loops cannot always be scanned by another nest of `do` and `while`-loops, even when the mapping is affine and unimodular. Sufficient conditions for mappings to yield scannable image domains have been given [13].

When these conditions are not satisfied, the method proposed by these authors to scan the image domain consists in scanning a finite subset of the approximate image domain and in checking on the fly whether the current point is an element of the actual iteration domain:

$$\mathbf{D}(S) = \{\ \vec{x}\ |\ \vec{x} \in \widehat{\mathbf{D}}(S)\ \wedge\ executed(\vec{x})\}.$$

This test is done thanks to a predicate called *executed*, expressed as a recurrence on loop predicates.

For a precise and general definition of this predicate, the reader is referred to [12]. For our running example, this predicate is:

$$\begin{array}{rcl} executed(w,x) & = & executed_2(w,x) \\ executed_2(w,x) & = & x \geq 1 \rightarrow P_2(w,x) \wedge executed_2(w,x-1) \\ & & x = 0 \rightarrow P_2(w,x) \wedge executed_1(w) \\ executed_1(w) & = & w \geq 1 \rightarrow P_1(w) \wedge executed_1(w-1) \\ & & w = 0 \rightarrow P_1(w) \end{array}$$

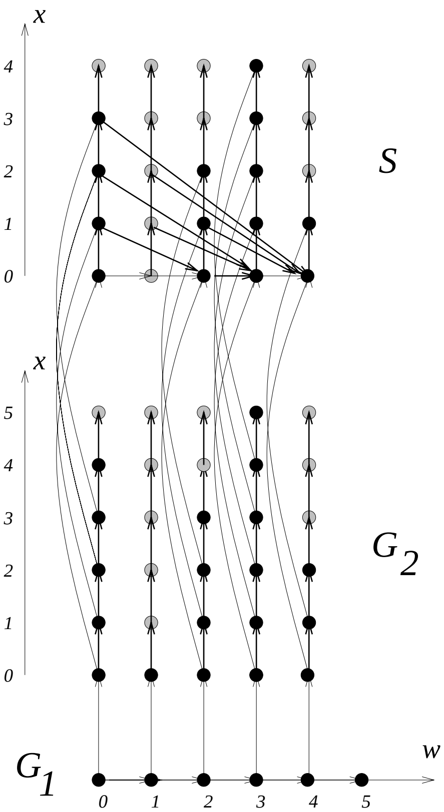

Figure 3: Dataflow and control dependence graph for Program WW. Each black dot represents an instance of G_1, G_2 or S for the arbitrary execution we consider. Gray dots represent possible instances that have to be considered, and thus both black and gray dots built the approximate iteration domains.

So we know now how to describe the operations spawned by a DCP. We now address the problem of finding the dependences among these operations.

2.4 Memory- and value-based dependences

Two operations can execute in parallel if they are independent, i.e. they do not interfere. Bernstein gave three sufficient conditions on two operations o_1 and o_2 for the program's semantics to be independent on the order in which these operations execute[1]. Let $R(o_1), M(o_1)$ $(R(o_2), M(o_2))$ be the set of memory cells read and modified by o_1 (o_2), respectively. Then, these operations are independent iff the three conditions below hold:

- C1: $M(o_1) \bigcap R(o_2) = \emptyset$
- C2: $M(o_2) \bigcap R(o_1) = \emptyset$
- C3: $M(o_1) \bigcap M(o_2) = \emptyset$

A few comments are in order here:

- If the first condition is not satisfied, then there is a *true dependence* or *producer-consumer dependence*, denoted by $o_1 \delta^t o_2$.

[1] These conditions are not necessary; for instance, executing `x:=x+1` and `x:=x+2` in any order does not change the semantics.

- If Condition C2 is false, then o_1 has read its input data in some memory cells and o_2 then reuses these cells to store its result. This is an *anti-dependence* or *consumer-producer dependence*, denoted by $o_1\bar{\delta}o_2$. There is an anti dependence on S in Program `WW`, corresponding to Edge e_8 in Figure 4.

- If Condition C3 is not satisfied, then there is an *output dependence* or *producer-producer dependence* denoted by $o_1\delta^0 o_2$. In Program `WW`, the output dependence between two instances $\langle S,\alpha,\beta\rangle$ and $\langle S,w,x\rangle$ of S is described by Edge e_7 in Figure 4.

If any condition C1, C2 or C3 is not satisfied, then o_1 and o_2 are said to be data dependent, denoted by $o_1\delta o_2$. Two operations o_1 and o_2 can execute in parallel if o_2 is not dependent on o_1 by transitive closure of δ. We say that a dependence from o_1 to o_2 is *satisfied* if o_1 executes before o_2. All dependences should be satisfied, thus limiting parallelism. Note that, should predicates P_1 and/or P_2 depend on array `a`, similar edges from S to G_1 and/or G_2 would just have to be added in Figure 4[2].

Edges	*Description*	*Conditions*
e_1	$\langle G_1, w-1\rangle\delta^c\langle G_1, w\rangle$	$w \geq 1$
e_2	$\langle G_1, w\rangle\delta^c\langle G_2, w, 0\rangle$	
e_3	$\langle G_2, w, x-1\rangle\delta^c\langle G_2, w, x\rangle$	$x \geq 1$
e_4	$\langle G_2, w, x\rangle\delta^c\langle S, w, x\rangle$	
e_5	$\langle S, w, x-1\rangle\delta^t\langle S, w, x\rangle$	$x \geq 1$
e_6	$\{\langle S,\alpha,\beta\rangle \mid \alpha+\beta = w+x-1, \alpha\geq 0, \beta\geq 0, \alpha < w\}\delta^t\langle S, w, 0\rangle$	$w \geq 1, x = 0$
e_7	$\{\langle S,\alpha,\beta\rangle \mid \alpha+\beta = w+x, \alpha\geq 0, \beta\geq 0, \alpha < w\}\delta^0\langle S, w, x\rangle$	$w \geq 1$
e_8	$\{\langle S,\alpha,\beta\rangle \mid \alpha+\beta-1 = w+x, \alpha\geq 0, \beta\geq 0, \alpha < w\}\bar{\delta}\langle S, w, x\rangle$	$w \geq 1$

Figure 4: Dependences in Program `WW`.

These dependences, however, are *memory-based dependences*. They are language- and program-dependent, and are not semantically related to the algorithm. On the contrary, *value-based dependences* or *data flows* capture the production and uses of computed values [1]. For instance, $\langle S,2,2\rangle$ in Program `A` is PC-dependent on both $\langle S,1,2\rangle$ and $\langle S,2,1\rangle$, but the only flow of data to $\langle S,2,2\rangle$ comes from $\langle S,2,1\rangle$. In the sequel, such a dataflow is denoted by Γ, e.g. $\langle S,1,2\rangle\Gamma\langle S,2,2\rangle$. Dataflow analysis for SCPs in the presence of arrays is now well understood [7, 22, 21, 24]. In the case of DCPs, a fuzzy array data flow analysis (FADA) has been proposed in [10]. The result of fuzzy array dataflow analysis is a multi-level conditional called *quast*. Each leaf is a *set* of potential dataflow sources. Notice that these sets may possibly be infinite. Each quast leaf is submitted to a *context* given by the conjunction of predicates appearing on the unique path from the quast's root to the leaf.

In Program `WW`, the source $\sigma(\langle S,w,x\rangle)$ of $\langle S,w,x\rangle$ given by FADA is:

$$\sigma(\langle S,w,x\rangle) = \left| \begin{array}{l} \textbf{if}\, x \geq 1 \\ \textbf{then}\, \{\langle S,w,x-1\rangle\} \\ \textbf{else} \left| \begin{array}{l} \textbf{if}\, w \geq 1 \\ \textbf{then}\, \{\langle S,\alpha,\beta\rangle \mid \alpha+\beta = w+x-1, \alpha\geq 0, \beta\geq 0, \alpha < w\} \cup \{\bot\} \\ \textbf{else}\, \{\bot\} \end{array}\right. \end{array}\right. \tag{4}$$

where $\bot$ means that the source operation does not exist, or more precisely, that any possible source operation lies outside the program segment. For instance, the context of the second leaf is $x < 1 \wedge w \geq 1$. The first two leaves give edges e_5 and e_6, displayed in Figure 5(a) and tabulated in Figure 4. In Figure 5(a), notice that some points have many incoming arrows, meaning that the real flow of value may be carried by any of them. These arrows correspond to the second leaf.

If there is no anti or output dependence, then the program has the *single-assignment property*. More memory is necessary, but since there are less constraints, the potential parallelism is greater. There exist

[2] Control dependences (δ^c type) are introduced in Section 4.

formal methods to convert SCPs into single-assignment form [6]. However, the case of DCPs is more intricate. Take for instance Program `I1`:

Program `I1`

```
if P then
     x = r_t
else
     x = r_e
end if
l = x
```

Program `I2`

```
   tmp = P
   if tmp then
        x1 = r_t
   else
        x2 = r_e
   end if
S: l = if tmp then x1 else x2
```

The single-assignment version `I2` of Program `I1` cannot be obtained without a dynamical mechanism to restore the flow of values in Statement S. Thus, even though converting a program into single-assignment form (SAF) generally exhibits more parallelism, restoring the flow of values may yield an intricate generated code. The pros and cons of SAF for DCPs are not well understood yet and more experiments are needed here. The method presented is this paper can handle both SA and non-SA programs.

2.5 Control dependences

2.5.1 Definition

There is a control dependence from operation o_1 to operation o_2 if the very execution of o_2 depends on the result of o_1. o_1 is called the *governing operation*. Such a dependence is denoted by $o_1 \delta^c o_2$. In particular, the very evaluation of a `while`-loop predicate (for instance, $\langle G_1, w\rangle$ in Program `WW`) is dependent on the outcome of the previous evaluation (e.g., on $\langle G_1, w-1\rangle$). The four control dependences of Program `WW`, call them $e_1..e_4$, appear in Fig. 4.

Notice that the outcome of a `while` predicate is given by anding the outcomes of all previous predicate instances plus the outcome of the current instance. For example, the outcome of $\langle G_1, w\rangle$ in Program `WW` is:

$$\bigwedge_{1 \leq w' \leq w} P_1(w').$$

Thus, a `while` predicate instance is both control and dataflow dependent on the previous predicate instances. This mixed dependence justifies the term *index dependence* coined by Griebl and Lengauer [14].

2.5.2 Description of control dependences

The case of the `if` construct Let us consider the following program piece:

```
G    if ( ... )
S         ...
     end if
```

where S is some statement in the `then` or `else` arm, perhaps surrounded by loops. Let c be the depth of the `if` construct, i.e. the number of loops surrounding G. Let $\vec{x}$ (resp. $\vec{y}$) be the iteration vector of G (resp. S). Then, there is a control dependence from $\langle G, \vec{x}\rangle$ to $\langle S, \vec{y}\rangle$ iff

$$\vec{y}[1..c] = \vec{x}. \qquad (5)$$

(if $c = 0$, then $\vec{x}$ and $\vec{y}[1..c]$ are equal to the vector of dimension 0 and equality (5) is true.)

The case of `while` loops Let us consider the following program piece:

```
G    while ( ... )
S        ...
     end while
```

where S is some statement in the `while`-loop body, perhaps surrounded by loops within the body. Let c be the depth of the `while` construct, i.e. the number of loops surrounding G. Let $\vec{x}$ (resp. $\vec{y}$) be the iteration vector of G (resp. S). Then, there is a control dependence from $\langle G, \vec{x}\rangle$ to $\langle S, \vec{y}\rangle$ iff

$$\vec{x}[1..c] = \vec{y}[1..c] \ \wedge\ \vec{x}[c+1] \leq \vec{y}[c+1] \tag{6}$$

We have now defined the various dependences that may appear in a program. The following section defines a suitable internal data structure for a parallelizing compiler to handle these dependences.

2.6 Internal data structures

2.6.1 Detailed dependence graph

The most intuitive structure is the detailed dependence graph. The vertices of this graph are program operations and the edges are dependences between these operations. When all data dependences are taken into account, the dependence graph for S in Program `WW` is depicted in Fig. 5(b). (There is no self control dependence on S.) When only dataflow dependences are taken into account, the dependence graph is shown in Figure 5(a). The leaves in (4) give the graph edges. In Figure 5(a), notice that some points have many incoming edges, meaning that the real flow of value may be carried by any of them. These edges correspond to the second leaf of (4).

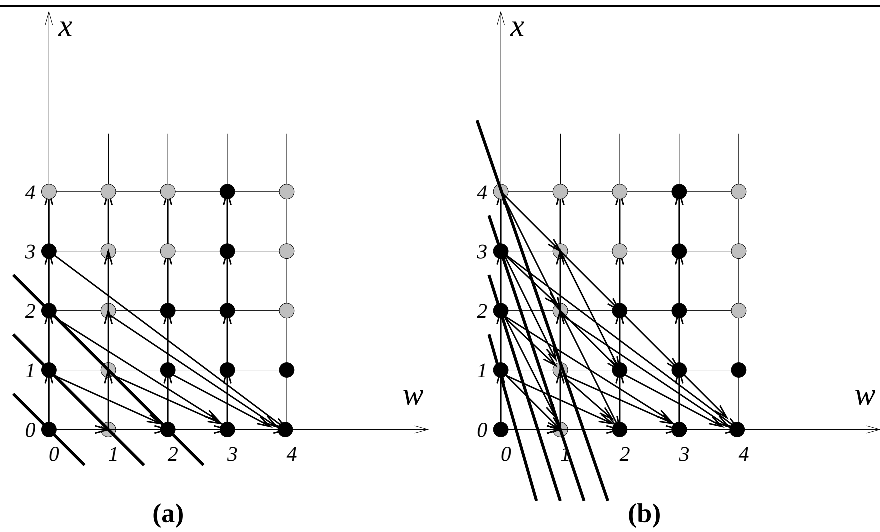

Figure 5: Dependence graphs for S in Program `WW`. Each dot represent a possible instance of S, but only dark dots denote real operations for the arbitrary execution we consider. Arrows represent data dependences: flow dependences in (a), and flow, output and anti dependences in (b). Dark lines represent possible *wavefronts*.

The detailed dependence graph has one vertex per operation, and thus is too big a data structure – it may even need an infinite number of vertices! We have to guarantee that sizes of internal data structures do not depend on sizes of program data structures nor on the number of spawned operations, i.e. we must be able to compile without knowledge of structure parameters values. We are thus looking for a *linearly described* graph, and the generalized dependence graph fulfills this requirement.

2.6.2 Generalized dependence graph

We augment the Generalized Dependence Graph (GDG) [8] to handle approximate iteration domains and possibly to include anti, output and control dependences. The latter are seen as regular data dependences and treated as such. The GDG is a directed multi-graph defined by:

A set $\mathcal{V}$ of vertices: Each vertex correspond to a statement in the program. More precisely, each vertex represents the set of operations the statement spawns. Note that the predicate expression of a `while` or an `if` is considered as a statement.

A set $\mathcal{E}$ of edges: There is an edge e from a source statement $t(e)$ (the edge's tail) to a sink statement $h(e)$ (the edge's head) if there is a dependence from $t(e)$ to $h(e)$. All dataflows (value-based dependences) incur an edge in the GDG; however, we will see in Section 4.4 that other types of dependences (e.g. control and memory-based) may or may not be taken into account. (Hence, corresponding edges may or may not be inserted in the GDG.) In any case, to each edge e is associated a set of constraints on the iteration vectors of $t(e)$ and $h(e)$.

A function $\widehat{\mathbf{D}}$ giving, for any statement S in $\mathcal{V}$, the conservative approximation $\widehat{\mathbf{D}}(S)$ of the iteration domain of S.

A function $\mathcal{R}$ giving, for each edge $e \in \mathcal{E}$, a relation on couples $(\vec{x}, \vec{y})$ described by a system of affine inequalities.

- If the edge corresponds to a dataflow, then this relation is given by the context of the corresponding quast leaf and the inequalities in the leaf's expression. By construction, $\mathcal{R}(e)$ is defined by affine inequalities, and thus is a polyhedron. Moreover, FADA guarantees that this polyhedron is not empty, a very useful property in the sequel. Notice $\vec{x}$ may take several values in a (polyhedral) set parametrized by $\vec{y}$, so the methods of [8, 9] can be applied.
- If the edge corresponds to a control dependence, then the relation captures equation (5) or (6).

3 Scheduling

3.1 Scheduling static control programs

Let Ω be the set of all operations, and $o_1, o_2 \in \Omega$ be two operations. *Scheduling* consists in choosing a set (generally, $\mathbb{N}$) and a strict order on this set (generally, $<$), and in finding a function from Ω to $\mathbb{N}$ such that either $o_1 \Gamma o_2 \Rightarrow \theta(o_1) < \theta(o_2)$ or $o_1 \delta o_2 \Rightarrow \theta(o_1) < \theta(o_2)$. If $\theta(o_1) = \theta(o_2)$, then o_1 and o_2 are scheduled to execute in parallel. This function is called the *scheduling function*, or, more simply, the *schedule*.

In Program `A`, $\langle S, i, j-1\rangle \Gamma \langle S, i, j\rangle$ if $j \geq 2$. On the other hand, $\langle S, i, j\rangle \not\Gamma \langle S, i', j\rangle)$, for any i'. Thus, a possible scheduling function for the operations spawned by Program `A` is $\theta(\langle S, i, j\rangle) = j - 1$. For a given j, all $\langle S, i, j\rangle, 1 \leq i \leq n$, are scheduled to execute in parallel.

Unfortunately, all programs do not have so simple schedules. Take for example Program `B`:

```
      program B
      do i = 1 , n
           do j = 1, n
S :           s = s+ a(i,j)
           end do
      end do
```

Suppose we cannot take benefit of algebraic properties of addition. Then, this program cannot be parallelized. Moreover, this program does not have a one-dimensional affine schedule [9]. However, a valid multi-dimensional component-wise affine schedule is, for instance:

$$\theta(\langle S, i, j\rangle) = \begin{pmatrix} i \\ j \end{pmatrix}.$$

In this case, the codomain of the scheduling function is $\mathbb{N}^2$, and the associated order is the strict lexicographical order, denoted by $\ll$. Hence, a more general definition of scheduling is either $o_1 \Gamma o_2 \Rightarrow \theta(o_1) \ll \theta(o_2)$, or $o_1 \delta o_2 \Rightarrow \theta(o_1) \ll \theta(o_2)$.

The *latency* of a schedule is, by definition $L = \text{Card}\ \theta(\Omega)$. For a one-dimensional schedule (whose period is 1), $L = \max \theta(\Omega) - \min \theta(\Omega) + 1$. Finally, notice that many different definitions appear in the literature: for some authors, schedules may have rational coefficients. Programs may have a single schedule for all statements or, on the contrary, one schedule for each statement. We will stick to the latter kind, and try to derive "affine-by-statement" schedules. In the sequel, for a statement S and an iteration vector $\vec{x}$, we denote $\theta_S(\vec{x})$ the logical execution date of $\langle S, \vec{x} \rangle$ instead of $\theta(\langle S, \vec{x} \rangle)$.

3.2 Scheduling dynamic control programs

On the contrary to SCPs, scheduling DCPs does not have an obvious meaning, since the scheduled operations may not execute at all. Scheduling an operation o_2 in a DCP means that, if this operation executes, then all preceding operations have been computed at previous scheduled dates. These preceding operations will be defined in Section 4.4.

If no `if` statement is allowed in DCPs and the only `while`-loop is the outermost loop, array dataflow analysis is exact and does not need tailored analyzes such as in [10]. An algorithm to schedule this restricted type of DCPs was previously proposed [2, 3]. This algorithm is extended in this paper to handle DCPs.

3.3 The need for multi-dimensional scheduling functions

This section answers the following question: Why should the scheduling function have possibly more than one dimension?

The main reason is that the class of DCPs includes all SCPs, and SCPs themselves require multi-dimensional schedules in the general case (see Program `B`). Moreover, they allow to easily express the behavior of programs built from `while` loops. Take for instance Program `W` (slightly modified from Program `simple` page ??):

```
        program W
        do w=0 by 1 while ( P )
S :         x = ...  x ...
        end do
R :     y = x
```

Since we cannot tell when predicate P evaluates to false, we have to consider a possibly non-terminating execution of the `while` loop. Valid schedules for S and R are

$$\theta_S(w) = \begin{pmatrix} 0 \\ w \end{pmatrix}, \quad \theta_R() = (1), \tag{7}$$

respectively. Since one cannot know at compile-time when Predicate P evaluates to false, one has to consider a possible non terminating `while`-loop. We also have to specify that $\langle R \rangle$ should execute after the last instance of S, which is unknown. A solution to this problem [13] is to use a *placeholder* denoted by δ, which essentially is a new variable equal to the execution date of the last instance of S. This placeholder is thus updated during execution, and the execution date of $\langle R \rangle$ is $\delta + 1$.

However, this method has two drawbacks according to us:

- Using placeholders is in a sense a *dynamic* scheduling. This is an acceptable choice, but the benefits of static scheduling are lost;
- Composition of schedules is not clear. For instance, let us consider the following program:

```
        program W
        do w=0 by 1 while ( P )
S1 :        x = ...  x ...
```

```
      end do
      do w=0 by 1 while ( P )
S2 :        z = ...  z ...
      end do
R :   y = x+z
```

Should the schedule of R be the maximum of the values of two placeholders, or an additional placeholder?

However, placeholders are necessary to code generation in the general case [11].

3.4 Existence of multi-dimensional schedules

Before proceeding on the scheduling problem, another question naturally arises: Do all DCPs have a multi-dimensional scheduling function?

To answer this question, we prove the following:

Proposition 1 *All DCPs respecting the restrictions of Section 2 have a multi-dimensional affine schedule.*

Proof (A constructive proof by induction on the structure of DCP ρ.)

$\rho =$`do w = 1 by 1 while` Q `end do.` Q is a SCP. Let θ be the schedule of a statement in Q, would the `while` loop be discarded. Then, $\begin{pmatrix} w \\ \theta \end{pmatrix}$ is a valid schedule for the selected statement of Q.

$\rho =$`if` p `then` Q `end if.` Q is a SCP. Let θ be the schedule of a statement in Q, would the conditional be discarded. Then (0) and $\begin{pmatrix} 1 \\ \theta \end{pmatrix}$ are valid schedules for p and the selected statement of Q, respectively.

$\rho = \rho_1; \rho_2$. ρ_1 and ρ_2 are DCPs. Let θ_1 (θ_2) be the schedule of a statement in ρ_1 (ρ_2). Then

$$\begin{pmatrix} 0 \\ \theta_1 \end{pmatrix}, \begin{pmatrix} 1 \\ \theta_2 \end{pmatrix},$$

are valid schedules for the selected statements of ρ_1 and ρ_2, respectively.

$\rho =$`if` p `then` Q_1 `else` Q_2 `end if.` Q_1 and Q_2 are SCPs. Let θ_1 (θ_2) be the schedule of a statement in Q_1 (resp. Q_2), would the conditional be discarded. Then,

$$(0), \quad \begin{pmatrix} 1 \\ \theta_1 \end{pmatrix}, \quad \begin{pmatrix} 1 \\ \theta_2 \end{pmatrix},$$

are valid schedules for the evaluation of p and for the selected statements of Q_1 and Q_2, respectively. (Notice that since instances of both Q_1 and Q_2 will not execute for a given value of the iteration vector, the first components of their schedules can be equal.)

□

Note that the proof did not try to minimize schedule dimension. Obviously, we should try to take benefit of special cases, such as the possible knowledge of an upper bound u on a `while` loop counter w.

4 Speculative execution

Intuitively, one gets speculative executions by ignoring or "cutting" control dependences. More formally:

Definition 2 *The execution of operation o is said to be speculative if there exists o_c such that $o_c \delta^c o$ and o_c executes after or simultaneously with o.*

For a detailed discussion of speculative execution, see [17, 3]. Notice that control dependences between instances of the same `while` predicate can be cut, but the corresponding dataflow cannot. This boils down to saying that index dependences cannot be cut.

However, thanks to scheduling functions, we can give a more precise definition of speculative execution which will allow to derive useful properties.

Definition 3 *The execution of operation o is speculative if at least one control dependence on o is not satisfied, i.e. there exists an operation o_c governing o whose execution date is later than the execution date of o:*

$$\exists o_c \in \Omega \mid o_c \delta^c o \wedge \theta(o_c) \geq \theta(o).$$

4.1 Legality of Speculative execution

Obviously, speculative execution is legal if and only if the semantics of the input program is preserved. Three necessary conditions can then be stated:

The control flow must be restored. Speculative operations are committed or not depending on the outcomes of governing operations. These governing operations must thus execute in finite time. Once a speculative operation is executed, the corresponding governing operation must executes in finite time. That is, the number of operations executed after or simultaneously with the speculative operation and before or simultaneously with the governing operation has to be finite.

As a consequence, notice that parallel fronts should be finite. When speculative operation is not brought into play, the only executed operations are those belonging to some (actual) iteration domain. On the contrary, speculative execution executes points from *approximate* iteration domains. Thus, we must take care that speculative fronts are finite or *limited* [14]. An easy way to guarantee finiteness of fronts is to enforce that fronts are not parallel to a nonnegative affine combination of the approximate domain's rays [3]. However, finite fronts do not imply that delays between speculative operations and their governing operations are finite (there may be an infinite number of finite fronts), but the converse is true.

The flow of data must be restored. When potential sources come from speculative operations, one has to take care that these operations were executed and committed before reading the datum.

Side-effects from speculative operations must be masked. These side-effects are writes to memory and exceptions (I/O operations are not considered). For a discussion of these issues, please read [3, 20]. In this paper, we will assume that no exception occurs and that each operation writes into its own private memory cell (i.e., the program has the single-assignment property). Then, speculative operations do not overwrite non-speculative results, and the initial memory state can be restored [3].

To illustrate the second and third dangers of speculative execution, and to show the limits of our method, let us study the following program:

```
     Program simple
G:   do w = 0 by 1 while ( P(x) )
S:     x = ...
     end do
R:   ...  = x
```

If this program is converted into single assignment form, there are no more output dependences on S. Remaining dependences are:

Edges	Dependences	Conditions
e_1	$\langle G, w\rangle \delta^c \langle S, w\rangle$	
e_2	$\langle G, w-1\rangle \delta^c \langle G, w\rangle$	$w \geq 1$
e_3	$\langle S, w\rangle \delta^t \langle R\rangle$	$w \geq 0$
e_4	$\langle S, w-1\rangle \delta^t \langle G, w\rangle$	$w \geq 1$

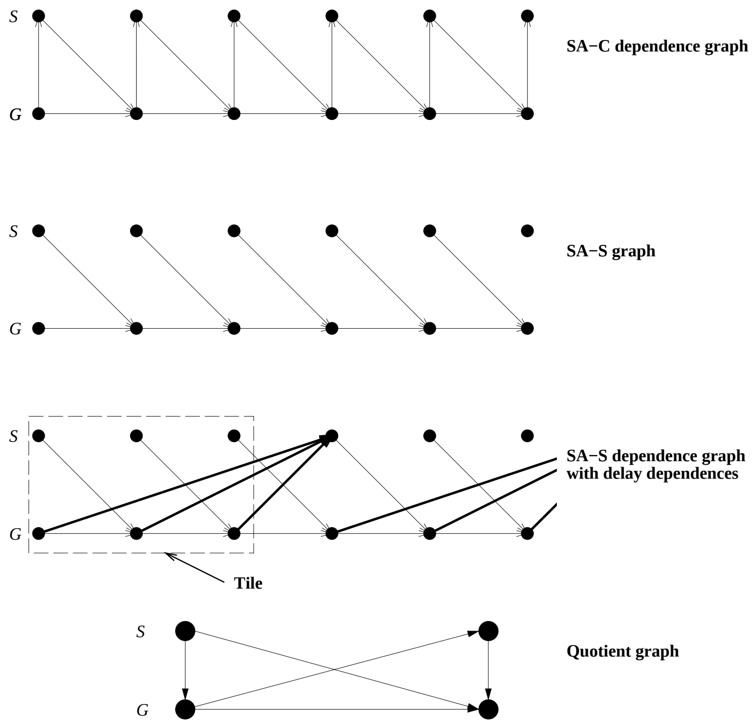

Figure 6: Dependence graphs for statements G and S of Program `simple`. From top to bottom: regular ("SA-C") dependence graph; dependence graph without edge e_1 – a topological sort yielding an infinite front; dependence graph where delay control dependences replace e_1; corresponding quotient graph, where supernodes appear in an acyclic graph.

The corresponding dependence graph appear in top of Figure 6. If control dependence e_1 is "cut", then the dependence graph is still consistent. However, a topological sort would execute all the possible instances of S simultaneously (see second graph in Figure 6):

- This topological sort yields an infinite front:

$$\{\langle S, w\rangle | \, w \geq 0\}. \tag{8}$$

  Equivalently, the schedule for S is $\theta(\langle S, w\rangle) = 0$.
- The read in $\langle R\rangle$ requires that the flow of data is re-constructed, and thus that the last instance of S is known. To know this instance, we have to know the outcome of all instances of G.

4.2 Restoring the flow of control

As we said, speculative execution should be used carefully. Intuitively, not taking a control dependence into account may unleash a nonterminating behavior. In the case of DCPs where the only `while` loop is the outermost loop, a necessary and sufficient condition to restore the flow of control is that fronts must be finite [2]. The proposition below is more general and subsumes the finiteness of fronts.

Proposition 2 *An operation o can be speculatively executed in a safe way iff the set $\Upsilon(o_c, o)$ of operations scheduled between o and o_c is finite, i.e. iff*

$$\Upsilon(o_c, o) = \{u | \, \theta(o) \lll \theta(u) \lll \theta(o_c)\} \tag{9}$$

is finite.

Proof Let L be the date of the last scheduled operation, and W be the work performed by the program:

$$W = \sum_{t=0}^{L} s(t), \tag{10}$$

where $s(t)$ is the cardinal of the front at time t:

$$s(t) = Card(\{u | \theta(u) = t\}).$$

A program can be executed in finite time on a finite number of processors iff W is finite. In particular, for a given operation o_c governing a speculative operation o, (10) implies that:

$$\sum_{t=\theta(o)}^{\theta(o_c)} s(t) < \infty,$$

which is equivalent to saying that $\Upsilon(o_c, o)$ is finite, hence the proposition. □

Testing this condition in a naive way would require to enumerate all possible statements (of whom u is an instance), and split the inequalities according to (1). Notice that enforcing this condition bound both the resources and the time required by speculation.

Front (8) given by topological sort has a ray along the w-axis. As said in section 4.1, our method forbids such an infinite front because this front is parallel to the ray. However, a more general condition is given by Proposition 9: $\theta(\langle G, 0\rangle) = 0$ and $\theta(\langle S, w\rangle) = 0$, hence:

$$\Upsilon(\langle G, 0\rangle, \langle S, 0\rangle) = \{\langle S, w\rangle \mid w \geq 0\},$$

which is not finite. (Thus, our method is not able to parallelize Program `simple`.)

Comments on this method Proposition 9 gives an a posteriori test on the given schedules (to be constructed in Section 5). However, one may try to take benefit of speculative execution using pseudo-affine schedules. Future work will tackle this issue, but this paragraph just presents the main idea. Roughly speaking, executing all possible instances of S, i.e. executing all elements of (8) in parallel, is "too speculative".

The mistake in the above example was to cancel all instances of dependence e_1 in the dependence graph. Instead of canceling all instances of a control dependence, a method is to replace them with delay dependences so as to bound speculative execution. For instance, control dependence e_1

$$\langle G, w\rangle \delta^c \langle S, w\rangle,$$

could be replaced by

$$\langle G, w\rangle \delta^R \langle S, w + r(w)\rangle,$$

where $r(w)$ is a nonnegative integer delay. Such dependences allow to tile the iteration domain, and to schedule each tile independently in a speculative way (see Figure 6).

However, constructing these delay dependences is still an open problem. Moreover,, schedules are in general not affine any more. In the case of Figure 6, valid pseudo-affine schedules for G and S would be:

$$\theta_G(w) = \begin{pmatrix} w \div 3 \\ w \,\%\, 3 \end{pmatrix}, \; \theta_S(w) = \begin{pmatrix} w \div 3 \\ 0 \end{pmatrix}.$$

Such schedules are beyond the scope of this report.

4.3 Restoring the flow of data: compensation dependences

Problem description If the source of a read is a singleton (as given by the fuzzy array dataflow analysis), then the identity of the source does not depend on the flow of control. In other words, if the read executes, the the source executes too.

However, if the source is not a singleton, then we cannot decide at compile-time which operation among the source set is the last executed one. Existence of a possible source depends on the outcome of all governing predicates from `while`s and `if`s, which is formalized by control dependences. Hence, care must be taken when cutting control dependences, since selecting the actual dataflow source depends on them. As a consequence, we must ensure that, given operations u, v, w such that $u\delta^c v$ and $v \in \sigma(w)$, if dependence $u\delta^c v$ is cut, then u still executes before w. To enforce this property, we insert a dependence from u to w. Intuitively, this dependence *compensates for* the cut control dependence, and is denoted by $u\delta^{comp}w$.

We saw that, in Program `simple`, executing operation $\langle R\rangle$ requires the knowledge of the outcomes of all instances of G. So, we insert compensation dependence $\langle G, w\rangle\delta^{comp}\langle R\rangle$, for all $w \geq 0$.

Here is another example:

```
S0 :   x = ...
G :    if ( ... ) then
S1 :     x = ...
       end do
R :    ...  = x
```

Speculative execution of S_1 can be scheduled before the execution of G. However, R needs to know who produced datum `x` among S_0 and S_1. Notice that this problems only appear because the flow of data is fuzzy: the source of `x` in R is $\{S_0, S_1\}$, the source for R in Program `simple` is $\{\langle S, w\rangle | \, w \geq 0\}$. We compensate edge $G\delta^c S_1$ by a *compensation dependence* $\langle G\rangle\delta^{comp}\langle R\rangle$.

Construction of compensation dependences Let us consider a control dependence edge e_1 in the GDG, from some instances of statement G to some instance of statement S, which we intend to cut:

$$\langle G, \vec{x}\rangle \; \delta^c \; \langle S, \vec{y}\rangle \; s.t. \; \mathcal{R}_{e_1}(\vec{x}, \vec{y}), \tag{11}$$

where $\mathcal{R}_{e_1}(\vec{x}, \vec{y})$ is a system on affine constraints on $\vec{x}, \vec{y}$, labelling edge e_1 in the GDG.

The problem is as follows: For any statement R, whose iteration vector is $\vec{z}$, such that there is a dataflow edge e_2 from $\langle S, \vec{y}\rangle$ to $\langle R, \vec{z}\rangle$ if $\mathcal{R}_{e_2}(\vec{y}, \vec{z})$ holds, construct the set:

$$C(\langle R, \vec{z}\rangle) = \{\langle G, \vec{x}\rangle \mid \mathcal{R}_{e_1}(\vec{x}, \vec{y}) \; \wedge \; \mathcal{R}_{e_2}(\vec{y}, \vec{z})\}$$

Since $\mathcal{R}_{e_1}(\vec{x}, \vec{y})$ and $\mathcal{R}_{e_2}(\vec{y}, \vec{z})$ are given by systems of affine constraints, computing $C(\langle R, \vec{z}\rangle)$ can easily be done: make the conjunction of both systems and eliminate variables $\vec{y}$. Hence, this boils down to projecting variables $\vec{y}$ out.

At first sight, this method has two drawbacks: first, it may be costly. Second, the resulting set cannot always be described as the integral points in a convex polyhedron; to be consistent, we may in the general case have to approximate the resulting set by its hull. However, the second problem seldom occurs due to the form of $\mathcal{R}_{e_1}$ given by (5) or (6).

Let us consider the program below:

```
G1 :   do w1 = 0 by 1 while ( ... )
G2 :         do w2 = 0 by 1 while ( ... )
S :                a(w1 + w2) = ...
             end do
R :          ... = a(k)
       end do
```

Control dependences on S are:

$$\langle G_1, w_1'' \rangle \delta^c \langle S, w_1', w_2' \rangle \ s.t.\ w_1'' \leq w_1' \tag{12}$$

and

$$\langle G_2, w_1'', w_2'' \rangle \delta^c \langle S, w_1', w_2' \rangle \ s.t.\ w_1' = w_1'', w_2' \geq w_2''.$$

Assume dependence (12) is cut. Then, since the source of $\langle R, w_1 \rangle$ is

$$\sigma(\langle R, w_1 \rangle) = \{\langle S, w_1', w_2' \rangle \mid w_1' \geq 0, w_2' \geq 0, w_1' \leq w_1, k = w_1' + w_2'\},$$

$C(\langle R, w_1 \rangle)$ is:

$$C(\langle R, w_1 \rangle) = \{\langle G_1, w_1'' \rangle \mid w_1'' \leq w_1', w_1' \geq 0, w_2' \geq 0, w_1' \leq w_1, k = w_1' + w_2'\},$$

that is:

$$C(\langle R, w_1 \rangle) = \{\langle G_1, w_1'' \rangle \mid w_1'' \leq w_1, w_1'' \leq k\}.$$

As a conclusion, cutting dependence (12) implies inserting a compensation dependence edge e_3 in the GDG such that $t(e_3) = G_1$ and $h(e_3) = R$, labeled with $\mathcal{R}_{e_3}(e) = \{w_1'' \leq w_1, w_1'' \leq k\}$.

4.4 Parallelization modes

Depending on whether speculative execution is brought into play (S) or not (conservative, C), and whether the program is converted into single assignment form (SA) or not (NSA), four parallelization modes exist. Each mode yields, for a given operation o_2, a set of preceding operations:

NSA-C The set of preceding operations is

$$\{o_1 \,|\, o_1 \delta^c o_2 \vee o_1 \delta o_2\}.$$

This is the mode of classical compilers.

SA-C The set of preceding operations is

$$\{o_1 \,|\, o_1 \delta^c o_2 \vee o_1 \Gamma o_2\}.$$

SA-S The set of preceding operations is

$$\{o_1 \,|\, o_1 \Gamma o_2 \ \vee \ o_1 \delta^{comp} o_2\}. \tag{13}$$

This mode speculates on operation executions but is able to give back the original semantics.

NSA-S The set of preceding operations is

$$\{o_1 \,|\, o_1 \delta o_2 \ \vee \ o_1 \delta^{comp} o_2\}. \tag{14}$$

This mode executes as many speculative operations as possible but is not able to "rollback" and restore the original semantics when these speculative executions happen to be mispredicted. This mode would require that the compiler knows very special properties on the algorithm; such a property was first described in [28] for "convergent `while` loops": when the stopping condition evaluates to *tt*, then all following iterations evaluate the condition to *tt* too. According to us, this is a dangerous property that a compiler should not assume.

We will thus restrict ourselves to the first three parallelization modes, and, in all cases, automatically derive a scheduling function to all program statements. Notice that all four sets of preceding operations may be infinite.

4.5 Examples

We illustrate the definitions above on three examples. The first example program cannot be parallelized without using speculative execution. On the contrary, the second example does not need speculation to be parallelized. The third program is slightly different from the second example; however, it cannot be parallelized without speculation, and moreover there exist no safe (affine) speculative schedules for this program. (Notice that in all three programs, scheduling functions are supposed given. Constructing them in an automatic way is the subject of Section 5.)

4.5.1 First example

```
       Program Iteratif
T :    x = a(n) + δ     /* δ > ε */
G :    do w = 1 by 1 while ( | x − a(n) | ≥ ε )
          x = a(n)
          do i = 1 , n
S :           a( i ) = a( i ) + a( i − 1 )
          end do
       end do
```

Let s be the iteration count of the `while` loop during the sequential execution. Then, this program executes in $s \times n$ tops. Moreover, this program cannot be parallelized, even if converted into single-assignment form. However, one may "bet" that the current iteration will not be the last one, and speculate. Formally, this boils down to canceling control dependences from $\langle G, w\rangle$ to all $\langle S, w, i\rangle$, for all i, $1 \leq i \leq n$. Only then can the program be parallelized. Figure 7 displays the corresponding parallel fronts (dark lines), assuming that the input program was first converted into single-assignment form (SA-S mode). This parallel program

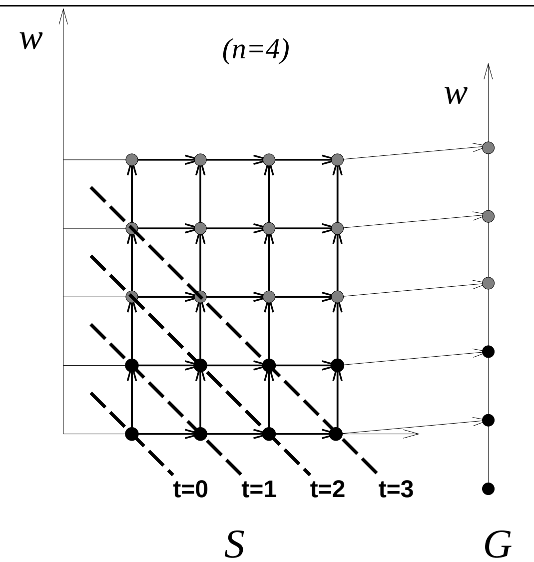

Figure 7: Approximate iteration domains for Statements G and S of Program `Iteratif`. Dataflows are displayed by thin arrows. Discarded control dependences are displayed in dashed lines. Bold lines correspond to parallel fronts for schedule $w + i - 2$ of S.

executes in $s + n$ tops on n processors. Possible schedules are:

$$\theta(\langle S, w, i\rangle) = w + i - 2,$$

and

$$\theta(\langle G, w\rangle) = w + n - 1.$$

Let us check that Proposition 2 is satisfied:

$$\forall i, i \le i \le n,\ \Upsilon(\langle G, w\rangle, \langle S, w, i\rangle) = \{\langle S, w', i'\rangle \,|\, \theta(\langle S, w, i\rangle \lll \theta(\langle S, w', i'\rangle) \lll \theta(\langle G, w\rangle)\},$$

that is,

$$\forall i, 1 \le i \le n,\ Card(\Upsilon(\langle G, w\rangle, \langle S, w, i\rangle)) = Card(\{(w', i') \,|\, w+i-2 \le w'+i'-2 \le w+n-1,\ w' \ge 1,\ 1 \le i' \le n\}.$$

The cardinal above is finite because the coefficient of w in $\theta(\langle S, w, i\rangle)$ is nonzero. Intuitively, a scheduling function whose w coefficient is zero yields infinite fronts along the w axis [2]. w coefficients cannot be negative (since that would correspond to executing the `while` loop in the order opposite to the sequential order), so $w \in \mathbb{N}^\star$. Notice that the smaller the value of the coefficient of w, the faster the execution (since the latency, for a given finite Ω, is minimized with respect to w when this coefficient is equal to 1). Hence, a schedule with w coefficient equal to 1 is in a sense the "optimal" speculative schedule.

4.5.2 A second example, without speculation

Let us go back to Program `WW`. The corresponding dependences are summed up in Figure 4 and depicted in Figure 8. Parallelization mode SA-C keeps all edges except for e_7 and e_8. On this example, some parallelism can be extracted without resorting to speculative execution. A topological sort shows that possible valid schedules are:

$$\theta(\langle G_1, w\rangle) = w,$$
$$\theta(\langle G_2, w, x\rangle) = w + x + 1,$$
$$\theta(\langle S, w, x\rangle) = w + x + 2.$$

(Notice that we do not need to check Proposition 2 since these schedules are not speculative.) If conversion into single-assignment were not applied (i.e. mode NSA-C is chosen), all edges e_1 though e_8 would have to be considered, and the fastest schedule would be $\theta(\langle S, w, x\rangle) = 3w + x + 2$ as can be checked by hand using topological sort.

4.5.3 An example with speculation

We now tackle a slightly different example, where a `while`-loop predicate, say P_1, depends on side-effects from the nest body. Suppose P_1 is a function of w and of a scalar variable s. To avoid adding a statement, we use a notation "à la C" where assignments are expressions:

```
      program WWb
G1 :  do w  = 0 by 1 while ( P1(w,s) )
G2 :    do x  = 0 by 1 while ( P2(w,x) )
S :       s  = a( w+x ) = a( w+x-1 )
        end do
      end do
```

A new dataflow dependence is thus added to dependences of Figure 4 :

Edge	*Description*	*Conditions*
e_9	$\{\langle S, w-1, x\rangle \,|\, x \ge 0\}\delta^t\langle G_1, w\rangle$	$w \ge 1$

Notice that the approximate source of $\langle G_1, w\rangle$ is an infinite set.

If the program is put into single assignment form (SA- mode), dependences e_1 to e_6 and e_9 are taken into account. The corresponding graph appears in Figure 9 (where only one instance of e_9, from $\{\langle S, 0, x\rangle \,|\, x \ge 0\}$ to $\langle G_1, 1\rangle$ is displayed to get a simpler figure.) This program does not have any parallelism. A solution is to cancel control dependence e_2. Then, the parallel fronts we previously found for S and G_2 are valid again.

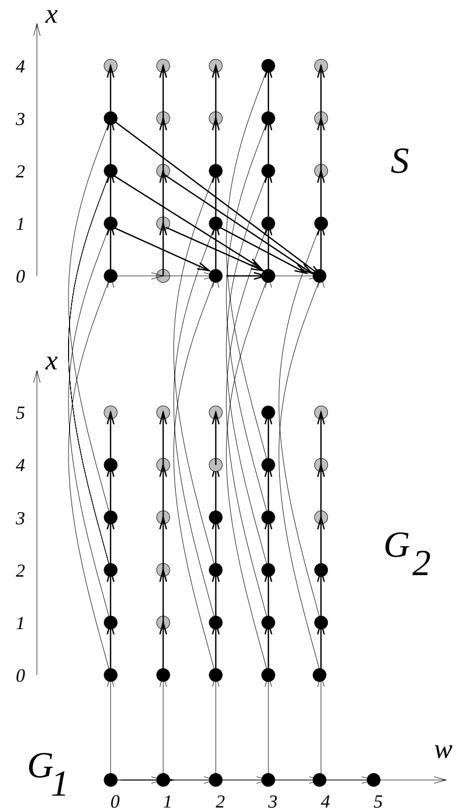

Figure 8: Graph of control and flow dependences for Program `WW`.

Unfortunately, scheduling G_1 now causes the following problem: $\langle G_1, w\rangle$ must execute after all operations $\langle S, w-1, x\rangle$, i.e.:

$$\theta(\langle G_1, w\rangle) > \max_{x\geq 0} \theta(\langle S, w-1, x\rangle) = \max_{x\geq 0} w + x.$$

This inequality cannot be satisfied if no upper bound on x is known. Using a second schedule dimension yield schedules:

$$\theta(\langle G_1, w\rangle) = \left(\begin{array}{c} 1 \\ w \end{array}\right),$$

$$\theta(\langle G_2, w, x\rangle) = \left(\begin{array}{c} 0 \\ w+x \end{array}\right),$$

$$\theta(\langle S, w, x\rangle) = \left(\begin{array}{c} 0 \\ w+x+1 \end{array}\right).$$

However, we are then in an extreme case where speculative execution may not terminate. According to these schedules, all evaluations of predicate P_1 are done before completion of all instances of S and G_2. However, we have no guarantee that all instances of the loop on `x` terminate, i.e. that for any w, there is an x_0 such that $P_2(w, x_0) = \mathit{ff}$. Just imagine that $P_1(w, s) = \mathit{ff}$ and $P_2(w, x) = \mathit{tt}$! This fact can be checked thanks to (9):

$$\begin{array}{rcl} \Upsilon(\langle G_1, w\rangle, \langle S, w', x'\rangle) & = & \left\{ \langle S, w'', x''\rangle \middle| \left(\begin{array}{c} 0 \\ w'+x'+1 \end{array}\right) \ll \left(\begin{array}{c} 0 \\ w''+x''+1 \end{array}\right) \ll \left(\begin{array}{c} 1 \\ w \end{array}\right) \wedge w'' \geq 0 \wedge x'' \geq 0 \right\} \\ & \cup & \left\{ \langle G_2, w'', x''\rangle \middle| \left(\begin{array}{c} 0 \\ w'+x'+1 \end{array}\right) \ll \left(\begin{array}{c} 0 \\ w''+x'' \end{array}\right) \ll \left(\begin{array}{c} 1 \\ w \end{array}\right) \wedge w'' \geq 0 \wedge x'' \geq 0 \right\} \end{array}$$

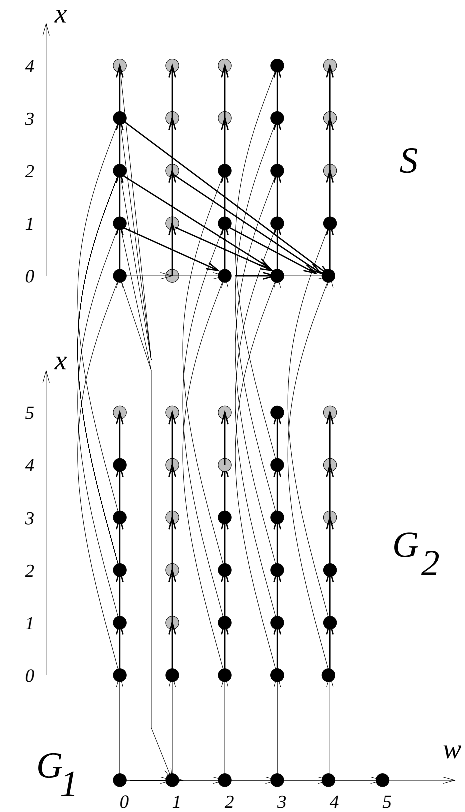

Figure 9: Control and flow dependences in Program `WWb`. Each dot denotes an instance of G_1, G_2 or S (respective iteration domains appear in this order from top to bottom.)

Obviously,

$$\Upsilon(\langle G_1, w\rangle, \langle S, w', x'\rangle) \;\supseteq\; \{\langle S, w'', x''\rangle \,|\, w'' \geq w' \wedge x'' \geq 0\}. \tag{15}$$

Hence $\Upsilon(\langle G_1, w\rangle, \langle S, w', x'\rangle)$ is infinite. Our method for speculative scheduling thus fails, and Program `WWb` is executed sequentially.

5 An algorithm for automatic static scheduling

Let us go back to Program `WW`. We can simultaneously execute all the operations belonging to a given wavefront depicted in Fig.5. Cases (a) and (b) correspond to single-assignment form SA-S and regular form NSA-S, respectively. Parallelism in the both cases can be expressed by wavefront equations: $w + x = K$ and $3w + x = K$, respectively, where K is a parameter. As expected, the amount of parallelism is smaller in the latter case, and the corresponding program latency is higher. (Latencies are equal to 8 and 14, resp.) Since approximate domains are infinite, one cannot know latencies at compile-time. However, a rule of thumb is to consider that the smaller the coefficients, the faster the execution so the better the schedule. The purpose of the algorithm below is to find the equations of these wavefronts.

5.1 Driving algorithm

The core of the method is an algorithm whose input is a GDG and whose output is a multidimensional affine-by-statement schedule (Section 5.2). However, from a given GDG, many sub-graphs can be derived by canceling some control dependences. This core algorithm has thus to be driven by an algorithm whose task is to try and find a sub-graph of the initial GDG whose schedule is in a sense optimal.

The driving algorithm is described in Figure 10. It takes as input a GDG $\mathcal{G}$ and a function `scheduling` (the core algorithm), and returns a valid, possibly speculative schedule for G. This driving algorithm first finds a non-speculative schedule. It then cancels one control (non "index") dependence at a time, and calls the core algorithm to obtain the corresponding schedule. As explained before, a good metric for schedules is latency, but the latter cannot be defined for DCPs. Thus, a rule of thumb is to pick the schedules whose coefficients are the smallest. Note that changing the metric (for instance, schedule delays) would not change the driving algorithm. Nevertheless, this algorithm can trivially be improved, for instance by considering all possible combinations of control dependences. The aim of the next section is to propose an algorithm for `scheduling`.

```
θc  := scheduling( 𝒢 )
for all non-index control dependences  d
     let 𝒢' := 𝒢  minus  d  plus  compensation dependences
     θ := scheduling( 𝒢' )
     if ( θ better than  θc ) then  θc  :=  θ end if
end for
return ( θc )
```

Figure 10: Driving algorithm looking for a speculative schedule.

5.2 Core algorithm

The aim of this part is to find, for a given GDG and for each statement S in $\mathcal{V}$, an integer d_S and a multi-dimensional component-wise-affine function θ_S from $\mathbf{D}(S)$ to $\mathbb{N}^{d_S}$ such that, for any edge e from $t(e)$ to $h(e)$ in $\mathcal{E}$, the *delay* Δ_e

$$\Delta_e(\vec{x},\vec{y}) = \theta_{t(e)}(\vec{x}) - \theta_{h(e)}(\vec{y}) \tag{16}$$

satisfies:

$$\Delta_e(\vec{x},\vec{y}) \gg \vec{0} \tag{17}$$

For any statement S, the codomain of θ_S is $\mathbb{N}^{d_S}$. However, we cannot describe the domain $\mathbf{D}(S)$ at compile-time. So, we *over-constrain* θ_S and require that it is nonnegative on the *approximate* domain:

$$\forall \vec{x} \in \widehat{\mathbf{D}}(S), \theta_S(\vec{x}) \geq \vec{0}. \tag{18}$$

Then, we use the fact that $\widehat{\mathbf{D}}(S)$ is a polyhedron defined by p affine inequalities:

$$\widehat{\mathbf{D}}(S) = \{\vec{x}|\, A\vec{x} - \vec{b} \geq \vec{0}\}. \tag{19}$$

The problem is as follows: for any statement S, construct a function θ_S satisfying (17) and (18), and defined on (19). To do this, we apply the following lemma:

Lemma 1 (Affine Form of Farkas' Lemma) *An affine function $\theta_S(\vec{x})[d]$ is non-negative on a polyhedron defined by (19) if there exists a set of non-negative integers $\mu_0, \ldots, \mu_p$ (the Farkas coefficients) such that*[3]*:*

$$\theta_S(\vec{x})[d] = \mu_0 + \Sigma_{k=1}^{p} \mu_k (A_{k\bullet}.\vec{x} - \vec{b}[k]). \tag{20}$$

The delay being the difference of two schedules, each delay can thus be expressed as a function of the μ's.

[3] The k^{th} component of a vector $\vec{x}$ is denoted by $\vec{x}[k]$ and the k^{th} row of a matrix A by $A_{k\bullet}$.

Let us go back to Program WW. Eq. (3) implies that there exist two integers ζ_0 and ζ_1 such that, for any d,

$$\theta_{G_1}(w)[d] = \zeta_0 + \zeta_1 w. \tag{21}$$

The conservative iteration domain $\widehat{\mathbf{D}}(S)$ of statement S is (2). Thus, there exist integer coefficients $\lambda_0, \lambda_1, \lambda_2$ and μ_0, μ_1, μ_2 such that

$$\theta_{G_2}(w,x)[d] = \lambda_0 + \lambda_1 w + \lambda_2 x \tag{22}$$

and:

$$\theta_S(w,x)[d] = \mu_0 + \mu_1 w + \mu_2 x. \tag{23}$$

Basically, the algorithm is identical to the one in [9]. Intuitively, we would like to find, for all statements, non-negative one-dimensional schedules satisfying (17) for any edge e. In this case, $d = 1$ and (17) is equivalent to:

$$\begin{array}{l} \vec{x} \in \widehat{\mathbf{D}}(t(e)),\ \vec{y} \in \widehat{\mathbf{D}}(h(e)),\ (\vec{x},\vec{y}) \in \mathcal{R}(e), \\ \Delta_e(\vec{x},\vec{y})[d] = \theta_{t(e)}(\vec{x})[d] - \theta_{h(e)}(\vec{y})[d] > 0. \end{array} \tag{24}$$

Initially, d is equal to 1. Then, the algorithm satisfies in a greedy way as many edges as possible until all of them can be canceled:

- If (24) can be satisfied for all statements and all edges for the current value of d, then the algorithm terminates.
- If no instance of (24) can be satisfied, then the greedy algorithm fails.
- Otherwise, we have to add a dimension to all schedules involved in unsatisfied constraints (24), and we increment d. We then go back to Step 1 to handle remaining schedules and edges.

The algorithm will thus iteratively try to satisfy all such constraints, adding one dimension to some schedules at each iteration.

Let $\mathcal{U}^{(1)}$ be the set of edges such that (24) is satisfied for $d = 1$. Its complement in $\mathcal{E}$ is such that:

$$\begin{array}{rcl} e \notin \mathcal{U}^{(1)} & \Rightarrow & \exists \vec{x} \in \mathbf{D}(t(e)),\ \exists \vec{y} \in \mathbf{D}(h(e)),\ (\vec{x},\vec{y}) \in \mathcal{R}(e), \\ & & s.t.\ \Delta_e(\vec{x},\vec{y})[1] = \theta_{t(e)}(\vec{x})[1] - \theta_{h(e)}(\vec{y})[1] = 0. \end{array}$$

How can we tell the elements of $\mathcal{U}^{(1)}$ from the others? If $\mathcal{R}(e)$ is a singleton and the dependence is uniform, then we can directly solve (24) for the Farkas coefficients. Otherwise, as remarked in [26], $\Delta_e(\vec{x},\vec{y})$ is defined on the set:

$$\{(\vec{x},\vec{y}) \mid \vec{y} \in \mathbf{D}(t(e)),\ \vec{x} \in \mathbf{D}(h(e)),\ (\vec{x},\vec{y}) \in \mathcal{R}(e)\}, \tag{25}$$

which is a non-empty convex polyhedron. The inequalities defining this set are just the conjunction of the inequalities defining $\mathbf{D}(t(e)), \mathbf{D}(h(e))$, and $\mathcal{R}(e)$. Let n_e be the number of resulting inequalities. These inequalities can collectively be written as:

$$\forall k,\ 1 \le k \le n_e,\ \Psi_{e,k}(\vec{x},\vec{y}) \ge 0.$$

Let ϵ_e be an auxiliary integer variable encoding the fact that e belongs to $\mathcal{U}^{(1)}$ or not. Then, if

$$\Delta_e(\vec{x},\vec{y}) - \epsilon_e$$

is a non-negative form for $\epsilon_e = 1$, then the one-dimensional causality constraint (24) is satisfied. Otherwise, $\epsilon_e = 0$. Since the domain of Δ_e is not empty, we can apply the Affine Form of Farkas' Lemma again: there is a set of non-negative integers $\nu_0, \ldots, \nu_{n_e}$ such that:

$$\Delta_e(\vec{x},\vec{y}) - \epsilon_e = \nu_0 + \Sigma_{k=1}^{n_e} \nu_k \Psi_{e,k}(\vec{x},\vec{y}). \tag{26}$$

This yields a system of linear equations. If this system can be solved with $\epsilon_e = 1$, then (24) is satisfied.

We did not precise, however, which solution should be picked among possibly many solutions. In SCPs, one may try to minimize schedule latencies. However, schedules for DCPs may be defined on non bounded domains. For instance, Statement S in Program `W` has schedule

$$\theta_S(w) = \begin{pmatrix} 0 \\ w \end{pmatrix},$$

whose latency is undefined. In such cases, an intuitive rule of thumb is to chose the μ's and ν's to be as small as possible, since this tends to reduce the latency. More formal criteria are given in [8].

Let us apply the above algorithm to Statements S and R in Program `W`. Prototype schedules are $\theta_S(w) = \lambda_0 + \lambda_1 w$ and $\theta_R() = \mu_0$, respectively. If we just take dataflow dependences into account, then the source of right-hand-side `x` in S is:

$$\sigma(\langle S, w\rangle) = \textbf{if } w \geq 1 \textbf{ then } \{\langle S, w-1\rangle\} \textbf{ else } \{\perp\}.$$

The source of `x` in R can be just any instance of S, if any, and FADA yields:

$$\sigma(\langle R\rangle) = \{\perp\} \cup \{\langle S, w\rangle | \, w \geq 0\}.$$

The first dependence is uniform, giving the inequality below ($d = 1$):

$$(24) \Rightarrow \Delta_{e_1}(w)[d] = \theta_S(w)[d] - \theta_S(w-1)[d] \geq \epsilon_1 \Leftrightarrow \lambda_1 \geq \epsilon_1.$$

The second edge is a parametrized set of dependences, and the method in [8] cannot be applied. Instead, we have to consider the delay:

$$\begin{aligned} \Delta_{e_2}(w)[d] &= \theta_R[d] - \theta_S(w)[d] - \epsilon_2 \\ &= \mu_0 - \lambda_0 - \lambda_1 w - \epsilon_2 \end{aligned} \tag{27}$$

On the other hand:

$$\Delta_{e_2}(w)[d] = \nu_0 + \nu_1 w \tag{28}$$

Equaling the members of (27) and (28) gives:

$$Constants: \quad \mu_0 - \lambda_0 - \epsilon_2 = \nu_0 \tag{29}$$

$$w: \quad -\lambda_1 = \nu_1 \tag{30}$$

Since all Farkas coefficients are non-negative, the only solution to the last equation is $\lambda_1 = \nu_1 = 0$. This implies that $\epsilon_1 = 0$, i.e. the first edge is not satisfied. Now, a possible solution to the entire system is $\epsilon_2 = \mu_0 = 1, \lambda_0 = \nu_0 = 0$. Thus, the first schedule components are $\theta_S(w)[1] = 0 + 0w = 0$ and $\theta_R[1] = 1$.

During the second iteration of the algorithm, $d = 2$ and the only edge still to be satisfied is the first one, i.e. $\lambda_1 \geq \epsilon_1$ for $\epsilon_1 = 1$. The smallest solution is $\lambda_1 = 1$. Since there is no condition on λ_0, it is set to 0. Thus, $\theta_S(w)[2] = w$, and we have automatically found the schedules in (7).

5.3 Program `WW` revisited

We now apply the algorithm of the previous section to automatically derive the nonspeculative scheduling function on Program `WW` (SA-C mode).

We handle one by one all the dependences of Figure 4. Dependence e_1 is uniform, and since the schedule prototype for G_1 is (21), this dependence yields:

$$\zeta_0 + \zeta_1 w - (\zeta_0 + \zeta_1(w-1)) \geq \epsilon_1,$$

that is:

$$\zeta_1 \geq \epsilon_1 \tag{31}$$

Then, since (22), dependence e_2 yields:

$$\lambda_0 + \lambda_1 w + \lambda_2 x - (\zeta_0 + \zeta_1 w) \geq \epsilon_2. \tag{32}$$

Edge e_3 is uniform. The delay is:

$$\lambda_0 + \lambda_1 w + \lambda_2 x - (\lambda_0 + \lambda_1 w + \lambda_2(x-1)) \geq \epsilon_3 \Longleftrightarrow \lambda_2 \geq \epsilon_3. \tag{33}$$

Edge e_4 yields the following constraint:

$$\mu_0 + \mu_1 w + \mu_2 x - (\lambda_0 + \lambda_1 w + \lambda_2 x) \geq \epsilon_4 \tag{34}$$

Edge e_5 is uniform too, hence:

$$\mu_0 + \mu_1 w + \mu_2 x - (\mu_0 + \mu_1 w + \mu_2(x-1)) \geq \epsilon_5 \Longleftrightarrow \mu_2 \geq \epsilon_5. \tag{35}$$

Edge e_6 subsumes a parametrized set of non-uniform dependences. The delay $\Delta_{e_6}(\alpha, \beta, w)$ is defined on a set described by the following Ψ inequalities:

$$w - 1 \geq 0, \alpha \geq 0, \beta \geq 0, w - \alpha - 1 \geq 0, \alpha + \beta - w + 1 \geq 0, w - \alpha - \beta - 1 \geq 0.$$

Thus, there exists a set of integer coefficients $\nu_0 ... \nu_7$ such that:

$$\begin{aligned} \Delta_{e_6}(\alpha, \beta, w) &= \mu_0 + \mu_1 w - (\mu_0 + \mu_1 \alpha + \mu_2 \beta) - \epsilon_6 \\ &= \nu_0 + \nu_1 w + \nu_2 \alpha + \nu_3 \beta + \nu_4 (w - \alpha - 1) \\ &\quad + \nu_5(\alpha + \beta - w + 1) + \nu_6 (w - \alpha - \beta - 1) \end{aligned} \tag{36}$$

Equaling the coefficients of the same variables gives:

$$Constants: \quad \nu_0 - \nu_4 + \nu_5 - \nu_6 = -\epsilon_6 \tag{37}$$

$$w: \quad \nu_1 + \nu_4 - \nu_5 + \nu_6 = \mu_1 \tag{38}$$

$$\alpha: \quad \nu_2 - \nu_4 + \nu_5 - \nu_6 = -\mu_1 \tag{39}$$

$$\beta: \quad \nu_3 + \nu_5 - \nu_6 = -\mu_2 \tag{40}$$

Since our aim is to have as small schedule latencies as possible, we have to look for small solution values. Eq.(31) is satisfied when $\zeta_1 = \epsilon_1 = 1$. Equations (33) and (35) are satisfied when $\lambda_2 = \mu_2 = \epsilon_3 = \epsilon_5 = 1$. Eq.(32) yields no constraint on ζ_0, so $\zeta_0 = 0$; and we can set $\lambda_0 = \lambda_1 = 1$. Eq. (40) is satisfied when $\nu_6 = 1, \nu_3 = \nu_5 = 0$. Then, Eq. (38) implies that $\mu_1 \geq 1$, and is satisfied when $\mu_1 = 1, \nu_1 = \nu_4 = 0$. Now Eq. (34) is satisfied for ϵ_4 when $\mu_0 = 2$. We have thus automatically found the expected schedules:

$$\theta_{G_1}(w) = w,$$

$$\theta_{G_2}(w, x) = w + x + 1,$$

$$\theta_S(w, x) = w + x + 2.$$

Suppose now that we map operations $\langle S, w, x \rangle$ on processor $p = w$. If t is the current value of the logical clock, then the corresponding *space-time mapping* [18] can be inverted, and $w = p, x = t - p - 1$. If we associate a memory cell `S(` w, x `)` to each operation $\langle S, w, x \rangle$ (since we assume conversion to SAF), then the skeleton of the generated code looks like:

```
program W
do t = 0 by 1 while ( notterminated() )
  forall p = 0 to t-1
     if  executed(p, t-p-1) then
        S( p, t-p-1 ) =
          if t-p-1 >= 1
          then S( p, t-p-2 )
          else if p >= 1
               then last(p, t-p)
               else a( t-2 )
  end forall
end do
```

Predicates *terminated* and *executed* are mandatory to restore the flow of control and have been defined by Griebl and Lengauer [12, 14]. The former detects termination and the latter checks whether the current couple (t, p) corresponds to an actual operation. Both predicates have been implemented by Griebl and Lengauer by signals between asynchronous processes. However, their implementation in a synchronous model through boolean arrays is feasible, and is the subject of in-progress joint work with Martin Griebl [11].

On the other hand, function *last* dynamically restore the flow of data, and returns the value produced by the last (according to order $\ll$) executed operation among the set passed as an argument. The overhead due to this function may reduce the benefits of parallelism; however, its implementation is quite obvious: the argument set is a $\mathbb{Z}$-polyhedron that *last* has to scan in the opposite lexicographical order. A slight modification of the algorithm in [4] would generate the following code for *last*:

```
function last ( w , x )
do α = w − 1 , 0 , −1
  β = w − α − 1
  if executed(α, β) then
     return S( α, β )
end do
return a( w + x − 1 )
```

This function does implement the result of a fuzzy array dataflow analysis since the returned value is the one produced by the last *executed* possible source, or the initial element of array `a` if no possible source executed. Obviously, many optimized implementation schemes for `last` can be crafted, but discussing this issue would take us too far afield and is left for future work.

6 Related work and conclusion

When the flow of control cannot be predicted at compile time, data dependence analysis can only be imprecise. For instance, one cannot solve the array dataflow problem [23, 7, 22], which gives for every consumed value the identity of the producer operation. This lack of precision translates into sets of possible producer operations. Note that this phenomenon may occur in two other situations: 1) in the presence of intricate or dynamic ("subscripted") array subscripts, and 2) when the compiler writer believes that current precise dependence analyzes are too expensive, and that approximate tests are sufficient [5]. In all three cases, a new scheduling algorithm has to be designed. The algorithm proposed in this paper is based on [8, 9, 26].

Future work should address the tiling of iteration domains, possibly though construction of delay dependences. With such dependences, the sets of preceding operations (13) and (14) become $\{o_1 | \, o_1 \Gamma o_2 \ \vee \ o_1 \delta^R o_2 \ \vee \ o_1 \delta^{comp} o_2\}$ and $\{o_1 | \, o_1 \delta o_2 \ \vee \ o_1 \delta^R o_2 \ \vee \ o_1 \delta^{comp} o_2\}$, respectively. However, the problem is then to automatically derive pseudo-affine schedules and generate code for them [15, 16].

One should also try and answer the following questions: Is it worthwhile to convert DCPs into single-assignment form? (Obviously, extensive experiments are needed here.) When should speculative execution be brought into play [25] ? How can we reduce the number of equations and unknowns in our method? (Solving such problems thanks to softwares such as MAPLE or PIP is costly.) Could our compile-time scheduling ease the work of the inspector in the method proposed in [27]? Indeed, the efficient compilation of DCPs probably needs a tight integration of compile-time and run-time techniques [25].

Acknowledgments

We would like to thank L. Bougé, M. Griebl, C. Lengauer, B. Lisper, X. Redon and F. Vivien for many vivid, brain-storming discussions.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, Mass, 1986.

[2] J.-F. Collard. Space-time transformation of while-loops using speculative execution. In *Proc. of the 1994 Scalable High Performance Computing Conf.*, pages 429–436, Knoxville, TN, May 1994. IEEE.

[3] J.-F. Collard. Automatic parallelization of while-loops using speculative execution. *Int. J. of Parallel Programming*, 23(2):191–219, April 1995.

[4] J.-F. Collard, P. Feautrier, and T. Risset. Construction of DO loops from systems of affine constraints. *Parallel Processing Letters*, 5(3), 1995.

[5] A. Darte and F. Vivien. Automatic parallelization based on multi-dimensional scheduling. Technical report, LIP, ENS Lyon, France, 1994. To appear.

[6] P. Feautrier. Array expansion. In *ACM Int. Conf. on Supercomputing, St Malo*, pages 429–441, 1988.

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

[8] P. Feautrier. Some efficient solutions to the affine scheduling problem, part I, one dimensional time. *Int. J. of Parallel Programming*, 21(5):313–348, October 1992.

[9] P. Feautrier. Some efficient solution to the affine scheduling problem, part II, multidimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, December 1992.

[10] P. Feautrier and J.-F. Collard. Fuzzy array dataflow analysis. Technical Report RR 94-24, LIP, ENS Lyon, France, July 1994. ftp: `lip.ens-lyon.fr`.

[11] M. Griebl and J.-F. Collard. Generation of synchronous code for automatic parallelization of `while` loops. In *Euro-Par95*, Stockholm, Sweden, 1995. To appear.

[12] M. Griebl and C. Lengauer. On scanning space-time mapped while loops. In B. Buchberger, editor, *Parallel Processing: CONPAR 94 – VAPP VI*, Lecture Notes in Computer Science 854, pages 677–688, Linz, Austria, 1994. Springer-Verlag.

[13] M. Griebl and C. Lengauer. On the space-time mapping of while-loops. *Parallel Processing Letters*, 1994. To appear. Also available as Report MIP-9304, Fakultät für Mathematik und Informatik, Universität Passau, Germany.

[14] M. Griebl and C. Lengauer. On the parallelization of loop nests containing `while` loops. In N. Mirenkov, editor, *Proc. Aizu Int. Symp. on Parallel Algorithm/Architecture Synthesis (pAs'95)*, Aizu-Wakamatsu, Japan, March 1995. IEEE. To appear.

[15] W. Kelly and W. Pugh. Finding legal reordering transformations using mappings. Technical Report CS-TR-3297, Dept. of CS, U. of Maryland, June 1994.

[16] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. Technical Report CS-TR-3317, Dept. of CS, U. of Maryland, July 1994.

[17] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, Gold Coast, Australia, May 1992.

[18] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR '93*, LNCS 715, pages 398–416. Springer-Verlag, 1993.

[19] B. Lisper. Detecting static algorithms by partial evaluation. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 31–42, June 1991.

[20] M. Martel. Etude et implémentation de méthodes numériques itératives basées sur l'exécution spéculative. Master's thesis, Ecole Normale Supérieure de Lyon, 1994.

[21] V. Maslov. Lazy array data-flow dependence analysis. In *Proc. 21st Annual ACM SIGPLAN-SIGACT Symp. POPL*, pages 311–325, January 1994.

[22] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. Array dataflow analysis and its use in array privatization. In *Proc. of ACM Conf. on Principles of Programming Languages*, pages 2–15, January 1993.

[23] W. Pugh and D. Wonnacott. Eliminating false data dependences using the omega test. In *ACM SIGPLAN PLDI*, 1992.

[24] W. Pugh and D. Wonnacott. An exact method for analysis of value-based data dependences. Technical Report CS-TR-3196, U. of Maryland, December 1993.

[25] L. Rauchwerger and D. Padua. Speculative run-time parallelization of loops. Technical Report 1339, CSRD - U. of Illinois at Urbana-Champaign, March 1994.

[26] X. Redon and P. Feautrier. Scheduling reductions. In *Supercomputing '94*, Manchester, England, July 1994. ACM.

[27] J. Saltz, H. Berryman, and J. Wu. Multiprocessors and runtime compilation. *Concurrency: Practice and Experience*, 3(6):573–592, December 1991.

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