Minimal Data Dependence Abstractions for Loop Transformations (Extended version)

Yi-Qing Yang, Corinne Ancourt, François Irigoin*

Centre de Recherche en Informatique
École Nationale Supérieure des Mines de Paris

Abstract. Many abstractions of program dependences have already been proposed, such as the Dependence Distance, the Dependence Direction Vector, the Dependence Level or the Dependence Cone. These different abstractions have different precisions. The *minimal* abstraction associated to a transformation is the abstraction that contains the minimal amount of information necessary to decide when such a transformation is legal. Minimal abstractions for loop reordering and unimodular transformations are presented. As an example, the dependence cone, which approximates dependences by a convex cone of the dependence distance vectors, is the minimal abstraction for unimodular transformations. It also contains enough information for legally applying all loop reordering transformations and finding the same set of valid mono- and multi-dimensional linear schedules as the dependence distance set.

KEYWORDS: parallelizing compiler, program transformation, data dependence abstraction.

Introduction

The aim of dependence testing is to detect the existence of memory access conflicts in programs. When two statements have a conflicting access to a datum, the *dependence* establishes that the execution order of the two statements cannot be modified without possible changes to the program semantics. A transformation can be applied if dependence relations are still maintained after the transformation. Many transformations require more than simple dependent/independent information to be legally applied to a set of statements. Dependence relations are represented by finite dependence abstractions that contain the data flow dependences of the statement set.
Many dependence abstractions have already been proposed, such as the Dependence Distance [23], the Dependence Direction Vector [30], the Dependence Level [1] or the Dependence Cone [14]. These different abstractions have different precisions. They are presented in Sections 2 and 3.
Depending on their precisions, the abstractions contain either not enough or sufficient or too much information to decide if a transformation T can be legally

* E-mail: {yang,ancourt,irigoin}@cri.ensmp.fr

applied or not. Section 4 describes the conditions that must be satisfied to legally perform T and the minimal information that must contain an abstraction to be used by the legality test of T. Among the *valid* abstractions, one contains the *minimal* information necessary to decide when T is legal. Section 5 gives the minimal abstraction associated with reordering transformations such as loop reversal, loop permutation, unimodular transformations, partitioning and parallelization. This notion of minimality is interesting when the compiler/parallelizer needs an *exact* answer to the question *could T be legally applied ?* from the simplest representation of the dependences.
Section 6 shows that the dependence cone, which is a *valid* abstraction for all loop reordering transformations and the *minimal* abstraction for unimodular transformations, can be used to compute valid linear schedules for nested loops. It lets us to find the same set of valid mono- and multi-dimensional linear schedules as the dependence distance vectors.

1 Notations

The notations used in the following sections are:

- $\vec{i}^n$ is an element of the loop nest iteration set I^n;
- $\ll$ denotes the lexicographic order, while $\prec$ is the execution order;
- $S(\vec{i})$ is the $\vec{i}$-th iteration of Statement S.
- $\delta^*_{\mathcal{A}}$ is the dependence relation represented by Abstraction $\mathcal{A}$;
- $\subset$ denotes the set inclusion operator.

2 Different Abstractions

Many finite abstractions have been designed to represent infinite sets of dependences in nested loops. They describe the data dependences with more or less effective accuracy. Depending on their precision, these abstractions contain either not enough, minimal or too much information to decide if a transformation can be legally applied. The different abstractions and their precisions are illustrated by example in Figure 1.

```
 DO I = 1, n
   DO J = 1, n
S:   T(I,J) = T(3I,J+1)
```

$$\left\{ \begin{array}{lll} 1 \le i \le n & 1 \le i' \le n & 3i = i' \\ 1 \le j \le n & 1 \le j' \le n & j+1 = j' \end{array} \right.$$

Fig. 1. Program P1 and Dependence system 1

The dependence test must decide the satisfiability of the dependence system 1 in Figure 1. The equalities characterize possible data access conflicts between `T(I,J)` and `T(3I,J+1)`, while the inequalities represent the constraints on the loop indices.

2.1 Dependences between Iterations

The abstraction DI(L) is exactly the set of dependent iterations between the statement instances of a loop nest L:

$$\mathrm{DI(L)} = \{(\vec{i}, \vec{i'}) | \exists S_1, S_2 \in \mathrm{L}, S_1(\vec{i})\delta^* S_2(\vec{i'})\}$$

DI(L) is obtained by solving an exact integer linear programming system for each couple of array references belonging to the nested loops.
For Program 1, DI represents in Figure 2 all the solutions of the dependence system. A parametric version of the dependent iterations can be expressed as: $\mathrm{DI}(P1) = \{((i,j),(i',j')) \mid i' = 3i, j' = j+1, 1 \leq i \leq n, 1 \leq j \leq n\}$.

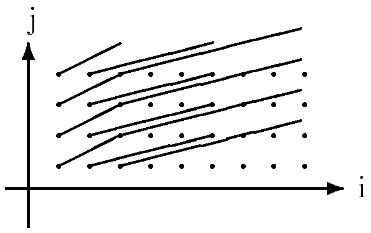

Fig. 2. DI for program 1

Since no approximation is made on the exhaustive list of dependent iterations, this abstraction cannot be used for infinite sets of dependences. Thus, parametric versions of the definition and abstractions approximating this exact set have been suggested. We quickly review the most important dependence abstractions, used for optimizing and parallelizing programs, in the following sections.

2.2 The Distance Vector

Dependence distance vectors are used to relatively characterize a set of dependent iterations:

$$\mathrm{D(L)} = \{\ \vec{d} \mid \exists(\vec{i},\ \vec{i'}) \in \mathrm{DI(L)},\ \vec{d}\ =\ \vec{i'} - \vec{i}\}$$

Taking Program 1, the dependence system 1 may be rewritten in terms of distance vectors:

$$\begin{cases} d_i = 2i & 1 \leq i \leq n & 1 \leq i + d_i \leq n \\ d_j = 1 & 1 \leq j \leq n & 1 \leq j + d_j \leq n \end{cases}$$

The dependences represented in Figure 3 are specified by:
$\mathrm{D}(P1) = \{(2k,1) \mid 1 \leq k \leq n\}$. When comparing to the representation of DI in Figure 2, note the different space: (i,j) is replaced by (d_i, d_j).
Compared to DI, the use of D reduces the amount of memory needed to store uniform dependences[2]. As an example D = $\{2\}$ represents the infinite set of

[2] also known as *constant* dependences

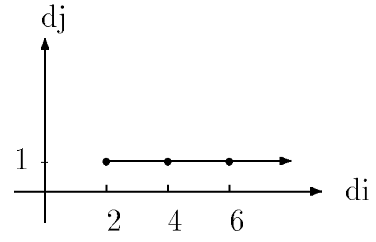

Fig. 3. D for Program 1

constant dependence iterations whose the parametric version is
DI $= \{(\vec{i},\ \vec{i'}), \vec{i'} = \vec{i} + 2, \vec{i} \geq 0\}$.
In many cases the dependence distance is not constant; therefore, abstractions approximating D like the *Dependence Polyhedron*, the *Dependence Cone*, the *Dependence Direction Vector* and the *Dependence Level* have been designed to cope with these cases.

2.3 The Dependence Polyhedron

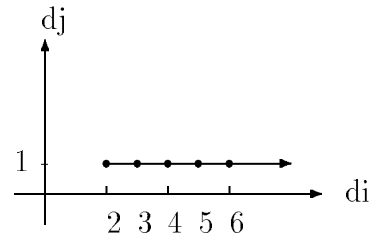

Fig. 4. DP for Program 1

The dependence polyhedron DP(L) approximates the set of distance vectors D(L) with the set of points that are convex combinations of D(L) vectors. It constrains the set of integer points of the D(L)'s convex hull.

$$\text{DP(L)} = \{\vec{v} = \sum_{1}^{k} \lambda_i \vec{d_i} \in Z^n \mid \vec{d_i} \in \text{D(L)}, \lambda_i \geq 0, \sum_{i=1}^{k} \lambda_i = 1\}$$

The polyhedron DP(L) can be concisely described by its *generating system* [26], which is a triplet made of three sets of vertices, rays and lines $(\{\vec{v_i}\}, \{\vec{r_j}\}, \{\vec{l_k}\})$. Even if DP(L) approximates D(L), it keeps all the information useful to legally apply reordering transformations as abstraction D. Figure 4 illustrates DP(L) for Program 1. The convex hull contains new vectors: $(3, 1), (5, 1)$,etc...

2.4 The Dependence Cone

The dependence cone DC(L) approximates D(L) with the set of points that are positive linear combination of D(L) vectors. It is defined as:

$$\mathrm{DC(L)} = \{\ \vec{v}\ =\ \sum_{i=1}^{k} \lambda_i \vec{d_i} \in Z^n \mid \vec{d_i} \in \mathrm{D(L)},\ \lambda_i \geq 0, \sum_{i=1}^{k} \lambda_i \geq 1\}$$

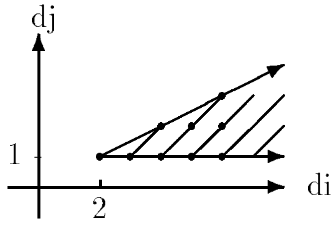

Fig. 5. DC for Program 1

The main advantage of Abstractions DP and DC is that, in many cases, only one structure (a polyhedron that is either a set of linear constraints or a generating system [26]) is necessary to the representation of information contained in D(L). The memory space gain is notable especially when the dependence set is not finite or is large with non-uniform dependences. DP and DC are easier to use than D(L) for computing loop partitioning [15].

Abstractions DP and DC can be automatically computed and accurately so when array subscript expressions are affine. Algorithms for computing DP(L) and DC(L) are described in [14, 33]. They use techniques as the simplex or the Fourier-Motzkin algorithm that have an exponential time complexity. However, as remark [24, 33], these techniques have shown a polynomial behavior in practice in this context.

2.5 The Dependence Direction Vector

Abstractions DP(L) and DC(L) represent approximated sets of D(L). These three abstractions contain the information useful for legally applying transformations that reorder iteration sets. However, transformations such as loop interchange or permutation, that only modify the sign or the order of the dependence distance vector, do not need the actual distance information. So, abstractions like DDV(L) [30] and DL(L) [1] relating only the sign or the level of dependence vectors have been suggested.

Each Dependence Direction Vector element is one of $\{<\ , =,\ >\}$. Other elements like $\leq, \geq, *$ may be used to summarize two or three DDV's. The DDV(L) abstracting the dependences for a loop nest L is defined as:

$$\mathrm{DDV} = \{\ (\psi_1, \psi_2, ..., \psi_n) \mid \exists S_1, S_2 \in \mathrm{L},\ S_1(\vec{i})\ \delta^*\ S_2(\vec{i'}), i_k\ \psi_k\ i'_k\ (1 \leq k \leq n),$$
$$\psi_i \in \{<, =, >\}\}$$

For Program 1, $\mathrm{DDV}(P1) = \{(<,\ <)\}$. It is represented in Figure 6.

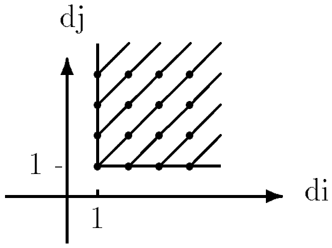

Fig. 6. DDV for Program 1

2.6 The Dependence Level

The dependence level DL has been introduced by Allen & Kennedy [1] for the vectorization and parallelization of programs. It gives the nest level of the outermost loop l that carries dependences, when the dependence vector component is positive. To maintain the program semantics, Loop l must be kept sequential. Thus, the lexico-positivity of the loop dependences are conserved and all inner loops may be parallelized if no other dependence exists. DL(L) is defined as:

$$\mathrm{DL(L)} = \{k |\ \exists(\psi_1, ..., \psi_n) \in \mathrm{DDV(L)}, 1 \leq i \leq k-1 \ \ \wedge \psi_i \ is \ = \ \wedge \ \ \psi_k \ is \ <\}$$

For Program 1, $\mathrm{DL}(P1) = \{1\}$. Dependences are represented in Figure 7.

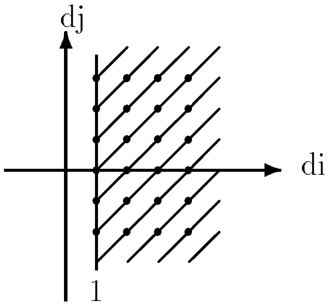

Fig. 7. DL for Program 1

3 Abstraction Precision

The six abstractions of dependences surveyed in the Section 2 have different precisions as was shown in Figures 2 to 7. Abstraction DI enumerates all dependent iterations when other abstractions express or approximate the dependence distance vectors of the loop nest.

To determine the precision of these abstractions, we define a minimal comparison set of dependent iterations $\widetilde{\mathrm{DI}}_{\mathcal{A}}$ for each abstraction $\mathcal{A}$. $\widetilde{\mathrm{DI}}_{\mathcal{A}}$ is the set of dependent iterations corresponding to the dependences represented by abstraction $\mathcal{A}$. $\widetilde{\mathrm{DI}}_{\mathcal{A}} = \{\ (\vec{i}, \vec{i} + \vec{d}) \ : \ \vec{i}\ , \vec{i} + \vec{d} \in I^n, \ \vec{d} \in \widetilde{\mathrm{D}}_{\mathcal{A}}\}$ where $\widetilde{\mathrm{D}}_{\mathcal{A}}$ is the set of dependence Distance vectors corresponding to the dependences represented by abstraction $\mathcal{A}$. To compare two ordinary abstractions $\mathcal{A}1$ and $\mathcal{A}2$, the two sets $\widetilde{\mathrm{DI}}_{\mathcal{A}1}$ and $\widetilde{\mathrm{DI}}_{\mathcal{A}2}$ are analyzed.

Definition 1
$\mathcal{A}1$ is more precise than $\mathcal{A}2$ (noted $\mathcal{A}1 \supset \mathcal{A}2$) if $\widetilde{DI}_{\mathcal{A}1} \subseteq \widetilde{DI}_{\mathcal{A}2}$.

To compare abstractions approximating D, the sets $\widetilde{\mathrm{D}}_{\mathcal{A}1}$ and $\widetilde{\mathrm{D}}_{\mathcal{A}2}$, having a lower memory cost than $\widetilde{\mathrm{DI}}$, could be used [3]. According to Definition 1, the hierarchy of abstraction precisions is: DI $\supseteq$ D $\supseteq$ DP $\supseteq$ DC $\supseteq$ DDV $\supseteq$ DL. Figures 2 to 7 illustrate this hierarchy.

4 Transformations

In order to exploit the implicit parallelism contained in programs, loop transformations are applied on program loop nests. These transformations reorder statements and iterations in order to explicit parallelism.
In the following sections, we distinguish three classes of transformations:

A program restructuring transformation reorders the statements and iterations of the program. It transforms the statement $S(\vec{i}^n)$ into a new one $S'(\vec{j}^m)$. As an example of this kind of transformation, loop distribution distributes the statements of a loop nest into different loop nests having the same iteration set as the initial code.

A reordering transformation is a particular program restructuring transformation that only reorders loop iterations without changing the order of statements in the loop nest. It is a one-to-one mapping between two iteration sets that might have different dimensions. It transforms the statement $S(\vec{i}^n)$ into a new one $S(\vec{j}^m)$ where n might be different than m. Strip mining, that reorders by partitioning the iteration set into several iteration blocks, belongs to this class of transformation.

A unimodular transformation is a particular reordering transformation that is a one-to-one mapping between two iteration sets having the same dimension. Loop interchange belongs to this class.

The conditions that must be verified, for any class of transformation, for legally applying the transformation are presented.

4.1 Legal Transformation

A transformation is *legal* if the program has the same input-output semantics after the transformation as before it. This cannot be decided in general and Definitions 2, 3, 4, 5 are based on the fact that the new execution of statement instances must maintain all the loop nest lexico-positive dependences [4]. In

[3] Proof is given in [33]

contrived cases, these definitions are too pessimistic and conclude that a transformation cannot be applied although it is semantically legal. For instance, the transposition of a *symmetric* matrix induces dependence relations that do not change the program semantics, even though they are not maintained by the transformation.

Program Restructuring Transformation

Definition 2 *Performing a restructuring transformation T on a loop nest L is* legal *and is noted* ***legal***$(\boldsymbol{T}, \boldsymbol{L})$ *if and only if: for any dependence $S_1(\vec{i}^n)\ \delta^*\ S_2(\vec{i'}^n)$ of L such that $T(S_1(\vec{i}^n)) = S'_1(\vec{j}^m)$ and $T(S_2(\vec{i'}^n)) = S'_2(\vec{j'}^m)$, the condition $S'_1(\vec{j}^m) \prec S'_2(\vec{j'}^m)$ is verified.*

Definition 2 ensures that the lexico-positivity of the dependences is maintained after transformation. When an abstraction $\mathcal{A}$ different from DI is used, Definition 3 gives the conditions for testing the legality of the transformation. The dependences are expressed here in function of $\widetilde{\mathrm{DI}}_{\mathcal{A}}$, the set of dependent iterations corresponding to the dependences of $\mathcal{A}$.

Definition 3 *Performing a restructuring transformation T on a loop nest L is* legal *according to the information provided by Abstraction $\mathcal{A}$, which is noted* ***legal***$(\boldsymbol{T}, \boldsymbol{L}, \mathcal{A})$ *if and only if: for any dependence $S_1(\vec{i}^n)\ \delta^*\ S_2(\vec{i'}^n)$ of L such that $\forall\ (\vec{i}, \vec{i'}) \in \widetilde{DI}_{\mathcal{A}}$, $T(S_1(\vec{i}^n)) = S'_1(\vec{j}^m)$ and $T(S_2(\vec{i'}^n)) = S'_2(\vec{j'}^m)$, the condition $S'_1(\vec{j}^m) \prec S'_2(\vec{j'}^m)$ is verified.*

If the application of T is legal according to the condition of Definition 3 then the condition of Definition 2 is also verified. The opposite proposition is not true.

Theorem 1 *Let T be a restructuring transformation. legal$(T, L, \mathcal{A}) \Longrightarrow$ legal(T, L)*

Proof: Assuming that T is legal according to the condition of Definition 3, then: $\forall(\vec{i}, \vec{i'}) \in \widetilde{\mathrm{DI}}_{\mathcal{A}}$ with $\mathrm{T}(S_1(\vec{i}^n)) = S'_1(\vec{j}^m)$ and $\mathrm{T}(S_2(\vec{i'}^n)) = S'_2(\vec{j'}^m)$, the condition $S'_1(\vec{j}^m) \prec S'_2(\vec{j'}^m)$ is verified. Because DI is the most precise dependence abstraction, we have $\mathrm{DI} \subseteq \widetilde{\mathrm{DI}}_{\mathcal{A}}$ that means that: $\forall(\vec{i}, \vec{i'}) \in \mathrm{DI} \Longrightarrow (\vec{i}, \vec{i'}) \in \widetilde{\mathrm{DI}}_{\mathcal{A}}$. It implies that $\forall(\vec{i}, \vec{i'}) \in \mathrm{DI}$ such that $\mathrm{T}(S_1(\vec{i}^n)) = S'_1(\vec{j}^m)$ and
$\mathrm{T}(S_2(\vec{i'}^n)) = S'_2(\vec{j'}^m)$, the condition $S'_1(\vec{j}^m) \prec S'_2(\vec{j'}^m)$ is also verified. $\square$

Reordering Transformation Reordering transformations only reorder the iteration set of a loop nest. So, the dependences $S_1(\vec{i}^n)\ \delta^*\ S_2(\vec{i'}^n)$ between references S_1 and S_2 correspond to dependences $\vec{i}^n\ \delta^*\ \vec{i'}^n$ between iterations. Such dependences can be simply represented using dependent iterations $\vec{i} \prec_\delta \vec{i'}$. All the dependences between the different statements of L can be characterized by a single set DI(L) defined as:
$\mathrm{DI(L)} = \{(i_1, i_2) | i_1 \prec_\delta i_2 \Leftrightarrow \exists S_1(i_1)\ \delta^*\ S_2(i_2) \wedge i_1 \neq i_2\}$
The legality condition for a reordering transformation, translating the preservation of the lexico-positivity constraints, follows:

Definition 4 *Performing a reordering transformation T on a loop nest L is* legal, *and is noted* ***legal(T, L)****, if and only if:* $\forall\, (\vec{i}^n, \vec{i'}^n) \in DI(L)$ *such that* $T(\vec{i}^n) = \vec{j}^m$ *and* $T(\vec{i'}^n) = \vec{j'}^m$ *the condition* $\vec{j}^m \prec \vec{j'}^m$ *is verified.*

As for restructuring transformations, when an abstraction $\mathcal{A}$ different from DI is used, a definition using dependences in function of $\widetilde{\mathrm{DI}}_{\mathcal{A}}$ is derived from Definition 4.

Unimodular Transformation A unimodular transformation [3, 29, 20] is a one-to-one mapping between two iteration sets I^n and I'^n having the same dimension. It corresponds to a unimodular change of basis M with
$\vec{I'} = \mathrm{M} \times \vec{I}$. The test of legality is: if $\vec{i_1}\delta^*\vec{i_2}$, then $i'_1 \ll i'_2$ must be verified after the transformation defined as $i'_1 = \mathrm{M} \times i_1$ and $i'_2 = \mathrm{M} \times i_2$. This condition can be expressed in terms of dependence distance vectors as: $\vec{d'} \gg 0$ must be verified after transformation, with $\vec{d} = i_2 - i_1$, $\vec{d'} = i'_2 - i'_1$ and $\vec{d'} = \mathrm{M} \times \vec{d}$.
Because the unimodular transformations reorder iteration sets only, a single set: $\mathrm{D(L)} = \{\vec{d} \mid \vec{d} = \vec{i_2} - \vec{i_1},\ \vec{i_1}\delta^*\vec{i_2} \wedge \vec{i_1} \neq \vec{i_2}\}$ can be used to characterize all loop nest dependences. The following definitions and theorem describe the legality tests for unimodular transformations:

Definition 5 *Performing a unimodular transformation T on a loop nest L is legal if and only if:* $\forall \vec{d}^n \in D(L)$ *such that* $T(\vec{d}^n) = \vec{d'}^n$*, the condition* $\vec{d'}^n \gg 0$ *is verified.*

As for reordering transformations, when an abstraction $\mathcal{A}$ different from D is used, a definition using $\widetilde{\mathrm{D}}_{\mathcal{A}}$ is derived from Definition 5.

4.2 Valid and Minimal Abstraction

The tests of legality for restructuring, reordering and unimodular transformations have been introduced in Section 4.1. These tests use the abstractions that are derived from the dependent iteration set [4] or from the dependence distance vector set [5] to verify the validity of a transformation. However, the information contained in these abstractions is often too rich and other abstractions less precise and having a lower memory cost could be used.
We define a *valid* abstraction associated with a transformation T as an approximate abstraction containing enough information to decide the legality of T. An abstraction $\mathcal{A}$ is not valid for a transformation T if the legality tests based on $\mathcal{A}$ conclude that T is not legal when it is legal according to Abstraction $\widetilde{\mathrm{DI}}_{\mathcal{A}}$. However, note that $\mathcal{A}$ may be valid for a particular form of T. Although DDV is not valid for applying any general form of unimodular transformations, it is valid for applying permutations (see Sections 5.2 and 5.3). Another trivial example

[4] DI or $\widetilde{\mathrm{DI}}_{\mathcal{A}}$ when $\mathcal{A}$ is different from DI

[5] D or $\widetilde{\mathrm{D}}_{\mathcal{A}}$ when $\mathcal{A}$ is different from D

is Identity which represents a particular case of transformations for which any abstraction is valid since it can always be applied legally .
Among the abstractions, DI is the most precise. So, if an abstraction $\mathcal{A}$ (different from DI) contains the same information as DI to decide that the transformation T is legal, then this abstraction $\mathcal{A}$ is *valid* for T.

Definition 6 *Let $\mathcal{A}$ be an abstraction less precise than DI and T a restructuring transformation. If for any loop nest L, legal(T, L, DI) $\Longrightarrow$ legal($T, L, \mathcal{A}$), then the abstraction $\mathcal{A}$ is* valid *for testing the legality of the transformation T.*

Several valid abstractions associated with a transformation may exist. In fact, all abstractions more precise than a valid abstraction are also valid.

Theorem 2 *Let $\mathcal{A}$ be a valid abstraction for testing the legality of a transformation T. Any abstraction $\mathcal{A}_i$ such that $\mathcal{A}_i \supseteq \mathcal{A}$ is also a valid abstraction for testing the legality of T.*

Proof: We know that $\mathcal{A}$ is a valid abstraction for T(L) and that $\mathcal{A}_i \supseteq \mathcal{A}$. Then, in accordance with Definition 6: legal(T, L, DI) $\Longrightarrow$ legal(T, L, $\mathcal{A}$).
Definition 3 gives the equivalence: legal(T, L, $\mathcal{A}$) $\Longleftrightarrow$ [$\forall(\vec{i}, \vec{i'}) \in \widetilde{\mathrm{DI}}_{\mathcal{A}}$ with $\mathrm{T}(S_1(\vec{i}^n)) = S'_1(\vec{j}^m)$ and $\mathrm{T}(S_2(\vec{i'}^n)) = S'_2(\vec{j'}^m)$ then $S'_1(\vec{j}^m) \prec S'_2(\vec{j'}^m)$]. By hypothesis $\mathcal{A}_i \supseteq \mathcal{A}$, so $\forall\, (\vec{i}, \vec{i'}) \in \widetilde{\mathrm{DI}}_{\mathcal{A}_i}$ then $(\vec{i}, \vec{i'}) \in \widetilde{\mathrm{DI}}_{\mathcal{A}}$. By combining this hypothesis with the previous equivalence, the following assertion is verified: $\forall\, (\vec{i}, \vec{i'}) \in \widetilde{\mathrm{DI}}_{\mathcal{A}_i}$ with $\mathrm{T}(S_1(\vec{i}^n)) = S'_1(\vec{j}^m)$ and $\mathrm{T}(S_2(\vec{i'}^n)) = S'_2(\vec{j'}^m)$, then $S'_1(\vec{j}^m) \prec S'_2(\vec{j'}^m)$. That is equivalent to: legal(T, L, $\mathcal{A}$) $\Longrightarrow$ legal(T, L, $\mathcal{A}_i$), and implies legal(T, L, DI) $\Longrightarrow$ legal(T, L, $\mathcal{A}_i$). $\square$
Among the valid abstractions associated with a transformation, there is one (sometimes several) that contains the *minimal* information necessary to decide whether the transformation is legal. This abstraction is called *minimal* and is defined by the following Definition:

Definition 7 *Let $\mathcal{A}1$ be a valid abstraction for the transformation T and $\mathcal{A}2$ be another abstraction such that $\mathcal{A}1 \supseteq \mathcal{A}2$ and $\neg\exists\mathcal{A}3 \,/\, \mathcal{A}1 \supseteq \mathcal{A}3 \supseteq \mathcal{A}2$. If $\exists L$ such that legal($T, L, \mathcal{A}1$) $\wedge$ $\neg$(legal($T, L, \mathcal{A}2$)), then abstraction $\mathcal{A}1$ is* minimal *for testing the legality of the transformation T.*

The minimality relation is not a total order since it is based on set inclusion and several abstractions may be minimal for a transformation. An abstraction $\mathcal{A}$ is defined minimal for the general form of the transformation T. Another abstraction, less precise than $\mathcal{A}$, may be minimal for a particular form of T. As an example DC is minimal for unimodular transformations while DDV is minimal for loop permutations.
Minimality is relative to a particular set of abstractions. The minimal abstractions associated with the reordering transformations that are presented in the next section are minimal for the set of six abstractions DI, D, DP, DC, DDV and DL.

To broaden this study to another abstraction, the new abstraction $\mathcal{A}$ must be first classified in the precision hierarchy of the abstractions. Let's assume that $\mathrm{D}_1 \supseteq \mathcal{A} \supseteq \mathrm{D}_2$ where D_1 and D_2 are one of the previously studied abstractions. Let T_1, T_2 and T_3 be the sets of transformations for which, respectively, (1) Abstraction D_1 is not valid (2) Abstraction D_2 is valid (3) Abstraction D_1 is valid when Abstraction D_2 is not valid. Then the characteristics of $\mathcal{A}$ are the following ones:

- $\mathcal{A}$ is not valid for each transformation of T_1
- $\mathcal{A}$ is valid for each transformation of T_2
- if $\mathcal{A}$ is valid according to Definition 7 for a transformation T of T_3 then it is minimal for T de facto.

Other criteria on abstractions such as the complexity of the computation and resolution times might be taken into account to select the *best* abstraction associated with a transformation. The computation time complexity associated with an abstraction depends on the time complexity of the dependence test used to calculate the dependence systems as well as on the complexity of the transformation legality test. It also depends on the practical distribution of dependences in real applications. Thus a comparison between abstraction computation times is extremely difficult. It may be based on the time complexity of the main operator manipulating the abstractions, which is, the union. We note that Abstractions DI (in its extended version), D (when only constant dependences occurred), DDV and DL manipulate lists of dependences while DI (parametric version), D (in actual applications), DP and DC use polyhedric representations. It is obvious that the union of several dependences is cheaper on lists than on polyhedrons.
The satisfiability test on individual dependence sets should be executed whatever the abstraction is. In order to obtain good accuracy, many satisfiability tests combine several simple tests. Tests having polynomial complexity and quickly solving particular cases are applied first. Accurate tests dealing with integer linear system satisfiability are applied last [22, 24, 33]. The individual tests are combined in order to find as soon as possible whether the dependence system does not have a solution. Experiments made on the Perfect Club programs [33] show that the simple test checking the equality of (numerical and symbolic) constant terms in the equations of dependence systems has a very low cost. It still detects that 76% of the dependence systems have no solution. Associated to other simple tests, these experiments conclude that expensive tests in a dependence testing context are not often executed on real applications. Experiments carried out with "expensive" dependence tests and dependence representations showed that their relative impact on the compiler speed is small [12].

5 Minimal Abstraction and Transformation

Dependence abstractions and loop transformations are linked. The dependences are used to test the legality of a transformation when it changes the program dependences. Because the abstractions have different precisions, a transformation may be declared illegal or legal depending on the abstraction which is used. Thus, choosing a valid abstraction associated with a transformation is important for programs.

Among the valid abstractions associated with a transformation, one (sometimes several) is the *minimal* abstraction. It contains the minimal information necessary to decide whether the transformation can be applied legally or not. This section focuses on the minimal abstraction associated with the following transformations: loop reversal, loop permutation, unimodular transformation, partitioning and parallelization.

For each transformation T, a section presents: (1) the effect of T on the dependences, (2) the minimal abstraction, among those presented in this paper, associated with T and (3) the legality test applied to T. The theoretical proofs of minimality of abstractions associated with these transformations are given in [33]. Each proof proceeds in two steps: (1) proves that the abstraction is valid for the transformation and (2) according to Definition 7, gives a counter example which shows that an abstraction less precise than the minimal abstraction does not contain enough information to decide the transformation legality. The first step of the proof of minimality for the loop reversal transformation is presented in the next section. The other loop transformation proofs follow the same scheme and are given in [33]. A counter example illustrating the second step is presented for each transformation.

5.1 Loop Reversal

A loop reversal transformation $\mathrm{Inv}_k\ (l_1, l_2, ..., l_n)$ applied on a n-dimensional loop nest reverses the execution order of loop l_k. The effect of the transformation on the dependence distance vector $\vec{d} = (d_1, .., d_k, .., d_n)$ is
$\mathrm{Inv}\ _k\ (\vec{d}) = (d_1, ..., -d_k, ..., d_n)$. So, the lexico-positivity of the dependences is maintained after transformation if the following condition is verified:
$\mathrm{legal}(\mathrm{Inv}_K, \mathrm{L}) \Longleftrightarrow \forall \vec{d} \in \mathrm{D(L)}, (d_1, ..., -d_k, ..., d_n) \gg 0.$

Theorem 3 *The minimal abstraction for the loop reversal transformation is the Dependence Level DL.*

DL is valid for the loop reversal transformation: Let's consider a loop nest L where Loop k can be legally reversed and $\mathrm{legal}(\mathrm{Inv}_k, \mathrm{L}, \mathrm{DL})$ is false. That means $\exists \vec{d} \in \widetilde{\mathrm{D}}_{\mathrm{DL}} | \vec{d} = (0, .., 0, d_k, *, ..), d_k > 0, \mathrm{Inv}_k(\vec{d}) = (0, .., 0, -d_k, *, ..) \ll 0$[6]
$\mathrm{Inv}_k\ (\vec{d}) \ll 0$ implies that $\mathrm{legal}(\mathrm{Inv}_k, \mathrm{L}, \mathrm{D})$ is false. Since according to Definition 5, Abstraction D is valid for any unimodular loop transformation and since $\mathrm{D} \supset \mathrm{DL}$ there is a contradiction. □

[6] The first element of a distance vector $\vec{d} \ll 0$, different from 0, is negative

```
DO I = 2, n
  DO J = 2, n
    DO K = 2,n
      A(I,J,K) = A(I-1, J,2K) + A(I,J-1,2K)
    ENDDO
  ENDDO
ENDDO
```

Fig. 8. Program 2

An example illustrating the previous theorem: In Program 2, the loop nest has two data flow dependence relations that are expressed with the following dependence abstractions:

- $\mathrm{D} = \{(1, 0, -k), (0, 1, -k) \mid 2 \leq k \leq n\}$
- $\mathrm{DDV} = \{(<, =, >), (=, <, >)\}$
- $\mathrm{DL} = \{1, 2\}$

Abstraction DL asserts that the 3-rd loop can be reversed because the constraints on the lexico-positive dependences are still maintained after reversing K : $3 \notin \mathrm{DL} \Longrightarrow \mathrm{DL} = \{1, 2\}$ after transformation. Now, imagine a dependence abstraction $\mathcal{A}$ less precise than DL and giving only general dependent/independent information on the set of statements on which the transformation must be applied. Then, not enough information is contained in $\mathcal{A}$ to verify if reversing the 3-rd loop is legal or not. When in doubt, the loop reversal is declared illegal. Using DL, loop K can be reversed. The new loop nest is figured in Program 3.
All abstractions DI, D, DC, DP, DDV and DL are *valid* for a loop reversal transformation. The test of legality associated with the minimal abstraction DL is: $\mathrm{legal}(\mathrm{Inv}_k, \mathrm{L}) \Longleftrightarrow \notin \mathrm{DL}(\mathrm{L}) \Longleftrightarrow \mathrm{projection}(\mathrm{DL}(\mathrm{L}), k) = \emptyset$.

5.2 Loop Permutation

A loop permutation transformation [2] Perm_P (L) performs permutation P on the n-dimensional iteration set of the loop nest $\mathrm{L} = (l_1, l_2, .., l_n)$. The new loop nest L′ is defined by $\mathrm{L}' = (l_{p[1]}, l_{p[2]}, ..., l_{p[n]})$ where $\{p[1], p[2], .., p[n]\}$ is a permutation of $\{1, 2, ..n\}$. The effect of the transformation on the dependence distance vector $\vec{d} = (d_1, .., d_k, .., d_n)$ is: $\vec{d'} = \mathrm{Perm}_P\ (\vec{d}) = (d_{p[1]}, ..., d_{p[k]}, ..., d_{p[n]})$. So, the lexico-positivity of the dependences is maintained after transformation if the following condition is verified:
$\mathrm{legal}(\mathrm{Perm}\ _P,\ \mathrm{L}) \iff \forall \vec{d} \in \mathrm{D}(\mathrm{L}),\ (d_{p[1]}, ..., d_{p[k]}, ..., d_{p[n]}) \gg 0$

Theorem 4 *For loop permutations, the minimal abstraction is the Dependence Direction Vector DDV.*

```
DO I = 2, n
  DO J = 2, n
    DO K = 2,n,-1
      A(I,J,K) = A(I-1, J,2K) + A(I,J-1,2K)
    ENDDO
  ENDDO
ENDDO
```

Fig. 9. Program 3

An example illustrating the previous assertion: The dependences of Program 3 in Figure 9 can be represented using the D, DDV and DL abstractions:

- $\mathrm{D} = \{(1, 0, k), (0, 1, k) \mid 2 \leq k \leq n\}$
- $\mathrm{DDV} = \{(<, =, <), (=, <, <)\}$
- $\mathrm{DL} = \{1, 2\}$

The use of abstraction DDV enables us to check that the permutation of the (I, J, K) loops into (K, I, J) loops is legal, because :
Perm $_P((<, =, <), (=, <, <)) = \{(<, <, =), (<, =, <)\}$. This is $\gg 0$, since the first element, different from "=", is $<$ which enforces the lexico-positivity of the dependences. In contrast, the use of DL [7] does not allow for concluding the legality of this permutation, because no dependence information is known on the 3-rd loop which will be the outmost loop after permutation.
All abstractions DI, D, DC, DP and DDV are *valid* for a loop permutation transformation. The test of legality associated with the minimal abstraction is then: $\mathrm{legal}(\mathrm{Perm}_P, \mathrm{L}) \iff \forall \overrightarrow{ddv} \in \mathrm{DDV}(\mathrm{L}), \mathrm{Perm}_P(\overrightarrow{ddv}) \gg 0$.

5.3 Unimodular Transformation

A unimodular transformation TU_{M} (L) performs a unimodular change of basis M on the iteration set I of the loop nest L. After transformation, the iteration set becomes $I' = \mathrm{M} \times I$. The determinant of M is equal to 1 or -1.
The effect of the transformation on the dependence distance vector $\vec{d}$ is
$\vec{d'} = \mathrm{TU}_{\mathrm{M}}(\vec{d}) = \mathrm{M} \times \vec{d}$. In order to maintain the lexico-positivity constraints of the dependences, the following condition must be verified:
$\mathrm{legal}(\mathrm{TU}_{\mathrm{M}}, \mathrm{L}) \iff \forall \vec{d} \in \mathrm{D}(\mathrm{L}),\ \mathrm{M} \times \vec{d} \gg 0$

[7] DL is the abstraction just under DDV in the abstraction precision order

Theorem 5 *The minimal abstraction for a unimodular transformation is the Dependence Cone DC.*

```
DO I = 1, n
  DO J = 1, n
     A(I,J) = A(I, 2J) + A(2I, J-I+1)
  ENDDO
ENDDO
```

Fig. 10. Program 4

An example illustrating the previous assertion: The program 4 loop nest contains two dependence relations. The first one characterizes data access conflicts between the two references $A(I,2J)$ and $A(I,J)$. The corresponding dependence system is $S = \{di = 0, dj = j, 1 \leq j \leq n\}$.
Abstractions D, DC and DDV represent these dependences in the following manner:

- $\mathrm{D}(dp1) = \{(0,x) \mid 1 \leq x\}$
- $\mathrm{DC}(dp1) = (\{(0,1)\}, \{(0,1)\}, \emptyset)$
- $\mathrm{DDV}(dp1) = \{(=,<)\}$

The second dependence relation represents data access conflicts between two references $A(2I, J-I+1)$ and $A(I,J)$. The corresponding dependence system is $S = \{di = i, di + dj = 1, 1 \leq i \leq n\}$.
Abstractions D, DC and DDV represent these dependences as:

- $\mathrm{D}(dp2) = \{(x, -x+1)\} \mid 1 \leq x\}$
- $\mathrm{DC}(dp2) = (\{(1,0)\}, \{(1,0),(1,-1)\}, \emptyset)$
- $\mathrm{DDV}(dp2) = \{(<,>)\}$

Before testing the legality of the transformation, the union of dependences must be computed. The definitions of union for the different abstractions are given in [14, 32, 33]. Using the same abstractions, this union is expressed as:

- $\mathrm{D(L)} = \mathrm{D}(dp1) \cup \mathrm{D}(dp2) = \{(0,x),(x,-x+1) \mid 1 \leq x\}$.
- $\mathrm{DC(L)} = \text{convex\_hull}(\mathrm{DC}(dp1) \cup \mathrm{DC}(dp2)) = (\{(0,1)\}, \{(0,1),(1,-1)\}, \emptyset)$.

Figure 11 illustrates these dependences.
Let's assume that we would like to know if the unimodular change of basis M is legal or not, with :

$$\mathrm{M} = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}.$$

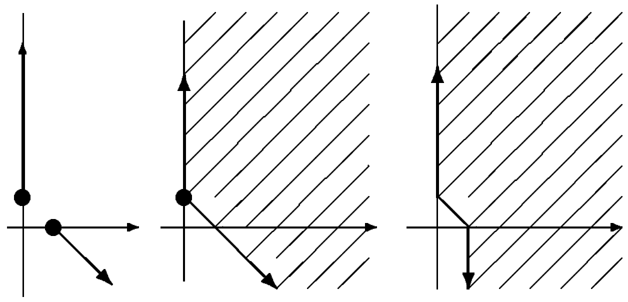

Fig. 11. D(L), DC(L) and DDV(L)

The computation of the lexico-positivity constraints having to be maintained amounts to:

- $\mathrm{M} \times \mathrm{D}(dp1) = (x, 0)$ and $\mathrm{M} \times \mathrm{D}(dp2) = (1, x)$, which are $\gg 0$
- $\mathrm{M} \times \mathrm{DC(L)} = (\{(1,1)\}, \{(1,1),(0,1)\}, \emptyset)$, which is $\gg 0$
- $\mathrm{M} \times \mathrm{DDV(L)} = \mathrm{M} \times \mathrm{DDV}(dp1) \cup \mathrm{M} \times \mathrm{DDV}(dp2)$

As we can see, abstractions D and DC let us decide that the unimodular transformation can be applied legally, while the DDV abstraction cannot conclude since $\mathrm{M} \times \mathrm{DDV}(dp2) = (< + >, <)$ is not necessary lexico-positive.
All abstractions DI, D, DP and DC are *valid* for a unimodular transformation. The test of legality associated with the minimal abstraction is:
$\mathrm{legal}(\mathrm{TU}_{\mathrm{M}}, \mathrm{L}) \iff \wedge_{1 \leq i \leq k}(\mathrm{M} \times \mathrm{DC}_i \gg 0)$ with $\mathrm{DC(L)} = (\cup_{1 \leq i \leq k} \mathrm{DC}_i)$.

5.4 Partitioning

A partitioning transformation [18, 19, 15] $\mathrm{Part}_{\mathrm{H}}$ applied on an n-dimensional loop nest splits the iteration space into blocks of some regular shape by using p families of parallel hyperplanes. The shape and size of the partitioned blocks are defined by the partitioning vector $\mathrm{H} = (\vec{h_1}, \vec{h_2}, ..., \vec{h_p})$, where the p families of hyperplanes are respectively orthogonal to $\vec{h_i}$ vectors. This transforms the iteration space I^n into a new one I^{p+n}.
Two iterations i_1 and i_2 of the initial iteration space belong to the same partitioned block if [15]:
$(\lfloor \vec{h_1} \times \vec{i_1} \rfloor, \lfloor \vec{h_2} \times \vec{i_1} \rfloor, ..., \lfloor \vec{h_n} \times \vec{i_1} \rfloor) = (\lfloor \vec{h_1} \times \vec{i_2} \rfloor, \lfloor \vec{h_2} \times \vec{i_2} \rfloor, ..., \lfloor \vec{h_n} \times \vec{i_2} \rfloor)$
According to the results of [15], performing a partitioning H is legal if the following condition is fulfilled:

$$(1) \quad \forall \vec{d} \in \mathrm{D(L)} \;\; \mathrm{H} \cdot \vec{d} \geq \vec{0}$$

For an abstraction $\mathcal{A}$ different from D, the polyhedron characterizing the dependences represented in $\mathcal{A}$ is noted $P_{\mathcal{A}}$. $P_{\mathcal{A}}$ is described by its generating system: $(\{\vec{v_i}\}, \{\vec{r_j}\}, \{\vec{l_k}\})$. Condition (1) can be expressed in terms of $\mathcal{A}$ in the following way:

$$\mathrm{H} \cdot P_{\mathcal{A}} \geq \vec{0} \Longrightarrow \ \mathrm{legal}(\mathrm{Part}_{\mathrm{H}}, \mathrm{L}, \mathcal{A})$$

where $\mathrm{H} \cdot P_{\mathcal{A}} \geq \vec{0}$ is equivalent to:

- $\forall v_i \in P_{\mathcal{A}}(\mathrm{L}),\ \mathrm{H} \cdot v_i \geq \vec{0}$,
- $\forall r_j \in P_{\mathcal{A}}(\mathrm{L}),\ \mathrm{H} \cdot r_j \geq \vec{0}$,
- $\forall l_k \in P_{\mathcal{A}}(\mathrm{L}),\ \mathrm{H} \cdot l_k = \vec{0}$,

Theorem 6 *The minimal valid abstraction associated with a partitioning transformation is DC.*

```
DO I = 1, n
  DO J = 1, n
    A(I,J) = A(2I,J+1) + A(2I, I+J)
  ENDDO
ENDDO
```

Fig. 12. Program 5

An example illustrating the previous assertion: Program 5 contains two anti-dependences. The first one characterizes data access conflicts between the two references $A(2I, J+1)$ and $A(I, J)$. Its dependence system is
$S = \{di = i, dj = 1, 1 \leq i \leq n\}$. The dependence can be represented using D, DC and DDV:

- $\mathrm{D}(dep1) = \{(x, 1) \mid 1 \leq x\}$
- $\mathrm{DC}(dep1) = (\{(1, 1)\}, \{(1, 0), (1, 1)\}, \emptyset)$
- $\mathrm{DDV}(dep1) = (<, <)$

The second anti-dependence characterizes data access conflicts between the two references $A(2I, I+J)$ and $A(I, J)$.
Its dependence system is $S = \{di = dj, di \geq 1\}$. Its representations by D, DC and DDV is:

- $\mathrm{D}(dep2) = \{(x, x) \mid 1 \leq x\}$
- $\mathrm{DC}(dep2) = (\{(1, 1)\}, \{(1, 1)\}, \emptyset)$
- $\mathrm{DDV}(dep2) = (<, <)$

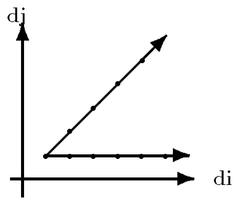

Fig. 13. D for Program 5

The union of the two dependences is expressed as:
• $\mathrm{D(L)} = \mathrm{D}(dep1) \cup \mathrm{D}(dep2) = \{(x,1),\ (x,x) \mid 1 \leq x\}$. D(L) is illustrated by Figure 13.
• $\mathrm{DC(L)} = \mathrm{convex\_hull}(\mathrm{DC}(dep1) \cup \mathrm{DC}(dep2)) = (\{(1,1)\}, \{(1,0),(1,1)\}, \emptyset)$.

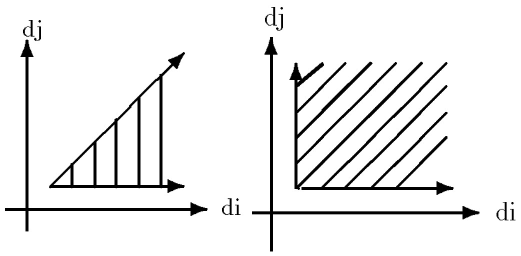

Fig. 14. DC and DDV for Program 5

• $\mathrm{DDV(L)} = \mathrm{DDV}(dep1) \cup \mathrm{DDV}(dep2) = (<,<)$.
DC(L) and DDV(L) are illustrated by Figure 14.
Let's assume that we would like to partition the iteration space according to the partitioning vector $\mathrm{H} = (h_1, h_2)$ where $h_1 = (\frac{1}{3}, -\frac{1}{3})$ and $h_2 = (\frac{1}{3}, 0)$. The lexico-positivity constraints for abstractions D, DC and DDV are expressed as follows:

- $\mathrm{H} \cdot \mathrm{D}(dep1) = (\frac{x-1}{3}, \frac{x}{3}\ )$ and $\mathrm{H} \cdot \mathrm{D}(dep2) = (0, \frac{x}{3})$. Since $x \geq 1 \Longrightarrow$ $\mathrm{H} \cdot \mathrm{D}(dep1) \geq \vec{0}$ and $\mathrm{H} \cdot \mathrm{D}(dep2) \geq \vec{0}$. $\mathrm{Part}_{\mathrm{H}}\mathrm{D(L)}$ is legal with respect to D.
- $\mathrm{H} \cdot \mathrm{DC(L)} = (\ \{(0, \frac{1}{3})\}, \{(\frac{1}{3}, \frac{1}{3}), (0, \frac{1}{3})\}, \emptyset)$. Since $s_i, r_j \in \mathrm{H} \cdot \mathrm{DC(L)} \geq \vec{0}$ and $d_k = \emptyset$, we have $\mathrm{H} \cdot \mathrm{DC(L)} \geq \vec{0}$. $\mathrm{Part}_{\mathrm{H}}\mathrm{DC(L)}$ is legal with respect to DC.
- Since $\mathrm{DDV(L)} = \{\ (<,<)\ \}$, $P(\mathrm{DDV(L)}) = (\ \{(1,1)\}, \{(1,0),(0,1)\}, \emptyset)$ and $\mathrm{H} \cdot \mathrm{DDV(L)} = (\ \{(0, \frac{1}{3})\}, \{(\frac{1}{3}, \frac{1}{3}), (-\frac{1}{3}, 0)\}, \emptyset)$. Since $s_1 = (0, \frac{1}{3}) \geq \vec{0}$ and $r_2 = (-\frac{1}{3}, 0) \leq \vec{0} \implies \neg(\mathrm{H} \cdot \mathrm{DDV(L)} \leq \vec{0}) \wedge \neg(\mathrm{H} \cdot \mathrm{DDV(L)} \geq \vec{0})$, $\mathrm{Part}_{\mathrm{H}}\mathrm{DDV(L)}$ is illegal.

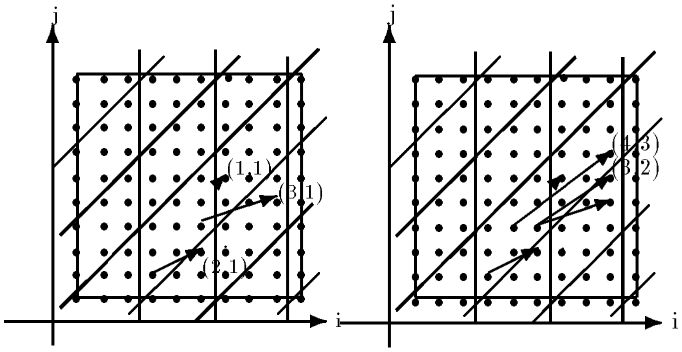

Fig. 15. Partitioning H with D, DC

Figures 15 and 16 show the H partitioning associated with the D, DC and DDV dependences. We can see that with D there is no dependence cycles between partitioned blocks. Note that the additional dependence distance vector belonging to DC does not introduce additional dependence relations between partitioned blocks. But two dependences represented by DDV(L): (1,3) and (2,1) give a cycle between two blocks. Then, $\mathrm{Part}_{\mathrm{H}}\mathrm{DDV}(\mathrm{L})$ is illegal.

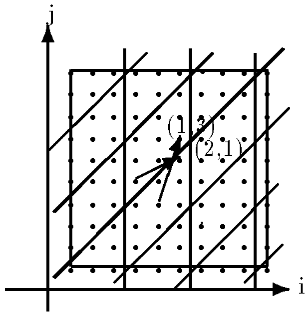

Fig. 16. Partitioning H with DDV

DC contains sufficient information to find that $\mathrm{Part}_{\mathrm{H}}$ is legal, because condition (1) is met. On the contrary, abstraction DDV won't find legality because the set of points described by: $\{s_1 = (0, \frac{1}{3}) \geq \vec{0} \wedge r_2 = (-\frac{1}{3}, 0) \leq \vec{0}\}$ does not meet (1).
In conclusion, all abstractions DI, D, DP and DC are *valid* for a loop partitioning transformation. The test of legality associated with the minimal abstraction is:
$(\wedge_{1 \leq i \leq k}(\mathrm{H} \cdot \mathrm{DC}_i \geq \vec{0})) \vee (\wedge_{1 \leq i \leq k}(\mathrm{H} \cdot \mathrm{DC}_i \leq \vec{0})) \Longrightarrow \mathrm{legal}(\mathrm{Part}_{\mathrm{H}}, \mathrm{L})$

In using DC, Irigoin and Triolet detail in [15] the algorithm for computing hyperplane partitioning, which is a generalization of strip-mining with no restriction on the stripping direction.

5.5 Parallelization

Performing a parallelization Paral on a loop nest L along the parallelizing vector $\overrightarrow{pv}$ transforms each loop i with $\overrightarrow{pv}(i) = 0$ in a parallel loop.
$\overrightarrow{pv}(k) = \begin{cases} 0\, the\ k-th\ loop\ is\ parallel \\ 1\, the\ k-th\ loop\ is\ sequential \end{cases}$
The effect of the parallelization on $\vec{d} = (d_1, .., d_k, .., d_n)$ is projection$(\vec{d}, \overrightarrow{pv}) = (d_{i_1}, d_{i_2}, ...d_{i_m})$ where i_l are the sequential loop indices. As the loop reversal transformation, the parallelization modifies the iteration execution order only in internal loop levels. Thus to maintain the lexico-positivity constraints, the loop levels being parallelized must not affect the initial lexico-positivity dependences. The following condition have to be verified to legally apply a parallelization : legal(Paral($\overrightarrow{pv}$), L) $\iff \forall \vec{d} \in$ D(L) such that $d_k = 0\ (1 \leq k \leq j-1) \wedge$
$d_j > 0 \wedge$ projection$(\vec{d}, \overrightarrow{pv}) = (d_{i_1}, d_{i_2}, ...d_{i_m})$ then $d_j \in \{d_{i_1}, d_{i_2}, ...d_{i_m}\}$.
To legally parallelize a loop nest, the minimal abstraction is DL.
An example illustrating the previous assertion:

```
DO I = 1, n
  DO J = 1, n
     A(I,J) = A(I-1,J-1)
  ENDDO
ENDDO
```

Fig. 17. Program 6

Taking Program 6, the dependences can be represented using D and DL:
D(L) $= \{(1,1)\}$ and DL(L) $= \{1\}$
Let's assume that we would like to know if the parallelization of the 2-nd loop is legal. Then $\overrightarrow{pv}$ is equal to $(1,0)$. The lexico-positivity constraint is expressed as: projection$(\{1\}, (1,0)) = \{1\}$
DL enables us to conclude that the parallelization of loop J is legal because the lexico-positivity constraints have been maintained after parallelization. An abstraction less precise giving, for example, only general dependent/independent information on the set of statements would not contain enough information for the legality test.
All abstractions DI, D, DC, DP, DDV and DL are *valid* for a loop parallelization transformation. Since the parallelization modifies the iteration execution order only in internal loop levels, the lexico-positivity constraints are maintained if the parallelized loop levels do not appear in the dependence levels. So the test

of legality associated with the minimal abstraction is:
$\mathrm{legal}(\mathrm{Paral}(\overrightarrow{pv}), \mathrm{L}) \Longleftarrow \mathrm{projection}(\mathrm{DL}(\mathrm{L}), \vec{1} - \overrightarrow{pv}) = \emptyset$

6 Linear Scheduling

Many parallelization methods such as linear scheduling have been suggested. A linear schedule defines a new execution order for a loop nest iteration domain such that the first n scanning directions carry the loop nest dependences while others are parallel directions. The n-th first loops of the transformed loop nest are sequential and the innermost loops are parallel.
The linear schedule is described by a *linear scheduling vector* (one-dimensional) or a *linear scheduling matrix* (multi-dimensional). The mapping from the multi-dimensional iteration set to the dimensional time space is defined by a loop transformation that involves the multiplication of the linear scheduling vector/matrix and the iteration set. The dependence abstraction is used for computing a *valid* linear schedule.
Most of the algorithms suggested in the literature [6, 27, 7, 8] require that the dependences be uniform or are approximated by dependence distance vectors. This condition is too restrictive for real programs. In this section, we show that the use of the dependence cone abstraction enables the obtaining of the same set of valid linear schedules as the dependence distance vector. However, when the computation of the linear schedule depends on particular distance vector characteristics (uniform) [6], the dependence cone that approximates the distance vectors cannot be used for defining the linear schedule but only for testing its validity.

6.1 One-dimensional Linear Schedule

The one-dimensional linear schedule is also called the *hyperplane method.* The scheduling is characterized by a vector $\vec{h}$ orthogonal to a hyperplane of simultaneously executable iterations .

Definition 8 *A linear schedule $\vec{h}$ is valid if the following condition is satisfied:* $\forall \vec{d} \in D,\ \ \vec{h} \cdot \vec{d} \geq 1$.

According to Definition 8, the set of valid linear schedules for D is defined by:

$$\mathrm{H} \ = \ \{\vec{h} \mid \ \forall \vec{d} \in \mathrm{D},\ \ \vec{h} \cdot \vec{d} \geq 1\} \qquad (1)$$

Likewise, the set of valid linear schedules for DC and DDV are defined as:

$$\mathrm{HC} \ = \ \{\overrightarrow{hc} \mid \ \forall \vec{d} \in \mathrm{DC}\ ,\ \ \overrightarrow{hc} \cdot \vec{d} \geq 1\} \qquad (2)$$

$$\mathrm{HDV} \ = \ \{\overrightarrow{hv} \mid \ \forall \vec{d} \in \widetilde{\mathrm{D}}_{\mathrm{DDV}}\ ,\ \ \overrightarrow{hv} \cdot \vec{d} \geq 1\} \qquad (3)$$

Since $\mathrm{D} \supseteq \mathrm{DC} \supseteq \mathrm{DDV}$, logically, we have $\mathrm{HDV} \subseteq \mathrm{HC} \subseteq \mathrm{H}$. In fact, we have the following theorem:

Theorem 7 *The set of valid linear schedules HC, defined by equation (2), is equal to the set H of valid linear schedules for D.*

Proof: Definitions (1) and (2) associated with the fact that D $\supset$ DC implies HC $\subseteq$ H. In order to prove H $\subseteq$ HC, we show that any vector $\vec{h}$ of H belongs to HC. Let $\vec{h} \in$ H and $\vec{d_i} \in$ D. Definition (1) implies: $\forall \vec{d_i} \in$ D, $\vec{h} \cdot \vec{d_i} \geq 1$ Let $\overrightarrow{dc}$ be a vector of DC. By definition, $\overrightarrow{dc} = \sum_{i=1}^{k} \lambda_i \vec{d_i}$ with $\lambda_i \geq 0$ and $\sum_{i=1}^{k} \lambda_i \geq 1$. Then, $\vec{h} \cdot \overrightarrow{dc} = \sum_i^k \lambda_i \vec{h} \cdot \vec{d_i} \geq (\sum_i^k \lambda_i) \times (\min_i(\vec{h} \cdot \vec{d_i}))$. Since $\sum_{i=1}^{k} \lambda_i \geq 1$, $\vec{h} \cdot \overrightarrow{dc} \geq (\min_i(\vec{h} \cdot \vec{d_i})) \geq 1. \Longrightarrow \vec{h} \in$ HC. Thus we have HC = H.□
The set of valid linear schedules H (or HC) can be represented by a polyhedron [16].

Definition 9 *The set of valid linear schedules H is a polyhedron. It can be defined by a linear system whose constraints are derived from the generating system of* $DC = \{\{\vec{s_i}\}, \{\vec{r_j}\}, \{\vec{l_k}\}\}$ *in the following way:*

if $\forall i\ \vec{s_i} \neq \vec{0}$ *, then H is given by :*

$$\begin{cases} \forall i, & \vec{h} \cdot \vec{s_i} \geq 1 \\ \forall j, & \vec{h} \cdot \vec{r_j} \geq 0 \\ \forall k, & \vec{h} \cdot \vec{l_k} = 0 \end{cases}$$

else if $\exists i\ \vec{s_i} = \vec{0}$ ***and there is no line*** $(|\{\vec{l_k}\}| = 0)$ *Then H is defined by :*

$$\begin{cases} \forall i\ t.q.\ \vec{s_i} \neq \vec{0}, & \vec{h} \cdot \vec{s_i} \geq 1 \\ \forall j, & \vec{h} \cdot \vec{r_j} \geq 1 \end{cases}$$

else $H = \emptyset$.

Since a DDV is also a polyhedron, it can be rewritten by using a generating system [13]. HDV, the set of valid linear schedules corresponding to DDV, can be computed similarly as HC.

```
      DO I = 1, N
         DO J = 1, N
S1:         V(I) = W(I+J)
S2:         W(I) = V(I+J)
         ENDDO
      ENDDO
```

Fig. 18. Program 7

Program 7 has four dependence relations:
$dep1:\ V(I) \rightarrow V(I)$,
$dep2:\ W(I) \rightarrow W(I)$,
$dep3:\ V(I+J) \longrightarrow V(I)$,
$dep4:\ W(I+J) \longrightarrow W(I)$

that can be expressed with the following different abstractions:

- distance vector
  - $D(dp1) = D(dp2) = \{(0, y) | 1 \leq y \leq n\}$
  - $D(dp3) = D(dp4) = \{(x, y) | 1 \leq x \leq n, 1 \leq y, 1 \leq x + y \leq n\}$
- dependence cone
  - $DC(dp1) = DC(dp2) = \{\{(0, 1)\}, \{(0, 1)\}, \emptyset\}$
  - $DC(dp3) = DC(dp4) = \{\{(1, 0)\},\ \{(0, 1), (1, -1)\}, \emptyset\}$
- dependence direction vector
  - $DDV(dp1) = DDV(dp2) = (=, <)$
  - $DDV(dp3) = DDV(dp4) = (<, *)$

The unions of all dependences for each abstraction are:

- $D(L) = \{(x, y) | x \geq 0, x + y \leq n, 1 \leq y\}$
- $DC(L) = \bigcup_{1 \leq i \leq 4} DC(dpi) = \{\{(0, 1)\}, \{(0, 1), (1, -1)\},\ \emptyset\}$
- $DDV(L) = \{(=, <), (<, *)\}$

In this example, DC(L) is equivalent to D(L). It is more compact than D(L) whose expression is a linear system.
H cannot be directly derived from D because D(L) has an infinite number of elements. In this case, its computation is equivalent to solving the following non-linear system:

$$\begin{cases} h_1 \times x + h_2 \times y \geq 1 \\ x \geq 0 \\ 1 \leq y \\ 1 \leq x + y \leq n \end{cases}$$

Otherwise, we can compute HC using Definition 9. It is defined by the following linear system on $\vec{h} = (h_1, h_2)$:

$$\begin{cases} h_2 \geq 1 \\ h_2 \geq 0 \\ h_1 - h_2 \geq 0 \end{cases} \quad \Longrightarrow \quad \begin{cases} h_2 \geq 1 \\ h_1 - h_2 \geq 0 \end{cases}$$

The generating system expression of this polyhedron is: $\{\{(1, 1)\}, \{(1, 0), (1, 1)\}, \emptyset\}$.
If we rewrite DDV(L) in a generating system form, using Definition 9, we can check that there is no legal schedule corresponding to DDV(L).
This example shows that DC is more convenient than D for computing the set of valid linear schedules, while DDV is not accurate enough.

6.2 Multi-dimensional Linear Schedule

The multi-dimensional linear schedule of a loop nest corresponds to a loop reordering transformation such that all dependences are carried by the k $(2 \leq k)$ outmost loops. The k-dimensional linear schedule is characterized by k linear vectors $(\vec{h_1}, ..., \vec{h_k})$.

Definition 10 *A linear schedule* $(\vec{h_1}, ..., \vec{h_k})$ *is valid if the following condition is satisfied:* $\forall \vec{d} \in D, \ \ (\vec{h_1}, .., \vec{h_k}) \ \cdot \ \vec{d} \gg 0$.

The set of valid linear schedules for D, DC and DDV are defined as:

$$\mathrm{HK} = \{(\vec{h_1}, ..., \vec{h_k}) \mid \ \forall \vec{d} \in \mathrm{D} \ , \ \ (\vec{h_1}, ..., \vec{h_k}) \cdot \vec{d} \gg 0\} \qquad (4)$$

$$\mathrm{HKC} = \{(\vec{h_1}, ..., \vec{h_k}) \mid \forall \vec{d} \in \mathrm{DC}, (\vec{h_1}, ..., \vec{h_k}) \cdot \vec{d} \gg 0\} \qquad (5)$$

$$\mathrm{HKDV} = \{(\vec{h_1}, ..., \vec{h_k}) \mid \forall \vec{d} \in \widetilde{\mathrm{D}}_{\mathrm{DDV}}, (\vec{h_1}, ..., \vec{h_k}) \cdot \vec{d} \gg 0\} \qquad (6)$$

Similarly to the one-dimensional schedule, the following theorem reports that HK = HKC.

Theorem 8 *The set of valid linear multi-dimensional schedules HKC defined by equation (5) is equal the set HK of valid linear multi-dimensional schedules for D.*

Proof: This proof is similar to the previous one. □

7 Related Work

Most dependence abstractions have originally been developed for particular transformations. Like these examples, the dependence direction vector was defined by M. Wolfe for loop interchanging [30] and the dependence level by Allen & Kennedy for the vectorization and the parallelization of programs [1]. The dependence direction vector has been the most popular dependence abstraction because it has been successfully used for some important transformations such as loop interchanging and loop permutation. Moreover, its computation is easy and its representation in the dependence graph handy. Most compiler systems have implemented distance and direction vector abstractions. A lot of work has been done on developing more advanced transformations but little effort has been spent on studying the valid and minimal dependence abstraction for a loop transformation.

In [31], M. Wolfe has posed a similar question, "What information is needed to decide when a transformation is legal?". A review of the related work on loop transformations and dependence abstractions, including some particular dependence information such as crossing threshold and cross-direction, is presented. A table gives the valid dependence abstraction supporting each considered transformation. However, some important abstractions, such as the dependence cone, and some advanced transformations, such as unimodular transformations and loop partitioning, were not considered.

V. Sarkar & R. Thekkath [25] developed a general framework for applying reordering transformations. They used the *dependence vector* abstraction whose element is either a distance value, in case of constant dependence, or a direction value, in the other cases. For each considered transformation, the rules for mapping dependence vectors, loop bound expressions and the loop nest are defined. These mapping rules were developed from the *dependence vector* abstraction. However, the dependence vector does not contain sufficient information to legally apply advanced transformations, such as unimodular transformations and loop tiling, without some risks of losing precision.

M. E. Wolf & M. S. Lam [28] introduced a new type of dependence vector for applying unimodular transformations and loop tiling. In their definition, each component d_i of the dependence vector can be an infinite range of integers, represented by $[d_i^{min}, d_i^{max}]$. This new dependence vector is more precise than V. Sarkar's dependence vector, and overcomes some drawbacks in the direction vector by using the value range of d_i instead of its sign. On the other hand, it is less precise than the dependence cone DC because it is only the projection of DC on the d_i axis. The test of legality based on this dependence vector is approximate because the addition, subtraction and multiplication operations defined on this dependence vector are conservative.

Our study deals with abstractions that are derived from the dependence distance vectors. But, abstraction such as the information given by array data flow analysis [9, 21] provides more precise information about dependencies than the dependence distance vectors. At that moment, these precise analyses are limited to static control programs. However, when computed, they give information sufficient to decide the legality of transformations such as the privatization [21] or linear scheduling computation [10, 11] while other abstractions might not be able to reach the necessary accuracy. In this study we restricted our attention to abstractions that can be computed on any program. Abstractions providing more precise information than the dependence distance vectors have to be introduced to deal with those transformations.

Linear scheduling computation techniques [10, 11, 17] exploit the additional information provided by array data flow graphs (especially, the sources of the dependent iterations) to restructure the program as they compute the optimal scheduling. The linear schedules that can be computed from DC are limited to loop reordering transformations while affine scheduling obtained by [10, 11, 17] is more general and may be combination of loop skewing, loop peeling and statement reordering.

Conclusion

The precision criteria for abstractions DI, D, DP, DC, DDV, and DL have been presented in this paper and report that the precision hierarchy of the abstractions is DI $\supseteq$ D $\supseteq$ DP $\supseteq$ DC $\supseteq$ DDV $\supseteq$ DL.

The minimal abstraction associated with a transformation, that contains the minimal information necessary to decide when such a transformation is legal, is defined for three different classes of transformations. The minimal abstraction for the reordering transformations: loop reversal, loop permutation, unimodular transformation, partitioning and parallelization have been identified and are respectively DL, DDV, DC, DC and DL. According to the definition, all the abstractions that are more precise than the minimal abstraction associated with the transformation are valid for such a transformation.

This paper shows that the dependence cone DC carries enough information for testing the legality of some advanced transformations such as unimodular transformations and loop partitionings. Moreover, this representation lets us to obtain the same set of valid linear schedules, both one- and multi-dimensional, as with abstraction D without any loss.

Acknowledgments

We wish to give special thanks to P. Jouvelot and F. Coelho for their constructive remarks and critical reading of this paper.

References

1. R. Allen, K. Kennedy, "Automatic Translation of FORTRAN Programs to Vector Form", *ACM Transactions on Programming Languages and Systems*, Oct. 1987.
2. U. Banerjee, "A Theory of Loop Permutations", *2nd Workshop on Languages and compilers for parallel computing,* 1989
3. U. Banerjee, "Unimodular Transformation of Double Loops", *3rd Workshop on Programming Languages and Compilers for Parallel Computing,* Irvine, 1990
4. U. Banerjee, "Loop transformations for restructuring compilers: the foundations", *Kluwer Academic Editor*, 1993
5. A. J. Bernstein, "Analysis of Programs for Parallel Processing", *IEEE Transactions on Electronic Computers*, Vol. EC-15, No. 5, Oct. 1966.
6. E. D'hollander, "Partitioning and Labeling of Index Sets in Do Loops with Constant Dependence Vectors", *International Conference on Parallel Processing*, 1989
7. A. Darte, Y. Robert, "Scheduling Uniform Loop Nests", *Rapport de Laboratoire de l'informatique du Paralle'lisme, Ecole Normale Sup'erieure de Lyon'*, No. 92-10, Fev. 1992.
8. A. Darte, T. Risset, Y. Robert, "Loop Nest Scheduling and Transformations", *Conference on Environement and Tools for Parallel Scientific Computing, CNRS-SNF,* Saint-Hilaire du Touvier, France, Sep. 1992.
9. P. Feautrier, "Dataflow Analysis of Scalar and Array References", *Int. Journal of Parallel Programming*, Vol. 20, No 1,February,1991, pp 23-53.
10. P. Feautrier, "Some Efficient Solutions to the Affine Scheduling Problem, Part I, One-Dimensional Time", *Int. Journal of Parallel Programming*, Vol 21, 1992.
11. P. Feautrier, "Some Efficient Solutions to the Affine Scheduling Problem, Part II, Multi-Dimensional Time", *Int. Journal of Parallel Programming*, Vol 21, 1992.
12. F. Irigoin, P. Jouvelot, R. Triolet, "Semantical Interprocedural Parallelization: an Overview of the PIPS Project", In *ACM International Conference on Supercomputing*, ICS'91, Cologne, Allemagne, 16-21 Juin 1991.
13. F. Irigoin, "Loop Reordering With Dependence Direction Vectors", In *Journées Firtech Systèmes et Télématique Architectures Futures: Programmation parallèle et intégration VLSI*, Paris, Nov. 1988.
14. F. Irigoin, R. Triolet, "Computing Dependence Direction Vectors and Dependence Cones with Linear Systems", *Rap.Int. CAI87E94 Ecole des Mines de Paris*,
15. F. Irigoin, R. Triolet, "Supernode Partitioning", In *Conference Record of Fifteenth ACM Symposium on Principles of Programming Languages*, 1988.

16. F. Irigoin, R. Triolet, "Dependence Approximation and Global Parallel Code Generation for Nested Loops", In *International Workshop Parallel and Distributed Algorithms*, Bonas, France, Oct. 1988.
17. W. Kelly, W. Pugh, "Finding Legal Reordering transformations using Mappings", In *Seventh Annual Workshop Languages and Compilers for Parallel Computing*, Ithaca, New York, August 8-10,1994.
18. R. Karp, R. Miller and S. Winograd, "The Organization of Computations for Uniform Recurrence Equations", *Journal of the ACM,* v. 14, n. 3, pp. 563-590, 1967
19. L. Lamport, "The Parallel Execution of DO Loops", *Communications of the ACM 17(2),* pp. 83-93, 1974
20. W. Li, K. Pingali, "A singular loop transformation framework based on non-singular matrices", In *Languages and Compilers for Parallel Computing*, Yale University, August 1992.
21. D. Maydan, S. Amarasinghe, M.Lam, "Array Data Flow Analysis and its use in Array Privatization", *Stanford Report*, 1993
22. Dror E. Maydan, John L. Hennessy, Monica S. Lam, " Efficient and Exact Dependence Analysis ", *ACM Sigplan PLDI'91* , Toronto, Ontario, Canada, June 1991
23. Y. Muraoka, "Parallelism Exposure and Exploitation in Programs", *PhD thesis*, Dept. of Computer Science, University of Illinois at Urbana-champaign, Feb. 1971.
24. William Pugh, "A Practical Algorithm for Exact Array Dependence Analysis", *Communications of the ACM* , August 1992, pp.102-114
25. V. Sarkar, R. Thekkath, "A General Framework for Iteration-Reordering Loop Transformations", In *Programming Language Design and Implementation,* San Francisco, June 1992.
26. A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley & Sons 1986.
27. W. Shang, J. A. B. Fortes, "Time Optimal Linear Schedules for Algorithms with Uniform Dependencies", *IEEE Transactions on Computers*, Vol. 40, No. 6, June 1991.
28. M.E. Wolf, M.S. Lam, "Maximizing Parallelism via Loop Transformations", In *Programming Languages and Compilers for Parallel Computing,* Aug. 1-3, 1990.
29. M.E. Wolf, M.S. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism", *Transactions on Parallel and Distributed Systems*, Oct. 1991.
30. M. Wolfe, "Optimizing Supercompilers for Supercomputers", *PhD Thesis, Dept. of Computer Science,* University of Illinois at Urbana-Champaign, October 1982.
31. M. Wolfe, "Experiences with Data Dependence and Loop Restructuring in the Tiny Research Tool", *Technical Report*, No. CS/E 90-016, Sep. 1990.
32. M. Wolfe, "Experiences with Data Dependence Abstractions", In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Germany, June 1991.
33. Y.Q. Yang, "Tests de Dependance et Transformations de programme", *PhD of University Pierre et Marie Curie* , November 93.