Minimal Data Dependence Abstractions for Loop Transformations

Yi-Qing Yang Corinne Ancourt François Irigoin*
Ecole des Mines de Paris/CRI
77305 Fontainebleau Cedex
France

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 precision. 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. The minimal abstractions for loop reordering and unimodular transformations are presented. As an example, the dependence cone, that 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 schedulings than the dependence distance set.

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 preserved after the transformation.

Many transformations require more than a simple dependent/independent information to be legally applied on 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 [Mura71], the Dependence Direction Vector [Wolf82], the Dependence Level [AlKe87] or the Dependence Cone [IrTr87]. 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 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 to 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 cheapest abstraction in time and space.

Section 6 shows that the dependence cone, which is a *valid* abstraction for all loop reordering transformations and the *minimal* abstraction for unimodular transformations, allows in addition to find the same set of valid mono- and multi-dimensional linear schedulings than 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.
- δ^*_{Ab} is the dependence relation represented by Abstraction Ab.

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 on example in Figure1.

*E-mail: <yang,ancourt,irigoin@cri.ensmp.fr>

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

Figure 1: Program 1

The dependence test must decide the satisfiability of the following dependence system:

$$\begin{cases} & 1 \le i \le n \\ 3i = i' & 1 \le j \le n \\ j+1 = j' & 1 \le i' \le n \\ & 1 \le j' \le n \end{cases}$$

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:

$$DI(L) = \{\ (\vec{i},\ \vec{i'}) \mid \exists S1, S2 \in L,\ S1(\vec{i})\ \delta^*\ S2(\vec{i'})\ \}$$

$DI(L)$ 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.
$DI(P1) = \{((i,j),(3i,j+1)) \mid 1 \le i \le n; 1 \le j \le n\}$.

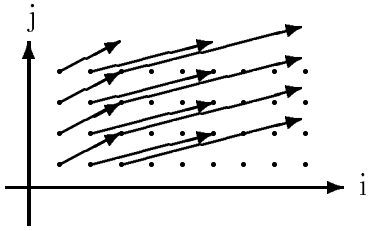

Figure 2: DI for program 1

Since no approximation is made on the exhaustive list of dependent iterations, this abstraction cannot be used for infinite set of dependences. Thus, 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:

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

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

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

The dependences represented in Figure 3 are specified by: $D(P1) = \{(2k, 1) \mid 1 \le k \le n\}$. When comparing with Figure 2, note the different space: (i,j) is replaced by (d_i, d_j).

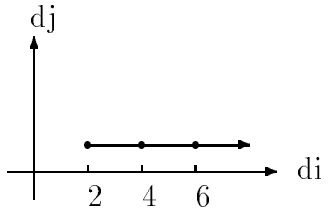

Figure 3: D for Program 1

Compared to DI, the use of D reduces the amount of memory needed to store uniform dependences[1]. However, in many cases, the dependence distance is not constant. So, 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

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.

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

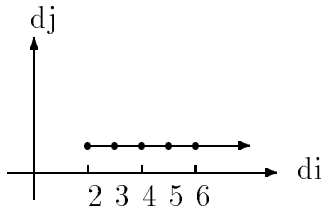

Figure 4: DP for Program 1

The polyhedron $DP(L)$ can be concisely described by its *generating system* [Schr86], 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 $DP(L)$

[1] usually known as *constant* dependences

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:

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

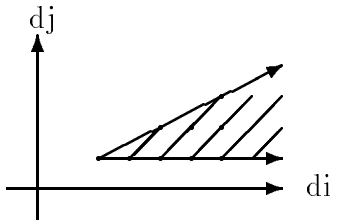

Figure 5: DC for Program 1

The main advantage of Abstractions DP and DC is that, in many cases, only one structure is necessary to the representation of the information contained in $D(L)$. They can be automatically computed and accurately so when array subscript expressions are affine. They are easy to use to decide program transformation validity. Algorithms for computing $DP(L)$ and $DC(L)$ are described in [IrTr87] and [Yang93].

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 to legally apply transformations that reorder iteration sets. However, transformations such as loop interchange or permutation, that only modify the sign or the order of the iteration set, do not need the actual distance information. So, abstractions like $DDV(L)$ [Wolf82] and $DL(L)$ [AlKe87] relating only the sign or the level of dependence vectors have been suggested.

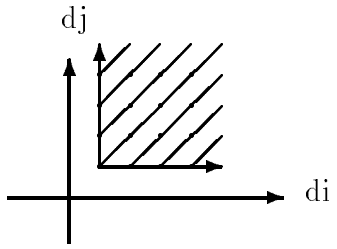

Figure 6: DDV for program 1

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:

$DDV = \{\ (\psi_1, \psi_2, ..., \psi_n)\ |\ \exists S1, S2 \in L,\ S1(\vec{i})\ \delta^*\ S2(\vec{i'}),\ i_k\ \psi_k\ i'_k\ (1 \leq k \leq n),\ \psi_i \in \{<, =, >\}\}$

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

2.6 The Dependence Level

The dependence level DL has been introduced by Allen & Kennedy [AlKe87] for the vectorization and parallelization of programs. It gives the nest level of the outermost loop l that carries dependences, when dependence vector component is positive. To preserve the program semantics, Loop l must be kept sequential. Then, the lexico-positivity of the loop dependences are preserved and all inner loops may be parallelized if no other dependence exists. $DL(L)$ is defined as:

$DL(L) = \{k\ |\ \exists(\psi_1, \psi_2, ..., \psi_n) \in DDV(L),\ \psi_i\ is\ =(1 \leq i \leq k-1)\ \wedge\ \psi_k = <\ \}.$

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

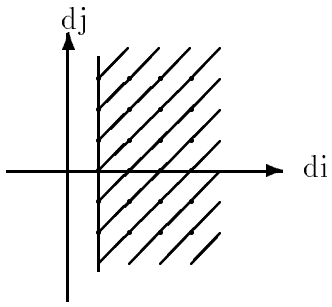

Figure 7: DL for program 1

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 the dependent iterations when the 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 $\widetilde{DI}_{Ab}$ for each abstraction Ab. $\widetilde{DI}_{Ab}$ is the set of dependent iterations corresponding to the dependences represented by abstraction Ab.

$\widetilde{DI}_{Ab} = \{\ (\vec{i}, \vec{i}+\vec{d}) : \vec{i}\ , \vec{i}+\vec{d} \in I^n,\ \vec{d} \in \widetilde{D}_{Ab}\}$

where $\widetilde{D}_{Ab}$ is the set of dependence Distance vectors corresponding to the dependences represented by abstraction Ab.

To compare two ordinary abstractions $Ab1$ and $Ab2$, the two sets $\widetilde{DI}_{Ab1}$ and $\widetilde{DI}_{Ab2}$ are analyzed.

Definition 1 *$Ab1$ is more precise than $Ab2$ (noted $Ab1 \supset Ab2$) if $\widetilde{DI}_{Ab1} \subseteq \widetilde{DI}_{Ab2}$.*

To compare abstractions approximating D, the sets $\widetilde{D}_{Ab1}$ and $\widetilde{D}_{Ab2}$, having a lower memory cost than $\widetilde{DI}$, could be used[2].
According to Definition 1, the hierarchy of abstraction precisions is: $DI \supseteq D \supseteq DP \supseteq DC \supseteq DDV \supseteq DL$. Figures 2,4,3,5,6,7 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 $S(\vec{i}^n)$ in a new one $S'(\vec{j}^m)$. As an example of this kind of transformations, 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 bijection 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 from m. Strip mining, that reorders by partitioning the iteration set into several iteration blocks, belongs to this class of transformations.

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

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

4.1 Legal Transformation

A transformation is *legal* if the program has the same semantics after transformation than before. The new execution of statement instances must preserve all the loop nest lexico-positive dependences [Bane93].

Program restructuring transformation

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

[2] Proof is given in [Yang93]

Definition 2 ensures that the lexico-positivity of the dependences is preserved after transformation. When an abstraction Ab 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 $\widetilde{DI}_{Ab}$, the set of dependent iterations corresponding to the dependences of Ab.

Definition 3 *Performing a restructuring transformation T^{Ab} on a loop nest L is* legal, *which is noted $legal(T, L, Ab)$ if and only if:*
for any dependence $S_i(\vec{i}^n)\ \delta^*\ S_j(\vec{i'}^n)$ of L such that
$\forall\ (\vec{i}, \vec{i'}) \in \widetilde{DI}_{Ab},\ T(S_i(\vec{i}^n)) = S'_i(\vec{j}^m)$ *and* $T(S_j(\vec{i'}^n)) = S'_j(\vec{j'}^m)$,
the condition $S'_i(\vec{j}^m) \prec S'_j(\vec{j'}^m)$ 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. $legal(T, L, Ab) \implies legal(T, L)$*

Proof:
Assuming that T^{Ab} is legal according to the condition of Definition 3, then:
$\forall(\vec{i}, \vec{i'}) \in \widetilde{DI}_{Ab}$ with $T(S_i(\vec{i}^n)) = S'_i(\vec{j}^m)$ and $T(S_j(\vec{i'}^n)) = S'_j(\vec{j'}^m)$, the condition $S'_i(\vec{j}^m) \prec S'_j(\vec{j'}^m)$ is verified.
Because DI is the most precise dependence abstraction, we have $DI \subseteq \widetilde{DI}_{Ab}$ that means that:
$\forall(\vec{i}, \vec{i'}) \in DI \implies (\vec{i}, \vec{i'}) \in \widetilde{DI}_{Ab}$.
It implies that
$\forall(\vec{i}, \vec{i'}) \in DI$ such that $T(S_i(\vec{i}^n)) = S'_i(\vec{j}^m)$ and $T(S_j(\vec{i'}^n)) = S'_j(\vec{j'}^m)$, the condition $S'_i(\vec{j}^m) \prec S'_j(\vec{j'}^m)$ is also verified.

Reordering transformation

Reordering transformations only reorder the iteration set of a loop nest. So, the dependences $S_i(\vec{i}^n)\ \delta^*\ S_j(\vec{i'}^n)$ between references S_i and S_j 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:
$DI(L) = \{(i_1, i_2) | i_1 \prec_\delta i_2 \Leftrightarrow \exists S_i(i_1)\ \delta^*\ S_j(i_2) \wedge i_1 \neq i_2\}$
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 Ab different from DI is used, a definition using dependences in function of $\widetilde{DI}_{Ab}$ is derived from Definition 4.

4.1.1 Unimodular transformation

An unimodular transformation [Bane90], [WoLa91], [LiPi92] is a bijection between two iteration sets I^n and I'^n having the same dimension. It corresponds to an unimodular change of basis M with $\vec{I'} = M \times \vec{I}$. 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 $i'_1 = M \times i_1$ and $i'_2 = M \times i_2$.
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 $\vec{d'} = M \times \vec{d}$.
Because the unimodular transformations reorder iteration sets only, a single set:
$D(L) = \{\vec{d} \mid \vec{d} = \vec{i_2} - \vec{i_1},\ \vec{i_1}\delta^*\vec{i_2} \wedge \vec{i_1} \neq \vec{i_2}\}$ can be used to characterize all loop nest dependences. The following definitions and theorem describe the legality tests for unimodular transformations:

Definition 5 *Performing an unimodular transformation T on a loop nest L is legal if and only if:*

$\forall \vec{d}^{\,n} \in D(L)$ *such that* $T_{unimod}(\vec{d}^{\,n}) = \vec{d'}^{\,n}$,

the condition $\vec{d'}^{\,n} \gg 0$ *is verified.*

As for reordering transformations, when an abstraction Ab different from D is used, a definition using $\widetilde{D}_{Ab}$ 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 [3] or from the dependence distance vector set [4] 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 computation or memory cost, could be used.
We define a *valid* abstraction associated to a transformation T as an approximate abstraction containing enough information to decide the legality of T. Among the abstractions, DI is the most precise. So, if an abstraction Ab (different from DI) contains the same information as DI to decide that the transformation T is legal, then this abstraction Ab is *valid* for T.

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

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

Theorem 2 *Let Ab be a valid abstraction for testing the legality of a transformation T. Any abstraction Ab_i such that $Ab_i \supseteq Ab$ is also a valid abstraction for testing the legality of T.*

Proof:
We know that Ab is a valid abstraction for $T(L)$
and that $Ab_i \supseteq Ab$. Then, in accordance with Definition 6: $legal(T, L, DI) \implies legal(T, L, Ab)$.
Definition 3 gives the equivalence:
$legal(T, L, Ab) \Longleftrightarrow [\ \forall(\vec{i}, \vec{i'}) \in \widetilde{DI}_{Ab}$ with $T(S_i(\vec{i}^{\,n})) = S'_i(\vec{j}^{\,m})$ and $T(S_j(\vec{i'}^{\,n})) = S'_j(\vec{j'}^{\,m})$ then $S'_i(\vec{j}^{\,m}) \prec S'_j(\vec{j'}^{\,m})]$.
By hypothesis $Ab_i \supseteq Ab$, so $\forall\ (\vec{i}, \vec{i'}) \in \widetilde{DI}_{Ab_i}$ then $(\vec{i}, \vec{i'}) \in \widetilde{DI}_{Ab}$.
By combining this hypothesis with the previous equivalence, the following assertion is verified: $\forall\ (\vec{i}, \vec{i'}) \in \widetilde{DI}_{Ab_i}$ with $T(S_i(\vec{i}^{\,n})) = S'_i(\vec{j}^{\,m})$ and $T(S_j(\vec{i'}^{\,n})) = S'_j(\vec{j'}^{\,m})$. Then $S'_i(\vec{j}^{\,m}) \prec S'_j(\vec{j'}^{\,m})$.
That is equivalent to:
$legal(T, L, Ab) \implies legal(T, L, Ab_i)$,
and implies $legal(T, L, DI) \implies legal(T, L, Ab_i)$.

Among the valid abstractions associated to a transformation, there is one 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 $Ab1$ be a valid abstraction for the transformation T and $Ab2$ be a another abstraction such that $Ab1 \supseteq Ab2$ and $\neg\exists Ab3$ / $Ab1 \supseteq Ab3 \supseteq Ab2$. If $\exists L$ such that $legal(T, L, Ab1) \wedge \neg(legal(T, L, Ab2))$, then abstraction $Ab1$ is* minimal *for testing the legality of the transformation T.*

Minimality is relative to a particular set of abstractions. The minimal abstractions associated to reordering transformations are presented in the next section.

5 Minimal Abstraction and Transformation

Dependence abstractions and loop transformations are linked. The dependences are used to test the legality

[3] DI or $\widetilde{DI}_{Ab}$ when Ab is different from DI
[4] D or $\widetilde{D}_{Ab}$ when Ab is different from D

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 to a transformation is important for programs.
Among the valid abstractions associated to a transformation, one is the *minimal* abstraction. It contains the minimal information necessary to decide if the transformation can be applied legally or not. This section focuses on the minimal abstraction associated to the following transformations: loop reversal, loop permutation, unimodular transformation, partitioning and parallelization.
For any T among these transformations, a section presents the effect of T on the dependences, the minimal[5] abstraction associated to T and the legality test applied to T.
Due to space limitation, the proofs of the minimality of an abstraction are not given in this paper. But a counter example, illustrating the transformation, shows that an abstraction less precise than the minimal abstraction does not contain enough information to decide of the transformation legality.

5.1 Loop reversal

A loop reversal transformation $Invers_k\ (l_1, l_2, ..., l_n)$ 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 $Invers_{\ k}\ (\vec{d}) = (d_1, ..., -d_k, ..., d_n)$. So, the lexico-positivity of the dependences is preserved after transformation if the following condition is verified:
$legal(Invers_K, L) \Longleftrightarrow$
$$\forall \vec{d} \in D(L), (d_1, ..., -d_k, ..., d_n) \gg 0.$$

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

An example illustrating the previous assertion:

```
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
```

Figure 8: Program 2

[5] among those presented in this paper

In Program 2, the loop nest has two data flow dependence relations that are expressed with the following dependence abstractions:
- $D = \{(1, 0, -k), (0, 1, -k) \mid 2 \leq k \leq n\}$
- $DDV = \{(<, =, >), (=, <, >)\}$
- $DL = \{1, 2\}$

Abstraction DL asserts that the 3-rd loop can be reversed because the constraints on the lexico-positive dependences are still preserved after reversing K :
$3 \notin DL \Longrightarrow DL = \{1, 2\}$ after transformation. Now, imagine a dependence abstraction Ab less precise than DL and giving only a general dependent/independent information on the set of statements on which the transformation must be applied. Then, not enough information is contained in Ab to verify if reversing the 3-rd loop is legal or not. In doubt, the loop reversal is declared illegal. Using DL, loop K can be reversed. The new loop nest is figured in Figure 9.
All abstractions DI, D, DC, DP, DDV and DL are *valid* for a loop reversal transformation. The test of legality associated to the minimal abstraction DL is:
$$legal(Invers_k, L) \Longleftrightarrow k \notin DL(L)$$
$$\Longleftrightarrow projection(DL(L), k) = \emptyset.$$

5.2 Loop permutation

A loop permutation transformation [Bane89] $Perm_P\ (L)$ performs permutation P on the n-dimensional iteration set of the loop nest $L = (l_1, l_2, .., l_n)$. The new loop nest L' is defined by $L' = (l_{p[1]}, l_{p[2]}, ..., l_{p[n]})$ 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:
$$\vec{d'} = Perm_P\ (\vec{d}) = (d_{p[1]}, ..., d_{p[k]}, ..., d_{p[n]}).$$
So, the lexico-positivity of the dependences is preserved after transformation if the following condition is verified:
$legal(Perm\ _P,\ L) \Longleftrightarrow$
$$\forall \vec{d} \in D(L),\ (d_{p[1]}, ..., d_{p[k]}, ..., d_{p[n]}) \gg 0$$

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

An example illustrating the previous assertion:

The dependences of Program 3 in Figure 9 can be represented using the D, DDV and DL abstractions:
- $D = \{(1, 0, k), (0, 1, k) \mid 2 \leq k \leq n\}$
- $DDV = \{(<, =, <), (=, <, <)\}$
- $DL = \{1, 2\}$

The use of abstraction DDV allows to verify 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.

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

Figure 9: Program 3

In contrast, the use of DL [6] does not allow to conclude the legality of this permutation, because no dependence information is known on the 3-rd loop which will be the outmost loop afer permutation.
All abstractions DI, D, DC, DP and DDV are *valid* for a loop permutation transformation. The test of legality associated to the minimal abstraction is then:
$legal(Perm_P, L) \iff$
$\forall \overrightarrow{ddv} \in DDV(L), Perm_P(\overrightarrow{ddv}) \gg 0.$

5.3 Unimodular Transformation

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

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

An example illustrating the previous assertion:

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

Figure 10: Program 4

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:

$$\begin{cases} di = 0 \\ dj = j \\ 1 \le j \le n \end{cases}$$

Abstractions D, DC and DDV represent these dependences in the following manner:
- $D(dp1) = \{(0, x) \mid 1 \le x\}$
- $DC(dp1) = (\{(0,1)\}, \{(0,1)\}, \emptyset)$
- $DDV(dp1) = \{(=, <)\}$

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:

$$\begin{cases} di = i \\ di + dj = 1 \\ 1 \le i \le n \end{cases}$$

The abstractions D, DC and DDV represent these dependences as:
- $D(dp2) = \{(x, -x + 1)\} \mid 1 \le x\}$
- $DC(dp2) = (\{(1,0)\}, \{(1,-1)\}, \emptyset)$
- $DDV(dp2) = \{(<, >)\}$

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 [IrTr87],[Wolf91] and [Yang93]. Using the same abstractions, this union is expressed as:
- $D(L) = D(dp1) \cup D(dp2) = \{(0, x), (x, -x+1) \mid 1 \le x\}$.
- $DC(L) = convex\_hull(DC(dp1) \cup DC(dp2)) = (\{(0,1)\}, \{(0,1), (1,-1)\}, \emptyset)$.

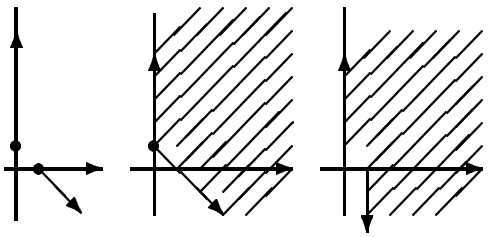

Figure 11: D(L), DC(L) and DDV(L)

- $DDV(L) = DDV(dp1) \cup DDV(dp2)$
$= \{(\le, <), (<, >)\}$.

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 :

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

The computation of the lexico-positivity constraints having to be preserved amounts to:
- $M \times D(dp1) = (x, 0)$ and $M \times D(dp2) = (1, x)$, which are $\gg 0$

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

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

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

5.4 Partitioning

A partitioning transformation [KMW67], [Lamp74], [IrTr88a], [Dhol89] $Part_H$ applied on a n-dimensional loop nest splits the iteration space into p parallel hyperplans. The shape and size of the partitioned blocks are defined by the partitioning vector $H = (\vec{h_1}, \vec{h_2}, ..., \vec{h_p})$, where the p hyperplans are 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 [IrTr88a]:
$(\lfloor \vec{h_1} \times \vec{i_1} \rfloor, \lfloor \vec{h_2} \times \vec{i_1} \rfloor, ..., \lfloor \vec{h_n} \times \vec{i_1} \rfloor) =$
$(\lfloor \vec{h_1} \times \vec{i_2} \rfloor, \lfloor \vec{h_2} \times \vec{i_2} \rfloor, ..., \lfloor \vec{h_n} \times \vec{i_2} \rfloor)$
According to the results of [IrTr88a], performing a partitioning H is legal if the following condition is verified:

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

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

$$H \cdot P_{Ab} \geq \vec{0} \Longrightarrow \quad legal(Part_H, L, Ab)$$

where $H \cdot P_{Ab} \geq \vec{0}$ is equivalent to:

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

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

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:

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

Figure 12: Program 5

$$\begin{cases} di = i \\ dj = 1 \\ 1 \leq i \leq n \end{cases}$$

The dependence can be represented using D, DC and DDV:
- $D(dep1) = \{(x,1) \mid 1 \leq x\}$
- $DC(dep1) = (\{(1,1)\}, \{(1,0)\}, \emptyset)$
- $DDV(dep1) = (<, <)$

The second anti-dependence characterizes data access conflicts between the two references $A(2I, I+J)$ and $A(I,J)$. Its dependence system is:

$$\begin{cases} di = dj \\ di \geq 1 \end{cases}$$

Its representations by D, DC and DDV is:
- $D(dep2) = \{(x,x) \mid 1 \leq x\}$
- $DC(dep2) = (\{(1,1)\}, \{(1,1)\}, \emptyset)$
- $DDV(dep2) = (<, <)$

The union of the two dependences is expressed as:
- $D(L) = D(dep1) \cup D(dep2)$
  $= \{(x,1),\ (x,x) \mid 1 \leq x\}$.

$D(L)$ is illustrated by Figure 13.

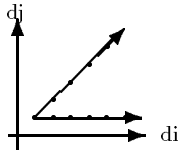

Figure 13: D for Program 5

- $DC(L) = convex\_hull(DC(dep1) \cup DC(dep2))$
  $= (\{(1,1)\}, \{(1,0),(1,1)\}, \emptyset)$.
- $DDV(L) = DDV(dep1) \cup DDV(dep2) = (<, <)$.

$DC(L)$ 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 $H = (h_1, h_2)$ 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:
- $H \cdot D(dep1) = (\frac{x-1}{3}, \frac{x}{3})$

$H \cdot D(dep2) = (0, \frac{x}{3})$.
Since $x \geq 1 \Longrightarrow H \cdot D(dep1) \geq \vec{0}$ and
$H \cdot D(dep2) \geq \vec{0}$, $Part_H^D(L)$ is legal with respect to D.

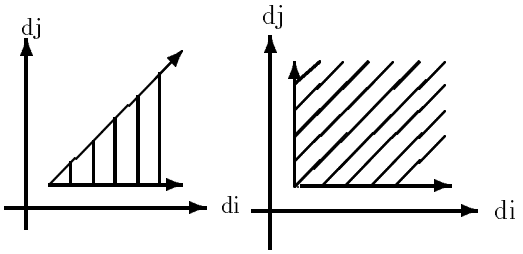

Figure 14: DC and DDV for Program 5

- $H \cdot DC(L) = (\ \{(0,\frac{1}{3})\},\{(\frac{1}{3},\frac{1}{3}),(0,\frac{1}{3})\},\emptyset)$

Since $s_i, r_j \in H \cdot DC(L) \geq \vec{0}$ and $d_k = \emptyset$, we have $H \cdot DC(L) \geq \vec{0} : Part_H^{DC}(L)$ is legal with respect to DC.

- $DDV(L) = \{\ (<,<)\ \}$

$P(DDV(L)) = (\ \{(1,1)\},\{(1,0),(0,1)\},\emptyset)$
$H \cdot DDV(L) = (\ \{(0,\frac{1}{3})\},\{(\frac{1}{3},\frac{1}{3}),(-\frac{1}{3},0)\},\emptyset)$
Since $s_1 = (0,\frac{1}{3}) \geq \vec{0})$ and $r_2 = (-\frac{1}{3},0) \leq \vec{0}$
$\Longrightarrow\ \neg(H \cdot DDV(L) \leq \vec{0}) \wedge \neg(H \cdot DDV(L) \geq \vec{0})$,
$Part_H^{DDV}(L)$ is illegal.
Figure 15 shows the H partitioning associated to the D dependences. We can see that there is no dependence cycles between partitioned blocks.

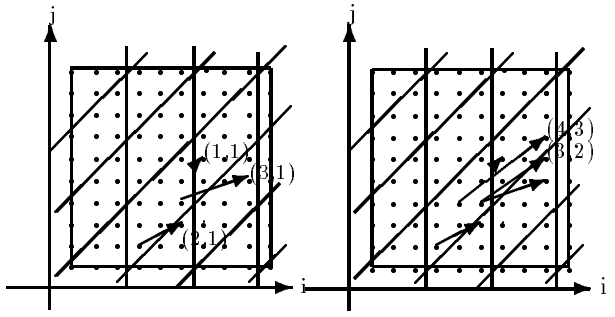

Figure 15: Partitioning H with D, DC

Figure 15 shows the H partitioning associated to the DC dependences. Note that the additional dependence distance vector belonging to DC does not introduce additional dependence relations between partitioned blocks.
Figure 16 shows the H partitioning associated to the DDV dependences. Here, two dependences represented by $DDV(L)$: (1,3) and (2,1) give a cycle between two blocks. Then, $Part_H^{DDV}(L)$ is illegal.
DC allows to conclude the legality of $Part_H$, because condition (1) is verified. On the contrary, abstraction DDV cannot conclude because the set of points described by: $\{s_1 = (0,\frac{1}{3}) \geq \vec{0} \wedge r_2 = (-\frac{1}{3},0) \leq \vec{0}\}$ verifies neither (1) nor (2).
In conclusion, all abstractions DI, D, DP and DC are

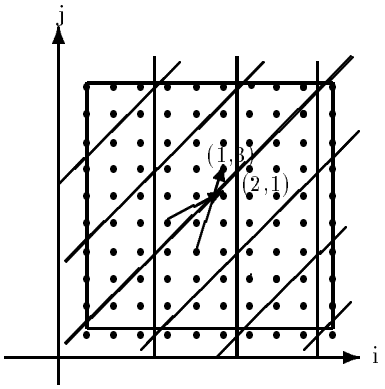

Figure 16: Partitioning H with DDV

valid for a loop partitioning transformation. The test of legality associated to the minimal abstraction is:
$(\wedge_{1\leq i\leq k}(H \cdot DC_i \geq \vec{0})) \vee (\wedge_{1\leq i\leq k}(H \cdot DC_i \leq \vec{0}))$
$\Longrightarrow legal(Part_H, L)$

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_k are the sequential loop indices. As the loop reversal transformation, the parallelization modifies the iteration execution order only in internal loop levels. Thus to preserve 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
$d_k = 0\ \ (1 \leq k \leq j-1)\ , d_j > 0$ and
$projection(\vec{d},\overrightarrow{pv}) = (d_{i_1}, d_{i_2}, ...d_{i_m})$ then $d_{i_1} = d_j$
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
```

Figure 17: Program 6

Taking Program 6, the dependences can be represented using D and DL:

- $D = \{(1,1)\}$
- $DL(L) = \{1\}$

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 permits to conclude that the parallelization of loop J is legal because the lexico-positivity constraints have been preserved after parallelization. An abstraction less precise giving, for example, only a 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 preserved if the parallelized loop levels do not appear in the dependence levels. So the test of legality associated to the minimal abstraction is:
$legal(Paral(\overrightarrow{pv}), L) \Longleftarrow projection(DL(L), \vec{1} - \overrightarrow{pv}) = \emptyset$

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, when 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 multidimensional 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 proposed in the literature [Dhol89],[ShFo91] [DaRo92] [DaRR92] require that the dependences are uniforms 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 allows to get the same set of valid linear schedulings as the dependence distance vector. However, when the computation of the linear scheduling depends on particular distance vector characteristics (uniform) [Dhol89], the dependence cone that approximates the distance vectors cannot be used for defining the linear scheduling but only for testing its validity.

6.1 One-dimensional linear scheduling

The one-dimensional linear scheduling 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 scheduling $\vec{h}$ is valid if the following condition is satisfied:* $\forall \vec{d} \in D, \quad \vec{h} \cdot \vec{d} \geq 1$.

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

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

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

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

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

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

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

Proof: Definitions (1) and (2) associated to the fact that $D \supset DC$ implies $H \supseteq HC$. 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:
$\forall \vec{d_i} \in D, \quad \vec{h} \cdot \vec{d_i} \geq 1$
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 schedulings H (or HC) can be represented by a polyhedron [IrTr88b].

Definition 9 *The set of valid linear schedulings 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 :*

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

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 [Irig88a]. HDV, the set of valid linear schedulings 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
```

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

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 Theorem 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 Theorem 9, we can verify that there is no legal scheduling corresponding to $DDV(L)$.
This example shows that DC is more convenient than D to compute the set of valid linear schedulings, while DDV is not accurate enough.

6.2 Multi-dimensional linear scheduling

The multi-dimensional linear scheduling 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 scheduling is characterized by k linear vectors $(\vec{h_1}, ..., \vec{h_k})$.

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

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

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

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

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

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

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

Proof: This proof is similar to the previous one.

7 Related work

Most dependence abstractions have originally been developed for particular transformations. As such examples, the dependence direction vector was proposed by

M. Wolfe for loop interchanging [Wolf82] and the dependence level by Allen & Kennedy for the vectorization and the parallelization of programs [AlKe87]. 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 few effort has been spent on studying the valid and minimal dependence abstraction for a loop transformation.

In [Wolf90], M. Wolfe has arisen a similar question "What information is necessary 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 [SaTh92] 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 loosing precision.

M. E. Wolf & M. S. Lam [WoLa90] 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 of 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 approximative because the addition, subtraction and multiplication operations defined on this dependence vector are conservative.

A comparison with other works using more precise abstractions than the dependence distance vectors such as the information provided by array data flow analysis [LALa93], [Feau92] and [Feau92] should be added to the results presented here.

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 to a transformation, that contains the minimal information necessary to decide when such a transformation is legal, is defined for three different class of transformations. The minimal abstraction for the reordering transformations: loop reversal, loop permutation, unimodular transformation, partitioning and parallelization have been identified and is respectively DL, DDV, DC, DC and DL. According to the definition, all the abstractions that are more precise than the minimal abstraction associated to 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 partitioning. Moreover, this representation allows to obtain the same set of valid linear schedulings, both one- and multi-dimensional, than with abstraction D without any loss.

Acknowledgments

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

References

[AlKe87] R. Allen, K. Kennedy, "Automatic Translation of FORTRAN Programs to Vector Form," *ACM Transactions on Programming Languages and Systems*, Vol. 9, No. 4. pp. 491-542, Oct. 1987.

[Bane89] U. Banerjee, " A Theory of Loop Permutations," *2nd Workshop on Languages and compilers for parallel computing,* 1989

[Bane90] U. Banerjee, " Unimodular Transformation of Double Loops," *3rd Workshop on Programming Languages and Compilers for Parallel Computing,* Irvine, 1990

[Bane93] U. Banerjee, "Loop transformations for restructuring compilers: the foundations," *Kluwer Academic Editor*, 1993

[Bern66] A. J. Bernstein, "Analysis of Programs for Parallel Processing," *IEEE Transactions on Electronic Computers*, Vol. EC-15, No. 5, Oct. 1966.

[Dhol89] E. D'hollander, "Partitioning and Labeling of Index Sets in Do Loops with Constant Dependence Vectors," *International Conference on Parallel Processing*, 1989

[DaRo92] A. Darte, Y. Robert, "Scheduling Uniform Loop Nests," *Rapport de Laboratoire de l'informatique du Parall'elisme, Ecole Normale Sup'erieure de Lyon'*, No. 92-10, Fev. 1992.

[DaRR92] 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.

[Feau92] P. Feautrier, "Some Efficient Solutions to the Affine Scheduling Problem, Part I, One-Dimensional Time," *International Journal of Parallel Programming*, Vol 21, 1992.

[Feau92] P. Feautrier, "Some Efficient Solutions to the Affine Scheduling Problem, Part II, Multi-Dimensional Time," *International Journal of Parallel Programming*, Vol 21, 1992.

[IrTr87] F. Irigoin, R. Triolet, "Computing Dependence Direction Vectors and Dependence Cones with Linear Systems," *Rapport Interne, Ecole des Mines de Paris*, No. CAI-87-E94,1987.

[Irig88a] F. Irigoin, "Loop Reordering With Dependence Direction Vectors," In *Journées Firtech Systèmes et Télématique Architectures Futures: Programmation parallèle et intégration VLSI*, Paris, Nov. 1988.

[IrTr88a] F. Irigoin, R. Triolet, "Supernode Partitioning," In *Conference Record of Fifteenth ACM Symposium on Principles of Programming Languages*, 1988.

[IrTr88b] 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.

[KMW67] R. Karp, R. Miller and S. Winograd, " The Organization of Computations for Uniform Recurence Equations," *Journal of the ACM,* v. 14, n. 3, pp. 563-590, 1967

[Lamp74] L. Lamport, " The Parallel Execution of DO Loops," *Communications of the ACM 17(2),* pp. 83-93, 1974

[LiPi92] W. Li, K. Pingali, " A singular loop transformation framework based on non-singular matrices," In *5th Workshop on Languages and Compilers for Parallel Computing*, Yale University, August 1992.

[LALa93] D. Maydan, S. Amarasinghe, M.Lam, "Array Data Flow Analysis and its use in Array Privatization," *Stanford Report*, 1993

[Mura71] Y. Muraoka, "Parallelism Exposure and Exploitation in Programs," *PhD thesis*, Dept. of Computer Science, University of Illinois at Urbana-champaign, February 1971.

[SaTh92] V. Sarkar, R. Thekkath, "A General Framework for Iteration-Reordering Loop Transformations," In *SIGPLAN'92 Conference on Programming Language Design and Implementation,* San Francisco, JUNE, 1992.

[Schr86] A. Schrijver, *Theory of Linear and Integer Programming*, John Wiley & Sons 1986.

[ShFo91] 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.

[WoLa90] M.E. Wolf, M.S. Lam, "Maximizing Parallelism via Loop Transformations," In *3rd Workshop on Programming Languages and Compilers for Parallel Computing,* Aug. 1-3, 1990.

[WoLa91] M.E. Wolf, M.S. Lam, "A Loop Transformation Theory and an Algorithm to Maximize Parallelism," *IEEE Transactions on Parallel and Distributed Systems*, Vol.2, No.4, Oct. 1991.

[Wolf82] M. Wolfe, " Optimizing Supercompilers for Supercomputers," *PhD Thesis, Dept. of Computer Science,* University of Illinois at Urbana-Champaign, October 1982.

[Wolf90] M. Wolfe, "Experiences with Data Dependence and Loop Restructuring in the Tiny Research Tool," *Technical Report*, No. CS/E 90-016, Sep. 1990.

[Wolf91] M. Wolfe, "Experiences with Data Dependence Abstractions," In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[Yang93] Y.Q. Yang, "Tests de Dependance et Transformations de programme ", *PhD of University Pierre et Marie Curie* , November 93.