Scanning Polyhedra with DO Loops

Corinne Ancourt François Irigoin*
ENSMP/CRI
77305 Fontainebleau Cedex
France

Abstract

Supercompilers perform complex program transformations which often result in new loop bounds. This paper shows that, under the usual assumptions in automatic parallelization, most transformations on loop nests can be expressed as affine transformations on integer sets defined by polyhedra and that the new loop bounds can be computed with algorithms using Fourier's pairwise elimination method although it is not exact for integer sets. Sufficient conditions to use pairwise elimination on integer sets and to extend it to pseudo-linear constraints are also given. A tradeoff has to be made between dynamic overhead due to some bound slackness and compilation complexity but the resulting code is always correct. These algorithms can be used to interchange or block loops regardless of the loop bounds or the blocking strategy and to safely exchange array parts between two levels of a memory hierarchy or between neighboring processors in a distributed memory machine.

Introduction

Optimizing transformations used by parallelizers to generate efficient code for supercomputers are applied in three steps. Dependence analysis provides conditions for transformations to preserve the sequential program semantics. Then the best transformation in a class is chosen with respect to a particular architecture. Finally the chosen transformation is applied to the program, at the source or machine code level.

This paper deals mainly with the third point which either introduces new conditions for transformation legality, or often is neglected or oversimplified, specifically for intricate transformations requiring new loop bounds generation. Related work is quite limited and is dealt with when appropriate. It is shown that a limited number of simple algorithms, easy to prove, based on linear algebra theory, can be used to solve in a unified way many loop bound generation problems, for different kinds of loop nest transformations. The goal is not to provide algorithms finely tuned for a specific transformation in a specific case but to present powerful but simple tools which can be used to experiment complicated transformations quickly.

Dependence theory and program transformations are now covered by an extensive bibliography and are quite well-known in the parallel programming community. It was not deemed necessary to recall the basics (see [22] or [2] or [32] for instance).

Three different code generation problems are introduced by examples and then formalized in section 1. The first problem is to enumerate with DO loops the tiles generated by a tiling transformation[1]. The second one is to enumerate, using DO loops again, the iterations contained in one tile, or more generally in any iteration set, in any given order such as defined by a loop interchange. The third one is to precisely enumerate the array elements accessed within a nest of loop so as to maintain the memory consistency and to copy them efficiently between two level of a memory hierarchy or between neighboring nodes of a distributed memory machine.

As a side effect, a generalized version of supernode partitioning[15] is introduced to encompass in a unique framework all tiling transformations from strip-mining to combinations of loop skewing, interchanging and jamming.

Then, in section 3, an algorithm based on Fourier's pair-wise elimination is described to compute loop bounds for scanning any polyhedron defined by a system of linear inequalities, such as those obtain after a loop nest linear rescheduling. This algorithm is used again in section 4 to obtain an *approximate* tile enumeration set of loops but care is taken at the tile level to enu-

*E-mail: `<ancourt@ensmp.fr>` `<irigoin@ensmp.fr>`

[1] It is not clear how these transformations should be called. They have also been called *blocking* and *partitioning* transformations, but blocking has no mathematical flavor and partitioning is too general. We hope nobody is going to introduce space *tessellation*!

merate only existing iterations. Finally a more complex algorithm is necessary in section 5 to preserve memory consistency when generating software data prefetch and store between a global and a local memory[4]. Pseudo-linear constraints have to be used to define the exact set of elements to move, the integer affine image of a polyhedron.

1 Overview of the Problems

To efficiently execute programs on multiprocessors, a mix of techniques has been advocated. Consider for instance program 1, which is an abstraction of a PDE solver with a 9 point stencil depicted in figure 1. To reduce the amount of communication with respect to the amount of computation on a distributed memory machine, Terrano suggests, in [27], to use an hexagon tiling.

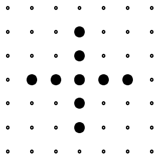

Figure 1: 9 point star PDE stencil

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

Figure 2: Program 1

Such a tiling is shown in figure 3 for program 1's triangular iteration set, assuming N=25. Each dot represents one iteration of the loop body. Each tile contains the dots which fall inside plus dots on some of the boundaries. Although it would have been graphically nicer to translate the boundaries so as to have no dots on them, they are drawn according to the set of constraints used.

These constraints can be strict or non-strict depending on their position with respect to the tile. For instance, tiles from figure 3 are defined by the following system of inequalities S (like *shape*):

$$S = \begin{cases} -3 \leq j < 3 \\ 0 \leq i + j < 6 \\ 0 \leq i < 6 \end{cases}$$

which leads to dissymmetric tiles as shown in figure 4. This dissymmetry is also visible if only non-strict comparisons are used to define a tile:

$$\begin{cases} -3 \leq j \leq 2 \\ 0 \leq i + j \leq 5 \\ 0 \leq i \leq 5 \end{cases} \tag{1}$$

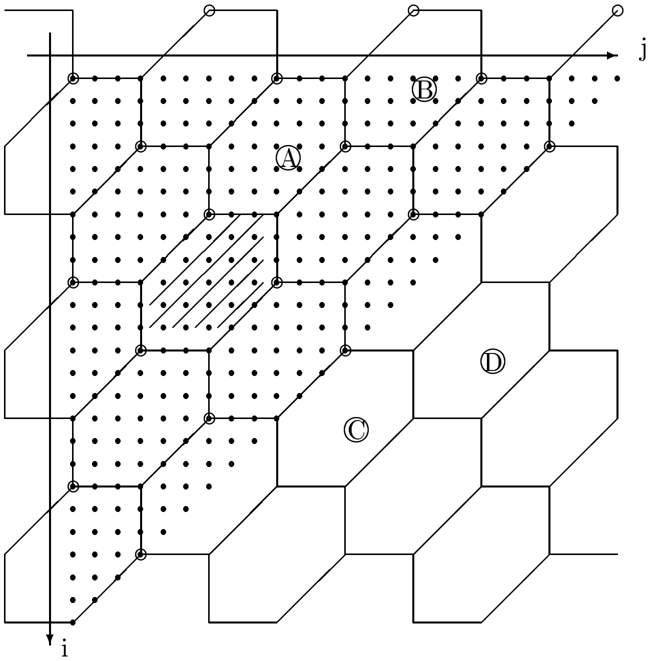

Figure 3: Hexagonal tiling for program 1

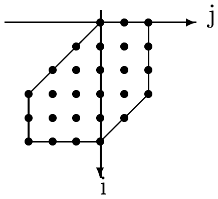

Figure 4: Iteration set for one tile

As in most tiling proposed up to now, all tiles are identical. They define a lattice which can be visualized by choosing an arbitrary origin for them. In figure 3, origins are highlighted by small circles when some iterations fall within the corresponding tile. Notice for instance that tile C contains four elements on its top left boundary, while tile A contains all its 27. Tile B is somewhere in the middle with 15 elements. Tile D is *empty*: none of its elements belongs to the iteration set. It should not be enumerated to cover the iteration set.

Unusual iteration sets, as shown in program 2, may also produce some unexpected empty tiles like tile E in figure 6. Such loop bounds are not very likely to appear in a program, at least if it has not been automatically generated by a formal calculus tool. However, it is important to notice a potential problem: although tile E does intersect with the iteration set, no integer point belongs to that intersection.

```
DO I = 1, 25
  DO J = 28 - 6 * I, 27 - 5 * I
    A(I,J) = PHI(A(I-2,J),A(I-1,J),A(I,J),
                 A(I+1,J),A(I+2,J),A(I,J-2)
                 A(I,J-1),A(I,J+1),A(I,J+2))
ENDDO
```

Figure 5: Program 2

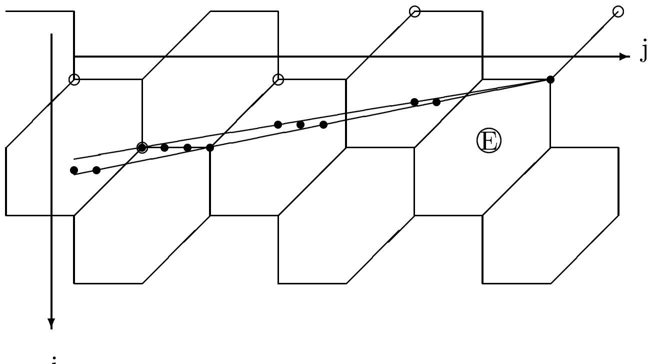

Figure 6: Possible tiling anomaly

1.1 Tile Set

As mentioned above, tile origins belong to a lattice defined by two generator vectors. Many equivalent choices are possible for these vectors and the following pair is chosen to explain how the set of non-empty tiles is derived:

$$L = \begin{pmatrix} -3 & 9 \\ 6 & -9 \end{pmatrix}$$

The origin of the lattice is (arbitrarily) chosen at point $(1,1)$ to simplify the calculations (see figure 3). Let's call t_1 and t_2 the coordinates of the tile origins in their own space, where each integer point can be mapped one-to-one with a tile. These tile coordinates are linked to the iteration coordinates (i_0, j_0) by the following equations:

$$\left\{ \begin{array}{lcl} i_0 & = & 1 - 3t_1 + 9t_2 \\ j_0 & = & 1 + 6t_1 - 9t_2 \end{array} \right.$$

Theses equations can be used in conjunction with a simple system B which is derived from the loop bounds of program 1 and which defines the iteration set:

$$B = \left\{ \begin{array}{l} 1 \leq i \leq 25 \\ 1 \leq j \leq 26 - i \end{array} \right.$$

and with a second system which defines the set of integer points belonging to a tile of a given origin (i_0, j_0) (see system 1 and figure 4):

$$\left\{ \begin{array}{l} -3 \leq j - j_0 \leq 2 \\ 0 \leq i - i_0 + j - j_0 \leq 5 \\ 0 \leq i - i_0 \leq 5 \end{array} \right.$$

to characterize the set of non-empty tiles, tiles (t_1, t_2) which contains at least one iteration (i, j). The tile origin coordinates (i_0, j_0) are eliminated from the system:

$$\left\{ \begin{array}{l} 1 \leq i \leq 25 \\ 1 \leq j \leq 26 - i \\ -3 \leq j - 1 - 6t_1 + 9t_2 \leq 2 \\ 0 \leq i + j + 2 - 3t_1 \leq 5 \\ 0 \leq i - 1 + 3t_1 - 9t_2 \leq 5 \end{array} \right. \qquad (2)$$

To generate efficient loop bounds for t_1 and t_2, i.e. to scan the non-empty tiles, variable i and j must be eliminated. In other words, a four dimensional polyhedron over (i, j, t_1, t_2) must be projected on the subspace (t_1, t_2).

The resulting set of tiles is displayed in figure 7 which translates into the following loop bounds:

```
DO T1 = 0, 8
  DO T2 = (T1+1)/3, 2*T1/3
```

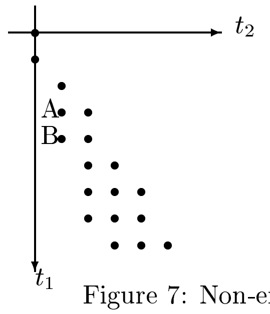

Figure 7: Non-empty tile set

Thus there is a need for an algorithm taking as input an iteration set defined by a system of linear inequalities B and a uniform tiling defined by a lattice L and giving as output loop bounds B_T to scan non-empty tiles.

Let T be the tile space, $(S, \vec{s})$ the system defining the shape of a tile and $\vec{t}$ be the coordinates of a tile. Let I be the iteration space, $(B, \vec{b})$ the system derived from the loop bounds, and $\vec{\imath}$ the coordinates of an iteration. The desired new loop bounds $(B_T, \vec{b}_T)$ are defined by:

$$\{\vec{t} \in T | \exists \vec{\imath} \in I \;\; s.t. \;\; B\vec{\imath} \leq \vec{b} \wedge S(\vec{\imath} - L\vec{t}) \leq \vec{s}\} = \qquad (3)$$

$$\{\vec{t} \in T | B_T \vec{t} \leq \vec{b}_T\}$$

Lots of notations seem to have been suddenly introduced but most of them were introduced with the equations related to program 1 and its hexagonal tiling. Systems of linear constraints are represented as a matrix of coefficients like B and a, possibly symbolic, constant term like $\vec{b}$.

1.2 Iteration Set Local to a Tile

Within a tile, different reorderings are usually compatible with the semantics of the initial program. Loop interchange[29][1] which let the compiler move a parallel loop inwards to use a vector unit is probably the simplest one. The hyperplane method[18][19] and its simplified version obtained by a combination of loop skewing[29][30] and loop interchange, as well as loop permutation[5] are based on more complex change of bases.

These transformations must map integer points onto integer points on a one-to-one basis to preserve the iteration set and the program semantics. The change of basis matrix U must be unimodular as is explained by

Banerjee for combination of loop interchanges[6], and more generally for global code generation for nested loops[16] and loop reordering[12][13]. The same criterion is also used to define more general linear loop transformations[28].

For instance, it might be decided to execute iterations of program 1 on a front by front basis, where iterations belonging to the same front are executed in parallel. This transformation is neither always legal nor desirable, but this is not the point here.

Fronts are shown on one tile in figure 3 by lines at 45°. This new execution ordering can be generated by replacing the initial basis. The new one should have a first basis vector linking a front to the next one and a second basis vector linking an iteration to a neighboring one that can be executed in parallel.

The following pair of vectors U_1 and the resulting change of coordinates meet these conditions:

$$U_1 = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix} \qquad \begin{cases} i = l_2 \\ j = l_1 - l_2 \end{cases}$$

and define a new local iteration set for a tile, shown in figure 8, whose constraints are:

$$SU_1 = \begin{cases} 0 \le l_2 \le 5 \\ -3 \le l1 - l_2 \le 2 \\ 0 \le l_1 \le 5 \end{cases}$$

Since l_1 is bounded it is easy to derive new loop bounds

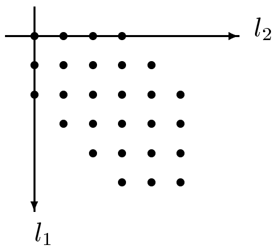

Figure 8: Skewed Iteration Set

for such an execution pattern:

```
DO L1 = 0, 5
  DO L2 = MAX(0, L1 - 2), MIN(5, L1 + 3)
    ...
```

However, it is not always that easy. Let's assume the desired change of basis is defined by:

$$U_2 = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix} \qquad \begin{cases} i = 2l_1 + l_2 \\ j = l_1 + l_2 \end{cases}$$

and define a new local iteration set for a tile, whose constraints are:

$$SU_2 = \begin{cases} -3 \le l_1 + l_2 \le 2 \\ 0 \le 3l_1 + 2l_2 \le 5 \\ 0 \le 2l_1 + l_2 \le 5 \end{cases}$$

These constraints cannot be direcly used to generate loop bounds because they all use l_1 and l_2 whereas the bounds of the external loop should only refer l_1.

Such unsuitable constraints are sometimes due to the partitioning. Let's assume the best partitioning for program 1 were a square partitioning, where squares are defined by diagonals and anti-diagonals. Each constraint would contain either $i - j$ or $i + j$ and it would not be possible to use them as loop bounds, even if the initial iteration ordering was preserved within each tile.

Thus there is a need for an algorithm taking as input a polyhedron defined by a system of inequalities and an ordered set of variables, and giving as output the same polyhedron defined by a new system of inequalities such that each variable is bounded by a min and a max expressions which only contain variables of higher rank in the ordering. There are no constraints on non ordered variables. They are used symbolically as part of the constant term. Such a system can be used to directly derive loop bounds.

The input polyhedron can result either of a tiling transformation and be the shape of tile S or of a change in the execution ordering and be the product BU of the initial iteration domain B by a change of basis matrix U or of a combination of both and be SU.

1.3 Array Elements Accessed Within a Tile

Program 1 was tiled as shown in figure 3 to execute on a multiprocessor machine. Different kinds of multiprocessors exist but whether the machine has a shared global memory and fast local memories or it is a distributed memory machine, there is a need to define which array elements are necessary to execute one tile and where they can be found.

As a first example, figure 9 shows which elements of array `A` should be available in the local memory of a processor before it can execute one hexagonal tile. Note that this set is non-convex. Four elements are missing in the bottom-left and top-right corners.

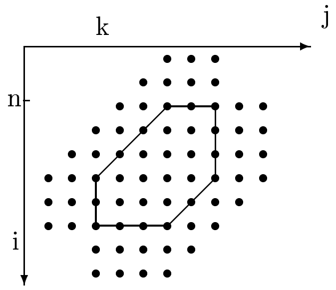

Figure 9: Non-convex array access pattern

As a second example, consider the following contrived code as defined by Jalby & al. in [10] and the resulting accessed set shown in figure 10. Only a small subset of accessed elements is displayed to show clearly the non-convexity. Since linear loop bounds define convex iteration sets, these array subsets cannot be easily enumerated with DO loops.

```
DO I = 1, 10
    DO J = 1, 20
        DO K = 1, 39
            A(3*I+K, J+K) = ...
        ENDDO
    ENDDO
ENDDO
```

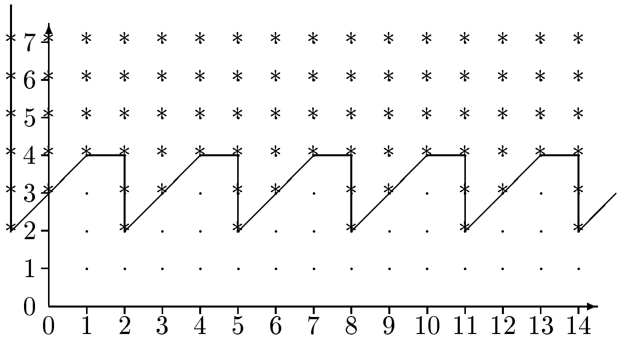

Figure 10: Non-convex array access pattern

As it might be necessary to move these sets exactly to preserve memory consistency and efficiently so as not to waste memory or interconnection network bandwidth, it would be useful to have an algorithm taking as input an iteration set defined by a system of linear inequalities B and a set of references R_1, R_2,... to an array `A` and giving as output a set of possibly non-linear loop bounds C and one array reference R such that each array element accessed by the input is accessed once and only once by the output.

$$\{\vec{a} \in A | \exists \vec{\imath}\,' \in I' \ \ s.t. \ \ C(\vec{\imath}\,') \ \wedge \ \vec{a} = R\vec{\imath} + \vec{r}\} = \{\vec{a} \in A | \exists \vec{\imath} \in I \ \ s.t. \ \ B\vec{\imath} \leq \vec{b} \wedge (\vec{a} = R_1(\vec{\imath}\,') + \vec{r}_1 \vee \vec{a} = R_2(\vec{\imath}\,') + \vec{r}_2)\}$$

For distributed memory machine, a pseudo-linear mapping function can also be given as input to restrict the elements accessed by the output loop and reference to those available on a given processor.

1.4 Loop Bound Generation Issues

A few points should be outlined before we proceed. First of all, the bounds of a loop nest define special constraints: each loop index can only appear in inner loop bounds and, as a result, the corresponding constraint matrices are row-echelon. Thus most systems of linear inequalities cannot be directly used to generate loop bounds.

Second, the projection, or more generally the affine image, of a polyhedron is not a polyhedron as is shown by figure 10. Since linear loop bounds generate convex sets, they cannot be used to enumerate such sets but they can be used to enumerate a superset.

Failing to eliminate variables by projection results in extra loops. For instance, a two dimensional array referenced within three nested loops should be copied with only two loops. At least one of the initial loop indices should be eliminated.

Elimination is not always possible. It may be necessary to use pseudo-linear constraints with modulo conditions, to add a test to guard the loop body and, if worse comes to worse, to generate an extra-loop.

Vector and matrix notations are used because the algorithms do not depend on the number of loops or variables. Examples are 2-dimensional only because it is almost impossible to draw properly 3-D cases.

2 Lattices for Tiling Transformations

It is assumed above that a tiling is defined by a lattice L and a set of constraints S. L and S are not independent and should be derived automatically for each automatic tiling method.

Many multidimensional blocking schemes have been proposed: supernode partitioning[15], loop tiling[31][33], loop quantization, mitred quantization[3], strip-mine and interchange, unroll and jam, wavefront blocking[24]. But up to now, these tiling methods share a few common properties.

Tiles should be equal up to an integer translation, so as to have some implicit load balancing and a unique code for all blocks. The tile space T should be an integer vector space to make block numbering easy and the tile set should be a polyhedron to remain in a linear framework. A linear relation between an iteration's coordinates, its block coordinates and its local coordinates is useful to test tile membership and to generate loop bounds for each tile.

2.1 Using Multidimensional Hyperplane Partitioning

The simplest way to partition a space in tiles is to use a family of regularly spaced hyperplanes. A unique vector $\vec{h}$ can define the hyperplanes orientation, their spacing via its norm and their ordering via its direction. This simple transformation was called hyperplane partitioning[15] and is a generalization of strip-mining with no restriction on the stripe direction.

A few hyperplane partitionings can be combined to define smaller tiles. Their definition vectors can be put together in a matrix H^T and the tile coordinates $\vec{t}$ of an iteration $\vec{\imath}$ are given by a pseudo-linear many-to-one mapping using the floor function $\vec{t} = \lfloor H^T\vec{\imath} \rfloor$. H must be free and H^T must meet some condition with the dependencies for the transformation to be legal, but this does not matter here.

We claim that any tile obtained by one of the techniques above can also be defined by a unique matrix H and that it is thus sufficient to be able to derive code from any matrix H to generate code for these techniques. However, most tilings which are applied by hand, like the hexagon tiling shown in figure 3, cannot be that simply generated. They can be dealt with only if L and S are explicitly given.

H^T defines a surjection from the iteration space I onto the integer tile space T. Each integer point $\vec{t}$ of T must be reached for some integer point $\vec{\imath}$ of I. To relate $\vec{t}$, the coordinates of tile containing $\vec{\imath}$, $\vec{\imath}$ itself and the local coordinates $\vec{l}$ it is necessary to get rid of the non-linearity due to the floor operator and to introduce some lattice L.

2.2 Examples

For instance, strip-mining loop J from program 1 by a factor of 3 is equivalent to defining ${H^T}_1$ while moving the I loop at innermost position can be expressed by ${H^T}_2$:

$${H^T}_1 = \frac{1}{3}\begin{pmatrix} 0 & 1 \\ 3 & 0 \end{pmatrix} \qquad {H^T}_2 = \frac{1}{3}(0\ 1)$$

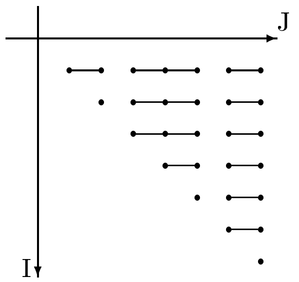

Figure 11: Tiling H^T_1

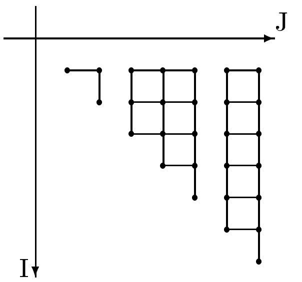

Figure 12: Tiling H^T_2

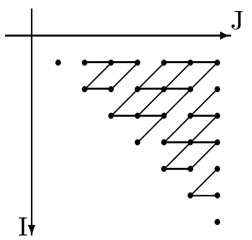

Figure 13: Tiling ${H^T}_3$

2.3 Shape Constraints

The shape system of constraints S is easy to derive from H^T. By definition a point $\vec{\imath}$ belongs to a tile of origin $\vec{\imath}_s$ if:

$$\vec{0} <= H^T(\vec{\imath} - \vec{\imath}_s) < \vec{1}$$

Let k be the smallest integer such that kH^T is an integer matrix. Then S is defined by the the following matrix and vector:

$$\begin{pmatrix} kH^T \\ -kH^T \end{pmatrix} (\vec{\imath} - \vec{\imath}_s) \leq \begin{pmatrix} (k-1)\vec{1} \\ \vec{0} \end{pmatrix}$$

2.4 Pseudo-Inverse Matrix

A solution to find a *linear* relation between $\vec{\imath}$ and $\vec{t}$ is to compute the pseudo-inverse of H^T and to introduce an auxiliary variable $\vec{\lambda}$:

$$\vec{s} = H^T H\vec{\lambda} \qquad \vec{\imath} = H\vec{\lambda} = H(H^T H)^{-1}\vec{s}$$

Since H is free, $H^T H$ can be inverted but $\vec{\imath}$ is not necessarily an integer point of the iteration space I. For instance the following tiling ${H^T}_3$ which defines a 1-D space of antidiagonal stripes of width 3 in a 2-D space leads to a non-linear equation which produces non integer $\vec{\imath}_s$ for odd $\vec{t}$.

$${H^T}_3 = \frac{1}{3}(1\ 1) \qquad \vec{\imath}_s = \frac{1}{2}\begin{pmatrix} 3 \\ 3 \end{pmatrix} \vec{t}$$

Tilings ${H^T}_1$, ${H^T}_2$ and ${H^T}_3$ for program 1 are shown in figures 11, 12 and 13. Elements belonging to the same tile are connected.

2.5 Smith Normal Form

Smith proved that any integer linear application can be described by a quasi-diagonal matrix[2] called D like *diagonal* if adequate unimodular changes of basis P and Q are performed in the domain and image spaces[3]. Multiplying H^T by k as above to get the smallest proportional integer matrix and preserving a transpose sign on D to remember its usage, we get:

$$H^T = \frac{1}{k} P D^T Q$$

We may assume that D^T has no null rows because the corresponding dimensions would be useless in the tile space but it may have null columns. If there are no null columns, D^T is an invertible square matrix and it defines bounded tiles which were called *supernode* in [15]. When D^T has only one non zero coefficient, it defines a basic hyperplane partitioning. Using the pseudo-inverse as in the previous section but in the Smith bases P and Q, $\vec{\imath}\,'$ and $\vec{t}\,'$ are related by the following equations:

$$\vec{\imath}\,' = kD\left(D^T D\right)^{-1}\vec{t}\,'$$

[2] See [26] for a precise definition.

[3] This means that any multidimensional hyperplane partitioning can be seen as a simple set of strip-mined loops if the proper basis are chosen.

The last coordinates of $\vec{\imath}\,'$ are all zero and the first ones are equal to $k/d_i \times s'_i$ where d_i is the i-th diagonal term of D. To get integer coordinates for all s'_i, k must be a multiple of d_i's LCM: this means that, for a given direction, not all hyperplane spacings are possible when each tile must contain as many points as any other one. The two unimodular bases P and Q will preserve these integer coordinates. Using the changes of basis $\vec{s} = P\vec{s}\,'$ and $\vec{\imath}\,' = Q\vec{\imath}$, the relation between $\vec{\imath}_s$, the tile origin coordinates in the iteration space, and $\vec{t}$, the tile coordinate in the tile space, becomes:

$$\vec{\imath}_s \;=\; kQ^{-1}D\big(D^TD\big)^{-1}P^{-1}\vec{s} \qquad (4)$$

and the lattice is defined by:

$$L \;=\; Q^{-1}D\big(D^TD\big)^{-1}P^{-1}$$

With example $H^T = (1/3, 1/3)$, equation (4) defines an integer origin for each stripe, i.e. a proper stripe numbering.

$$Q = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \qquad P = (1) \qquad D^T = (1\ 0) \qquad k = 3$$

$$\vec{\imath}_s = 3\begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix}\left(\begin{pmatrix} 1 & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix}\right)^{-1}(1)\vec{t}$$

$$= \quad \begin{pmatrix} 0 \\ 3 \end{pmatrix}\vec{t}$$

With supernode partitioning[15], D is a square invertible matrix and equation (4) becomes much simpler and implies that $\left(H^T\right)^{-1}$ be integer:

$$\vec{\imath}\,' \;=\; k\left(D^T\right)^{-1}\vec{s}\,' \;=\; \left(H^T\right)^{-1}\vec{s}\,'$$

Thus it is possible to derive a lattice matrix L and a tile shape S from any legal multidimensional hyperplane partitioning H^T and from any tiling obtained automatically.

3 Scanning a Polyhedron

A naive solution would be to project the polyhedron on each basis vector to find two loop bounds for each loop and to add a test to the loop body to make sure the iteration has to be executed.

Very specific algorithms were developped to handle special changes of basis, the permutations obtained when interchanging loops. Different kinds of loop bounds are distinguished: *rectangular* ones, *triangular* ones and so one. Each case is handled by a specific algorithm.

Another algorithm, solving the same problem for two dimensional spaces, is outlined in [28]. It is based on generating systems. Images of extreme vertices are extreme vertices of image polyhedron and loop bounds can be derived from them. This algorithm is not as general as the following one because it assumes that the generating system is known, which is usually true when loop bounds are available.

The basic idea here is to use a projection algorithm to find loop bounds for each dimension. Fourier pairwise elimination[9] cannot be used without care because it is only valid for rational and real polyhedra and provides a simple inclusion instead of an equality for integer points.

```
row_echelon(S)
/* S, S' and SP are lists of parametric linear
   constraints over a n-dimensional space*/
S' := SP := S;
/* compute projections in reverse order to avoid
   redundant computations */
for i := n-1 to 1 by -1 do
    SP := fourier(SP, i+1);
    S' := union(S',SP);
done
/* sort constraints in S' by increasing constraint rank */
S' := sort(S');
/* try to eliminate redundant constraints innermost
   first because they would be executed more often */
for i := number_of_constraints(S') to 1 by -1 do
    /* let S'(i) be the i-th constraints in S' */
    if not last(S',S'(i)) then
        S'(i) := complement(S'(i));
        if fourier_empty_p(S') then
            /* 0 is the trivial constraint 0 <= 0 */
            S'(i) := 0;
        else
            /* restore its original value */
            S'(i) := complement(S'(i));
        endif
    else
        /* preserve S'(i) as a unique lower or upper
           loop bound */
    endif
    S'(i) := normalize(S'(i),rank(S'(i)));
done
return S';
end
```

Figure 14: Algorithm *row_echelon*

Let's detail algorithm *row_echelon* informally. System S' always defines the same set of integer points. It is initialized as S and is then augmented by the projections of S on smaller and smaller subspaces. The last projection provides constant bounds for the outermost loop, and the next to the last provides bounds using only the outermost index for the second loop, and so on.

Fourier-Motzkin pair-wise projection algorithm is used and useless points can be added at any time. However the initial system S is included in S' and useless iterations generated by the outer loops result in empty ranges for the inner loops.

In general, the total number of constraints in S' after this first phase is huge. Hopefully many of them are redundant. However, they can not all be eliminated because at least one MIN and one MAX constraints are needed for each index variable. Hence, the test with `last`.

The redundancy test does not have to be exact. The better it is, the more constraints are eliminated. However, the generated loops are always correct when too many constraints are used. A very simple redundancy test, called `fourier_empty_p` based on Fourier-Motzkin feasability test, is shown. Faster tests should be used.

Finally the constraints must be normalized to have $+1$ or -1 as coefficient for the corresponding index variable.

A first version of this algorithm is presented in [12]. It uses the same idea but useless projections are performed. It was implemented in an experimental phase of PTRAN to compute loop bounds after loop interchange and the hyperplane method[13].

4 Tiling

Algorithm *tiling* is very simple once *fourier* and *row_echelon* are available. The lattice relations between the tile space and the iteration space and the constraints defining a tile shape are used to build large linear systems containing all necessary inequalities

The key point is that an approximate algorithm, Fourier-Motzkin pairwise projection, is used to compute the bounds to scan the tiles and that exact bounds are used to scan iterations within each tile.

To show correctness is to prove equality (3). Right to left inclusion is obvious because the iteration set conditions are used in each tile and because *row_echelon* does not perform any projection. Left to right inclusion is due to the proper inclusion for *fourier*: there may be too many tiles in B_S but each iteration $\vec{\imath} \in B$ has a tile in B_S.

Two kinds of optimization can be performed to decrease the control overhead due to `BS`. First, inequalities redundant with respect to `BT` can be eliminated. In the most favorable case, it means that the iteration set bounds B are not used at all. Second, a set of tests can also be computed to distinguish between full tiles whose integer points must all be computed (once again, B can be ignored) and partial tiles which intersect one of the iteration set boundaries.

Note that the system `B'` is likely to be much more intricate than a system directly derived from usual linear loop bounds (see for instance figure 7) and that algorithm `row_echelon` must cope with it. Note also that system `BS` is likely to be complicated and that some kind of loop reordering might have to be applied to exploit, for instance, a multiprocessor with vector units. Once again, algorithm designed to only deal with so-called *real* programs are likely to fail when their input code has been automatically generated. .

```
tiling(B, HT)
/* B, BS and BT are arrays of constraints, L and S
define the tiling */

/* compute loop bounds BS for tile enumeration */
build B' over T x I according to system (3)
```

$$\begin{pmatrix} 0 & B \\ -SL & S \end{pmatrix} \begin{pmatrix} \vec{t} \\ \vec{\imath} \end{pmatrix} \leq \begin{pmatrix} \vec{b} \\ \vec{s} \end{pmatrix}$$

```
/* eliminate unwanted variables using standard projection */

for all i in I do
    B' := fourier(B',i);

/* apply previous algorithm to get a system of loop bounds */

BT := row_echelon(B');

/* compute loop bound BT for one tile's iteration
enumeration */

build system B'' using iteration set and tile
membership conditions:
```

$$\begin{pmatrix} B \\ S \end{pmatrix} \vec{\imath} \leq \begin{pmatrix} \vec{b} \\ \vec{s} + SL\vec{t} \end{pmatrix}$$

```
BS := row_echelon(B'');
return BS, BT;
end
```

Figure 15: Algorithm *tiling*

5 Affine Image

The key idea is to build a large polyhedra in the product of the domain and image spaces and to eliminate variables from the domain space in the constraints without introducing new image elements. First the application equations can be solved using Hermite or Smith Normal form. Then inequalities have to be dealt with.

5.1 Legal Integer Pairwise Elimination

Great care must be taken not to modify the affine image integer point set when one variable from the domain polyhedra is eliminated. To preserve these image points, integer divisions must be introduced.

$$Let\ \ E_j \ =\ \alpha_j \ +\ \sum_{l=1, l\neq k}^{n} a_{jl}\ i_l$$

be an integer linear expression, R a set of constraints, a_{jl} and α_j integers, a_{pk} and a_{qk} positive integers and i_k a variable.

Theorem 1

$$Let \quad S \;=\; \left\{ \begin{array}{llll} E_p + a_{pk}\, i_k \le 0 & (C1) \\ E_q - a_{qk}\, i_k \le 0 & (C2) \\ \quad R & \end{array} \right.$$

$$and \quad S_{/k} \;=\; \left\{ \begin{array}{ll} \frac{E_q + a_{qk} - 1}{a_{qk}} \le \frac{-E_p}{a_{pk}} & (C12) \\ \quad R & \end{array} \right.$$

where $S_{/k}$ is derived from S using integer divisions[4] *to eliminate i_k. Then:*

$$proj(\lfloor S\rfloor^2, i_k) \;=\; proj(\lfloor S_{/k}\rfloor, i_k)$$

.

The pseudo-linear system $S_{/k}$ does not necessarily define a polyhedron since integer divisions may introduce *holes* into a convex polyhedron. Therefore, this elimination operation is not an internal operation. Thus, we introduce sufficient conditions to eliminate variable by simple pair-wise elimination without modifying the projection. They will be used to eliminate as many variables as possible while preserving the projection and the system linearity. These conditions are given in the next two theorems.

Theorem 2

$$Let \quad S \;=\; \left\{ \begin{array}{l} E_p + a_{pk}\, i_k \le 0 \\ E_q - a_{qk}\, i_k \le 0 \\ \quad R \end{array} \right.$$

$$and \quad S' \;=\; \left\{ \begin{array}{l} a_{pk}\, E_q \le -\, a_{qk}\, E_p \\ \quad R \end{array} \right.$$

where S' is obtained from S using the pair-wise elimination method and R is any system.

$$a_{pk} = 1 \vee a_{qk} = 1 \Longrightarrow$$

$$proj(\lfloor S\rfloor, i_k) = \lfloor proj(S, i_k)\rfloor = proj(\lfloor S'\rfloor, i_k)$$

This theorem is illustrated by the following figures.

In this first figure, the pairwise elimination of the variable i_2 from the polyhedron $R \bigcap C1 \bigcap C2$ introduces an additional integer point to the image polyhedron (polyhedron after the projection represented by $\bigcirc$ on the axis i_1). Indeed, the result of the pairwise elimination method is the segment [4,27], when any integer point belonging to the polyhedron $R \bigcap C1 \bigcap C2$ is the projection of point 4. The projection of integer points is different from the projection of real points because the difference between the slopes of the 2 lines is too small for containing an integer point for $i_1 = 4$.

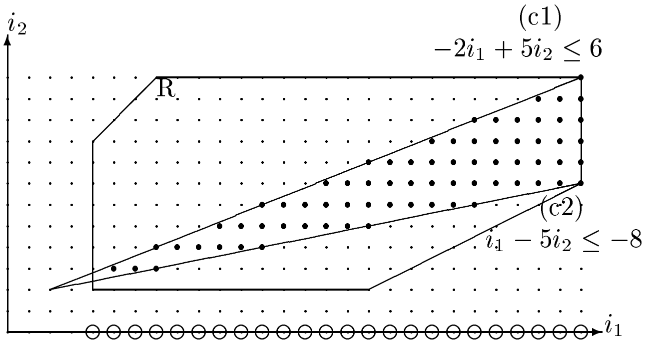

Illustration of theorem 2

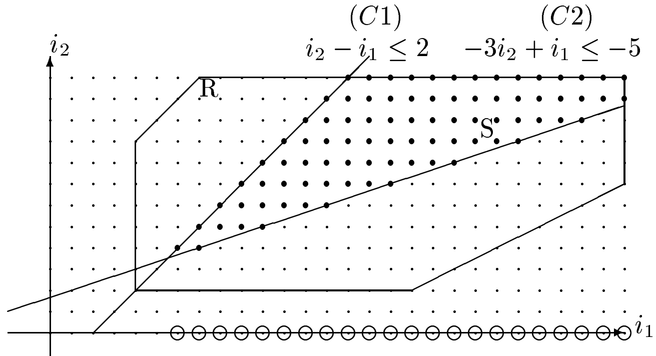

Illustration of theorem 2

The second figure shows that if one line $(C1)$ has a slope of 1[3] , each point of the image polyhedron is actually the projection of the an integer point of the polyhedra $R \bigcap C1 \bigcap C2$, because $(C1)$ contains one integer point for each value of the projection set.

When the constraint $(C12)$ resulting from integer pair-wise elimination can be proved redundant with R, it is not preserved in the system. A linear condition to determine that $(C12)$ is redundant is now presented.

Theorem 3

$$Let \qquad S \;=\; \left\{ \begin{array}{ll} E_p + a_{pk}\, i_k \le 0 & (C1) \\ E_q - a_{qk}\, i_k \le 0 & (C2) \\ \quad R & \end{array} \right.$$

(C) $a_{pk}E_q + a_{pk}a_{qk} - a_{pk} \le -a_{qk}E_p$ *redundant*[4] *for* R

$$\Longrightarrow \; proj(\lfloor S\rfloor, i_k) = proj(\lfloor R\rfloor, i_k)$$

.

In other words, if $(C12)$ is the constraint resulting from i_k's elimination in $C1 \bigcap C2$ and if constraint C is met, then $(C12)$ is redundant for $\lfloor S\rfloor$. This theorem is illustrated by the following figure:

[4] Different definitions exist for non-positive integers. Here the remainder is always assumed to be positive.

[2] $\lfloor S\rfloor$ is the set of integer points belonging to S

[3] that is equivalent of having a coefficient of 1 for the variable i_2 to eliminate

[4] we use the redundancy criteria described in [7]

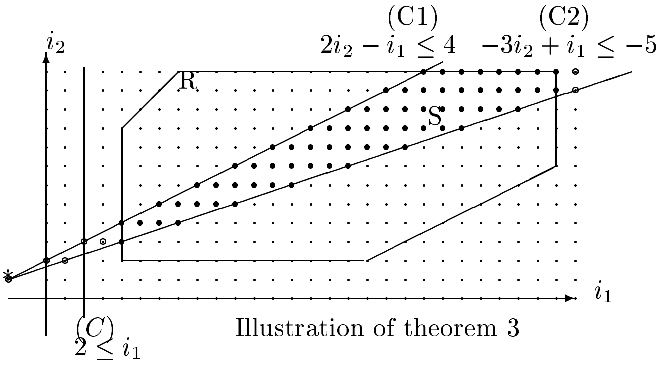

Illustration of theorem 3

In this example, system S is:

$$S = \begin{cases} 2i_2 - i_1 - 4 \le 0 & (C1) \\ -3i_2 + i_1 + 5 \le 0 & (C2) \\ R & \end{cases}$$

and condition

$$(C) \;\; 2(i_1+5) + 6 - 2 \le -3(-i_1 - 4) \implies 2 \le i_1$$

which is obviously redundant with R.

The figure shows that, if the constraint (C) is redundant for R, then $(C12) = proj(\lfloor C1 \bigcap C2 \rfloor, i_k)$ is redundant for $proj(\lfloor S \rfloor, i_k)$ and $proj(\lfloor S' \rfloor, i_k) = proj(\lfloor R \rfloor, i_k)$.

5.2 Using Integer Divisions

When none of these two conditions is met, it is still possible to eliminate variable i_k by combining the two inequalities and by introducing integer division *(theorem 1)* to get $S_{/k}$.

Assume that i_l is one variable of the inequality $(C12)$ appearing either in the left hand side of $(C12)$ or the right one. It is possible to rewrite $(C12)$ as $a_{hl} i_l \le E_l$ where E_l is a pseudo-linear function of E_p, E_q, a_{pk}, and a_{qk}.

$$\textbf{Example1}: \;\; S1 = \begin{cases} -i_3 + 3\,i_1 \le 3 \\ i_2 - 3\,i_1 \le -2 \\ -i_2 \le 0 \end{cases}$$

$$\iff \;\; S1 = \begin{cases} i_2 + 2 \le 3\,i_1 \le 3 + i_3 & (I) \\ -i_2 \le 0 \end{cases}$$

Inequality (I) is equivalent to $\frac{i_2+4}{3} \le \frac{3+i_3}{3}$ which can be rewritten as $i_2 + 4 \le 3(\frac{3+i_3}{3}) + 2$, or $i_2 \le 3(\frac{i_3}{3}) + 1$

$$\textbf{Example2}: \;\; S2 = \begin{cases} -i_2 + 3\,i_1 \le 1 \\ i_2 - 3\,i_1 \le 0 \\ -i_2 \le 0 \end{cases}$$

$$\iff \;\; S2 = \begin{cases} i_2 \le 3\,i_1 \le i_2 + 1 & (I) \\ -i_2 \le 0 \end{cases}$$

Inequality (I) is equivalent to $\frac{i_2+2}{3} \le \frac{i_2+1}{3}$ which cannot be simply transformed in a linear constraint on i_2 or in a constraint containing integer division. In the other hand, it can be transformed in constraints using the function Modulo: $(I) \iff Mod(i_2, 3) = 0 \; or \; 2$

This elimination possibility is important. Without it, some occurrences of variables to project could not be eliminated using pair-wise elimination without modifying the projection. On the other hand, constraints expressions are more complicated and contain integer divisions.

The algorithm to compute the pseudo-linear constraints defining exactly the image polyhedra which characterizes array elements referenced in a loop nest is now briefly outlined.

5.3 Computation of the Image Polyhedra Constraints

The set of referenced array elements is the affine image of the index set I by the access function R. The image polyhedron is computed from index set I by a change of basis from index basis $\vec{i}$ to image basis $\vec{t}$. If the dimension d of the affine image is less than dimension n of iteration space, then only d loops are enough to scan the image polyhedron. Thus, $n-d$ variables can be eliminated out of the image polyhedron constraints defined by system S.

The first step of the algorithm consists in projecting as many useless variable occurrences as possible using the pair-wise elimination method for constraints satisfying conditions of theorems 2 and 3.

In the second step, redundant constraints are eliminated. All redundant constraints on a useless variable can be eliminated if the variable does not appear in a constraint of superior rank. At least two constraints on useful variables must be kept to generate loop bounds.

Finally, integer divisions are introduced in constraint expressions. The remaining useless variables are eliminated out of S by combining pairs of constraints and by introducing integer divisions, if the variable does not appear in a constraint of superior rank.

The final system may still contain some useless variables, because we did not manage to prove otherwise. Occurrences of these variables in the constraints express, like integer divisions, the non convexity of a polyhedron affine image. However, this never happened in any tested case.

5.4 Generation of the Nested Loops

Let SI be the set of constraints computed by the previous algorithm for the image polyhedra. Let SI_1 be its linear subset, and $SI_2 = SI - SI_1$ its pseudo-linear subset.

To generate the nested loops defining the image polyhedra, the algorithm *row_echelon* described in section3 is applied to the SI_1. Inequalities of SI_2 are added as loops bounds or used in a guard if the variable of higher rank in the inequality appears in both inequality sides.

5.5 Examples

Let's consider the program proposed by Jalby & al. in [10], figure 10, the nested loops generated from the computed image polyhedra are:

```
DO T1 = -29, 17
DO T2 = MAX(4, 4+3*IDIV(-T1, 3)),
           MIN(60, 48+3*IDIV(2-T1, 3))
   A(T2, T1+T2) = ...
```

The new nested loops access only once each array element referenced by `A` in the original program. The number of generated loops is minimal and equal to the affine image dimension.

The case where we have several references to the same array having uniform dependences is interesting because it can be dealt with the case where we have only one reference. In the following examples, the two programs reference the same set of elements of `A`. To compute the set of array elements referenced in the first program is equivalent to evaluate the referenced array elements of the second.

```
DO I = 1, N                  DO I = 1, N
DO J = 1,M                   DO J = 1, M
 ... = A(I,J)                DO X = 0, 1
     + A(I+1,J)              DO Y = 0, 1 - X
     + A(I-1,J)              ...= A(I+X-Y,J)
ENDDO                        ENDDO
```

To compute the set of elements used in the execution of one tile, represented in figure 9, the previous transformation is used before applying the preceding algorithms. The nested loops generated from the computed image polyhedra is:

```
DO I=n-2,n+7
DO J= MAX(k-2, n+k+1-I,k-n -7 +I),
                     MIN(7+k,n+k+10-I)
IF (IDIV(I-n-2*J+2*k-6,2) <= IDIV(7-I+n,2)
&& IDIV(-I+n+2*J-2*k-11,2) <= IDIV(2-n+I,2))
    ..... =     A(I,J)
ENDDO
```

Conclusion

Three different algorithms generating loop bounds for three different classes of program transformations have been presented. They are general because they encompass most transformations and combination of transformations on nests of loops. They are also general with respect to loop bounds. They do not require rectangular, triangular or trapezoïdal loops. Any polyhedral iteration set is acceptable. `MIN` and `MAX` operators may appear in bound expressions. Their generality and simplicity make complicated optimizations easy to try.

Additional tests and different code versions could be generated using the same algorithms if ultimate optimization is sought. The same technique could be extended to deal with loop alignment and loop peeling. It would also be interesting to use it to generate code for non-perfectly nested loops by adding guards to non-innermost statements and by using the new guards to move the statements out of the loops using a redundancy test.

It would be interesting to use these algorithms to generate code for the partitioning strategy studied and successively improved by [21], [23], [25] and [11]. This partitioning assumes that independent sets of iterations exist because the dependence vector are long enough. Set origins are all in one tile and each set is defined by a lattice. This is dual to the tiling problem studied here, since the outer loops are used to scan one tile while the inner loops are used to scan the intersection of a lattice and of an iteration set.

The correctness of these algorithms is easy to prove because they are based on well-known linear algebra concepts. A tradeoff between accuracy in redundance checking and complexity is possible, but correctness is always preserved. Worst case complexity is clearly exponential in the space dimensions and constraint number but it is not too much of an issue; the number of loop nests increases linearly with the program size, and not like its square as for dependence testing. Also an exponential worst case complexity may produce a polynomial average complexity as is observed in linear programming for the simplex algorithm. If bound constraints are diagonal (i.e. rectangular loops) or if bounds coefficients are 1 or -1 (i.e. rectangular loops), complexity is not a problem.

Our algorithm usefulness may be questionned since real programs do not contain complicated loop bounds and subscript expressions. Faster algorithms, dealing with simple usual cases, might be preferred. But however simple real cases are, compiler algorithms should be able to cope with unexpected ones, at least by detecting them. Also, however complex these algorithms seem to be, they are pretty straightforward and easy to write and shorter in lines of code than others. Moreover the tricky cases they can handle may let people try new optimizations, based for instance, on 3-D tilings, and, mainly, may occur when a suite of program transformations are applied. The initial code is simple but it does not stay so during the restructuring process.

Algorithms *row_echelon* and *tiling* were first implemented in an experimental phase of the IBM PTRAN system. A second implementation is underway in the PIPS project [17]. Algorithm *image* is implemented on a stand-alone basis. It should be used in a second phase of the PIPS project to generate data exchanges between local memories and a global memory.

References

[1] J. R. Allen, K. Kennedy, *Automatic Loop Interchange*, SIGPLAN'84 Symposium on Compiler Construction, SIGPLAN Notices, 19, 1984

[2] R. Allen, D. Callahan, K. Kennedy, *Automatic Decomposition of Scientific Programs for Parallel Execution*, ACM Symposium on Principles of Programming Languages, Munich, 1987

[3] A. Aiken, A. Nicolau, *Loop Quantization: An Analysis and Algorithm*, Tech. Rep. 87-221, Cornell University, 1987

[4] C. Ancourt, *Génération de code pour multiprocesseurs à mémoires locales*, Thèse de l'Université Pierre et Marie Curie, in progress

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

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

[7] M. C. Cheng, *General Criteria for Redundant and Nonredundant Linear Inequalities*, Journal of Optimization Theory and Applications, vol. 53, No 1, April 1987.

[8] R. J. Duffin, *On Fourier's Analysis of Linear Inequality Systems*, Mathematical Programming Study 1, North-Holland, 1974

[9] J.B.J. Fourier, *Analyse de travaux de l'Académie Royale des Sciences, pendant l'année 1824, partie mathématique*, Histoire de l'Académie Royale des Sciences de l'Institut de France, 1827.

[10] K. Gallivan, W. Jalby and D. Gannon, *On the Problem of Optimizing Data Transferts for Complex Memory Systems*, Proceeding of the ACM Int'l Conf. on Supercomputing, St-Malo, 1988.

[11] E. D'Hollander, *Partitioning and Labeling of Index Sets in DO Loops with Constant Dependence Vectors*, 1989 Int'l Conference on Parallel Processing, pp. II-139, II-144 (Aug. 1988)

[12] F. Irigoin, *Code Generation for the Hyperplane Method and Loop Interchange*, report ENSMP-CAI-88-E102, CAI, Ecole des Mines de Paris, 1988

[13] F. Irigoin, *Loop Reordering with Dependence Direction Vectors*, Journées Firtech Systèmes et Télématique Architecture Futures: programmation parallèle et intégration VLSI, Paris, 9-10 novembre 1988

[14] F. Irigoin, R. Triolet, *Computing Dependence Direction Vectors and Dependence Cones with Linear Systems*, report ENSMP-CAI-87-E94, CAI, Ecole des Mines de Paris, 1987

[15] F. Irigoin, R. Triolet, *Supernode Partitioning*, ACM Symposium on Principles of Programming Languages, San-Diego, 1988

[16] F. Irigoin, R. Triolet, *Dependence Approximation and Global Parallel Code Generation for Nested Loops*, International Workshop on Parallel and Distributed Algorithms, Bonas, Oct. 3-6, 1988, North-Holland

[17] F. Irigoin, P. Jouvelot, R. Triolet, *Overview of the PIPS project*, International Workshop on Compilers for Parallel Computers, Paris, December 3-5, 1990.

[18] 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

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

[20] D. Loveman, *Program Improvement by Source-to-Source Transformations*, J. of the ACM, V. 20, n. 1, pp. 121-145

[21] P. A. Padua Haiek, *Multiprocessors: Discussion of Some Theoretical and Practical Problems*, PhD Dissertation, Report No. UIUCDCS-R-79-990, University of Illinois at Urbana-Champaign, 1979

[22] D. A. Padua, M. J. Wolfe, *Advanced Compiler Optimizations for Supercomputers*, Communications of the ACM, Vol. 29, n. 12, 1986

[23] J.-K. Peir, *Program Partitioning and Synchronization on Multiprocessors Systems*, Ph.D. Thesis, report UIUCDCS-R-86-1259, University of Illinois at Urbana-Champaign (March 1986)

[24] A. Porterfield, *Software Methods for Improvement of cache Performance on Supercomputer Applications*, Rice COMP TR89-93, Rice University, 1989

[25] W. Shang, J. A. Fortes, *Independent Partitioning of Algorithms with Uniform Dependencies*, 1988 Int'l Conference on Parallel Processing, pp. 26-33 (Aug. 1988)

[26] A. Schrijver, *Theory of Linear and Integer Programming*, Wiley, 1986

[27] A. E. Terrano, *Optimal Tiling for Iterative PDE Solvers*, Frontiers of Massively Parallel Computation, 1988

[28] M. Wolf, M. Lam, *Maximizing Paralellism via Loop Transformations*, 3rd Workshop on Programming Languages and Compilers for Parallel Computing, Irvine, 1990

[29] M. Wolfe, *Optimizing Supercompilers for Supercomputers*, Ph.D. thesis University of Illinois, Urbana, Rep. no UIUCDCS-R-82-1105, 1982.

[30] M. Wolfe, *Loop Skewing: The Wavefront Method Revisited*, Int'l Journal of Parallel Programming, V. 15, n. 4, 1986, pp. 279-294

[31] M. Wolfe, *Iteration Space Tiling for Memory Hierarchies*, in Parallel Processing for Scientific Computing, G. Rodrigue (ed.), SIAM, 1989, pp. 357-361

[32] M. Wolfe, *Optimizing Supercompilers for Supercomputers*, MIT Press, 1989

[33] M. Wolfe, *More Iteration Space Tiling*, Supercomputing 89, Reno, 1989, pp. 655-664