Non-unimodular Transformations of Nested Loops

J. Ramanujam*
Dept. of Electrical and Computer Engineering
Louisiana State University, Baton Rouge, LA 70803
jxr@max.ee.lsu.edu

Abstract

This paper presents a linear algebraic approach to modeling loop transformations. The approach unifies apparently unrelated recent developments in supercompiler technology. Specifically we show the relationship between the dependence abstraction called *dependence cones,* and *fully permutable loop nests.* Compound transformations are modeled as matrices. Non-singular linear transformations presented here subsumes the class of unimodular transformations. Non-unimodular transformations (with determinant ≥ 1) create "holes" in the transformed iteration space. We change the step size of loops in order to "step aside from these holes" when traversing the transformed iteration space. For the class of non-unimodular loop transformations, we present algorithms for deriving the loop bounds, the array access expressions and step sizes of loops in the nest. The algorithms are based on the Hermite Normal Form of the transformation matrix. We illustrate the use of this approach in several problems such as generation of tile sets, distributed memory code generation and dependence analysis. This approach provides a framework for optimizing programs for a variety of architectures.

1 Introduction

The interactions among loop transformations is of particular importance for the success of parallelizing compilers. The order of applying a set of transformations has a great impact on the performance of the loop on a given target parallel machine. In addition, it is becoming increasingly common for programmers of highly concurrent machines to specify parallelism using language constructs. Hence, compound loop transformations have received great attention recently.

This paper presents an approach for restructuring nested loops using general non-singular linear transformations. We present a linear algebraic view of a subset of loop transformations that results in a single notation for expressing those transformations. Many loop transformations are linear transformations applied to the iteration space and the dependence matrix. We use the matrix associated with the linear transformation. Applying transformations $T_1, T_2, \ldots, T_{k-1}, T_k$ (in that order) is equivalent to the matrix product $M_k \times M_{k-1} \times \ldots \times M_2 \times M_1$ where M_i is the matrix associated with the transformation T_i. The approach unifies apparently unrelated recent developments in supercompiler technology. Specifically we show the relationship between the dependence abstraction called *dependence cones,* and *fully permutable loop nests.* We outline a general technique for deriving loop bounds for unimodular transformations which we then use in the case of non-unimodular transformations.

Extant techniques are applicable to the realization of unimodular transformations only. Non-singular linear transformations presented here subsumes the class of unimodular transformations. Non-unimodular transformations (with determinant ≥ 1) create "holes" in the transformed iteration space. We change the step size of loops in order to "step aside from these holes" when traversing the transformed iteration space. For the class of non-unimodular loop transformations, we present algorithms for deriving the loop bounds, the array access expressions and step sizes of loops in the nest. In order to derive the step sizes, we compute the Hermite Normal Form of the transformation matrix; the step sizes are the entries on the diagonal of this matrix. In addition to these advantages, the approach provides a controlled search of the space of possible loop optimizations. We illustrate the use of this approach in several problems such as generation of tile sets, distributed memory code generation and dependence analysis.

2 Background

2.1 Dependences

Dependence analysis is critical to the success of parallelizing compilers [2, 5, 28, 25]. We deal with sets of perfectly nested loops, whose upper and lower bounds are all linear, enclosing a loop body with affine array references. That is, each subscript of an array variable index expression must be an affine expression over the scalar integer variables of the program. We assume that the loop body does not contain any goto's exiting one or many loops and that all statements are at the same nesting level. We assume familiarity with definitions of the types of dependences.

Dependence relations are often represented in *Iteration Space Graphs* (ISG's); for a K-nested loop with index set $(I_1, I_2, \ldots, I_K)$, the nodes of the ISG are points on a K-dimensional discrete Cartesian space

*Supported in part by NSF grant CCR–9210422, by the Louisiana Board of Regents through contract LEQSF (1991-94)-RD-A-09 and by the Halliburton Foundation.

and a directed edge exists between the iteration defined by $\vec{I_1}$ and the iteration defined by $\vec{I_2}$ whenever a dependence exists between statements in the loop constituting the iterations $\vec{I_1}$ and $\vec{I_2}$. Many dependences that occur in practice have a constant distance in each dimension of the iteration space. In such cases, the vector $\vec{d} = \vec{I_2} - \vec{I_1}$ is called the `distance vector`. An algorithm has a number of such dependence vectors; the dependence vectors of the algorithm are written collectively as a `dependence matrix` $D = [\vec{d_1}, \vec{d_2}, \ldots, \vec{d_n}]$. Methods to calculate data dependence vectors can be found in [2, 5, 28, 25, 30].

2.2 Extreme vectors

Given a set of n distinct dependence vectors ($n \geq K$), one can find a set of K vectors say $\vec{e_1}, \ldots, \vec{e_K}$ such that any dependence vector $\vec{d_i}$, $i = 1, \ldots, n$ can be expressed as a nonnegative linear combination of vectors $\vec{e_1}, \ldots, \vec{e_K}$, *i.e*

$$c_i \vec{d_i} = \sum_{j=1}^{K} a_{i,j} \vec{e_j}$$

where $i = 1, \ldots, n$ and $a_{i,j} \geq 0$ and $c_i > 0$ [4, 21, 22]. The set of vectors $e_1, \ldots, e_K$ are referred to as `extreme vectors`. Extreme vectors are not necessarily unique.

2.3 Terminology and assumptions

The set of computations in a perfectly nested loop is called the *computation domain.* For the purpose of this paper, only the structural information of the algorithm (nested loop), *i.e.,* the computation domain and the dependence matrix D are needed. With the assumption on loop bound linearity, the sets of computations considered are finite convex polyhedra of some iteration space Z^K, where K is the number of loops in the nest which is also the dimensionality of the iteration space. Each element, an iteration of the loop body S, is referred to by its iteration vector $\vec{J}$.

The iteration set $J \subset Z^K$ of any given algorithm is described as a set of integer points (vectors) whose convex hull IS is a non-degenerate (full dimensional) polyhedron, i.e.,

$$IS = \left\{ \vec{x} : P\vec{x} \leq \vec{q}, P \in Z^{a \times K}, \vec{q} \in Z^a, \vec{x} \in Z^K \right\}$$

where a is a positive integer and

$$J \subseteq \left\{ \vec{j} : \vec{j} \in IS \wedge j \in Z^K \right\}.$$

Matrix P is called the *constraint matrix* of J; vector $\vec{q}$ is called the *size vector.* The above relation simply states that IS is a polyhedron. Extreme points of polyhedron IS are always integers and belong to the iteration space J because IS is the convex hull of iteration space J. Different *size vectors* correspond to instances of the same algorithm that differ only in their sizes (but have the same shape of the computation domain). The following example illustrates the concepts introduced here.

```
for i = 1 to N do
  for j = i + 1 to M do
    ...
  endfor
endfor
```

The computation domain is given by:

$$\begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 1 & -1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} \leq \begin{bmatrix} -1 \\ N \\ -1 \\ M \end{bmatrix}$$

For convenience, we represent the computation domain as the matrix

$$\left[\begin{array}{rr|r} -1 & 0 & -1 \\ 1 & 0 & N \\ 1 & -1 & -1 \\ 0 & 1 & M \end{array}\right]$$

which is just the P matrix augmented with the size vector $\vec{q}$.

2.4 Related work

Dowling [9] presents an integer programming solution to the problem of optimal code parallelization using unimodular matrices transform a given nested loop into a parallel form. Banerjee [7] and Wolf and Lam [23] present unimodular transformations of loop nests and algorithms to derive transformations for maximal parallelism. Barnett and Lengauer [8] conclude that unimodularity is not essential; to derive code for non-unimodular transformations, they generate piecewise loop bounds which are not exact. Lu [14] uses conditionals in their transformed code to ensure correctness of the transformation. The techniques in this paper obviate the need for such conditionals. Wolfe [27] presents a technique that searches through a search space comprising both linear and non-linear transformations of the iteration space. In contrast to these approaches, our work presents a theory of non-singular linear transformations of the iteration space characterized by integral matrices whose determinant is an arbitrary integer. We shown an example of the need for non-unimodular transformations; the transformations are derived through modifying the step size of the loops. Li and Pingali [13] have independently derived a technique for code generation for general non-singular loop transformations. They present a completion algorithm which given a partial transformation generates a full non-singular transformation matrix that preserves dependences in the loop nest.

3 Loop transformations

A dependence matrix a collection of dependence distance vectors for a nested loop – the distance vectors are the columns of the dependence matrix. For a K-nested loop with n dependence vectors $d_i, i = 1, \ldots, n$, the dependence matrix D is a $K \times n$ matrix. Let $i_1, i_2, \ldots, i_K$ be the loop index variables for the loop nest levels. The vector $(i_1, i_2, \ldots, i_K)$ is referred to as the *nest vector.*

Loop transformations are applied to the loop nest vector and the dependence matrix. Let $\vec{I} = (I_1, \ldots, I_K)$ and $\vec{J} = (J_1, \ldots, J_K)$ be two iterations of a nested loop such that $\vec{I} \prec \vec{J}$ (read $\vec{I}$ precedes $\vec{J}$ in sequential execution) and there is a dependence of constant distance $\vec{d_i}$ between them. Applying a linear transformation T to the iteration space (nest vector) also changes the dependence matrix since $T(\vec{J})-T(\vec{I}) = T(\vec{J}-\vec{I}) = T\ \vec{d_i}$. All dependence vectors are *positive vectors,* i.e., the first non-zero component should be positive. We do not include *loop-independent* dependences which are zero vectors. A transformation is legal if the resulting dependence vectors are still *positive* vectors [5]. Thus, the linear algebraic view leads to a simpler notion of the legality of a transformation. For a K-nested sequential loop, the $K \times K$ identity matrix (I_K) denotes sequential execution order. Any unimodular transformation can be realized through reversal, interchange and skewing [7, 23].

3.1 Iteration Space Tiling

Loop tiling [3, 11, 18, 20, 23, 25, 26] is a transformation that decomposes a K-dimensional iteration space into K-dimensional "blocks" composed of a number of iterations where each block executes atomically; hence there should be no dependence cycles (deadlock) among the blocks. Tiling is extremely useful in distributed memory machines and hierarchical shared memory machines in exploiting locality of reference in the presence of dependences.

3.2 Relation between dependence cones and fully permutable nests

Tiles in K-dimensional (K-D) spaces are defined by K families of parallel hyperplanes (or planes) each of which is a $K-1$ dimensional hyperplane. To form K-dimensional tiles from a K-dimensional iteration space (assuming that the dependence matrix D (a $K \times n$ matrix) whose columns are dependence vectors has a rank K), the vectors normal to the K tile boundaries should be linearly independent. Let H denote the $K \times K$ matrix whose rows are the vectors $\vec{h_1}, \ldots, \vec{h_K}$. Thus, the rank of H must be K in order to define K-D tiles. In matrix notation, let $D^+ = HD$. The tiling condition implies that, for tiles to be legal [11, 20],

$$D^+_{ij} \geq 0 \qquad i = 1, \ldots, K \qquad j = 1, \ldots, n. \quad (1)$$

All entries in D^+ are nonnegative. Since, the rows of H are linearly independent, H must be non-singular and H^{-1} exists. Thus, the dependence matrix D can written as

$$D = H^{-1}D^+, \text{ where } D^+_{ij} \geq 0. \quad (2)$$

Since every entry in D^+ is nonnegative, the above relation implies that every dependence vector (every column of the matrix D) can be expressed as *nonnegative* linear combinations of the columns of the matrix H^{-1}. Thus the columns of H^{-1} are extreme vectors (by definition) for the set of dependence vectors $\vec{d_j}$ which constitute the columns of matrix D. Thus, tiles are either defined by a set of extreme vectors (for a given set of dependence vectors) or by a set of valid tiling planes given by a set of vectors normal to the tiling planes. Irigoin [11] introduced a dependence abstraction called *dependence cones* which can be automatically computed from systems of linear equalities (describing dependence conditions) and inequalities given by the constraint matrix P and the size vector $\vec{q}$ describing the computation domain. It is the generating system of a convex hull given by three sets of vertices, rays and lines. When the matrix D has full row rank (none of the loops are completely parallel), we have a pointed cone. The extreme vectors described here are an approximation to the dependence cone generated by exactly K extreme vectors and the approximate cone generated by the extreme vectors contains the dependence cone. If H is unimodular, H^{-1} is unimodular; hence the the columns of H^{-1} generate an integral polyhedral cone enclosing the *dependence cone.* It is of interest to compute the "smallest" enclosing approximate cone.

Also, condition (1) states that all entries in D^+ must non-negative. Therefore, *all loop permutations are legal* which is a necessary condition for generating strip-mined code with all the strip loops moved to the outer level. A loop nest in this form is called a *fully permutable nest* [23].

3.3 Generating the smallest approximate cone

Finding the tiling planes and hence the extreme vectors can be formulated as that of finding a transformation H such that

$$\sum_{k=1}^{d} H_{i,k} D_{k,j} \geq 0 \quad i = 1, \ldots, d \quad j = 1, \ldots, n$$

and H must be non-singular. We restrict our attention to lower triangular H with unit diagonal, since these are unimodular matrices. A nonzero entry in D^+ say D^+_{ij} implies that communication is incurred due the jth dependence vector poking the ith tile boundary. The amount communication across a tile boundary defined by the ith row of H is a function of the sum of the entries in the ith row of the transformed dependence matrix D^+. The ith row of H, namely $\vec{H_i}$ is the vector normal to the $K-1$ dimensional hyperplane containing $\vec{e_1}, \ldots, \vec{e_{i-1}}, \vec{e_{i+1}}$, and $\vec{e_K}$. The lower the sum of the entries in D^+, the more "minimal" the cone induced by the extreme vectors.

Thus, the problem of finding a set of "minimal" extreme vectors can be formulated as a linear programming problem of finding a transformation matrix H

that is lower triangular with unit diagonal as follows: Find H such that

$$\sum_{i=1}^{K}\sum_{j=1}^{n}\sum_{k=1}^{K} H_{i,k}D_{k,j}$$

is minimized subject to the constraints that

$$\begin{aligned}\sum_{k=1}^{K} H_{i,k}D_{k,j} &\geq 0 \; i=1,\ldots,K \; j=1,\ldots,n \\ H_{i,k} &= 0 \text{ if } k>i \\ H_{i,k} &= 1 \text{ if } k=i.\end{aligned}$$

4 Unimodularity is insufficient: Non-unimodularity and Communication

In this section, we present an example to show the need for non-unimodular transformations. In 2-dimensional iteration spaces, the transformation matrix is not necessarily unimodular [18]. For example, consider the dependence matrix

$$D = \begin{bmatrix} 0 & 1 & 2 & 1 \\ 1 & 1 & -1 & 0 \end{bmatrix}.$$

the extreme vectors are $(2,-1)$ and $(0,1)$ in which case the transformation matrix H is:

$$H = \begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix}$$

whose determinant is 2. From a transformational view, we need to find a non-singular matrix H

$$H = \begin{bmatrix} 1 & h_1 \\ h_2 & 1 \end{bmatrix}$$

such that

$$D^+ = HD = \begin{bmatrix} h_1 & h_1+1 & 2-h_1 & 1 \\ 1 & h_2+1 & 2h_2-1 & h_2 \end{bmatrix}$$

has nonnegative entries. The optimal solutions for linear programming problems occur at the *simplex* vertices or corners of the feasible region. The corners of the feasible region for the above set of constraints are: $h_1 = 0, h_2 = 1/2$ and $h_1 = 2, h_2 = 1/2$. Since we require that H is non-singular, its determinant must be nonzero *i.e.,* $1 - h_2h_1 \neq 0$, the second simplex point $h_1 = 2, h_2 = 1/2$ cannot be considered. Therefore, the solution is $h_1 = 0, h_2 = 1/2$. In general, we need to consider only those simplex vertices where the matrix H would be non-singular. The choice among those depends on the communication volume which is the topic of the ensuing discussion. Among unimodular matrices of the form

$$H = \begin{bmatrix} 1 & 0 \\ a & 1 \end{bmatrix},$$

$a = 1$ gives the minimum communication volume whose value is 9. If we look for non-singular matrices of the form

$$H = \begin{bmatrix} 1 & h_1 \\ h_2 & 1 \end{bmatrix},$$

$h_1 = 0$ and $h_2 = \frac{1}{2}$ gives the minimum value of $\sum_{i=1}^{d}\sum_{j=1}^{n} D^+_{i,j}$ which is 7. We note that H in this case has a determinant 2 and thus can not be generated by unimodular transformations. The reader is referred to [18] for details.

5 Issues in code generation

We address the following issues which arise in code generation for a general non-singular linear transformation T applied to a loop nest.

5.1 Computation of new array index expressions:

Let $\vec{x'} = T\vec{x}$ be the image of $\vec{x}$ under T. Let the nested loop involve array references which are given by the array index expressions. We assume that the index expressions are affine functions of the loop indices i.e., of the form $A_i\vec{x} + \vec{b_i}$, where A_i is an $M \times K$ matrix (assuming that the array being referenced is of dimension M) and the vector $\vec{b_i}$ is an M-element vector referred to as the *offset vector.* We want the same array elements accessed in $\vec{x}$ and $\vec{x'}$. Therefore we need $A_i\vec{x} + \vec{b_i}$ accessed in $\vec{x'}$ as well. In order to do this, we change $A_i\vec{x} + \vec{b_i}$ to $A_iT^{-1}\vec{x'} + \vec{b_i}$, since $\vec{x} = T^{-1}\vec{x'}$.

5.2 Computation of new loop bounds:

Specifying a loop involves writing the values of the upper and lower limits of each loop index with the constraint that the limit of an index d $(2 \leq d \leq K)$ is an affine function of the values of loop indices $x_1^{'}, \ldots, x_{d-1}^{'}$ where $x_j^{'}$ are the transformed loop indices. The computation domain is:

$$P\vec{x} \leq \vec{q}$$

Let T be a non-singular transformation applied to the loop nest, then

$$P\vec{x} \leq \vec{q} \Longrightarrow PT^{-1}T\vec{x} \leq \vec{q} \Longrightarrow PT^{-1}\vec{x'} \leq \vec{q}$$

In order to find the new loop bounds:

1. Compute PT^{-1}.
2. Convert PT^{-1} into row echelon form:
   (a) Form K groups $G_i, i = 1,\ldots,K$ of inequalities – the ith group of inequalities consist of all rows of PT^{-1} which have a nonzero entry in the ith column $(1 \leq i \leq K)$. The group G_1 corresponds to the new outermost loop index, G_2 corresponds to the new next loop index and so on.

(b) Eliminate inequalities in one group using those in another group so that the inequalities in group G_i do not have non-zero coefficients to the right of the ith column. This will result in G_i corresponding to the row echelon form.

(c) In the modified group G_i, all inequalities which have a negative entry in column i give lower bounds and all inequalities with positive entries in column i give upper bounds. The loop lower bound is the *maximum* corresponding to the former while the upper bound is the *minimum* corresponding to the latter.

The reader should note that the loop bounds computed thus are exact for unimodular loop transformations. These bounds have to be further refined for non-unimodular loop transformations and this will be discussed in detail in the next section. Consider the loop shown below:

---

for $i = 1$ to N do
  for $j = i + 1$ to M do
    ...
  endfor
endfor

The computation domain for this loop is:

$$\left[\begin{array}{cc|c} -1 & 0 & -1 \\ 1 & 0 & N \\ 1 & -1 & -1 \\ 0 & 1 & M \end{array}\right]$$

Let loop interchange be applied to the nest; interchange is equivalent to multiplication by the transpose of the 2×2 identity matrix. On computing PT^{-1}, we have two groups of inequalities. The first one is:

$$\left[\begin{array}{cc|c} 0 & -1 & -1 \\ 0 & 1 & N \\ -1 & 1 & -1 \end{array}\right]$$

and the second one is:

$$\left[\begin{array}{cc|c} -1 & 1 & -1 \\ 1 & 0 & M \end{array}\right]$$

Note that a row of this augmented matrix belongs as many groups as the number of nonzeros in that row. If we replace the first row of the second group by the sum of the first row of the second group and the first row of the first group, we get the new second group as follows:

$$\left[\begin{array}{cc|c} -1 & 0 & -2 \\ 1 & 0 & M \end{array}\right]$$

Since this group has nonzeros only in the first column, we are done with loop limits for the outer "j" loop – it runs from 2 to M; the lower bound is extracted from the row with coefficient -1 and the upper bound is extracted from the row with coefficient 1. From the first group we have

$$-i \le -1$$

$$i \le N$$

$$-j + i \le -1 \quad \Longrightarrow i \le j - 1$$

Therefore, the lower bound of the inner loop is 1; the upper bound is represented by 2 inequalities both of which hold. Hence we combine these as

$$i \le min(N, j - 1)$$

Therefore the transformed loop is:

---

for $j = 2$ to M do
  for $i = 1$ to $min(N, j - 1)$ do
    ...
  endfor
endfor

---

6 Code derivation for non-unimodular transformations – the Hermite Normal Form approach

We generalize unimodular transformations described by [7, 24] to derive non-singular linear transformations defined by integral matrices whose determinant is not restricted to ± 1. This section presents details of code generation for non-unimodular loop transformations. Iteration spaces defined by nested loops form bounded integer lattices. The theory presented here is based on the fact that the lattice generated by the columns of a matrix T is the same as the lattice generated by the *Hermite Normal Form* (defined below) of the matrix T. The entries on the diagonal of the Hermite Normal Form of T give the loop step sizes. In the case of n-nested loops with step size 1, the iteration space is a lattice generated by the columns of the $n \times n$ identity matrix $I_{n \times n}$ [19], since any point in the iteration space can be written as integral linear combinations of the columns of $I_{n \times n}$. If the step size along loop level j is $s_j \ne 1$, then the corresponding iteration space is generated by generated by a matrix which differs from $I_{n \times n}$ in that the entry $I_{j,j} = s_j$.

6.1 Definitions

An $m \times n$ matrix C is unimodular if it is integral and its determinant is ± 1. The set

$$\mathcal{L}(T) = \{y \in Z^m : y = Tx, x \in Z^n\}$$

where T is an $m \times n$ matrix is called the *lattice* generated by the columns of T.

Theorem 1 *If T is an integral $m \times n$ matrix, and C is an $n \times n$ unimodular matrix, then $\mathcal{L}(T) = \mathcal{L}(TC)$.*

Proof: Let $x = Cz$. Then we have

$$\mathcal{L}(T) = \{y \in Z^m : y = TCz, Cz \in Z^n\}$$

Since C is an integer matrix, $z \in Z^n$ implies $Cz \in Z^n$. Since C is unimodular, C^{-1} is also unimodular and integral; hence $Cz \in Z^n$ implies that $C^{-1}Cz \in Z^n$ or $z \in Z^n$. Thus, it follows that $\{z : Cz \in Z^n\} = \{z : z \in Z^n\}$. Therefore, $\mathcal{L}(T) = \mathcal{L}(TC)$. $\square$

An $n \times n$ non-singular matrix H is said to be in *Hermite Normal Form* if:

1. H is lower triangular and $h_{ij} = 0$ for $i < j$.
2. $h_{ii} > 0$ for $i = 1, \cdots, n$.
3. $h_{ij} \leq 0$ and $|h_{ij}| < h_{ii}$ for $i > j$

Result 1 *If T is an $m \times n$ integer matrix with $rank(T) = m$, then there exists an $n \times n$ unimodular matrix C such that:*

- *$TC = (H, 0)$ and H is in its Hermite Normal Form (and)*
- *$H^{-1}T$ is an integral matrix.*

$(H, 0)$ is called the Hermite Normal Form *of T. For a non-singular square matrix T of size n, H and C are square matrices of size $n \times n$.*

The right multiplication of T by a unimodular matrix C corresponds to a sequence of elementary column operations defined below:

1. multiply a column j by -1.
2. interchange two columns j and k.
3. add an integral multiple of column j to column k.

Hence, for any given matrix T, its HNF matrix H can be found by a sequence of elementary column operations. The Hermite Normal form can be computed in polynomial time [15, 22]. The reader is referred to [15, 22] for the algorithms.

6.2 Exact loop bounds

Let l_k and u_k ($k = 1, \ldots, n$) be the lower and upper bounds of the transformed loops as calculated using the procedure given in the last section. Since floors and ceilings are used in computing respectively the lower and upper bounds, l_k and u_k are accurate boundaries of the transformed iteration space; where as the closest integer point computed using the steps in section 5.2 need not be the image of a point in the source iteration space. In the following discussion, we use L_k and U_k to refer to the exact bounds. Since both H and T generate the same lattice, every point in the transformed iteration space can be written as $H\vec{x}$ where the vector $\vec{x} = (x_1, \ldots, x_n)$. Let $I_1, \ldots, I_n$ be the loop variables of the transformed nest. Let e_k denote the expression $\left(l_k - \sum_{j=1}^{j=k-1} \frac{H[k,j]}{H[j,j]} I_k\right) \, mod \; H[k,k]$.

$$L_k = \begin{cases} l_k & \text{if } e_k = 0 \\ l_k + H[k,k] - e_k & \text{otherwise.} \end{cases}$$

Since conditionals are cumbersome in loop bound evaluation, the exact lower bound L_k can be rewritten as:

$$l_k + \left(\left\lceil \frac{l_k}{H[k,k]} \right\rceil - \left\lfloor \frac{l_k}{H[k,k]} \right\rfloor \right) (l_k + H[k,k] - e_k)$$

The exact upper bound is derived similarly. The calculation of exact loop bounds obviates the need for expensive runtime conditional tests as used in [3, 8, 14].

6.3 Example of code derivation

We use the following loop skeleton to illustrate the application of our techniques:

Example 1:
  for $i = 1$ to N
    for $j = 1$ to N
      $\cdots$
    endfor
  endfor

Consider the following non-unimodular transformation

$$T = \begin{bmatrix} 1 & 1 \\ -1 & 1 \end{bmatrix}.$$

whose determinant is 2. We assume that T is a valid transformation. In order to restructure the loop nest, we first find the *HNF* of T. The matrix H (the HNF of T) is derived through the following column operation:

- column 2 = column 2 − column 1

and the matrix H is:

$$T = \begin{bmatrix} 1 & 0 \\ -1 & 2 \end{bmatrix}.$$

Therefore, in the transformed loop nest, the step size of the outer loop is 1 and the inner loop is 2. The matrix T^{-1} is:

$$T^{-1} = \begin{bmatrix} \frac{1}{2} & \frac{-1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{bmatrix} = \frac{1}{2} \begin{bmatrix} 1 & -1 \\ 1 & 1 \end{bmatrix}$$

The computation domain (P matrix augmented with vector $\vec{q}$) is:

$$\left[\begin{array}{cc|c} -1 & 0 & -1 \\ 1 & 0 & N \\ 1 & -1 & -1 \\ 0 & 1 & N \end{array}\right]$$

Figure 1: Effect of T on example 1.

In order to compute the loop bounds, we first compute PT^{-1}.

$$PT^{-1} = \left[\begin{array}{cc|c} \frac{-1}{2} & \frac{1}{2} & -1 \\ \frac{1}{2} & \frac{-1}{2} & N \\ \frac{-1}{2} & \frac{-1}{2} & -1 \\ \frac{1}{2} & \frac{1}{2} & N \end{array}\right]$$

which is the same as:

$$PT^{-1} = \left[\begin{array}{cc|c} -1 & 1 & -2 \\ 1 & -1 & 2N \\ -1 & -1 & -2 \\ 1 & 1 & 2N \end{array}\right]$$

If we replace row 1 by the sum of rows 1 and 3 and if we replace row 2 by the sum of rows 2 and 4, we get the bounds of the outer loop from

$$\left[\begin{array}{cc|c} -2 & 0 & -4 \\ 2 & 0 & 4N \end{array}\right]$$

Thus, the new outer loop limits are 2 and $2N$ and the step size is 1. Similarly, we can derive the loop limits of the inner loop whose step size is 2. Since the outer loop variable i is even, all the expressions $2-i, i-2N, i-2, 2N-i$ are all even; hence the loop bounds derived are exact. The transformed loop is:

for $i = 2$ to $2N$ by 1
  for $j = max(2-i, i-2N)$ to $min(i-2, 2N-i)$ by 2
    $\cdots$
  endfor
endfor

We illustrate the application of our technique for this example in Figure 1, which depicts both the original and the transformed iteration spaces for $N = 3$.

7 Application to tiling

In tiling a nest of loops, the set of tiles (referred to as the tile set) is defined as a collection of origins of the tiles. An origin of a tile, is the lexicographically earliest (smallest) iteration that belongs to the tile. Hence, the origins of tiles in K-dimensions are generated by a matrix whose columns form the basis of the lattice. In many cases, the lattice of tile origins are defined by a user through an input matrix or automatically generated by an optimizing compiler. We use the same terminology as [3] here. Let L be a $K \times K$ matrix specifying the lattice of tile origins for a loop nest with a computation domain given by (P, q). We show how to generate the origins of the tile sets. First we find the Hermite Normal Form of L which gives the step sizes of the tile loops; then we use the loop bound generation technique discussed in the last section.

We use the following loop nest (from [3]):

Example 2:

for $i = 1$ to N
  for $j = 1$ to $N + 1 - i$
    ...
  endfor
endfor

for which tile origins are given by the lattice generated generated by the columns of L, where L is:

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

The HNF of L is found by using the following two column operations:

1. column 2 = column 2 $+3*$ column 1

2. column 1 = $-1*$ column 1

$$H = LC$$

$$\begin{bmatrix} 3 & 0 \\ -6 & 9 \end{bmatrix} = \begin{bmatrix} -3 & 9 \\ 6 & -9 \end{bmatrix} \begin{bmatrix} -1 & 3 \\ 0 & 1 \end{bmatrix}$$

The step size of the outer loop is 3 and the inner loop is 9. The matrix L^{-1} is:

$$L^{-1} = \begin{bmatrix} \frac{1}{3} & \frac{1}{3} \\ \frac{2}{9} & \frac{1}{9} \end{bmatrix}$$

The computation domain (P matrix augmented with vector $\vec{q}$) is:

$$\left[\begin{array}{rr|r} -1 & 0 & -1 \\ 1 & 0 & N \\ 0 & -1 & -1 \\ 1 & 1 & N+1 \end{array}\right]$$

In order to compute the loop bounds, we first compute PL^{-1}.

$$PL^{-1} = \left[\begin{array}{rr|r} \frac{-1}{3} & \frac{-1}{3} & -1 \\ \frac{1}{3} & \frac{1}{3} & N \\ \frac{-2}{9} & \frac{-1}{9} & -1 \\ \frac{5}{9} & \frac{4}{9} & N+1 \end{array}\right]$$

which is the same as:

$$PL^{-1} = \left[\begin{array}{rr|r} -1 & -1 & -3 \\ 1 & 1 & 3N \\ -2 & -1 & -9 \\ 5 & 4 & 9N+9 \end{array}\right]$$

If we replace row 1 by the sum of rows 1 and 4 and if we replace row 2 by the sum of rows 2 and 3, we get the bounds of the outer loop as $3N - 9$ and $9N - 3$. Similarly, we can derive the loop boundss of the inner loop. The loop limits derived here using steps outlined in 5.2 need not be exact. The exact lower bound L_j is

$$max(3 - i, 9 - 2i) + \left(\left\lceil \frac{max(3-i, 9-2i)}{9} \right\rceil - \left\lfloor \frac{max(3-i, 9-2i)}{9} \right\rfloor \right) * (9 - (max(3-i, 9-2i) + 2i) \, mod \, 9)$$

The transformed loop is:

for $i = 3N - 9$ to $9N - 3$ by 3
  for $j = L_j$ to $min(3N - i, \lfloor 9N + 9 - 5i/4 \rfloor)$ by 9
    ...
  endfor
endfor

The points in the transformed iteration space are the origins of the tile sets, where each tile set has at least iteration space point in it. We note here that for the sake of completeness, one should calculate the exact upper bounds as well.

8 Other applications

8.1 Application in distributed memory code generation

Current efforts in distributed memory code generation start with a user-defined decomposition of data structures [10]. Extensions to a serial language allow users to specify the decomposition of arrays. A processor is said to own elements of data allocated to it. The Fortran-D research group at Rice notes: "We find that selecting a data decomposition is one of the most important intellectual steps in developing data parallel scientific codes" [10]. The compiler automatically partitions the program using the *owner compute* rule by which each processor computes values of data it owns. The data owned by a processor is known as its *local index set* and the set of loop iterations that compute owned data is referred to as the *local iteration set* [10].

Non-singular transformations may be needed in deriving the local iteration sets; we compute the owner loop bounds so that processors avoid executing expensive guards looking for work to do. Consider the loop shown below:

Example 3:

for $i = l_1$ to u_1
  for $j = l_2$ to u_2
    $A[2i + j - 3, i + j + 2] = \cdots$
  endfor
endfor

Assume for this example that the locally owned part of A in some processor is given by $A[(5:9),(15:19)]$. The subscript functions can be represented in matrix form as: $A\vec{x} + \vec{b}$, or

$$\begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} -3 \\ 2 \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix}.$$

where $\vec{c} = (c_1, c_2)^T$ corresponds to a point in the data space and $\vec{x}$ corresponds to a point in the iteration space. In order to enforce the *owner compute* rule, $5 \leq c_1 \leq 9$ and $15 \leq c_2 \leq 19$. That is,

$$5 \leq 2i + j - 3 \leq 9$$

$$15 \leq i + j + 2 \leq 19$$

which is equivalent to the following system of inequalities:

$$\left[\begin{array}{rr|r} -2 & -1 & -8 \\ 2 & 1 & 12 \\ -1 & -1 & -13 \\ 1 & 1 & 17 \end{array}\right]$$

which when reduced to row echelon form gives the following local iteration set for the processor avoiding expensive guards:

for $i = max(l_1, -9)$ to $min(u_1, -1)$
  for $j = max(l_2, -2i + 8, -i + 13)$ to
      $min(u_2, -2i + 12, -i + 17)$
    /* access local data */
    $A[2i + j - 3, i + j + 2] = \cdots$
  endfor
endfor

8.2 Application of HNF to dependence analysis

The Hermite Normal Form is also useful in solving systems of dependence equations which are systems of linear diophantine equations. It can be used to check if there are no integer solutions; if there are integer solutions, HNF can be used to find a general form of the solutions. Let T, H and C be matrices as defined in Result 1 in section 6.1 and let $rank(T) = m$. Let S be the set of solutions to the equation $Tx = b$ *i.e.,*

$$S = \{x \in Z^n : Ax = b, b \in Z^m\}$$

Let $C = (C_1, C_2)$ where C_1 is the matrix formed by the first m columns of C and C_2 is the matrix with the remaining $n - m$ columns of C. The following results from [15] can be directly used in dependence analysis:

1. S is nonempty if and only if $H^{-1}b \in Z^m$
2. If S is nonempty, every solution $x \in S$ is of the form
$$x = C_1 H^{-1} b + C_2 w$$
where $w \in Z^{n-m}$

The first statement can be used to test if there are integer solutions. If there are integer solutions, the second statement provides a general solution to the system. From this point on, one could apply the Power Test to detect any solutions within the range of the variables [29].

9 Discussion

Compound loop transformations modeled as linear transformations have received great attention recently. In this paper, we presented a linear algebraic approach to modeling loop transformations. The approach unifies apparently unrelated recent developments in supercompiler technology. Specifically we showed the relationship between the dependence abstraction called *dependence cones,* and *fully permutable loop nests.* We then presented techniques for code generation for non-unimodular loop transformations. Unimodular transformations [7, 23] form a subclass of non-singular linear transformations presented here. Non-unimodular transformations (with determinant ≥ 1) create "holes" in the transformed iteration space. We change the step size of loops in order to "step aside from these holes" when traversing the transformed iteration space. For the class of non-unimodular loop transformations, we presented algorithms for deriving the loop bounds, the array access expressions and step sizes of loops in the nest. To derive the step sizes, we compute the Hermite Normal Form of the transformation matrix; the step sizes are the entries on the diagonal of this matrix. We then illustrated the use these techniques in important applications such as generating tile sets, deriving local iteration sets in distributed memory code generation, and in the area of dependence analysis. The technique presented here holds promise in creating a framework for optimizing programs for a wide variety of architectures.

References

[1] J. R. Allen and K. Kennedy. Automatic Loop Interchange. *Proc. SIGPLAN 84 Symposium on Compiler Construction,* pages 233–246, June 1984.

[2] J. R. Allen and K. Kennedy. Automatic Translation of FORTRAN Programs to Vector Form. *ACM Trans. Programming Languages and Systems,* Vol. 9, No. 4, pages 491–542, October 1987.

[3] C. Ancourt and F. Irigoin, Scanning polyhedra with DO loops. In *Proc. Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP),* pages 39–50, April 1991.

[4] A. Bachem. The Theorem of Minkowski for Polyhedral Monoids and Aggregated Linear Diophantine Systems. *Optimization and Operations Research – Proc. of Workshop,* University of Bonn, October 1977, Lecture Notes in Economics and Mathematical Systems, Vol. 157, pages 1–14, Springer Verlag.

[5] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, Massachusetts, 1988.

[6] U. Banerjee. A Theory of Loop Permutations. *Languages and Compilers for Parallel Computing,* D. Gelernter, A. Nicolau and D. Padua (Eds.) Pitman, London, 1990, pp. 54–74.

[7] U. Banerjee. Unimodular Transformation of Double Loops. *Advances in Languages and Compilers for Parallel Processing,* A. Nicolau et al., (Eds.), Pitman, London, 1991, pp. 192–219.

[8] M. Barnett and C. Lengauer. Loop Parallelization and Unimodularity. Report ECS-LFCS-92-197, University of Edinburgh, 1992.

[9] M. Dowling. Optimal Code Parallelisation using Unimodular Transformations. *Preprints in Optimization, Carolo-Wilhelmina Universität zu Braunschweig, W. Germany,* July 1988.

[10] S. Hiranandani, K. Kennedy and C. Tseng. Compiler Optimization for Fortran D on MIMD Distributed Memory Machines. in *Proc. Supercomputing 91,* pages 86–100, November 1991.

[11] F. Irigoin and R. Triolet. Supernode Partitioning. *Proc. 15th Annual ACM Symp. Principles of Programming Languages,* San Diego, CA, Jan. 1988, 319-329.

[12] L. Lamport. The Parallel Execution of DO Loops. *Communications of the ACM,* Vol. 17, No. 2, pages 83–93, February 1974.

[13] W. Li and K. Pingali. A Singular Loop Transformation Framework Based on Non-singular Matrices. *Proc. 5th Workshop on Languages and Compilers for Parallel Computing,* August 1992.

[14] L. Lu. A Unified Framework for Systematic Loop Transformations. In *Proc. Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming (PPoPP),* pages 28–38, April 1991.

[15] G. Nemhauser and L. Wolsey. *Integer and Combinatorial Optimization.* Wiley-Interscience series in Discrete Mathematics and Optimization, John Wiley and Sons, New York, 1988.

[16] M. Newman. *Integral Matrices.* Academic Press, New York, 1972.

[17] J. Ramanujam and P. Sadayappan. Tiling of Iteration Spaces for Multicomputers. *Proc. 1990 International Conference on Parallel Processing,* Vol 2, pages 179-186, August 1990.

[18] J. Ramanujam. *Compile-time Techniques for Parallel Execution of Loops on Distributed Memory Multiprocessors.* Ph.D. Thesis, The Ohio State University, September 1990.

[19] J. Ramanujam. A linear algebraic view of loop transformations and their interaction. in *Parallel Processing for Scientific Computing,* D. Sorensen (Editor), SIAM Press, March 1991.

[20] J. Ramanujam and P. Sadayappan. Multidimensional Iteration Space Tiling for Nonshared Memory Machines. in *Proc. Supercomputing 91,* pages 111–120, November 1991.

[21] H. Salkin and K. Mathur. *Foundations of Integer Programming.* North-Holland, New York, NY, 1989.

[22] A. Schrijver. *Theory of Linear and Integer Programming.* Wiley-Interscience series in Discrete Mathematics and Optimization, John Wiley and Sons, New York, 1986.

[23] M. Wolf and M. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Trans. Parallel and Distributed Syst.,* October 1991.

[24] M. Wolf and M. Lam. A Data Locality Optimizing Algorithm. *Proc. ACM SIGPLAN 91 Conf. Programming Language Design and Implementation,* pp. 30–44, June 1991.

[25] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* Pitman Publishing, London and the MIT Press, Cambridge, Massachusetts, 1989.

[26] M. Wolfe. More Iteration Space Tiling. *Proc. Supercomputing 89,* Reno, Nevada, pages 655–664, November 1989.

[27] M. Wolfe. Massive Parallelism through Program Restructuring. *Tech. Report CS/E 90-009,* Oregon Graduate Institute, June 1990.

[28] M. Wolfe and U. Banerjee. Data Dependence and Its Application to Parallel Processing. *International Journal of Parallel Programming,* Vol. 16, No. 2, pages 137–178, 1987.

[29] M. Wolfe and C. Tseng. The power test for data dependence. To appear in *IEEE Trans. Parallel and Distributed Systems.,* 1992.

[30] H. Zima and B. Chapman. *Supercompilers for parallel and vector supercomputers.* ACM Press Frontier Series, 1990.