Beyond Unimodular Transformations

J. Ramanujam*

Department of Electrical and Computer Engineering

Louisiana State University, Baton Rouge, LA 70803-5901

Email: jxr@gate.ee.lsu.edu

Abstract

This paper presents an approach to modeling loop transformations using linear algebra. Compound transformations are modeled as integer matrices. Non-singular linear transformations presented here subsumes the class of unimodular transformations. The loop transformations included are the unimodular transformations – reversal, skewing, permutation – and a new transformation, namely *stretching*. Non-unimodular transformations (with determinant ≥ 1) create "holes" in the transformed iteration space, rendering code generation difficult. We solve this problem by suitably changing the step size of loops in order to "skip" 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. 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 use the theory of Hessenberg matrices in the derivation of exact loop bounds for non-unimodular transformations. We illustrate the use of this approach in several problems such as generation of tile sets, and distributed memory code generation. This approach provides a framework for optimizing programs for a variety of architectures.

*Supported in part by an NSF Young Investigator Award CCR–9457768, an NSF grant CCR–9210422, and by the Louisiana Board of Regents through contract LEQSF (1991-94)-RD-A-09.

1 Introduction

Loop transformations are widely used by automatic parallelizers to generate efficient code for a variety of high performance computers [1, 4, 20, 24]. 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.

Banerjee [4], Dowling [6], Wolf and Lam [19] and this author [14] have derived unimodular transformations that model compound loop transformations using integer matrices with unit determinant. This paper presents an approach for restructuring nested loops using general non-singular linear transformations, the matrices associated with which need not have unit determinants. Non-singular linear transformations presented here subsumes the class of unimodular transformations. Non-unimodular transformations (with determinant $\geq$ 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.

Section 2 presents the terminology and assumptions used in this paper. Section 3 discusses unimodular loop transformations and introduces *loop stretching;* in addition, we argue that unimodularity is not sufficient for optimizing communication with tiling for distributed memory machines. Issues in code generation for unimodular and non-unimodular transformations are discussed in Section 4. Section 5 presents solutions to the issues of loop step size (based on the Hermite Normal Form of the transformation matrix) and exact loop bound generation (based on Hessenberg matrices), in addition to an example illustrating the use of our techniques. In Section 6, we discuss applications of the Hermite Normal From to such applications as tile set generation, and distributed memory code generation. Section 7 presents a discussion of related work. Section 8 summarizes the paper and points to other applications and work in progress.

2 Background

A *data dependence* between two references arises from their reading or writing a common memory location. This dependence is called an *input dependence* if both the references are reads; input dependences do not constrain program execution ordering. If at least one of the two references is a write, the dependence can be a flow, anti or output dependence. Let $\mathcal{W}$ and $\mathcal{R}$ be two references to a memory location, and let $\mathcal{W}$ be a write and $\mathcal{R}$ be the read. If $\mathcal{W}$ precedes $\mathcal{R}$ in sequential execution, then the dependence is called a *flow dependence.* If $\mathcal{R}$ precedes $\mathcal{W}$ in sequential execution, then the dependence is called an *anti dependence.* If both the accesses to the memory location are writes, we have an *output dependence.*

Data dependences are characterized by the *dependence distance,* which is the number of iterations that separate the sink of the dependence from its source. *Dependence directions* which represent the sign of the dependence distance are also used to characterize dependences. Dependence distance and direction are represented as vectors called *dependence vectors.* The elements of the vector read from left to right indicate the dependence from the outermost to the innermost loop. We will use dependence vectors as columns vectors. A *dependence matrix* has as its columns, a set of dependence vectors. All dependence vectors are *positive vectors, i.e.,* the first non-zero component in the vector (read left to right) must be positive. Any loop transformation is legal if after transformation, the dependence vectors remain positive vectors. A detailed discussion of dependences including dependence testing can be found in [1, 3, 20, 23, 24].

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 nested loop, *i.e.,* the computation domain and the distance matrix D are needed. With the assumption on loop bound linearity, the sets of computations considered are finite convex polyhedra of some iteration space in Z^n, where n 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 $J \subset Z^n$ of a given nested loop is described as a set of integer points (vectors) whose convex hull $\mathcal{I}$ is a non-degenerate (full dimensional) polyhedron, *i.e.,*

$$\mathcal{I} = \{\vec{x} : P\vec{x} \leq \vec{q}, P \in Z^{a \times n}, \vec{q} \in Z^a, \vec{x} \in Z^n\}$$

where a is a positive integer and

$$J \subseteq \left\{\vec{j} : \vec{j} \in \mathcal{I} \wedge j \in Z^n\right\}.$$

Matrix P is called the *constraint matrix* of J; vector $\vec{q}$ is called the *size vector.* The above relation simply states that $\mathcal{I}$ is a polyhedron. Extreme points of polyhedron $\mathcal{I}$ are always integers and belong to the iteration space J because $\mathcal{I}$ is the convex hull of iteration space J. In addition, extreme points remain extreme points under rank-preserving linear transformations. 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

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

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

3 Loop transformations

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

Loop transformations are applied to the loop nest vector and the dependence matrix. Let $\vec{I} = (I_1, \ldots, I_n)$ and $\vec{J} = (J_1, \ldots, J_n)$ be two iterations of a nested loop such that $\vec{I} \prec \vec{J}$ (read $\vec{I}$ precedes $\vec{J}$ in sequential

Figure 1: Loop transformations: interchange, reversal and skewing

Figure 2: Loop stretching

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 distance 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 [3]. Note that the legality definition applies to bounded and unbounded sets of dependence vectors; constant distance vectors are bounded dependence vectors. Thus, the linear algebraic view leads to a simpler notion of the legality of a transformation. For an n-nested sequential loop, the $n \times n$ identity matrix ($I_{n\times n}$) denotes sequential execution order. Loop transformations correspond to elementary row operations on matrices [4].

- Interchanging rows corresponds to *loop interchange, permutation and circulation.*
- Multiplying a row by -1 corresponds to *loop reversal.*
- Replacing a given row k by row k plus an integral multiple of another row i is equivalent to loop skewing. If $k > i$, it is called *inner loop skewing;* if $k < i$, it is *outer loop skewing.*

Figure 1 shows the effect of each transformation on a 3×3 iteration space. Banerjee [4] showed that any unimodular transformation can be realized through reversal, interchange and skewing. In linear algebraic terms, the matrix associated with a unimodular transformation can be derived by elementary row operations on an identity matrix [14]. Let $\vec{x}$ denote the nest vector of a loop nest. Let T denote a unimodular transformation which results in the transformed nest vector $\vec{y}$, *i.e.,* $\vec{y} = T\vec{x}$. A computation domain $[P, \vec{q}]$ on tranformation T becomes $[PT^{-1}, \vec{q}]$, since $P\vec{x} \leq \vec{q} = PT^{-1}T\vec{x} \leq \vec{q} = PT^{-1}\vec{y} \leq \vec{q}$.

3.1 Loop stretching

Loop stretching is a transformation that corresponds to changing the step size of a loop by an integer factor f. Stretching is always legal since it does not alter the order of execution. Loop stretching is the inverse of loop step normalization [20, 24]. Multiplying a row by f where $f > 1$ corresponds to stretching. For example, the transformation,

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

changes the step size of the inner loop to 2. Figure 2 illustrates the effect of T on a 3×3 iteration space. Note that within the boundary of the transformed iteration space, there are points which are skipped over; we refer to these points as "holes." These are the points in the transformed iteration space which are not images of any point in the original iteration space, *i.e.,* these are vectors $\vec{y}$ for which $T^{-1}\vec{y}$ is not integral.

Any nonsingular integer linear transformation can be realized through combinations of reversal, stretching, interchange and skewing – this follows from the fact that a series of row operations applied to the identity matrix can generate any integer matrix [18].

3.2 Unimodularity is insufficient: Non-unimodularity and Communication

In this section, we present an example to show the need for non-unimodular transformations. Iteration space tiling [9, 13, 16, 19, 20, 21] is a transformation that decomposes an n-dimensional iteration space into n-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 [13, 16, 17]. In [16], we modeled the problem of finding the best tiling transformation for distributed memory machines as (a linear programming problem) that of finding a matrix T such that the entries in $D^+ = TD$ are non-negative (so that there are no dependence cycles) and the sum of the entries in D^+ is a minimum; the sum of the entries in D^+ corresponds to the amount of communication (flux or volume) required. In 2-dimensional iteration spaces, the best transformation matrix

Figure 3: Effect of unimodularity of communication

is not necessarily unimodular [13]. For example, consider the distance matrix

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

The tiling transformation T with the least communication is

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

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

$$T = \begin{bmatrix} 1 & t_1 \\ t_2 & 1 \end{bmatrix}$$

such that

$$D^+ = TD = \begin{bmatrix} t_1 & 2 - t_1 & 1 \\ 1 & 2t_2 - 1 & t_2 \end{bmatrix}$$

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: $t_1 = 0, t_2 = 1/2$ and $t_1 = 2, t_2 = 1/2$. Since we require that T is non-singular, its determinant must be nonzero *i.e.,* $1 - t_2 t_1 \neq 0$, the second simplex point $t_1 = 2, t_2 = 1/2$ cannot be considered. Therefore, the solution is $t_1 = 0, t_2 = 1/2$. In general, we need to consider only those simplex vertices where the matrix T would be non-singular. Among unimodular matrices of the form

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

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

$$T = \begin{bmatrix} 1 & t_1 \\ t_2 & 1 \end{bmatrix},$$

$t_1 = 0$ and $t_2 = \frac{1}{2}$ gives the smallest communication volume. Figure 3 illustrates the effect of unimodularity on communication. In Figure 3, we do not show the communication arising due to the dependence vector

$(1, 0)^t$. The tile shape in Figure 3(a) is generated by the non-unimodular matrix

$$\begin{bmatrix} 1 & 0 \\ 1 & 2 \end{bmatrix},$$

and the tile shape shape in Figure 3(b) is generated by the unimodular matrix

$$\begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

We note that T in this case has a determinant 2 and thus can not be generated by unimodular transformations. The reader is referred to [13] for details.

4 Issues in code generation

4.1 Unimodular transformations

Assume that a unimodular transformation T is applied to a loop nest. The issues in code generation for T are:

1. Computation of new array index expressions.

2. Computation of new loop bounds.

4.1.1 Computation of new array index expressions:

Let $\vec{x}$ denote the nest vector for the original iteration space and $\vec{y}$ denote the transformed iteration space nest vector, *i.e.,* $\vec{y} = T\vec{x}$ is the image of $\vec{x}$ under T. We assume that the array 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 $m \times n$ 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 $\vec{y}$. Therefore we need $A_i\vec{x} + \vec{b_i}$ accessed in $\vec{y}$ as well. In order to do this, we change $A_i\vec{x} + \vec{b_i}$ to $A_iT^{-1}\vec{y} + \vec{b_i}$, since $\vec{x} = T^{-1}\vec{y}$.

4.1.2 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 a loop index y_i $(2 \leq i \leq n)$ is an affine function of the values of loop indices

$y_1, \ldots, y_{i-1}$ where y_i 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

$$P\vec{x} \leq \vec{q} \Longrightarrow PT^{-1}T\vec{x} \leq \vec{q} \Longrightarrow PT^{-1}\vec{y} \leq \vec{q}$$

In order to find the new loop bounds:

1. Compute PT^{-1}.
2. Convert PT^{-1} into row echelon form:
   (a) Form n groups $G_i, i = 1, \ldots, n$ of inequalities – the ith group of inequalities consists of all rows of PT^{-1} which have a nonzero entry in the ith column ($1 \leq i \leq n$). The group G_1 corresponds to the new outermost loop index, G_2 corresponds to the new next loop index and so on. The groups G_i are not disjoint. An inequality (row) belongs to as many groups as the number of nonzero coefficients in it.
   (b) For i from 1 to $n - 1$, eliminate inequalities in group G_i using those in groups G_{i+1} to G_n so that the inequalities in group G_i have a coefficient of zero for y_j $(i + 1 \leq j \leq n)$, *i.e.,* all coefficients to the right of the ith column are zero. This will render G_i in 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 for the ith loop in the transformed nest. The loop lower bound is the ceiling of the *maximum* corresponding to the former while the upper bound is the floor of the *minimum* corresponding to the latter.

The reader should note that the loop bounds computed thus are exact for unimodular loop transformations. We note here that Fourier-Motzkin elimination can be used as well for deriving loop bounds. These bounds have to be further refined for non-unimodular loop transformations and this will be discussed in detail in Section 5.2.

4.2 Non-unimodular transformations

In this section, we contrast code generation for non-unimodular transformations with that for unimodular transformation. We use the following loop skeleton to illustrate the application of our techniques:

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

Figure 4: Effect of non-unimodular transformations on example 1.

Example 1:

for $i = 1$ to N

for $j = i$ to N

$A[i, j] = \cdots$

endfor

endfor

Consider the following non-unimodular transformation (with determinant 2):

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

Figure 4 shows the original iteration space and the transformed iteration space for the loop in example 1 under T.

If we generate code in this case as if T were a unimodular transformation, we generate the following incorrect code:

---

for $i = 2$ to $2N$

for $j = \max(0, i - 2N)$ to $\min(i - 2, 2N - i)$

$A\left[\frac{i-j}{2}, \frac{i+j}{2}\right] = \cdots$

endfor

endfor

The code generated above traverses all points within the boundary of the transformed iteration space in Figure 4(b). In order for the code to be correct, the step size of the inner loop should be 2 and the bounds of the inner loop have to be corrected. We note here that computation of array index expressions for the transformed nest is the same as with unimodular transformations; therefore, we will not discuss it any further.

The points in the transformed iteration space in Figure 4(b) that have to be skipped (holes) are those for which there is no corresponding point in the original iteration space, *i.e.,* points $\vec{y}$ in the transformed iteration space for which $T^{-1}\vec{y}$ is not integral are the holes. This suggests the following simple code generation strategy – in the body of the code generated as if T were unimodular, add a guard for each iteration as shown below:

---

for $i = 2$ to $2N$

  for $j = \max(0, i - 2N)$ to $\min(i - 2, 2N - i)$

    if $T^{-1} \begin{bmatrix} i \\ j \end{bmatrix} \in Z^2$ **then**

      $A\left[\frac{i-j}{2}, \frac{i+j}{2}\right] = \cdots$

  endfor

endfor

This incurs expensive runtime overhead. In the next section, we show how to generate code which avoids expensive runtime overhead. Thus, the remaining issues in code generation (discussed next) are:

1. Computation of loop step sizes.
2. Computation of correct loop bounds.

5 Code generation for non-unimodular transformations – the Hermite Normal Form approach

We generalize unimodular transformations described by [4, 19] to derive non-singular linear transformations defined by integer 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 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 $I_{n\times n}$ [14], 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 $s_j \neq 1$, then the corresponding iteration space is generated by a matrix which differs from $I_{n\times n}$ in that the entry $I_{j,j} = s_j$.

5.1 The Hermite Normal Form and loop step sizes

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 $m \times n$ integer 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, *i.e.,* $h_{ij} = 0$ 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 integer matrix.*

$(H, 0)$ is called the Hermite Normal Form *of T. For a non-singular square matrix T of size $n \times n$, 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 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 [12, 18]. The reader is referred to [12, 18] for the algorithms.

To compute the step sizes, we first compute the Hermite Normal Form, H of the transformation matrix, T. Since the columns of H and T generate the same lattice, the entries on the diagonal of H are the step sizes of the loops in the nest.

5.2 Exact loop bounds

In this section, we discuss how to correct the loop bounds generated by using the procedure in Section 4.1.2. For this, we need closed form expressions for the solution of lower triangular matrices. Fortunately, the theory of lower Hessenberg matrices [7] comes to our rescue and allows us to derive closed form solutions for inverting lower triangular matrices. The reader may note that, while the theory of Hessenberg matrices is actually an application of Cramer's rule and determinant expansions, the derived closed form solutions are quite useful in a compiler that implements the transformations.

5.2.1 Hessenberg matrices

Every valid point $\vec{y}$ in the transformed iteration space must be the image of a point $\vec{x}$ in the original iteration space. This means that $T^{-1}\vec{y}$ must be integral. Since the columns of the matrices T and H generate the same lattice, it is sufficient to check that $H^{-1}\vec{y}$ is integral for every point $\vec{y}$ in the transformed iteration space. These checks are expensive since they add to runtime overhead. Given that the step sizes are correct, if we hit the right starting point for each loop, then we generate all and only the valid points in the transformed iteration space.

Consider the problem of solving the linear system $H\vec{z} = \vec{y}$, where H is lower triangular. The problem and the solution ($\vec{z}$) are shown below for a 3×3 matrix, H.

$$
\begin{aligned}
h_{1,1}z_1 &= y_1 \\
h_{2,2}z_2 &= y_2 - h_{2,1}z_1 \\
h_{3,3}z_3 &= y_3 - h_{3,1}z_1 - h_{3,2}z_2 + h_{3,2}h_{2,1}z_1
\end{aligned}
$$

$$
\begin{aligned}
z_1 &= \frac{1}{h_{1,1}}y_1 \\
z_2 &= \frac{1}{h_{2,2}}\left(y_2 - \frac{h_{2,1}}{h_{1,1}}y_1\right) \\
z_3 &= \frac{1}{h_{3,3}}\left(y_3 - \frac{h_{3,1}}{h_{1,1}}y_1 - \frac{h_{3,2}}{h_{3,2}}y_2 + \frac{h_{3,2}}{h_{2,2}}\frac{h_{2,1}}{h_{1,1}}y_1\right)
\end{aligned}
$$

To derive closed form expressions for z_i, we use determinants of the Hessenberg matrix M associated with H defined as follows:

$$M = \begin{bmatrix} y_1 & -1 & 0 & \cdots & 0 & 0 \\ y_2 & -\frac{h_{2,1}}{h_{1,1}} & -1 & \cdots & 0 & 0 \\ y_3 & -\frac{h_{3,1}}{h_{1,1}} & -\frac{h_{3,2}}{h_{2,2}} & \cdots & 0 & 0 \\ y_4 & -\frac{h_{4,1}}{h_{1,1}} & -\frac{h_{4,2}}{h_{2,2}} & \cdots & . & . \\ . & . & . & \cdots & . & . \\ . & . & . & \cdots & . & . \\ y_{n-1} & -\frac{h_{n-1,1}}{h_{1,1}} & -\frac{h_{n-1,2}}{h_{2,2}} & \cdots & -\frac{h_{n-1,n-2}}{h_{n-2,n-2}} & -1 \\ y_n & -\frac{h_{n,1}}{h_{1,1}} & -\frac{h_{n,2}}{h_{2,2}} & \cdots & -\frac{h_{n,n-2}}{h_{n-2,n-2}} & -\frac{h_{n,n-1}}{h_{n-1,n-1}} \end{bmatrix}$$

Notice that M is obtained from H by dividing each column in H by the diagonal element on that column, then changing the sign of each element of H, dropping its last column, and adding the vector $\vec{y}$ as the new first column. Let $M[i:j]$ refer to the sub-matrix of M formed by rows i thru j and columns i thru j where $i \leq j$. For example,

$$M[1:3] = \begin{bmatrix} y_1 & -1 & 0 \\ y_2 & -\frac{h_{2,1}}{h_{1,1}} & -1 \\ y_3 & -\frac{h_{3,1}}{h_{1,1}} & -\frac{h_{3,2}}{h_{2,2}} \end{bmatrix}$$

Let $D[i:j]$ be the determinant of $M[i:j]$. Thus,

$$D[1:3] = \begin{vmatrix} y_1 & -1 & 0 \\ y_2 & -\frac{h_{2,1}}{h_{1,1}} & -1 \\ y_3 & -\frac{h_{3,1}}{h_{1,1}} & -\frac{h_{3,2}}{h_{2,2}} \end{vmatrix} = -\frac{h_{3,2}}{h_{2,2}} D[1:2] - \frac{h_{3,1}}{h_{1,1}} D[1:1] + y_3$$

In general,

$$D[1:i] = y_i - \sum_{j=1}^{i-1} \frac{h_{i,j}}{h_{j,j}} D[1:j]$$

and

$$z_i = \frac{D[1:i]}{h_{i,i}}$$

Thus, using lower Hessenberg matrices, we have closed form expressions for z_i.

5.2.2 Correction of loop bounds

Let the nest vector for the original iteration space be $\vec{x} = (x_1, \cdots, x_n)$ and $\vec{y} = (y_1, \cdots, y_n)$ be the nest vector for the transformed iteration space under T. Let l_i and u_i $(i = 1, \ldots, n)$ be the lower and upper bounds of the transformed loops as calculated using the procedure given in Section 4.1.2. Since floors and ceilings are used in computing respectively the lower and upper bounds, l_i and u_i are accurate boundaries of the transformed iteration space if T were unimodular; where as the closest integer point computed using the steps in Section 4.1.2 need not be the image of a point in the source iteration space. While no point outside the boundary computed as above belongs to the transformed iteration space, there may be points inside which do not belong. In the following discussion, we use L_i and U_i $(i = 1, \ldots, n)$ to refer to the exact bounds.

To generate only the valid points in the transformed iteration space, the loop body of the transformed nest must be executed only for $\vec{y}$ for which $H^{-1}\vec{y}$ is integral. From the above, results, it follows that for $H^{-1}\vec{x}$ to be integral, $D[1:i]$ mod $h_{i,i} = 0$ for $1 \leq i \leq n$. We note here the bounds of the outermost loop are always exact, since the image of a corner of a polyhedron is a corner under invertible linear transformations. We apply the correction to the loop bounds for loops $y_i, i = 2, \cdots, n$. The basic idea behind the correction is as follows: if l_i is not exact, then there exists an integer z_i such that $l_i \leq z_i < l_i + h_{i,i}$ and z_i is exact. We need to find z_i without searching for it. The theory of Hessenberg matrices comes to our rescue. We use the following modified form of the Hessenberg matrix:

$$M^L[1:i] = \begin{bmatrix} y_1 & -1 & 0 & \cdots & 0 & 0 \\ y_2 & -\frac{h_{2,1}}{h_{1,1}} & -1 & \cdots & 0 & 0 \\ y_3 & -\frac{h_{3,1}}{h_{1,1}} & -\frac{h_{3,2}}{h_{2,2}} & \cdots & 0 & 0 \\ . & . & . & \cdots & . & . \\ y_{i-1} & -\frac{h_{i-1,1}}{h_{1,1}} & -\frac{h_{i-1,2}}{h_{2,2}} & \cdots & -\frac{h_{i-1,i-2}}{h_{i-2,i-2}} & -1 \\ l_i & -\frac{h_{i,1}}{h_{1,1}} & -\frac{h_{i,2}}{h_{2,2}} & \cdots & -\frac{h_{i,i-2}}{h_{i-2,i-2}} & -\frac{h_{i,i-1}}{h_{i-1,i-1}} \end{bmatrix}$$

The only difference between M^L used here and the matrix M in Section 5.2.1 is that, we use l_i (the loop bound calculated by the procedure in Section 4.1.2 for loop variable y_i) instead of y_i. Let $D^L[1:i]$ be the determinant of the matrix $M^L[1:i]$. $D^U[1:i]$ and $M^U[1:i]$ are similarly defined (using u_i in the place of y_i). Let e_i^L denote the expression $D^L[1:i]$ mod $h_{i,i}$ and e_i^U denote the expression $D^U[1:i]$ mod $h_{i,i}$. If l_i is exact, then $e_i^L = 0$. If l_i is not exact, $e_i^L \neq 0$; hence, $z_i = l_i + h_{i,i} - e_i^L$. Similarly, if u_i is not exact, then the correct bound is $U_i = u_i - e_i^U$. Thus, the exact lower bound is given by

$$L_i = l_i + \left(h_{i,i} - e_i^L\right) \bmod h_{i,i}.$$

The exact upper bound is

$$U_i = u_i - e_i^U.$$

The exact bounds derived here obviate the need for expensive runtime checks used in [2, 5, 11].

5.3 Example of code generation

We refer to the loop in Example 1; for convenience, we reproduce the example below:

Example 1:

for $i = 1$ to N

  for $j = i$ to N

    $A[i, j] = \cdots$

  endfor

endfor

Consider the following non-unimodular transformation

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

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:

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

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:

$$[P, \vec{q}] = \left[\begin{array}{rr|r} -1 & 0 & -1 \\ 1 & 0 & N \\ 1 & -1 & 0 \\ 0 & 1 & N \end{array}\right]$$

In order to compute the loop bounds, we first compute $[PT^{-1}, \vec{q}]$:

$$[PT^{-1}, \vec{q}] = \left[\begin{array}{rr|r} -\frac{1}{2} & \frac{1}{2} & -1 \\ \frac{1}{2} & -\frac{1}{2} & N \\ 0 & -1 & 0 \\ \frac{1}{2} & \frac{1}{2} & N \end{array}\right]$$

which can be scaled to become:

$$[PT^{-1}, \vec{q}] = \left[\begin{array}{rr|r} -1 & 1 & -2 \\ 1 & -1 & 2N \\ 0 & -1 & 0 \\ 1 & 1 & 2N \end{array}\right]$$

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

$$\left[\begin{array}{rr|r} -1 & 0 & -2 \\ 1 & 0 & 2N \end{array}\right]$$

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. The (incorrect) inner loop bounds are:

$$l_2 = \max(0, i - 2N) \text{ and } u_2 = \min(2N - i, i - 2)$$

In all cases, the lower bound as calculated is 0. The $M^L[1:2]$ and $M^U[1:2]$ matrices are:

$$M^L[1:2] = \begin{bmatrix} i & -1 \\ l_2 & 1 \end{bmatrix} \qquad M^U[1:2] = \begin{bmatrix} i & -1 \\ u_2 & 1 \end{bmatrix}$$

Thus, $D^L[1:2] = i + l_2$ and $D^U[1:2] = i + u_2$. Therefore, the corrected loop bounds are:

$$\begin{aligned} L_2 &= l_2 + \left(h_{2,2} - (D^L[1:2] \bmod h_{2,2})\right) \bmod h_{2,2} = \max(0, i-2N) + (2 - (i+l_2) \bmod 2) \bmod 2 \\ U_2 &= u_2 - \left(D^U[1:2] \bmod h_{2,2}\right) = \min(i-2, 2N-i) + (i+l_2) \bmod 2 \end{aligned}$$

Since $l_2 = 0$, we can write L_2 as $L_2 = i \bmod 2$. The correct transformed code:

Example 1 (transformed):

for i=2 to 2N by 1

  for j= $\max(0, i-2N)$ $+i$ mod 2 to $\min(i-2, 2N-i)$ $-i$ mod 2 by 2

    $A\left[\frac{i-j}{2}, \frac{i+j}{2}\right] = \cdots$

  endfor

endfor

We note here that the correction factor applied to loop bounds follows a regular pattern. In ongoing work [15], we have characterized the sequence of applied correction factors as a finite-state machine that depends on the computed loop step size and the shape of the source iteration space. The use of the finite-state machine reduces runtime overhead.

6 Application of the code generation algorithm

6.1 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 n-dimensional iteration spaces 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 [2] here. Let L be an $n \times n$ matrix specifying the lattice of tile origins for a loop nest with a computation domain given by $[P, \vec{q}]$. We show how to generate the origins of the tile sets.

We use the following loop nest (from [2]):

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 by the columns of L, where L is:

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

The problem of enumerating tile origins is equivalent to applying the transformation denoted by the matrix L^{-1}:

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

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

$$[P, \vec{q}] = \left[\begin{array}{cc|c} -1 & 0 & -1 \\ 1 & 0 & N \\ 0 & -1 & -1 \\ 1 & 1 & N+1 \end{array}\right]$$

In order to compute the loop bounds, we first compute PL (since $L = (L^{-1})^{-1}$).

$$[PL, \vec{q}] = \left[\begin{array}{cc|c} 3 & -9 & -1 \\ -3 & 9 & N \\ -6 & 9 & -1 \\ 3 & 0 & N+1 \end{array}\right].$$

If we replace row 1 by the sum of rows 1 and 3, we get the bounds of the outer loop as $\frac{2}{3}$ and $\frac{N+1}{3}$. Similarly, we can derive the loop bounds of the inner loop. The loop that traverses the origins of non-empty tiles is:

---

for $x = \frac{2}{3}$ to $\frac{N+1}{3}$

for $y = \frac{1+3x}{9}$ to $min(\frac{N+3x}{9}, \frac{6x-1}{9})$

. . .

endfor

endfor

If we shift the outer loop to 0, we have:

for $x = 0$ to $\frac{N-1}{3}$
    for $y = \frac{1+x}{3}$ to $min(\frac{N+3x+2}{9}, \frac{2x+1}{3})$
        $\cdots$
    endfor
endfor

The points in the transformed iteration space are the origins of non-empty tiles.

6.2 Distributed memory code generation

Current efforts in distributed memory code generation start with a user-defined decomposition of data structures [8]. 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" [8]. 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* [8].

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 $i = 1$ to N do
        for $j = 1$ to N
            $D[4i - 2j, i + j] = \cdots$
        endfor
    endfor

Assume for this example that the locally owned part of B in some processor is given by $D[(a_{l,1} : a_{h,1}), (a_{l,2} :$

$a_{h,2})]$. The subscript functions can be represented in matrix form as $A\vec{x} + \vec{b}$, or

$$\begin{bmatrix} 4 & -2 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \end{bmatrix},$$

where $\vec{c} = (c_1, c_2)^T$ corresponds to a point in the data space and $(i, j)^T$ corresponds to a point in the iteration space. In order to enforce the *owner compute* rule, $a_{l,1} \leq c_1 \leq a_{h,1}$ and $a_{l,2} \leq c_2 \leq a_{h,2}$. That is,

$$a_{l,1} \leq 4i - 2j \leq a_{h,1}$$

$$a_{l,2} \leq i + j \leq a_{h,2}.$$

Recall that if the access function is $A\vec{x} + \vec{b}$, on applying a transformation T to the loop, the access function becomes $AT^{-1}\vec{y} + \vec{b}$ where $\vec{y} = T\vec{x}$ is a vector of the transformed loop indices. Therefore if a transformation $T = A$ is legal, the access function becomes $AA^{-1}\vec{y} + \vec{b} = \vec{y} + \vec{b}$; this renders distributed memory code generation easy. For Example 3, to generate distributed memory code for this example, we first apply the transformation T which is the same as the data access function represented as a matrix:

$$T = \begin{bmatrix} 4 & -2 \\ 1 & 1 \end{bmatrix}$$

The Hermite Normal Form of T is

$$H = \begin{bmatrix} 2 & 0 \\ -1 & 3 \end{bmatrix}$$

The output generated by our algorithm for T applied to the loop in Example 3 is:

---

for $x = 4 - 2N$ to $4N - 2$ by 2 do

$\quad l = \max\left(\lceil \frac{6-x}{2} \rceil, \lceil \frac{6+x}{4} \rceil\right)$

$\quad L = l + (3 - (l + \frac{x}{2}) \bmod 3) \bmod 3$

$\quad u = \min\left(\lfloor \frac{6N+x}{4} \rfloor, \lfloor \frac{6N-x}{2} \rfloor\right)$

$\quad U = u - (u + \frac{x}{2}) \bmod 3$

$\quad$ **for** $y = L$ to U by 3 do

$\quad\quad$ /* access local data */

$\quad\quad D[x, y] = \cdots$

$\quad$ **endfor**

endfor

In order to ensure that only local data is accessed, we need to include those in the bounds. We get:

$x_{\text{low}} = \max(4 - 2N, a_{l,1})$

$X_{\text{LOW}} = x_{\text{low}} + \big(x_{\text{low}} \bmod 2 - (4 - 2N) \bmod 2\big) \bmod 2$

$x_{\text{high}} = \min(4N - 2, a_{h,1})$

$X_{\text{HIGH}} = x_{\text{high}} - x_{\text{high}} \bmod 2$

for $x = X_{\text{LOW}}$ to X_{HIGH} by 2 do

$l = \max\left(\lceil\frac{6-x}{2}\rceil, \lceil\frac{6+x}{4}\rceil\right)$

$L = l + \big(3 - (l + \frac{x}{2}) \bmod 3\big) \bmod 3$

$u = \min\left(\lfloor\frac{6N+x}{4}\rfloor, \lfloor\frac{6N-x}{2}\rfloor\right)$

$U = u - (u + \frac{x}{2}) \bmod 3$

$L = \max(L, a_{l,2})$

$U = \min(U, a_{h,2})$

for $y = L$ to U by 3 do

/* access local data */

$D[x, y] = \cdots$

endfor

endfor

In the following example, we assume that the transformation A (the matrix part of the access function) is not legal and show how to generate code by just adjusting loop bounds. Consider the loop shown below:

Example 4:

for $i = l_1$ to u_1

for $j = l_2$ to u_2

$D[2i + j - 3, i + j + 2] = \cdots$

endfor

endfor

Assume for this example that the locally owned part of D in some processor is given by $D[(x_1 : x_2), (y_1 :$

$y_2)]$. 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, $x_1 \le c_1 \le x_2$ and $y_1 \le c_2 \le y_2$. That is,

$$x_1 \le 2i + j - 3 \le x_2$$

$$y_1 \le i + j + 2 \le y_2$$

which is equivalent to the following system of inequalities:

$$\left[\begin{array}{rr|c} -2 & -1 & -x_1 - 3 \\ 2 & 1 & x_2 + 3 \\ -1 & -1 & -y_1 + 2 \\ 1 & 1 & y_2 - 2 \end{array}\right]$$

We reduce the above system to row echelon form that results in the following local iteration set for the processor avoiding expensive guards:

---

for $i = \max(l_1, x_1 - y_2 + 5)$ to $\min(u_1, x_2 - y_1 + 5)$

  for $j = \max(l_2, -2i + x_1 + 3, -i + y_1 - 2)$ to $\min(u_2, -2i + x_2 + 3, -i + y_2 - 2)$

    /* access local data */

    $D[2i + j - 3, i + j + 2] = \cdots$

  endfor

endfor

The code generation technique is also applicable for `cyclic` and `block-cyclic` forms of data decomposition as well. 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.

7 Related work

Dowling [6] presents an integer programming solution to the problem of optimal code parallelization using unimodular matrices that transform a given nested loop into parallel form. Banerjee [4] and Wolf and Lam [19] present unimodular transformations of loop nests and algorithms to derive transformations for maximal parallelism. Barnett and Lengauer [5] conclude that unimodularity is not essential; to derive code for non-unimodular transformations, they generate piecewise loop bounds which are not exact. Ancourt and Irigoin [2] and Lu [11] use conditionals in their transformed code to ensure correctness of the transformation. The techniques in this paper obviate the need for such conditionals. The techniques presented in [2, 11, 5] result in runtime checks. Wolfe [22] 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 integer matrices whose determinant is an arbitrary integer. We show an example of the need for non-unimodular transformations; the transformations are derived through modifying the step size of the loops. Li and Pingali [10] 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. Their solution decomposes a compound transformation represented by a non-singular matrix into a unimodular transformation followed by a non-unimodular transformation; the second transformation does not change the relative order of execution of iterations. The loop bounds generated involve complex integer operations. In ongoing work [15], we have characterized the sequence of correction factors as a finite-state machine. This reduces the overhead arising from expensive integer operations in loop bound calculations. In addition, their technique does not directly work for apllications such as distributed memory code generation.

8 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. We presented techniques for code generation for non-unimodular loop transformations. Unimodular transformations [4, 19] 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 "skip" 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, and deriving local iteration sets in distributed memory code generation. The Hermite Normal Form is also useful in dependence analysis. The technique presented here holds promise in creating a framework for optimizing programs for a wide variety of architectures.

Acknowledgments

We thank the anonymous referees for their comments on an earlier draft of the paper.

References

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

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

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

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

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

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

[7] D. Heller. A Determinant Theorem with Applications to Parallel Algorithms. *SIAM Journal of Numerical Analysis,* Vol. 11, pp. 484–496, 1974.

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

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

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

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

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

[13] J. Ramanujam. *Compile-time Techniques for Parallel Execution of Loops on Distributed Memory Multiprocessors.* Ph.D. Thesis, The Ohio State University, (UMI–9111789) September 1990.

[14] J. Ramanujam. A linear algebraic view of loop transformations and their interaction. In *Proc. Fifth SIAM Conference on Parallel Processing for Scientific Computing,* Chapter 78, pp. 543–548, 1992.

[15] J. Ramanujam. Efficient Code Generation for Loop Transformations. *In preparation,* April 1994.

[16] J. Ramanujam and P. Sadayappan. Tiling Multidimensional Iteration Spaces for Nonshared Memory Machines. *Journal of Parallel and Distributed Computing,* Vol. 16, No. 2, pp. 108–120, October 1992.

[17] R. Schreiber and J. Dongarra. Automatic Blocking of Nested Loops. Technical Report, University of Tennessee, Knoxville, TN, August 1990.

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

[19] M. Wolf and M. Lam. A Loop Transformation Theory and an Algorithm to Maximize Parallelism. *IEEE Trans. Parallel and Distributed Syst.,* Vol. 2, No. 4, pp. 452–471, October 1991.

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

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

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

[23] M. Wolfe and C. Tseng. The power test for data dependence. *IEEE Trans. Parallel and Distributed Systems.,* Vol. 3, No. 5, pp. 591–601, September 1992.

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