Transformations for Imperfectly Nested Loops

Induprakas Kodukula
Keshav Pingali
Department of Computer Science,
Cornell University, Ithaca, NY 14853.
{prakas,pingali}@cs.cornell.edu

October 20, 1996

Abstract

Loop transformations are critical for compiling high-performance code for modern computers. Existing work has focused on transformations for perfectly nested loops (that is, loops in which all assignment statements are contained within the innermost loop of a loop nest). In practice, most loop nests, such as those in matrix factorization codes, are imperfectly nested. In some programs, imperfectly nested loops can be converted into perfectly nested loops by loop distribution, but this is not always legal. In this paper, we present an approach to transforming imperfectly nested loops directly. Our approach is an extension of the linear loop transformation framework for perfectly nested loops, and it models permutation, reversal, skewing, scaling, alignment, distribution and jamming.

1 Introduction

Modern compilers perform a variety of loop transformations, like permutation, skewing, reversal, scaling, distribution and jamming, to generate high quality code for high-performance computers [14]. Determining an optimal sequence of loop transformations for enhancing parallelism or locality of reference is a very difficult problem. For perfectly nested loops (that is, loops in which all assignment statements are contained within the innermost loop), polyhedral methods can be used to synthesize sequences of linear loop transformations (permutation, skewing, reversal and scaling) for enhancing parallelism and locality [1, 3, 4, 13, 10, 12, 2]. The key idea is to model the iterations of the loop nest as points in an integer lattice, and to model linear loop transformations as nonsingular matrices mapping one lattice to another. A sequence of loop transformations is modeled by the product of matrices representing the individual transformations; since nonsingular matrices are closed

[1]This research was supported by NSF grant CCR-9503199, ONR grant N00014-93-1-0103, and a grant from Hewlett-Packard Corporation.

under product, this means that a sequence of linear loop transformations is also represented by a nonsingular matrix. The problem of finding an optimal sequence of linear loop transformations is thus reduced to the problem of finding an integer matrix that satisfies some desired property. This formulation has permitted the full machinery of matrix methods and lattice theory to be applied to the loop transformation problem for perfectly nested loops, and the resulting technology is mature enough that it has been incorporated into many industrial compilers.

Unfortunately, most loops in real programs are **imperfectly** nested — that is, the assignment statements are nested in some but not all of the loops in the loop nest. The simplest examples are matrix factorization codes. For instance, Cholesky factorization is usually written as an imperfectly nested loop with three loops. All six permutations of these three loops compute the same result, but their performance, even on sequential machines, can be quite different. The polyhedral methods that were developed for perfectly nested loops cannot be applied directly to permute the loops in Cholesky factorization. In some programs, loop distribution can be applied to transform an imperfectly nested loop into a set of perfectly nested ones. However, loop distribution is not always legal; in particular, it is not legal in any of the matrix factorization codes.

Therefore, we are faced with two problems. First, how do we specify transformations of imperfectly nested loops? Second, how do we find legal and desirable transformations? The usual way to solve the first problem is to use techniques developed by the systolic array community for scheduling statements in loop nests on systolic arrays. These schedules specify mappings from statement instances to processor/time axes; these mappings are usually restricted to be affine functions of loop variables [9]. It is straight-forward to interpret these schedules or mappings as loop transformations in which each assignment statement in a loop nest is transformed by a possibly different linear (or pseudo-linear) function to a target iteration space. These techniques were extended by Feautrier in his theory of **schedules** in multi-dimensional time [6, 7]; a related approach is Kelly and Pugh's **mappings**[8]. The second problem — finding legal and desirable transformations — is solved by searching the space of transformations. However, this process is usually very expensive for two reasons. First, standard dependence abstractions like distance and direction cannot be used; instead, relatively expensive tests based on techniques like parametric integer programming are necessary. Second, the frameworks themselves provide little help in producing desirable transformations. This should be contrasted with the case of perfectly nested loops. In that case, determining a parallel outermost loop merely requires finding a vector in the null space of the columns of the dependence matrix [3]. Similarly, general 'completion procedures' have been developed for producing complete transformations from partial ones [10].

In this paper, we present an approach to transforming imperfectly nested loops that trades off generality for simplicity. Roughly speaking, if two assignment statements are contained within the same set of loops, we require that they be transformed the same way (although statement alignment may 'shift' the iterations of these statements by different amounts). Although this is more restrictive than schedules or mappings, it is sufficient to reason about loop permutations in matrix factorization codes. The advantage of this restriction is that it permits us to extend the technology for transforming perfectly nested loops to imperfectly nested loops in a more or less straight-forward way. In particular, we can use standard dependence abstractions like distances and directions, and loop transformations can be modeled by matrices. The rest of the paper is organized as follows. In Section 2, we model the execution of statements in an imperfectly nested loop using integer vectors called **instance vectors** which are a generalization of iteration vectors used in the context of perfectly nested loops. In Section 3, we present our version of dependence distances and directions, and show how these can be computed using standard integer linear programming techniques. In Section 4, we show how permutation, skewing, reversal, scaling, alignment, statement reordering, distribution and jamming can be modeled as linear transformations in our framework. In the rest of the paper, we do not consider distribution and jamming. In Section 5, we show how code can be generated from a matrix representing a sequence of loop transformations. In Section 6, we discuss a completion procedure which generates a complete transformation from a partial one; for lack of space, we omit details but show how it works using permutation of the loops in Cholesky factorization. Finally, we discuss future work in Section 7.

2 Iteration Space Formulation

Since a statement in a loop is executed many times, we need a way of identifying a particular dynamic instance of a given statement. In the context of perfectly nested loops, it is standard to use an **iteration vector**, which is simply a sequence of integers specifying the iteration number for each loop in the loop nest. For imperfectly nested loops, we generalize this notion to **instance vectors**; instance vectors carry additional information to identify the statement under consideration. Like iteration vectors, instance vectors have two key properties. The first property is that for a given program, all vectors have the same length, so it is meaningful to add or subtract vectors. The second property is that the execution order between two dynamic instances corresponds to lexicographic order on the corresponding instance vectors.

2.1 Program Order

We use the following pseudo-code as the running example in this section.

```
do I = 1..N
  do J = f(I)..g(I)
    S1
    S2
  enddo
  S3
enddo
```

Figure 1(a) shows the abstract syntax tree (AST) for this program. Internal nodes in trees represent loops, while leaves represent 'atomic' statements (for example, assignment statements or conditionals whose internal structure is of no interest for loop transformations). As is standard, subtree structure reflects syntactic nesting while the left-to-right order of the children of a node reflects the sequential execution order of those children. We now define a reflexive relation $\preceq_S$ that captures syntactic order in a program.

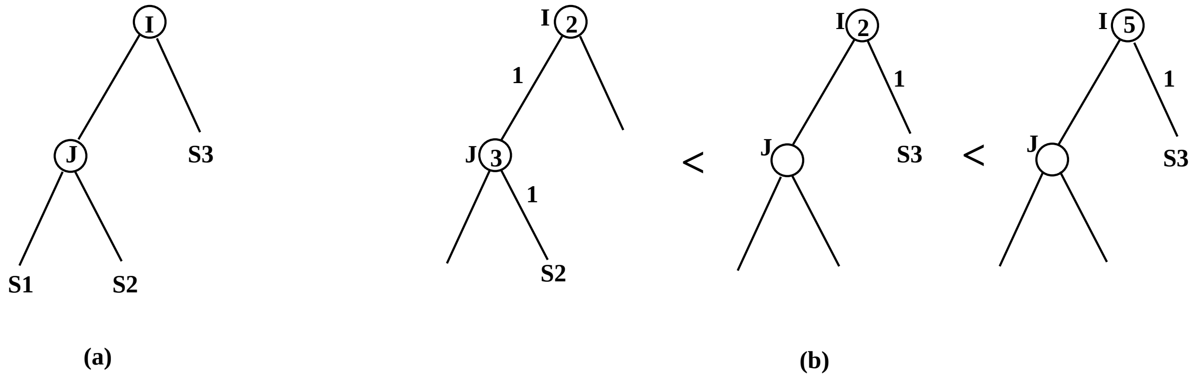

Figure 1: Abstract Syntax Trees and Program Order

Definition 1 *An atomic statement S_1 is said to occur syntactically before an atomic statement S_2, written as $S_1 \preceq_S S_2$, if S_1 is encountered before S_2 in a depth-first walk of the AST.*

A dynamic instance of a statement in an imperfectly nested loop is specified by the values of the loop index variables of loops surrounding that statement. In terms of the AST, this corresponds to an assignment of integers to all internal nodes on the path in the AST from root to that statement. It is convenient to view a dynamic instance of a statement as a partially labeled AST. To identify the statement in the AST, the edges on the path from root to that statement are labeled 1. The values of the relevant loop index variables are specified by labeling the corresponding internal nodes on that path with integers. Figure 1 shows some labeled AST's for the running example. The leftmost AST in Figure 1(b) corresponds to an execution of S2 with I=2 and J=3. The middle AST corresponds to an execution of S3 with I=3 and the rightmost AST corresponds to execution of S3 with I=5. These three executions are ordered in time with respect to each other, as is shown in Figure 1(b). For future reference, we define the $\leq$ order on dynamic instances.

Definition 2 *Let DI_1 and DI_2 be dynamic instances of statements S_1 and S_2 respectively. Let P_1 and P_2 be integer vectors containing the labels in DI_1 and DI_2 respectively for the common loops of S_1 and S_2, in outside-in order. $DI_1 \leq DI_2$ if (i) $P_1 < P_2$ or (ii) $P_1 = P_2$ and $S_1 \preceq_S S_2$.*

Dynamic instances are mapped to instance vectors by a function **L**. In the first step, a partially labeled AST T is converted to a fully labeled AST F by a procedure **M** whose behavior can be described as follows.

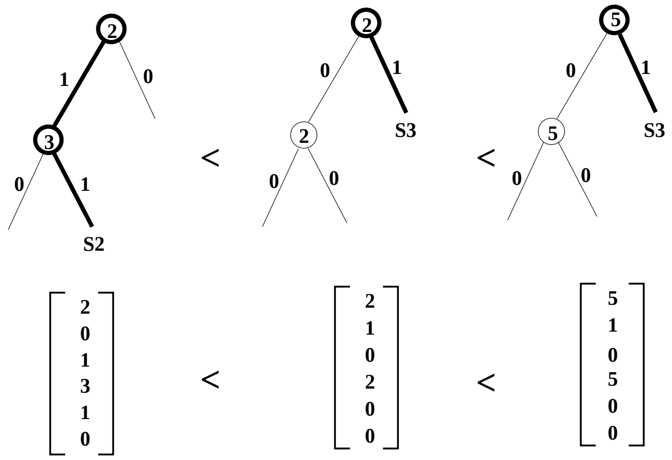

Figure 2: Instance Vectors and Lexicographic Order

1. **Every unlabeled edge in T is assigned the label 0.**
2. **Every leaf node in T is assigned the label ϵ.**
3. **Every unlabeled internal node n is assigned the label of its nearest labeled ancestor in T.**

If T is a partially labeled AST, the expression $\mathbf{M}(T)$ will denote the fully labeled AST generated by procedure $\mathbf{M}$.

In the second step, we collect all labels in a fully labeled AST into a vector. This is accomplished by walking over the tree in depth-first order, visiting the children of a node in right to left order, and concatenating labels. This is defined formally using a function $\mathbf{R}(N)$, defined below, which collects the labels in the subtree below node n.

$$\mathbf{R}(N) = \begin{array}{ll} \epsilon & \text{if N is a leaf,} \\ Label(N)//Label(e_m)//..//Label(e_1)//R(n_m)//..//R(n_1) & \text{otherwise.} \end{array} \quad (1)$$

where m is the number of children of node N, $n_1...n_m$ are the children of N in left to right order in the AST, and edge e_i is the edge from N to child n_i. The operation // denotes vector concatenation. $\mathbf{R}(root)$ is said to be the **Instance vector** representing the particular statement execution.

Definition 3 *The function* $\mathbf{L}$ *maps a partially labeled AST to an instance vector.*
$\mathbf{L}$*(T: partially labeled AST)* $= \mathbf{R}$*(root(*$\mathbf{M}$*(T)));.*

Instance vectors are integer vectors, and we will let $\leq$ denote lexicographic order on these vectors.

Theorem 1 **L** *is one-to-one. Furthermore, if* E_1 *and* E_2 *are two dynamic instances and* E_1 *precedes* E_2 *in execution order, then* $\mathbf{L}(E_1) \leq \mathbf{L}(E_2)$.

Proof: Follows immediately from the definition of **L**. □

Definition 4 *For any dynamic instance* DV_s*, entries of the instance vector* $\mathbf{L}(DV_s)$ *for nodes labeled by procedure* **M** *are referred to as* **padded positions** *of that instance vector.*

For example, in Figure 2, the entries for the J loop in instance vectors for dynamic instances of S3 are padded positions. Intuitively, these instance vectors can be viewed as an embedding of the iteration space of statement S3 into the 'global' iteration space of the nested loops (for example, iteration I of statement S3 is mapped to iteration (I, I) of the nested loop). Given our choice of padding, this corresponds to a 'diagonal' embedding of the lower dimensional space into the higher dimensional one. There are other reasonable ways to define this embedding, but we have not explored these alternatives.

Lemma 1 *All dynamic instances of a particular statement have the same padded positions*

Proof: Follows from Definition 4 and Equation 1. □

Lemma 2 *The instance vectors for a perfectly nested loop have* **no** *padded positions.*

Proof: Follows from Definition 4 and Equation 1. □

In the rest of the paper, we use the term padded positions of S to mean the padded positions of all the instance vectors of all the dynamic instances of S. For future discussion, it is useful to define a function $\mathbf{L}^{-1}$ which converts an instance vector back to a dynamic instance.

Definition 5 $\mathbf{L}^{-1}(IV : InstanceVector, A : AST)$ *takes an instance vector* IV*, and an* AST*, and returns the partially labeled* AST P *corresponding to* IV*. It can be described as follows.*

1. *Identify the relevant statement in the* AST *by examining* IV*.*
2. *Label all edges from the root of the* AST *to statement* S *with* 1*.*
3. *Label each loop surrounding* S *with the entry for that loop in* IV*.*

2.2 Optimizing Single edges

A simple optimization on edge label assignment permits instance vectors to reduce to iteration vectors for perfectly nested loops. As described above, edge labels serve to identify the path from the root to a particular statement. However, if a node N has only one edge E coming out of it (i.e. the loop corresponding to N has only one statement), then a label on E is redundant. To eliminate this edge from the instance vector, we use ϵ for its label.

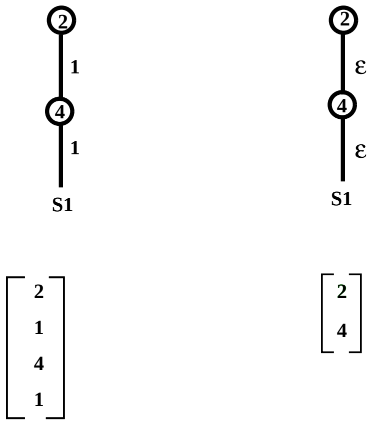

Figure 3: Optimizing Instance Vectors

```
do I = 1..N
  do J = I+1..N
    S1: A(J) = A(J) / A(I)
  end do
end do
```

Figure 3 shows a perfectly nested loop, and instance vectors with and without this optimization. Note that instance vectors are identical to iteration vectors once this optimization has been carried out. In the rest of the paper, we will assume that this optimization has been performed.

3 Dependence Analysis

We now discuss the representation of dependences in our framework, and show how dependences may be computed. An advantage of our framework is that we can use standard dependence tests for computing distance/direction vectors. The following highly simplified version of Cholesky factorization is the running example in this section.

```
do I = 1..N
  S1: A(I) = sqrt (A(I))
  do J = I+1..N
    S2: A(J) = A(J) / A(I)
  end do
end do
```

In this program, there is a flow dependence from S1 to S2 because S1 writes to A and S2 reads from A. Suppose that S1 writes to some array location in iteration I_w of the outer loop, and that this location is read by statement S2 in iteration (I_r, J_r). The instance vector for the statement execution performing the write is $[I_w, 0, 1, I_w]'$. Similarly, the instance vector for the statement execution performing the read is $[I_r, 1, 0, J_r]'$. The difference between these two instance vectors is $[I_r - I_w, 1, -1, J_r - I_w]'$. To compute the appropriate distance/direction vectors corresponding to this difference, we set up the following set of affine constraints.

$$\begin{aligned} 1 \leq I_r \leq N, I_r < J_r \leq N, 1 \leq I_w &\leq N(\text{loop bounds}) \\ I_w &\leq I_r(\text{read after write}) \\ I_r &= I_w(\text{same array location}) \end{aligned} \tag{2}$$

Note that these are integer linear inequalities similar to those that arise in the context of dependence analysis of perfectly nested loops. To obtain the appropriate direction information, we introduce two new variables as follows:

$$\Delta 1 = I_r - I_w, \Delta 2 = J_r - I_w \tag{3}$$

We treat Equations 2 and 3 as a single system of equations, and project the solution of this system onto $\Delta 1$ and $\Delta 2$, using any integer linear programming tool, such as the Omega tool-kit [11]. In our example, it is easy to see that $\Delta 1 = 0$ and $\Delta 2 = +$. Therefore, the flow dependence in the above example will be represented in our framework as $[0, 1, -1, +]'$.

Using a similar procedure, we can determine the other dependences in this code. These dependences can be collected into a dependence matrix; for our example, this matrix is the following: $\begin{bmatrix} 0 & 1 & 0 \\ 1 & -1 & 0 \\ -1 & 1 & 0 \\ + & 0 & 1 \end{bmatrix}$.

The general procedure for computing dependences performs this analysis for all pairs of reads and writes in a program; for lack of space, a detailed description is omitted.

4 Transformations

In this section, we show how matrices may be used to model loop transformations in our framework. The transformations we can model include (i) imperfectly nested loop permutation, skewing, reversal and scaling, (ii) statement reordering, (iii) statement alignment, and (iv) distribution and jamming.

4.1 An overview

There are three subtle issues that arise in transforming imperfectly nested loops, and we discuss them using the simplified version of Cholesky factorization shown again below.

```
do I = 1..N
  S1: A(I) = sqrt (A(I))
  do J = I+1..N
    S2: A(J) = A(J) / A(I)
  end do
end do
```

We note first that in the context of imperfectly nested loops, transformations that operate on multiple loops, such as permutation and skewing, are not uniquely defined. In the example shown above, it is clear what permuting the I and J loops means for statement S2 since it is nested within both loops, but what does it mean for statement S1? A plausible definition is that the index space of S1 should not be changed by this loop permutation. However, there is no particular reason to prefer this definition; indeed, the commonly used strategy of performing transformations after sinking all statements into the innermost loop will in general change the index space of S1. We take a similar approach — as discussed in Section 2, our instance vectors define an embedding of the iteration space of S1 into a global iteration space, in effect, and transformations to this global iteration space may result in transformations to the index space of S1. In particular, loop permutation is represented simply by a permutation matrix which permutes instance vector positions that correspond to the loops being interchanged. The matrix and transformed instance vectors for the interchange of the I and J loops are shown below. It is coincidental that instance vectors of S1 are left unchanged by permutation in this example.

$$\begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} * \begin{bmatrix} I & I \\ 0 & 1 \\ 1 & 0 \\ I & J \end{bmatrix} = \begin{bmatrix} I & J \\ 0 & 1 \\ 1 & 0 \\ I & I \end{bmatrix}$$

From this discussion, it should be clear how other loop transformations are represented by matrices in our framework. For example, skewing the outer loop by the inner loop in our running example is represented as follows:

$$\begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} I & I \\ 0 & 1 \\ 1 & 0 \\ I & J \end{bmatrix} = \begin{bmatrix} 0 & I-J \\ 0 & 1 \\ 1 & 0 \\ I & J \end{bmatrix}$$

Skewing transforms instance vectors of S2 just as it does in perfectly nested loops. However, its effect on instance vectors of S1 is subtle. First, note that the new outer loop index is 0; this means that all instances of S1 must be executed in the very first iteration of the new outer loop. This may seem surprising but it is a consequence of the 'diagonal embedding' of the iterations of S1 into the global iteration space: the new outer loop is orthogonal to this diagonal, so all iterations of S1 are done in the very first iteration of the outer loop. Our code generation procedure, described in Section 5, introduces an extra loop around S1 to take case of this enumeration. A second point to note is that we do not require that padded positions be transformed by the transformation matrix. In the transformed instance vectors

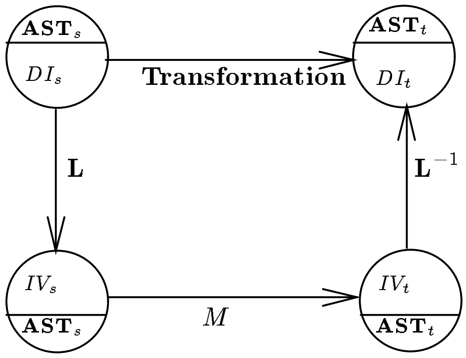

Figure 4: Role of Matrices in Representing Transformations

for `S1`, the entry in the outer loop position is `0`, but the entry in the inner loop position is `I`, and not `0`. This has an impact on how we test for the legality of a transformation; clearly, it is not sufficient to test that transformed dependence vectors are lexicographically positive, as in the case of perfectly nested loops. The test for legality is described in Section 5.

The role of matrices in our framework is made precise in Figure 4. A loop transformation T maps a dynamic instance of a statement in the source program (shown as DI_s) to a dynamic instance in the target program (shown as DI_t). We can convert the dynamic instance DI_s into an instance vector IV_s using the $\mathbf{L}$ operator discussed in Section 2. M, the matrix representing the transformation T, maps the old AST to a new AST, and maps every instance vector IV_s to a new instance vector IV_t such that the dynamic instance obtained by applying the $\mathbf{L}^{-1}$ operator to IV_t is precisely DI_t. This is a weaker assertion than stating that $\mathbf{L}(T(DI_s) = M * \mathbf{L}(DI_s)$. As mentioned before, M is not required to transform padded positions of iteration vectors consistently, so the entries in the padded positions of IV_t may be different in general from the entries in the padded positions of $\mathbf{L}(DI_t)$.

We have already discussed permutation and skewing. For completeness, we note that loop reversal is represented by an identity matrix with one change — the diagonal entry of the row corresponding to the loop being reversed has a -1. Loop scaling is represented by an identity matrix with one change — the diagonal entry of the row being scaled has an entry equal to the scale factor.

4.2 AST Transformations

Statement reordering changes the structure of the AST. In our framework, it is represented by a permutation matrix where the permutation matrix interchanges the positions corresponding to the statements that are being interchanged. The following example shows the matrix that reorders the J loop and S1 (both contained in the outer I loop) from the Cholesky example.

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} I & I \\ 0 & 1 \\ 1 & 0 \\ I & J \end{bmatrix} = \begin{bmatrix} I & I \\ 1 & 0 \\ 0 & 1 \\ I & J \end{bmatrix}$$

Loop distribution and jamming are represented by non-square matrices. Here is a version of the simplified Cholesky fragment after loop distribution.

```
do I = 1..N
  S1;
enddo
do I = 1..N
  do J = I+1..N
    S2;
  enddo
enddo
```

In the AST, loop distribution corresponds to splitting an edge of the AST into two and represents the replication of the edge and the entire subtree that may be split as a consequence. In the particular case of distribution illustrated above, the transformation is given by

$$\begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} I & I \\ 0 & 1 \\ 1 & 0 \\ I & J \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ I & I \\ I & I \\ I & J \end{bmatrix}$$

We illustrate how loop jamming works by transforming the simplified Cholesky code after loop distribution back to its original form. Conceptually, this fuses two subtrees of the AST into a single subtree. In this particular instance, the transformation is represented by the matrix

$$\begin{bmatrix} 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} 0 & 1 \\ 1 & 0 \\ I & 0 \\ 0 & I \\ 0 & J \end{bmatrix} = \begin{bmatrix} I & I \\ 0 & 1 \\ 1 & 0 \\ 0 & J \end{bmatrix}$$

4.3 Statement Alignment

Statement alignment of a particular statement with respect to a loop surrounding it is the identity matrix with one additional entry. The additional entry appears in the row positions corresponding to the loop and the column position corresponding to the column. The value of the entry corresponds to the amount that the statement gets aligned with respect to the loop. Here is alignment of S1 in the Cholesky code fragment with respect to the I loop by +1:

$$\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} * \begin{bmatrix} I & I \\ 0 & 1 \\ 1 & 0 \\ I & J \end{bmatrix} = \begin{bmatrix} I+1 & I \\ 0 & 1 \\ 1 & 0 \\ I & J \end{bmatrix}$$

5 Code Generation

In this section, we restrict our discussion to statement reordering, and permutation,skewing,reversal and scaling of imperfectly nested loops. We describe how to check for legality of transformations, and then show how to generate code from matrices that represent legal transformations. More generally, using our framework, we can show that all six permutations of the loops in Cholesky factorization are legal.

5.1 Legality

Given an initial AST (say AST_i), a dependence matrix D, and a transformation matrix M, how do we check that M is a legal transformation? Intuitively, M is legal if (i) we can generate a new AST from it, (ii) dependent dynamic instances in the source program are properly ordered in the transformed program, and (iii) there is a one-to-one and onto map between dynamic instances in the source and transformed programs. We explain each of these points in detail next.

5.2 Generating the new AST

Since statement reordering is the only transformation that changes the AST, we can show that a legal transformation matrix M must have a certain 'block structure' from which the AST can be recovered easily. The key observation is the following. The first row of M describes how the outermost loop is transformed. If this loop has c children, the submatrix $P = M[2..(c+1), 2..(c+1)]$ must be an $c \times c$ permutation matrix that describes how the children of the root node are permuted by statement reordering. The rest of the matrix M (that is, $M[(c+2)..n, (c+2)..n]$) describes how these c children are themselves transformed; therefore, this submatrix can be decomposed into c block matrices, such that the block structure of this submatrix has the same shape as the permutation matrix P. From these c block matrices, we can recursively determine the portion of new AST structure rooted at each of the c children of the root node. Figure 5 shows the case when $c = 3$. Any transformation matrix M that does not have this block structure is clearly illegal.

The pseudo-code for generating the transformed AST is given in Figure 6.

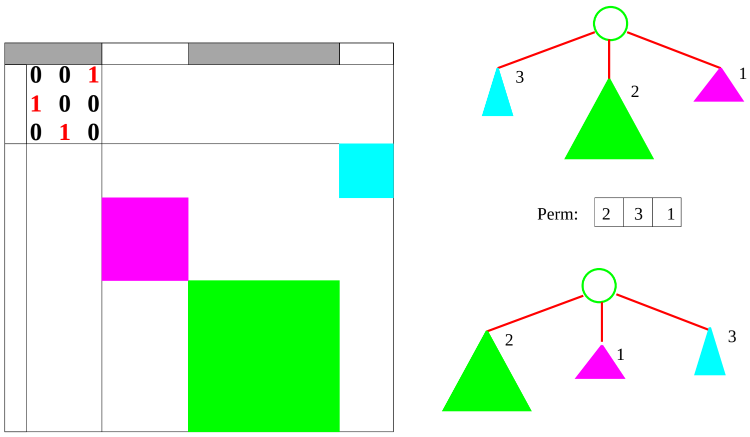

Figure 5: Block Structure of Transformation Matrix

```
Procedure NewAST (M, n) : returnsAST
{
    /* Discover the AST structure for the given transformation */
    /* M is the transformation matrix and n is the "current" node */
    Create a new node NN to represent node n in new AST;
    if (n is not a leaf node)
      let c = number of children of node n;
      Assert P = T(2 : (c + 1), 2 : (c + 1)) is a permutation matrix;
      Construct vector Perm[1..c] where Perm[j] = i if
      child j in old AST becomes child i in new AST;
      ColumnPtr = c+2;
      for j = 1 to c do
        /*Identify column of M where submatrix for child Perm[j] starts*/
        ColumnStart[Perm[j]] = ColumnPtr;
        /*Size(q) = size of instance vector for subtree rooted at q*/
        ColumnPtr = ColumnPtr + Size(Child(Perm[j])) +1;

      RowPtr = c+2;
      for j = 1 to c do
        SubM =
          M[RowPtr:RowPtr+Size(Child(j)),ColumnStart[j]:ColumnStart[j]+Size(Child(j))];
        NewTree = NewAST (SubT, Child(j));
        Make NewTree into Perm[j] child of NN;
        RowPtr = RowPtr + Size(Child(j)) +1;

    endif;
      return NN;
}
```

Figure 6: Algorithm to discover structure of transformed AST

5.3 Dependences

In addition to ensuring that a transformation matrix M has the proper block structure for generating a new AST, we must ensure that dependent dynamic instances are ordered properly in the transformed code. Recall from Section 4 that a dynamic instance DI in the source program is mapped to a dynamic instance $\mathbf{L}^{-1}(M(\mathbf{L}(DI)), AST_f)$ where AST_f is the AST of the transformed program. Suppose that there is a dependence vector d where the source of the dependence is statement $S1$ and the target is statement $S2$. In the case of perfectly nested loops, we would check that $T \cdots d > 0$. For imperfectly nested loops, the only relevant loops are the ones that are common to both $S1$ and $S2$, so we can project $T.d$ onto these common loops, and verify that this projection is a positive vector. With one caveat, this is essentially the test we use. The caveat is that even if the projection of $T.d$ onto the common loops is 0, the dependence may still be satisfied by the syntactic ordering of $S1$ and $S2$ in the new AST. This motivates the following definition.

Definition 6 *Let D be the dependence matrix for some program, and let M be a transformation matrix for this program. M is said to be* **legal** *if it has the block structure described in Section 5.2, and if following condition is true for all dependences d in D.*

If dependence d is from an instance of statement $S1$ to an instance of statement $S2$, let P be the projection of the vector $M.d$ onto that subset of dimensions containing only the loops common to $S1$ and $S2$. Then, either (i) $P > 0$, or (ii) $P = 0$ and $S_1 \preceq_S S_2$ in the new AST.

Note that a transformation matrix M is legal even if there is a dependence d such that $P = 0$ and $S_1 = S_2$ (in this case, we say that d is left **unsatisfied** by M). In other words, two dependent instances of a statement S_1 may be mapped by M to the same dynamic instance of S_1 in the AST produced by the algorithm in Figure 6. As explained in Section 4.1, the mapping induced by a transformation matrix M from dynamic instances in the source program to dynamic instances in the transformed program is not necessarily one-to-one; therefore, we will in general need to add extra loops around atomic statements to enumerate over all dynamic instances. Definition 6 implies that these extra loops must satisfy all dependences not satisfied by the loops in the AST constructed by the algorithm in Figure 6. The following result states this precisely.

Theorem 2 *Let M be a legal transformation matrix for some program, and let AST_s and AST_t be the source and transformed abstract syntax trees. Suppose that in the source program, there is a dependence from instance DI_1 of a statement S_1 to instance DI_2 of a statement S_2. Let TI_1 and TI_2 be the dynamic instances in the transformed program corresponding to DI_1 and DI_2 respectively.*

1. *$TI_1 \leq TI_2$.*
2. *If $TI_1 = TI_2$, then S_1 and S_2 are identical.*

Proof:

1. Follows from Definition 2, and Definition 6 of a legal transformation.
2. Follows from the definition of dynamic instances.

□

5.4 Augmentation with extra loops

Once the AST has been constructed, the next step in code generation is to add extra loops around an atomic statement if multiple instances of it in the source program get mapped to a single instance of that statement in the new AST. We show how to add the additional loops, using the following program as an example.

```
do I = 1..N
  S1: B(I) = B(I-1) + A(I-1,I+1)
  do J = I..N
    S2: A(I,J) = f();
  enddo
enddo
```

Executions of `S1` and `S2` are represented by initial instance vectors $[I, 0, 1, I]'$ and $[I, 1, 0, J]'$ respectively. Dependence analysis on this code produces the matrix $D = \begin{bmatrix} 1 & 1 \\ 0 & -1 \\ 0 & 1 \\ 1 & -1 \end{bmatrix}$. Suppose that the transformation matrix $M = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$. It is easy to verify that both the transformed dependence vectors are legal, and that M is a legal transformation. Note that all instances of `S1` are mapped to iteration 0 of the transformed loop nest.

Therefore, we must add an extra loop around `S1`; since `S1` has self-dependences, this loop must carry these dependences. We do this augmentation in two stages. For each statement, we first determine how the iteration space of that statement is transformed by M; this can be expressed compactly by a matrix which we call the **per-statement transformation** for that statement. In our example, the **per-statement transformations** M_{S1} and M_{S2} are respectively $\begin{bmatrix} 0 \end{bmatrix}$ and $\begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$. In general, these matrices can be computed by adding appropriate columns of M and projecting onto loop positions. For example, for `S1`, we note that the general instance vector is $[I, 0, 1, I]'$. Therefore, we add the first and fourth columns of M and project onto loop I; this gives us the matrix $[0]$ as desired. For lack of space, we omit the details of the general algorithm.

Definition 7 *For any statement S nested in k loops, let $I_S = [i_1, \ldots, i_k]'$ denote the loop values corresponding to an dynamic instance DI of S. Let M be any legal transformation* *matrix. Let AST_i and AST_f be the initial and transformed ASTs respectively. Then, the* **per-statement transformation** *M_S is defined as a $k \times k$ matrix which satisfies the condition:*

$$M_S * I_S = \mathbf{L}^{-1}(M * \mathbf{L}(DI, AST_i), AST_f) \tag{4}$$

The equation in Definition 7 means that the entry for a loop L in the left hand side vector is the same as the label for that loop in the dynamic instance represented by the right hand side. As in our example, the per statement transformation for a statement nested in k loops need not have rank k; therefore, we need to add additional loops. It is obvious that these additional loops around a statement S do not violate any inter-statement dependendences involving S — we only need to ensure that they carry any self-dependences of S left unsatisfied by M. The algorithm for adding these loops is identical to the completion procedure given by Li and Pingali in the context of perfectly nested loops [10], and is shown in Figure 7.

Theorem 3 *Let M be a legal transformation. For a statement S, let $D_s = d_1, d_2, \ldots, d_k$ be the set of self dependences of S unsatisfied by M. Let T_S be the* **per-statement transformation** *corresponding to S. Let $\mathbf{D}_s$ be the set $d'_1, d'_2, \ldots, d'_k$ obtained by projecting each element of D_s onto the entries corresponding to loops surrounding S. Then the following are true:*

1. *$\forall d' \in \mathbf{D}_s, T_s * d' = \vec{0}$*
2. *If rank(T_s) = r, k-r rows can be added at the end of T_s to augment it to T'_s, such that $\forall d' \in \mathbf{D}_s, T'_s * d'$ is lexicographically positive, and T'_s has rank k.*

Proof:

1. Follows trivially from the fact that all these vectors correspond to dependences that have not been satisfied and M is legal.
2. From part 1, it is easy to see that all the rows of T_s are orthogonal to all the vectors in $\mathbf{D}_s$. We use the procedure in Figure 7 to add the remaining k-r rows to T_s.

□

For our example, the augmentation procedure will complete the per-statement transformation for `S1` to produce the rank-1 matrix $\begin{bmatrix} 0 \\ 1 \end{bmatrix}$. For `S2`, the corresponding matrix is $\begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$.

5.5 Generating Loop Bounds

The final step in code generation is to determine for each statement what are the loop bounds and steps for the surrounding loops. We deal with it in this section.

Definition 8 *Let T_S be an $l \times k$ transformation matrix of rank k produced by the procedure in Figure 7. From T_S, construct a new matrix N_S by deleting every row that is either zero or is a linear combinations of previous rows in T_S. This new matrix is called the* **non-singular per-statement transformation** *for statement S.*

```
Procedure Complete (T_s, D_s)
{
    /* Complete T_s into a rank k matrix */
    NumNewRows = 0;
    for (i = 1; i ≤ k-r; i++)
       /*Height returns row number of first non-zero row of a matrix. */
       h = Height(D_s);
       e_h = unit vector of length k, with a 1 at position h;
       Append e_h to T_s;
       Delete all vectors of height h from D_s;
       NumNewRows = NumNewRows +1;
       if D_s has no more entries, then
          break
    endfor
    /* if NumNewRows != (k-r), we need to add (k-r-NumNewRows rows) */
    if (NumNewRows != (k-r)) then
       Append rows spanning the null space of rows of T_s;
    endif
    return
}
```

Figure 7: Adding loops around statements

Theorem 4 *N_S is a $k \times k$ non singular matrix.*

Proof: Obvious from the fact that T_S is of rank k and from the construction of N_S. $\Box$

In our skewing example, matrix N_{S1} is [1], while N_{S2} is $\begin{bmatrix} 1 & -1 \\ 0 & 1 \end{bmatrix}$.

Definition 9 *The loops surrounding S in the transformed AST (after augmentation) corresponding to the rows of T_S that are retained in N_S are called the* **non-singular loops** *of S. Any other loop surrounding S is defined to be a* **singular loop** *surrounding S.*

Theorem 5 *Any dynamic instance of S in the initial program is transformed to a unique set of loop labels for the* **non-singular loops** *surrounding S in the transformed program.*

Proof: Let $[i_1, i_2, \ldots, i_k]'$ correspond to the loop labels of a dynamic instance of S in the initial program. Let $[i_1^{'}, i_2^{'}, \ldots, i_k^{'}]'$ represent the loop labels of a dynamic instance of S in the transformed program corresponding to the **non-singular loops** of S. Then, it is easy to verify that $[i_1^{'}, i_2^{'}, \ldots, i_k^{'}]' = N_S * [i_1, i_2, \ldots, i_k]'$. The theorem follows from the nonsingularity of N_S $\Box$

Lemma 3 *Given N_S, and the initial loop bounds of all loops surrounding S, we can determine the bounds and steps of all* **non-singular loops** *surrounding S after transformation.*

Proof: As we have already mentioned, N_S is an integer non-singular matrix that represents the transformation from the loops surrounding S initially to the **non-singular loops** surrounding S after transformation. We can use an approach identical to the one in [10] used for perfectly nested loops for this purpose. $\Box$

Thus, for any statement, we can generate the loop bounds and steps for all **non-singular** loops surrounding it after transformation. The only remaining issue is regarding the **singular** loops surrounding S. We deal with it as follows. Let k be the row number of T_S corresponding to a **singular loop** of S, represented by r_k. R_k is a linear combination of a certain number (l) of linearly independent rows preceding it in T_S(and all of which appear in N_S). Let $r_1, \ldots, r_l$ denote these rows and let $r_k = m_1 * r_1 + \ldots m_k * r_k$. Given $m_1, \ldots, m_k$, and given the loop bounds corresponding to $r_1, \ldots, r_k$, it is easy to determine the bounds for the loop corresponding to r_k. In addition, a particular iteration of this singular loop is executed only if for that iteration, the value of the loop index variable (i_k) satisfies the condition: $i_k = \sum_{j=1}^{l} m_j * i_j$, where i_j corresponds to the value of the loop index variable corresponding to r_j.

For our running example, the final code generated is the following:

```
do I = 1-N..0
  do J = 1-I..min(N,N-I)
    A(I+J,J) = f()
  enddo
  if (I = 0) then
    do I2 = 1..N
      B(I2) = B(I2-1) + A(I2-1,I2+1)
    enddo
  endif
enddo
```

which, using standard optimizations, can be simplified to

```
do I = 1-N..-1
  do J = 1-I..N
    A(I+J,J) = f()
  enddo
enddo
do J = 1..N
  A(J,J) = f();
enddo
do I2 = 1..N
  B(I2) = B(I2-1) + A(I2-1,I2+1)
enddo
```

6 Completion Procedure

A major advantage of the matrix-based technology for perfectly nested loop transformations is that it provides a way to generate desired loop transformations. For example, in earlier work, Li and Pingali [10] have described a **completion procedure** which, given a dependence matrix and the first few rows of a desired transformation, automatically appends additional rows to the matrix to produce a complete transformation matrix that satisfies all

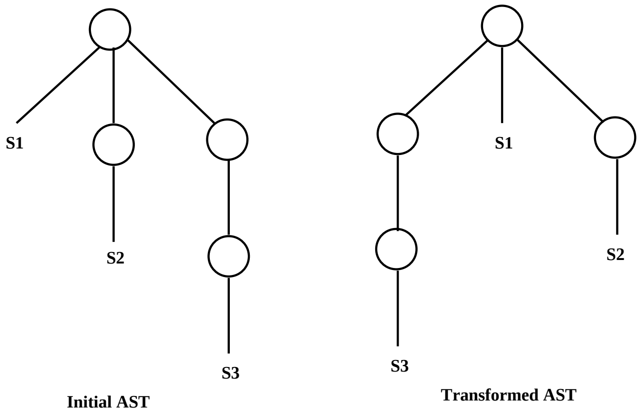

Figure 8: Initial and Final AST for Cholesky

dependences [10]. We have developed a similar procedure for imperfectly nested loops. For lack of space, we do not describe this procedure here, but illustrate its behavior using code for Cholesky factorization (as mentioned earlier, one of the goals of our work was to permit us to reason about loop permutations in matrix factorization codes).

```
do K = 1..N
  S1: A[k][k] = sqrt (A[k][k]);
  do I = K+1..N
    S2: A[i][k] = A[i][k] / A[k][k];
  enddo
  do J = K+1..N
    do L = K+1..J
      S3: A[j][l] = A[j][l] - A[j][k] * A[l][k];
    enddo
  enddo
enddo
```

This code fragment is represented before transformation by the AST on the left in Figure 8. Dependence analysis on this code fragment produces the dependence matrix $\begin{bmatrix} 0 & 0 & + & 1 \\ 0 & 1 & 0 & -1 \\ 1 & -1 & 0 & 0 \\ -1 & 0 & 0 & 1 \\ 0 & + & 0 & 0 \\ 0 & + & 0 & 0 \\ + & - & + & 1 \end{bmatrix}$. A partial transformation with the intention of interchanging the k and j loops produces a first row of transformation equal to $\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}$. Our completion procedure completes this transformation to $\begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$. The AST produced after transformation is the AST on the right in Figure 8. It turns out that the per-statement transformation in this case is non-singular for each statement and no augmentation is necessary. The final code produced by the code generation is shown below: (this corresponds to the traditional **left looking Cholesky** code).

```
do K = 1..N
  do J = K..N
    do L = 1..K-1
      S3: A[j][k] = A[j][k] - A[j][l] * A[k][l];
    enddo
  enddo
  S1: A[k][k] = sqrt (A[k][k]);
  do I = K+1..N
    S2: A[i][k] = A[i][k] / A[k][k];
  enddo
enddo
```

7 Conclusions

We have described an approach to transforming imperfectly nested loops which trades off generality for simplicity. We require that all atomic statements contained in the same set of loops be transformed identically (except for statement alignment). Although this is more restrictive than other approaches, it is sufficiently general to capture permutation of imperfectly nested loops in matrix factorization codes, which constitute a significant proportion of imperfectly nested loops in scientific codes. The advantage of this restriction is that it permits us to extend the technology for transforming perfectly nested loops to imperfectly nested loops in a more or less straight-forward way. In particular, we can use standard dependence abstractions like distances and directions, and loop transformations can be modeled by matrices. The linear framework allows us to look for good transformation efficiently (for example, parallelizing a loop requires finding a row in the nullspace of the dependence matrix) and also permits automatic completion in an efficient manner. In short, by drawing on the same factors that made the matrix based approach practical for transforming perfectly nested loops, we hope to obtain a practical approach to transforming imperfectly nested loops.

We would like to extend this work to incorporate loop distribution and loop fusion into the completion procedure. Currently, these two transformation can be expressed in our framework, but we do not make use of these two transformations in our completion procedure. We are also implementing our loop transformations in the Polaris compiler test-bed [5].

References

[1] C. Ancourt and F. Irigoin. Scanning polyhedra with do loops. In **Principle and Practice of Parallel Programming**, pages 39–50, April 1991.

[2] E. Ayguadé and Jordi Torres. Partitioning the statement per iteration space using non-singular matrices. In **1993 ACM International Conference on Supercomputing**, pages 407–415, Tokyo, jul 1993.

[3] Uptal Banerjee. A theory of loop permutations. In **Languages and compilers for parallel computing**, pages 54–74, 1989.

[4] Uptal Banerjee. Unimodular transformations of double loops. In **Languages and compilers for parallel computing**, pages 192–219, 1990.

[5] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The next generation in parallelizing compilers. Technical Report 1375, Center for Supercomputing Research and Development (CSRD), University of Illinois Urbana-Champaign.

[6] Paul Feautrier. Some efficient solutions to the affine scheduling problem - part i: one dimensional time. **International Journal of Parallel Programming**, October 1992.

[7] Paul Feautrier. Some efficient solutions to the affine scheduling problem - part ii: multi-dimensional time. **International Journal of Parallel Programming**, December 1992.

[8] Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In **The 5th Symposium on the Frontiers of Massively Parallel Computation**, pages 332–341, McLean, Virginia, feb 1995.

[9] S.Y. Kung. **VLSI Array Processors**. Prentice-Hall Inc, 1988.

[10] Wei Li and Keshav Pingali. A singular loop transformation based on non-singular matrices. **International Journal of Parallel Programming**, 22(2), April 1994.

[11] William Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In **Communications of the ACM**, pages 102–114, August 1992.

[12] J. Ramanujam. Optimal code parallelization using unimodular transformations. In **Proceedings of Supercomputing**, 1992.

[13] M. E. Wolf and M. S. Lam. An algorithmic approach to compound loop transformations. In **Languages and compilers for parallel computing**, pages 243–273, 1990.

[14] M. Wolfe. **High Performance Compilers for Parallel Computing**. Addison-Wesley Publishing Company, 1995.