Multi-transformations: Code Generation and Validity *

Peter M.W. Knijnenburg
High Performance Computing Division,
Dept. of Computer Science, Leiden University,
Niels Bohrweg 1, 2333 CA Leiden, the Netherlands
e-mail: peterk@cs.leidenuniv.nl

Eduard Ayguadé and Jordi Torres
Departament d'Arquitectura de Computadors,
Universitat Politècnica de Catalunya,
Gran Capità s/núm, Mòdul D6
08034-Barcelona, Spain
email: {eduard,torres}@ac.upc.es

Abstract

In this paper we present a generalization of the framework of unimodular loop transformations for parallelizing compilers, called multi-transformations. Multi-transformations consist of applying a different unimodular transformation to the iteration space of each statement in the loop body, and include also alignments. Two key aspects are considered in this paper: the generation of efficient code that traverses the different transformed iteration spaces and the test to decide the legality of the multi-transformation. Some examples are used in the paper that show the usefulness of multi-transformations. In parallelizing compilers for shared-memory they allow an easy exploitation of parallelism; for distributed-memory multiprocessors they allow the generation of code that follows the owner-computes rule and exploits locality of references.

Keywords: Parallelizing compilers, Unimodular loop transformations, Efficient code generation, Data dependences, Validity of transformations.

*This research was partially supported by Esprit BRA APPARC under grant no. 6634, and by the Ministry of Education of Spain under contract TIC-880/92.

1 Introduction

Loop transformations have been recognized to be one of the most important components of the parallelizing and vectorizing technology for current supercomputers. The aim is to transform nested loop structures into semantically equivalent versions with more opportunities to parallilize them [Pol88, Wol91].

Most existing compilers apply a set of basic loop transformations, one at a time. In each step it has to be decided whether a transformation is legal (that is, respects data dependences) and beneficial to apply. Recently, it has been proposed to specify transformations by *unimodular* and *non-singular integer matrices* [Ban93, LP94]. Such a matrix represents a linear mapping from the original iteration space to the target iteration space. A wide range of basic loop transformations can be represented in this way, including loop interchange, loop reversal, loop skewing [Ban91] and loop scaling [LP94]. In fact, any linear transformation modelled with a non-singular matrix can be viewed as a composotion or product of these four basic transformations.

The transformation matrix is used to translate (array) references in the original loop to references in the target loop as well as to determine the loop bounds of the new loop by means of Fourier-Motzkin elimination [Ban91, Ban93,

LP94, WL91]. Traditionally, such a transformation applies to the whole loop. Recently, it has been argued that it can be profitable to apply different transformations to different statements in a loop [AT93, TALV93, DRR93, KPR94]. In [AT93] it has been shown how different displacements $\mathtt{d}_i$ for each statement S_i can be used to break dependences. This allows to consider loop alignment in a unified way with other loop transformations. In [TALV93] this technique has been applied to eliminate non-local references when compiling for non-uniform memory architectures. However, although they allow diffent displacements for different statements, the transformation (not necessarely unimodular) they apply afterwards is the same for all of them. In [DRR93] a parallel scheduling technique for loop nests is proposed which extends the hyperplane method [Lam74] by using a different affine scheduling for each statement in the nest.

In this paper we extend the framework of linear loop transformations by applying a different unimodular transformation $\mathcal{U}_i$ and displacement $\mathtt{d}_i$ to each statement S_i in the loop body. We call such a transformation a *multi-transformation.*

We outline, by means of some examples, two possible applications for the technique discussed in this paper. First, it can be used to enhance the parallelism of a loop by reordering statements and thereby breaking dependences. Second, it can be used to improve the data locality of a loop and allow an HPF-like compiler for distributed-memory machines to generate SPMD code following the owner-computes rule.

We also describe the automatic generation of efficient code from the specification of the multi-transformation and the original loop bounds and subscript expressions. The code generated scans a transformed iteration space that is the composition of the transformed iteration space for each statement in the loop body. It is very important to optimize this code in order not to hide the benefits obtained by parallelization. Another key aspect considered in the paper is the validity of a multi-transformation. We show how to test whether all dependences in the transformed iteration space remain lexicographically positive.

The paper is organized as follows. Section 3 shows the use of multi-transformations when compiling loops for a distributed memory machine with data mappings specified in a HPF-like language. Before that, section 2 give some preliminaries and introduce some notation. In section 4 we give the definition of multi-transformations and discuss the basic algorithms used to generate efficient code for the resulting loop structure. In section 6 we show how the validity of a multi-transformation can be established. For these two sections we discuss a working example (section 5) oriented to exploit parallelism by breaking dependences and recurrences using multi-transformations. Finally, in section 7, we give a brief discussion and relate other work in the area.

2 Preliminaries

In this section we give some notation and terminology used in this paper. First, we define lower and upperbounds for the loops we will consider.

Definition 2.1 *Let $\mathcal{I} = \{\mathtt{I}_1, \ldots, \mathtt{I}_n\}$ be a finite collection of loop indices. With respect to this collection $\mathcal{I}$ we define:*

1. *A* basic (lower or upper) bound *is an affine expression $a_0 + a_1\mathtt{I}_1 + \cdots + a_n\mathtt{I}_n$ where, for all i, $a_i \in \mathbf{Z}$. Note that some of the a_i may be zero.*
2. *Let $a \in \mathbf{N}^+$ and let B be a basic bound. A* simple lowerbound *is an expression $\lceil \frac{1}{a} B \rceil$. A* simple upperbound *is an expression $\lfloor \frac{1}{a} B \rfloor$.*
3. *A* compound lowerbound *is an expression $\max(L_1, \ldots, L_m)$ where each L_i is a simple lowerbound. A* compound upperbound *is an expression $\min(U_1, \ldots, U_m)$ where each U_i is a simple upperbound.*
4. *Basic, simple and compound bounds are also called* admissible bounds.

In the sequel of the paper we assume that all bounds of loops to be transformed are admissible. The techniques presented are such that all resulting loop structures have admissible bounds. Every perfectly nested loop $\mathcal{L}$ gives rise to a system of inequalities $\mathcal{S}(\mathcal{L})$, given by

$$\mathcal{S}(\mathcal{L}) = \left\{ \begin{array}{ccccc} \mathtt{L}_1 & \leq & \mathtt{I}_1 & \leq & \mathtt{U}_1 \\ & & \vdots & & \\ \mathtt{L}_n & \leq & \mathtt{I}_n & \leq & \mathtt{U}_n \end{array} \right.$$

Such a system should be read as the *conjunction* of the individual clauses. This system has the property that every bound $\mathtt{L}_i$ and $\mathtt{U}_i$ is admissible and only involves variables $\mathtt{I}_1, \ldots, \mathtt{I}_{i-1}$. We call this the *standard form* of a system of inequalities. Consider a system of inequalities involving the variables $\mathtt{I}_1, \ldots, \mathtt{I}_n$

$$\{e_1 \leq e'_1, \ldots, e_k \leq e'_k\}$$

where each e_i is an admissible lowerbound and each e'_i is an admissible upperbound. Any such system can be brought in standard form using *Fourier-Motzkin elimination* [DE73, Ban93]. Hence such system can be used to define the bounds of a perfectly nested loop. This is one of the basic steps in applying a unimodular transformation [Ban93, LP94]. Note that it is crucial that all expressions are admissible for Fourier-Motzkin elimination to be applicable.

Next we define a class of loop structures which will be delivered by the transformations we propose in this paper. This class has been introduced by Chamski [Cha94].

Definition 2.2 *The class of* nested loop sequences *is inductively defined as the smallest class of loop structures closed under*

1. *Any block of statements is a (trivial) nested loop sequence.*
2. *If $\mathcal{L}$ and $\mathcal{L}'$ are two nested loop sequences, then so is $\mathcal{L}; \mathcal{L}'$ ($\mathcal{L}$ followed by $\mathcal{L}'$).*
3. *If $\mathcal{L}$ is a nested loop sequences, then so is the following loop,*

```
DO  I = L, U
      L
ENDDO
```

   where I *is a new loop variable.*

A nested loop sequence is called *well-formed* iff all bounds are affine expressions over the loop counters of the surrounding loops. In the sequel we will only consider well-formed nested loop sequences which we will call simply a nested loop sequence. For an example of a nested loop sequence, see Figure 9 in section 5.

Finally, we need the following notion. Consider a block of statements in a nested loop sequence. Then we can construct a perfectly nested loop with bounds the bounds of the enclosing loops and a body that only includes the statements of the block. We call this loop nest the *local nest* of the block.

3 Motivation

In this section we show the motivation for Multi-transformations. In particular, we show how they can be useful in a HPF-like parallelizing compiler for distributed memory machines. This compiler is assumed to perform the parallelization of loops based on the data layout (either specified by the user or automatically generated by a tool) and on the owner-computes rule.

According to the owner-computes rule, a loop should be parallelized so that each processor executes the portion of the iteration space that performs computations on the data elements owned by that processor. In other words, the array subscripts on the left-hand side of the assignment statements decide the parallelization of the iteration space. For instance, consider the loop in figure 1, taken from routine *calc*2 from *swm*256, a program in the SPEC benchmark set [Dix91]. Here we have taken N = 16 and M = 8 for simplicity. Assume that, due to the data layout preferences of other loops or procedures, the reaching data layout for the arrays is the following (expressed in HPF notation):

```
CHPF$ ALIGN WITH UNEW :: UOLD, Z, CV, H
CHPF$ ALIGN WITH PNEW :: POLD, CU
CHPF$ DISTRIBUTE UNEW(CYCLIC, *)
CHPF$ DISTRIBUTE PNEW(*, CYCLIC)
```

As dictated by the owner-computes rule, the first statement of the loop would lead to the parallelization of the inner I loop in such a way that each processor would execute those iterations that update the elements UNEW(I+1,J) owned by it. For instance, the iteration I = 1 should be executed by the processor that owns row 2 of matrix UNEW. Figure 2.a shows the assignment of iterations to processors that fulfils this property. Using the same reasoning, the second statement of the loop body would lead to the parallelization of the outer J loop so that each processor would execute those iterations that update the elements PNEW(I,J) owned by it, as shown in Figure 2.b. For instance, iteration J = 1 should be executed by the processor that owns column

Figure 3: Iteration space after applying a different linear transformation to each statement.

```
 CHPF$ INDEPENDENT
 CAPR$ DO PAR ON PNEW(:, 1)
       DO I' = 1,1
          DO J' = 1,16
             PNEW(J',I') = POLD(J',I')-TDTSDX*(CU(J'+1,I')-CU(J',I'))
1                         -TDTSDY*(CV(J',I'+1)-CV(J',I'))

          ENDDO
       ENDDO
 CHPF$ INDEPENDENT
 CAPR$ DO PAR ON UNEW(2~1, :)
       DO I' = 2,8
          DO J' = 1,8
             UNEW(I',J') = UOLD(I',J')+
1                TDTS8*(Z(I',J'+1)+Z(I',J'))*(CV(I',J'+1)+CV(I'-1,J'+1)+CV(I'-1,J')
2                   +CV(I',J'))-TDTSDX*(H(I',J')-H(I'-1,J'))

             PNEW(J',I') = POLD(J',I')-TDTSDX*(CU(J'+1,I')-CU(J',I'))
1               -TDTSDY*(CV(J',I'+1)-CV(J',I'))

          ENDDO
          DO J' = 9,16
             PNEW(J',I') = POLD(J',I')-TDTSDX*(CU(J'+1,I')-CU(J',I'))
1               -TDTSDY*(CV(J',I'+1)-CV(J',I'))

          ENDDO
       ENDDO
 CHPF$ INDEPENDENT
 CAPR$ DO PAR ON UNEW(9~1, :)
       DO I' = 9,17
          DO J' = 1,8
             UNEW(I',J') = UOLD(I',J')+
1                TDTS8*(Z(I',J'+1)+Z(I',J'))*(CV(I',J'+1)+CV(I'-1,J'+1)+CV(I'-1,J')
2                   +CV(I',J'))-TDTSDX*(H(I',J')-H(I'-1,J'))

          ENDDO
       ENDDO
```

Figure 4: Resulting code that scans the transformed iteration space in Figure 3

1 of matrix PNEW. Notice that the two parallelization strategies are not compatible, making it difficult to generate parallel code. In fact, the compiler should generate conditional statements to test at run-time data ownership and dynamically decide the iterations of the I and J loops for which each processor would execute each statement in the loop body.

Now we show how multi-transformations are useful to generate the parallel code that satifies the owner-computes rule. Assume that the iteration space of the first statement is transformed by applying a unit shift to dimension I and the iteration space of the second statement is transposed with respect to the one of the first statement. Figure 3 shows the resulting iteration space after combining the two transformations. Notice that now we could parallelize the $\mathtt{I}'$ dimension and assign iterations in that dimension so that the owner-computes rule is satisfied, as shown in the bottom right corner of Figure 3. The parallel code can be automatically generated from the formal specification of these two transformations (as described in Section 4). This code will scan the transformed iteration space so that each processor only executes those iterations of the parallel $\mathtt{I}'$ loop that are needed, and for each iteration of the sequential $\mathtt{J}'$ loop the statements that have to be executed.

The code after applying the automatic transformation procedure is shown in figure 4. In this code we are using the CHPF$ INDEPENDENT directive to indicate the loops that should be run in parallel. In addition to this directive we also use the CAPR$ DO PAR directive that has a similar meaning but is more complete. This directive is defined by the FORGE programming model [Inc] and is used here because of the ON clause which selects a distributed array to control the loop distribution. Loop iterations are distributed in such a way that references to the control array occur in the processor that owns those elements.

4 Multi-transformations

In this section we discuss a notion of multiple mappings for perfectly nested loops. We call these mappings *multi-transformations*. Furthermore, we show how efficient code for the loop structure resulting from a multi-transformation can be generated. In section 6 we show how the validity of a multi-transformation can be decided.

First, consider a perfectly nested loop $\mathcal{L}$ with statements $S_1, \ldots, S_N$ in its body.

```
DO  I1 = L1, U1
    ...
    DO  In = Ln, Un
        S1
        ...
        SN
    ENDDO
    ...
ENDDO
```

where all bounds are admissible bounds over the loop indices of surrounding loops. The local nest of statement S_i is:

```
DO  I1 = L1, U1
    ...
    DO  In = Ln, Un
        Si
    ENDDO
    ...
ENDDO
```

The local loop nest of a statement is used for determining the iteration space for this statement after a multi-transformation.

Definition 4.1 *Consider a perfectly nested loop $\mathcal{L}$ of nesting depth n and with N statements in its body. A* multi-transformation $\mathcal{M}$ *for $\mathcal{L}$ consists of*

- *a collection $\mathcal{T}$ of $n \times n$ unimodular transformations $\mathcal{T} = \{\mathcal{U}_i : 1 \leq i \leq N\}$, and*
- *a collection $\mathcal{D}$ of n dimensional displacement vectors $\mathcal{D} = \{\mathtt{d}_i : 1 \leq i \leq N\}$.*

Let $\mathcal{L}$ be a perfectly nested loop, and let $\mathcal{M} = (\mathcal{T}, \mathcal{D})$ be a multi-transformation for $\mathcal{L}$. The application of $\mathcal{M}$ to $\mathcal{L}$ consists of transforming the local nest of statement S_i by the unimodular transformation $\mathcal{U}_i$ and adding the displacement vector $\mathtt{d}_i$ thereby obtaining a *local transformed loop nest*. By this transformation, iteration $\vec{\mathtt{I}}$ is mapped to iteration $\mathcal{U}_i\vec{\mathtt{I}} + \mathtt{d}_i$ for statement S_i.

In this way, we obtain different transformed iteration spaces for the different statements in $\mathcal{L}$. We have to compute the union of these iteration spaces and scan them in the lexicographical order. Note that the union of convex polytopes is not necessarily convex itself. Hence the required loop structure cannot be one single perfectly nested loop, but will rather be a nested loop sequence.

We now show how to generate code for the loop structure resulting from the application of a multi-transformation. First, we construct all local transformed loop nests. Second, we construct one large loop, with an iteration space that properly contains the union of the individual transformed iteration spaces. The body of the loop consists of the transformed statements, guarded in such a way that they are only executed in the iterations belonging to the local transformed iteration space of this statement. We regard this loop as an *intermediate representation* of the desired nested loop sequence. Third, we remove these guards by splitting up the iteration space.

Transformation $\mathcal{U}_i$ and displacement $\mathtt{d}_i$ are intended to be applied to statement S_i. We denote the resulting statement by $(\mathcal{U}_i + \mathtt{d}_i)\,(S_i)$. In [AT93, TALV93] it has been explained how to apply such a transformation. We give a short review of the theory. Iteration $\vec{\mathtt{I}}$ is mapped to $\vec{\mathtt{I}}' = \mathcal{U}_i\vec{\mathtt{I}} + \mathtt{d}_i$. Hence references to $\vec{\mathtt{I}}$ in S_i have to be replaced by $\mathcal{U}_i^{-1}(\vec{\mathtt{I}}' - \mathtt{d}_i) = \mathcal{U}_i^{-1}\vec{\mathtt{I}}' - \mathcal{U}_i^{-1}\mathtt{d}_i$. Note that the difference with an ordinary unimodular transformation [Ban93] lies in the extra displacement component. Loop bounds are transformed likewise and Fourier-Motzkin elimination can be applied to obtain a standard form [Ban93]. The resulting local loop nest is denoted as:

```
DO  I'_1 = L_1i, U_1i
    ...
    DO  I'_n = L_ni, U_ni
        (U_i + d_i) (S_i)
    ENDDO
    ...
ENDDO
```

Next, these local loop nests have to be composed into one resulting, global nest. Intuitively, we can generate the code depicted in figure 5. Although this code is semantically correct, it is very inefficient. Nevertheless, it gives a clear picture of the kind of transformation and its result we are considering in this paper. We now show how to transform the above code into an efficient form. In [KB95] we have developed the necessary theory for doing this automatically. We briefly show how to apply this theory in the present case.

The main technical observation is that the IF-statements in the intermediate code for the transformed loop are of a special form. They precisely define a convex polytope, namely, the iteration space of the local transformed loop nest. We call these IF-statements *guards*.

Suppose $\mathcal{L}$ is a loop nest with loop indices $\mathtt{I}_1, \ldots, \mathtt{I}_n$ and admissible loop bounds, giving rise to the following system of inequalities $\mathcal{S}(\mathcal{L})$:

$$\mathtt{L}_1 \leq \mathtt{I}_1 \leq U_1, \ldots, \mathtt{L}_n \leq \mathtt{I}_n \leq \mathtt{U}_n$$

Note that a lowerbound $\mathtt{L}_k$ may involve the indices $\mathtt{I}_1, \ldots, \mathtt{I}_{k-1}$, and likewise for an upperbound $\mathtt{U}_k$. These bounds define the iteration space $\mathcal{P}$ for $\mathcal{L}$. Suppose that the condition of a guard in $\mathcal{L}$ is of the form

$$\mathtt{L}'_1 \leq \mathtt{I}_1 \leq U'_1, \ldots, \mathtt{L}'_n \leq \mathtt{I}_n \leq \mathtt{U}'_n$$

defining a polytope $\mathcal{P}'$ inside $\mathcal{P}$.

We now want to generate a loop structure that scans $\mathcal{P}$ in the same order as $\mathcal{L}$, but that executes different code in the intersection of $\mathcal{P}$ and $\mathcal{P}'$, and in the remainder of $\mathcal{P}$. In [KB95] we show how to isolate a polytope inside another polytope and how to generate efficient code for scanning this partitioned space. The result of this techniques is a nested loop sequence. This nested loop sequence partitiones the iteration space $\mathcal{P}$ in convex regions with the iteration space $\mathcal{P}'$ isolated. We can execute different code in the different regions: In $\mathcal{P}'$ we execute the statement of the guard together with all other guards, and in the remainder we execute only these other guards. Effectively, we have evaluated the condition of the IF-satement *at compile time* and have partitioned the loop accordingly. In this way we have removed one guard. We can itererate the procedure to remove all guards. Space considerations prevent us from going into details. Consult [KB95] for a proof of the next proposition.

Proposition 4.2 *Given a perfectly nested loop $\mathcal{L}$ containing guards, we can construct a semantically equivalent nested loop sequence $\mathcal{N}$ without*

```
DO  I'_1 = min{L_1i : 1 ≤ i ≤ N}, max{U_1i : 1 ≤ i ≤ N}
    ...
    DO  I'_n = min{L_ni : 1 ≤ i ≤ N}, max{U_ni : 1 ≤ i ≤ N}
        IF (L_11 ≤ I'_1 ≤ U_11)& ··· &(L_n1 ≤ I'_n ≤ U_n1) THEN (U_1 + d_1) (S_1)
        ⋮
        IF (L_1N ≤ I'_1 ≤ U_1N)& ··· &(L_nN ≤ I'_n ≤ U_nN) THEN (U_N + d_N) (S_N)
    ENDDO
    ...
ENDDO
```

Figure 5: Intermediate code for the loop structure resulting from a multi-transformation.

guards in which moreover all bounds are simple bounds.

We now want to apply Proposition 4.2 to the present case of code generation for multi-transformations. However, Proposition 4.2 cannot be applied immediately, since the loop bounds are not admissible (see Definition 2.1): lowerbounds contain minimum functions and upperbounds maximum functions. Therefore we replace the loop bounds by integer constants which are garanteed to be smaller than the lowerbounds, and larger than the upperbounds. In this way we obtain a rectangular iteration space, properly containing all local iteration spaces. The resulting bounds are thus admissible. We can compute these constants using Banerjee's theory on the maximum and minimum values that affine functions can take [Ban88]. After this phase we can legally apply the theory from [KB95]. We obtain a nested loop sequence that exactly scans the iteration space which consists of the union of the local iteration spaces obtained in the multi-transformation but avoid the use guards.

5 Some Examples

In this section we show with a couple of examples the usefulness of multi-transformations. In the first example we show how to transform statements involved in a cycle in the dependence graph in order to exploit inherent parallelism and to reduce the number of explicit synchronizations needed to execute the loop in parallel. In the second example, we show how multi-transformations are useful to transform non-uniform dependences into uniform ones.

In the first example we consider a loop where there are several recurrences in the dependence graph. Consider the following loop:

```
    DO I = 1, 16
      DO J = 1, 8
S1:     A(I, J) = B(I-1, J+1) ...
S2:     B(I, J) = A(I, J-1) ...
S3:     C(I, J) = C(I-1, J-1) + B(I, J)
     ENDDO
    ENDDO
```

The dependence graph and iteration space are shown in Figures 6.a and 6.b, respectively. Notice that there are two recurrences (one traversing the first and the second statement and another one including the third statement.

These two recurrences prevent the exploitation of parallelism out of the loop. Assume that we apply the following multi-transformation:

$$\mathcal{U}_1 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \mathtt{d}_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$\mathcal{U}_2 = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \qquad \mathtt{d}_2 = \begin{pmatrix} -1 \\ 0 \end{pmatrix}$$

$$\mathcal{U}_3 = \begin{pmatrix} -1 & 1 \\ 1 & 0 \end{pmatrix} \qquad \mathtt{d}_3 = \begin{pmatrix} 15 \\ 0 \end{pmatrix}$$

Observe that $(\mathcal{U}_1 + \mathtt{d}_1)(S_1)$ represents a loop interchange, $(\mathcal{U}_2 + \mathtt{d}_2)(S_2)$ a loop interchange with a unit shift in the first dimension, and $(\mathcal{U}_3 + \mathtt{d}_3)(S_3)$ a skew of the space plus a shift. After applying this multi-transformation, the iteration space for the transformed loop becomes the one shown in Figure 7. One can see that in this transformed space the dependences, forming the two recurrences in the original graph, now allow the parallelization of the

Figure 7: Transformed iteration space after applying a multi-transformation.

```
    DO I' = 0,22
       DO J' = 1,16
S1'       IF (1 <= I' <= 8 & 1 <= J' <= 16) A(J',I') = B(J'-1,I'+1) ...
S2'       IF (0 <= I' <= 7 & 1 <= J' <= 16) B(J',I'+1) = A(J',I') ...
S3'       IF (0 <= I' <= 22 & max(1,16-I') <= J' <= min(16,23-I')) C(J',I'+J'-15) =
                                                    C(J'-1,I'+J'-16) + B(J',I'+J'-15)
       ENDDO
    ENDDO
```

Figure 8: Intermediate code that scans the transformed iterationspace shown in Figure 6

```
DO I' = 0,0
   DO J' = 1,15
      S2'
   ENDDO
   DO J' = 16,16
      S2'  S3'
   ENDDO
ENDDO
DO I' = 1,7
   DO J' = 1,15-I'
      S1'  S2'
   ENDDO
   DO J' = 16-I',16
      S1'  S2'  S3'
   ENDDO
ENDDO
```

```
DO I' = 8,8
   DO J' = 1,7
      S1'
   ENDDO
   DO J' = 8,15
      S1'  S3'
   ENDDO
   DO J' = 16,16
      S1'
   ENDDO
ENDDO
DO I' = 9,15
   DO J' = 16-I',23-I'
      S3'
   ENDDO
ENDDO
DO I' = 16,22
   DO J' = 1,23-I'
      S3'
   ENDDO
ENDDO
```

Figure 9: Code after removing guards.

outer $\mathtt{I}'$ loop. Only dependence $S_2\ \delta\ S_3$ with distance $\langle 0,0\rangle$ in the original graph has become a non-uniform data dependence and needs explicit synchronization to allow the parallelization of the $\mathtt{I}'$ loop. The offset $\mathtt{d}_3$ added to the transformed iteration space of S_3 makes the dependence $S_2\ \delta\ S_3$ lexicographically positive in the transformed iteration space. The value of $\mathtt{d}_3$ is the minimum value that guarantees the 'lexicographically positiveness' of the transformed distance. This minimum value for the offset can be obtained from the transformed distance by assuming a generic value for $\mathtt{d}_3$ and then instantiate it to ensure this condition.

The intermediate code, obtained form the transformed local iteration spaces, is shown is Figure 8. After removing the guards, we obtain the code shown in figure 9.

In the second example we show how a non-uniform data dependence that prevents parallel loop execution can be transformed into a uniform one by using a multi-transformation, thus allowing the loop to run in parallel. Consider the following loop:

```
      DO I = 1, 16
        DO J = 1, 8
S1:        A(I, J) = ...
S2:        B(I, J) = A(I-J, J) ...
S3:        C(I, J) = B(I-1, J-1) ...
        ENDDO
      ENDDO
```

The dependence graph for this loop is shown in Figure 10.a. If we apply the following multi-transformation

$$\mathcal{U}_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad \mathtt{d}_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$\mathcal{U}_2 = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \qquad \mathtt{d}_2 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$\mathcal{U}_3 = \begin{pmatrix} 1 & -1 \\ 0 & 1 \end{pmatrix} \qquad \mathtt{d}_3 = \begin{pmatrix} -1 \\ -1 \end{pmatrix}$$

then the original iteration space is transformed into the one shown in Figure 10.b. Notice that the iteration space for statement S_1 remains unchanged and the iteration spaces for S_2 and S_3 are skewed with respect to S_1. In addition, S_3 is shifted with respect to S_2. As a result of this transformation, every dependence in the orginal dependence graph is transformed to $\langle 0,0\rangle$ and the outer $\mathtt{I}'$ loop can be fully parallelized.

6 Validity of a multi-transformation

In this section we discuss how the validity of a multi-transformation can be decided. The

Figure 10: (a) Dependence graph and (b) iteration space after multi-transformation

theory we present in this section is based on [Kni94].

Suppose $\mathcal{L}$ is a loop nest with loop indices $\mathtt{I}_1, \ldots, \mathtt{I}_n$ and admissible loop bounds, giving rise to the following system of inequalities $\mathcal{S}(\mathcal{L})$:

$$\mathtt{L}_1 \leq \mathtt{I}_1 \leq U_1, \ldots, \mathtt{L}_n \leq \mathtt{I}_n \leq \mathtt{U}_n$$

Note that a lowerbound $\mathtt{L}_k$ may involve the indices $\mathtt{I}_1, \ldots, \mathtt{I}_{k-1}$, and likewise for an upperbound $\mathtt{U}_k$. These bounds define the iteration space $\mathcal{P}$ for $\mathcal{L}$.

Let S_1 and S_2 be statements in $\mathcal{L}$ and suppose that they are involved in a dependence δ. We assume that S_1 defines a subscripted variable

$$S_1: \quad \mathtt{A}(e_1, \ldots, e_d) \quad = \quad \cdots$$

and that S_2 uses a subscripted variable

$$S_2: \quad \cdots \quad = \quad \cdots \mathtt{A}(e'_1, \ldots, e'_d) \cdots$$

Our aim is to define a collection of polytopes which contain all dependence information. We only consider *memory based dependences* [PW92]. These polytopes are called *dependence polytopes*.

A possible dependence can be carried by each loop in the nest, or is a loop independent dependence. Therefore, for each $1 \leq m \leq n$, we define a polytope $\mathcal{P}_m$ which encodes the dependences carried by the mth loop, and a polytope $\mathcal{P}_\infty$ for the loop independent dependences. Each dependence polytope $\mathcal{P}_m$ is a subset of $\mathcal{P} \times \mathcal{P}$. Each point $\langle \vec{\mathtt{I}}, \vec{\mathtt{I}}' \rangle$ in $\mathcal{P}_m$ will code a dependence from iteration $\vec{\mathtt{I}}$ to iteration $\vec{\mathtt{I}}'$ with a direction vector

$$\langle \underbrace{=, \cdots, =}_{m-1\times}, <, *, \cdots, * \rangle \tag{1}$$

The polytope $\mathcal{P}_\infty$ has analogous properties.

We construct $\mathcal{P}_m$ in three stages. First, we construct $\mathcal{P} \times \mathcal{P}$ by introducing fresh variables $\mathtt{I}'_1, \ldots, \mathtt{I}'_n$.

$$\begin{cases} \mathtt{L}_1 \leq \mathtt{I}_1 \leq \mathtt{U}_1, \ldots, \mathtt{L}_n \leq \mathtt{I}_n \leq \mathtt{U}_n \\ \mathtt{L}_1[\vec{\mathtt{I}}'/\vec{\mathtt{I}}] \leq \mathtt{I}'_1 \leq \mathtt{U}_1[\vec{\mathtt{I}}'/\vec{\mathtt{I}}], \ldots, \\ \qquad \mathtt{L}_n[\vec{\mathtt{I}}'/\vec{\mathtt{I}}] \leq \mathtt{I}'_n \leq \mathtt{U}_n[\vec{\mathtt{I}}'/\vec{\mathtt{I}}] \end{cases} \tag{2}$$

Second, we add the constraints that enforce the two references to be to the same location.

$$e_1 = e'_1[\vec{\mathtt{I}}'/\vec{\mathtt{I}}], \ldots, e_d = e'_d[\vec{\mathtt{I}}'/\vec{\mathtt{I}}] \tag{3}$$

Third, we add a constraint to filter out the points $\langle \vec{\mathtt{I}}, \vec{\mathtt{I}}' \rangle$ such that $\vec{\mathtt{I}}'$ is lexicographically larger than $\vec{\mathtt{I}}$. In particular, we filter out the points such that the direction vector of the dependence from $\vec{\mathtt{I}}$ to $\vec{\mathtt{I}}'$ is of the form 1. The constraint is given by

$$\mathtt{I}_1 = \mathtt{I}'_1, \ldots, \mathtt{I}_{m-1} = \mathtt{I}'_{m-1}, \mathtt{I}_m < \mathtt{I}'_m \qquad (4)$$

Summarizing, $\mathcal{P}_m$ is defined by the system of inequalities given by equations (2), (3) and (4). Finally, we construct a polytope $\mathcal{P}_\infty$ containing the loop independent dependences from the system of inequalities given by equations (2), (3) and

$$\mathtt{I}_1 = \mathtt{I}'_1, \ldots, \mathtt{I}_n = \mathtt{I}'_n \qquad (5)$$

Example. Consider the first example in section 5 and the dependence $S_2\ \delta\ S_3$. The dependence polytopes are given by: $\mathcal{P}_1 = \emptyset$, $\mathcal{P}_2 = \emptyset$ and $\mathcal{P}_\infty$ is given by

$$1 \leq \mathtt{I}_1 \leq 16 \quad 1 \leq \mathtt{I}_2 \leq 8$$

$$\mathtt{I}_1 \leq \mathtt{I}'_1 \leq \mathtt{I}_1 \quad \mathtt{I}_2 \leq \mathtt{I}'_2 \leq \mathtt{I}_2$$

We see from this description that every dependence runs from $\langle i, j \rangle$ to $\langle i, j \rangle$, which means that the distance is $\langle 0, 0 \rangle$. □

Now assume that S_1 is to be transformed by $\mathcal{U}_1$ and $\mathtt{d}_1$, and S_2 by $\mathcal{U}_2$ and $\mathtt{d}_2$. Consider the matrix $\mathcal{U}_1 \otimes \mathcal{U}_2$ given by

$$\mathcal{U}_1 \otimes \mathcal{U}_2 = \begin{pmatrix} \mathcal{U}_1 & 0 \\ 0 & \mathcal{U}_2 \end{pmatrix}$$

If $\mathcal{U}_1$ and $\mathcal{U}_2$ are unimodular, then so is $\mathcal{U}_1 \otimes \mathcal{U}_2$. Let the vector $\mathtt{d}$ be given by

$$\mathtt{d} = \begin{pmatrix} \mathtt{d}_1 \\ \mathtt{d}_2 \end{pmatrix}$$

Now we want to transform each $\mathcal{P}_m$ by the unimodular transformation $\mathcal{U}_1 \otimes \mathcal{U}_2$ and displacement $\mathtt{d}$. This can be done in exactly the same way as transforming a loop by a unimodular transformation and a displacement. This is briefly discussed in section 4, and in more detail in [AT93, TALV93, Kni94]. The resulting polytope is denoted as $\mathcal{P}^*_m$:

$$\mathcal{P}^*_m = (\mathcal{U}_1 \otimes \mathcal{U}_2 + \mathtt{d})\,(\mathcal{P}_m)$$

A point in $\mathcal{P}^*_m$ is given by $\langle \mathcal{U}_1 \vec{\mathtt{I}} + \mathtt{d}_1, \mathcal{U}_2 \vec{\mathtt{I}}' + \mathtt{d}_2 \rangle$, which is exactly the dependence in the transformed iteration spaces.

Example. (continued) We have to transform $\mathcal{P}_\infty$ from the previous example. The transformed polytope

$$\mathcal{P}^*_\infty = \left(\mathcal{U}_2 \otimes \mathcal{U}_3 + \begin{pmatrix} \mathtt{d}_2 \\ \mathtt{d}_3 \end{pmatrix} \right) (\mathcal{P}_\infty)$$

where $\mathcal{U}_2$, $\mathcal{U}_3$, $\mathtt{d}_2$ and $\mathtt{d}_3$ are given in section 5. $\mathcal{P}^*_\infty$ is given by the following system of inequalities.

$$0 \leq \mathtt{J}_1 \leq 7 \quad 1 \leq \mathtt{J}_2 \leq 16$$

$$16 + \mathtt{J}_1 - \mathtt{J}_2 \leq \mathtt{J}'_1 \leq 16 + \mathtt{J}_1 - \mathtt{J}_2 \quad \mathtt{J}_2 \leq \mathtt{J}'_2 \leq \mathtt{J}_2$$

We see that the transformed dependence is indeed non-uniform. For instance, there is a dependence from $\langle 1, 8 \rangle$ to $\langle 9, 8 \rangle$ with distance $\langle 8, 0 \rangle$; and a dependence from $\langle 2, 9 \rangle$ to $\langle 9, 9 \rangle$ with distance $\langle 7, 0 \rangle$. □

Now we show how to test whether all dependences remain lexicographically positive. This is the condition that the dependence δ is preserved by the transformation [ZC90, Wol91]. We consruct new polytopes $\mathcal{Q}^k_m$ for $1 \leq k \leq n$ by adding 'lexicographical possitiveness' constraints to $\mathcal{P}^*_m$:

$$\mathtt{J}'_1 = \mathtt{J}_1 \ \ \& \ \cdots \ \& \ \ \mathtt{J}'_{k-1} = \mathtt{J}_{k-1} \ \ \& \ \ \mathtt{J}'_k < \mathtt{J}_k$$

Observe that $\mathcal{Q}^k_m$ contains an integer point $\langle \vec{\mathtt{J}}, \vec{\mathtt{J}}' \rangle$ iff there exists a dependence from $\vec{\mathtt{I}}$ to $\vec{\mathtt{I}}'$ such that for the transformed dependence, running from $\vec{\mathtt{J}} = (\mathcal{U}_1 + \mathtt{d}_1)\vec{\mathtt{I}}$ to $\vec{\mathtt{J}}' = (\mathcal{U}_2 + \mathtt{d}_2)\vec{\mathtt{I}}'$, it is the case that the direction vector is the form

$$\langle \underbrace{=, \cdots, =}_{k-1\times}, >, *, \cdots, * \rangle$$

Hence in case any of the polytopes $\mathcal{Q}^k_m$ contain integer points, the dependence is violated.

Example. (continued) Given the description of $\mathcal{P}^*_\infty$ above, it is easy to see that both the conditions $\mathtt{J}'_1 < \mathtt{J}_1$ and $\mathtt{J}_1 = \mathtt{J}'_1$ & $\mathtt{J}'_2 < \mathtt{J}_2$ yield inconsistent systems of inequalities. Hence the dependence $\langle 0, 0 \rangle$ is not violated by the transformation. □

Proposition 6.1 *Let $\mathcal{L}$ be a loop nest containing N statements $S_1, \ldots, S_N$. Let $\mathcal{M} = (\mathcal{T}, \mathcal{D})$ be a multi-transformation for $\mathcal{L}$. Then the application of $\mathcal{M}$ on $\mathcal{L}$ is valid if and only if for each dependence δ from S_i to S_j it is the case that the polytopes $\mathcal{Q}^k_m$ obtained as described above from $\mathcal{U}_i \otimes \mathcal{U}_j$ and displacement vectors $\mathtt{d}_i$ and $\mathtt{d}_j$ do not contain integer points.*

Note that we can check whether a polytope contains integer points using the Omega test by Pugh [Pug92]. Conservatively, we can check whether a polytope is empty by checking whether the system of inequalities that defines it is inconsistent. We can use Fourier-Motzkin elimination for this purpose [BW94, KB95].

7 Conclusion

In this paper we have shown how different unimodular transformations can be used for different statements in a loop nest. We have focussed on three aspects: some examples of their applicability, the legality of the transformed dependences, and the generation of efficient code for the transformed loop structure avoiding guards.

Multi-transformations have been presented as a mapping from an original iteration space to a new iteration space with more opportunities to exploit parallelism and/or data locality. We have also shown how this framework is valid for generating SPMD code from HPF-like directives specifying alignment and distribution of data in a compiler that uses the owner-computes rule. Multi-transformations include any kind of alignment: inter- or intra-dimensional, and uniform (like shift) or non-uniform (like skewing), and reordering of statements.

Chamski [Cha95] has proposed an algorithm for scanning unions of convex polytopes, based on the Parametric Integer Programming tool by Feautrier [Fea88]. Our techniques are based on Fourier-Motzkin elimination and seem to be more easily controlable in an interactive compilation environment. In [AT93, TALV93] a different strategy is proposed for scanning the union of a collection of iteration spaces. The main idea there is to isolate the itersection of these iteration spaces where all statements have to be executed, and scanning the remaining border by means of guarded statements. However, the transformations considered in those papers only differ in a displacement vector. Hence the resulting local iteration spaces will overlap to a large extend in all practical situation. This means that the borders of the intersection are small relative to the entire iteration space. Hence this technique is easy and acceptable for that situation. However, it is not acceptable when multi-transformations are used in which case there exists a large area outside the intersection. Hence, if we would apply this technique, then the part of the iteration space we would have to scan using guarded statements would be a large fraction of the entire iteration space.

The validity problem for multi-transformations is highly non-trivial. If there exists a dependence from statement S_1 to S_2, for iteration $\vec{\mathtt{I}}$ to $\vec{\mathtt{I}}'$, then the transformed dependence distance is $(\mathcal{U}_2 + \mathtt{d}_2)\vec{\mathtt{I}}' - (\mathcal{U}_1 + \mathtt{d}_1)\vec{\mathtt{I}}$ for possibly different unimodular transformations and displacements. Hence the transformed dependence distances are not easily related to the original dependence distances. This shows that even for uniform dependences it is difficult to decide the validity of a multi-transformation. These problems are worsened if we also consider non-uniform dependences as we do in this paper. The technique discussed in section 6 is a new and powerful approach to the validity problem. This technique relies heavily on Fourier-Motzkin elimination for constructing a collection of dependence polytopes and for deciding whether these are empty or not. Although this technique may be expensive for deeply nested loop structures and complex index functions, it has been observed before [Pug92, BW94] that in all practical situations the algorithm runs fast.

Pugh *et al.* [KPR94, PW92] has proposed a different approach to multi-transformations. Their approach is based on the Omega package, which contains a collection of procedures for manipulating formulas in the first order theory of Presburger arithmetic. Mappings and the validity problem for them can be formulated in this theory. Since the theory is decidable, although via a generally very expensive procedure (at least $\mathcal{O}(2^{2^{cn}})$ [Rab77]), it can be used for this purpose. The present work, however, is firmly rooted in well-known techniques and can be implemented in restructuring compilers possessing the machinery needed for unimodular transformations without great difficulty.

We are currently extending this framework in order to use non-singular integer matricesi [LP94]. The problem with these kind of matrices is that they induce different strides for the transformed local nests. By guarding the statements in the resulting global nest, one can easily achieve that the correct iterations are be-

ing executed for every transformed statement, but the code would become too inefficient.

References

[AT93] E. Ayguadé and J. Torres. Partitioning the statement per iteration space using non-singular matrices. In *Proc. 7th ACM Int. Conf. Supercomputing*, 1993.

[Ban88] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Norwell, 1988.

[Ban91] U. Banerjee. Unimodular transformations of double loops. In *Advances in Languages and Compilers for Parallel Processing*, chapter 10. The MIT Press, 1991.

[Ban93] U. Banerjee. *Loop Transformations for Restructuring Compilers.* Kluwer Academic Publishers, Norwell, 1993.

[BW94] A.J.C. Bik and H.A.G. Wijshoff. Implementation of Fourier-Motzkin elimination. Technical Report 94-42, Dept. of Computer Science, Leiden University, 1994.

[Cha94] Z.S. Chamski. Nested loop sequences: Towards efficient loop structures in automatic parallelization. In *Proc. 27th Hawaii Int. Conf. on System Sciences*, pages 14–22, 1994.

[Cha95] Z.S. Chamski. Enumeration of dense non-convex iteration sets. In *Proc. EuroMicro Workshop on Parallel and Distributed Processing*, pages 156–163. IEEE Press, 1995.

[DE73] G.B. Dantzig and B.C. Eaves. Fourier-Motzkin elimination and its dual. *J. of Combinatorial Theory*, 14:288–297, 1973.

[Dix91] K. Dixit. The SPEC benchmarks. *Parallel Computing*, 17:1195–1209, 1991.

[DRR93] A. Darte, T. Risset, and Y. Robert. Loop nest scheduling and transformations. In J.J. Dongarra and B. Tourancheau, editors, *Environments and Tools for Parallel Scientific Computing*, volume 6 of *Advances in Parallel Computing*, pages 309–332. North Holland, 1993.

[Fea88] P. Feautrier. Parametric integer programming. *Operations Research*, 22(3):243–268, 1988.

[Inc] Applied Parallel Research Inc. *xHPF Version 1.2, User's Guide*, may 1994 release edition.

[KB95] P.M.W. Knijnenburg and A.J.C. Bik. On reducing overhead in loops. Technical Report 95-07, Dept. of Computer Science, Leiden University, 1995.

[Kni94] P.M.W. Knijnenburg. On the validity problem for unimodular transformations. Technical Report 94-40, Dept. of Computer Science, Leiden University, 1994.

[KPR94] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. Technical Report UMIACS-TR-94-87, Dept. of Computer Science, Univ. of Maryland, 1994.

[Lam74] L. Lamport. The parallel execution of DO loops. *Comm. of the ACM*, 17(2):83–93, 1974.

[LP94] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *Int'l J. of Parallel Programming*, 22(2):183–205, 1994.

[Pol88] C. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, Boston, 1988.

[Pug92] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. *Comm. of the ACM*, 8:102–114, 1992.

[PW92] W. Pugh and D. Wonnacott. Going beyond integer programming with the Omega test to eliminate false data dependences. Technical Report CS-TR-2993, Dept. of Computer Science, Univ. of Maryland, 1992.

[Rab77] M.O. Rabin. Decidable theories. In J. Barwise, editor, *Handbook of Mathematical Logic*, chapter C.3, pages 595–629. North Holland, 1977.

[TALV93] J. Torres, E. Ayguadé, J. Labarta, and M. Valero. Align and distribute-based linear loop transformations. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proc. 6th Int. Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 321–339, Berlin, 1993. Springer Verlag.

[WL91] M.E. Wolf and M.S. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):430–439, 1991.

[Wol91] M.J. Wolfe. *Optimizing Supercompilers for Supercomputers.* The MIT Press, 1991.

[ZC90] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers.* ACM Press, New York, 1990.