Towards Unimodular Transformations for Non-perfectly Nested Loops *

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

Abstract

In this paper we discuss a possibility to extend unimodular transformations to non-perfectly nested loops. The main idea behind this extension is to convert a non-perfectly nested loop into a perfectly nested one by moving code into to innermost loop and properly guarding it to avoid multiple execution. This form of the loop can be viewed as an intermediate form for the transformation. Having obtained a perfectly nested loop, unimodular transformations can be applied in the usual way. The result of this transformation still contains guards. We show how to eliminate these guards. We illustrate our technique by transforming the ijk-version of LU-decomposition to the ikj-version.

1 Introduction

Automatic transformation of sequential programs into a parallel form is an important and challenging subject in High Performance Computing research [PW86, Wol91, ZC90]. Recently important progress has been made by the realization that DO loops scan higher dimensinal polytopes [AI91, WL91], that Fourier-Motzkin elimination or integer programming techniques can be used to generate bounds for DO loops [WL91, CFR93, Ban91], and that unimodular [Ban93, Ban91, WL91] or non-singular integer matrices [LP92] can be used for expressing and evaluating a host of transformations. Also, these techniques can be used for finding dependences and validating the use of a particular transformation [Ban93, Ban91, Fea91, Kni94, Pug92]. Complementary to this direction, Pugh has developed a framework based on Pressburger arithmetic to express transformations and evaluate dependence analysis [KPR94, KP94, PW93].

In this paper we discuss a possibility to extend the theory developed for loop transformations based on unimodular matrices to the case where the loop to be transformed is non-perfectly nested. The original theory of unimodular loop transformations only deals with perfectly nested loops. This is a mayor drawback for these kind of transformations to be used in practice. A large number of codes, notably codes for LU-decomposition, consist of non-perfectly nested loops. Hence it seems important that this restriction be removed.

*This research was partially supported by Esprit BRA APPARC under grant no. 6634

We propose the following strategy for applying unimodular transformations to non-perfectly nested loops. This strategy amounts to reducing the problem of applying a unimodular transformation to a non-perfectly nested loop to the problem of applying this transformation to a perfectly nested one. First, we rewrite the non-perfectly nested loop into a pefectly nested one. This is achieved by moving code from between the loops to the innerloop and guarding these code fragments in order to garantee that these fragments are only executed in the right initial iteration. The resulting loop can be considered as an *intermediate structure* for the transformation. We can then apply a unimodular transformation to this structure. The theory needed for applying such a transformation is well-known and is discussed in great detail by Banerjee [Ban93, Ban94]. After this phase, we have obtained a perfectly nested loop again (which differs from the original loop in the way it traverses its iteration space), which contains a number of guarded statements in its body. Obviously, this situation is not acceptable. The overhead induced by evaluating these guards, which will evaluate to 'true' in only a small number of iterations, is in general much to high for the loop to be used in an application code. Therefore we develop a theory of eliminating these guards. Using this theory, we can remove every guard present in the loop. The resulting code fragment may still not be in the preferred form. Hence other loop transformations, as developed in [Pol88, Wol91, ZC90], may have to be applied.

A careful reader may have noticed that the first step in the transformation strategy (obtaining a perfectly nested loop) is not always semantics preserving: If the innerloop is zero trip, then this scheme fails. Hence we have to impose a restriction on the class of loops we consider. This restriction precisely disallows loops with zero trip innerloops. We call these kind of loops non-zero-trip. We show how to recognize non-zero-trip loops automatically.

Feautrier [Fea92a, Fea92b], and Darte and Robert [DR93] discuss how to use integer programming techniques to find *schedules* for non-perfectly nested loops. They do not consider general transformations. In [KP94, KPR94], Pugh *et al.* discuss how the Omega package they have developed can be used for transforming non-perfectly nested loops. The techniques they propose are based on Pressburger arithmetic, which is decidable, whereas the central technique used in this paper is Fourier-Motzkin elimination [AI91, DE73, Ban93, BW94a]. In our opinion, a mayor advantage of using Fourier-Motzkin elimination is that this technique is easily implemented and quite powerful. In [Kni94, KB95, BW94b] we discuss other applications of Fourier-Motzkin elimination in a restructuring compiler. Moreover, Fourier-Motzkin elimination comprises a central step in the application of a unimodular or non-singular matrix transformation and hence will already be present in modern restructuring compilation environments. Hence the theory we discuss in this paper can presumably be embedded in an existing compiler easily.

The paper is organized as follows. In section 2 we give some background theory and introduce notation. In section 3 we define the class of non-perfectly nested loops we consider in this paper. We define the notion of non-zero-trip loops and show to recognize these loops. We also discuss briefly the validity problem for unimodular transformations in the present framework. In section 4 we show how to eliminate certain kinds of guards from loop structures. This section is stand alone and the techniques discussed can be applied in other contexts as well. In section 5 we give an extended example of the use of the techniques. We show how to transform the ijk-version of LU-decomposition to the ikj-version. Finally, in section 6 we give a brief discussion.

Acknowledgement. The author wishes to thank Arjan Bik for carefully reading a draft version of this paper.

2 Preliminaries

In this section we give some preliminaries we use in the rest of 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 variables or 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, $a_i \in \mathbf{Z}$.

2. *Let $a \in \mathbf{Z}$ and let B be a basic bound. A* simple lowerbound *is an expression $\left\lceil \frac{1}{a}B \right\rceil$. A* simple upperbound *is an expression $\left\lfloor \frac{1}{a}B \right\rfloor$.*

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.*

Note that each basic bound can be considered a simple lower or upperbound, and each simple bound can be considered a compound bound. In the sequel of the paper we will make this identification when no confusion can arise. Basic, simple and compound bounds are also called *admissible*. All other expressions for bounds, like $\mathtt{I}_1 * \mathtt{I}_2$ or an indirection $\mathtt{IND(I)}$, are called *inadmissible*. The reason for this distinction is that admissible bounds can be used and are obtained in the process of Fourier-Motzkin elimination, see below.

Every perfectly nested loop $\mathcal{L}$ with basic bounds gives rise to a system of inequalities $\mathcal{S}(\mathcal{L})$, given by

$$\mathcal{S}(\mathcal{L}) = \begin{cases} \mathtt{L}_1 & \leq & \mathtt{I}_1 & \leq & \mathtt{U}_1 \\ & & \vdots & & \\ \mathtt{L}_n & \leq & \mathtt{I}_n & \leq & \mathtt{U}_n \end{cases} \quad (1)$$

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 $\mathtt{U}_i$ only involves variables $\mathtt{I}_1, \ldots, \mathtt{I}_{i-1}$. We call this the *standard form* of a system of inequalities. Note that for loops with compound lower and upperbounds we also can define such a system of inequalities. Since compound upperbounds may contain floor and minimum fuction, we use the following equivalences to obtain a standard form.

- $\mathtt{I} \leq \left\lfloor \frac{1}{a}B \right\rfloor$ iff $a\mathtt{I} \leq B$ (since $\mathtt{I}$ is integer).
- $\mathtt{I} \leq \min(U_1, \ldots, U_m)$ iff $\mathtt{I} \leq U_1$ & $\cdots$ & $\mathtt{I} \leq U_m$.

For compound lowerbounds a similar standard form can be deduced.

Any system of inequalities involving the variables $\mathtt{I}_1, \ldots, \mathtt{I}_n$ can be brought in standard form using *Fourier-Motzkin elimination* [DE73, Ban93]. We try to give some intuition. Consider a set $\mathcal{C} = \{\varphi_1, \ldots, \varphi_k\}$ of inequalities, where each inequality φ_i is of the form $e \leq e'$ for two affine expressions

e and e' over the variables $\mathtt{I}_1, \ldots, \mathtt{I}_n$. Then Fourier-Motzkin elimination consists of the following process. First, rewrite all expressions involving the variable $\mathtt{I}_n$ to the form $L \leq \mathtt{I}_n, \ldots, \mathtt{I}_n \leq U$. Then each inequality obtained in this way bounds $\mathtt{I}_n$ by expressions only involving the variables $\mathtt{I}_1, \ldots, \mathtt{I}_{n-1}$. Hence these expressions can be used to generate loop bounds. Now consider the system obtained by forming all inequalities $L \leq U$, for lowerbounds L and upperbounds U from the previous step, together with all inequalities from the original system not involving $\mathtt{I}_n$. This is a system of inequalities only involving the variables $\mathtt{I}_1, \ldots, \mathtt{I}_{n-1}$. Hence we can recursively continue the process, finally ending with a system of inequalities which consist only of the variable $\mathtt{I}_1$ and contstants.

Let $[\![\mathcal{C}]\!] = \{x \in \Re^n : \varphi_1(x) \wedge \cdots \wedge \varphi_k(x)\}$. We say that $\mathcal{C}$ is *inconsistent* iff $[\![\mathcal{C}]\!] = \emptyset$. We say that an inequality φ is *redundant* for $\mathcal{C}$ iff $[\![\mathcal{C} \cup \{\varphi\}]\!] = [\![\mathcal{C}]\!]$. The Fourier-Motzkin elimination algoritm can be used to decide whether $\mathcal{C}$ is inconsistent. We also have the following lemma, the proof of which is elementary.

Lemma 2.2 *φ is redundant for $\mathcal{C}$ if and only if $\mathcal{C} \cup \{\neg\varphi\}$ is inconsistent.*

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 2.3 *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, where* I *is a new loop variable.*

```
DO  I = L,U
      ℒ
ENDDO
```

Note that any perfectly nested loop can be considered as a nested loop sequence. Note also that two different NLSs $\mathcal{N}$ and $\mathcal{N}'$ can scan the same iteration space in the same order. For instance, $\mathcal{N}'$ may be obtained from $\mathcal{N}$ by iteration set partitioning. We call such loop structures $\mathcal{N}$ and $\mathcal{N}'$ *(semantically) equivalent.*

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 of the block. We call this loop nest the *local nest* of the block.

3 Transforming non-perfectly nested loops

In this section we define the class of loop structures we consider in this paper. This family of loops is called *non-perfectly nested loops*, and these loops form the input of the transformation technique we discuss. Then we show how to convert these loops into perfectly nested ones under some mild assumptions, namely, that the loop is non-zero-trip (see below). We show how these assumptions can be validated for a given loop automatically. Finally we discuss briefly how the validity of a transformation can be decided after the conversion.

3.1 Non-perfectly nested loops

In this section we discuss a simple way of transforming a non-perfectly nested loop obeying some additional constraints into a perfectly nested loop. This perfectly nested loop can be used as input for a unimodular transformation. The class of non-perfectly nested loops we consider in this paper consists of loops of the following form.

```
DO  I_1 = L_1, U_1
    S_1
    DO  I_2 = L_2, U_2
        S_2
        ...
        DO  I_n = L_n, U_n
            S_n
        ENDDO
        ...
        S'_2
    ENDDO
    S'_1
ENDDO
```

We assume that all bounds $\mathtt{L}_k$ and $\mathtt{U}_k$ are *basic bounds* with respect to the collection of variables $\{\mathtt{I}_1, \ldots, \mathtt{I}_{k-1}\}$. The reason why the bounds need to be basic is explained by Proposition 3.1 below. This condition is not Note that this implies that $\mathtt{L}_1$ and $\mathtt{U}_1$ are integer constants. For each i, S_i and S'_i are assumed to be blocks of statements, not containing `goto` statements which jump out of this block. We furthermore assume that every block S_k and S'_k only references loop indices $\mathtt{I}_1, \ldots, \mathtt{I}_k$.

The basic idea behind converting an imperfectly nested loop into a perfectly nested one is the following. A statement appearing before a loop can also be executed in the first iteration of the loop before the body of the loop. Similarly, a statement appearing after the loop can also be executed in the last iteration of the loop. At this point we assume that none of the loops are empty. Hence, the following two program fragments are equivalent.

```
S_1                    DO  I = L, U
DO  I = L, U               IF (I = L) S_1
    S_2                    S_2
ENDDO                      IF (I = U) S_3
S_3                    ENDDO
```

This generalizes easily to loops which are more deeply nested. To be precise, the following two program fragments are equivalent.

```
S1                          DO  I1 = L1,U1
DO  I1 = L1,U1                  ...
    ...                         DO  In = Ln,Un
    DO  In = Ln,Un                  IF (I1 = L1 & ··· & In = Ln)  S1
        S2                          S2
    ENDDO                           IF (I1 = U1 & ··· & In = Un)  S3
    ...                         ENDDO
ENDDO                           ...
S3                          ENDDO
```

The above two observations immediately lead to a procedure for converting a non-perfectly nested loop into a perfectly nested one. This perfectly nested loop can be transformed using unimodular transformations. The result will be a perfectly nested loop of (more or less) the same shape, namely, a loop in which a great number of statements are guarded by conditions on the loop variables. These guards are conjunctions of inequalities between affine expressions in the loop variables, and are called *affine guards*. For a formal definition, see section 4. The loop as it stands is far too inefficient to be used in practice: for each iteration a number of guards has to be evaluated, which most of the time will evaluate to false. In the sequel of the paper we show how to remove affine guards. The resulting loop structure will be a nested loop sequence (see section 2).

3.2 Non-zero-trip loops

In this section discuss the possibility that subloops in a non-perfectly nested loop may be empty. First note that the transformation of a non-perfectly nested loop into a perfectly nested one is not semantically valid if this loop would contain empty subloops. The condition we have to impose on loops in order to avoid this situation is the following. Consider a loop with loop indices $\mathtt{I}_1, \ldots, \mathtt{I}_n$ as defined above. If, for all $1 \leq k < n$, it is the case that for all values of the loop indices $\mathtt{I}_1, \ldots, \mathtt{I}_k$, the lowerbound $\mathtt{L}_{k+1}$ is smaller than or equal to the upperbound $\mathtt{U}_{k+1}$, then each loop in the nest will always have at least one iteration. Hence loops having this property can be turned into perfectly nested loops using the transformation described above. Loops having this property are called *non-zero-trip (NZT) loops*.

We now show how to recognize NZT loops using Fourier-Motzkin elimination. First we observe that the condition for a loop with loop indices $\mathtt{I}_1, \ldots, \mathtt{I}_n$ and basic bounds to be NZT is equivalent to the condition that there does not exist $1 \leq k < n$ such that for some value of the loop indices $\mathtt{I}_1, \ldots, \mathtt{I}_k$ (within their bounds) the lowerbound $\mathtt{L}_{k+1}$ is larger than the upperbound $\mathtt{U}_{k+1}$.

Next we can state the following proposition concerning the testing of this new condition.

Proposition 3.1 *A loop with loop indices $\mathtt{I}_1, \ldots, \mathtt{I}_n$ and basic bounds is NZT if and only if for all $1 \leq k < n$, the following system of inequalities*

$$\mathtt{L}_1 \leq \mathtt{I}_1 \leq \mathtt{U}_1, \ldots, \mathtt{L}_k \leq \mathtt{I}_k \leq \mathtt{U}_k, \mathtt{U}_{k+1} < \mathtt{L}_{k+1}$$

is inconsistent.

Using the above lemma we can check whether a given loop is NZT by applying the Fourier-Motzkin elimination algorithm $n-1$ times, where n is the nesting depth of the loop.

The statement of the above proposition does not hold if the loop bounds are not basic. Consider the following loop.

```
DO  I = 1,2
    DO  J = ⌈I/3⌉ , ⌊I/3⌋
        ...
    ENDDO
ENDDO
```

Then this loop would be NZT according to the proposition but it is empty. The reason for this is that from the viewpoint of Fourier-Motzkin elimination, the iteration space is a (non-empty) line segment. However, this line segment does not contain integer points, which are the iteration points of the loop.

3.3 Validity of a unimodular transformation

In general, a unimodular transformation $\mathcal{U}$ is valid iff for each non-trivial dependence distance vector d it is the case that $\mathcal{U}d$ is lexicographically positive. In [Kni94] we have shown how to use Fourier-Motzkin elimination to obtain a polytope representing all dependence distance vectors present in a loop. We have also shown in that paper how to deal with affine guards when constructing this polytope. Briefly, we restrict the polytope to those values for which the guard holds. Space considerations prvent us from going into details. The reader is referred to [Kni94] for a proof of the next propostion.

Proposition 3.2 *Given a loop $\mathcal{L}$ containing affine guards and a unimodular transformation $\mathcal{U}$. Then we can construct a polytope $\mathcal{P}$ and a matrix $\mathcal{U}^*$ such that $\mathcal{U}$ is valid if and only if $\mathcal{U}^*\mathcal{P}$ is lexicographicallly positive. This last assertion can be checked effectively.*

4 Removing affine guards

In this section we show how to remove certain kinds of `IF`-statements from nested loop sequences. We call these `IF`-statements *affine guards.*

Definition 4.1 *Let $\mathcal{L}$ be a perfectly nested loop.*

1. *A condition of the form $e \leq e'$ where e and e' are affine expressions in the loop variables, is called a* simple affine condition *for $\mathcal{L}$.*
2. *The conjunction of one or more simple affine conditions for $\mathcal{L}$ is called an* affine condition *for $\mathcal{L}$.*
3. *An `IF`-statement of which the condition is an affine condition is called an* affine guard *for $\mathcal{L}$.*

Note that we can express most other comparison operations on integers by the following identifications.

- $n < m$ iff $n + 1 \leq m$

- $n = m$ iff $n \leq m$ and $m \leq n$
- $n \geq m$ iff $m \leq n$

Suppose we have a (local) loop nest containing an affine guard.

```
DO  I_1 = L_1, U_1
    ...
    DO  I_n = L_n, U_n
        IF (e_1 ≤ e'_1 & ··· & e_m ≤ e'_m)  S_1
        S_2
    ENDDO
    ...
ENDDO
```

Then the affine condition $(e_1 \leq e'_1 \ \& \cdots \& \ e_m \leq e'_m)$ determines a subspace of the iteration space of this loop. The basic idea is to extract this subspace from the iteration space. In this subspace, both statements S_1 and S_2 have to be executed. In the rest of the iteration space only statement S_2 needs to be executed. We proceed as follows.

Consider the system of inequalities given by the loop bounds and the affine conditions:

$$\mathtt{L}_1 \leq \mathtt{I}_1 \leq \mathtt{U}_1, \ldots, \mathtt{L}_n \leq \mathtt{I}_n \leq \mathtt{U}_n, e_1 \leq e'_1, \ldots, e_m \leq e'_m$$

We have three possibilities.

1. The system is inconsistent. This means that for no value of the loop variables within the loop bounds, the condition holds. Hence we may remove the IF-statement from the nest without affecting its semantics.

2. The affine condition is redundant with respect to the system of inequalities defined by the loop bounds. This means that for every value of the loop variables within the loop bounds the condition holds. Hence we can replace the IF-statement with its body without affecting the semantics of the nest.

3. The system is neither inconsistent, nor is the condition redundant. This is the most interesting and probably most frequent case. In this case, the body of the IF-statement has to be executed in part of the iteration space of the loop, and not in the remainder. The rest of this section is devoted to dealing with this case.

Using Fourier-Motzkin elimination we can rewrite the system of inequalities given by the loop bounds and the affine conditions to the form

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

This system defines a convex polytope within the original iteration space. In this subspace the condition holds. Hence it defines that part of the iteration space in which both S_1 and S_2 have to be executed. Note that, by construction, we have for every $1 \leq k \leq n$,

$$\mathtt{L}_k \leq \mathtt{L}'_k \leq \mathtt{U}'_k \leq \mathtt{U}_k$$

Hence we can use the old bounds of the loop to determine the parts of the iteration space in which only S_2 has to be executed.

Intuitively, we would like to generate the following code.

```
DO  I1 = L1, L1' - 1        DO  I1 = L1', U1'        DO  I1 = U1' + 1, U1
      S2                          S1                       S2
ENDDO                             S2                 ENDDO
                            ENDDO
```

However, the lower and upperbounds for the subspace may be general bounds. That is, $\mathtt{L}'$ may be of the form $\max(e_1, e_2, \ldots)$. Hence the first loop above is inadmissible and we cannot recursively continue this process and remove guards from S_2.

Fortunately, we can remedy this situation as follows. Consider a polytope given by $\mathtt{L} \leq \mathtt{I} \leq \mathtt{U}$, and a half-space given by $\mathtt{L}' \leq \mathtt{I}$, where $\mathtt{L}'$ is simple. Then the following code scans first the part of the polytope which lies outside the half-space, and then the intersection of the polytope and the half-space. In the first part, we execute a block of statements S_1 and in the second part (the intersection) a block S_2. We call the first part the *preloop*, and the second part the *intersection loop*.

```
DO  I = L, min(L' - 1, U)        DO  I = max(L, L'), U
      S1                               S2
ENDDO                            ENDDO
```

Likewise, for a half-space given by $\mathtt{I} \leq \mathtt{U}'$ with $\mathtt{U}'$ simple, we generate the following code. The first loop scans the intersection of the polytope and the half-space, and the second loop scans the remainder of the polytope. In the instersection we execute a block S_2 and in the remainder of the polytope a block S_1. We call the first loop the *intersection loop*, and the second loop the *postloop*.

```
DO  I = L, min(U, U')        DO  I = max(L, U' + 1), U
      S2                           S1
ENDDO                        ENDDO
```

Since both $\mathtt{L}'$ and $\mathtt{U}'$ are assumed to be simple, the resulting loop structures are admissible. Please note that irrespective of the position of the half-space and the polytope, the loops as defined above precisely scan the correct portion of the entire space. Note also that some of the loops may be empty. Using these observations we arrive at the following algorithm for isolating a polytope inside another polytope, resulting in a nested loop sequence scanning this space. This algorithm is based on a technique described in [BW95].

Algorithm Let $\mathcal{L}$ and $\mathcal{L}'$ be perfectly nested loops, scanning polytopes $\mathcal{P}$ and $\mathcal{P}'$, respectively. Let $\mathcal{L}$ be given by the system of inequalities

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

where all bounds may be compound. Let $\mathcal{L}'$ be given by a collection of half-spaces

$$l_1 \leq \mathtt{I}_1, \ldots, l_k \leq \mathtt{I}_1, \mathtt{I}_1 \leq u_1, \ldots, \mathtt{I}_1 \leq u_m, \quad \ldots \quad , l_1' \leq \mathtt{I}_n, \ldots, \mathtt{I}_n \leq u_{m'}'$$

where each bound is simple. Suppose that we want to execute a block of statements S_1 in the intersection of $\mathcal{P}$ and $\mathcal{P}'$, and a block S_2 in $\mathcal{P} - \mathcal{P}'$. We construct a loop structure that performs this task as follows.

For each i from 1 to n do the following.

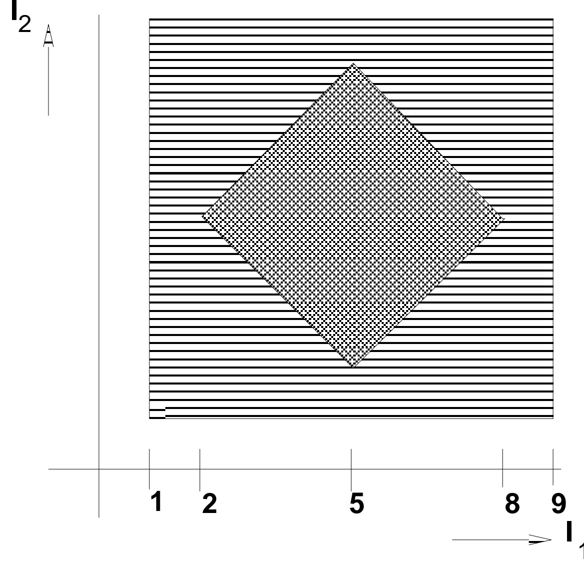

Figure 1: Two intersecting iteration spaces

- Let the *active* loop be $\mathtt{L}_i \leq \mathtt{I}_i \leq \mathtt{U}_i$.
- For each half-space $l \leq \mathtt{I}_i, \ldots, \mathtt{I}_i \leq u$ from $\mathcal{P}'$ do
  - If the half-space is a lowerbound, then partition the active loop accordingly. Set the current active loop to the intersection loop of this partitioning.
  - If the half-space is an upperbound, then partition the active loop accordingly. Set the current active loop to the intersection loop of this partioning.
- After this phase we have an ordered list of pre- and postloops, and an active loop. For each pre- and postloop, generate a loop with body the loop defined by the bounds from $\mathcal{L}$

  $$\mathtt{L}_{i+1} \leq \mathtt{I}_{i+1} \leq \mathtt{U}_{i+1}, \ldots, \mathtt{L}_n \leq \mathtt{I}_n \leq \mathtt{U}_n$$

  and body S_2.
- The body of the active loop is given the loop structure obtained in the next iterations over i. If $i = n$, then the body is given by S_1.

End Algorithm

Example Consider the following loops $\mathcal{L}$ and $\mathcal{L}'$. $\mathcal{L}$ is given by $1 \leq \mathtt{I}_1 \leq 9, 1 \leq \mathtt{I}_2 \leq 9$, and $\mathcal{L}'$ is given by $2 \leq \mathtt{I}_1 \leq 8,\ \max(7 - \mathtt{I}_1, \mathtt{I}_1 - 3) \leq \mathtt{I}_2 \leq \min(3 + \mathtt{I}_1, 13 - \mathtt{I}_1)$. See figure 1. Then the above algorithm yields the following pre-, intersection and postloop, respectively, for $\mathtt{I}_1$:

$$1 \leq \mathtt{I}_1 \leq 1 \quad 2 \leq \mathtt{I}_1 \leq 8 \quad 9 \leq \mathtt{I}_1 \leq 9$$

The body of the pre- and postloop consists of the innerloop of $\mathcal{L}$, with body S_2. Continuing with the active loop, we obtain the following two pre-, one intersection and two postloops, respectively,

for $\mathtt{I}_2$:

$$\begin{array}{rcccl}
1 & \leq & \mathtt{I}_2 & \leq & \min(6-\mathtt{I}_1, 9) \\
\max(7-\mathtt{I}_1, 1) & \leq & \mathtt{I}_2 & \leq & \min(\mathtt{I}_1-4, 9) \\
\max(7-\mathtt{I}_1, 1, \mathtt{I}_1-3) & \leq & \mathtt{I}_2 & \leq & \min(3+\mathtt{I}_1, 9, 13-\mathtt{I}_1) \\
\max(7-\mathtt{I}_1, 1, \mathtt{I}_1-3, 14-\mathtt{I}_1) & \leq & \mathtt{I}_2 & \leq & \min(3+\mathtt{I}_1, 9) \\
\max(4+\mathtt{I}_1, 7-\mathtt{I}_1, 1, \mathtt{I}_1-3) & \leq & \mathtt{I}_2 & \leq & 9
\end{array}$$

The body of the pre- and postloops consists of the block S_2; the body of the active loop consists of S_1.

Please observe that for some values of $\mathtt{I}_1$, some of the above loops over $\mathtt{I}_2$ are empty. For example, for $\mathtt{I}_1 = 2$, the second and the fourth loop are empty. Moreover, a number of bounds are redundant, that is, are always satisfied given the other bounds. For example, since $1 \leq \mathtt{I}_1 \leq 10$, $\min(6 - \mathtt{I}_1, 10) = 6 - \mathtt{I}_1$. In [KB95] we discuss a number of techniques to remove these empty loops and redundant bounds. Using these techniques, we can generate the following loop structure to scan the iteration space depicted in figure 1. Alternatively, we could have used the techniques described by Chamski [Cha94] which are based on the Parametric Integer Programming tool by Feautrier [Fea88].

```
DO  I1 = 1,1
    DO  I2 = 1,9
        S2
    ENDDO
ENDDO
```

```
DO  I1 = 2,5
    DO  I2 = 1,6 - I1
        S2
    ENDDO
    DO  I2 = 7 - I1,3 + I1
        S1
    ENDDO
    DO  I2 = 4 + I1,9
        S2
    ENDDO
ENDDO
```

```
DO  I1 = 6,8
    DO  I2 = 1,I1 - 4
        S2
    ENDDO
    DO  I2 = I1 - 3,13 - I1
        S1
    ENDDO
    DO  I2 = 14 - I1,9
        S2
    ENDDO
ENDDO
```

```
DO  I1 = 9,9
    DO  I2 = 1,9
        S2
    ENDDO
ENDDO
```

End Example

Until now we have discussed how to eliminate affine guards in perfectly nested loops. However, the result of this elimination is a nested loop sequence. Hence, if we want to continue the process of eliminating affine guards, we have to extend the theory to the case of nested loop sequences. Briefly, we can apply the following strategy.

- Given an affine guard in a nested loop sequence $\mathcal{N}$, first construct its local nest. This is a perfectly nested loop $\mathcal{L}$.
- Eliminate the affine guard from $\mathcal{L}$ using the above procedure, yielding a nested loop sequence $\mathcal{N}'$.
- Replace the local nest $\mathcal{L}$ in $\mathcal{N}$ by the nested loop sequence $\mathcal{N}'$. Since $\mathcal{N}'$ scans exactly the same iteration space as $\mathcal{L}$ does, this is straightforward and details are left as an exercise for the reader.

We arrive at the following proposition.

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

5 Example: LU-decomposition

In this section we give an extended example of the use of the transformations defined in the previous sections. In particular, we show how to transform the *ijk*-version of LU-decomposition to the *ikj*-version using only standard transformations. In fact, we can show that *every* version of LU-decomposition can be obtained this way. We have chosen for this particular version because it is easy and serves well as an illustration of the proposed techniques. This example shows both the power of the proposed transformation strategy, and also the way it could be used in an interactive restructuring environment.

Consider the following kernel for LU-decomposition for a 1000×1000 matrix A.

```
DO  I = 1,999
    DO  J = I + 1,1000
        A(J,I) = A(J,I)/A(I,I)
        DO  K = I + 1,1000
            A(J,K) = A(J,K) - A(J,I) * A(I,K)
        ENDDO
    ENDDO
ENDDO
```

The following loop structure is the result of transforming this non-perfectly nested loop into a perfectly nested one.

```
DO  I = 1,999
    DO  J = I + 1,1000
        DO  K = I + 1,1000
            IF(K = I + 1)  A(J,I) = A(J,I)/A(I,I)
            A(J,K) = A(J,K) - A(J,I) * A(I,K)
        ENDDO
    ENDDO
ENDDO
```

In order to obtain the *ikj*-version of the loop, we first apply the unimodular transformation specified by the matrix

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

This is simply the interchange of the `J` and `K` loop. We obtain the following loop.

```
DO  I = 1,999
    DO  K = I + 1,1000
        DO  J = I + 1,1000
            IF(K = I + 1)  A(J,I) = A(J,I)/A(I,I)
            A(J,K) = A(J,K) - A(J,I) * A(I,K)
        ENDDO
    ENDDO
ENDDO
```

Now we eliminate the IF-statement and obtain the following loop.

```
DO  I = 1,999
    DO  K = I + 1,I + 1
        DO  J = I + 1,1000
            A(J,I) = A(J,I)/A(I,I)
            A(J,K) = A(J,K) - A(J,I) * A(I,K)
        ENDDO
    ENDDO
    DO  K = I + 2,1000
        DO  J = I + 1,1000
            A(J,K) = A(J,K) - A(J,I) * A(I,K)
        ENDDO
    ENDDO
ENDDO
```

Next we apply loop distribution to the first loop on K. Moreover, since the first loop in the resulting structure is a single trip loop, in which the body does not depend on the loop index K, this loop can be removed. This yields the following code.

```
DO  I = 1,999
    DO  J = I + 1,1000
        A(J,I) = A(J,I)/A(I,I)
    ENDDO
    DO  K = I + 1,I + 1
        DO  J = I + 1,1000
            A(J,K) = A(J,K) - A(J,I) * A(I,K)
        ENDDO
    ENDDO
    DO  K = I + 2,1000
        DO  J = I + 1,1000
            A(J,K) = A(J,K) - A(J,I) * A(I,K)
        ENDDO
    ENDDO
ENDDO
```

Finally, a restructuring compiler can recognize that the two loops on K can be merged to yield the ikj-version of LU-decomposition.

```
DO  I = 1,999
    DO  J = I + 1,1000
        A(J,I) = A(J,I)/A(I,I)
    ENDDO
    DO  K = I + 1,1000
        DO  J = I + 1,1000
            A(J,K) = A(J,K) - A(J,I) * A(I,K)
        ENDDO
    ENDDO
ENDDO
```

6 Conclusion

In this paper we have discussed a way of using unimodular transformations for non-perfectly nested loops. We had to impose the condition that the loops be non-zero-trip in order for our approach to be semantically valid. We have given an algorithm to decide whether a given loop is NZT. The result of the transformation is a perfectly nested loop containing affine guards. We have given an algorithm to eliminate these guards. This algorithm can be used in other situations as well.

There are a number of extensions to the theory of this paper. First, we would like to remove the NZT assumption. The problem now arises how to enforce a single trip of an innerloop, which was empty in the original code. A possibility is to rewrite the loop bounds as $\mathtt{L} \leq \mathtt{I} \leq \max(\mathtt{L}, \mathtt{U})$. However, now we have introduced an inadmissible bound. Nevertheless, this bound has a very special form which might be exploited. Second, we would like to use non-singular integer instead of unimodular matrices. The problem that arises is that the resulting iteration space contains holes, and the transformed loop has non-unit stride. The `IF`-elimination algorithm does not trivially extend to this case.

We have a final remark concerning `IF`-elimination. Although this algorithm obviously can be used to remove all affine guards in a loop, there exists the possibility of code explosion. This problem is worsened if one tries to simplify the bounds obtained in the elimination process. Nevertheless, as the example in section 5 shows, we feel that in practice this will not cause serious problems.

References

[AI91] C. Ancourt and F. Irigoin. Scanning polyhedra with DO loops. In *Proc. 3rd ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, pages 39–50, 1991.

[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.

[Ban94] U. Banerjee. *Loop Parallelization.* Kluwer Academic Publishers, Norwell, 1994.

[BW94a] 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.

[BW94b] A.J.C. Bik and H.A.G. Wijshoff. Iteration set partitioning. Submitted to ICS'95, 1994.

[BW95] A.J.C. Bik and H.A.G. Wijshoff. On strategies for generating sparse codes. Technical Report 95-01, Dept. of Computer Science, Leiden University, 1995.

[CFR93] J.-F. Collard, P. Feautrier, and T. Risset. Construction of DO loops from systems of affine constraints. Technical Report 93-15, LIP, Lyon, 1993.

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

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

[DR93] A. Darte and Y. Robert. Affine-by-statement scheduling of uniform loop nests over parametric domains. 1993.

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

[Fea91] P. Feautrier. Dataflow analysis of array and scalar references. *Int. J. of Parallel Programming*, 20(1):23–53, 1991.

[Fea92a] P. Feautrier. Some efficient solutions to the affine scheduling problem I. one-dimensional time. *Int. J. of Parallel Programming*, 21(5):313–347, 1992.

[Fea92b] P. Feautrier. Some efficient solutions to the affine scheduling problem II. multi-dimensional time. *Int. J. of Parallel Programming*, 21(6):389–420, 1992.

[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.

[KP94] W. Kelly and W. Pugh. Finding legal reordering transformations using mappings. In *Proc. 7th Ann. Workshop on Languages and Compilers for Parallel Computing*, 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.

[LP92] W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. In *Proc. 5th Workshop on Language and Compilers for Parallel Computers*, 1992.

[Pol88] C. Polychronolpoulos. *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.

[PW86] D.A. Padua and M.J. Wolfe. Advanced compiler optimizations for supercomputers. *Comm. of the ACM*, 29(12):1184–1201, 1986.

[PW93] W. Pugh and D. Wonnacot. An exact method for analysis of value-based array data dependence. In *Proc. 6th Ann. Workshop on Languages and Compilers for Parallel Computing*, volume 768 of *Lecture Notes in Computer Science*, pages 546–565. Springer Verlag, Berlin, 1993.

[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. 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.