GENERALISING THE UNIMODULAR APPROACH TO RESTRUCTURE IMPERFECTLY NESTED LOOPS

Jingling Xue
Department of Mathematics, Statistics and Computing Science
University of New England
Armidale 2351, Australia

ABSTRACT

Although overcoming some limitations of the generate-and-test approach, unimodular transformations are limited to perfect loop nests only. Extending the unimodular approach, this paper describes a framework that enables the use of unimodular transformations to restructure imperfect loop nests. The concepts used previously for perfect loop nests, such as iteration vector, iteration space and lexicographic order, are generalised and some new concepts like preorder tree are introduced. Multiple unimodular transformations are allowed, one each statement in the loop nest. A code generation algorithm is provided that produces a possibly imperfect loop nest to scan an iteration space that is given as a union of sets of affine constraints.

Keywords: Imperfectly nested loops, loop transformation, iteration space, preorder tree.

1. Introduction

The automatic restructuring of Fortran code for parallel machines has focused mainly on the exploitation of DO loops. In the past decade or so, the state of restructuring compilers has evolved from the generate-and-test approach\(^1\), to the unimodular approach\(^2\,3\,4\,5\) and its generalised non-singular model\(^6\), and more recently to schedules\(^7\). A compiler based on generate-and-test focused on the application of individual transformations on pairs of loops. It generates an order of transformations, tests for legality, generates the code, and then repeats the same process until a desirable solution is attained. Although applicable to both perfect and imperfect loop nests, this approach is expensive. In contrast, the unimodular approach unifies three individual transformations: loop interchange, skewing, and reversal. A compiler based on this approach finds one single transformation by solving one single optimisation problem for both legality and parallelism/locality. Thus, this approach generates code only once, by constructing DO loops from a conjunction of affine constraints. One major limitation is that the unimodular approach can be applied to perfect loop nests only.

A recent study shows that a majority of the loops in scientific code are imperfectly nested, and a majority of the performance-increasing techniques developed in the past two decades assume that loops are perfectly nested\(^8\). We can identify two approaches to handling imperfect loop nests. One approach converts imperfect to perfect loop nests and then applies the unimodular approach. Techniques used include loop distribution, loop fusion, Abu-Sufah’s non-basic-to-basic loop trans-
formation\textsuperscript{9}, and scalar forward substitution followed by loop distribution\textsuperscript{8}. The other approach, which is also the approach we take in this paper, aims at some extensions of the unimodular approach. In\textsuperscript{10}, matrices are used to model such transformations as loop distribution and loop fusion. In\textsuperscript{7}, unimodular transformations are generalised to schedules so that a much broader set of transformations can be represented. A code generation algorithm for implementing schedules was also developed. However, their work was not geared specifically for restructuring loop nests.

In this paper, we describe a framework that enables the use of unimodular matrices to restructure imperfect loop nests. The concepts previously used for perfect loop nests, such as iteration vector, iteration space and lexicographic order, are generalised to characterise imperfect loop nests (Section 2). Preorder trees are introduced as an intermediate representation to facilitate the generation of the restructured code (Section 3). We show how to restructure an imperfect loop nest by applying multiple unimodular transformations, one on each statement of the loop nest (Section 4). Finally, we discuss a code generation method that takes an iteration space and produces a possibly imperfect loop nest to scan the iteration space (Section 5). Our algorithm subsumes those for perfect loop nests\textsuperscript{2,4} while retaining the philosophy of generating code by scanning affine constraints.

2. Modeling Loop Nests

In this preliminary study, the class of loop nests is restricted to those that are specified by the following simple grammar:

\[
L \rightarrow LL | \text{do-loop} L | \text{atomic-statement}
\]

where the nonterminal \(L\) represents a loop nest, the terminal do-loop a DO loop, and the terminal \text{atomic-statement} an atomic statement. An \text{atomic statement} is a code segment that is transformed as a basic unit. In the context of loop transformations, an atomic statement is usually a single assignment statement, possibly guarded with some conditionals. In this paper, all loops are assumed to have unity loop strides.

Informally, a loop nest is \textit{perfect} if all statements are inside the innermost loop of the loop nest and \textit{imperfect} otherwise.

Example 1

\[
\begin{array}{|c|c|}
\hline
\text{(a) A \textbf{PERFECT LOOP NEST}} & \text{(b) A \textbf{IMPERFECT LOOP NEST}} \\
\hline
L_1 & \text{DO } i = 1, n \\
S_1 & a(i) = b(i) + 1 \\
S_2 & d(i) = a(i - 1) \ast d(i) \\
L_1 & \text{DO } i = 1, n \\
S_1 & b(i) = 0 \\
L_2 & \text{DO } j = 1, i \\
S_2 & b(i) = b(i) + a(i, j) \\
\hline
\end{array}
\]

To indicate the concatenation of two vectors \(a\) and \(b\), we write \(a \# b\). For example, \((4, 2) \# (3) = (4, 2, 3)\) and \((1) \# () = (1)\), where () denotes the null vector.

2.1. Iteration Vector of a Statement

An imperfect loop nest contains statements that are imperfectly nested in loops of varying depths. To identify an instance of such a statement by an iteration vector, we must include in the iteration vector not only the values of surrounding
loop variables but also the values indicating the lexical order of the statement.

The following procedure, when called with arguments L and ( ), will return the iteration vector V for each of the statements in a loop nest L.

**Algorithm 1** Procedure to compute the iteration vectors from a loop nest.

\[
\text{Iteration\_Vector}(L, V)
\]

CASE L OF

\[
\text{DO } x_{k+1} = \cdots L';
\]

\[
\text{Iteration\_Vector}(L', V\#(x_{k+1}))
\]

\[
L_1 \cdots L_m;
\]

FOR \( h = 1, m \) DO Iteration\_Vector \((L_h, V\#(h))\)

Atomic Statement \( S \) : /* the base case */

OUTPUT \( S : V \)

In this and Algorithms 2 and 3, it is assumed that \( L \), if it is an atomic statement, will match the \( L_1 \cdots L_m \) (with \( m = 1 \)) before it matches the Atomic Statement \( S \).

Using this procedure, we obtain the iteration vectors for Example 1:

<table>
<thead>
<tr>
<th>(a) A PERFECT LOOP NEST</th>
<th>(b) AN IMPERFECT LOOP NEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_1 ) : (1, 1)</td>
<td>( S_1 ) : (1, 1)</td>
</tr>
<tr>
<td>( S_2 ) : (1, 2)</td>
<td>( S_2 ) : (1, 2, j, 1)</td>
</tr>
</tbody>
</table>

By definition, the iteration vectors for different statements satisfy two properties:

1. The \( i \)-th entries, if defined, are either all loop entries or all statement entries.
2. The last component of every iteration vector is a statement entry.

The definition of iteration vector is consistent with that for perfect loop nests.

In the case of an \( n \)-deep perfect loop nest, the iteration vector for the \( i \)-th statement \( S_i \) in the loop body has the form \((x_1, \cdots, x_n, i)\), where the first \( n \) entries are loop entries. If all statements inside the innermost loop are atomic, each iteration of the loop body can be identified by the iteration vector \((x_1, \cdots, x_n)\), as previously.

**2.2 Statement Space of a Statement**

An iteration vector contains both loop and statement entries. Let \( \text{loop}(x) \) be a boolean function such that \( \text{loop}(x) \) is true if and only if \( x \) is a loop entry.

The statement space of a statement \( S \) is defined as follows:

\[
I_S = \{ (x_1, \cdots, x_S) \mid \forall i : \text{loop}(x_i) \implies L_i \leq x_i \leq U_i \}
\]

- \((x_1, \cdots, x_S)\) is the iteration vector of \( S \), and
- "DO \( x_i = L_i, U_i \)" is the \( i \)-th enclosing loop of \( S \).

The following procedure, when called with arguments L, ( ) and \{ \}, will return the statement space \( \{ V \mid P \} \) for each of the statements in a loop nest L.

**Algorithm 2** Procedure to compute the statement spaces from a loop nest.

\[
\text{Iteration\_Space}(L, V, P)
\]

CASE L OF

\[
\text{DO } x_{k+1} = L_{k+1}, U_{k+1} \mid L';
\]

\[
\text{Iteration\_Space}(L', V\#(x_{k+1}), P \wedge (L_{k+1} \leq x_{k+1} \leq U_{k+1}))
\]

\[
L_1 \cdots L_m;
\]

FOR \( h = 1, m \) DO Iteration\_Space \((L_h, V\#(h), P)\)

Atomic Statement \( S \) : /* the base case */

OUTPUT \( I_S : \{ V \mid P \} \)
The statement spaces for Example 1 are computed as follows:

<table>
<thead>
<tr>
<th>(a) A Perfect Loop Nest</th>
<th>(b) An Imperfect Loop Nest</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_{S_1} = {(i,1) \mid 1 \leq i \leq n}$</td>
<td>$I_{S_1} = {(i,1) \mid 1 \leq i \leq n}$</td>
</tr>
<tr>
<td>$I_{S_2} = {(i,2) \mid 1 \leq i \leq n}$</td>
<td>$I_{S_2} = {(i,2,j,1) \mid 1 \leq i \leq n, 1 \leq j \leq i}$</td>
</tr>
</tbody>
</table>

By definition, a statement space is always a convex polyhedron.

2.3. Iteration Space of a Loop Nest

The iteration space for a general loop nest is the union of all statement spaces in the loop nest. Note that different statement spaces may have different dimensions. This implies that the iteration vectors in an iteration space may have incompatible lengths. In addition, an iteration space is not a convex polyhedron, in general.

2.4. Lexicographic Order

The lexicographic order “less than” $<$ on two iteration vectors $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_m)$ is defined inductively as follows:

$n = 1 \land m = 1$: $(x_1, \ldots, x_n) < (y_1, \ldots, y_m) \iff x_1 < y_1$

$n > 1 \land m > 1$: $(x_1, \ldots, x_n) < (y_1, \ldots, y_m) \iff \begin{cases} x_1 < y_1, \text{ or} \\ x_1 = y_1 \land (x_2, \ldots, x_n) < (y_2, \ldots, y_m) \end{cases}$

According to the definition of iteration vector, there do not exist two iteration vectors such that one is the prefix of the other. Therefore, $<$ is well-defined.

Hence, we draw the following conclusion: In a sequential execution, a loop nest is executed according to the lexicographic order of its iteration vectors.

Up to now, we have completed our task of generalising iteration space and some associated concepts in the unimodular approach to characterise general loop nests.

3. Preorder Tree

We have just described a simple procedure to compute the iteration space from a loop nest. In practice, a restructuring compiler must also solve the reverse problem efficiently. This problem, known as the code generation problem, is to construct a loop nest to iterate over an arbitrary iteration space lexicographically. To facilitate the code generation process, the concept of preorder tree is introduced next.

A preorder tree for a loop nest is a general (ordered) tree, where

1. The (dummy) root has no associated value. It is introduced to ensure that every loop nest can be represented as a tree. For example, the preorder tree in Figure 4 would degenerate into a forest if the dummy root were deleted.
2. Each internal node is labeled with a DO loop of the loop nest.
3. Each leaf node is labeled with a statement of the loop nest.
4. The parent of a node is the loop that most closely encloses the loop or statement at that node.
5. The children (siblings) of a node are the loops or statements that, if listed from left to right, appear in their lexical order.

The preorder trees for Example 1 are depicted in Figure 1.
By the definition of preorder tree, we have the following theorem:

**Theorem 1** Given a preorder tree for a loop nest, this loop nest can be reconstructed by a preorder traversal of the preorder tree.

The following procedure, when called with arguments L and the dummy root T, will create the preorder tree for a loop nest L.

**Algorithm 3** Procedure to build the preorder tree from a loop nest.

```
PreorderTree (L, T)
CASE L OF
  DO x_{k+1} = \ldots \ L':
    Create a node T' labeled with this loop, and link it as T's rightmost child
    PreorderTree(L', T')
  L_{k-1}: L_m;
  FOR k = 1, m DO PreorderTree (L_k, T)
Atomic Statement S / * the base case */
    Create a node labeled with S and link it as T's rightmost child
```

For completeness, a procedure is given next which, when called with arguments T, ( ) and { }, returns the statement spaces for all statements in the preorder tree T. Here, T is a pointer pointing to the dummy root of the preorder tree.

**Algorithm 4** Procedure to compute the statement spaces from a preorder tree.

```
IterationSpace2 (T, V, P)
  WHILE there is an unvisited child of T DO
    / * visit T's children from left to right */
    Let T' be T's leftmost unvisited child, which is T's i-th child from the left
    IF T' is a leaf node THEN / * the base case */
      Let S be the statement associated with T'
      OUTPUT IS : \{ V#(i) | P \}
    ELSE
      Let “DO x = L, U” be the loop associated with T'
      IF T' is T's single child THEN / * i = 1 */
        IterationSpace2 (T', V#(i, x), P \wedge (L \leq x \leq U))
      ELSE
        IterationSpace2 (T', V#(i, x), P \wedge (L \leq x \leq U))
```

We conclude this section by summarising the results presented so far in Figure 2. The dashed arrow indicates the missing link from “iteration space” to “preorder tree”; the development of the code generation algorithm is the topic of Section 5.
4. Unimodular Transformations

The iteration space of a loop nest is a union of statement spaces, where each statement space is a convex polyhedron. Loop transformations are cast as unimodular transformations from statement spaces to statement spaces. Such a framework allows a restructuring compiler to apply several transformations at one step, each on distinct statement space.

Let $x$ be the iteration vector of a statement $S$. The notation $x_L$ denotes the vector obtained from the $x$ with all its statement entries removed. Then the statement space of a statement $S$ can be specified as a convex polyhedron:

$$ I_S = \{ x \mid Ax_L \leq b \} $$

For example, the statement space of $S_1$ in Example 1(b) can be put into the form:

$$ I_{S_1} = \{(i,1) \mid 1 \leq i \leq n\} = \{(i,1) \mid \begin{bmatrix} -1 \\ 1 \end{bmatrix} \begin{bmatrix} i \\ 1 \end{bmatrix} \leq \begin{bmatrix} -1 \\ n \end{bmatrix} \} \] $$

where $x = (i,1)$ and $x_L = (i)$. To ensure that the statement space of $S$ remains to be convex in the restructured code, we require that $S$ be enclosed by the same number of loops before and after the loop nest is restructured. Let $y$ ($I'_S$) be the iteration vector (statement space) of $S$ in the restructured code. We use two separate mappings to transform $I_S$ to $I'_S$: one defines the iteration vector $y$ (or essentially, the values of statement entries in $y$), and the other is an affine transformation ($T_S, \alpha_S$), where $T_S$ is unimodular and $\alpha_S$ is an additive constant vector, such that $T_Sx_L + \alpha_S = y_L$. The transformed statement space $I'_S$ is a convex polyhedron:

$$ I'_S = \{ y \mid AT_S^{-1}(y_L - \alpha_S) \leq b \} $$

When $\alpha_S = 0$, the unimodular transformation $T_S$ describes all combinations of loop interchange, skewing and reversal. By considering $S$ and its surrounding loops in isolation as a perfect loop nest, we are essentially transforming this perfect loop nest using the affine transformation ($T_S, \alpha_S$). We must replace the old loop variables $x_L$ in the array subscripts of statement $S$ by the new ones $y_L$ — a simple process because $T_Sx_L + \alpha_S = y_L$, where $T_S$ is invertible.

As an example, consider the problem of restructuring the imperfect loop nest in Example 1(b). The statement spaces $I_{S_1}$ and $I_{S_2}$ are listed in Section 2.2:

- Transform $I_{S_1}$ by mapping $(i,1)$ to $(1,i')$, where
  $$ T_{S_1} = \begin{bmatrix} 1 \end{bmatrix}, \quad T_{S_1}[i] = [i'] $$

- Transform $I_{S_2}$ by mapping $(i,2,j,1)$ to $(2,i',j',1)$, where
  $$ T_{S_2} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad T_{S_2} \begin{bmatrix} i \\ j \end{bmatrix} = \begin{bmatrix} i' \\ j' \end{bmatrix} $$

These two transformed spaces map the original statement spaces $I_{S_1}$ and $I_{S_2}$ to:

$$ I'_{S_1} = \{(1,i') \mid 1 \leq i' \leq n\} $$

$$ I'_{S_2} = \{(2,i',j',1) \mid 1 \leq i' \leq j', 1 \leq j' \leq n\} $$

(1)
Note that $T_{S_1}$ and $T_{S_2}$ transform the outer loop $i$ of the program differently.

Using our code generation method to be presented shortly, we obtain the following code that iterates over the transformed iteration space $I'_{S_1} \cup I'_{S_2}$ lexicographically:

\begin{align*}
\text{DO } i' = 1, n \quad \text{/* parallelisable */} \\
\quad b(i') = 0 \\
\text{DO } i' = 1, n \\
\quad \text{DO } j' = i', n \quad \text{/* parallelisable */} \\
\quad b(i) = b(i') + a(j', i')
\end{align*}

By analysing the dependencies in the program, a restructuring compiler can verify that the two transformations we have applied are legal because they preserve the dependencies in the original program, and in addition, that both the first $i'$ loop and the $j'$ loop can be run in parallel.

As another example, consider the imperfect loop nest:

\begin{align*}
&L_1 \quad \text{DO } i = 1, n \\
&L_{2,1} \quad \text{DO } j = 1, n \\
&S_1 \quad a(i, j) = a(i - 1, j) + a(i, j - 1) \times b(i - 1) \\
&L_{2,2} \quad \text{DO } k = 1, i \\
&S_2 \quad b(i) = b(i) + a(k, i - k + 1)
\end{align*}

There are two statement spaces:

\begin{align*}
&I_{S_1}: \{(i, 1, j, 1) \mid 1 \leq i, j \leq n\} \\
&I_{S_2}: \{(i, 2, k, 1) \mid 1 \leq i \leq n, 1 \leq k \leq i\}
\end{align*}

Figure 3(b) shows the iteration space $I_{S_1} \cup I_{S_2}$, which are annotated with the data dependencies of the program summarised below:

- The dependence from $a(i, j)$ to $a(i - 1, j)$ has distance vector $(1, 0)$.
- The dependence from $a(i, j)$ to $a(i, j - 1)$ has distance vector $(0, 1)$.
- The dependence from $a(i, j)$ to $a(k, i - k + 1)$ has direction vector $(\leq)$.
- The (value-based) dependence from the LHS $b(i)$ of $S_2$ to the RHS $b(i)$ of the same statement has distance vector $(0, 1)$.
• The dependence from the LHS \( b(i) \) of \( S_2 \) to \( b(i-1) \) has distance vector \( (1) \). Because there is a cycle of data dependences between \( S_1 \) and \( S_2 \), it is illegal to use loop distribution to convert the program into perfectly nested loops. This means that the existing unimodular approach cannot be used.

In our framework, we may restructure the loop nest as follows (Figure 3(b)):

• Applying a loop skewing followed by a loop interchange, which is modeled by the following unimodular transformation:

\[
T_{S_1} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}, \quad T_{S_1} \left[ \begin{array}{c} i \\ j \end{array} \right] = \left[ \begin{array}{c} i' \\ j' \end{array} \right]
\]

to the \( i \) and \( j \) loops transforms \( I_{S_1} \) to:

\[
I'_{S_1} = \{(i', 1, j', 1) \mid 1 \leq i' - j' \leq n, 1 \leq j' \leq n\}
\]

• Applying the index shifting transformation

\[
T_{S_2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad T_{S_2} \left[ \begin{array}{c} i \\ j \end{array} \right] = \left[ \begin{array}{c} i' \\ j' \end{array} \right]
\]

to the \( i \) and \( k \) loops transforms \( I_{S_2} \) to:

\[
I'_{S_2} = \{(i', 2, k', 1) \mid 2 \leq i' \leq n + 1, 1 \leq k' \leq n\}
\]

Our code generation method will generate the following code to scan \( I'_{S_1} \cup I'_{S_2} \):

DO \( i' = 2, n + 1 \)
DO \( j' = \text{max}(1, i' - n), \min(i' - 1, n) \) /* parallelisable */
\[
a(i' - j', j') = a(i' - j' - 1, j') + a(i' - j', j' - 1) \ast b(i' - j' - 1)
\]

DO \( k' = 1, j' \)
\[
b(i') = b(i') + a(k', i' - k' + 1)
\]

DO \( i' = n + 2, 2n \)
DO \( j' = i' - n, n \) /* parallelisable */
\[
a(i' - j', j') = a(i' - j' - 1, j') + a(i' - j', j' - 1) \ast b(i' - j' - 1)
\]

The two transformations we have applied are legal, because the five dependences in the original program are transformed into \((1,0), (1,1), (=), (0,1)\) and \((<)\), which are all lexicographically positive. The two \( j' \) loops can be executed in parallel because they do not carry any dependencies.

We are currently investigating how to systematically choose a combination of multiple transformations to exploit collectively the available parallelism in a loop nest. We are also studying the required dependence test for legality.

5. Code Generation

Let \( S_1, \ldots, S_m \) be all statements in a loop nest. Let unimodular transformations be applied individually to their statement spaces \( I_{S_1}, \ldots, I_{S_m} \); the transformed statement spaces \( I'_{S_1}, \ldots, I'_{S_m} \) are all convex polyhedra. In the transformed iteration space \( \bigcup \bigcup I'_{S_i} \), the iterations are executed lexicographically with respect to that space. This section describes a code generation method that produces code to iterate over such a space in lexicographic order. For example, with the transformed iteration space \( I'_{S_1} \cup I'_{S_2} \) in (1) as input, our method generates the code (2).

Like a statement space, a transformed statement space is specified by a conjunction of affine constraints on the loop variables surrounding the statement. Unlike a statement space, in a transformed statement space it is possible that a loop variable is not explicitly bounded from below and/or above by any existing constraints. Take
the $I_{S_2}$ in (1) for example: $1 \leq i' \leq j'$ consists of a lower bound of $i'$: $1 \leq i'$, and a lower bound of $j'$: $i' \leq j'$, while $1 \leq j' \leq n$ represents both a lower and upper bound for $j'$. This means that neither of the four constraints is an upper bound of $i'$. A transformed statement space is \textit{explicitly defined} if every loop variable has at least one lower bound constraint and one upper bound constraint. Thus, the $I_{S_2}$ is not explicitly defined. To make a transformed statement space explicitly defined, we can simply add to it the missing bounds using either the Fourier-Motzkin elimination \cite{fourier} or Feautrier’s parametric integer programming (PIP) \cite{feautrier}. For example, adding $i' \leq n$ to the $I_{S_2}$ will make it explicitly defined. \textit{Note that the $I_{S_2}$ before and after $i' \leq n$ is added represent the same convex polyhedron because all missing bounds introduced this way are redundant with respect to the original set of constraints.}

To construct a loop nest to scan an iteration space, our code generation procedure $\text{Code\_Gen}(I)$ given below proceeds in three steps:

\begin{algorithm}
\caption{Code generation algorithm.}
\begin{algorithmic}
\State $\text{Code\_Gen}(I)$
\Comment{$I$ is an iteration space given as a set of statement spaces $*$/}
\State $\text{Explicitly\_Defined}(I)$
\State \textbf{Step 1:} $\text{Explicitly\_Defined}(I)$
\State \textbf{Step 2:} (a) Create a tree with \textit{Root} and $T$ set appropriately as follows:
\State \hspace{1cm} $\text{Root}$
\State \hspace{1cm} $T$
\State \hspace{1cm} (b) $\text{Tree\_Gen}(I, T, 1)$
\State \textbf{Step 3:} $\text{Loop\_Gen}(\text{Root})$
\end{algorithmic}
\end{algorithm}

\textbf{Step 1:} Procedure $\text{Explicitly\_Defined}(I)$ examines every statement space in the iteration space $I$ and introduces extra constraints, if necessary, to ensure that every statement space is explicitly defined. This step ensures that the bounds of all loop variables are well-defined (see \textbf{Case 1} in the following page).

\textbf{Step 2:} Starting with the initial tree in (a), procedure $\text{Tree\_Gen}(I, T, 1)$, called in (b), constructs a preorder tree for the iteration space $I$, starting from the first entry of an iteration vector. Hence, the “1” in the parameter list.

\textbf{Step 3:} With the preorder tree obtained in Step 2, $\text{Loop\_Gen}(\text{Root})$, written based on Theorem 1, does a preorder traversal on it to generate the loop nest.

Let us consider how to generate code for the iteration space $I_{S_1} \cup I_{S_2}$, where

$I_{S_1} = \{(i,1) | 1 \leq i \leq 10\}$

$I_{S_2} = \{(i,2,j,1) | 1 \leq j \leq i, i + j \leq 6\}$

The $S_2$’s statement space is not explicitly defined, because all constraints are loop bounds of $j$. Using the Fourier-Motzkin elimination, we obtain the loop bounds of $i$: $1 \leq i \leq 5$. So in the first step, $\text{Explicitly\_Defined}$ will modify $I_{S_2}$ to:

$I_{S_2} = \{(i,2,j,1) | 1 \leq i \leq 5, 1 \leq j \leq i, i + j \leq 6\}$

In the second step, $\text{Tree\_Gen}$ takes this modified iteration space and produces the preorder tree as depicted in Figure 4. Finally, $\text{Loop\_Gen}$ applies a preorder traversal to this preorder tree to yield the loop nest shown on the right.

$\text{Tree\_Gen}$ is given in Algorithm 6. Figure 5 illustrates the use of this procedure in constructing the preorder tree in Figure 4.
Given an iteration space, we need to generate a preorder tree whose associated loop nest iterates over its points in their lexicographic order. For a statement at a leaf node, all its enclosing loops can be found at its ancestor nodes (excluding the root), with the outermost loop associated with the ancestor the highest in the tree, and the next to outermost loop associated with the ancestor the second highest in the tree, and so on. This produces the desired result in such a way that the points in the iteration space are executed in lexicographic order.

Tree\_Gen(I, T, level) builds a preorder tree recursively, level by level, starting from the first level. We shall use (x_1, x_2, \ldots) to represent the iteration vector of a statement, where the subscripts indicate the levels of the corresponding entries, with the first entry at level 1. There are two cases to consider:

**Case 1:** x_{level} is a loop variable entry (lines 10–39). The objective is to extract from the iteration space I the loop bounds of x_{level}. Each statement space in I, which is initially explicitly defined (Step 2 of Algorithm 5), contains at least one lower bound and one upper bound of x_{level}; all such loop bounds of x_{level} define the loop limits for which the statement S should be executed along the dimension x_{level}. In general, the loop limits for different statements do not match exactly. Hence, it is impossible to generate one single x_{level} loop to execute all statements. The basic idea here is to partition the union of the loop limits for all statements so that with each partition, each statement is either executed at every iteration within the partition or not executed at all. Instead of generating a single x_{level} loop, we generate a sequence of x_{level} loops, one for each partition.

We generate these partitions recursively as follows. We choose arbitrarily a loop bound of x_{level}, B, in I. Let \{L, U\} be a permutation of \{B, -B\} such that L is the lower bound of x_{level}. We split I into two disjoint regions: one in the half space L, denoted I_L and the other in the opposite half space U, denoted I_U. It is possible that some statement spaces in I are contained entirely in one of the two half spaces, in which case, the intersection of them with the opposite half space are empty. In the extreme case, all statement spaces in I can be contained entirely in one half space (i.e., B), which implies that the other region is empty. In the case when a statement space is contained partially in both regions, some of its specifying constraints can become redundant. All these cases will be detected in line 22. Since either I_L or I_U may be empty, we distinguish four cases in lines 23–36. Note that both regions are empty only if the iteration space I in Step 2 of Algorithm 5 is empty. In the case when both regions are nonempty (lines 32–36), we split T into two nodes T_U and T_L and copy the loop bounds of x_{level} from T to these two nodes.
We add the new upper bound \( U \) to \( T_U \) and the new lower bound \( L \) to \( T_L \). We place \( T_U \) to the left of \( T_L \), because \( I_U \) should be executed before \( I_L \). Finally, we call \texttt{TreeGen}(I_U, T_U, level) and \texttt{TreeGen}(I_L, T_L, level) to repeat recursively the same splitting process at the same level on both regions \( I_U \) and \( I_L \).

If there are no loop bounds of \( x_{\text{level}} \) left in the iteration space \( I \), we create a child node \( T_C \) for \( T \) and repeat the same splitting process at level+1 (lines 38–39).

If the iteration space \( I \) contains a single statement space, it will be less efficient to run through lines 15–39. Instead, in lines 11–14, we transfer the loop bounds of \( x_{\text{level}} \) from \( I \) to the node \( T \), create a child node \( T_C \) for \( T \) and invoke \texttt{TreeGen}(I, T, level+1).

\begin{algorithm}
\textbf{Algorithm 6} Procedure to construct a \textit{pre-order} tree from an iteration space.
\begin{algorithmic}[1]
\State \textbf{TreeGen} \( (I, T, \text{level}) \)
\If {\( x_{\text{level}} \) is a statement entry} \Then
\State Let \( I_1, \ldots, I_m \) be partitions of the statement spaces in \( I \) such that \( x_{\text{level}} = i \) in \( I_i \).
\For {\( i = 1, m \)} \Do
\If {\( I_i \) has one statement space and \( x_{\text{level}} \) is the last iteration vector entry} \Then
\State Label \( T_i \) with the statement for \( I_i \). /* a base case; \( T_i \) is a leaf node */
\Else
\State \texttt{TreeGen}(\( I_i, T_i, \text{level} + 1 \))
\EndIf
\EndIf
\EndFor
\Else /* \( x_{\text{level}} \) is a loop entry */ \EndIf
\Else /* \( I \) is a single statement space */ \EndIf
\State Move into \( T \) from \( I \) all loop bounds of loop variable \( x_{\text{level}} \).
\State Create a child node \( T_C \) for \( T \).
\State \texttt{TreeGen}(\( I, T_C, \text{level} + 1 \))
\Else /* \( I \) consists of more than one statement space */ \EndIf
\State Let \( C \) be the polyhedron defined by the loop bounds from \( T \) and its ancestors.
\If {there is a loop bound of \( x_{\text{level}} \) in \( I \) \Then}
\State Remove from \( I \) arbitrarily a loop bound of \( x_{\text{level}} \), say, \( B \)
\State Let \( \{L, U\} \) be a permutation of \( \{B, -B\} \) such that \( L \) is a lower bound of \( x_{\text{level}} \).
\State Let \( I_L \) be the intersection of \( I \) with the half-space \( L \).
\State Let \( I_U \) be the intersection of \( I \) with the half-space \( U \).
\State Eliminate all redundant constraints in \( I_L \) and \( I_U \) with respect to \( C \).
\If {\( I_U = \emptyset \) and \( I_U = \emptyset \) \Then}
\State Delete \( T \) /* a base case; reached only if \( I = \emptyset \) initially */
\ElseIf {\( I_L = \emptyset \) \Then}
\State Merge \( U \) with the loop bounds of \( T \).
\State \texttt{TreeGen}(\( I_U, T_L, \text{level} \))
\ElseIf {\( I_U = \emptyset \) \Then}
\State Merge \( L \) with the loop bounds of \( T \).
\State \texttt{TreeGen}(\( I_L, T_U, \text{level} \))
\Else /* \( I_L \) and \( I_U \) are not empty */ \EndIf
\State Split \( T \) into two nodes \( T_L \) and \( T_U \) such that \( T_U \) is to the left of \( T_L \).
\State Copy the loop bounds of \( x_{\text{level}} \) from \( T \) to \( T_L \) and \( T_U \).
\State Merge \( L \) (\( U \)) with the loop bounds of \( T_L \) (\( T_U \)).
\State \texttt{TreeGen}(\( I_I, T_U, \text{level} \))
\State \texttt{TreeGen}(\( I_L, T_L, \text{level} \))
\EndIf
\Else /* \( I_L \) and \( I_U \) are not empty */ \EndIf
\State Create a child node \( T_C \) for \( T \).
\State \texttt{TreeGen}(\( I, T_C, \text{level} + 1 \))
\EndIf
\EndIf
\EndAlgorithm}

11
Figure 5: Trace of $\text{TreeGen}(I, T; \text{level})$ in building a preorder tree. In each call (except (d) and (f)), $T$ is the node marked with $\Delta$; $I$ is the union of the statement spaces below $T$; $\text{level}$ is the actual value passed from the previous call. $T$ in (d) is the $T$ in (c). $T$ in (f) is the $T_C$ in (e).
Case 2: $x_{level}$ is a statement entry (lines 2 – 9). The key observation is that the iterations in $I_k$ should be executed before those in $I_{k+1}$.

6. Conclusion

In this paper, we extended the unimodular approach to provide a framework that allows multiple unimodular transformations to be used to restructure imperfect loop nests. The concepts previously used for perfect loop nests, such as iteration vector, iteration space and lexicographic order were generalised to characterise imperfect loop nests. Preorder trees were introduced as an intermediate representation to facilitate the generation of the transformed code. A code generation method was given that generates code for an iteration space by scanning a union of sets of affine constraints. This method extends the existing ones, such as the Fourier-Motzkin elimination and Feautrier's PIP, while retaining the philosophy of generating code by scanning affine constraints.

7. Acknowledgements

Thanks to referee 1 for his/her many constructive comments. Thanks to referee 3 for his/her criticism, which nevertheless helps to improve this paper in presentation. This work is supported by an Australian Research Council Grant A49600987.

References