UNIVERSITÄT PASSAU
Fakultät für Mathematik und Informatik

# Automatic Code Generation in the Polytope Model

Diplomarbeit

Autor:
Sabine Wetzel

Aufgabenstellung:
Prof. Christian Lengauer
Lehrstuhl für Informatik
Universität Passau

Betreuer:
Martin Griebl

Passau, 8. November 1995

**Eidesstattliche Erklärung**

Hiermit erkläre ich eidesstattlich, daß ich diese Diplomarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe, und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind. Die Diplomarbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt.

Passau, 8. November 1995

**Abstract**

In recent years, the goal of parallelizing sequential programs has been a focus of research. One possible automatic method is based on the polytope model. In this method, the conversion proceeds in three steps: modelling the loop nest by a polytope, applying a coordinate transformation and retransfering the new polytope into the parallel program. In this diploma thesis, the center of interest is the third step, the code generation. We propose code generation techniques for two possible transformations: one provided by the Lamport scheduler which deals with perfectly nested programs and iteration-based transformations leading to unimodular transformation matrices and the other provided by the Feautrier scheduler which can deal with imperfectly nested loops and piecewise affine, statement-based transformations. Therefore, we extend the original iteration-based polytope model and present several algorithms for this general and complex method of code generation.

**Acknowledgements**

I would like to thank the following people at the University of Passau:
Martin Griebl for his extensive and substantial support. He gave many valuable suggestions on the development of the algorithms.
Prof. Christian Lengauer read this thesis more than once, found many mistakes and gave detailed suggestions. Particulary, I have to stress his various comments on the English version.
All members of the LooPo-Team as there are: Nils Ellmenreich, who specially improved my English, Peter Faber, Max Geigl, Robert Günz, Harald Keimer, Klaus Köbler, Radko Kubias, Frank Schuler, Christian Wieninger, and Alexander Wüst.
Finally, I want to thank Jean-François Collard for many discussions on this thesis.

# Contents

# 1 Introduction

In recent years, a lot of research effort has been invested in the automatic parallelization of sequential programs since both writing parallel programs from scratch and converting sequential into parallel programs by hand is cumbersome and error-prone.

One possible way of automating parallelization is based on a mathematical model: the polytope model [1, 12]. In this framework, the conversion is done in three steps. First, a loop nest is modelled as a geometric object, a polytope. Second, a coordinate transformation is applied to this polytope. The result is a target polytope containing all information for the parallel program, which is derived in the third step.

For reasons of flexibility, one should be able to obtain individual transformations for different statements in the program [5, 6, 16]. Therefore, we use a refined model where the polytope model is at statement-level. Some problems appear in the third step when generating the target program from many polytopes. This will be the focus of this thesis. Up to now, only little work has been done on this subject [11].

Chapter 2 presents our theoretical basis and some notational conventions. Chapter 3 describes the code generation for Lamport transformations which uses the original polytope model. In Chapter 4, the code generation for Feautrier transformations is explained which is based on the refined polytope model. The last chapter concludes with some possible improvements and some remarks about the implementation.

# 2 Definitions and Notations

## 2.1 Matrices

This thesis assumes a familiarity of the reader with the basic concepts of linear algebra like linear dependence and independence, rank, determinant and inverse.

**Definition 2.1.1 (denominator)**
Let $A'$ be a matrix of $\mathbb{Q}^{m\times n}$. A value $d \in \mathbb{Z}$ is called *a denominator of* $A'$, iff the following is valid:

$$\exists\ A \in \mathbb{Z}^{m\times n} : A' = \tfrac{1}{d} \cdot A$$

If we talk about *the* denominator, it is always the smallest possible integer value which satisfies this equality.

In the this section, we prove two theorems related to the denominator of a matrix which are important for the theory and the implementation. Then, we define some terms and state some theorems of linear algebra taken from [7, 14, 17].
Let $A'$ be a square matrix of $\mathbb{Q}^{n\times n}$ with $A' = \frac{1}{d} \cdot A$, where $A \in \mathbb{Z}^{n\times n}$ and $d \in \mathbb{Z}$ is the denominator of $A'$.

**Theorem 2.1.2**
The determinant of $A'$ is defined by:

$$\det(A') = \left(\tfrac{1}{d}\right)^n \det(A)$$

Proof:
See any linear algebra book.

**Theorem 2.1.3**
The inverse of the square matrix $A'$ can be computed by:

$$(A')^{-1} = d \cdot A^{-1}$$

Proof:

$$\begin{aligned}(A')^{-1} &= I \cdot (A')^{-1} \\ &= A^{-1} \cdot A \cdot (A')^{-1} \\ &= d \cdot A^{-1} \cdot \tfrac{1}{d} \cdot A \cdot (A')^{-1} \\ &= d \cdot A^{-1} \cdot A' \cdot (A')^{-1} \\ &= d \cdot A^{-1}\end{aligned}$$

where $I$ is the $(n \times n)$-dimensional identity matrix.

**Definition 2.1.4 (unimodular)**
A square matrix is *unimodular* iff it is integral and has determinant 1 or $-1$.

**Definition 2.1.5 (invertible, regular, singular)**
A matrix is *invertible* (or *regular*) iff it is square and of full rank. A matrix which is not invertible is called *singular*.

**Remark 2.1.6**
The inverse of a unimodular matrix is unimodular. The inverse of a non-unimodular invertible matrix is not integral.

**Definition 2.1.7 (Hermite normal form)**
A matrix $B$ of $\mathbb{Q}^{m \times n}$ is in *Hermite normal form* (HNF) if it consists of an arbitrary number of consecutive zero columns and has the following properties:

- $B$ is not singular,
- $B$ is lower triangular,
- all entries of $B$ are non-negative,
- the maximum of every row is placed on the diagonal.

**Theorem 2.1.8**
Let $A$ be a matrix of the $\mathbb{Q}^{m \times n}$ with full row rank. Then, the following holds:

- $A$ can be brought into HNF with a finite number of elementary column operations.
- $A$ has a unique HNF.

- If $A$ is unimodular, its HNF is the identity matrix.
- There exists an algorithm which determines the HNF of $A$ in polynomial time.

Proof: [14].

**Remark 2.1.9**
Generally, the matrices treated in this thesis have less than 5 rows and columns. Thus, an algorithm for the determination of the HNF was implemented [2] which is easier to understand but slower than the algorithm of Schrijver [14].

## 2.2 Polytopes

In this section, we define some general concepts of the theory of polyhedra and give some conventions for notation. Most of the definitions are taken from [14].

**Definition 2.2.1 (halfspace)**
Let $a_1, ..., a_n \in \mathbb{R}, b \in \mathbb{R}$.
Then, a *halfspace* $H$ of $\mathbb{R}^n$ is defined by the following set:

$$H = \{(x_1, ..., x_n) \in \mathbb{R}^n \mid a_1 x_1 + ... + a_n x_n \leq b\}$$

**Definition 2.2.2 (convex)**
Let $\alpha \in [0, 1]$. The set $A$ is called *convex* iff:

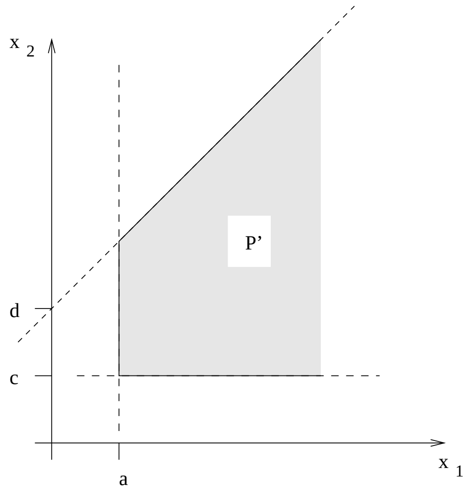
$$\forall\ x, y \in A : (\alpha x + (1 - \alpha) y) \in A$$

Thus, given two points $x$ and $y$ of $A$, every point on the line between $x$ and $y$ is an element of $A$.

**Remark 2.2.3**
Every halfspace and every intersection of a finite number of halfspaces is convex.

**Definition 2.2.4 (polyhedron)**
A *polyhedron* in $\mathbb{R}^n$ is the intersection of a finite number of halfspaces in $\mathbb{R}^n$.

An example of a polyhedron and the corresponding halfspaces is depicted in Figure 1.



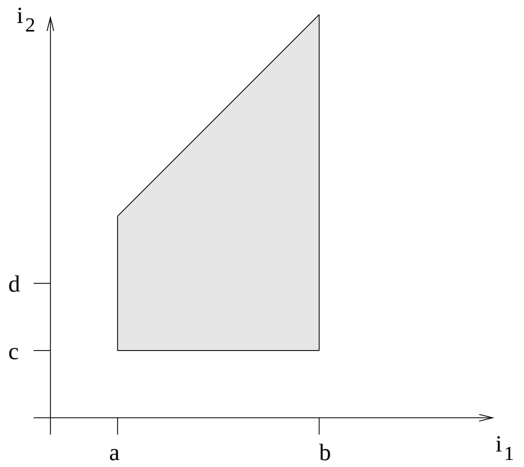

$$P': \begin{array}{l} x_1 \geq a \\ x_2 \geq c \\ x_2 \leq x_1 + d \end{array}$$

Figure 1: Polyhedron

**Definition 2.2.5 (polytope)**
A *polytope* is a bounded polyhedron.
See the example in Figure 2.

**Remark 2.2.6**
A polytope in $\mathbb{R}^n$ can also be defined as the convex hull of a finite number of points in $\mathbb{R}^n$.

**Definition 2.2.7 (quasi-convex)**
A set is said to be a *quasi-convex polytope* iff it is the union of a finite number of polytopes [17].
An example is shown in Figure 3.




$$P: \begin{array}{l} x_1 \geq a \\ x_1 \leq b \\ x_2 \geq c \\ x_2 \leq x_1 + d \end{array}$$

Figure 2: Polytope

**Definition 2.2.8 (lattice)**
Let $A$ be a matrix of $\mathbb{Z}^{m \times n}$. The set

$$\{y \mid y = A \cdot x \wedge x \in \mathbb{Z}^n\}$$

is called the *lattice* generated by the columns of $A$.

**Remark 2.2.9**
All points in the polytope which are interesting for our code generation are the intersection points of the polytope with a lattice, usually generated by the identity matrix. At these points, several actions have to take place. If we talk about a point in the polytope, we always consider these intersection points.

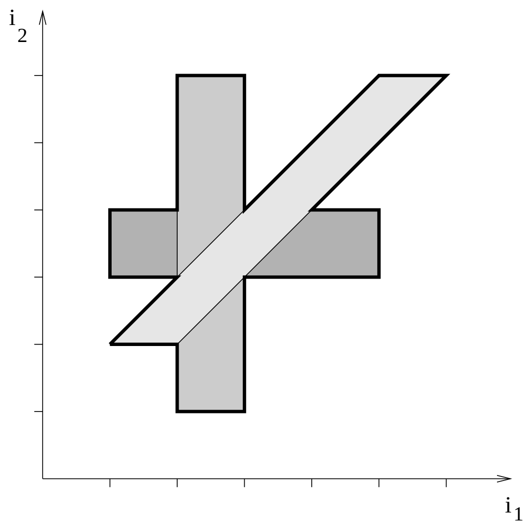



Figure 3: Quasi-convex polytope

**Notation 2.2.10**
In the following chapters, $P \in \mathbb{R}^n$ is a quasi-convex polytope which is the union of the polytopes $P_1, ..., P_k$, formally:

$$P = \bigcup_{\mu=1}^{k} P_\mu$$

## 2.3 Source and Target Program

In this section, we give a formal view of our source and target programs. First of all, some notational conventions are defined. Then, we explain the general form of our programs, which will be restricted for the source program at the end of this section.

**Notation**
A variable which is used as a loop index (Item 4c below) is called *loop variable*.

Loop variables are named $i_1, \ldots, i_\mu$ and $j_1, \ldots, j_\mu$ ($\mu \in \mathbb{N}$) in this thesis. *(Structure) parameters* are variables which are treated as constants. They are named $n_1, \ldots, n_\mu$.

**Program Format**
Our programs have the following features:

1. The basic data type is $\mathbb{Q}$. The only basic data structures are zero- or one- or multi-dimensional arrays. Array indices are affine functions in the loop variables and the structure parameters. Note that a product between loop indices and parameters is forbidden as array index.

2. Arrays are not allowed as loop indices. Loop variables cannot be initialized and modified outside a loop statement. Thus, the only variables appearing in the program are either loop variables or arrays.

3. Structure parameters must be positive or zero.

4. Only four kinds of statements are permitted in our programs: the skip statement, the assignment, the for loop and the if-then-else statement.

   (a) skip
   The skip statement is introduced to simplify the handling of user-defined dependences [10].

   (b) assignment
   As it is not allowed to write to a loop variable outside a loop statement, the left-hand side of an assignment is an array. The right-hand side of an assignment may contain functions in the loop variables, the structure parameters and the arrays.
   Examples of legal and illegal assignment statements are:

   | | |
   |---|---|
   | $a[i_1, i_2] = 3i_1 + 2n_1 + 5$ | legal |
   | $a[i_1, i_2] = n_1 \cdot\ i_1 + b[i_1]$ | legal |
   | $a[n_1 \cdot i_1] = 7$ | illegal |

   (c) for loop
   A loop can be either sequential or parallel. In both cases, it has the

same syntactical format except for the keyword: **for** (sequential) and **parfor** (parallel).
The $j$-th for loop of a program has, in general, a lower bound $l_j$, an upper bound $u_j$, a stride $s_j$, a loop body $B_j$, and a loop index which is a loop variable.
The loop bounds $l_j$ and $u_j$ may contain minimum or maximum expressions. Thus, the *upper bound* $u_j$ can be either

$$u_j = \min(u_{j1}, \ldots, u_{jr})$$

or

$$u_j = \max(u_{j1}, \ldots, u_{jr})$$

where $r \in \mathbb{N}$.
$u_{j\mu}$ ($\mu \in \{1, \ldots, r\}$) is an affine function in the loop variables $i_1, \ldots, i_{j-1}$ and the structure parameters. Thus, the bounds of the $j$-th loop can only depend on the outer loops. Note that we do not allow a product of loop variables and structure parameters in loop bounds.
Analogously, we define the *lower bound* $l_j$ as minimum or maximum expression of affine functions $l_{j\mu}$, but a non-unimodular transformation matrix will cause a more complicated offset for every affine function $l_{j\mu}$, called the *lower bound offset* $\delta_{j\mu}$ (Section 4.3.4). Thus, a lower bound has the following format:

$$l_j = \min(l_{j1} + \delta_{j1}, \ldots, l_{jr} + \delta_{jr})$$

or

$$l_j = \max(l_{j1} + \delta_{j1}, \ldots, l_{jr} + \delta_{jr})$$

The lower bound offsets $\delta_{j\mu}$ are affine functions in the loop indices and the parameters combined with a modulo operation % with an element of $\mathbb{Q}$, for example: $(j_1 - 1)\%2$.
The *stride* $s_j$ is an element of $\mathbb{Q}$.
The *loop body* $B_j$ may contain any sequence of statements or be empty. A program is called *perfectly nested* if the innermost loop body is empty or it contains an assignment or a skip statement and the loop bodies of all surrounding loops have no assignments

or skip statements. Otherwise, the program is said to be *imperfectly nested*. Examples of a perfectly and an imperfectly nested program are given in Figure 4.

perfectly nested

**for** $i_1 = 1$ **to** 10
  **for** $i_2 = i_1$ **to** $i_1 + 5$
    $a[i_1, i_2] = 0$
  **end**
**end**

imperfectly nested

**for** $i_1 = 1$ **to** 10
  $a[i_1] = 2$
  **for** $i_2 = 2$ **to** $2i_1 + 1$
    $b[i_1, i_2] = a[i_2] + 3$
  **end**
**end**

Figure 4: Example for (im-)perfectly nested loops

(d) if-then-else statement
The condition of an if-then-else statement is a boolean function. The branches may contain any sequence of statements. If the else-branch is empty, we omit it.

5. Note that there also exist assignments and if-then-else statements without surrounding loops.

6. The program can also consist of a parallel combination of programs. We introduce a binary parallel execution operator, denoted by $\|$. This operator combines two programs in parallel. The parallel execution operator may also be indexed by a set $M$ of elements of $\mathbb{N}\backslash\{0\}$. $\|_{y \in M} P'_y$ denotes the parallel execution of programs $P'_y$ which have indices in $M$.

**Constraints imposed on the source program**
The source program must satisfy additional restrictions:

1. No if-then-else statements are allowed.

2. Any stride $s_j$ is $\pm 1$; it is omitted if it is $+1$.
3. The lower bound offsets $\delta_{j\mu}$ are 0.
4. The lower bounds $l_j$ consist only of maximum expressions, the upper bounds $u_j$ only of minimum expressions.
5. There are only sequential for loops.
6. There is no parallel execution operator $\|$.

**Notation**
The set of all assignments and skip statements of a program $P'$ is called $\Omega_{P'}$.

## 2.4 Index Space

Every assignment or skip statement $S$ surrounded by one or more for loops has an *index space*, also called the *iteration space*. If no loops surround $S$, its index space is *empty*. Otherwise let $i_1, \ldots, i_k$ $(k \in \mathbb{N})$ be the loop variables of the loops surrounding $S$.
Then, the index space is defined by the following system of inequalities:

$$\begin{array}{l} l_1 \leq i_1 \leq u_1 \\ \vdots \\ l_k \leq i_k \leq u_k \end{array}$$

where $l_q$ and $u_q$ $(q \in \{1, \ldots, k\})$ are the lower and upper bound of the $q$-th surrounding loop.
We can normalize this system of inequalities:

$$\begin{array}{cccc} i_1 & - & l_1 & \geq 0 \\ -i_1 & + & u_1 & \geq 0 \\ \vdots & & & \\ i_k & - & l_k & \geq 0 \\ -i_k & + & u_k & \geq 0 \end{array}$$

Assume that $l_q$ is a maximum expression:

$$l_q = \max\{l_{q1}, \ldots, l_{qr}\} \ \ (r \in \mathbb{N})$$

Then, the inequality:

$$i_q \ - \ l_q \ \geq 0$$

represents the inequalities:

$$\begin{array}{lll} i_q & - & l_{q1} \ \geq 0 \\ \vdots & & \\ i_q & - & l_{qr} \ \geq 0 \end{array}$$

The same applies for a minimum expression for lower bounds and for the upper bounds.
As explained in Section 2.2, every system of inequalities can be represented by a polyhedron. In our programs, every variable $i_1, \ldots, i_k$ is bounded by its lower and upper bound and, therefore, a polytope is described by the system of inequalities. The polytope representing the index space of a statement will be the basis of our considerations in Chapter 4.
Let us consider an example:

**for** $i_1 = 1$ **to** $5$
    **for** $i_2 = i_1$ **to** $\min(i_1 + 5, 7)$
        $S$
    **end**
**end**

The index space of $S$ is defined by:

$$\begin{array}{rrrl} i_1 & & -1 & \geq 0 \\ -i_1 & & +5 & \geq 0 \\ -i_1 & +i_2 & & \geq 0 \\ i_1 & -i_2 & +5 & \geq 0 \\ & -i_2 & +7 & \geq 0 \end{array}$$

and is described by the polytope of Figure 5.

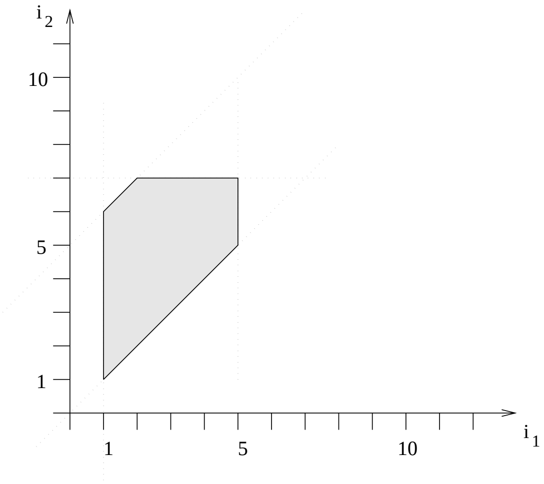



Figure 5: Index space of the example

Note that every statement $S$ of a program has its own index space, named $D_S$, and its own polytope, named $P_S$. We omit $S$ if the context is clear.
Every lattice point of the polytope $P_S$ constitutes an iteration of $S$.
Here, we have constructed a polytope for a statement $S$ from the loop bounds of the surrounding loops. Reversely, the loop bounds can be derived from the index space.

## 2.5 Schedule and Allocation

We define the concepts of a schedule and an allocation as in [16].
Let $S$ be a statement, $D_S$ the index space of statement $S$ and $\Omega_{P'}$ the set of all assignments in a program $P'$.

**Definition 2.5.1 (schedule)**
A *schedule* is a function

$$\Theta_S : \Omega_{P'} \times D_S \to \mathbb{Q}^k$$

which assigns a $k$-dimensional ($k \in \mathbb{N}$) execution time to every instance of a statement $S$ with respect to the index space $D_S$. We omit the statement if the context is clear.

**Definition 2.5.2 (allocation)**
A function

$$\Pi_S : \Omega_{P'} \times D_S \to \mathbb{Q}^{k'}$$

which assigns a processor of a $k'$-dimensional processor array ($k' \in \mathbb{N}$) to every instance of a statement $S$ is called an *allocation*.

**Remark 2.5.3**
Schedules and allocations are at least one-dimensional.

**Definition 2.5.4 (transformations, piecewise, elementary)**
We talk about *transformations* if we mean to cover both schedules and allocations. A transformation which is varying for several parts of a polytope is called *piecewise (affine) transformation*, for example:

$$\Theta_S(i_1, i_2) = \textbf{if } (i_1 > 3) \textbf{ then } i_2 \textbf{ else } i_1 + 2$$

Any other than a piecewise transformation is called *elementary*, for example, the two-dimensional schedule:

$$\Theta_S(i_1) = \begin{pmatrix} i_1 & +5n_1 & +3 \\ & & 1 \end{pmatrix}$$

**Remark 2.5.5**
The definition of the transformation differs from those of the common literature where a transformation consists of a schedule *and* an allocation. In this thesis, a transformation is either a multi-dimensional schedule or a multi-dimensional allocation.

**Definition 2.5.6 (transformation matrix)**
Let $\Theta_1, \ldots, \Theta_k$ be a $k$-dimensional elementary schedule of a statement $S$ and $\Pi_1, \ldots, \Pi_{k'}$ a $k'$-dimensional elementary allocation of the same statement, where $k, k' \in \mathbb{N}$.
Then, a *transformation matrix* is the matrix whose rows are defined by the coefficients of schedule and allocation. We name the transformation matrix $T$.

**Remark 2.5.7**
There is no restriction in the order of the transformations in the transformation matrix. Generally, the schedule occupies the upper rows if a synchronous program is the goal of the code generation. Otherwise, the allocation occupies the upper rows. We need to recall this order when constructing the target program. The reason for this choice of transformation matrices is given in Section 4.3.1.

**Remark 2.5.8**
The derivation of transformation matrices for piecewise transformations is discussed in Section 4.4.1.2.

**Example 2.5.9**
Let:

$$\begin{array}{lll} \Theta(i_1, i_2) = & 3i_1 & +i_2 \\ \Pi(i_1, i_2) = & & i_2 \end{array}$$

The corresponding transformation matrix $T$ for a synchronous target program is defined by:

$$T = \begin{pmatrix} 3 & 1 \\ 0 & 1 \end{pmatrix}$$

**Remark 2.5.10 (Relation between index points)**
With the help of the transformation matrix $T$ we can define a relation between an index point of the target index space $(j_1, \ldots, j_{k+k'})$ and an index point of the source index space $(i_1, \ldots, i_{k+k'})$ by:

$$\begin{pmatrix} j_1 \\ j_2 \\ \vdots \\ j_{k+k'} \end{pmatrix} = T \cdot \begin{pmatrix} i_1 \\ i_2 \\ \vdots \\ i_{k+k'} \end{pmatrix}$$

where $k$ is the dimensionality of the schedule and $k'$ the dimensionality of the allocation.

If the transformation matrix is invertible, we can determine the relationship between a source index point and a target index point by:

$$\begin{pmatrix} i_1 \\ i_2 \\ \vdots \\ i_{k+k'} \end{pmatrix} = T^{-1} \cdot \begin{pmatrix} j_1 \\ j_2 \\ \vdots \\ j_{k+k'} \end{pmatrix}$$

## 2.6 Homogeneous Coordinates

Homogeneous coordinates are used in computer graphics in order to describe or manipulate graphical objects. Originally, this method was applied in proofs to problems in $n$-space. These problems have a corresponding problem in $(n+1)$-space, where a solution can be found more easily. This solution is then projected back to $n$-space. We use homogeneous coordinates in order to transform affine problems into linear problems.

Example:
A point $(x, y, z) \in \mathbb{R}^3$ can be represented by the point $(wx, wy, wz, w) \in \mathbb{R}^4$, where $w \in \mathbb{R}\backslash\{0\}$ is called the *scale factor*. We can also project a 4-space point $(a, b, c, d)$ back to 3-space by dividing all components by the scale factor $d$: $(\frac{a}{d}, \frac{b}{d}, \frac{c}{d})$.

This transformation method can be generalized from 3- and 4-space to $n$- and $(n+1)$-space.

Throughout this thesis, we choose $w = 1$.

The advantage of this representation is that, beside scaling and rotation, also translation can be expressed as a matrix multiplication.

Example:
Let $(t_1, t_2, t_3)$ be the translation vector of a point $(x, y, z)$, thus:

$$\begin{pmatrix} x' \\ y' \\ z' \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} = \begin{pmatrix} x + t_1 \\ y + t_2 \\ z + t_3 \end{pmatrix}$$

can be replaced by the following matrix multiplication when expressed in homogeneous coordinates:

$$\begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + t_1 \\ y + t_2 \\ z + t_3 \\ 1 \end{pmatrix}$$

We denote the homogeneous equivalent of a transformation matrix $T$ by $\widetilde{T}$. It is constructed by appending new columns for the coefficients of every structure parameter and one column for the additive constant. For every inserted column, we insert a unit row.

Example:
Let:

$$\begin{array}{rcccc} \Theta(i_1, i_2) = & 3i_1 & +i_2 & & +1 \\ \Pi(i_1, i_2) = & & i_2 & +2n_1 & +5 \end{array}$$

Then, the transformation matrix is defined by:

$$T = \begin{pmatrix} 3 & 1 \\ 0 & 1 \end{pmatrix}$$

Its homogeneous representation $\widetilde{T}$ is:

$$\widetilde{T} = \begin{pmatrix} 3 & 1 & 0 & 1 \\ 0 & 1 & 2 & 5 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In order to avoid ambiguity, the transformation matrix $T$ is sometimes called *non-homogeneous* in contrast to the *homogeneous* transformation matrix $\widetilde{T}$. More information about homogeneous coordinates can be found in books on computer graphics [8, 9, 13].

Assume that we have $k'$ structure parameters in the source program. Note that the relation between a point of the source index space $(i_1, \ldots, i_k)$ and a point of the target index space $(j_1, \ldots, j_k)$ is slightly different from the method in Section 2.5.4:

$$\begin{pmatrix} j_1 \\ \vdots \\ j_k \\ n_1 \\ \vdots \\ n_{k'} \\ 1 \end{pmatrix} = \widetilde{T} \begin{pmatrix} i_1 \\ \vdots \\ i_k \\ n_1 \\ \vdots \\ n_{k'} \\ 1 \end{pmatrix}$$

where $k$ is the number of dimensions in schedule and allocation and $k'$ is the number of structure parameters in the program.
Since the rows in $\widetilde{T}$ which represent structure parameters and constants are the unit vectors, $\widetilde{T}$ is invertible iff the corresponding transformation matrix $T$ is invertible. If $\widetilde{T}$ is invertible, we get:

$$\begin{pmatrix} i_1 \\ \vdots \\ i_k \\ n_1 \\ \vdots \\ n_{k'} \\ 1 \end{pmatrix} = \widetilde{T}^{-1} \begin{pmatrix} j_1 \\ \vdots \\ j_k \\ n_1 \\ \vdots \\ n_{k'} \\ 1 \end{pmatrix}$$

The code generation methods which we discuss in the following two chapters are distinguished by the different forms of output of the schedulers:

1. code generation at the basis of Lamport transformations, which are very simple, iteration-based transformations, and

2. code generation at the basis of Feautrier transformations, which are more complicated and statement-based.

Note that our methods can also be applied to transformations which are created by other schedulers and which are similar to the Lamport or Feautrier transformations.

# 3 Code Generation for Lamport Transformations

## 3.1 Basics

The Lamport scheduler is a very restrictive method for the determination of transformations. It is based on the simple iteration-based polytope model [1, 12].
As we do not treat transformations at the statement-level but on the iteration-level, the source program has to be perfectly nested. Moreover, the schedule is one-dimensional, integral and elementary. The allocation is integral and elementary too, but it may be multi-dimensional. It is choosen with respect to the schedule such that we obtain an invertible unimodular transformation matrix. The transformations do not include structure parameters or additive constants. Thus, a non-homogeneous transformation matrix is sufficient.
In this section, we describe the method for generating the target program. For the formal background, we refer to [1, 12].

## 3.2 Method

First of all, we take a short look at how to construct the target program out of a $k$-dimensional source program:

1. Compose the transformation matrix $T$.

2. The inverse $T^{-1}$ does exist because $T$ is unimodular and of full rank.

3. The following equality is valid:

$$\begin{pmatrix} i_1 \\ i_2 \\ \vdots \\ i_k \end{pmatrix} = T^{-1} \begin{pmatrix} j_1 \\ j_2 \\ \vdots \\ j_k \end{pmatrix}$$

4. All source indices can be represented by the corresponding linear combination of the target indices with the help of this equality.

5. To get the index space of the target program, every loop variable $i_1, \ldots, i_k$ must be replaced by the corresponding linear combination. To construct the target program, the system of inequalities has to be rewritten to get the correct loop bounds. Therefore, a polytope projection method like Fourier-Motzkin elimination [14], PIP [4] or Weispfenning elimination [15] is applied.

6. The result is the index space of the target program. It is convex and can be transformed back into a perfectly nested program. The loop body contains the transformed statements, where every occurrence of a loop variable $i_1, \ldots, i_k$ is replaced by the corresponding linear combination of $j_1, \ldots, j_k$.

## 3.3 Example

The following example is taken from [1]. The source program is shown in Figure 6.

**for** $i_1 = 0$ **to** $n$
  **for** $i_2 = 0$ **to** $2n + 40$
    $x[i_1, i_2] = x[i_1 - 1, i_2] + x[i_1, i_2 - 1]$
  **end**
**end**

Figure 6: Source program of Example 3.3

The Lamport scheduler yields to the following schedule and allocation:

$$\Theta(i_1, i_2) = i_1 + i_2$$
$$\Pi(i_1, i_2) = i_1$$

For details of how to obtain schedule and allocation, see [1].
Thus, our unimodular transformation matrix is given by:

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

Inverting this matrix, we get:

$$T^{-1} = \begin{pmatrix} 0 & 1 \\ 1 & -1 \end{pmatrix}$$

With the help of $T^{-1}$, we can relate the loop indices of the source program $i_1, i_2$ and the loop indices of the target program $j_1, j_2$:

$$\begin{pmatrix} i_1 \\ i_2 \end{pmatrix} = T^{-1} \begin{pmatrix} j_1 \\ j_2 \end{pmatrix}$$

Thus, we get

$$\begin{aligned} i_1 &= j_2 \\ i_2 &= j_1 - j_2 \end{aligned}$$

Note that $j_1$ is the loop variable of the time loop and $j_2$ is the variable of the processor loop.
We replace every occurrence of $i_1, i_2$ in the source statement and in the source index space by its linear combination of $j_1$ and $j_2$.
The source index space is described by the following system of inequalities:

$$\begin{array}{lllll} i_1 & & & & \geq 0 \\ -i_1 & & +n & & \geq 0 \\ & i_2 & & & \geq 0 \\ & -i_2 & +2n & +40 & \geq 0 \end{array}$$

To determine the new loop bounds, we have to substitute $i_1$ and $i_2$ in the source index space by $j_2$ and $j_1 - j_2$ and get the target index space:

$$\begin{array}{lllll} & j_2 & & & \geq 0 \\ & -j_2 & +n & & \geq 0 \\ j_1 & -j_2 & & & \geq 0 \\ -j_1 & +j_2 & +2n & +40 & \geq 0 \end{array}$$

In the target index space, index $j_2$ (representing a processor loop) only depends on structure parameters and an additive constant. And $j_1$ (representing the time loop) depends on $j_2$ plus structure parameters and the constant value. Thus, we can directly construct an asynchronous target program (where the time loop is innermost). If we want to have a synchronous program (the time loop is outermost), the computed index space is not of the desired form because the outer bound $j_1$ depends on the inner bound $j_2$. Thus, we append one of the polytope projection methods [4, 14, 15] to get the concrete loop bounds.
The synchronous and asynchronous target programs of the example are depicted in Figure 7.

1. Synchronous:
   **for** $j_1 = 0$ **to** $3n + 40$
     **parfor** $j_2 = \max(0, j_1 - 2n - 40)$ **to** $\min(n, j_1)$
       $x[j_2, j_1 - j_2] = x[j_2 - 1, j_1 - j_2] + x[j_2, j_1 - j_2 - 1]$
     **end**
   **end**

2. Asynchronous:
   **parfor** $j_2 = 0$ **to** $n$
     **for** $j_1 = j_2$ **to** $j_2 + 2n + 40$
       $x[j_2, j_1 - j_2] = x[j_2 - 1, j_1 - j_2] + x[j_2, j_1 - j_2 - 1]$
     **end**
   **end**

Figure 7: Target programs of the example

# 4 Code Generation for Feautrier Transformations

## 4.1 Basics

As we have seen in the previous chapter, the Lamport scheduler was introduced for an iteration-based derivation of transformations; only perfectly nested programs can be treated. Moreover, there are a lot of restrictions on the set of possible transformations which reduces the potential parallelism.
A more general method to derive schedule and allocation is the Feautrier scheduler [16, 5, 6]. The transformations are at the statement level such that also imperfectly nested loop programs can be treated. All elementary transformations are one- or multi-dimensional, can be non-integral and have structure parameters and a constant as additive component, i.e., elementary transformations are affine functions in the loop variables and parameters. Note that no multiplication of loop indices and parameters is allowed. Moreover, schedule and/or allocation can be piecewise affine functions.
When using the Feautrier transformations, the following problems in the generation of the target program can occur:

- How to handle statement-based transformations?
- How to deal with piecewise transformations?
- How to manage an arbitrary dimensionality in schedule and allocation?
- How to manage structure parameters and constants in the transformations?
- How to handle singular transformation matrices?
- How to handle non-unimodular transformation matrices?

The basic concepts we use in solving these problems are presented in the following section.

## 4.2 Overview

The derivation of the target program can be divided into three steps:

1. Division of the source program into program parts (Section 4.4.1):
   A *program part* is a perfectly nested program with exactly one statement and elementary transformations. Parts are constructed from a source program as follows.
   With respect to the index space of a statement, the source program is divided into perfectly nested program pieces containing one statement (Section 4.4.1.1). For any statement whose schedule or allocation is piecewise, the result is divided once more in order to obtain elementary transformations for every program part (Section 4.4.1.2).

2. Polytope transformation of all program parts (Section 4.3):
   Every source program part is transformed into a target program part. The transformation matrix can be singular. Therefore, the code generation of one program part is divided in several steps:

   (a) elimination of linearly dependent rows from the transformation matrix (Section 4.3.1),

   (b) extension of the transformation matrix to a basis, if necessary (Section 4.3.2),

   (c) construction of the target program part for a unimodular transformation matrix (Section 4.3.3) and for a non-unimodular transformation matrix (Section 4.3.4),

   (d) reinsertion of the transformations which were eliminated in step 2a (Section 4.3.5).

   Steps 2a and 2b render the transformation matrix invertible and step 2d is needed in order to keeping the original transformations.

3. Merge of the target program parts (Section 4.4.2):
   Every target program part is described by a polytope representing the index space and a transformed statement. As a consequence, the complete target program is described by a quasi-convex polytope. Different possibilities exist for the construction of the target program. These methods are also called *merging*.

The code generation for one program part is presented in the following section and subsequently we show the generation and merging of the program parts.

## 4.3 Transformation of the Program Parts

### 4.3.1 Elimination of Linearly Dependent Vectors in Schedule and Allocation

With the Feautrier method, schedule and allocation are calculated independently. Thus, it is possible to get a transformation matrix with linearly dependent rows.
But we need transformation matrices which are invertible in order to be able to apply the same code generation method as for the Lamport transformations. Therefore, we have to eliminate the linearly dependent transformations.
To illustrate the development of this algorithm, we present a motivating example.
Assume that we have a program in two variables $i_1$ and $i_2$ and the following one-dimensional schedule and two-dimensional allocation for a statement $S$:

$$\Theta(i_1, i_2) = i_1 + i_2$$

$$\Pi(i_1, i_2) = \begin{pmatrix} 2i_1 & +2i_2 & +5 \\ & i_2 & +3 \end{pmatrix}$$

As transformation matrix, we get:

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

and:

$$\tilde{T} = \begin{pmatrix} 1 & 1 & 0 \\ 2 & 2 & 5 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{pmatrix}$$

We search linearly dependent rows in the non-homogeneous transformation matrix $T$. The second row is two times the first row. Thus, we eliminate the second row from the matrix and save the corresponding linear combination. Note that we have to take care of the structure parameters and constants while saving. In this example: $j_2 = 2 \cdot j_1 + 5$.
We get:

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

and:

$$\tilde{T} = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 3 \\ 0 & 0 & 1 \end{pmatrix}$$

Thus, the dimensionality of $T$ is equivalent to the number of variables and $T$ is invertible in this example.

Generally, schedule and allocation are written into the transformation matrix. The linearly dependent rows are eliminated by the following algorithm:

Input: Matrix $A = \begin{pmatrix} a_1 \\ \vdots \\ a_k \end{pmatrix}$
where $a_\mu$ represents vectors ($\mu \in \{1, \ldots, k\}$).

Output: Matrix with linearly independent vectors $B = \begin{pmatrix} b_1 \\ \vdots \\ b_\rho \end{pmatrix}$
where $\rho \leq k$ and $\{b_1, \ldots, b_\rho\} \subset \{a_1, \ldots, a_k\}$.

Take the first row $a_1$ of $A$. If $a_1 = (0, \ldots, 0)$ then the vector is dropped (after saving the value), because $\vec{0}$ is defined to be linearly dependent. If $a_1$ is not equal to $\vec{0}$, it is written into $B$ and defined as $b_1$. Assume we have already found the linearly independent vectors $b_1, \ldots, b_\omega$ . Say, we have to determine whether $a_\sigma$ is linearly dependent of the previous vectors $a_1, \ldots, a_{\sigma-1}$, in other words, whether $a_\sigma$ is a linear combination of $b_1, \ldots, b_\omega$. If no, write $a_\sigma$ into $B$ and call it $b_{\omega+1}$.
In order to determine whether a vector is linearly dependent, we use Echelon reduction [1] to get the rank of the matrix formed by $b_1, \ldots, b_\omega$ and $a_\sigma$. If it is smaller than the number of rows in the original matrix, the inserted row is linearly dependent.
Every row which is a linear combination of higher rows is dropped from $A$ and is saved externally.
The termination of the algorithm is guaranteed by the finite number of rows in $A$.

We will reinsert the extracted transformations after the construction of the target program (Section 4.3.5). Thus, the schedule and allocation produced by the Feautrier scheduler are not modified.
Note that, when constructing an asynchronous program, the allocation is written into the transformation matrix before the schedule. In this case, the extraction algorithm tries to preserve the processor dimensions, because the probability of eliminating a row increases with the number of the row.

### 4.3.2 Extension to a Basis

One problem of the result of the Feautrier scheduler is that the constructed transformation matrix often does not have the full rank of $n$ but is $m$-dimensional with $m < n$.
Our goal is to extend the transformation matrix to a base of $\mathbb{Q}^n$ in order to get a one-to-one mapping between every point of the source index space and every point of the target index space.
We develop an algorithm on the basis of the non-homogeneous transformation matrix $T$.

Assume that

$$T = \begin{pmatrix} t_{11} & \dots & t_{1n} \\ \vdots & & \vdots \\ t_{m1} & \dots & t_{mn} \end{pmatrix}$$

has no linear dependent rows. Otherwise, we eliminate them with the method presented in Section 4.3.1.
Let $\vec{i} = (i_1, \dots, i_n)$ be a point of the source index space and $\vec{j} = (j_1, \dots, j_m)$ a point of the target index space. We have:

$$\begin{pmatrix} j_1 \\ \vdots \\ j_m \end{pmatrix} = T \begin{pmatrix} i_1 \\ \vdots \\ i_n \end{pmatrix}$$

We distinguish the cases:

- $m > n$
  $m$ is bigger than $\text{rank}(T)$. This is impossible, because of the elimination of linear dependent rows.

- $m < n$
  There exist exactly $m$ equalities in $n$ unknowns. There is no exact solution. Thus, the given transformation matrix does not specify a bijection between source and target index points.

- $m = n$
  We have the equalities:

$$\begin{pmatrix} j_1 \\ \vdots \\ j_n \end{pmatrix} = T \begin{pmatrix} i_1 \\ \vdots \\ i_n \end{pmatrix}$$

As a consequence, we have a one-to-one mapping between source and target index point iff the transformation matrix is square and of full rank.

In order to transform the original transformation matrix into a square matrix of full rank, we insert linearly independent identity rows at the end of the matrix, spanning the missing dimensions. Every additional identity row describes the insertion of a new processor (or time) layer at the innermost position of the program. Thus, the original transformations and, therefore, the outer loops of the program remain unchanged.

#### 4.3.3 Program Generation for a Unimodular Transformation Matrix

The generation of the target program proceeds analogously to the generation of the target program of the Lamport transformations. We have a perfect loop nest with one statement and an invertible transformation matrix.
Assume that the homogeneous transformation matrix is unimodular. How to handle non-unimodular matrices is shown in Section 4.3.4.

Generally:

1. We can describe the relation between a source index point $(i_1, \ldots, i_k)$ and a target index point $(j_1, \ldots, j_k)$ by the following equality:

$$\begin{pmatrix} i_1 \\ \vdots \\ i_k \\ n_1 \\ \vdots \\ n_{k'} \\ 1 \end{pmatrix} = \widetilde{T}^{-1} \begin{pmatrix} j_1 \\ \vdots \\ j_k \\ n_1 \\ \vdots \\ n_{k'} \\ 1 \end{pmatrix}$$

2. Now, we replace every occurrence of $i_1, \ldots, i_k$ in the source program part by the affine linear combinations of $j_1, \ldots, j_k, n_1, \ldots, n_{k'}$ and a constant.

3. To get the target program part, we must transform the system of inequalities describing the loop bounds by applying one polytope projection method. The order in which we solve the system of inequalities

determines whether the program part will be synchronous or asynchronous.

4. Now we can construct the target program with the help of the target index space and the transformed statements in analogy to the code generation for the Lamport transformations.

#### 4.3.4 Program Generation for a Non-Unimodular Transformation Matrix

Feautrier's transformations often result in non-unimodular matrices. In this section, we explain the problems appearing with non-unimodularity. Then, we present the original algorithm of Xue [17] for non-unimodular, integer, non-homogeneous transformation matrices. This method is extended in order to handle non-unimodular, integer, homogeneous transformation matrices by reducing the affine problem to the linear problem and determining the solution with Xue's algorithm. Subsequently, we present an algorithm for solving non-unimodular, non-integer, homogeneous transformation matrices where the scaling problem is also reduced to the linear problem.

##### 4.3.4.1 Problems with Non-Unimodular Transformation Matrices

If we have a non-unimodular transformation matrix $T$, some integer points are not in the target index space: we have so-called 'holes' in the target index space.

Let $(j_1, \ldots, j_n)$ be a hole of the target index space. We can determine the corresponding point in the source index space $(i_1, \ldots, i_n)$ with the help of the inverse $T^{-1}$:

$$\begin{pmatrix} i_1 \\ \vdots \\ i_n \end{pmatrix} = T^{-1} \begin{pmatrix} j_1 \\ \vdots \\ j_n \end{pmatrix}$$

The inverse $T^{-1}$ is fractional because $T$ is non-unimodular. Thus, $(i_1, \ldots, i_n)$ must be a non-integer vector. This cannot be correct, because every lattice point of the source index space is an element of $\mathbb{Z}^n$ and $(i_1, \ldots, i_n)$ is a point of the source index space.

We skip the holes by changing strides and lower bound offsets of the target program part. The method of deriving strides and offsets is presented in the following section.
For more details see Xue [17].

**4.3.4.2 Step and Offset Derivation for Non-Unimodular, Integer, Non-Homogeneous Transformation Matrices** The algorithm presented by Xue works as follows:

1. Determine the loop bounds as usual. We get $l_{kr}$ and $u_{kr}$, where $k$ describes the dimension of the loop bounds and $r$ the $r$-th term in a minimum or maximum expression.

2. Determine the Hermite normal form $\Delta$ of the non-homogeneous transformation matrix $T$.

3. Then, the $k$-th stride is defined by

$$s_k = \Delta_{kk}$$

   where $\Delta_{kk}$ is the $k$-th diagonal element of $\Delta$.

4. The lower bound offset $\delta_{kr}$ is determined by the following algorithm. Determine the vector $(X_{1,r}, \ldots, X_{k,r}, *)$, where the last $n-k$ components are unspecified, such that the following equality is valid:

$$\Delta \cdot \begin{pmatrix} X_{1,r} \\ \vdots \\ X_{k-1,r} \\ X_{k,r} \\ \star \end{pmatrix} = \begin{pmatrix} j_1 \\ \vdots \\ j_{k-1} \\ \lceil l_{kr} \rceil \\ \star \end{pmatrix}$$

Then, the lower bound offset $\delta_{kr}$ is defined by:

$$\delta_{kr} = (-X_{k,r} \cdot \Delta_{kk})\%\Delta_{kk}$$

where % defines the modulo operation.

Note that the upper loop bounds require no change.
Let us consider an example. Let the source program part be:

**for** $i_1 = 1$ **to** $5$
    **for** $i_2 = 1$ **to** $5$
        */*loop body*/*
    **end**
**end**

Assume that the transformation matrix $T$ is defined by:

$$T = \begin{pmatrix} 3 & 2 \\ 1 & 0 \end{pmatrix}$$

Inverting $T$, we get:

$$T^{-1} = \frac{1}{2} \cdot \begin{pmatrix} 0 & 1 \\ 1 & -3 \end{pmatrix}$$

The Hermite normal form $\Delta$ of $T$ is:

$$\Delta = \begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix}$$

The strides can be derived from the diagonal of $\Delta$:

$$s_1 = 1$$
$$s_2 = 2$$

Let us construct a synchronous program.
Computing the loop bounds, for example with the Fourier-Motzkin algorithm, we get:

$$l_{11} = 5 \ , \ \ u_{11} = 25$$
$$l_{21} = 1 \ , \ \ u_{21} = 5$$
$$l_{22} = \tfrac{1}{3} \cdot (j_1 - 10) \ , \ \ u_{22} = \tfrac{1}{3} \cdot (j_1 - 2)$$

We can compute the lower bound offsets by the given definition of step 4 of the method:

- $\begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} X_{1,1} \\ \star \end{pmatrix} = \begin{pmatrix} \lceil l_{11} \rceil \\ \star \end{pmatrix}$

  $X_{1,1} = \lceil l_{11} \rceil = \lceil 5 \rceil = 5$

  $\delta_{11} = (-X_{1,1} \cdot \Delta_{11}) \% \Delta_{11} = (-5 \cdot 1) \% 1 = 0$

- $\begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} X_{1,1} \\ X_{2,1} \end{pmatrix} = \begin{pmatrix} j_1 \\ \lceil l_{21} \rceil \end{pmatrix}$

  $X_{1,1} = j_1$
  $X_{1,1} + 2 \cdot X_{2,1} = \lceil l_{21} \rceil$

  $-X_{2,1} = \frac{1}{2} \cdot (j_1 - \lceil l_{21} \rceil) = \frac{1}{2} \cdot (j_1 - 1)$

  $\delta_{21} = (-X_{2,1} \cdot \Delta_{22}) \% \Delta_{22} = (j_1 - 1) \% 2$

- $\begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} X_{1,2} \\ X_{2,2} \end{pmatrix} = \begin{pmatrix} j_1 \\ \lceil l_{22} \rceil \end{pmatrix}$

$$-X_{2,2} = \frac{1}{2} \cdot (j_1 - \lceil l_{22} \rceil) = \frac{1}{2} \cdot \left( j_1 - \left\lceil \frac{1}{3}(j_1 - 10) \right\rceil \right)$$

$$\delta_{22} = (-X_{2,2} \cdot \Delta_{22}) \% \Delta_{22} = \left( j_1 - \left\lceil \frac{1}{3}(j_1 - 10) \right\rceil \right) \% 2$$

Thus, the target program is:

**for** $j_1 = 5 + 0$ **to** $25$ **step** $1$
    **parfor** $j_2 = \max\left( 1 + (j_1 - 1)\%2, \left\lceil \frac{1}{3}(j_1 - 10) \right\rceil + \left( j_1 - \left\lceil \frac{1}{3}(j_1 - 10) \right\rceil \right)\%2 \right)$
        **to** $\min\left( 5, \left\lfloor \frac{1}{3}(j_1 - 2) \right\rfloor \right)$
        **step** $2$
            */*transformed loop body*/*
    **end**
**end**

Figure 8 shows the polytope describing the target index space of the example, where the dots represent the iteration points.

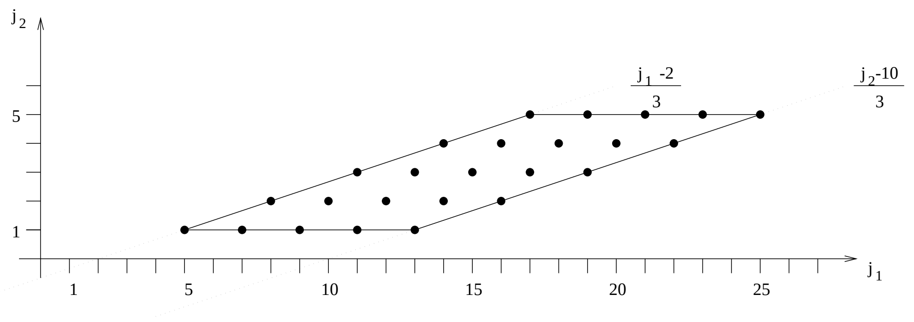



Figure 8: Polytope of the example

Note that, for example, the point $p = (6, 1)$ is skipped. For $j_1 = 6$, $j_2$ is described by:

$$\textbf{parfor } j_2 = \max\Big(\ 1 + (5\%2), \Big\lceil\ \tfrac{1}{3}(\ -4)\Big\rceil + \Big(\ 6 - \Big\lceil\ \tfrac{1}{3}(\ -4)\Big\rceil\Big)\ \%2\Big)$$
$$\textbf{to } \min\Big(\ 5, \Big\lfloor\ \tfrac{1}{3}(\ 4)\Big\rfloor\Big) \textbf{ step } 2$$

Thus,

$$\textbf{parfor } j_2 = \max(2,0) \textbf{ to } \min(5,1) \textbf{ step } 2$$

i.e.

$$\textbf{parfor } j_2 = 2 \textbf{ to } 1 \textbf{ step } 2 = \{\}$$

There exists no point at coordinate $j_1 = 6$, even though the target program part without strides would specify an iteration at this coordinate.

**4.3.4.3 Step and Offset Derivation for Non-Unimodular, Integer, Homogeneous Transformation Matrices** The reason for introducing homogeneous coordinates is to simplify the translation of the source polytope. Let $(o_1, \ldots, o_n)$ be the translation vector consisting of the structure parameter and constant part of schedule and allocation.
Let $(i'_1, \ldots, i'_n)$ be a source index point before the translation. The according translated index point $(i_1, \ldots, i_n)$ is defined by:

$$\begin{pmatrix} i_1 \\ \vdots \\ i_n \end{pmatrix} = \begin{pmatrix} i'_1 \\ \vdots \\ i'_n \end{pmatrix} + \begin{pmatrix} o_1 \\ \vdots \\ o_n \end{pmatrix} = \begin{pmatrix} i'_1 + o_1 \\ \vdots \\ i'_n + o_n \end{pmatrix}$$

Point $(j'_1, \ldots, j'_n)$ in the non-translated and $(j_1, \ldots, j_n)$ in the translated target space have the same relationship:

$$\begin{pmatrix} j_1 \\ \vdots \\ j_n \end{pmatrix} = \begin{pmatrix} j'_1 \\ \vdots \\ j'_n \end{pmatrix} + \begin{pmatrix} o_1 \\ \vdots \\ o_n \end{pmatrix} = \begin{pmatrix} j'_1 + o_1 \\ \vdots \\ j'_n + o_n \end{pmatrix}$$

Thus, instead of applying the algorithm to $j_\mu$, we apply it to $j_\mu + o_\mu$, for all $\mu \in \{1, \ldots, n\}$.

We extend the algorithm of Xue in order to determine strides and lower bound offsets for a translated polytope:

1. Determine the loop bounds with the help of $\widetilde{T}$. We get $l_{kr}$ and $u_{kr}$, where $k \in \{1, \ldots, n\}$ and $r$ is the $r$-th term in a minimum or maximum expression.

2. Determine the Hermite normal form $\Delta$ of the non-homogeneous transformation matrix $T$.

3. Then, the $k$-th stride is defined by:

$$s_k = \Delta_{kk}$$

4. The lower bound offsets can be determined by the derivation of the vector $(X_{1,r}, \ldots, X_{k,r})$ which is defined by the following equality:

$$\Delta \cdot \begin{pmatrix} X_{1,r} \\ \vdots \\ X_{k-1,r} \\ X_{k,r} \\ \star \end{pmatrix} = \begin{pmatrix} j_1 & - & o_1 \\ \vdots & & \\ j_{k-1} & - & o_{k-1} \\ \lceil l_{kr} \rceil & - & o_k \\ \star & & \end{pmatrix}$$

   Then, the lower bound offset $\delta_{kr}$ is defined by:

$$\delta_{kr} = (-X_{k,r} \cdot \Delta_{kk}) \% \Delta_{kk}$$

Let us consider an example.
Take the same loop as in the example of the previous paragraph. Schedule and allocation are given by:

$$\begin{array}{rrrr} \Theta(i_1, i_2) = & 3i_1 & +2i_2 & +2 \\ \Pi(i_1, i_2) = & i_1 & & +3 \end{array}$$

Thus, the homogeneous transformation matrix $\widetilde{T}$ is defined by:

$$\widetilde{T} = \begin{pmatrix} 3 & 2 & 2 \\ 1 & 0 & 3 \\ 0 & 0 & 1 \end{pmatrix}$$

The inverted transformation matrix is:

$$\widetilde{T}^{-1} = \begin{pmatrix} 0 & 1 & -3 \\ \frac{1}{2} & -\frac{3}{2} & \frac{7}{2} \\ 0 & 0 & 1 \end{pmatrix}$$

Let us construct a synchronous program. Computing the loop bounds, for instance with the Fourier-Motzkin elimination algorithm, we get:

$$\begin{array}{l} l_{11} = 7,\ u_{11} = 27 \\ l_{21} = 4,\ u_{21} = 8 \\ l_{22} = \frac{1}{3} \cdot (j_1 - 3),\ u_{22} = \frac{1}{3} \cdot (j_1 + 5) \end{array}$$

The Hermite normal form $\Delta$ of $T$ is:

$$\Delta = \begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix}$$

The strides can be read from the diagonal of $\Delta$:

$$\begin{array}{l} s_1 = 1 \\ s_2 = 2 \end{array}$$

We can compute the lower bound offsets by:

- $\begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} X_{1,1} \\ \star \end{pmatrix} = \begin{pmatrix} \lceil l_{11} \rceil & - & 1 \\ & \star & \end{pmatrix}$

  $-X_{1,1} = -6$

$\delta_{11} = 0$

- $\begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} X_{1,1} \\ X_{2,1} \end{pmatrix} = \begin{pmatrix} j_1 & - & 1 \\ \lceil l_{21} \rceil & - & 2 \end{pmatrix}$

  $-X_{2,1} = \frac{1}{2} \cdot (j_1 - 3)$

  $\delta_{21} = (j_1 - 3)\%2$

- $\begin{pmatrix} 1 & 0 \\ 1 & 2 \end{pmatrix} \cdot \begin{pmatrix} X_{1,2} \\ X_{2,2} \end{pmatrix} = \begin{pmatrix} j_1 & - & 1 \\ \lceil l_{22} \rceil & - & 2 \end{pmatrix}$

  $-X_{2,2} = \frac{1}{2} \cdot \left( j_1 + 1 - \left\lceil \frac{1}{3}(j_1 - 3) \right\rceil \right)$

  $\delta_{22} = \left( j_1 + 1 - \left\lceil \frac{1}{3}(j_1 - 3) \right\rceil \right) \%2$

Thus, the target program is:

$$
\begin{array}{l}
\textbf{for } j_1 = 7 \textbf{ to } 27 \textbf{ step } 1 \\
\quad \textbf{parfor } j_2 = \max\left( 4 + (j_1 - 3)\%2, \left\lceil \frac{1}{3}(j_1 - 3) \right\rceil + \right. \\
\qquad\qquad\qquad \left. \left( j_1 + 1 - \left\lceil \frac{1}{3}(j_1 - 3) \right\rceil \right) \%2 \right) \\
\qquad \textbf{to } \min\left( 8, \left\lfloor \frac{1}{3}(j_1 + 5) \right\rfloor \right) \\
\qquad \textbf{step } 2 \\
\qquad\qquad \textit{/*transformed loop body*/} \\
\quad \textbf{end} \\
\textbf{end}
\end{array}
$$

Figure 9 depicts the polytope describing the index space of the example, where the dots describe the iteration space. Note that we simply translate the polytope by 2 in the direction of the $j_1$-axis and by 3 in the direction of the $j_2$-axis.

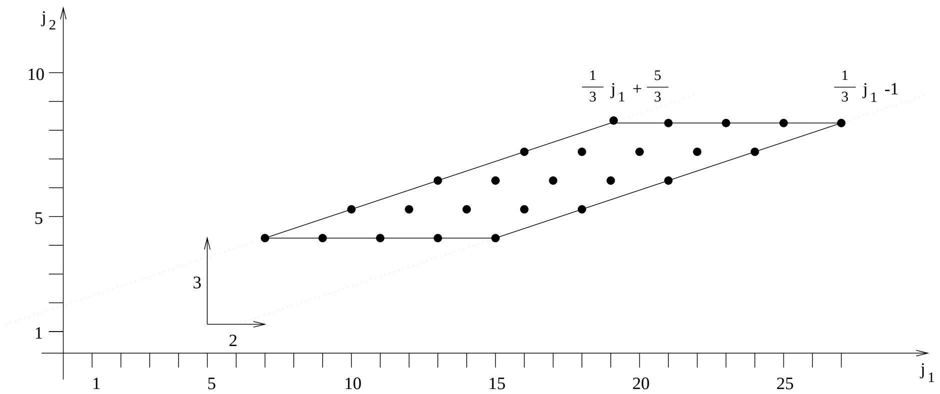



Figure 9: Polytope of the translated example

**4.3.4.4 Step and Offset Derivation for Non-Unimodular, Non-Integer, Homogeneous Transformation Matrices** The Feautrier scheduler often produces a non-integer transformation matrix. Xue's algorithm only works for integer transformation matrices. We extend his method with the help of the denominator. Thus:

1. The transformation matrix $\widetilde{T}$ is defined by:

   $$\widetilde{T} = \frac{1}{d} \cdot \widetilde{T'}$$

   where $\widetilde{T'}$ is an integer matrix and $d$ is the denominator of $\widetilde{T}$.

2. Find the target program for $\widetilde{T'}$ with the help of the algorithms described in the previous sections. Note that also a translation is possible.

3. Let $(j_1, \ldots, j_\mu)$ be a point of the target index space defined by $\widetilde{T}$, $(j'_1, \ldots, j'_\mu)$ a point of the target index space defined by $\widetilde{T'}$ and $(i_1, \ldots, i_\mu)$ a source index point. Let $\nu$ be the number of structure parameters in the program.

   Then, the following equalities are valid:

$$\begin{pmatrix} j_1 \\ \vdots \\ j_\mu \\ n_1 \\ \vdots \\ n_\nu \\ 1 \end{pmatrix} = \widetilde{T} \cdot \begin{pmatrix} i_1 \\ \vdots \\ i_\mu \\ n_1 \\ \vdots \\ n_\nu \\ 1 \end{pmatrix}$$

and

$$\begin{pmatrix} j'_1 \\ \vdots \\ j'_\mu \\ n_1 \\ \vdots \\ n_\nu \\ 1 \end{pmatrix} = \widetilde{T'} \cdot \begin{pmatrix} i_1 \\ \vdots \\ i_\mu \\ n_1 \\ \vdots \\ n_\nu \\ 1 \end{pmatrix}$$

Thus,

$$\begin{pmatrix} j'_1 \\ \vdots \\ j'_\mu \\ n_1 \\ \vdots \\ n_\nu \\ 1 \end{pmatrix} = \widetilde{T'} \cdot \begin{pmatrix} i_1 \\ \vdots \\ i_\mu \\ n_1 \\ \vdots \\ n_\nu \\ 1 \end{pmatrix} = d \cdot \widetilde{T} \cdot \begin{pmatrix} i_1 \\ \vdots \\ i_\mu \\ n_1 \\ \vdots \\ n_\nu \\ 1 \end{pmatrix} = d \cdot \begin{pmatrix} j_1 \\ \vdots \\ j_\mu \\ n_1 \\ \vdots \\ n_\nu \\ 1 \end{pmatrix}$$

We can substitute every instance of $j'_k$ ($k \in \{1, \ldots, \mu\}$) in the computed target program by $d \cdot j_k$. Then, we only have to transform the program such that we get the indices $j_k$ instead of $d \cdot j_k$. We divide every loop bound and every stride by $d$.

Let us consider an example. We start with the same program as in the previous paragraphs. Schedule and allocation are defined by:

$$\begin{array}{rl} \Theta(i_1, i_2) = & \frac{3}{5}i_1 \quad + \frac{2}{5}i_2 \\ \Pi(i_1, i_2) = & \frac{1}{5}i_1 \end{array}$$

There is no translation. Thus, the transformation matrix $T$ is defined by:

$$T = \frac{1}{5} \cdot \begin{pmatrix} 3 & 2 \\ 1 & 0 \end{pmatrix}$$

$T'$ is defined by:

$$T' = \begin{pmatrix} 3 & 2 \\ 1 & 0 \end{pmatrix}$$

We determine the target program for $T'$ with the algorithm presented in Section 4.3.

$$\begin{array}{l}
\textbf{for } j'_1 = 5 \textbf{ to } 25 \textbf{ step } 1 \\
\quad \textbf{parfor } j'_2 = \max\Big( 1 + (\, j'_1 - 1)\,\%2, \Big\lceil \frac{1}{3}(\, j'_1 - 10) \Big\rceil + \\
\qquad\qquad\qquad\qquad \Big(\, j'_1 - \Big\lceil \frac{1}{3}(\, j'_1 - 10) \Big\rceil \Big)\,\%2 \Big) \\
\qquad \textbf{to } \min\Big( 5, \Big\lfloor \frac{1}{3}(\, j'_1 - 2) \Big\rfloor \Big) \\
\qquad \textbf{step } 2 \\
\qquad\qquad /^{*}\textit{transformed loop body in } j'_1 \textit{ and } j'_2{}^{*}/ \\
\quad \textbf{end} \\
\textbf{end}
\end{array}$$

We know:

$$\begin{pmatrix} j'_1 \\ j'_2 \end{pmatrix} = 5 \cdot \begin{pmatrix} j_1 \\ j_2 \end{pmatrix}$$

Thus, we get the intermediate solution:

$$
\begin{array}{l}
\textbf{for } 5j_1 = 5 \textbf{ to } 25 \textbf{ step } 1\\
\quad \textbf{parfor } 5j_2 = \max\Big(\,1 + (\,5j_1 - 1)\,\%2, \Big\lceil\,\tfrac{1}{3}(\,5j_1 - 10)\Big\rceil +\\
\qquad\qquad\qquad\qquad \Big(\,5j_1 - \Big\lceil\,\tfrac{1}{3}(\,5j_1 - 10)\Big\rceil\Big)\,\%2\Big)\\
\qquad \textbf{to } \min\Big(\,5, \Big\lfloor\,\tfrac{1}{3}(\,5j_1 - 2)\Big\rfloor\Big)\\
\qquad \textbf{step } 2\\
\qquad\qquad /^*\textit{transformed loop body in } 5j_1 \textit{ and } 5j_2\,^*/\\
\quad \textbf{end}\\
\textbf{end}
\end{array}
$$

Introducing the denominator, we get:

$$
\begin{array}{l}
\textbf{for } j_1 = \frac{5}{5} \textbf{ to } \frac{25}{5} \textbf{ step } \frac{1}{5}\\
\quad \textbf{parfor } j_2 = \max\Big(\frac{1+(\,5j_1-1)\%2}{5},\\
\qquad\qquad\qquad \frac{\lceil\,\frac{1}{3}(\,5j_1-10)\rceil + (\,5j_1 - \lceil\,\frac{1}{3}(\,5j_1-10)\rceil)\%2}{5}\Big)\\
\qquad \textbf{to } \min\Big(\,\frac{5}{5}, \frac{\lfloor\,\frac{1}{3}(\,5j_1-2)\rfloor}{5}\Big)\\
\qquad \textbf{step } \frac{2}{5}\\
\qquad\qquad /^*\textit{transformed loop body in } j_1 \textit{ and } j_2\,^*/\\
\quad \textbf{end}\\
\textbf{end}
\end{array}
$$

Figure 10 shows the polytope describing the index space of the example where the dots represent the iteration points. Note that we have simply compressed the polytope by a factor of 5.

### 4.3.5 Reinsertion of the Eliminated Rows into the Target Program Part

We have eliminated some rows of the transformation matrix in Section 4.3.1, which we now have to reinsert, such that the original transformations are not modified.

Assume that we eliminated row $r_\mu$ from the transformation matrix and that $r_\mu$ is the following linear combination of the first $r_1, \ldots, r_{\mu-1}$ rows, where $r_0$ is the value of the structure parameters and the constant:

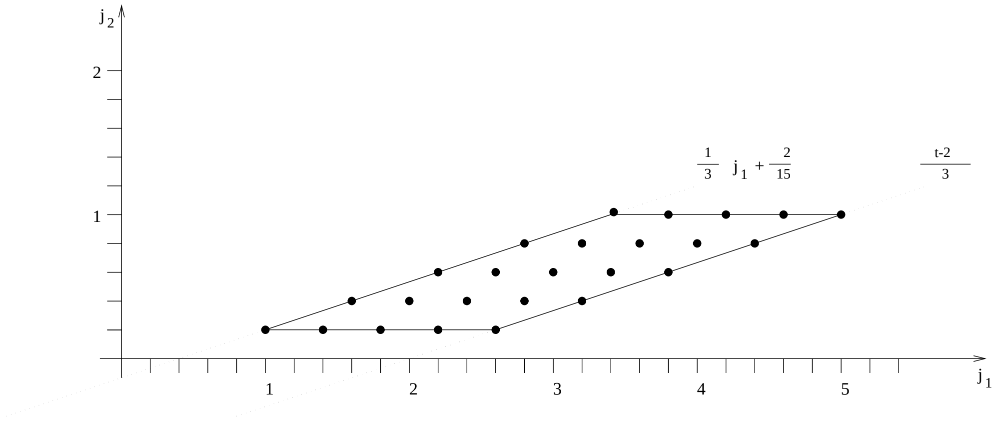



Figure 10: Polytope of the shrinked example

$$r_\mu = a_1 \cdot r_1 + \ldots + a_{\mu-1} \cdot r_{\mu-1} + r_0$$

Then, we insert a **for** loop or a **parfor** loop, respectively, at the $\mu$-th dimension of the loop nest, where the lower and upper bound are both:

$$a_1 \cdot j_1 + \ldots + a_{\mu \ -1} \cdot j_{\mu \ -1} + r_0$$

where $j_1, \ldots, j_{\mu \ -1}$ are the indices of the outermost loops.
The type of the reinserted loop (**for** or **parfor**) depends on the kind of transformation (schedule or allocation) which we eliminated.
Assume that the synchronous target program of the example in Section 4.3.1 is:

**for** $j_1 = l_1$ **to** $u_1$
    **parfor** $j_2 = l_2$ **to** $u_2$
        $/ * \mathit{body} * /$
    **end**
**end**

and we know, that row 2 is two times row 1 plus five ($r_2 = 2r_1 + 5$). The eliminated row was an allocation row. Thus, we insert a new second dimension, a **parfor** loop with lower and upper bound $2j_1+5$. The target program becomes:

**for** $j_1 = l_1$ **to** $u_1$
  **parfor** $j_3 = 2j_1 + 5$ **to** $2j_1 + 5$
    **parfor** $j_2 = l_2$ **to** $u_2$
      $/ * \, body * /$
  **end**
**end**

## 4.4 Generation and Merging of Program Parts

In this section, we present a method for generating program parts and merging the solutions.

### 4.4.1 Generation of Program Parts

As we have already seen in the overview (Section 4.2), the Feautrier scheduler yields individual transformations for every statement in the source program. Thus, we split the program into one piece of program per statement with respect to the surrounding index space.
Note that the result of the first step is, in general, only one program part if the corresponding transformation is elementary. One piece is divided once more if the corresponding transformation is defined piecewise.
The result of the partitioning is a set of perfectly nested program parts with one statement. Every program part has exactly one elementary multidimensional schedule and allocation.
The details of this method are presented in the subsequent sections.

**4.4.1.1 Program Parts for Statements** In this section, we describe the division of the source program if we have different statements.

Let us consider a motivating example:

**for** $i_1 = 0$ **to** 5
    **for** $i_2 = 0$ **to** 5
        $S_1$
        $S_2$
    **end**
**end**

The schedule and the allocation are defined by:

$$\Theta_{S_1}(i_1, i_2) = i_1$$
$$\Pi_{S_1}(i_1, i_2) = -i_2$$

$$\Theta_{S_2}(i_1, i_2) = \quad \textbf{if } (i_2 - 1) \geq 0 \textbf{ then } i_1 + 1 \; \textbf{else } i_1 + i_2$$

$$\Pi_{S_2}(i_1, i_2) = i_2 + 1$$

We divide this program into two program parts:

$P'_1$: **for** $i_1 = 0$ **to** 5
        **for** $i_2 = 0$ **to** 5
            $S_1$
        **end**
    **end**

$P'_2$: **for** $i_1 = 0$ **to** 5
        **for** $i_2 = 0$ **to** 5
            $S_2$
        **end**
    **end**

The index space of $P'_1$ is equal to the index space of $P'_2$ because of the perfectly nested source program:

$$0 \leq i_1 \leq 5$$
$$0 \leq i_2 \leq 5$$

Generally, the index space of every statement is determined by analysing the surrounding loops. As a consequence, imperfectly nested loops can be treated as well. Schedule and allocation are retained for the program parts.
In our example:
The statement of $P_1'$ has the transformations:

$$\Theta_{S_1}(i_1, i_2) = i_1$$
$$\Pi_{S_1}(i_1, i_2) = -i_2$$

and the statement of $P_2'$:

$$\Theta_{S_2}(i_1, i_2) = \quad \textbf{if } (i_2 - 1) \geq 0 \quad \begin{array}{l} \textbf{then } i_1 + 1 \\ \textbf{else } i_1 + i_2 \end{array}$$

$$\Pi_{S_2}(i_1, i_2) = i_2 + 1$$

**4.4.1.2 Program Parts for Piecewise Transformations** A program part must be divided once more if the corresponding statement is subject to piecewise transformations (schedule or allocation or both).
A piecewise transformation is described by a binary tree. There exists an elementary transformation for every leaf of this tree. Thus, we replace the old program part by a new program part for every leaf, where the index space of the new program part is a piece of the index space of the old program part. This piece depends on the conditions of the nested if-then-else transformation.
Assume that we have the schedule:

$$\textbf{if } \mathit{condition} \quad \begin{array}{l} \textbf{then} < \mathit{schedule}_1 > \\ \textbf{else} < \mathit{schedule}_2 > \end{array}$$

The statement must be executed at time $< \mathit{schedule}_1 >$ when the condition is true and at time $< \mathit{schedule}_2 >$ when the condition is false. Thus, we divide the index space into two parts, one with the points which satisfy the condition and the other with the points which do not.
We treat piecewise allocations analogously.

In our example, the schedule of statement 2 is not an elementary transformation. The index space of statement 2 is represented by the following system of inequalities:

$$\begin{array}{llll} i_1 & & & \geq 0 \\ -i_1 & & +5 & \geq 0 \\ & i_2 & & \geq 0 \\ & -i_2 & +5 & \geq 0 \end{array}$$

Thus, we can divide $P'_2$ into the following subprograms.
$P'_{2,1}$ with the index space (add the condition $(i_2 - 1) \geq 0$):

$$\begin{array}{llll} i_1 & & & \geq 0 \\ -i_1 & & +5 & \geq 0 \\ & i_2 & & \geq 0 \\ & -i_2 & +5 & \geq 0 \\ & i_2 & -1 & \geq 0 \end{array}$$

$P'_{2,2}$ with the index space (add the condition $not((i_2 - 1) \geq 0)$):

$$\begin{array}{llll} i_1 & & & \geq 0 \\ -i_1 & & +5 & \geq 0 \\ & i_2 & & \geq 0 \\ & -i_2 & +5 & \geq 0 \\ & -i_2 & & \geq 0 \end{array}$$

The transformations of $P_{2,1}$ are:

$$\Theta_{S_{2,1}}(i_1, i_2) = i_1 + 1$$
$$\Pi_{S_{2,1}}(i_1, i_2) = i_2 + 1$$

and the transformations for $P_{2,2}$ are:

$$\Theta_{S_{2,2}}(i_1, i_2) = i_1 + i_2$$
$$\Pi_{S_{2,2}}(i_1, i_2) = i_2 + 1.$$

Note that the statements $S_{2,1}$ and $S_{2,2}$ are copies of $S_2$.

### 4.4.2 Merging of Program Parts

At this point, the source program is divided into several program parts (Section 4.4.1), which we transform with the algorithms of Section 4.3.3 or 4.3.4. The results are several target program parts, represented by the polytopes $P_1, \ldots, P_k$ and the transformed statements $S_1, \ldots, S_k$. The complete ($m$-dimensional) target program is described by the union of these polytopes: the quasi-convex polytope $P = P_1 \cup \ldots \cup P_k$, where $m$ is the maximum of the dimensionalities of $P_1, \ldots, P_k$.
The goal of this section is to derive the target program $P'$ from the quasi-convex polytope $P$.
Different possibilities exist for solving this problem.
The most general possibility is the run-time solution described in Section 4.4.2.1.
In Section 4.4.2.2 some alternatives are presented: the parallel execution operator solution for asynchronous programs and the compile-time solution.

**4.4.2.1 Run-Time Solution** The run-time solution can briefly be described by constructing the convex hull or any convex superset of the quasi-convex polytope and deciding at run time whether a point of the set is member of the polytope or not.
The run-time solution can be applied to synchronous or asynchronous target programs.
The only restriction is that the schedules (or allocations) of all statements must have the same dimensionality. The reason for this restriction is given at the end of this section.

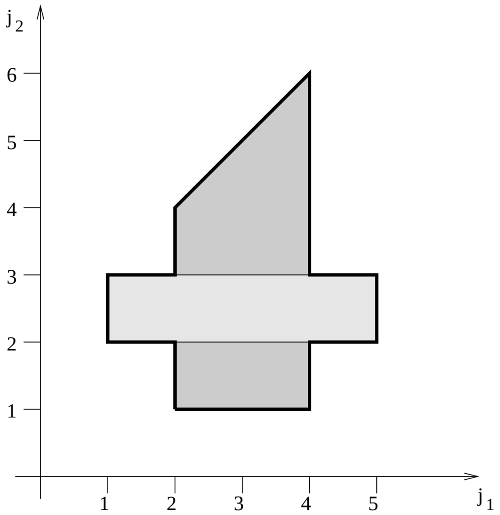
First of all, let us discuss a motivating example.
Assume that we have two program parts $P_1'$ and $P_2'$ with

$P_1'$: **for** $j_1 = 1$ **to** $5$
  **parfor** $j_2 = 2$ **to** $3$
    $S_1$
  **end**
**end**

and

$P_2'$: **for** $j_1 = 2$ **to** $4$
  **parfor** $j_2 = 1$ **to** $j_1 + 2$
    $S_2$
  **end**
**end**

The program parts $P_1'$ and $P_2'$ have to be merged to a target program $P'$. The index space of $P_1'$ is represented by the convex polytope $P_1$ and the index space of $P_2'$ by $P_2$. The quasi-convex polytope $P$ is formed by $P_1 \cup P_2$ (Figure 11).



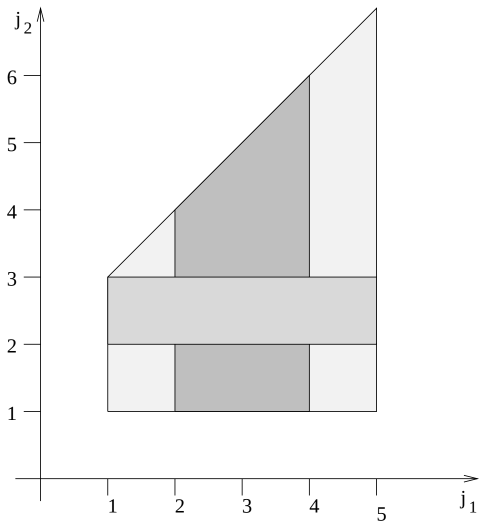

$$
\begin{aligned}
P_1 :\ & 1 \le j_1 \le 5 \\
& 2 \le j_2 \le 3 \\
P_2 :\ & 2 \le j_1 \le 4 \\
& 1 \le j_2 \le j_1 + 2
\end{aligned}
$$

Figure 11: Polytope $P$

Now, a convex set $P_{con}$ is constructed around the quasi-convex polytope, such that the target program consists of exactly one loop nest. Formally:

$$P_1 \subseteq P_{con} \wedge P_2 \subseteq P_{con} \wedge P_{con} \text{ convex}$$

The easiest way of deriving $P_{con}$ (Figure 12) is to take the minimum (the maximum) of all lower bounds (all upper bounds) of every dimension.




$$P_{con}: \quad 1 \leq j_1 \leq 5$$
$$\qquad 1 \leq j_2 \leq \max\{3, j_1 + 2\}$$

Figure 12: Convex set of polytope $P$

Thus, we get the following values for the lower and upper bounds of the target program $P'$:

$$\begin{aligned} l_1 &= \min\{1, 2\} = 1 \\ u_1 &= \max\{5, 4\} = 5 \\ l_2 &= \min\{2, 1\} = 1 \\ u_2 &= \max\{3, j_1 + 2\} \end{aligned}$$

Having constructed the index space for the target program, we have to look at the actions which have to take place. If $(j_1, j_2) \in \mathbb{Z}^2$ is an element of the quasi-convex polytope $P$, an action has to take place. This means that $S_1$ has to be executed if $(j_1, j_2) \in P_1$, and $S_2$ if $(j_1, j_2) \in P_2$. We know that $(j_1, j_2)$ is in $P_1$ iff the inequalities $1 \leq j_1 \leq 5$ and $2 \leq j_2 \leq 3$ are valid (analogously for $(j_1, j_2) \in P_2$).

Thus, the target program of our example is:

```
for j1 = 1 to 5
    parfor j2 = 1 to max{3, j1 + 2}
        if (1 ≤ j1 ≤ 5 ∧ 2 ≤ j2 ≤ 3)
            S1
        fi
        ||
        if (2 ≤ j1 ≤ 4 ∧ 1 ≤ j2 ≤ j1 + 2)
            S2
        fi
    end
end
```

Let us generalize this example.
To form the convex superset $P_{con}$ of $P$, the $r$-th lower bound $l_r$ of $P'$ ($r \in \{1, \ldots, m\}$) is defined by the minimum of the $r$-th lower bounds $l_r^{(1)}, \ldots, l_r^{(k)}$ of all program parts:

$$l_r = \min\{l_r^{(1)}, \ldots, l_r^{(k)}\}$$

The $r$-th upper bound $u_r$ of $P'$ is given by the maximum of the $r$-th upper bounds $u_r^{(1)}, \ldots, u_r^{(k)}$ of all program parts:

$$u_r = \max\{u_r^{(1)}, \ldots, u_r^{(k)}\}$$

Note that, if a lower or an upper bound of a program part is not defined, caused by the fact that the dimensionality of this program part is less than $m$, this bound is omitted in the previous minimum or maximum expression. Every point $(j_1, \ldots, j_n) \in \mathbb{Z}^n$ must execute one or more (maximally $k$) actions, if they are element of $P$. If $(j_1, \ldots, j_n)$ is not an element of $P$, no action will take place.

Every statement $S_\mu$ in the target program $P'$ is surrounded by an if-statement in order to decide if the iteration point is element of $P_\mu$ or not:

> **if** $((j_1, \ldots, j_n) \in \ P_\mu)$
>   $S_\mu$
> **fi**

for all $\mu \in \{1, \ldots, k\}$.

These statements can be executed in parallel.

Note that this algorithm can also be applied when the $k$ polytopes $P_\mu$ do not overlap. In this case, it is not necessary to construct the convex set $P_{con}$, because the $k$ program parts $P'_1, \ldots, P'_k$ could be executed in sequence without creating conflicts. Our method does so anyway, causing a bigger run-time overhead, but is not compromising correctness.

If one program part $k'$ has $m' < m$ dimensions, the if-clause surrounding statement $S'_k$ is written after the $m'$-th for-loop in the target program.
Example:

Given the target program parts:

$P'_1$: **for** $j_1 = 1$ **to** $4$
  $S_1$
**end**

$P'_2$: **for** $j_1 = 2$ **to** $5$
  **parfor** $j_2 = j_1$ **to** $j_1 + 3$
    $S_2$
  **end**
**end**

the target program is:

**for** $j_1 = 1$ **to** $5$
  **if** $(1 \le j_1 \le 4)$
    $S_1$
  **fi**
  $\|$
  **parfor** $j_2 = j_1$ **to** $j_1 + 3$
    **if** $(2 \le j_1 \le 5 \wedge j_1 \le j_2 \le j_1 + 3)$
      $S_2$
    **fi**
  **end**
**end**

Instead of deriving an arbitrary convex superset of the quasi-convex polytope, it would be better to construct the convex hull, which is the smallest possible enclosing polytope, in order to reduce the number of points in the target index space. This would cause more minimum and maximum expressions in the loop bounds of the target program and the computation of the convex hull is complex, compared with the previous method.
Figure 13 illustrates that also the convex hull is not optimal in connection with the number of index points.

At this point, we have considered program parts with unit-strides in all dimensions.
Assume that the program parts $P'_1, \ldots, P'_k$ have non-unit strides, say, $s_1^{(1)}, \ldots, s_m^{(1)}$ for $P'_1$, ... and $s_1^{(k)}, \ldots, s_m^{(k)}$ for $P'_k$.
We calculate the greatest common divisor gcd of all strides of the dimension $r$. This is the $r$-th stride of the target program:

$$s_r = \gcd\left( s_r^{(t)} \mid t \in \{1, \ldots, k\}\right)$$

If there is a $t \in \{1, \ldots, k\}$ such that $s_r^{(t)}$ does not exist, it is not taken into account.
Let $l_r^{(t)}$ be the lower bound of the $r$-th dimension of the $t$-th program. Note that $l_r^{(t)}$ can contain a minimum or maximum expression.
In the if-statement preceding the loop body, we have to insert the following

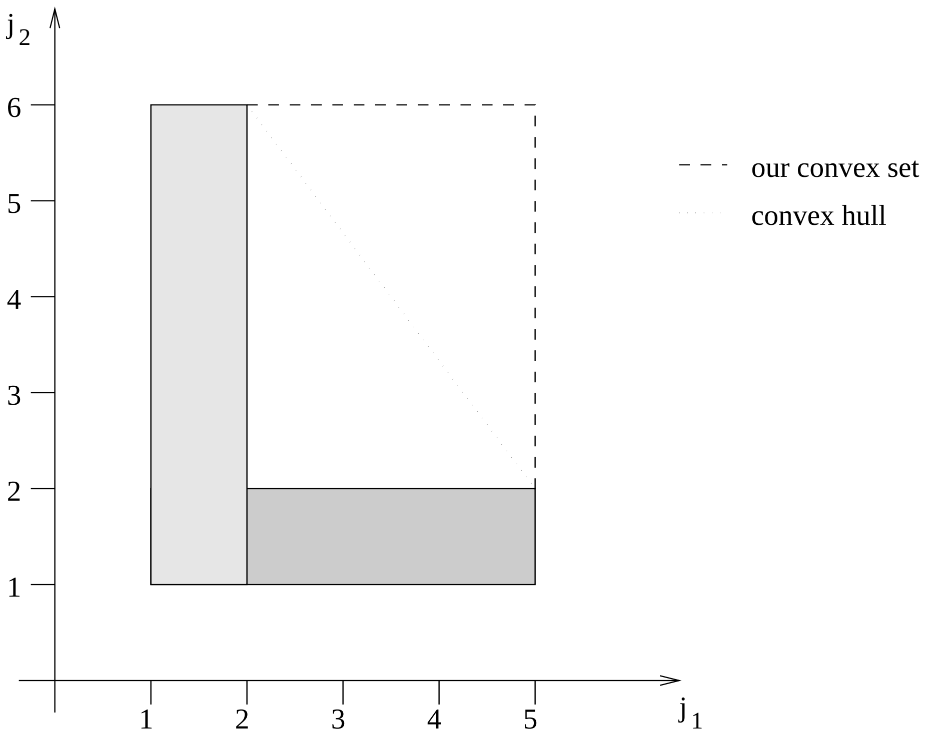



Figure 13: Comparison between our convex set and the convex hull including the quasi-convex polytope $P_1 \cup P_2$

new expression in order to determine whether the iteration point is an element of the lattice points of $P_t$:

$$\left(\left(j_1 - l_1^{(t)}\right)\ \%\frac{s_1^{(t)}}{s_1} = 0\right) \wedge \ldots \wedge\ \left(\left(j_1 - l_n^{(t)}\right)\ \%\frac{s_n^{(t)}}{s_n} = 0\right)$$

Let us consider an example:

$P_1'$: **for** $j_1 = 2$ **to** $10$ **step** $3$
    **parfor** $j_2 = 1$ **to** $5$ **step** $2$
        $S_1$
    **end**
**end**

$P_2'$: **for** $j_1 = 5$ **to** $11$ **step** $4$
  **parfor** $j_2 = 2$ **to** $6$ **step** $4$
    $S_2$
  **end**
**end**

The index space and the iteration points of these polytopes are shown in Figure 14.

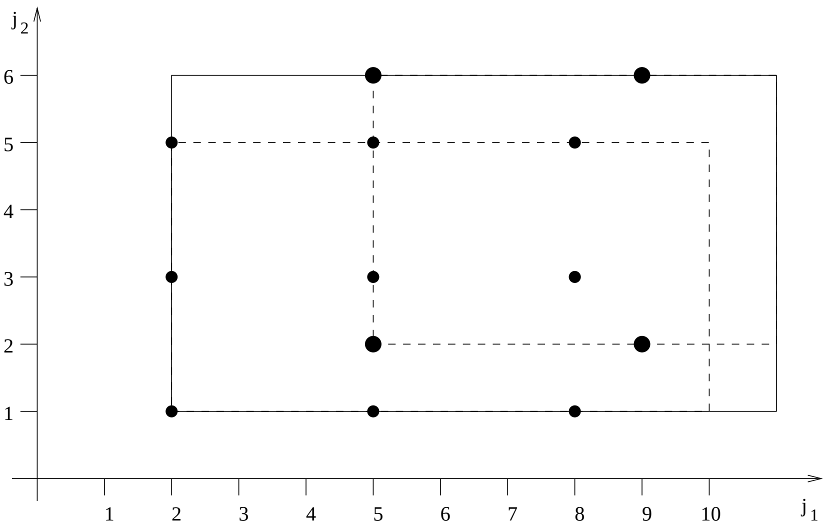



Figure 14: Polytopes of the example

Then, the run-time solution for the target program is:

**for** $j_1 = 2$ **to** $11$ **step** $\gcd(3,4)$
  **parfor** $j_2 = 1$ **to** $5$ **step** $\gcd(2,4)$
    **if** $\left(2 \le j_1 \le 10 \wedge 1 \le j_2 \le 5 \wedge (j_1 - 2)\%\frac{3}{1} = 0 \wedge (j_2 - 1)\%\frac{2}{2} = 0\right)$
      $S_1$
    **fi**
    $\|$

**if** $\left(5 \leq j_1 \leq 11 \wedge 2 \leq j_2 \leq 6 \wedge (j_1 - 5)\%\frac{4}{1} = 0 \wedge (j_2 - 2)\%\frac{4}{2} = 0\right)$
    $S_2$
**fi**
**end**
**end**

Finally, let us explain why the same dimensionality of all schedules (and allocations) is obligatory. Different dimensionalities lead to varying orders of sequential and parallel for-loops in the target program parts. As a consequence, we would have to merge for example a time loop of one program part with a processor loop of another part.

Example:
The program parts:

**for** $j_1 = l_1^{(1)}$ **to** $u_1^{(1)}$
    **for** $j_2 = l_2^{(1)}$ **to** $u_2^{(1)}$
        $S_1$
    **end**
**end**

and

**for** $j_1 = l_1^{(2)}$ **to** $u_1^{(2)}$
    **parfor** $j_2 = l_2^{(2)}$ **to** $u_2^{(2)}$
        $S_2$
    **end**
**end**

cannot be merged in the second dimension because we would have to combine the sequential loop of program part 1 with the parallel loop of part 2. The different types of loops in the second dimension result from a two-dimensional schedule for $S_1$ and a one-dimensional schedule for $S_2$.

**4.4.2.2 Alternatives** There exist some alternatives to the run-time solution. These alternatives differ for synchronous and asynchronous target programs and are described in the subsequent sections.

**4.4.2.2.1 Parallel Execution Operator Solution for Asynchronous Programs** A very simple method for merging the target program parts is the parallel execution operator solution.
In an asynchronous target program, every distinct program part has its own time dimension and, therefore, the program parts can be executed in parallel.
Example:

$P_1'$: **parfor** $j_1 = 1$ **to** $5$
    **for** $j_2 = j_1$ **to** $j_1 + 10$
        $S_1$
    **end**
**end**

$P_2'$: **parfor** $j_1 = 2$ **to** $4$
    **for** $j_2 = 5$ **to** $8$
        $S_2$
    **end**
**end**

The union of $P_1'$ and $P_2'$ yields the target program:

$P_1'$
$||$
$P_2'$

Note that the job of merging is left to the implementation of the parallel execution operator.

**4.4.2.2.2 Compile-Time Solution** As we have seen in the run-time solution, there are several iteration points enumerated by the target program where no statement is executed. These points are skipped with the help of if-then-else statements.

To avoid this program overhead, the compile-time solution divides the quasi-convex target polytope into several proper polytopes at compile time, which can be transformed directly into loop nests.
Note that the compile-time solution can be applied to synchronous or asynchronous programs. Therefore, the loops in the program parts are always denoted by **for**. We assume that the number of schedules and allocations is equal in every program part for the same reasons as in the run-time solution.
Before considering an example, we define $\mathcal{L}_\mu^{(q)}$ ($\mu \in \{1,\ldots,k\}$, $q \in \{0,\ldots,dim\{P'_\mu\}\}$) as the abbreviation for the rest of the program $P'_\mu$ after discarding the $q$ outermost loops.

Example 1:
Let the target program be represented by the quasi-convex polytope of Figure 15 which is composed of the program parts:

$P'_1$: **for** $j_1 = 1$ **to** $5$
        **for** $j_2 = 2$ **to** $4$
            $S_1$
        **end**
    **end**
$P'_2$: **for** $j_1 = 2$ **to** $4$
        **for** $j_2 = 1$ **to** $6$
            $S_2$
        **end**
    **end**

The first dimension can be divided into three parts:

- $1 \leq j_1 < 2$: only $S_1$ has to be executed,
- $2 \leq j_1 \leq 4$: $S_1$ and $S_2$ have to be executed, sometimes simultaniously (to be precised later on),
- $4 < j_1 \leq 5$: only $S_1$ has to be executed.

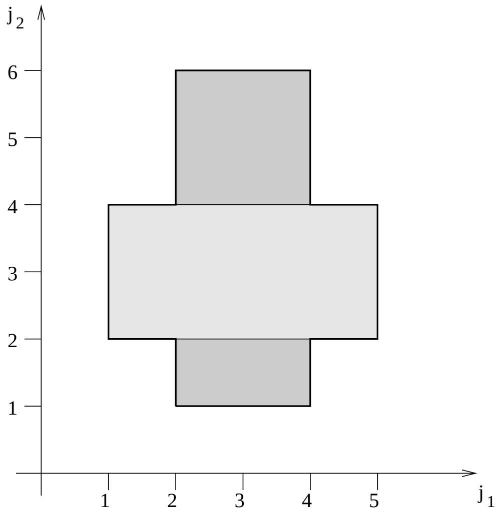



Figure 15: Polytope of the Example 1

The loop nests for the first and the third part are easy to construct:

**for** $j_1 = 1$ **to** $2 - 1$
$\mathcal{L}_1^{(1)}$
**end**

and

**for** $j_1 = 4 + 1$ **to** $5$
$\mathcal{L}_1^{(1)}$
**end**

Looking at the second part, we have to divide the second dimension, too:

- $1 \leq j_2 < 2$: only $S_2$ must be executed,
- $2 \leq j_2 \leq 4$: $S_1$ and $S_2$ have to be executed simultaniously,
- $4 < j_2 \leq 6$: only $S_2$ has to be executed.

Therefore, the loop nest is defined by:

$$
\begin{array}{l}
\textbf{for } j_1 = 2 \textbf{ to } 4 \\
\quad \textbf{for } j_2 = 1 \textbf{ to } 2-1 \\
\quad\quad S_2 \\
\quad \textbf{end} \\
\quad \textbf{for } j_2 = 2 \textbf{ to } 4 \\
\quad\quad S_1 \| S_2 \\
\quad \textbf{end} \\
\quad \textbf{for } j_2 = 4+1 \textbf{ to } 6 \\
\quad\quad S_2 \\
\quad \textbf{end} \\
\textbf{end}
\end{array}
$$

where $S_1 \| S_2$ denotes the simultanious execution of the statements $S_1$ and $S_2$ because $S_1$ and $S_2$ are mapped to the same iteration points in the target polytope.
Combining these loop nests, the target program of Example 1 is given by:

$$
\begin{array}{l}
\textbf{for } j_1 = 1 \textbf{ to } 2-1 \\
\quad \mathcal{L}_1^{(1)} \\
\textbf{end} \\
\textbf{for } j_1 = 2 \textbf{ to } 4 \\
\quad \textbf{for } j_2 = 1 \textbf{ to } 2-1 \\
\quad\quad S_2 \\
\quad \textbf{end} \\
\quad \textbf{for } j_2 = 2 \textbf{ to } 4 \\
\quad\quad S_1 \| S_2 \\
\quad \textbf{end} \\
\quad \textbf{for } j_2 = 4+1 \textbf{ to } 6 \\
\quad\quad S_2 \\
\quad \textbf{end} \\
\textbf{end} \\
\textbf{for } j_1 = 4+1 \textbf{ to } 5 \\
\quad \mathcal{L}_1^{(1)} \\
\textbf{end}
\end{array}
$$

For the development of the code generation algorithm we introduce the notion of *limit points* of the $k$-th dimension as the minimum and maximum

coordinates of the projection of the polytopes to the $k$-th axis. The formal definition of the limit points is presented later on.
Note that the complete target program can be derived with the help of the limit points and the sets of polytopes which overlap between two successive limit points of every dimension.
In the previous example, the limit points of the first dimension are given by $1, 2, 4$ and 5 and the sets of polytopes which overlap between two successive limit points are: in 1 only $P_1$, between 2 and 4 $P_1$ and $P_2$, and in 5 only $P_1$.
When dealing with constant bounds, as in Example 1, it is very easy to determine the limit points and the sets of overlapping polytopes. A problem appears when having structure parameters and minimum/maximum expressions in loop bounds. It is not obvious any more, which of the polytopes do overlap and where. For this to happen, minimum and maximum expressions must have at least one parameter or index argument, otherwise they can be reduced to a constant bound.

Example 2:
Consider the two program parts $P_1'$ and $P_2'$ which can be represented by the polytopes in Figure 16:

$P_1'$: **for** $j_1 = 1$ **to** $n$
  $\mathcal{L}_1^{(1)}$
**end**
$P_2'$: **for** $j_1 = 3$ **to** $5$
  $\mathcal{L}_2^{(1)}$
**end**

Note that we do not know the concrete right bound of polytope $P_1$ at compile time and, therefore, the determination of the sets containing the overlapping polytopes is impossible.
To solve this problem, we enumerate all possible permutations of the loop bounds. In this example, $n$ can be either between 1 and 3 or between 3 and 5 or greater than 5 where the non-determinism at the points 3 and 5 is avoided by a special order. For every permutation, we develop one target program. Then, it is decided at run time which of the programs has to be executed.

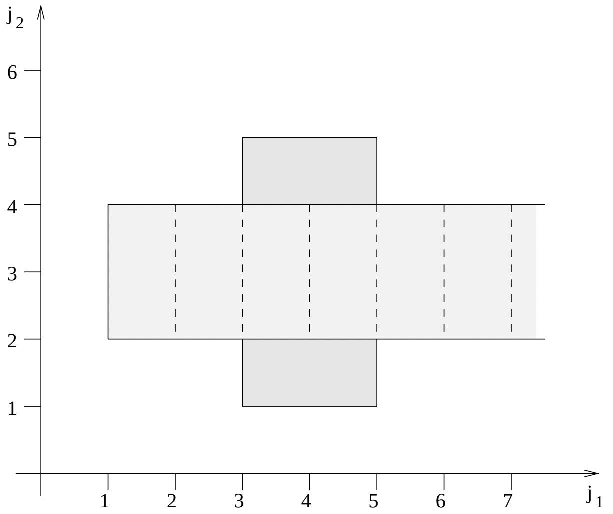



Figure 16: Polytope of the Example 2

Let us generalize this example.
In order to construct the target program, a function *merge* is introduced, which merges recursively the target program parts. The arguments of merge are the programs $\mathcal{L}_1^{(q)}, \ldots, \mathcal{L}_k^{(q)}$ which are given by:

$\mathcal{L}_1^{(q)}$: **for** $j_1 = l_1$ **to** $u_1$
$\quad\quad\mathcal{L}_1^{(q+1)}$
**end**

$\vdots$

$\mathcal{L}_k^{(q)}$: **for** $j_1 = l_k$ **to** $u_k$
$\quad\quad\mathcal{L}_k^{(q+1)}$
**end**

The complete target program is constructed by merge$(\mathcal{L}_1^{(0)}, \ldots, \mathcal{L}_k^{(0)})$ = merge$(P'_1, \ldots, P'_k)$, where $P'_1, \ldots, P'_k$ are the target program parts created by the algorithms of Section 4.3. Moreover we consider the part $\mathcal{L}_x^{(q)}$ $(x \in \{1, \ldots, k\})$ which has no surrounding loops a statement which has to be executed in parallel with the loop nests constructed in this step.

In order to define the limit points, we need a multiset $\mathcal{N}$ which consists of all lower and upper bounds of the $q$-th dimension of all programs$P_1', \ldots, P_k'$:

$$\mathcal{N} = \{l_1, u_1, \ldots, l_k, u_k\}.$$

Then, a limit point is formally defined by $\alpha_\mu \in \mathcal{N} \times \{0,1\}$ ($\mu \in \{1, \ldots, 2k\}$). The tuple represents one loop bound of $\mathcal{N}$. The first element of the pair $\alpha_\mu$ is the value of the corresponding bound and the second is a boolean value which is 0 if $\alpha_\mu$ is a lower bound and 1 otherwise.
The projection to the first and to the second coordinate is given by the functions

$$\sigma_1 : N \times \{0,1\} \to N$$

and

$$\sigma_2 : N \times \{0,1\} \to \{0,1\}$$

Besides, we define an order $\preceq$ between every two points $\alpha$ and $\alpha'$ of $\mathcal{N} \times \{0,1\}$:

$$\begin{aligned} &\alpha \preceq \alpha' \Longleftrightarrow \\ &\qquad \sigma_1(\alpha) < \sigma_1(\alpha') \vee \\ &\qquad (\sigma_1(\alpha) = \sigma_1(\alpha') \wedge \neg\, (\sigma_2(\alpha) = 1 \wedge \sigma_2(\alpha') = 0)) \end{aligned}$$

For the determination of the order of all limit points in the quasi-convex target polytope, we permute $\{\alpha_1, \ldots, \alpha_{2k}\}$. The set of all permutations is denoted by $\mathcal{P}$.

Let $(\widehat{\alpha}_1, ..., \widehat{\alpha}_{2k})$ be a permutation of $\mathcal{P}$. It is called *valid*, if $\widehat{\alpha}_1 \preceq \ldots \preceq \widehat{\alpha}_{2k}$ is true. A target program is constructed for every permutation and is executed if this permutation is valid at run time. Therefore, we utilize nested if-then-else statements where the condition is the check of validity of one permutation and the body is defined by the corresponding target program.
Note that we have non-determinism in the valid permutations if $\mathcal{N}$ is a multiset.

Assume that the permutation $(\widehat{\alpha}_1, ..., \widehat{\alpha}_{2k})$ is valid. Thus, the order of the limit points is known. Then, we can introduce a set $K_\mu$ collecting the indices of the polytopes which overlap between $\widehat{\alpha}_\mu$ and $\widehat{\alpha}_{\mu+1}$ ($\mu \in \{1, \ldots, 2k - 1\}$). If the polytope $P_m$ is 'starting' at the point $\widehat{\alpha}_\mu$, i.e., $\sigma_1(\widehat{\alpha}_\mu)$ is the lower bound of $P'_m$, we have to add $m$ to the set $K_{\mu-1}$ in order to determine $K_\mu$. In analogy, we have to remove $m$ from $K_{\mu-1}$, if $\sigma_1(\widehat{\alpha}_\mu)$ is the upper bound of $P'_m$.
Formally:

$$K_0 = \{\}$$

$$K_\mu = \begin{cases} K_{\mu-1} \cup \{m\} & \text{, if } \sigma_1(\widehat{\alpha}_\mu) = l_m \\ K_{\mu-1} \backslash \{m\} & \text{, if } \sigma_1(\widehat{\alpha}_\mu) = u_m \end{cases}$$

To obtain a program from the collective information as there is the limit points $\widehat{\alpha}_1, ..., \widehat{\alpha}_{2k}$ and the sets $K_\mu$ representing the polytopes which overlap between $\widehat{\alpha}_\mu$ and $\widehat{\alpha}_{\mu+1}$, we must know whether the limit point $\widehat{\alpha}_\mu$ belongs to loop nest $\mu - 1$ or $\mu$.
If at least one polytope 'ends' at the point $\sigma_1(\alpha_\mu)$, formally $K_{\mu-1} \not\subset K_\mu$, $\sigma_1(\alpha_\mu)$ is element of the upper bound of loop nest $\mu - 1$. Otherwise, a new polytope 'starts' at the point $\sigma_1(\alpha_\mu)$, formally $K_{\mu-1} \subset K_\mu$ and, therefore, $\sigma_1(\alpha_\mu)$ is the lower bound of loop nest $\mu$.
The lower bound of the first loop nest is always $\sigma_1(\alpha_1)$ and the upper bound of the last loop nest is always $\sigma_1(\alpha_{2k})$.
The lower bound of one loop nest (except the first) is the upper bound of the previous loop nest plus 1.

Inside one loop nest $\mu$, we have to merge all bodies $\mathcal{L}_\nu^{(q+1)}$, where $\nu \in K_\mu$.

The result of merging $\mathcal{L}_1^{(q)}, \ldots, \mathcal{L}_k^{(q)}$ for a valid permutation $(\widehat{\alpha}_1, ..., \widehat{\alpha}_{2k})$ is depicted in Figure 17.
Let us consider two examples, one where two polytopes overlap and the second where three polytopes touch each other at one point.

merge($\mathcal{L}_1^{(q)}, \ldots, \mathcal{L}_k^{(q)}$) =

**for** $j_1 = \widehat{\alpha}'_1$ **to** $\widehat{\alpha}'_2$
    $merge\left\{\mathcal{L}_y^{(q+1)} | y \in K_1\right\}$
**end**
**for** $j_1 = \widehat{\alpha}'_2 + 1$ **to** $\widehat{\alpha}'_3$
    $merge\left\{\mathcal{L}_y^{(q+1)} | y \in K_2\right\}$
**end**
$\vdots$
**for** $j_1 = \widehat{\alpha}'_{2k-1} + 1$ **to** $\widehat{\alpha}_{2k}$
    $merge\left\{\mathcal{L}_y^{(q+1)} | y \in K_{2k-1}\right\}$
**end**

where $\widehat{\alpha}'_\mu = \begin{cases} \sigma_1(\widehat{\alpha}_\mu) - 1 & , \quad K_{\mu-1} \subset K_\mu \\ \sigma_1(\widehat{\alpha}_\mu) & , \quad K_{\mu-1} \not\subset K_\mu \end{cases}$

for all $\mu \in \{2, \ldots, 2k-1\}$

Figure 17: Result of the merging function for a valid permutation $(\widehat{\alpha}_1, ..., \widehat{\alpha}_{2k})$

Example 3:
There are two polytopes $P_1$ and $P_2$. Say the permutation $(l_1, l_2, u_1, u_2)$ is valid. Figure 18 shows one possible arrangement.

Now we can determine the sets $K_\mu$:

$$\begin{array}{ll} K_1 = K_0 \cup \{1\} = \{1\} & \text{because } \sigma_1(\widehat{\alpha}_1) = l_1 \\ K_2 = K_1 \cup \{2\} = \{1, 2\} & \text{because } \sigma_1(\widehat{\alpha}_2) = l_2 \\ K_3 = K_2 \backslash \{1\} = \{2\} & \text{because } \sigma_1(\widehat{\alpha}_3) = u_1 \end{array}$$

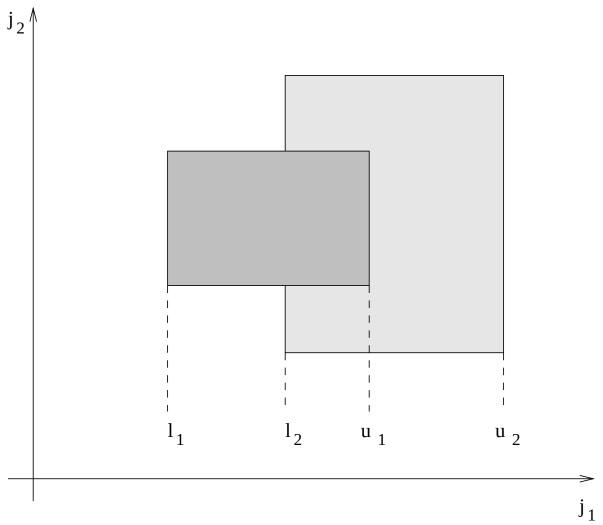



Figure 18: Polytopes of Example 3

Thus, the target program is given by:

**for** $j_1 = l_1$ **to** $l_2 - 1$
    $\mathcal{L}_1^{(1)}$    $//K_1 \subset K_2$
**end**
**for** $j_1 = l_2$ **to** $u_1$
    merge$(\mathcal{L}_1^{(1)}, \mathcal{L}_2^{(1)})$    $//K_2 \not\subset K_3$
**end**
**for** $j_1 = u_1 + 1$ **to** $u_2$
    $\mathcal{L}_2^{(1)}$
**end**

Example 4:
There are three polytopes $P_1$ to $P_3$. Say, the permutation $(l_1, l_2, l_3, u_1, u_2, u_3)$ is valid, where $l_2 = l_3 = u_1$. Figure 19 depicts one possible arrangement. We can determine the sets $K_\mu$:

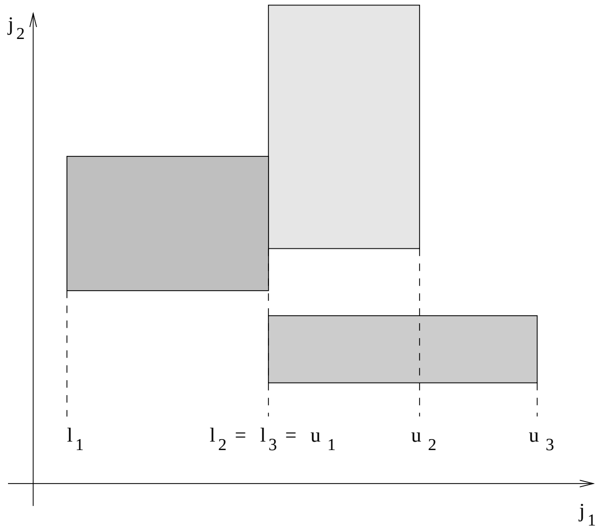



Figure 19: Polytopes of Example 4

| | |
|---|---|
| $K_1 = \{1\}$ | because $\sigma_1(\widehat{\alpha}_1) = l_1$ |
| $K_2 = K_1 \cup \{2\} = \{1,2\}$ | because $\sigma_1(\widehat{\alpha}_2) = l_2$ |
| $K_3 = K_2 \cup \{3\} = \{1,2,3\}$ | because $\sigma_1(\widehat{\alpha}_3) = l_3$ |
| $K_4 = K_3 \backslash \{1\} = \{2,3\}$ | because $\sigma_1(\widehat{\alpha}_4) = u_1$ |
| $K_5 = K_4 \backslash \{2\} = \{3\}$ | because $\sigma_1(\widehat{\alpha}_5) = u_2$ |

Thus, the target program is given by:

**for** $j_1 = l_1$ **to** $l_2 - 1$
    $\mathcal{L}_1^{(1)}$ $//K_1 \subset K_2$
**end**
**for** $j_1 = l_2$ **to** $l_3 - 1$
    merge$(\mathcal{L}_1^{(1)}, \mathcal{L}_2^{(1)})$ $//K_2 \subset K_3$
**end**
**for** $j_1 = l_3$ **to** $u_1$
    merge$(\mathcal{L}_1^{(1)}, \mathcal{L}_2^{(1)}, \mathcal{L}_3^{(1)})$ $//K_3 \not\subset K_4$
**end**

**for** $j_1 = u_1 + 1$ **to** $u_2$
  merge$(\mathcal{L}_2^{(1)}, \mathcal{L}_3^{(1)})$  $//K_4 \not\subset K_5$
**end**
**for** $j_1 = u_2 + 1$ **to** $u_3$
  $\mathcal{L}_3^{(1)}$
**end**

Note that the second loop nest can be omitted, because $l_2 = l_3$, i.e., the loop is empty.
In this example, we can see why it is important to have the order $\preceq$ for bounds which are equal. Calculate this example when exchanging, for example, $l_3$ and $u_1$. The result will be incorrect!

Up to now, we did not take care of polytopes which have non-unit strides in several or all dimensions. In analogy to the run-time solution, we assume that $s_1^{(1)}, \ldots, s_m^{(1)}$ are the strides of $P_1'$, ... and $s_1^{(k)}, \ldots, s_m^{(k)}$ of $P_k'$, where $m$ is the dimensionality of the program parts.
We merge all program parts as described above, but surround every statement $S_t$ $(t \in \{1, \ldots, k\})$ by an if-statement:

**if** $\left(\left(j_1 - l_1^{(t)}\right) \%s_1^{(t)} = 0 \wedge \ldots \wedge \left(j_m - l_m^{(t)}\right) \%s_m^{(t)} = 0\right)$
  $S_t$
**fi**

where $j_1, \ldots j_m$ are the loop indices and $l_r^{(t)}$ $(r \in \{1, \ldots, m\})$ is the lower bound of the $r$-th dimension of program $P_t'$.
Note that the strides of all loop nests in the complete target program are equal to 1.

Example 5:
Consider the following program parts:

$P'_1$: **for** $j_1 = 5$ **to** $10$ **step** $3$
  **for** $j_2 = 4$ **to** $7$ **step** $1$
    $S_1$
  **end**
**end**

$P'_2$: **for** $j_1 = 5$ **to** $10$ **step** $4$
  **for** $j_2 = 2$ **to** $10$ **step** $2$
    $S_2$
  **end**
**end**

The corresponding polytopes are shown in Figure 20. The dots represent the index points of polytope $P_1$ and the crosses these of $P_2$.

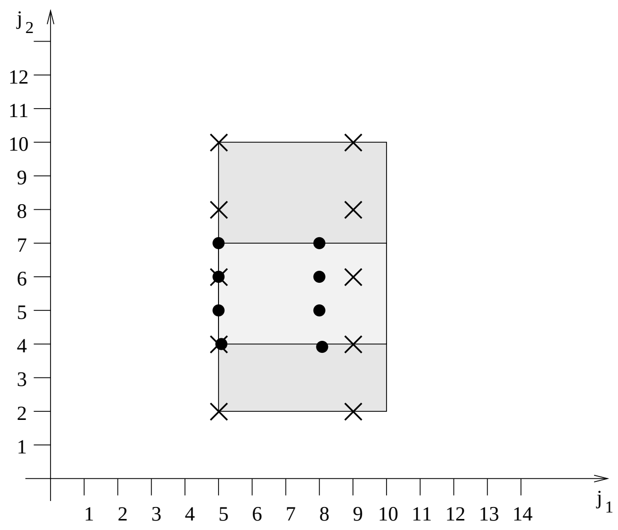



Figure 20: Polytopes of Example 5

The complete target program is given by:

$$
\begin{array}{l}
\textbf{for } j_1 = 5 \textbf{ to } 10 \textbf{ step } 1 \\
\quad \textbf{for } j_2 = 2 \textbf{ to } 3 \textbf{ step } 1 \\
\quad\quad \textbf{if}((j_1 - 5)\,\%4 = 0 \wedge (j_2 - 2)\,\%2 = 0) \\
\quad\quad\quad S_2 \\
\quad\quad \textbf{fi} \\
\quad \textbf{end} \\
\quad \textbf{for } j_2 = 4 \textbf{ to } 7 \textbf{ step } 1 \\
\quad\quad \textbf{if}((j_1 - 5)\,\%3 = 0 \wedge (j_2 - 4)\,\%1 = 0) \\
\quad\quad\quad S_1 \\
\quad\quad \textbf{fi} \\
\quad\quad \| \\
\quad\quad \textbf{if}((j_1 - 5)\,\%4 = 0 \wedge (j_2 - 2)\,\%2 = 0) \\
\quad\quad\quad S_2 \\
\quad\quad \textbf{fi} \\
\quad \textbf{end} \\
\quad \textbf{for } j_2 = 8 \textbf{ to } 10 \textbf{ step } 1 \\
\quad\quad \textbf{if}((j_1 - 5)\,\%4 = 0 \wedge (j_2 - 2)\,\%2 = 0) \\
\quad\quad\quad S_2 \\
\quad\quad \textbf{fi} \\
\quad \textbf{end} \\
\textbf{end}
\end{array}
$$

Note that we only had one valid permutation because the bounds of all program parts in all dimensions are constant.

# 5 Conclusion

Let us first comment on our implementation and then suggest some improvements.

We encountered no problem in the generation and transformation of the program parts. The only difficulties appeared when merging the target program parts.
As explained in Section 4.4.2.1, the run-time solution specifies some iteration points where no actions have to take place and, therefore, the target program is inefficient. We became aware of this disadvantage and developed another merging strategy: the compile-time solution where the index space of the quasi-convex polytope is divided into several parts.
When implementing this solution, we discovered a new problem already with very simple programs: the target program can become very large and the generation can take long time. In an example, the merging of four program parts had run for about 20 minutes and the target program had grown to approximately 130 MB of disk space before we interrupted the generation process. The reason for this excessive size is the necessity to compute all permutations of the loop bounds in every dimension. In the example, we had 8 loop bounds, thus, we generated 40320 permutations, i.e., also target programs which we had to combine by if-then-else statements.
As a consequence, we tried to improve the compile-time solution by eliminating as many permutations as possible. This is only practicable if we have many constant bounds. But: the amount of space increases with the number of structure parameters and minimum/maximum expressions in the loop bounds of one dimension.

Thus, we implemented a compile-time merge only for the first dimension. All other dimensions are merged with the help of the run-time solution. Applying the compile-time solution for all dimensions would result in an exponential increase in the number of permutations and would also increase the length of the program.
In contrast to the compile-time solution, the run-time solution always provides readable target programs with the disadvantage of enumerating more iteration points than necessary and deciding at run time which of the statements have to be executed.

We see the following possibilities to improve the target code generation in the future:

1. During the extension of the basis of the transformation matrix (Section 4.3.2), we insert a unit row for every missing row. It should be examined whether an insertion of a non-identity row will be of advantage.

2. Up to now, either a synchronous or an asynchronous target program can be generated. It may be interesting to allow any nesting order of parallel and sequential loops in the future.

3. In our target programs, array indices consist of affine functions (i.e., $a\,[j_1 + 2j_2 + 5n_1 + 10]$). An algorithm presented by Jean-François Collard [3] can probably be applied to our programs in order to simplify the expressions of array indices.

# References

[1] U. Banerjee. *Loop Transformations for Restructuring Compilers: The Foundations.* Kluwer, 1993.

[2] G.H. Bradley. Algorithms for Hermite and Smith normal matrices and linear diophantine equations. *Mathematics of Computation*, 25(116):897–907, 1971.

[3] J.-F. Collard. Code generation in automatic parallelizers. In C. Girault, editor, *Proc. Int. Conf. on Applications in Parallel and Distributed Computing, IFIP W.G. 10.3*, pages 185–194. North-Holland, April 1994.

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

[5] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part I. One-dimensional time. *Int. J. Parallel Programming*, 21(5):313–348, October 1992.

[6] P. Feautrier. Some efficient solutions to the affine scheduling problem. Part II. Multidimensional time. *Int. J. Parallel Programming*, 21(6):389–420, October 1992.

[7] R.S. Garfinkel and G.L. Nemhauser. *Integer Programming.* John Wiley & Sons, 1972.

[8] G. Grosche. *Projektive Geometrie I, II.* Teubner, 1957.

[9] D. Hearn and M.P. Baker. *Computer Graphics.* Prentice-Hall, 1986.

[10] H. Keimer. Vergleich von Verfahren zur Datenabhängigkeitsanalyse. Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, to appear, 1996.

[11] W. Kelly, W. Pugh, and E. Rosser. Code generation for multiple mappings. Technical Report CS-TR-3317, Dept. of Computer Science, Univ. of Maryland, 1994.

[12] C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, Lecture Notes in Computer Science 715, pages 398–416. Springer-Verlag, 1993.

[13] E.A. Maxwell. *General Homogeneous Coordinnates in Space of Three Dimensions*. Cambridge Univ. Press, 1951.

[14] A. Schrijver. *Theory of Linear and Integer Programming.* Series in Discrete Mathematics. John Wiley & Sons, 1986.

[15] V. Weispfenning. Parametric Linear and Quadratic Optimization by Elimination. Technical report, Fakultät für Mathematik und Informatik, Universität Passau, 1994.

[16] C. Wieninger. Automatische Methoden zur Parallelisierung im Polyedermodell. Diplomarbeit, Fakultät für Mathematik und Informatik, Universität Passau, 1995.

[17] J. Xue. Automating non-unimodular transformations of loop nests. *Parallel Computing*, 20(5):711–728, May 1994.