

# Note for *More Legal Transformations for Locality*

Ying Cao

March 3, 2020

## Contents

<b>1</b>	<b>What problem is this paper going to solve?</b>	<b>2</b>
<b>2</b>	<b>Background</b>	<b>2</b>
<b>3</b>	<b>Affine transformations for Locality</b>	<b>2</b>
3.1	Formulation . . . . .	2
3.2	An example . . . . .	3
3.3	How dependences can be expressed exactly using linear (in)equalities ? . . . . .	3
3.4	Legal Transformation Space . . . . .	4
3.5	Properties . . . . .	5
<b>4</b>	<b>Finding Legal Transformations</b>	<b>5</b>
<b>5</b>	<b>Some useful reading</b>	<b>5</b>

# 1 What problem is this paper going to solve?

Exploiting data locality is one of the keys to achieve high performance level. The basic framework for increasing the cache hit rates aims at *moving references to a given memory cell (or cache line) to neighboring iterations of some innermost loop*. Such a transformation modifies the operation execution order, the existence of a good solution highly depends on data dependences.

## 2 Background

1. When loop bounds and conditionals only depend on surrounding loop counters, formal parameters and constants, the iteration domain can always be specified by a set of linear inequalities defining a polyhedron.
  - (a) The term *polyhedron* will be used in a broad sense to denote a *convext set of points in a lattice* (also called  $\mathbb{Z}$ -polyhedron or lattice-polyhedron).
2. Each statement may include **one or several** references to arrays.
3. The subscript function  $f(x)$  maps iteration vectors to array elements. When the subscript function  $f(x)$  of a reference is affine, we can write it as  $f(x) = Fx + a$  where  $F$  is called the *subscript matrix*.

For the example below, the **reference to the array**  $B$  is  $B[f(x)]$  with  $f\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 1 & 1 \\ 2 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}$

```
1  for(int i = 1; i < n; ++i){
2      for(int j = 1; j < n; ++j){
3          if(i <= n + 2 - j)
4              S1:  B[i + j][2 * i + 1] = ...
5          }
6      }
```

## 3 Affine transformations for Locality

### 3.1 Formulation

The goal of a transformation is to modify the original execution order of the operations. To do this, the optimizing compilers build schedules at the statement level by finding a function which specify an execution time for each instance of the corresponding statement.

The schedule functions are usually choosen affine, thus have the following shape:

$$\theta_S(x_S) = T_S x_S + t_S \quad (1)$$

1.  $x_S$  is the iteration vector.
2.  $T_S$  is a constant transformation matrix.
3.  $t_S$  is a constant vector possibly including affine parameteric expresseoins using the structure parameters of the program (i.e. the symbolic constants, mostly array sizes or iteration bounds).

Linear transformations can express most of the useful transformations.

### 3.2 An example

Consider an original polyhedron defined by the system  $Ax + c \geq 0$ , and the transformation function  $\theta$  leads to the target index  $y = Tx$ , then we can deduce that the transformed polyhedron can be defined by:

$$(AT^{-1})y + c \geq 0$$

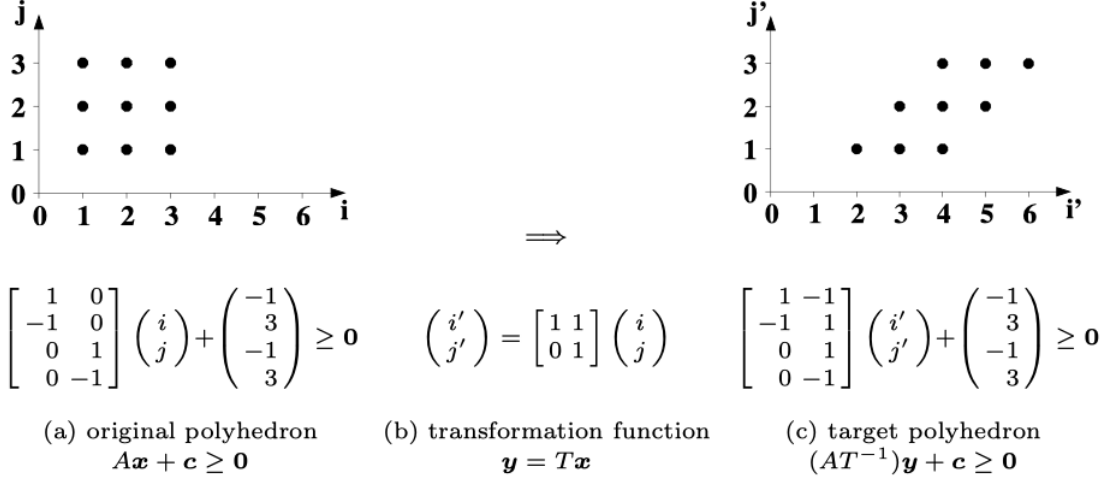


Figure 1: a skewing transformation

### 3.3 How dependences can be expressed exactly using linear (in)equalities ?

A convenient way to represent the scheduling constraints between the program operations is the **Dependence Graph**. In this directed graph:

1. **Each** program statement is represented using a unique vertex;
2. The existing dependence relations between statement instances are represented using edges.
  - **Each vertex is labelled with the iteration domain** of the corresponding statement
  - **Each edges is labelled with the dependence polyhedra** which describes the dependence relation between the source and destination statements.

A statement  $R$  **depends** on a statement  $S$  (written  $S\delta R$ ) if there exists an operation  $S(x_1)$ , an operation  $R(x_2)$  and a memory location  $m$  such that:

1.  $S(x_1)$  and  $R(x_2)$  refer the same memory location  $m$ , and at least one of them writes to that location.
2.  $x_1$  and  $x_2$  respectively belong to the iteration domain  $S$  and  $R$ ;
3. in the original sequential order,  $S(x_1)$  is executed before  $R(x_2)$ .

From this definition, it is easy to describe the **dependence polyhedra** of **each** dependence relation between two statements with affine (in)equalities. Such a dependence system has following components:

1. *Same memory location*:  $F_S x_S + a_S = F_R x_R + a_R$ ;
2. *Iteration domains*:  $A_S x_S + c_S \geq 0$  and  $A_R x_R + c_R \geq 0$ ;
3. *Precedence order*:  $P_S x_S - P_R x_R + b \geq 0$ ;
  - (a) this constraint can be separated into a disjunction of as many parts as there are common loops to both  $S$  and  $R$ .
  - (b) each case corresponds to a common loop depth, and is called *dependence level*.

<i>SCoP decomposition</i>	
<i>for</i> (i=1; i<=N; i++)	
S1	SCoP 1 of depth 0
-----	
<i>for</i> (j=1; j<=i*i; j++)	// Non-affine SCoP boundary: i*i
-----	
S2	SCoP 2 of depth 1
<i>for</i> (k=1; k<=j; k++)	global parameters: N,i,j
if (j >= 2) then	loop iterator: k
S3	
S4	
-----	
<i>for</i> (p=1; p<=N; p++)	SCoP 3 of depth 1
S5	global parameters: N
S6	loop iterator: p

Figure 2: SCoP decomposition

### 3.4 Legal Transformation Space

Consider the transformations as scheduling functions, the time interval in the target program between the executions of two operations  $R(x_R)$  and  $S(x_S)$  is:

$$\Delta_{R,S} \begin{pmatrix} x_S \\ x_R \end{pmatrix} = \theta_R(x_R) - \theta_S(x_S). \quad (2)$$

If  $\mathcal{D}_{S\delta R}$  is not empty, then:

$$\Delta_{R,S} \begin{pmatrix} x_S \\ x_R \end{pmatrix} - 1 \geq 0. \quad (3)$$

The affine scheduling function can be expressed in terms of  $D$  and  $d$  by applying Farkas Lemma.

**Lemma 1.** (Affine form of Farkas Lemma [15]) *Let  $\mathcal{D}$  be a nonempty polyhedron defined by the inequalities  $Ax + b \geq 0$ . Then any affine function  $f(x)$  is nonnegative everywhere in  $\mathcal{D}$  iff it is a positive affine combination:*

$$f(x) = \lambda_0 + \Lambda^T (Ax + b), \text{ with } \lambda_0 \geq 0 \text{ and } \Lambda^T \geq 0,$$

where  $\lambda_0$  and  $\Lambda^T$  are called Farkas multipliers.

Figure 3: Affine form of Farkas Lemma.

According to this Lemma, **for each edge in the dependence graph**, we can find a positive vector  $\lambda_0$  and matrix  $\Lambda^T$  such that:

$$T_R x_R + t_R - (T_S x_S + t_S) - 1 = \lambda_0 + \Lambda^T \left( D \begin{pmatrix} x_S \\ x_R \end{pmatrix} + d \right), \lambda_0 \geq 0, \Lambda \geq 0 \quad (4)$$

### 3.5 Properties

- There is self-temporal reuse when  $x_r \in \ker F$ .
- The reuse can be exploited if the transformed iteration order follows one of the reuse directions.
- (??) *Then we have to find an orthogonal vector to the chosen reuse direction to be the first part of the transformation matrix  $T$ .*
- If this partial transformation do not violate dependences, we have many choices for the completion procedure<sup>1</sup> (??).
- To generalize, it is easy to consider not only a reuse direction vector, but a reuse direction space. in order for the transformation function to be instance-wise.

## 4 Finding Legal Transformations

TBD

## 5 Some useful reading

1. Complex transformations such as Tiling can be achieved using linearizing transformations [2].
2. there is a wide range of existing data locality improvement methods, for the single processor case as well as compiling techniques for parallel systems, a well-known method is using space-time mappings [3].

## References

- [1] M. Griebl, C. Lengauer, and S. Wetzel, “Code generation in the polytope model,” in *Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No. 98EX192)*, pp. 106–111, IEEE, 1998.
- [2] J. Xue, “On tiling as a loop transformation,” *Parallel Processing Letters*, vol. 7, no. 04, pp. 409–424, 1997.
- [3] C. Lengauer, “Loop parallelization in the polytope model,” in *International Conference on Concurrency Theory*, pp. 398–416, Springer, 1993.

---

<sup>1</sup>I do not understand what is the completion procedure here? The author refers to [1] which explains completion to a unimodular transformation matrix.