

More Legal Transformations for Locality

Cédric Bastoul¹ and Paul Feautrier²

¹ Laboratoire PRiSM, Université de Versailles Saint Quentin
45 avenue des États-Unis, 78035 Versailles Cedex, France

`cedric.bastoul@prism.uvsq.fr`

² École Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon, France

`paul.feautrier@ens-lyon.fr`

Abstract. Program transformations are one of the most valuable compiler techniques to improve data locality. However, restructuring compilers have a hard time coping with data dependences. A typical solution is to focus on program parts where the dependences are simple enough to enable any transformation. For more complex problems is only addressed the question of checking whether a transformation is legal or not. In this paper we propose to go further. Starting from a transformation with no guarantee on legality, we show how we can correct it for dependence satisfaction with no consequence on its locality properties. Generating code having the best locality is a direct application of this result.

1 Introduction

Exploiting data locality is one of the keys to achieve high performance level in most computer systems and hence one of the main challenges for optimizing compilers. 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. Let us consider for instance two accesses to the same memory cell. It seems probable that the longer the time interval between these accesses is, the higher the probability of the first reference to be evicted from the cache. Since such a transformation modifies the operation execution order, the existence of a good solution highly depends on data dependences.

To bypass the dependence problem, most of the existing methods apply only to perfect loop nests in which dependences are non-existent or have a special form (fully permutable loop nests) [18]. To enlarge their application domain some preprocessing, e.g. *loop skewing* or *code sinking*, may enable them [18,1,8]. More ambitious techniques do not lay down any requirement on dependences, but are limited to propose *solution candidates* having some locality properties then to *check* them for legality [10,5]. If the candidate is proved to violate dependences, then another candidate having less interesting properties is studied. In this paper, we present a method that goes beyond checking by adjusting an optimizing transformation for dependence satisfaction, without modifying its locality properties. This technique can be used to correct a transformation candidate as well as to replace preprocessing.

This paper is organized as follows. In section 2 is outlined the background of this work. Section 3 deals with the transformations in the polyhedral model and focuses on both their dependences constraints and locality properties. Section 4 shows how it is possible to correct a transformation for legality. Lastly, section 6 concludes and discusses future work.

2 Background and Notations

A loop in an imperative language like C or FORTRAN can be represented using a n -entry column vector called its *iteration vector*:

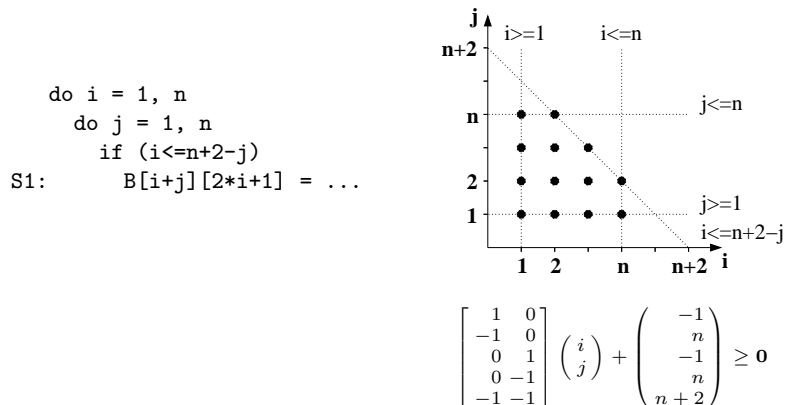
$$\mathbf{x} = \begin{pmatrix} i_1 \\ i_2 \\ \vdots \\ i_n \end{pmatrix},$$

where i_k is the k^{th} loop index and n is the innermost loop. The surrounding loops and conditionals of a statement define its *iteration domain*. The statement is executed once for each element of the iteration domain. 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 [11]. The term *polyhedron* will be used in a broad sense to denote a *convex set of points in a lattice* (also called \mathbb{Z} -polyhedron or lattice-polyhedron), i.e. a set of points in a \mathbb{Z} vector space bounded by affine inequalities [15]. A maximal set of consecutive statements in a program with such polyhedral iteration domains is called a *static control part* (SCoP) [4]. Figure 1 illustrates the correspondence between static control and polyhedral domains. Each integral point of the polyhedron corresponds to an *operation*, i.e. an instance of the statement. The notation $S(\mathbf{x})$ refers to the operation instance of the statement S with the iteration vector \mathbf{x} . The execution of the operations follows *lexicographic order*. This means in a n -dimensional polyhedron, the operation corresponding to the integral point defined by the coordinates $(a_1 \dots a_n)$ is executed before those corresponding to the coordinates $(b_1 \dots b_n)$ iff

$$\exists i, 1 \leq i < n, (a_1 \dots a_i) = (b_1 \dots b_i) \wedge a_{i+1} < b_{i+1}.$$

Each statement may include one or several *references* to arrays (or scalars, i.e. some particular cases of arrays). When the subscript function $f(\mathbf{x})$ of a reference is affine, we can write it $f(\mathbf{x}) = F\mathbf{x} + \mathbf{a}$ where F is called the *subscript matrix* and \mathbf{a} is a constant vector. For instance, the reference to the array B in figure 1(a) is $B[f(\mathbf{x})]$ with $f\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 1 & 1 \\ 2 & 0 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix}$.

In this paper, matrices are always denoted by capital letters, vectors and functions in vector spaces are not. When an element is statement-specific, it is subscripted like A_S , excepted in general formulas where all elements are statement-specific in order to avoid too heavy notations.

(a) surrounding control of S_1 (b) iteration domain of S_1 **Fig. 1.** Static control and corresponding iteration domain

3 Affine Transformations for Locality

3.1 Formulation

The goal of a transformation is to modify the original execution order of the operations. A convenient way to express the new order is to give for each operation an execution date. However, defining all the execution dates separately would usually require very large scheduling systems. Thus optimizing compilers build schedules at the statement level by finding a function specifying an execution time for each instance of the corresponding statement. These functions are chosen affine for multiple reasons: this is the only case where we are able to decide exactly the transformation legality and where we know how to generate the target code. Thus, scheduling functions have the following shape:

$$\theta_S(\mathbf{x}_S) = T_S \mathbf{x}_S + \mathbf{t}_S, \quad (1)$$

where \mathbf{x}_S is the iteration vector, T_S is a constant transformation matrix and \mathbf{t}_S is a constant vector (possibly including affine parametric expressions using the structure parameters of the program i.e. the symbolic constants, mostly array sizes or iteration bounds).

It has been extensively shown that linear transformations can express most of the useful transformations. In particular, loop transformations (such as loop reversal, permutation or skewing) can be modeled as a simple particular case called unimodular transformations (the T_S matrix has to be square and has determinant ± 1) [2,16]. Complex transformations such as tiling [17] can be achieved using linear transformations as well [19]. These transformations modify the source polyhedra into target polyhedra containing the same points, thus with a new lexicographic order. Considering an original polyhedron defined by the system

of affine constraints $A\mathbf{x} + \mathbf{c} \geq \mathbf{0}$ and the transformation function θ leading to the target index $\mathbf{y} = T\mathbf{x}$, we deduce that the transformed polyhedron can be defined by $(AT^{-1})\mathbf{y} + \mathbf{c} \geq \mathbf{0}$ (there exists more convenient way to describe the target polyhedron as discussed in [3]). For instance, let us consider the polyhedron in figure 2(a) and the transformation function $\theta\left(\begin{smallmatrix} i \\ j \end{smallmatrix}\right) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix}$. The corresponding transformation is a well known *iteration space skewing* and the resulting polyhedron is shown in figure 2(c).

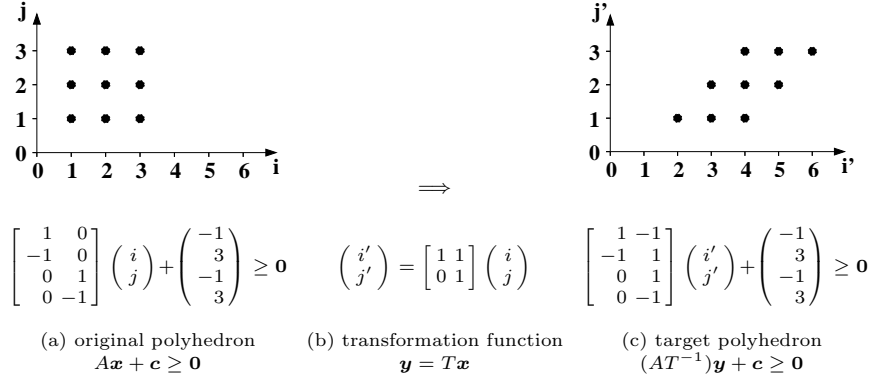


Fig. 2. A skewing transformation

3.2 Legality

It is not possible to apply any transformation to a program without changing its semantics. Statement instances reading and writing the same memory location should be kept in the same order. These operations are said to be *dependent* on each other. The dependence relations inside a program must direct the transformation construction. In this section, we recall how dependences can be expressed exactly using linear (in)equalities. Then we show how to build the legal transformation space where each program transformation has to be found.

Dependence Graph A convenient way to represent the scheduling constraints between the program operations is the *dependence graph*. In this directed graph, each program statement is represented using a unique vertex, and the existing dependence relations between statement instances are represented using edges. Each vertex is labelled with the iteration domain of the corresponding statement and the edges with the dependence polyhedra describing the dependence relation between the source and destination statements. The dependence relation can be defined in the following way:

Definition 1. A statement R **depends** on a statement S (written $S\delta R$) if there exists an operation $S(\mathbf{x}_1)$, an operation $R(\mathbf{x}_2)$ and a memory location m such that:

1. $S(\mathbf{x}_1)$ and $R(\mathbf{x}_2)$ refer the same memory location m , and at least one of them writes to that location;
2. \mathbf{x}_1 and \mathbf{x}_2 respectively belong to the iteration domain of S and R ;
3. in the original sequential order, $S(\mathbf{x}_1)$ is executed before $R(\mathbf{x}_2)$.

From this definition, we can easily describe the *dependence polyhedra* of each dependence relation between two statements with affine (in)equalities. In these polyhedra, every integral point represents a dependence between two instances of the corresponding statements. The systems have the following components:

1. *Same memory location*: assuming that m is an array location, this constraint is the equality of the subscript functions of a pair of references to the same array: $F_S\mathbf{x}_S + \mathbf{a}_S = F_R\mathbf{x}_R + \mathbf{a}_R$.
2. *Iteration domains*: both S and R iteration domains can be described using affine inequalities, respectively $A_S\mathbf{x}_S + \mathbf{c}_S \geq \mathbf{0}$ and $A_R\mathbf{x}_R + \mathbf{c}_R \geq \mathbf{0}$.
3. *Precedence order*: this constraint can be separated into a disjunction of as many parts as there are common loops to both S and R . Each case corresponds to a common loop depth and is called a *dependence level*. For each dependence level l , the precedence constraints are the equality of the loop index variables at depth lesser to l : $x_{R,i} = x_{S,i}$ for $i < l$ and $x_{R,l} > x_{S,l}$ if l is less than the common nesting level. Otherwise, there are no additional constraints and the dependence only exists if S is textually before R . Such constraints can be written using linear inequalities: $P_S\mathbf{x}_S - P_R\mathbf{x}_R + \mathbf{b} \geq \mathbf{0}$.

Thus, the dependence polyhedron for $S\delta R$ at a given level l and for a given pair of references p can be described using the following system of (in)equalities:

$$\mathcal{D}_{S\delta R,l,p} : D \begin{pmatrix} \mathbf{x}_S \\ \mathbf{x}_R \end{pmatrix} + \mathbf{d} = \begin{bmatrix} F_S & -F_R \\ A_S & 0 \\ 0 & A_R \\ P_S & -P_R \end{bmatrix} \begin{pmatrix} \mathbf{x}_S \\ \mathbf{x}_R \end{pmatrix} + \begin{pmatrix} \mathbf{a}_S - \mathbf{a}_R \\ \mathbf{c}_S \\ \mathbf{c}_R \\ \mathbf{b} \end{pmatrix} \begin{matrix} = \\ \geq \end{matrix} \mathbf{0} \quad (2)$$

There is a dependence $S\delta R$ if there exists an integral point inside $\mathcal{D}_{S\delta R,l,p}$. This can be easily checked with some linear integer programming tool like PipLib³ [6]. If this polyhedron is not empty, there is an edge in the dependence graph from the vertex corresponding to S up to the one corresponding to R and labelled with $\mathcal{D}_{S\delta R,l,p}$. For the sake of simplicity we will ignore subscripts l and p and refer in the following to $\mathcal{D}_{S\delta R}$ as the only dependence polyhedron describing $S\delta R$.

Legal Transformation Space Considering the transformations as scheduling functions, the time interval in the target program between the executions of two operations $R(\mathbf{x}_R)$ and $S(\mathbf{x}_S)$ is

$$\Delta_{R,S} \begin{pmatrix} \mathbf{x}_S \\ \mathbf{x}_R \end{pmatrix} = \theta_R(\mathbf{x}_R) - \theta_S(\mathbf{x}_S). \quad (3)$$

³ PipLib is freely available at <http://www.prism.uvsq.fr/~cedb>

If there exists a dependence $S\delta R$, i.e. if $\mathcal{D}_{S\delta R}$ is not empty, then $\Delta_{R,S} \begin{pmatrix} \mathbf{x}_S \\ \mathbf{x}_R \end{pmatrix} - \mathbf{1}$ must be a nonnegative form in $\mathcal{D}_{S\delta R}$ (intuitively, the time interval between two operations $R(\mathbf{x}_R)$ and $S(\mathbf{x}_S)$ such that $R(\mathbf{x}_R)$ depends on $S(\mathbf{x}_S)$ must be at least $\mathbf{1}$, the smallest time interval: this guarantees that the operation $R(\mathbf{x}_R)$ is executed after $S(\mathbf{x}_S)$ in the target program). This affine function can be expressed in terms of D and \mathbf{d} by applying Farkas Lemma (see Lemma 1) [7].

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

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

where λ_0 and Λ^T are called Farkas multipliers.

According to this Lemma, for each edge in the dependence graph, we can find a positive vector λ_0 and matrix Λ^T (the Farkas multipliers) such that:

$$T_R\mathbf{x}_R + \mathbf{t}_R - (T_S\mathbf{x}_S + \mathbf{t}_S) - \mathbf{1} = \lambda_0 + \Lambda^T \left(D \begin{pmatrix} \mathbf{x}_S \\ \mathbf{x}_R \end{pmatrix} + \mathbf{d} \right), \lambda_0 \geq 0, \Lambda^T \geq 0. \quad (4)$$

This formula can be split in as many equalities as there are independent variables (\mathbf{x}_S and \mathbf{x}_R components, parameters and scalar value) by equating their coefficients in both sides of the formula. The Farkas multipliers can be eliminated by using the Fourier-Motzkin projection algorithm [15] in order to find the constraints on the transformation functions. The constraint systems describe the legal transformation space, where each integral point corresponds to a legal solution.

3.3 Properties

Program transformations for locality aim at bringing the processing of some memory cells closer. The general framework using affine schedules is to find partial transformation functions (only the first few dimensions of the functions are defined) such that the partial execution dates of the operations referring to a given datum are the same. In this way, the operations have neighboring schedules and the datum may stay in the cache during the time intervals between the accesses. The framework ends by applying a completion procedure to achieve an invertible transformation function (see [18] for references).

For instance, let us consider self-temporal locality and a reference $B[f(\mathbf{x})]$ to an array B with the affine subscript function $f(\mathbf{x}) = F\mathbf{x} + \mathbf{a}$. Two instances of this reference, $B[f(\mathbf{x}_1)]$ and $B[f(\mathbf{x}_2)]$ refers the same memory location iff $f(\mathbf{x}_1) = f(\mathbf{x}_2)$, that is when $F\mathbf{x}_1 + \mathbf{a} = F\mathbf{x}_2 + \mathbf{a}$, then iff $F\mathbf{x}_r = \mathbf{0}$ with $\mathbf{x}_r = \mathbf{x}_1 - \mathbf{x}_2$. Thus there is self-temporal reuse when $\mathbf{x}_r \in \ker F$. The basis vectors of $\ker F$ give the reuse directions for the reference $B[f(\mathbf{x})]$, if $\ker F$ is empty, there is no self-temporal reuse for the corresponding reference. 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 in order for the transformation function to be instance-wise, by considering artificial dependences [13,9] or not [3]. For instance, let us consider the following pseudo-code:

```

do i = 1, n
  do j = 1, n
S1:    ... B[j] ...

```

the subscript function of the reference $B[j]$ is $f\left(\begin{smallmatrix} i \\ j \end{smallmatrix}\right) = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \left(\begin{smallmatrix} i \\ j \end{smallmatrix}\right)$, the kernel of the subscript matrix is then $\ker F = \text{span} \{(1, 0)\}$. Thus there is reuse generated by the reference $B[j]$, and we can exploit it thank to a transformation matrix built with an orthogonal vector to the reuse direction, e.g. $[0 \ 1]$ and its completion to a unimodular transformation matrix as described in [9]: $T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. The transformation function would be $\theta\left(\begin{smallmatrix} i \\ j \end{smallmatrix}\right) = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \left(\begin{smallmatrix} i \\ j \end{smallmatrix}\right)$, i.e. a loop interchange (the reader may care to verify that this solution do exploit the reuse of the reference $B[j]$). It is easy to generalize this method for several references by considering not only a reuse vector, but a reuse direction space (built with one basis vector per reference). It appears that there are a lot of liberty degrees when looking for a transformation improving self-temporal locality, since it is possible to choose the reuse direction space, the completion method and the constant vector of the transformation function.

Let us consider self-temporal locality and a transformation candidate before completion $\theta_{S_c}(\mathbf{x}_S) = T_{S_c}\mathbf{x}_S$. This function has the property that, modified in the following way:

$$\theta_S(\mathbf{x}_S) = C_S T_{S_c} \mathbf{x}_S + \mathbf{t}_S, \quad (5)$$

where C_S is an invertible matrix and \mathbf{t}_S is a constant vector, the locality properties are left unmodified for each time step. Intuitively, if θ_{S_c} gives the same execution date for \mathbf{x}_1 and \mathbf{x}_2 , then the transformed function θ_S does it as well. In the same way if the dates are different with θ_{S_c} , then the transformed function θ_S returns different dates. But while the values of C_S and \mathbf{t}_S do not change the self-temporal locality properties⁴, they can change the legality of the transformation.

Transformation expressions similar to (5) and having the same type of degrees of freedom can be used to achieve every type of locality (*self* or *group - temporal* or *spatial*) [16,5]. The challenge is, considering the candidate transformation matrices T_{S_c} , to find the *corrected matrices* $C_S T_{S_c}$ and the constant vectors \mathbf{t}_S in order for the transformation system to be legal for dependences.

⁴ A more formal discussion on this property, showing that locality transformations have to respect rank constraints not modified by C_S and \mathbf{t}_S can be found in [5].

4 Finding Legal Transformations

Optimizing compilers typically decouple the properties that the transformation functions have to satisfy to achieve optimization and legality. The basic framework is first to find the best transformations (e.g. for data locality improvement, which references carry the most reuse and necessitate new access patterns, which rank constraints should be respected by the corresponding transformation functions, etc.), then to *check* if a candidate transformation is legal or not⁵. If not, build and try another candidate, and so on. The major advantage of such a framework is to focus firstly on the most interesting properties, and the main drawback is to forsake these properties if a legal transformation is not directly found after a simple check of a candidate solution. We saw in section 3.3 that there exists an infinity of transformation functions having the same properties as a candidate transformation (see formula 5). Thus, it is not possible to check all these transformations to find a legal one. In this section we study another way: we show how to find, when possible, the unknown components $C_S T_{S_c}$ and t_S of formula 5 in order to correct the transformations for legality.

This problem can be solved in an iterative way, each dimension being considered as a stand-alone transformation. Each row of $C_S T_{S_c}$ is a linear combination of the rows of T_{S_c} . Thus, the unknown in the i^{th} algorithm iteration are, for each statement, the linear combination coefficients building the i^{th} row of $C_S T_{S_c}$ from T_{S_c} and the constant factor of the corresponding t_S entry. After each iteration, we have to update the dependence graph because there is no need to consider the dependences already satisfied. Thus, to find a solution is easier as the algorithm iterates. The algorithm is shown in figure 3.

Let us illustrate how the algorithm works using the example in figure 4. Suppose that an optimizing compiler would like to exploit the data reuse generated by the references to the array A of the program in figure 4(a) and that it suggests the transformation candidates in figure 4(b). As shown by the graph describing the resulting operation execution order, where each arrow represents a dependence relation and each backward arrow is a dependence violation, the transformation system is not legal. The correction algorithm modifies successively each transformation dimension. Each stand-alone transformation splits up the operations into sets such that there are no backward arrows between sets. The algorithm stops when there are no more backward arrows or when every dimension has been corrected. Then any polyhedral code generator, like CLooG⁶ [3], can generate the target code. Choosing transformation coefficients as small as possible (step 1(c)i) is a heuristic helping code generators to avoid control overhead.

The correctness of the algorithm comes from two properties: (1) the target transformations are legal, (2) the C_S matrices are invertible. The legality is achieved because each transformation part is chosen in the legal transformation

⁵ This can be done easily by instantiating the transformation functions in the legal transformation space as defined in section 3.2 then checking for the feasibility of the constraint system with any linear algebra tool.

⁶ CLooG is freely available at <http://www.prism.uvsq.fr/~cedb>

Correction Algorithm: Adjust a transformation system to respect dependences.

Input: a dependence graph DG, the transformation candidates $\theta_{S_c}(\mathbf{x}_S) = T_{S_c}\mathbf{x}_S$.

Output: the legal transformations $\theta_S(\mathbf{x}_S) = C_S T_{S_c} \mathbf{x}_S + \mathbf{t}_S$.

1. for dimension $i = 1$ to maximum dimension of T_{S_c}
 - (a) build the legal transformation space with:
 - for each edge in DG, the constraints of (4) for the i^{th} row of T_{R_c} and T_{S_c}
 - the constraints equating the i^{th} row entries of each $C_S T_{S_c}$ with a linear combination of T_{S_c} entries whose coefficients are unknown
 - (b) for each statement, remove from the solution space the trivial solution where $\forall j \geq i$ the linear combination coefficient of the j^{th} row of T_{S_c} is null
 - (c) if the solution space is empty, return \emptyset , else
 - i. pick the solution giving for each statement the minimum values for the entries of the i^{th} row of $C_S T_{S_c}$ and the i^{th} element of \mathbf{t}_S
 - ii. update DG: for each edge in DG, add to the dependence polyhedron the constraint equating the i^{th} dimension of $C_S T_{S_c} \mathbf{x}_S + \mathbf{t}_S$ of the statements labelling the source and destination vertices (this may empty the polyhedron for integral solutions)
 - iii. if every dependence polyhedra in DG are empty, goto 2
 - iv. for each statement, update the candidate transformation T_{S_c} :
 - replace a row such that the corresponding linear combination coefficient is not null with the i^{th} row
 - replace the i^{th} row with the i^{th} row of $C_S T_{S_c}$
 2. return the transformation functions $\theta_S(\mathbf{x}_S) = C_S T_{S_c} \mathbf{x}_S + \mathbf{t}_S$.
-

Fig. 3. Algorithm to correct the transformation functions

space (step 1a). The second property follows from the updating policy (step 1(c)iv): at start the C_S matrices are identities. During each iteration, we exchange their rows, multiply some rows by non null constants (as guaranteed by step 1b) and add to these rows a linear combination of the other rows. Each of these transformations does not modify the invertibility property.

5 Related Work

In compensation of the need for very simple dependences, first works on compiler techniques for improving data locality discuss *enabling transformations* to modify the program in such a way that the proposed method can apply. Wolf and Lam [16] proposed in their seminal *data locality optimizing algorithm* to use *skewing* and *reversal*⁷ to enable *tiling* as in previous works on automatic parallelization. McKinley et al. [14] proposed a technique based on a detailed cost model that drives the use of *fusion* and *distribution* mainly to enable loop *permutation*. Such methods are limited by the set of directives they use (like *fuse*

⁷ An exhaustive survey on loop transformations can be found in [18].

or *skew*) and because they have to apply them in a definite order. We claim that giving (and correcting) scheduling functions is more complete and has better compositionality properties.

A significant step on preprocessing techniques to produce fully permutable loop nests has been achieved by Ahmed et al. [1]. They use the Farkas Lemma to find a valid *code sinking*-like transformation if it exists. But this transformation is still independent from the optimization itself and it is limited to produce a fully permutable loop nest. The method proposed in this paper may find solutions while it is not possible to achieve such a loop nest. Recently, Griebl et al. [8] proposed to use well known space-time mapping [12] as a preprocessing technique for *tiling*. Our method can be included in their framework to find legal time mapping with locality properties.

Reasoning directly on scheduling functions, Li and Pingali proposed a completion algorithm to build a non-unimodular transformation function from a partial matrix, such that starting from a legal transformation, the completed transformation stay legal for dependences [13]. In the same spirit, Griebl et al. extended an arbitrary matrix describing a legal transformation to a square invertible matrix [9]. In contrast, we show in this paper how to find the valid functions before completion.

6 Conclusion and Future Work

In this paper is presented a general method correcting a program transformation for legality with no consequence on the data locality properties. It has been implemented in the Chunky prototype [5], advantageously replacing usual *enabling* preprocessing techniques and saving a significant amount of interesting transformations from being ignored. It could be used combined with a wide range of existing data locality improvement methods, for the single processor case as well as compiling techniques for parallel systems using space-time mappings [12].

Further implementation work is necessary to handle real-life benchmarks in our prototype and to provide full statistics on corrected transformations. Moreover, the question of scalability is left open since, for several tenth of deeply nested statements, the number of unknown in the constraint systems can become embarrassingly large. Splitting up the problem according to the dependence graph is a solution under investigation.

Acknowledgements

The authors would like to thank the CC'12 International Conference on Compiler Construction anonymous reviewers for having inspired this paper by pointing out their interest on this part of our work. We also wish to thank the Euro-Par anonymous reviewers for their help in improving the quality of the paper.

References

1. N. Ahmed, N. Mateev, and K. Pingali. Tiling imperfectly-nested loop nests. In *SC'2000 High Performance Networking and Computing*, Dallas, november 2000.
2. U. Banerjee. Unimodular transformations of double loops. In *Advances in Languages and Compilers for Parallel Processing*, pages 192–219, Irvine, august 1990.
3. C. Bastoul. Efficient code generation for automatic parallelization and optimization. In *ISPDC'03 IEEE International Symposium on Parallel and Distributed Computing*, pages 23–30, Ljubljana, october 2003.
4. C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral transformations to work. In *LCPC'16 International Workshop on Languages and Compilers for Parallel Computers, LNCS 2958*, pages 209–225, College Station, october 2003.
5. C. Bastoul and P. Feautrier. Improving data locality by chunking. In *CC'12 International Conference on Compiler Construction, LNCS 2622*, pages 320–335, Warsaw, april 2003.
6. P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.
7. P. Feautrier. Some efficient solutions to the affine scheduling problem: one dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, october 1992.
8. M. Griebl, P. Faber, and C. Lengauer. Space-time mapping and tiling – a helpful combination. *Concurrency and Computation: Practice and Experience*, 16(3):221–246, march 2004.
9. M. Griebl, C. Lengauer, and S. Wetzel. Code generation in the polytope model. In *PACT'98 International Conference on Parallel Architectures and Compilation Techniques*, pages 106–111, 1998.
10. I. Kodukula, N. Ahmed, and K. Pingali. Data-centric multi-level blocking. In *ACM SIGPLAN'97 Conference on Programming Language Design and Implementation*, pages 346–357, Las Vegas, june 1997.
11. D. Kuck. *The Structure of Computers and Computations*. John Wiley & Sons, Inc., 1978.
12. C. Lengauer. Loop parallelization in the polytope model. In *International Conference on Concurrency Theory, LNCS 715*, pages 398–416, Hildesheim, August 1993.
13. W. Li and K. Pingali. A singular loop transformation framework based on non-singular matrices. *International Journal of Parallel Programming*, 22(2):183–205, April 1994.
14. K. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, july 1996.
15. A. Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, 1986.
16. M. Wolf and M. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 30–44, New York, june 1991.
17. M. Wolfe. Iteration space tiling for memory hierarchies. In *3rd SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, december 1987.
18. M. Wolfe. *High performance compilers for parallel computing*. Addison-Wesley Publishing Company, 1995.
19. J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.

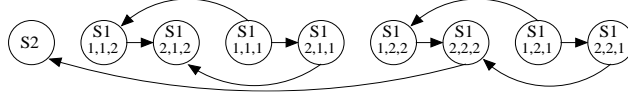
```

do i = 1, n
  do j = 1, n
    do k = 1, n
S1:      A(j,k) = A(j,k) + B(i,j,k) / A(j,k-1)
S2:      c = A(n,n) + 1

```

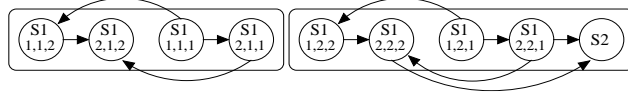
(a) Original program

$$\theta_{S1c} \begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}; \theta_{S2c} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$



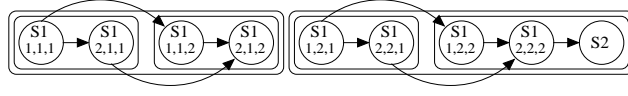
(b) Transformation function candidates

$$\theta_{S1c} \begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}; \theta_{S2c} = \begin{pmatrix} n \\ 0 \\ 0 \end{pmatrix}$$



(c) First correction iteration

$$\theta_{S1c} \begin{pmatrix} i \\ j \\ k \end{pmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} i \\ j \\ k \end{bmatrix} + \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}; \theta_{S2c} = \begin{pmatrix} n \\ 0 \\ 0 \end{pmatrix}$$



(d) Second and last correction iteration

```

do j = 1, n
  do k = 1, n
    do i = 1, n
S1:      A(j,k) = A(j,k) + B(i,j,k) / A(j,k-1)
S2:      c = A(n,n) + 1

```

(e) Target program

Fig. 4. Iterative transformation correction principle ($n = 2$ for graphs)