Parallel Processing Letters

ADJUSTING A PROGRAM TRANSFORMATION FOR LEGALITY

CÉDRIC BASTOUL
Laboratoire PRiSM, Université de Versailles Saint Quentin
45 avenue des États-Unis, 78035 Versailles Cedex, France
cedric.bastoul@prism.uvsq.fr

and

PAUL FEAUTRIER
LIP, École Normale Supérieure de Lyon
46 Allée d'Italie, 60364 Lyon, France
paul.feautrier@ens-lyon.fr

Received (received date)
Revised (revised date)
Communicated by (Name of Editor)

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

The task of optimizing compute-bound programs is crucial for present day supercomputers if one notices that most of these machines run at a few percent of their peak performance. The problem can be stated as a combinatorial optimization problem, but due to the compexity of real-life programs and computers, this approach is not practical. Most of the time, one starts from a first implementation, and tries to improve its performance by successive transformations. Beside improving the performances, a transformation must be legal, i.e. must not change the final results of the program. This is usually enforced by using only transformations that respect dependences [**?**]. While selecting an optimizing transformation is not too difficult for an experienced programmer, adjusting this transformation for legality is a tedious and error-prone process.

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) [26]. To enlarge their application domain some preprocessing, e.g. *loop skewing* or *code sinking*, may be applied [26,1,14]. More ambitious techniques do not lay down any requirement on dependences, but are limited to propose *solution candidates* then to *check* them for legality [17,8]. If the candidate is proved to violate dependences, then the proposed transformation is discarded and another candidate, perhaps having less interesting properties is studied. In this paper, we present a method that goes beyond checking by adjusting – if possible – a transformation for dependence satisfaction, without modifying its optimizing properties. This technique can be used to correct a transformation candidate as well as to replace preprocessing. The technique has been designed in the context of locality-improving transformations, but can be applied in many other cases. On the other hand, only transformations which can be represented as affine transformations in iteration space can be corrected in this way.

This paper is organized as follows. In section we outline the background of this work. Section deals with the transformations in the polyhedral model and focuses on both their dependences constraints and locality properties. Section shows how it is possible to correct a transformation for legality. Section compare our proposal to previous work in the field of locality enhancement. Lastly, section concludes and discusses future work.

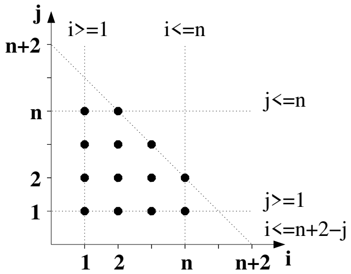
2. Background and Notations

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

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

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 be specified by a set of linear inequalities defining a polyhedron [18]. 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 [23]. A maximal set of consecutive statements in a program with such polyhedral iteration domains is called a *static control part* (SCoP) [7]. 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 $S(\vec{x})$ refers to the operation instance of the statement S with the iteration vector $\vec{x}$. 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 $(a_1 ... a_n)$ is executed before those corresponding to the

```
      do i = 1, n
        do j = 1, n
          if (i<=n+2-j)
S1:         B[i+j][2*i+1] = ...
```



$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -1 & -1 \end{bmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ n \\ -1 \\ n \\ n+2 \end{pmatrix} \geq \vec{0}$$

(a) surrounding control of S_1 (b) iteration domain of S_1

Figure 1: Static control and corresponding iteration domain

coordinates $(b_1 ... b_n)$ iff

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

We will use $\ll$ and $\underline{\ll}$ for the strict and non strict lexicographic order, respectively.

Each statement may include one or several *references* to arrays (or scalars, i.e. some particular cases of arrays). When the subscript function $f(\vec{x})$ of a reference is affine, we can write it $f(\vec{x}) = F\vec{x} + \vec{a}$ where F is called the *subscript matrix* and $\vec{a}$ is a constant vector. For instance, the reference to the array B in figure 1(a) is $B[f(\vec{x})]$ 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; the subscript may be omitted when it is clear from the context.

3. Affine Transformations

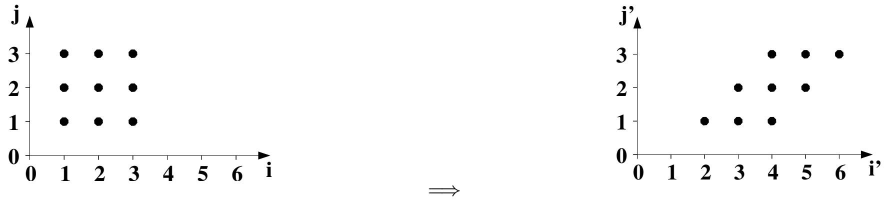
3.1. Formulation

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:

$$\theta_S(\vec{x}_S) = T_S\vec{x}_S + \vec{t}_S, \tag{1}$$

where $\vec{x}_S$ is the iteration vector, T_S is a constant transformation matrix and $\vec{t}_S$ is a constant vector (possibly including structure parameters).

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) [5,24]. Complex transformations such as tiling [25] can be achieved using linear transformations as well [27]. 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 $A\vec{x}+\vec{c} \geq \vec{0}$ and the transformation function θ leading to the target index $\vec{y} = T\vec{x}$, we deduce that the transformed polyhedron can be defined by $(AT^{-1})\vec{y} + \vec{c} \geq \vec{0}$ (there exits more convenient way to describe the target polyhedron as discussed in [6]). For instance, let us consider the polyhedron in figure 2(a) and the transformation function $\theta\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix}$. The corresponding transformation is a well known *iteration space skewing* and the resulting polyhedron is shown in figure 2(c).



$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} -1 \\ 3 \\ -1 \\ 3 \end{pmatrix} \geq \vec{0} \qquad \begin{pmatrix} i' \\ j' \end{pmatrix} = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix} \qquad \begin{bmatrix} 1 & -1 \\ -1 & 1 \\ 0 & 1 \\ 0 & -1 \end{bmatrix}\begin{pmatrix} i' \\ j' \end{pmatrix} + \begin{pmatrix} -1 \\ 3 \\ -1 \\ 3 \end{pmatrix} \geq \vec{0}$$

(a) original polyhedron $A\vec{x} + \vec{c} \geq \vec{0}$ (b) transformation function $\vec{y} = T\vec{x}$ (c) target polyhedron $(AT^{-1})\vec{y} + \vec{c} \geq \vec{0}$

Figure 2: A skewing transformation

3.2. Legality

In general, applying an arbitrary transformation to a program will change its semantics.

Two operations are said to be *in dependence* if they share a variable (memory cell) and at least one the operations modifies it. This definition was suggested by Bernstein [9] and is the most widely used one in works about program transformation. It is a sufficient condition for parallelism, but is by no means necessary, as the well known case of *reductions* shows.

Many tests have been designed for dependence checking. Most of these are based on sufficient conditions for independence. They give an approximate, conservative answer. The best known examples are the GCD-test [3], and the Banerjee test [4]. On the other hand, one can use classical algorithms from Linear Integer programming to get exact answers, as in the Omega-test [22] and the Simplex-Gomory testtest [12]. In the same way many dependence representations are possible, from the simplest ones as *dependence levels* [2] to the most precise as *dependence polyhedra* [16]. We chose for in this paper to use the most precise representation of dependences: the dependence polyhedra. However, many authors have noticed that approximate dependences can also be expressed as dependence polyhedra. Hence, our method applies whatever representation is choosen, provided the approximation is conservative.

In section , we recall how dependences in a SCoP can be expressed exactly using linear (in)equalities. Then we show in section how to build the legal transformation space where each program transformation has to be found.

3.2.1. Dependence Graph

A convenient way to represent the scheduling constraints 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.

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 exits an operation $S(\vec{x}_1)$, an operation $R(\vec{x}_2)$ and a memory location m such that:*

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

From this definition follows the description of the *dependence polyhedron* by affine (in)equalities. In these polyhedra, every integral point represents a dependence between two instances of the corresponding statements. The constraints 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: $F_S\vec{x}_S + \vec{a}_S = F_R\vec{x}_R + \vec{a}_R$.

2. *Iteration domains*: both S and R iteration domains can be described using affine inequalities, respectively $A_S\vec{x}_S + \vec{c}_S \geq \vec{0}$ and $A_R\vec{x}_R + \vec{c}_R \geq \vec{0}$.

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: $P_S\vec{x}_S - P_R\vec{x}_R + \vec{b} \geq \vec{0}$.

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:

$$\mathcal{D}_{S\delta R,l,p} : D \begin{pmatrix} \vec{x}_S \\ \vec{x}_R \end{pmatrix} + \vec{d} = \left[\begin{array}{cc} F_S & -F_R \\ \hline A_S & 0 \\ 0 & A_R \\ P_S & -P_R \end{array}\right] \begin{pmatrix} \vec{x}_S \\ \vec{x}_R \end{pmatrix} + \left(\begin{array}{c} \vec{a}_S - \vec{a}_R \\ \hline \vec{c}_S \\ \vec{c}_R \\ \vec{b} \end{array}\right) \begin{array}{c} = \\ \hline \geq \end{array} \vec{0} \qquad (2)$$

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 PipLib[a] [11]. 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, 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 $S\delta R$.

3.2.2. Legal Transformation Space

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

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

If there exists a dependence $S\delta R$, i.e. if $\mathcal{D}_{S\delta R}$ is not empty, then $\Delta_{R,S} \begin{pmatrix} \vec{x}_S \\ \vec{x}_R \end{pmatrix}$ must be lexicopositive in $\mathcal{D}_{S\delta R}$ (intuitively, the time interval between two operations $R(\vec{x}_R)$ and $S(\vec{x}_S)$ such that $R(\vec{x}_R)$ depends on $S(\vec{x}_S)$ must be at least $(0, \ldots, 0, 1)^T$, the smallest time interval: this guarantees that the operation $R(\vec{x}_R)$ is executed after $S(\vec{x}_S)$ in the target program). This condition represents as many constraints as there are points in $\Delta_{R,S}$. Fortunately, all these constraints can be compacted in a small set of affine constraints with the help of Farkas Lemma [13].

Lemma 1 *(Affine form of Farkas Lemma [23]) Let $\mathcal{D}$ be a nonempty polyhedron defined by the inequalities $A\vec{x}+\vec{b} \geq \vec{0}$. Then any affine function $f(\vec{x})$ is nonnegative everywhere in $\mathcal{D}$ iff it is a positive affine combination:*

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

[a] PipLib is freely available at `http://www.prism.uvsq.fr/~cedb`

λ_0 and $\vec{\lambda}^T$ are called Farkas multipliers.

$\Delta_{R,S}$ is a vector. For it to be lexicopositive, some of its components must be constrained to be either non negative or strictly positive. Let us apply Farkas Lemma to one of the constrained components. We can find a non-negative scalar λ_0 and a non-negative vector $\vec{\lambda}^T$ such that:

$$T_{R,\bullet}\vec{x}_R + t_R - (T_{S,\bullet}\vec{x}_S + t_S) - \delta = \lambda_0 + \vec{\lambda}^T \left(D \begin{pmatrix} \vec{x}_S \\ \vec{x}_R \end{pmatrix} + \vec{d} \right) \tag{4}$$

In this formula, $T_{R,\bullet}$ and $T_{S,\bullet}$ are corresponding rows in the T_R and T_S matrices, and δ is zero or one according to the position of the rows.

This formula can be split in as many equalities as there are independent variables ($\vec{x}_S$ and $\vec{x}_R$ components and parameters) by equating their coefficients in both sides of the formula. The Farkas multipliers can be eliminated by using the Fourier-Motzkin projection algorithm [23]. The result is a system of affine constraints on the coefficients of the transformation (the elements of $T_{R,\bullet}$ and $T_{S,\bullet}$. The important point is that this system is the same for all rows of the scheduling matrices and depends only on the dependence to be satisfied. Furthermore, it depends linearly on the value of δ. These systems completely characterize the legal transformations of a program, and can be computed once and for all as soon as the dependences are known.

4. Correcting Transformations

Both optimizing compilers and programmers have a tendency to think of performances first and to check legality afterward. The basic framework is first to find the best transformation (e.g. in the case of 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 not[c]. If the check fails, build and test 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. In this section we will show how it is possible to correct a candidate transformation for dependences, firstly when it can be described using explicit constraints as discussed in section . Then in section we study the special case of data locality improvement where the transformation properties are hidden.

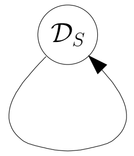
4.1. Transformations With Explicit Properties

Experts or optimizing compilers have a wide choice of optimizing transformations for a given program. Each transformation has a more or less precise cost

[c]This can be done easily by instantiating the transformation functions in the space of all affine transformation as defined in section , then checking whether it belong to the legal subset using any linear algebra tool.

model which helps in deciding whether to apply the transformation or not. In the polyhedral framework, many transformations are related to well chosen scheduling functions [5,13,10]. For instance, generalized loop interchange is associated to schedules whose matrix is a permuation matrix [5]. Trying to use these transformations as they are may result in a negative dependence test. Let us consider the code in Figure 3. An expert or an optimizing compiler may decide that moving the i-loop innermost would result in better locality. But because of complex dependences, using directly the loop interchange transformation $\theta_S\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix}$ is not legal. This will lead usually to rejecting the transformation.

```
   do i = 1, n
     do j = 1, n
S:     a(j+1) = 1/3 * (a(j)+a(j+1)+a(j+2))
```



$\mathcal{D}_{S\delta S,1,<a(j),a(j'+1)>}$
$\mathcal{D}_{S\delta S,1,<a(j+1),a(j')>}$
$\mathcal{D}_{S\delta S,1,<a(j+1),a(j'+1)>}$
$\mathcal{D}_{S\delta S,1,<a(j+1),a(j'+2)>}$
$\mathcal{D}_{S\delta S,1,<a(j+2),a(j'+1)>}$
$\mathcal{D}_{S\delta S,2,<a(j+1),a(j')>}$
$\mathcal{D}_{S\delta S,2,<a(j+2),a(j'+1)>}$

Figure 3: Original Hyperbolic-PDE program and dependence graph

The polyhedral model allow more flexibility when defining such transformations. Moreover, one can work with an incompletely specified transformations and use the legality constraints as a way of solving for the missing coefficients. The method consists in statingthe constraints the transformation has to satisfy, then solving these constraints and the legality constraints 4, using PipLib for instance. If the system does not have a solution, we conclude that there is no legal instance of the proposed transformation. For instance, "innermosting" the i-loop in the code in Figure 3 means that we are looking for a transformation $\theta_S\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} T_{1,1} & T_{1,2} \\ T_{2,1} & T_{2,2} \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} t_1 \\ t_2 \end{pmatrix}$ with as only constraints $T_{2,1} = 1$ and $T_{2,2} = 0$. By solving the system we find the solution $\theta_S\begin{pmatrix} i \\ j \end{pmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix}$. Expressed using classical transformation techniques, it is a combination of loop skewing and loop interchange. It leads to the target program in Figure 4 and as expected to a better cache behavior (on a i386 1GHz system with 128Ko L1 cache memory and n=33000 the number of cache misses of the original program is 68M but 43M for the target one).

4.2. Transformations With Implicit Properties: Data Locality

```
        do  i' = 2, 2*n
          do j' = max(i'-n,1), min(i'-1,n)
            j = i'-j' ;
            i = i' ;
S:          a(j+1) = 1/3 * (a(j)+a(j+1)+a(j+2))
```

Figure 4: Final Hyperbolic-PDE program

Cache are used in most computer systems to compensate for the mismatch between processor and memory performance (at the time of writing, factors of 10 to 100 are commonplace). Caches work by exploiting locality, i.e. the fact that accesses to each memory cell and its close neighbors have a tendency to cluster in the program code. While this is found to work very well for ordinary programs, it fails for compute-bound codes where large datasets are accessed according to very regular patterns. The basic framework for increasing cache hit rates is to move references to a given memory cell (or cache line) to neighboring iterations of some innermost loop. This reduces the elapsed time between two acesses and hence decreases the probability that the cell has been evicted from the cache. Such a transformation usually changes the execution order of the program, hence it must be checked for legality before being applied.

Another way of expressing the same intuition is to assign an execution date (a schedule) to each operation, and to take care that the date of accesses to the same memory cell are "almost" equal. This is usually obtained by requiring that the outer components of the schedule are equal. The methods continues by applying a completion procedure to achieve an invertible transformation function (see [26] for references).

For instance, let us consider self-temporal locality and a reference $B[f(\vec{x})]$ to an array B with the affine subscript function $f(\vec{x}) = F\vec{x} + \vec{a}$. Two instances of this reference, $B[f(\vec{x}_1)]$ and $B[f(\vec{x}_2)]$ refers the same memory location iff $f(\vec{x}_1) = f(\vec{x}_2)$, that is when $F\vec{x}_1 + \vec{a} = F\vec{x}_2 + \vec{a}$, then iff $F\vec{x}_r = \vec{0}$ with $\vec{x}_r = \vec{x}_1 - \vec{x}_2$. Thus there is self-temporal reuse when $\vec{x}_r \in \ker F$. The basis vectors of $\ker F$ give the reuse directions for the reference $B[f(\vec{x})]$; if $\ker F$ is trivial, there is no self-temporal reuse for the corresponding reference. Reuse can be exploited if the transformed iteration order follows one of the reuse directions. Then we have to find a vector orthogonal 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 one-to-one either by considering artificial dependences [20,15] or not [6]. As an example, 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 $f\begin{pmatrix} i \\ j \end{pmatrix}=\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix}$, 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 [15]: $T = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. The transformation function would be $\theta\begin{pmatrix} i \\ j \end{pmatrix}=\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}\begin{pmatrix} i \\ j \end{pmatrix}$, 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 direction vector, but a reuse direction space (built with one basis vector per reference). It appears that there are a lot of degrees of freedom 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 $\theta_{Sc}(\vec{x}_S) = T_{Sc}\vec{x}_S$. This function has the property that, modified in the following way:

$$\theta_S(\vec{x}_S) = C_S T_{Sc}\vec{x}_S + \vec{t}_S, \tag{5}$$

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

It is clearly not possible to check all these transformations for legality. In the following we study another way: we show how to find, when possible, the unknown components $C_S T_{Sc}$ and $\vec{t}_S$ of formula 5 in order to construct a legal transformation.

Correcting formulaes 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*) [24,8]. The challenge is, considering the candidate transformation matrices T_{Sc}, to find the *corrected matrices* $C_S T_{Sc}$ and the constant vectors $\vec{t}_S$ in order for the transformation system to be legal for dependences.

This problem can be solved in an iterative way, each dimension being considered as a stand-alone transformation. Each row of $C_S T_{Sc}$ is a linear combination of the rows of T_{Sc}. Thus, the unknown in the i^{th} algorithm iteration are, for each statement, the linear combination coefficients building the i^{th} row of $C_S T_{Sc}$ from

[b] This amount to noticing that the amount of locality in the transformed program is linked to the *rank* of T_{Sc}. For a more formal discussion, see [8].

T_{Sc} and the constant factor of the corresponding $\vec{t}_S$ entry. After each iteration, we have to update the dependence graph since, by a property of lexicographic order, there is no need to consider the already satisfied dependences. Thus, to find a solution is easier as the algorithm iterates. The algorithm is shown in figure 5.

Let us illustrate how the algorithm works using the example in figure 6. 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 6(a) and that it suggests the transformation candidates in figure 6(b). As shown by the graph describing the resulting

---

Correction Algorithm: adjust a transformation system to respect dependences

---

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

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

1. for dimension i = 1 to maximum dimension of T_{Sc}
   (a) build the legal transformation space with:
      - for each edge in DG, the constraints of (4) for the i^{th} row of T_{Rc} and T_{Sc}
      - the constraints equating the i^{th} row entries of each $C_S T_{Sc}$ with a linear combination of T_{Sc} 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_{Sc} 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 $C_S T_{Sc}$ and the i^{th} element of $\vec{t}_S$
      ii. update DG: for each edge in DG, add to the dependence polyhedron the constraint equating the i^{th} dimension of $C_S T_{Sc}\vec{x}_S + \vec{t}_S$ 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_{Sc}:
         - 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 $C_S T_{Sc}$
2. return the transformation functions $\theta_S(\vec{x}_S) = C_S T_{Sc}\vec{x}_S + \vec{t}_S$.

Figure 5: Algorithm to correct the transformation functions

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 CLooG[d] [6], 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 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

Since they cannot deal with (complex) dependences, the earliest works on locality improvement discuss *enabling transformations* to modify the program in such a way that the proposed method can apply. Wolf and Lam [24] proposed in their seminal *data locality optimizing algorithm* to use *skewing* and *reversal* to enable *tiling* as in previous works on automatic parallelization. McKinley et al. [21] 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* or *skew*) and because they have to apply them in a definite order. We claim that proposing (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 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 even when it is not possible to extract such a loop nest.

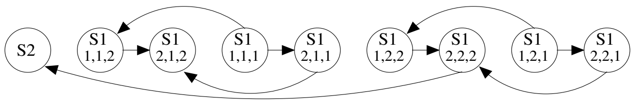
The method of Griebl et al. [14] is quite different. Their aim is to minimize the amount of communication in a distributed program, which is indeed a kind of locality optimization. They first take care of dependences by finding a legal space-time transformation (i.e. a schedule and a placement) and then tile in space-time to achieve the optimal granularity. Adapting these ideas to cache optimization seems by no mean obvious, althought it is an interesting subject for further reasearch.

[d] CLooG is freely available at `http://www.prism.uvsq.fr/~cedb`

```
        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
```

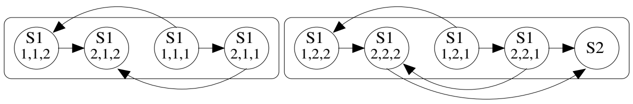
(a) Original program

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



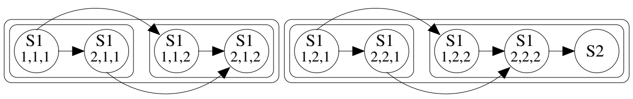
(b) Transformation function candidates

$$\theta_{S1c}\begin{pmatrix} i \\ j \\ k \end{pmatrix} = \left[\begin{array}{ccc} 1 & 0 & 0 \\ \hline 0 & 1 & 0 \\ 0 & 0 & 1 \end{array}\right]\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} i \\ j \\ k \end{bmatrix} + \left(\begin{array}{c} 0 \\ \hline 0 \\ 0 \end{array}\right);\ \theta_{S2c} = \left(\begin{array}{c} n \\ \hline 0 \\ 0 \end{array}\right)$$



(c) First correction iteration

$$\theta_{S1c}\begin{pmatrix} i \\ j \\ k \end{pmatrix} = \left[\begin{array}{ccc} 1 & 0 & 0 \\ 0 & -1 & 0 \\ \hline 0 & 0 & 1 \end{array}\right]\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & -1 \\ 0 & 0 & 0 \end{bmatrix}\begin{bmatrix} i \\ j \\ k \end{bmatrix} + \left(\begin{array}{c} 0 \\ 0 \\ \hline 0 \end{array}\right);\ \theta_{S2c} = \left(\begin{array}{c} n \\ n \\ \hline 0 \end{array}\right)$$



(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

Figure 6: Iterative transformation correction principle ($n = 2$ for graphs)

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 [20]. In the same spirit, Griebl et al. [15] extended an arbitrary matrix describing a legal transformation to a square invertible matrix. In contrast, we show in this paper how to find the valid functions before completion.

6. Conclusion and Future Work

In this paper we presented a general method correcting a program transformation for legality with no consequence on data locality properties. It has been implemented in the Chunky prototype [8], 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 for parallel systems using space-time mappings [19].

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 manifesting 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] J. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, october 1987.

[3] U. Banerjee. Data dependence in ordinary programs. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, November 1976.

[4] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic, 1988.

[5] U. Banerjee. Unimodular transformations of double loops. In *Advances in Languages and Compilers for Parallel Processing*, pages 192–219, Irvine, august 1990.

[6] 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.

[7] 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.

[8] 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.

[9] A. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, october 1966.

[10] A. Cohen, S. Girbal, and O. Temam. A polyhedral approach to ease the composition of program transformations. In *Euro-Par'10 International Euro-Par Conference*, Pisa, august 2004.

[11] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988.

[12] P. Feautrier. Dataflow analysis of scalar and array references. *International Journal of Parallel Programming*, 20(1):23–53, february 1991.

[13] P. Feautrier. Some efficient solutions to the affine scheduling problem: one dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, october 1992.

[14] 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.

[15] 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.

[16] F. Irigoin and R. Triolet. Computing dependence direction vectors and dependence cones with linear systems. Technical Report ENSMP-CAI-87-E94, Ecole des Mines de Paris, Fontainebleau (France), 1987.

[17] 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.

[18] D. Kuck. *The Structure of Computers and Computations.* John Wiley & Sons, Inc., 1978.

[19] C. Lengauer. Loop parallelization in the polytope model. In *International Conference on Concurrency Theory, LNCS 715*, pages 398–416, Hildesheim, August 1993.

[20] 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.

[21] 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.

[22] W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the third ACM/IEEE conference on Supercomputing*, pages 4–13, Albuquerque, august 1991.

[23] A. Schrijver. *Theory of linear and integer programming.* John Wiley & Sons, 1986.

[24] 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.

[25] M. Wolfe. Iteration space tiling for memory hierarchies. In *3rd SIAM Conference on Parallel Processing for Scientific Computing*, pages 357–361, december 1987.

[26] M. Wolfe. *High performance compilers for parallel computing.* Addison-Wesley

Publishing Company, 1995.
[27] J. Xue. On tiling as a loop transformation. *Parallel Processing Letters*, 7(4):409–424, 1997.