Mapping Deviation: A Technique to Adapt or to Guard Loop Transformation Intuitions for Legality

Cédric Bastoul
University of Strasbourg and Inria, Strasbourg, France
cedric.bastoul@unistra.fr

Abstract

Parallel architectures are now omnipresent in mainstream electronic devices and exploiting them efficiently is a challenge for all developers. Hence, they need the support of languages, libraries and tools to assist them in the optimization or parallelization task. Compilers can provide a major help by automating this work. However they are very fragile black-boxes. A compiler may take a bad optimization decision because of imprecise heuristics or may turn off an optimization because of imprecise analyses, without providing much control or feedback to the end user. To address this issue, we introduce mapping deviation, a new compiler technique that aims at providing a useful feedback on the semantics of a given program restructuring. Starting from a transformation intuition a user or a compiler wants to apply, our algorithm studies its correctness and can suggest changes or conditions to make it possible rather than being limited to the classical go/no-go answer. This algorithm builds on state-of-the-art polyhedral representation of programs and provides a high flexibility. We present two example applications of this technique: improving semi-automatic optimization tools for programmers and automatically designing runtime tests to check the correctness of a transformation for compilers.

Categories and Subject Descriptors D 3.4 [*Programming languages*]: Processor — Compilers; Optimization

Keywords Compilation; Dependence Analysis; Loop Transformations; Polyhedral Model

1. Introduction

High-level loop transformations are a key tool to map efficiently compute-intensive applications to multi-core systems which are now powering most computing platforms. Finding the best transformation for a given input code and a given target architecture has been a business for experts and a major research field in compilation for several decades [1, 7, 14, 25, 30, 33, 42]. The most important constraint that experts and compilers are facing while trying to find the best optimization is to preserve the original program semantics. If it is proven to be too challenging for the expert or if the analyses of the compiler are not powerful enough, the best transformations may be discarded. Worse, the compiler most probably applied some irreversible preprocessing to ease its analyses for nothing, while it may severely hamper performance.

Early compiler techniques were dealing with semantics preservation by relying on so-called *enabling transformations*: a specific pre-processing driven by a data dependence analysis to make the desired transformation possible, e.g., loop skewing for Wolf and Lam's data locality optimization technique [41] or loop splitting for Allen and Kennedy's parallelization algorithm [1]. Modern compiler optimization techniques guarantee legality by construction: data dependencies and causality conditions are encoded within the algorithms that can issue only a legal sequence of transformations [7, 14, 25, 31]. Hence, to maximize their efficiency, some preprocessing is applied to the input program to remove as many data dependencies as possible before the optimization step, e.g., privatization, total memory expansion, static single assignment form [23], index-set splitting [18] etc. Often, there is no easy way back from those pre-processing techniques while they may have a serious impact on memory use and/or control overhead.

In this paper we use a radically different (yet possibly complementary) transformation-centric approach. Starting from a transformation an expert or a compiler would like to apply, we study its impact with respect to data dependencies and we extend the limited go/no-go answer of usual legality checking by trying, if necessary, to suggest limited changes to the transformation to make it legal, or conditions to ensure its legality at runtime. Our work builds on the state-of-the-art, relation-based polyhedral representation of programs to target complex sequences of loop transformations and to achieve precise analyses. Unlike existing transformation correction schemes [5, 27, 36], we reason about any component of the program representation, in their most general form.

Specifically, this paper brings three contributions:

- We revisit the formulation of violated dependence analysis introduced by Vasilache et al. [35] with the relation-based polyhedral abstraction, which allows to consider more input codes, more transformations and facilitates our strategy.
- We introduce *mapping deviation*, an original solution that brings the ability to reason about the legality of transformations when selected dimensions of selected relations of the program's polyhedral representation have been modified by adding specific deviation parameters. Compared to related work on loop transformation semantic feedback, mapping deviation uses a radically different approach based on safe-space manipulation and offers a significantly wider application domain.
- We contribute two example applications of mapping deviation: correcting a general polyhedral transformation for legality and generating an array overlapping test to check at runtime whether a given transformation is legal or not, when the pointer alias analysis is deficient.

CC'16, March 17–18, 2016, Barcelona, Spain
ACM. 978-1-4503-4241-4/16/03...
http://dx.doi.org/10.1145/2892208.2892216

This paper is organized as follows. We present our motivation through an example that illustrates the benefits of our approach in Section 2. In Section 3, we introduce the useful background, in particular the various elements of a relation-based polyhedral representation of programs. We build on this background to introduce our mapping deviation technique in Section 4 and we present two example applications in Section 5. Finally, we discuss related work in Section 6 and we conclude in Section 7.

2. Motivation

While compiler techniques typically reason about what they *can* do with respect to data dependencies to optimize a given code, we propose a very different approach by reasoning about what we *want* to do to optimize that code. To illustrate our transformation-centric approach and how it can be used, let us consider the code in Figure 1. This program applies two classical image filters to an image stored in the array `Img` and uses the array `Tmp` to store the intermediate result. We will consider two scenarios.

```
/* Mean Blur Filter */
for (i = 1; i < L - 1; i++)
  for (j = 1; j < W - 1; j++)
S1: Tmp[i][j] =
      (Img[i-1][j-1] + Img[i-1][j] + Img[i-1][j+1] +
       Img[i  ][j-1] + Img[i  ][j] + Img[i  ][j+1] +
       Img[i+1][j-1] + Img[i+1][j] + Img[i+1][j+1])/9;

/* Roberts Edge Detection Filter */
for (i = 1; i < L - 2; i++)
  for (j = 2; j < W - 1; j++)
S2: Img[i][j] = abs(Tmp[i  ][j] - Tmp[i+1][j-1]) +
                abs(Tmp[i+1][j] - Tmp[i  ][j-1]);
```

Figure 1. Running Example: `Mean-Roberts` Edge-Detect Filter

1. **First scenario, an optimization expert programmer** notices the high reuse of the array `Tmp` between the loops and wants to achieve data locality by fusing them. To avoid writing the code including a prologue and an epilogue due to non-matching loop bounds, he/she uses a semi-automatic, script-based transformation tool such as UTF [22], URUK [16], CHiLL [8] or Clay [2] to achieve that transformation and to generate the code. Unfortunately he/she gets a no-go from the data dependence analysis of the tool (or worse, a wrong code if no such analysis is possible, like when using Goofi [28] or XLanguage [11]) because that transformation would violate data dependencies (e.g., `Tmp[i+1][j]` would be used by the second statement while it would not have been produced yet by the first statement – which is only producing `Tmp[i][j]` during the same iteration). Hence, the programmer has to redesign his optimization.

   The technique described in this paper goes beyond checking the legality of the transformation by computing a limited deviation to make it legal. In this case it would suggest to complement the loop fusion with a minimal shifting to delay the execution of the iterations of the second statement until each consumed value has been produced. Most probably, the programmer would have corrected his transformation in the same way, but our technique computes and applies this deviation automatically.

   Few previous works addressed this issue [5, 36], however, we bring a different, general solution not tied to a specific optimization family (as data locality optimization for [5]) or to a representation structure (as the URUK representation for [36]).

2. **Second scenario, an optimizing compiler** with data locality optimization strategy could compute the same loop fusion + shifting transformation to exploit the data reuse that the optimization expert assisted by our technique could find, if its pointer alias analysis is powerful enough to ensure that `Img` and `Tmp` are pointing disjoint storage spaces. If it is not the case, the transformation is not considered at all because the compiler over-approximates data dependencies to guarantee the correctness of the target program, and this discards the proposed transformation. A recent work points out how relevant is such a scenario for LLVM [10].

   Our technique goes beyond naive conservative policies by computing a test on `Img` and `Tmp` relatively to the desired transformation to ensure it is possible. In this case, the test would check the array addresses are organized in the address space in a way that guarantees the transformation does preserve the original program semantics. The compiler can use this test to decide whether to use the optimized version or the original version at runtime.

   To the best of our knowledge, no other work can address this problem relatively to a given transformation. We solved it by exploiting the polyhedral representation of input programs and desired transformations.

3. Polyhedral Abstraction of Programs

The polyhedral model is a mathematical representation that encodes individual dynamic executions of statements (called statement *instances*) of a subset of imperative programs [15]. It is an alternative (complementary) abstraction to syntactic representations, such as abstract syntax trees, which are closer to the structure of the program rather than to its dynamic behavior. Its application domain is typically loop-based program parts where conditions, loop bounds and array subscripts are affine forms of outer loop counters and fixed parameters, but extensions exist to support up to full functions [6, 38]. Several compilers have the ability to raise such static control program parts (SCoPs) to a polyhedral form such as GCC [34], LLVM [19] or IBM XL [7]. Our work applies to any SCoP, in their most general form.

Roughly, program restructuring in the polyhedral model is a three step process. First, a program which can fit the model is raised to a polyhedral representation using three main abstractions: *iteration domains* which define the sets of statement instances, *scheduling* which encodes their relative ordering and *accesses* which express their data accesses. Next, an optimizing algorithm [7, 14, 25] or an expert [8, 16, 22] designs a new scheduling to reorder statement instances while respecting data dependencies. Finally the code implementing the new scheduling is generated by polyhedra scanning [3, 32].

This model has two key properties for our transformation-centric strategy. First, because it operates at the instance level, it is expressive enough to encode and to apply arbitrarily complex sequences of loop transformations [16]. Second, it makes possible an exact, instance-wise, data dependence analysis [14]. Hence, building on the polyhedral model, our work can apply to a large range of optimizations while trying to fix the exact, minimum set of offending elements in the transformation sequence. To achieve this goal, we build on its state-of-the art mathematical representation, based on polyhedral relations rather than on functions. It allows to consider and/or to represent consistently more general programs (supporting non-unit strides, OR conditions, general data accesses through approximations etc.) and transformations (supporting index-set splitting, stride transformation etc.) [4]. Such relations are defined in Section 3.1, and we review the required abstractions in the following sections.

3.1 Polyhedral Relations

A *relation* is a mapping from a set of input coordinates in an input space to a set of output coordinates in an output space. A *polyhedral relation* $\mathcal{R}(\vec{p})$ is a finite union of convex relations $\mathcal{R}(\vec{p}) = \bigcup_i \mathcal{R}_i(\vec{p})$, each convex relation being a function associating to $\vec{p}$ a relation that can be represented using m affine constraints in the following way:

$$\mathcal{R}_i(\vec{p}) = \left\{ \vec{x}_{in} \to \vec{x}_{out} \;\middle|\; \exists \vec{l_i} : R_i \begin{pmatrix} \vec{x}_{out} \\ \vec{x}_{in} \\ \vec{l_i} \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \right\},$$

where:

- $\vec{x}_{in} \in \mathbb{Z}^{dim(\vec{x}_{in})}$ is an input coordinate,
- $\vec{x}_{out} \in \mathbb{Z}^{dim(\vec{x}_{out})}$ is an output coordinate,
- $\vec{l_i} \in \mathbb{Z}^{dim(\vec{l_i})}$ is the vector of *local variables* (also referred as *set variables* or *existentially quantified variables* in the literature),
- $\vec{p} \in \mathbb{Z}^{dim(\vec{p})}$ is the vector of *parameters* (unknown but fixed integer values, a.k.a. *free variables*),
- $R_i \in \mathbb{Z}^{m \times (dim(\vec{x}_{out}) + dim(\vec{x}_{in}) + dim(\vec{l_i}) + dim(\vec{p}) + 1)}$ is an integer matrix.

Polyhedral relations were originally suggested by Kelly and Pugh as a convenient object to represent programs and to compute data dependencies [22]. However, they had to respect various constraints and could not be used directly to represent all aspects of a SCoP (e.g., scheduling had to be invertible). Modern implementations of polyhedral relations such as `isl` [39] allow the use of the full expressiveness of relations. In this work, we make an unrestricted use of this representation[1]. Without loss of generality, we will only consider convex relations without local variables to simplify notations.

3.2 Abstracting Instances: Iteration Domains

The key aspect of the representation of programs in the polyhedral model is to consider *statement instances*. A statement instance is *one* particular execution of a statement.

Each instance of a statement enclosed inside a loop can be associated with the value of its outer loop counters: its *iteration vector*. Hence, a compact way to represent all the instances of a given statement is to consider the set of all possible iteration vectors. This set is called the statement's *iteration domain*. It can be conveniently described by all the constraints on the enclosing loop iterators. When those constraints are affine and depend only on the outer loop counters and some parameters, the set of constraints defines a $\mathbb{Z}$-polyhedron. The general form of the iteration domain of a statement S is a degenerated relation without input dimensions (or exactly, a constant input) and where output dimensions correspond to the statement's iteration space:

$$\mathcal{D}_S(\vec{p}) = \left\{ () \to \vec{\imath}_S \;\middle|\; D_S \begin{pmatrix} \vec{\imath}_S \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \right\},$$

where $\vec{\imath}_S \in \mathbb{Z}^{dim(\vec{\imath}_S)}$ is the statement's iteration vector and $D_S \in \mathbb{Z}^{m_{\mathcal{D}_S} \times (dim(\vec{\imath}_S) + dim(\vec{p}) + 1)}$ is an integer matrix where $m_{\mathcal{D}_S}$ is the number of constraints. For instance, the iteration domain of the first statement in the code in Figure 1 is:

[1] Note that while the general representation of convex relations shows only inequalities, equalities (which may be obviously captured via inequalities) are allowed as well.

$$\mathcal{D}_{S1}\begin{pmatrix} L \\ W \end{pmatrix} = \left\{ () \to \begin{pmatrix} i \\ j \end{pmatrix} \;\middle|\; \left[\begin{array}{cc|cc|c} 1 & 0 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & -2 \\ 0 & 1 & 0 & 0 & -1 \\ 0 & -1 & 0 & 1 & -2 \end{array}\right] \begin{pmatrix} i \\ \hline j \\ L \\ \hline W \\ 1 \end{pmatrix} \geq \vec{0} \right\},$$

where each row of the D_{S1} matrix corresponds to a constraint on the loop counters (here, simply the loop bounds).

3.3 Abstracting Ordering: Scheduling Relations

Iteration domains do not provide any information about the relative execution order between statement instances. This information is provided by *scheduling relations*. They describe a relation between statement instances and logical dates in a logical time. It is denoted θ_S for a given statement S. Logical dates may be multidimensional, like clocks: the first dimension could correspond to days (most significant), the next one to hours (less significant), the third one to minutes and so on, a *lexicographical* order. A scheduling relation for a statement S is a mapping where input dimensions correspond to its iteration space and output dimensions correspond to the target logical time:

$$\theta_S(\vec{p}) = \left\{ \vec{\imath}_S \to \vec{t}_S \;\middle|\; T_S \begin{pmatrix} \vec{\imath}_S \\ \vec{\imath}_S \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \right\},$$

where $\vec{\imath}_S \in \mathbb{Z}^{dim(\vec{\imath}_S)}$ is the statement's iteration vector in the original code, $\vec{t}_S \in \mathbb{Z}^{dim(\vec{t}_S)}$ is the statement's target logical scheduling time, and $T_S \in \mathbb{Z}^{m_{\theta_S} \times (dim(\vec{\imath}_S) + dim(\vec{t}_S) + dim(\vec{p}) + 1)}$ is an integer matrix where m_{θ_S} is the number of constraints. For instance, the following scheduling relations for the two statements of the code in Figure 1 model the same ordering as the original code:

$$\theta_{S1}\begin{pmatrix} L \\ W \end{pmatrix} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} \;\middle|\; \left[\begin{array}{ccc|cc|cc|c} -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 \end{array}\right] \begin{pmatrix} t_1 \\ t_2 \\ \hline t_3 \\ i \\ \hline j \\ L \\ \hline W \\ 1 \end{pmatrix} = \vec{0} \right\},$$

$$\theta_{S2}\begin{pmatrix} L \\ W \end{pmatrix} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} t_1 \\ t_2 \\ t_3 \end{pmatrix} \;\middle|\; \left[\begin{array}{ccc|cc|cc|c} -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 \end{array}\right] \begin{pmatrix} t_1 \\ t_2 \\ \hline t_3 \\ i \\ \hline j \\ L \\ \hline W \\ 1 \end{pmatrix} = \vec{0} \right\}.$$

Here the first scheduling dimension is 0 for the first statement and 1 for the second one, ensuring all instances of the first statement are executed before any instance of the second one. The other dimensions (set to i then j for both statements) express that the target ordering follows the original lexicographic order for both statements. Hence the order of the instances in the target time space is the same as in the original code.

Scheduling relations can encode arbitrarily complex sequences of loop transformations such as skewing, interchange, shifting, tiling etc. Several automatic [7, 30, 40] or semi-automatic [8, 16, 22] program transformation frameworks have been proposed on top of particular cases of such relations like scheduling *functions*.

3.4 Abstracting Memory Accesses: Access Relations

The access relation models the memory reference behavior for a given access. It maps statement instances to memory cells for a given statement and array reference. Hence, it is a relation where input dimensions correspond to the iteration space of the statement and the output dimensions are the accessed array dimensions:

$$\mathcal{A}_{S,r}(\vec{p}) = \left\{ \vec{\imath}_S \to \vec{a}_{S,r} \;\middle|\; A_{S,r} \begin{pmatrix} \vec{a}_{S,r} \\ \vec{\imath}_S \\ \vec{p} \\ 1 \end{pmatrix} \geq \vec{0} \right\},$$

where r is the memory reference number in the statement, $\vec{\imath}_S \in \mathbb{Z}^{dim(\vec{\imath}_S)}$ is the statement's iteration vector, $\vec{a}_{S,r} \in \mathbb{Z}^{dim(\vec{a}_{S,r})}$ is the access vector where the i^{th} element corresponds to the index of the i^{th} array dimension and $A_{S,r} \in \mathbb{Z}^{m_{\mathcal{A}_{S,r}} \times (dim(\vec{a}_{S,r}) + dim(\vec{\imath}_S) + dim(\vec{p}) + 1)}$ is an integer matrix where $m_{\mathcal{A}_{S,r}}$ is the number of constraints. For instance, the access relation for the first access of the first statement (`Tmp[i][j]`) in Figure 1 is:

$$\mathcal{A}_{S1,1}\begin{pmatrix} L \\ W \end{pmatrix} = \left\{ \begin{pmatrix} i \\ j \end{pmatrix} \to \begin{pmatrix} a^1_{S1,1} \\ a^2_{S1,1} \end{pmatrix} \;\middle|\; \left[\begin{array}{cc|cc|cc|cc} -1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 1 & 0 & 0 & 0 \end{array}\right] \begin{pmatrix} a^1_{S1,1} \\ a^2_{S1,1} \\ \hline i \\ j \\ \hline L \\ W \\ \hline 1 \end{pmatrix} = \vec{0} \right\},$$

where each row of the $A_{S1,1}$ matrix corresponds to the subscript of a dimension of the array `Tmp`.

3.5 Abstracting Dependencies: Dependence Relations

Not all schedules do preserve the original program semantics. Hence, it is critical to formalize the necessary information to preserve it. This is done through the *data dependency* abstraction. This abstraction characterizes the fact that some ordered statement instances access the same memory location [43].

Several dependency abstractions have been proposed, with various accuracy to support a given kind of transformation and/or some constraints on the compilation time [20]. The most precise abstraction is the *dependences between iterations*, defined by Irigoin and Triolet [21] since it encodes the exact set of dependent instances, even when they are from different loop nests. For some subsets of input codes and desired transformations, simple abstractions may encode the exact information [43]. However, in our context of general domain, access and scheduling relations, where the desired transformation is unknown a priori, it is necessary to use the most precise abstraction to avoid missing a deviation opportunity.

Hence, in this work we rely on dependences between iterations, however we will use the term *dependence relations* for consistency and to highlight the fact that unlike most existing work, we use access relations instead of access functions to describe them. A dependence relation $\delta_{S,r_S \xrightarrow{d} T,r_T}(\vec{p})$ is a mapping from instances and accessed memory locations of a *source* statement S to instances and accessed memory locations of a *target* statement T, at a given *dependence depth* d (defined below), for a given pair of memory references r_S and r_T. The dependence relation is not empty for two instances iff they access the same memory locations and the source instance accesses the memory locations first. The dependence relation is built using three sub-relations:

1. **Existence condition** of the instances: the instances must belong to the iteration domains of their respective statements. The constraints involved are those of the iteration domains, see Section 3.2.
2. **Conflict condition** of the memory locations: the memory locations must belong to the access relation of their respective accesses and they must refer to the same locations. The constraints involved are those of the access relations, see Section 3.4, plus the equality of the access dimensions.
3. **Causality condition** of the instances: the instances of the source statement in the dependence relation are executed before the corresponding instances of the target statement. The constraints involved in this condition depend on the situation. They can be separated into a disjunction with as many parts as common loops to both S and T. Each part corresponds to a common loop depth called a *dependence depth*. For a given dependence depth $d > 0$, the causality condition has two parts:
   - the equality of the iteration vector elements at depth less than d: $i^x_S = i^x_T$ for $x < d$,
   - $i^d_T \geq i^d_S$ if S is textually before T, $i^d_T > i^d_S$ otherwise.

   If no loop is shared by S and T, there is no causality constraint and the dependency may only exist if S is textually before T.

Figure 2 shows the general form of a dependence relation with causality constraints. We use the notation $M^{\vec{v}}$ for the submatrix of M made of the columns of M to be multiplied with the vector elements of $\vec{v}$, and $I^{\text{rows},\bullet}$ for the matrix made of selected rows of the identity matrix.

The complete information about data dependencies of an input program is stored in the *data 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 are labelled with the dependence relation between the source and destination statements.

4. Removing Dependence Violations

The heart of our approach is to study the impact of a program transformation with respect to data dependencies. This is done through violated dependence analysis detailed in Section 4.1. We build on this analysis to propose our mapping deviation technique to find corrections or conditions to guarantee the legality of a transformation. We present this approach in Section 4.2 and useful variations of the base algorithm in Section 4.3.

4.1 Abstracting Violations: Violation Relations

Data dependence relations provide the exact sets of instance couples such that their relative order must be preserved by a program transformation. In the relation formalism, the legality of a transformation is guaranteed if, for any couple of iteration vectors $\vec{\imath}_S$ and $\vec{\imath}_T$ involved in a dependence relation $\delta_{S,r_S \xrightarrow{d} T,r_T}(\vec{p})$,

$$\theta_S(\vec{\imath}_S) \prec \theta_T(\vec{\imath}_T),$$

where $\prec$ denotes the lexicographical order. Intuitively, this means that the source instance has to be executed before the target instance after the scheduling has been applied.

Violation relations aim at encoding the exact sets of instance couples in dependence relation such that the relative order has been reversed by a given transformation. We revisit here the formulation of Vasilache et al. to benefit from the more general relational representation which allows to support more codes and transformations than the function-based original characterization [35].

A violation relation $\upsilon_{S,r_S \xrightarrow{d,v} T,r_T}(\vec{p})$ is a mapping from scheduled instances and accessed memory locations of a source statement S to scheduled instances and accessed memory locations of a target statement T, at a given dependence depth d and a violation depth v (defined below), for a given pair of memory references r_S and r_T. The violation relation is not empty for two scheduled instances iff they are involved in a dependence relation and the source instance is scheduled after or at the same time as the target instance. The violation relation is built using three subrelations:

1. **Dependency existence condition** between the instances, see Section 3.5.
2. **Scheduling condition** of the instances, see Section 3.3.

$$\delta_{S,r_S \xrightarrow{d} T,r_T}(\vec{p}) = \left\{ \begin{pmatrix} \vec{\imath}_S \\ \vec{a}_{S,r_S} \end{pmatrix} \rightarrow \begin{pmatrix} \vec{\imath}_T \\ \vec{a}_{T,r_T} \end{pmatrix} \;\middle|\; \left[\begin{array}{c|c|c|c|c|c} D^{\vec{\imath}_S}_S & 0 & 0 & 0 & D^{\vec{p}}_S & D^c_S \\ \hline 0 & 0 & D^{\vec{\imath}_T}_T & 0 & D^{\vec{p}}_T & D^c_T \\ \hline A^{\vec{\imath}_S}_{S,r_S} & A^{\vec{a}_{S,r_S}}_{S,r_S} & 0 & 0 & A^{\vec{p}}_{S,r_S} & A^c_{S,r_S} \\ \hline 0 & 0 & A^{\vec{\imath}_T}_{T,r_T} & A^{\vec{a}_{T,r_T}}_{T,r_T} & A^{\vec{p}}_{T,r_T} & A^c_{T,r_T} \\ \hline 0 & I & 0 & -I & 0 & 0 \\ \hline I^{1..d-1,\bullet} & 0 & -I^{1..d-1,\bullet} & 0 & 0 & 0 \\ \hline I^{d,\bullet} & 0 & -I^{d,\bullet} & 0 & 0 & \vec{0} \text{ or } -\vec{1} \end{array}\right] \begin{pmatrix} \vec{\imath}_S \\ \hline \vec{a}_{S,r_S} \\ \hline \vec{\imath}_T \\ \hline \vec{a}_{T,r_T} \\ \hline \vec{p} \\ \hline 1 \end{pmatrix} \begin{array}{c} \geq \\ \hline \geq \\ \hline \geq \\ = \\ = \\ \hline = \\ \geq \end{array} \vec{0} \right\},$$

Figure 2. Data Dependence Relation – Exact set of instances of a target statement T accessing data through a target reference r_T that depend on instances of a source statement S accessing data through a source reference r_S, at loop depth d. The top part of the constraint matrix corresponds to the existence condition, the middle part to the conflict condition and the bottom one to the causality condition.

$$\upsilon_{S,r_S \xrightarrow{d,v} T,r_T}(\vec{p}) = \left\{ \begin{pmatrix} \vec{\imath}_S \\ \vec{a}_{S,r_S} \\ \vec{\imath}_S \end{pmatrix} \rightarrow \begin{pmatrix} \vec{\imath}_T \\ \vec{a}_{T,r_T} \\ \vec{\imath}_T \end{pmatrix} \;\middle|\; \left[\begin{array}{c|c|c|c|c|c} \multicolumn{6}{c}{\Delta_{S,r_S \xrightarrow{d} T,r_T}} \\ \hline T^{\vec{\imath}_S}_S & 0 & T^{\vec{\imath}_S}_S & 0 & 0 & 0 & T^{\vec{p}}_S & T^c_S \\ 0 & 0 & 0 & T^{\vec{\imath}_T}_T & 0 & T^{\vec{\imath}_T}_T & T^{\vec{p}}_T & T^c_T \\ \hline 0 & 0 & I^{1..v-1,\bullet} & 0 & 0 & -I^{1..v-1,\bullet} & 0 & 0 \\ 0 & 0 & I^{v,\bullet} & 0 & 0 & -I^{v,\bullet} & 0 & \vec{0} \text{ or } -\vec{1} \end{array}\right] \begin{pmatrix} \vec{\imath}_S \\ \hline \vec{a}_{S,r_S} \\ \hline \vec{\imath}_S \\ \hline \vec{\imath}_T \\ \hline \vec{a}_{T,r_T} \\ \hline \vec{\imath}_T \\ \hline \vec{p} \\ \hline 1 \end{pmatrix} \begin{array}{c} = or \geq \\ \text{(see Fig. 2)} \\ \hline \geq \\ \geq \\ \hline = \\ \geq \end{array} \vec{0}, \right\},$$

Figure 3. Dependence Violation Relation – Exact set of instances of a target statement T accessing data through a target reference r_T that have been scheduled before, or at the same time than, instances of a source statement S accessing data through a source reference r_S at scheduling dimension v, although they are in dependence relation at loop depth d. The top part of the constraint matrix corresponds to the dependency existence condition, the middle part to the scheduling condition and the bottom one to the causality violation condition.

3. **Causality violation condition** of the scheduled instances: the scheduled instances of the source statement in the violation relation are executed after or at the same time as the corresponding scheduled instances of the target statement. The constraints involved in this condition depend on the situation. They can be separated into a disjunction with as many components as scheduling dimensions in both S and T (i.e. the minimum scheduling depth between S and T). Each component corresponds to a given scheduling depth and is called a *violation depth*. For a given violation depth $v > 0$, the causality violation condition is made of two parts:
   - the equality of the scheduling elements at depth less than v: $t^x_S = t^x_T$ for $x < v$,
   - $t^v_T < t^v_S$ if v is less than the minimum number of scheduling dimensions between S and T,
     $t^v_T \leq t^v_S$ if v is equal to the minimum number of scheduling dimensions between S and T.

   If the minimum number of scheduling dimensions is zero, there is no causality violation condition and the violation may only exist if a dependency exists.

Figure 3 shows the general form of a violation relation, where $\Delta_{S,r_S \xrightarrow{d} T,r_T}$ is the dependence constraint matrix (see Section 3.5) where additional columns set to zero have been added for the corresponding $\vec{\imath}_S$ and $\vec{\imath}_T$ dimensions.

Note that we consider here *potential* violations as violations. For instance, when two instances in a dependence relation have the same scheduling, the violation exists since it is not possible to know which one will be executed first from the scheduling. However, the code generation step may still generate (by chance) a correct code. To compute only definite violations, simply consider that there is no violation if the minimum number of scheduling dimension is zero and that the second part of the causality violation condition is always $t_{T,v} < t_{S,v}$. Finally, if the scheduling relation is a union, the violation analysis has to be performed on each component of the union.

For a given set of scheduling relations and a data dependence graph, violations relations form a *violated dependence graph* that we use as a basis for strategies to make a given transformation legal.

4.2 Mapping Deviation

When a violated dependence graph has been computed for a given scheduling, it is possible to reason about it to modify either the input program or the transformation to avoid violations. Some transformations like, e.g., privatization, or array expansion [13, 23], do not generate new violations, but can remove some of them at the price of a higher memory footprint. If they are enough to enable a scheduling, applying them *after* violated dependence analysis ensures the minimal expansion is used.

Another way is to modify the scheduling itself. Starting from an incorrect transformation sequence, the goal is to restore the original program semantics for the final program while deviating from the desired transformation as little as possible. To solve this problem, we propose the mapping deviation approach. It aims at computing *safe spaces* where no dependence violation exist, with respect to parametric changes (*mapping deviations*) in any dimension of any relation of the polyhedral representation. When applied only to the scheduling relations, it shares the same goal as Vasilache et al.'s algorithm [36], i.e., starting from an illegal scheduling, find a constant (possibly parametric) shifting with minimal deviation. However the two techniques are quite different. Contrary to Vasilache et al. the algorithm presented here is not iterative on depth (however it can be modified in this way to reduce its complexity, see the depth-by-depth variant in Section 4.3), and not tied to the scheduling only. The most fundamental difference is that, rather than reasoning about the space where a violation exists (to find the extremal amount of time units between a source instance and a target instance), we work on the space where no violation exists. This strategy is the key for the flexibility of our technique which can be used to reason about any component of the polyhedral representation. In addition, it also makes its implementation much easier.

The algorithm to compute safe spaces and solutions to mapping deviations is depicted in Figure 4. We use the general term "map-

MAPPING DEVIATION ALGORITHM

- **Input:** M: original mapping relation set; DDG: data dependence graph; VDG: violated dependence graph
- **Output:** M_{shift}: set of corrected mapping relations

1. choose the set of mapping relations M_{shift} to correct by shifting (at least one mapping from each violated dependence in VDG)
2. **for each** mapping relation $\mathcal{M}$ in M_{shift} **do**
   (a) extend $\mathcal{M}$ with *p deviation parameters*, where p is the maximal violation depth of all the violations where $\mathcal{M}$ is involved
   (b) shift the i^{th} mapping dimension of $\mathcal{M}$ with the i^{th} deviation parameter, $0 \leq i \leq p$ (i.e., in all constraints, substitute the i^{th} mapping dimension t^i with $t^i - Ci$, where Ci is the i^{th} deviation parameter).
3. build the list V of dependence violation relations which have to be recomputed with respect to the modified mappings
4. $\mathcal{D}_{shift} \leftarrow$ universe
5. **for each** dependence violation relation $\mathcal{V}$ in V **do**
   (a) replace the original mapping(s) with the shifted mapping(s) in $\mathcal{V}$ (i.e., introduce the deviation parameters as in step 2)
   (b) compute solutions to $\mathcal{V}$ as a *quast* on deviation and global parameters using, e.g., PIP
   (c) build a union of polyhedra $\overline{\mathcal{V}}$ such that each union component corresponds to the parametric conditions leading to no solution (bottom or $\perp$ in PIP) in the *quast*; in those polyhedra, deviation parameters are promoted to variables while global parameters remain parameters
   (d) $\mathcal{D}_{shift} \leftarrow \mathcal{D}_{shift} \cap \overline{\mathcal{V}}$
6. **if** $\mathcal{D}_{shift}$ is empty **then**
   (a) return $\emptyset$
   else
   (a) choose a low deviation solution to $\mathcal{D}_{shift}$ (computed using, e.g., PIP), i.e., a *correcting value* for each deviation parameter (possibly parametric itself)
   (b) replace the deviation parameters in M_{shift} with their corresponding correcting value and remove the deviation parameters
   (c) return M_{shift}

Figure 4. An Algorithm to Compute the Safe Space of a Transformation Relatively to a Mapping Deviation

ping" (a relation being a mapping) to highlight the fact that we can apply the same methodology to study any component of the polyhedral representation. The intuitive idea is the following:

- First, we introduce new *deviation parameters* and we use them to *shift* the mapping dimensions to be corrected. A deviation parameter is specific to a given dimension of a given mapping relation. We create one deviation parameter per dimension to be corrected in each statement in a violation (even transitively). Then we shift each dimension to correct with its own deviation parameter (i.e., we substitute t with $t - C$ in the relation if t is the dimension to correct and C is its deviation parameter).
- Next, we find the constraints on the deviation parameters such that no violation exists. We rely on Parametric Integer Programming (using the PIP tool) for this task [12]. For this we express the violation relations using the shifted mapping and we ask PIP for a solution. The solution is provided as a *quasi-affine selection tree*, or *quast*, i.e., a selection tree based on constraints on parameters. Each path to $\perp$ (no solution) in the quast corresponds to a part of the parameter space where the violation does not exist. We gather all those parts in a union of polyhedra defining the "safe" part of the space with respect to parameters for a given violation relation. Lastly we intersect all the unions for all the violation relations.
- Finally, we select a solution in the "safe" part of the space (if it exists) such that it has a minimum deviation from the original mapping. This selection highly depends on the structure of the mapping (are there dimensions more/less important than others, e.g., *beta* dimensions [16]?). Our general strategy is to minimize the absolute value of deviations with decreasing priority, from the first to the last mapping dimensions. It is again an optimization problem where we use PIP. It is easy to adapt the strategy to any mapping structure that is known.

Relation parametric shifting is a very expressive transformation. From a syntactic point of view, the effect of a successful scheduling correction on the generated code may correspond, in the case of a large deviation, up to a long sequence of loop shifting, fusion, distribution, peeling, index set splitting and code motion. Hence, this strategy allows a large range of possible corrections while targeting the scheduling part. Pouchet et al. showed that this class of transformations has, in general, the minimal impact on performance [29]. As a result, it is expected to generate only a limited offset to the desired optimization.

THEOREM 1. *(completeness) The Mapping Deviation Algorithm finds deviations to correct a set of mapping relations if and only if such deviations exist.*

Proof. The proof of completeness lies in the manipulation of exact violation spaces by the algorithm. On one hand, it choses a solution in the space that is complementary to the exact instance-wise violation space, hence if a solution exists then the deviated relations are legal. On the other hand, if correct solutions exist then they belong to that space, otherwise they would violate at least one dependency. ∎

A complete example of scheduling correction using this algorithm is presented in Figure 5. We use a simplified version of the running example introduced in Section 2: a user wishes to transform the original program in 5(a) using the scheduling relation in 5(b), most probably to improve locality for the array A. Using this

scheduling, a code generator could generate the code in 5(c), it corresponds to a loop fusion. Unfortunately it is easy to see that in the target code, $S2$ is now consuming data before $S1$ produces them: a violation analysis would show that $\Upsilon_{S1,1\xrightarrow{0,2}S2,2}$ is not empty. We can try to correct this scheduling.

Because there are only two statements, it is not necessary to shift both. We choose to apply a deviation to the scheduling of $S1$. This translates to a parametric shifting that impacts the dependence violation relation $\Upsilon_{S1,1\xrightarrow{0,2}S2,2}$, but also $\Upsilon_{S1,1\xrightarrow{0,1}S2,2}$. Hence it is necessary to consider both, as shown in Figure 5(e) and 5(f) where we applied the deviation to both relations. Next, for each relation, we compute the "safe" space where, depending on the deviation parameters, no violation exists. This is done using PIP [12] on the violation relation constraints. The quasts and their conversion to "safe" spaces are shown in 5(g) and 5(h). The intersection of the violation-relative "safe" spaces gives the global "safe" space shown in 5(i).

Each part of the "safe" space union contains possible solutions. Moreover, constraints on global parameters forces versioning: different corrections may be applied depending on the global parameter values. In our example, we can see from 5(i), that any correction is correct for $N < 1$ (because the dependency does not exist), but not for $N >= 1$. We are free to chose any solution for $N >= 1$. However they are not equivalent with respect to deviation from the original scheduling. For instance two possible corrections are shown in 5(j) and 5(k). One of them totally cancels the loop fusion, while the other slightly modifies it. Our general approach is to apply the smallest possible shifting, lexicographically. Hence we achieve the correction shown in 5(k). If the correction algorithm fails, then the scheduling is considered as illegal and is simply discarded.

4.3 Variations of the Mapping Deviation Algorithm

Mapping deviation has been implemented in Candl[2], a tool dedicated to data dependence analysis. The base algorithm proposes to restrict the number of shifted dimensions for a given mapping to the maximum violation depth where it is involved (see Step 2 of Figure 4). However it is easy to derive useful variants from the base algorithm to fit different polyhedral framework implementations.

Full-Depth Variant A depth-by-depth approach is possible by adding an outer loop on mapping depth to the algorithm. At the n^{th} iteration, only the correcting parameter for the n^{th} dimension is added to all mappings involved in a dependence relation, and the correcting values of the $n-1$ first dimensions are integrated for violation relation construction.

Depth-by-Depth Variant Shifting at depth d can generate violations at depth $d' > d$ (e.g., when shifting results in a loop fusion). While we can try to solve it at depth $< d$, we can allow to correct it at a further depth. To enable this, we simply need to add correcting parameters to all dimensions.

5. Applications

5.1 Correcting Transformation Scripts

The scheduling relation abstraction is too complex to be used directly by end-users that are not very familiar with the polyhedral model. However, optimization experts can strongly benefit from its ability to compute exact data dependence analysis to check or (using our work) to correct their transformations, and from automatic code generation to handle the complex and error-prone optimization implementation in a transparent way. Several frameworks have been proposed to provide a high-level interface on top of a polyhedral engine, UTF [22] being arguably the very first of them. The URUK [16], the CHiLL [8] and the Clay [2] frameworks allow the composition of any complex sequence of classical loop transformations (without necessary intermediate legality tests for, e.g., URUK or Clay). All these frameworks translate internally a transformation script composed with high-level syntactic transformation primitives (like *skew*, or *interchange* or *tile*) to a set of scheduling relations, or to a simpler form, e.g., scheduling functions.

Using mapping deviation on the scheduling relations as shown in Figures 4 and 5, it is possible to correct them in the same way as general scheduling. However either UTF, URUK, CHiLL or Clay is relying on a specific scheduling structure with special dimensions which must be preserved during the process of combining transformations, and consequently, during correction.

In all these frameworks, odd scheduling dimensions (the first dimension being 1) express the lexicographic ordering of loops and statements. They are called β in URUK or Clay and *auxiliary loops* in CHiLL or UTF. We use here the name β for short. If we call α the even dimensions, the general form of the scheduling vector is $(\beta_1, \alpha_1, \beta_2, \alpha_2 \ldots, \beta_n, \alpha_n, \beta_{n+1})^T$. β-like dimensions are restricted to be non-parametric constant values. The vector formed from β values for a given scheduling union component is called a β-*vector*. Each β-vector must be unique and cannot be a prefix of other β-vectors. These invariants must hold after a correction process.

LEMMA 1. *The correction algorithm based on mapping deviation preserves the scheduling structure based on* β*, except for the last* β *dimension which can be updated to respect that structure.*

Proof. The correction algorithm may change the scheduling only by adding a parametric constant to each dimension. Such shifting cannot generate new iterations. Moreover, in the case of β-like dimensions, the deviation constant cannot be parametric because those dimensions are not linked directly or indirectly to parameters. Hence, PIP cannot issue parametric conditions involving β-like dimensions, and they remain constant after correction.

β-vector uniqueness is not guaranteed to be preserved by the correction algorithm. However, two iterations cannot share the same scheduling because it is considered as a violation in the violated dependence analysis. Thus, the last β-like dimension of statements with the same β-vector (or one β-vector and some β-prefix) is not meaningful for ordering (it is a free dimension). It follows they can be updated in such a way that each β vector is unique and is not the prefix of other β-vectors, provided other β values are updated to respect the relative orders, if necessary. ∎

Hence, by analyzing parametric constants which have been added to some scheduling dimensions, corrections can translate to a sequence of shifting (for α dimensions) and reordering (for β dimensions) as defined in UTF, URUK, CHiLL or Clay. Note that the minimum deviation from the original transformation may undo it entirely if no better solution is found.

To illustrate how mapping deviation can concretely assist users in designing optimizations, we present how it is used in Clint, a loop-transformation environment based on direct manipulation of iteration domain visualization [44]. Clint has three synchronized interactive panels shown in Figure 6: (1) is the visualization where iteration domains can be direclty moved, cut, skewed etc. (2) is the transformation script depicting user's actions in a textual way and (3) is the original or the target source code with colors mapped similarly to the visualization. Here, the code panel contains the original code of our running example described in Section 2.

In Clint, the expert's intuition about a profitable loop fusion simply translates into selecting (click) and moving (drag) an iteration domain visualization onto the other one, which corresponds to action (a): the original loop is shaded out while the intended position is light dashed green overlay. This action generates the two first lines of the transformation script in panel (2) where each primitive

[2] http://periscop.github.io

```
     for (i = 0; i <= N; i++)
S1:    A[i] = B[i];

     for (i = 0; i <= N; i++)
S2:    C[i] = A[i+1];
```

(a) Original Program

$$\theta_{S1} = \left\{ (\ i \ ) \rightarrow \begin{pmatrix} 0 \\ i \\ 0 \end{pmatrix} \right\}$$

$$\theta_{S2} = \left\{ (\ i \ ) \rightarrow \begin{pmatrix} 0 \\ i \\ 1 \end{pmatrix} \right\}$$

(b) Illegal scheduling

```
     for (i = 0; i <= N; i++) {
S1:    A[i] = B[i];
S2:    C[i] = A[i+1];
     }
```

(c) Illegal Target Code

$$\theta_{S1} = \left\{ (\ i \ ) \rightarrow \begin{pmatrix} 0+C1 \\ i+C2 \\ 0 \end{pmatrix} \right\}$$

(d) Deviation of θ_{S1}

$\Upsilon_{S1,1 \overset{0,2}{\rightarrow} S2,2}$:

$$\left[\begin{array}{cccc|cccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \end{array}\right] \begin{pmatrix} \underline{i} \\ a_{\underline{S1,1}} \\ t_{S1,1} \\ t_{S1,2} \\ t_{S1,3} \\ \hline i' \\ a_{\underline{S2,2}} \\ t_{S2,1} \\ t_{S2,2} \\ t_{S2,3} \\ \hline N \\ C1 \\ C2 \\ \hline 1 \end{pmatrix} \begin{matrix} \geq \\ \geq \\ = \\ = \\ = \vec{0} \\ = \\ = \\ = \\ = \\ \geq \end{matrix}$$

(e) Violated Dependencies With Deviated θ_{S1}

$\Upsilon_{S1,1 \overset{0,1}{\rightarrow} S2,2}$:

$$\left[\begin{array}{cccc|cccc|cccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ \hline 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \end{array}\right] \begin{pmatrix} \underline{i} \\ a_{\underline{S1,1}} \\ t_{S1,1} \\ t_{S1,2} \\ t_{S1,3} \\ \hline i' \\ a_{\underline{S2,2}} \\ t_{S2,1} \\ t_{S2,2} \\ t_{S2,3} \\ \hline N \\ C1 \\ C2 \\ \hline 1 \end{pmatrix} \begin{matrix} \geq \\ \geq \\ = \\ = \\ = \vec{0} \\ = \\ = \\ = \\ = \\ \geq \end{matrix}$$

(f) Potential Violations With Deviated θ_{S1}

```
if (N >= 1)
  if (C1 >= 0)
    if (-C1 >= 0)
      if (C2 >= 0)
        solution
      else
        ⊥
    else
      ⊥
  else
    ⊥
else
  ⊥
```

$$\left\{ \begin{array}{l} N \geq 1 \\ C1 = 0 \\ C2 < 0 \end{array} \right. \cup$$

$$\left\{ \begin{array}{l} N \geq 1 \\ C1 > 0 \end{array} \right. \cup$$

$$\left\{ \begin{array}{l} N \geq 1 \\ C1 < 0 \end{array} \right. \cup$$

$$\{ \ N < 1$$

(g) Quast and safe space for $\Upsilon_{S1,1 \overset{0,2}{\rightarrow} S2,2}$

```
if (N >= 1)
  if (C1 >= 1)
    solution
  else
    ⊥
else
  ⊥
```

$$\left\{ \begin{array}{l} N \geq 1 \\ C1 < 1 \end{array} \right. \cup$$

$$\{ \ N < 1$$

(h) Quast and safe space for $\Upsilon_{S1,1 \overset{0,1}{\rightarrow} S2,2}$

$$\{ \ N < 1 \quad \cup \left\{ \begin{array}{l} N \geq 1 \\ C1 = 0 \\ C2 < 0 \end{array} \right. \quad \cup \left\{ \begin{array}{l} N \geq 1 \\ C1 < 0 \end{array} \right.$$

(i) Result of the intersection of safe spaces

$$\theta_{S1} = \left\{ (\ i \ ) \rightarrow \begin{pmatrix} 0-1 \\ i+0 \\ 0 \end{pmatrix} \right\}$$

```
     for (i = 0; i <= N; i++)
S1:    A[i] = B[i];

     for (i = 0; i <= N; i++)
S2:    C[i] = A[i+1];
```

(j) High Deviation Depth 1 Correction

$$\theta_{S1} = \left\{ (\ i \ ) \rightarrow \begin{pmatrix} 0+0 \\ i-1 \\ 0 \end{pmatrix} \right\}$$

```
S1: A[0] = B[0];
     for (i = 0; i < N; i++) {
S1:    A[i+1] = B[i+1];
S2:    C[i] = A[i+1];
     }
S2: C[N] = A[N+1];
```

(k) Low Deviation Depth 2 Correction

Figure 5. Example of Scheduling Correction

targets the loop or the statement indexed by its β-vector. Unfortunately, the transformation is not legal: thin red arrows depict dependencies violated by the intended transformation. Mapping deviation automatically computes in real time the minimal shifting transformation to make the transformation legal and generates the two last lines of the transformation script in panel (2). The correction translates visually to action (b): the iteration domain is automatically moved to the closest place where the transformation is legal. The corrected position is light solid green polygon. The final situation preserves the expert's intuition and optimization benefits as much as possible while making the transformation legal.

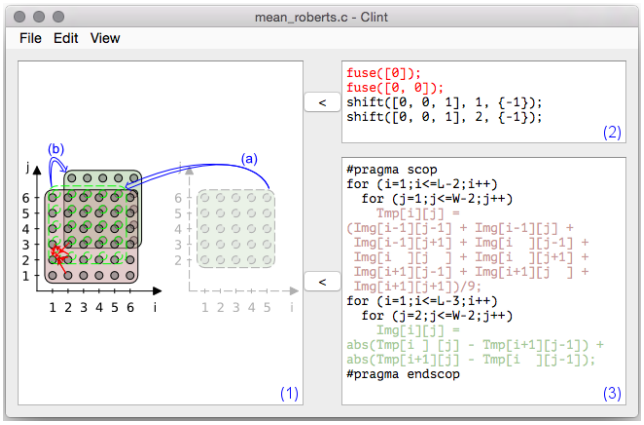

Figure 6. Automatic Correction by Mapping Deviation in Clint

In the context of semi-automatic tools for code manipulation, mapping deviation enables an elaborated semantic feedback in addition to the go/no-go answer provided by dependence analysis and to the syntactic help provided by automatic code generation. It is a significant step forward because trying to rework an automatically generated incorrect version of a code (e.g., after an inadequate tiling) is often even more complex than trying to optimize the original code directly. Instead, users can provide a general intuition about the best optimization strategy and the algorithm can check and adjust it to keep the intuition while ensuring legality.

5.2 Creating Runtime Array Overlapping Tests

Compiler alias analysis tries to determine whether two pointers access the same memory location or not. If the analysis is unable to decide, compilers use a conservative strategy and suppose they may alias. The consequence is to consider over-approximated data dependencies and to discard a large range of optimization opportunities. Doerfert et al. show it is one of the main reasons, if not the main reason, why LLVM's polyhedral framework is rejecting candidate code regions in SPEC 2000 and the PolyBench suite (their study states, e.g., that possible aliasing is a cause of rejection for 1093 regions over 1862 candidate regions in SPEC 2000) [10]. They propose a runtime test based on the computation of access ranges to ensure no conflict is possible at runtime with significant speedups on the PolyBench suite. However their technique does not depend of the transformation itself, while it may tolerate harmless aliasing and overlapping.

Using mapping deviation on the access relation, it is possible to derive a runtime test to check the exact necessary and sufficient conditions on array overlapping for a given transformation to be legal. The limit of our solution, shared with Doerfert et al.'s work, is that multidimensional arrays must use a consecutive space in memory, with no overlapping between different dimensions. To simplify, we also consider that all array elements are of the same type.

The technique is the following. First we consider a transformation, e.g., the fusion + shifting of the code in Figure 1, with the hypothesis that if two arrays have different names, then they do not overlap. The transformation should be legal within this hypothesis, i.e., each violation relation should define an empty space. We adapt the safe-space computation algorithm of Figure 4 in this way: at step 1, we select all access relations except those of one arbitrarily chosen array (choosing the one with the highest number of accesses is better for performance). Let us call the chosen array `A`. At step 2.(a) the deviation for all selected access relations is to add a deviation parameter specific to each array to their first access dimension only, and to consider the access is now to the array `A` (i.e., the deviation would change an access to `B[i][j]` into an access to `A[i+`C_B`][j]` where C_B is the deviation parameter for the array `B`). Then at step 6, there is no need to choose a solution to $\mathcal{D}_{shift}$: its constraints form the test we are looking for (union parts are linked with logical OR while constraints in sets are composed with logical AND).

Intuitively, we are considering that all references are to possibly different parts of the same array (seen as the global storage space). Each deviation parameter represents the relative position of its corresponding array inside the reference array (or equivalently, its position in the global storage space). Original violation relations were known to be empty spaces because the transformation was legal, irrespectively from array overlapping. Hence the algorithm computes the conditions, with respect to the relative position of each array for the violation relations to remain empty spaces. These conditions can be trivially translated to runtime tests: each deviation parameter is translated to the relative position of its corresponding array with respect to the reference array (C_B becomes `A-B`) and other values are multiplied by the size of one dimension of the reference array.

Back to the running example, let us suppose we would like to apply the corrected transformation shown in Figure 6 while being unsure about array overlapping. We chose to apply deviation on the array `Tmp` (e.g, `Tmp[i][j]` is considered as `Img[i+C][j]`) and the deviation analysis is performed for each violation relation impacted by this change. The safe space we get from the algorithm is $C \geq L \cup C \leq -L$. Let us suppose the type of `Img` is `int(*)[W]`, then the runtime test to ensure the correctness of the transformation with respect to array overlapping is shown in Figure 7. It is used to select either the optimized or the original version at runtime[3]. While the running example is very simple for clarity and space reasons (yet, note that generating the test doesn't require the array sizes which may not be known at compile time), let us recall that the mapping deviation algorithm is exact and complete: it can create necessary and sufficient runtime tests for any SCoP and any scheduling relation.

```
if ((Tmp >= Img + L * W) || (Img >= Tmp + L * W)) {
  // Here using the transformed kernel is safe
} else {
  // Otherwise we can use the original kernel there
}
```

Figure 7. Generated Runtime Test for Versioning Figure 1

Evaluating the benefits of mapping deviation has to be done on a case by case basis with respect to the optimization strategy because it works *with respect to a given transformation*: a study on one

[3] In general, a special care should be taken when generating high-level runtime tests such as in Figure 7 since, e.g., pointer arithmetic between pointers to different segments is undefined in C. Hence additional tests may be necessary to ensure the semantics of the pointer arithmetic. Since it is a technical, language dependent issue, we do not address it in this paper.

strategy says little about how relevant it may be for another one. We present here the general theory of mapping deviation. Evaluating it on various strategies is left for a separate study.

In the context of static optimization, when the exact memory mapping of data is not known, mapping deviation provides a new static/dynamic solution for SCoPs. It may enable complex optimization and parallelization at the price of a test to be evaluated at runtime outside computational loops, hence with an arguably negligible overhead with respect to the kernel execution time. While its benefits depend on the optimization strategy, it is both more precise and less restrictive than existing ones because it ensures that the necessary requirements are met specifically for the desired transformation and hence tolerating safe aliasing or overlapping. Moreover, its algorithm uses classical polyhedral operations which are already integrated in production compilers or available through robust libraries (through, e.g., `isl` [39] in GCC 5 or LLVM 3.6) which makes it easy to implement.

6. Related Work

Enabling transformations is a specialized form of reasoning on the correctness for a given program transformation. They have been used widely to enable specific syntactic transformations, e.g., Wolf and Lam rely on loop skewing and reversal to enable loop tiling for optimizing data locality [41], Allen and Kennedy use loop splitting to enable loop interchange to expose parallel loops [1]. McKinley et al. exploit loop splitting similarly to maximize data locality [26].

Several techniques have been proposed to *complete* affine schedules to produce a legal transformation. Some of them address specifically the completion of the constant part: in the context of the UTF framework, Kelly and Pugh apply *schedule alignment* where the alignment technique completely set the constant part of the schedules [22]. Also addressing the constant part of the schedule, Darte et al. use *retiming* to enable their parallelizing transformation [9]. More generally, Li and Pingali [24] and Griebl et al. [17] complete the last dimensions of a partially defined legal transformation which does not violate dependencies such that it stays legal.

The first work on *correcting* a general transformation to respect legality has been proposed by Bastoul and Feautrier [5]. Their technique can complete partially defined affine schedules, not only the constant part, and apply complex corrections in the specific context of data locality optimizing schedules. However, the complexity of the technique makes it impractical even for moderately large multidimensional problems. In comparison, our technique is scalable since it reasons on each violation relation separately, which is known to scale well [35] (basically, it has the same complexity as an exact data dependence test, already in use in production compilers). Vasilache et al. also proposed a scalable correction technique for fully specified illegal transformations [36]. While we focus on the same type of adjustment, Vasilache et al. s technique is dedicated to the correction of scheduling functions with a specific structure. Mapping deviation is much more general because it can reason on any part of the polyhedral representation. Moreover, it addresses general, relation-based abstractions without specific structure.

Morvan et al. propose, instead of correcting the transformation itself, to insert *wait states* in the target code to correct dependencies violated by their loop coalescing transformation [27]. While it has a different goal than our technique, it offers an interesting complementary way to consider for more general corrections.

Several tools have been released to bring semantic feedback to users for debugging or loop-level optimization purpose such as Pareon from VectorFabrics [37]. Our work is different because it corrects automatically the user's (or compiler's) intuition instead of providing intuition about how to correct or to optimize.

7. Conclusion

In this paper, we presented a transformation-centric approach to optimization. It has the potential to avoid discarding too early a desirable, efficient solution, and to avoid applying counter-productive pre-processing to remove harmless data dependencies. We introduced mapping deviation, a new methodology to reason about the legality of a transformation, while considering parametric deviations of any dimension of any element of the polyhedral representation of a program and of its transformation. We derived from mapping deviation a transformation correction technique that achieves a new milestone in flexibility, capable of addressing general scheduling relations without any particular structure. We also showed that reasoning on access relations, mapping deviation can be used to complement alias analysis, bringing the first method to build runtime tests to check the legality of a desired transformation with respect to array overlapping.

Ongoing work aims at studying other applications of mapping deviation, e.g., to minimize the memory footprint of a running application by maximizing safe array overlapping.

Acknowledgments

The author would like to thank Oleksandr Zinenko for designing Figure 6 using the Clint tool [44]. Many thanks also to CC's anonymous reviewers for their help in improving the quality of this paper.

References

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

[2] L. Bagnères, O. Zinenko, S. Huot, and C. Bastoul. Opening polyhedral compiler's black box. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Barcelona, 2016.

[3] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, Sept. 2004.

[4] C. Bastoul. *Contributions to High-Level Program Optimization*. Habilitation Thesis. Paris-Sud University, France, Dec. 2012.

[5] C. Bastoul and P. Feautrier. Adjusting a program transformation for legality. *Parallel processing letters*, 15(1):3–17, Mar. 2005.

[6] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul. The polyhedral model is more widely applicable than you think. In *Proceedings of the Intl. Conf. on Compiler Construction (ETAPS CC'10)*, LNCS, pages 283–303, Paphos, Cyprus, Mar. 2010.

[7] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proc. of the 2008 ACM Conf. on Programming language design and implementation (PLDI'08)*, Tucson, AZ, USA, June 2008.

[8] C. Chen, J. Chame, and M. Hall. A framework for composing high-level loop transformations. Technical Report 08-897, USC Computer Science, June 2008.

[9] A. Darte, G.-A. Silber, and F. Vivien. Combining Retiming and Scheduling Techniques for Loop Parallelization and Loop Tiling. *Parallel Processing Letters*, 7(4):379–392, 1997.

[10] J. Doerfert, C. Hammacher, K. Streit, and S. Hack. SPolly: Speculative Optimizations in the Polyhedral Model. In *Proceedings of the 3rd International Workshop on Polyhedral Compilation Techniques*, pages 55–60, Berlin, Germany, Jan. 2013.

[11] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. Garzaran, D. Padua, and K. Pingali. A language for the compact representation of multiple program versions. In *Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC)*, volume 4339 of *Lect. Notes in Computer Science*, pages 136–151, Hawthorne, New York, Oct. 2005.

[12] P. Feautrier. Parametric integer programming. *RAIRO Recherche Opérationnelle*, 22(3):243–268, 1988. http://www.piplib.org.

[13] P. Feautrier. Array expansion. In *ICS*, pages 429–441, St Malo, France, July 1988.

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

[15] P. Feautrier and C. Lengauer. Polyhedron model. In *Encyclopedia of Parallel Computing*, pages 1581–1592. Springer US, 2011.

[16] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, June 2006.

[17] M. Griebl, C. Lengauer, and S. Wetzel. Code generation in the polytope model. In *Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'98)*, pages 106–111, 1998.

[18] M. Griebl, P. Feautrier, and C. Lengauer. Index set splitting. *Int. Journal of Parallel Programming*, 28(6):607–631, 2000.

[19] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Grösslinger, and L.-N. Pouchet. Polly-polyhedral optimization in llvm. In *IMPACT 2011 First International Workshop on Polyhedral Compilation Techniques*, Chamonix, France, 2011.

[20] F. Irigoin. Dependence abstractions. In *Encyclopedia of Parallel Computing*, pages 552–556. 2011.

[21] 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.

[22] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical Report UMIACS-TR-92-126.1, University of Maryland Institute for Advanced Computer Studies, 1993.

[23] V. Lefebvre and P. Feautrier. Automatic storage management for parallel programs. *Parallel Computing*, 24(3–4):649–671, 1998.

[24] 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.

[25] A. Lim. *Improving Parallelism and Data Locality with Affine Partitioning*. PhD thesis, Stanford University, 2001.

[26] K. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Trans. on Programming Languages and Systems*, 18(4):424–453, July 1996.

[27] A. Morvan, S. Derrien, and P. Quinton. Efficient Nested Loop Pipelining in High Level Synthesis using Polyhedral Bubble Insertion. In *IEEE International Conference on Field-Programmable Technology (FPT'11)*, New Delhi, India, Dec. 2011.

[28] R. Müller-Pfefferkorn, W. Nagel, and B. Trenkler. Optimizing cache access: A tool for source-to-source transformations and real-life compiler tests. In *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference*, pages 72–81, Pisa, august 2004.

[29] L.-N. Pouchet, C. Bastoul, A. Cohen, and J. Cavazos. Iterative optimization in the polyhedral model: Part II, multidimensional time. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 90–100, Tucson, Arizona, June 2008. ACM Press.

[30] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, and P. Sadayappan. Combined iterative and model-driven optimization in an automatic parallelization framework. In *Conference on Supercomputing (SC'10)*, New Orleans, LA, Nov. 2010.

[31] L.-N. Pouchet, U. Bondhugula, C. Bastoul, A. Cohen, J. Ramanujam, P. Sadayappan, and N. Vasilache. Loop transformations: Convexity, pruning and optimization. In *38th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (PoPL'11)*, pages 549–562, Austin, TX, Jan. 2011.

[32] F. Quilleré, S. Rajopadhye, and D. Wilde. Generation of efficient nested loops from polyhedra. *International Journal of Parallel Programming*, 28(5):469–498, Oct. 2000.

[33] K. Stock, M. Kong, T. Grosser, L.-N. Pouchet, F. Rastello, J. Ramanujam, and P. Sadayappan. A framework for enhancing data reuse via associative reordering. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14*, page 10, Edinburgh, United Kingdom, June 2014.

[34] K. Trifunovic, A. Cohen, D. Edelsohn, F. Li, T. Grosser, H. Jagasia, R. Ladelsky, S. Pop, J. Sjödin, and R. Upadrasta. Graphite two years after: First lessons learned from real-world polyhedral compilation. In *GCC Research Opportunities Workshop (GROW'10)*, Pisa, Italy, 2010.

[35] N. Vasilache, C. Bastoul, S. Girbal, and A. Cohen. Violated dependence analysis. In *ACM International Conference on Supercomputing (ICS'06)*, pages 335–344, Cairns, Australia, June 2006.

[36] N. Vasilache, A. Cohen, and L.-N. Pouchet. Automatic correction of loop transformations. In *PACT*, pages 292–304, Brasov, Romania, Sept. 2007.

[37] VectorFabrics. Pareon. http://www.vectorfabrics.com.

[38] A. Venkat, M. Shantharam, M. Hall, and M. Mills Strout. Non-affine extensions to polyhedral code generation. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, page 185:194, Orlando, FL, USA, February 2014.

[39] S. Verdoolaege. *isl*: An integer set library for the polyhedral model. In *Mathematical Software - ICMS 2010, Third International Congress on Mathematical Software*, pages 299–302, Kobe, Japan, Sept. 2010.

[40] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor. Polyhedral Parallel Code Generation for CUDA. *ACM Transactions on Architecture and Code Optimization*, 9(4), 2013.

[41] M. Wolf and M. Lam. A data locality optimizing algorithm. In *ACM SIGPLAN'91 Conf. on Programming Language Design and Implementation*, pages 30–44, New York, June 1991.

[42] M. J. Wolfe. *Optimizing supercompilers for supercomputers*. PhD thesis, University of Illinois at Urbana-Champaign, Oct. 1982.

[43] Y.-Q. Yang, C. Ancourt, and F. Irigoin. Minimal data dependence abstractions for loop transformations. *International Journal of Parallel Programming*, 23(4):359–388, 1995.

[44] O. Zinenko, S. Huot, and C. Bastoul. Clint: A direct manipulation tool for parallelizing compute-intensive program parts. In *Visual Languages and Human-Centric Computing (VL/HCC), 2014 IEEE Symposium on*, pages 109–112. IEEE, 2014.